Master Thesis
Software Engineering
Thesis no: MSE-2006:14
June 2006

The role of quality requirements in software architecture design

Karol Kazimierz Wnukiewicz

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author:
Karol Kazimierz Wnukiewicz
E-mail: kkwn05@student.bth.se
Web: http://www.student.bth.se/~kkwn05

External advisor:
Zbigniew Huzar
Wroclaw University of Technology
Wybrzeze Wyspianskiego 27
50-370 Wroclaw, Poland
E-mail: Zbigniew.Huzar@pwr.wroc.pl

University advisor:
Mikael Svahnberg
Department of Systems and Software Engineering
E-mail: Mikael.Svahnberg@bth.se

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax : + 46 457 271 25

He nodded, he shrugged. He shrugged again.
"A what?" he said.
"An S.E.P"
"An S… ?"
"…E.P."
"And what's that?"

Douglas Adams, *Life, the Universe, and Everything*

Abstract

An important issue during architectural design is that besides functional requirements, software architecture is influenced greatly by quality requirements [9][2][7], which often are neglected. The earlier quality requirements are considered, the less effort is needed later in the software lifecycle to ensure a sufficient software quality levels. Errors due to lack of their fulfilment are the most expensive and difficult to correct. Therefore, attention to quality requirements is crucial during an architectural design. The problem is not only to gather the system's quality requirements, but to establish a methodology that helps to deal with them during the software development. Literature has paid some attention to software architecture in the context of quality requirements, but there is still lack of effective solutions in this area.

To alleviate the problem, this paper lays out important concepts and notions of quality requirements in a way they can be used to drive design decisions and evaluate the architecture to estimate whether these requirements are fulfilled. Important concepts of software architecture area are presented to indicate how important quality requirements are during the design and what are the consequences of their lack in a software system. Moreover, a quality requirement-oriented design method is proposed as an outcome of the literature survey. This method is a model taking quality requirements into account at first, before the core functionality is placed.

Besides the conceptual solution to the identified problems, this paper also suggests a practical method of handling quality requirements during a design. A recommendation framework for choosing the most suitable architectural pattern from a set of quality attributes is also proposed. Since the literature provides insufficient qualitative information about quality requirement issues in terms of software architectures, an empirical research is conducted as means for gathering the required data. Therefore, a systematic approach to support and analyze architectural designs in terms of quality requirements is prepared. Finally, quality requirement-oriented and pattern-based design method is further proposed as a result of investigating patterns as a tool for addressing quality requirements at different abstraction levels of a design. The research is concerned with the analysis of software architectures against one or more desired software qualities that ought to be achieved at the architectural level.

Keywords: *Non-functional requirements, patterns, quality attributes, quality models, architectural design, architecture evaluation.*

Streszczenie

Proces projektowania architektury systemu informatycznego jest determinowany nie tylko przez wymagania funkcjonalne, lecz również przez wymagania niefunkcjonalne [9][2][7] formułowane podczas analizy wymagań. Specyfikacja wymagań często pomija w opisie te wymagania, kładąc całkowity nacisk na funkcjonalność.
Projektanci winni jednak dążyć do uzyskania systemu informatycznego, którego struktura odzwierciedlałaby oba typy wymagań w danej dziedzinie problemu. Im wcześniej brane są pod uwagę wymagania niefunkcjonalne, tym wyższy poziom końcowej jakości oprogramowania zostanie uzyskany. Wysiłki związane z uzyskaniem nieosiągniętej jakości systemu są bardzo kosztowne, a efekty trudne do osiągnięcia. Stąd uwaga nad wymaganiami niefunkcjonalnymi jest konieczna podczas projektowania architektury. Problem nie polega tylko na właściwej specyfikacji tych wymagań, ale również na ustanowieniu metodyki, która pozwoli na ich realizację. Literatura poświęca trochę uwagi projektowaniu systemów w kontekście wymagań niefunkcjonalnych, jednakże wciąż brak jest efektywnych rozwiązań w tej dziedzinie.

Praca przedstawia rolę i charakter wymagań niefunkcjonalnych w kontekście czynników mających wpływ na decyzje procesu projektowania architektury i jej późniejszą ocenę wyznaczającą poziom spełnienia tych wymagań. Ponadto, zagadnienia związane z architekturą systemu informatycznego zostaną przedstawione by określić istotę wymagań niefunkcjonalnych oraz konsekwencje, jakie wiążą się z ich brakiem. Praca proponuje model projektowy (ang. *quality requirement-oriented design method*) jako rezultat przeglądu sztuki, w którym wymagania niefunkcjonalne brane są pod uwagę w pierwszej kolejności, tj. przed wymaganiami funkcjonalnymi.

Następnie, aby zilustrować to podejście, praca przedstawia praktyczną realizację omawianego problemu. Ze względu na brak informacji w literaturze, które mogłyby posłużyć w tym nowatorskim podejściu, dane zostały skompletowane na podstawie empirycznych badań. Dzięki tym pomiarom powstał tzw. *Recommendation Framework*, czyli narzędzie wspomagające proces projektowania architektury, które na podstawie pożądanych charakterystyk jakości w oparciu o zbiór wymagań niefunkcjonalnych dokonuje wyboru architektury sytemu zdefiniowanej ze zbioru wzorców projektowych. W kolejnej części praca opisuje model procesu projektowania, który podobnie jak poprzedni jest zorientowany na wymagania niefunkcjonalnie. Jednakże ta propozycja (ang. *quality requirement-oriented and pattern-based design method*) wykorzystuje wzorce projektowane zróżnicowane pod względem poziomu abstrakcji systemu, na jakim mogą zostać wykorzystane. Badania prowadzone w tej pracy mają na celu analizę procesu projektowania systemów informatycznych ukierunkowanego na spełnienie jednego lub kilku charakterystyk jakości.

Słowa kluczowe: Wymagania niefunkcjonalne, architektura sytemu informatycznego, projektowanie, wzorce projektowe, modele jakościowe, charakterystyki jakości, jakość oprogramowania.

Acknowledgements

First and foremost, I would like express my sincere gratitude and appreciation to my advisors Zbigniew Huzar and Mikael Svahnberg for their knowledge and patience. Without their assistance this research could not have been completed.

I am also grateful to my parents, Kazimierz and Ewa for their support and love, my sisters Alicja and Malgorzata, my god-son Jakub and his father – my brother-in-law Jacek.

In addition, special thanks are addressed to the interviewed people who spent their precious time on answering the questionnaire, for their interest and effort during the evaluation.

My thanks to them all.

Karol Kazimierz Wnukiewicz

Table of contents

Figures and tables

Figures:

Tables:

Chapter One – Introduction

1.1 Background of the study

The importance of architectural design is widely recognized in software engineering. It is commonly known that an architecture is designed to ensure system functionality, i.e. meet the system functional requirements. A requirement specification is an outcome of requirements engineering activities and besides the mentioned functional requirements, it contains requirements that are not concerned with the functionality. Different from functional requirements that describe *'what'* the system will do, quality requirements (also called *non-functional requirements* or *system properties*) describe *'how'* it will do it. They are in many cases either unclearly stated or even neglected during the requirements specification. This leaves quality attributes impossible to identify, measure, and in consequence – address in software architecture. Hence, to predict explicitly quality attributes of a system, quality requirements need to be specified in sufficient detail.

Software architecture design is often based on architects intuition and previous experience. Little methodological support is available, but there are still no effective solutions to guide the architectural design. Perhaps the most difficult activity is the transformation from requirement specification into software architecture. One key task that remains especially non-trivial is how to handle quality requirements.

The challenge of an architectural design is to develop a software architecture with the desired quality levels. Quality requirements set the boundary for the final quality of a designed system. The problem is to get an early indication of the quality attributes in the resulting architecture. Software architecture is concerned with structures of high-level components and relationships among them. Certain combinations of components are recognized to address some quality attributes. In consequence, quality requirements can be addressed by the architectural design and furthermore – influence the software quality.

1.2 Aims and objectives

The main aim is to investigate the concept of quality requirements in software architecture design. **Figure 1** is a general illustration of the problem. This thesis aims to discuss the quality requirements' impact on software architecture and the design activity. It also involves how to ensure and verify the fulfilment of these requirements. The overall research aim of this thesis is to identify, analyze, and propose a method for addressing quality requirements during software architecture design.

Objectives:

- Specify software architecture in the context of quality requirements.
- Identify and classify quality requirements which influence the selection of software architecture.
- Discuss the specification of quality attributes and their relationship with quality requirements.

- Study software architecture design as a method of achieving quality requirements.
- Analyze the relationship between quality requirements and types of software architecture structures.
- Provide recommendations for software architecture design in terms of quality requirements achievement.
- Investigate existing solutions.
- Verify proposed solutions.

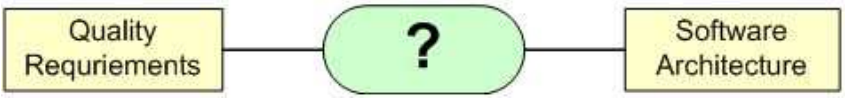

Figure 1 - The gap between software architecture and quality requirements

This thesis is divided into two main parts: the first one presents a state-of-the-art discussing the concepts from the literature and presenting answers to several objectives stated above. The second part is a practical research solution. It consists of three proposed methods which handle quality requirements in software architecture design.

1.3 Value of the study

Software architects need to understand the meaning of quality attributes and quality requirements that constrain these attributes, based on which software architecture is developed. The interest of software architecture design in terms of quality requirements increases. More attention should be paid to explore their context, so that guidance is provided how to design from quality requirements to software architecture that addresses these requirements. That is why an methodological support for moving from quality attributes towards software architecture is required, whereas little existing research was found in this area. Possibly, if such mature, verified solutions existed, quality requirements would receive more attention during a system development. This paper is only an attempt of providing such solutions and further analysis and verification of this thesis results are required in order to use the proposed method commonly.

1.4 Research scope and limitations

The domain of this thesis is based on the quality requirements that are able to be fulfilled at the highest abstraction level, i.e. during the software architecture design. The requirements engineering phase is omitted; it is assumed that the quality requirements are specified and prepared, so that an architect is ready to take them into account in design activities.

This paper concentrates on all of the eight architectural patterns categorized and described by Buschmann et al. [9]. The ISO/IEC 9126 quality model [20] is used for the quality attributes description. Several quality attributes have passive influence on software architecture at the design level as they are only observable during the system execution (operational quality requirements [7]). These are excluded as they are neither benefit nor liability at the architectural design level.

The author thinks there are four major limitations in this research. Firstly, quality requirements are often vague, neglected or weakly specified. Secondly, the lack of information about the quality attribute influence on architectural patterns and various types of software architecture structures in general. Small number of literature sources is available are

used to collect data about quality attributes impact. These are: Buschmann et al. [9], Bosch [7], and Svahnberg and Wohlin [29][31]. Thirdly, neither Buschmann et al. [9] nor Bosch [7] use the ISO/IEC 9126 quality model [20] for specifying quality attributes. Finally, the lack of design methods that guide software architecture design for quality. This however lacks in similar quantitative recommendation frameworks in the field of software architecture design. Only one, similar research was found that investigated this area by Svahnberg and Wohlin [31] and the research in this thesis is based on this paper.

1.5 Structure of this study

This paper is organized as follows. It is important to settle on definitions and terms used in this paper. Much confusion can be avoided by agreeing on a set of terminology and establishing a glossary will help to avoid misunderstandings caused by the wide variety of definitions in this software engineering field. Hence, little background information and is required. **Chapter Two** defines software architecture, its descriptions and related issues. **Chapter Three** presents the notion of software quality to magnify the concept of quality requirements and quality attributes. Several quality models are described as a method for quality attribute specification. **Chapter Four** defines the software architecture design in terms of quality requirements and in general. This part also presents the definition of patterns, their categorization and influence on quality attributes. Software architecture evaluation is also presented in this part as a part of the design process that ensures and verifies whether a software architecture has fulfilled its requirements. **Chapter Five** illustrates the empirical study construction and its results required for further analysis. **Chapter Six** uses the questionnaire outcomes to define a recommendation framework, i.e. a design support for choosing an architectural pattern that suits best the given quality requirements. Afterwards, the validity of the framework is verified in **Chapter Seven**. Finally, the entire research is summarized in **Chapter Eight** and some significant conclusions and recommendations are given.

Chapter Two – Software Architecture

2.1 Introduction

The result from the software architecture design activity is a software architecture, which has become important field of study in recent years. This increased focus is a result of software architecture benefits including system understanding, documentation, communication tool, architectural drifts, and components reusability.

Software architecture deals with the design and implementation of the structure of the system at high abstraction level. It results in a composition of a number of architectural elements called components in a certain way (as means) to satisfy the software functionality and quality requirements.

More attention is paid to exploring its context, but there is still no single, standard or commonly accepted definition of the term. Many authors and researchers have provided its own, but it is hard to find a one-good, suitable definition. There is also a difference between the terms *architecture* and *design* which are often used as synonyms. *Architectural patterns* are similarly considered as *architectural styles*; *quality requirements* are referred as *non-functional requirements*, *system properties*, *constrains* and many others. The lack of a clear specification and hence misunderstandings among these terms causes much confusion in software engineering. Therefore, it is important to settle on definitions and terms used in this paper. Much confusion can be avoided by agreeing on a set of terminology and establishing a glossary will help to avoid misunderstandings caused by the wide variety of definitions.

2.2 Definition

A number of definitions of software architecture have been proposed so far. One of popular is introduced by Bass et al. in [2]. As it was used in many research documents, it will not be presented here. Most of found were concerned about the system structure, its parts and the relationships among them. Of course they vary in detail, but generally involves the system presented as a view of components and connectors. The process of dividing the system into these components and connectors is called software architecture design and software architecture is an artefact of this activity. Software architecture definitions leave open questions about levels of abstraction that should be provided by an architecture. The literature also emphasizes that software architecture is seen as a method to address the system's complexity.

Another, less popular definition that seemed to be rather exhaustive and especially useful to the purpose of this thesis will be brought:

> *"A software architecture is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system. The software architecture of a system is an artefact. It is the result of the software design activity."* [9, p. 384]

In this paper, software architecture is concerned about the point of designing towards quality requirements. This definition by Buschmann et al. reveals important aspects of the architecture as a structure and this structured form covers system requirements, including quality requirements. The software architecture allows for early evaluation of a design to verify whether quality requirements are covered. "*Software architecture is closely coupled to how well a system achieves various quality attributes"* [3, p. 5]. Therefore, architecture should be regarded as an important issue covering the quality aspects of software applications and much attention shall be placed on modelling and describing the software architecture as a design artefact. The following sections depict different approaches to architectural design.

2.3 Common elements

Software architecture is commonly defined in terms of *components* and *connectors* – represents their topology. The Unified Modelling Language (UML) in version 2.0 defines a component as follows:

> "*a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behaviour in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics)*" [33, p. 6].

Moreover, a connector is defined as *"a link that enables communication between two or more instances. The link may be realized by something as simple as a pointer or by something as complex as a network connection*" [33, p. 7].

There are many kinds of components, some of which have the same properties. Components, which encapsulate some coherent set of functionality, interact with each other using interfaces they provide in a defined way to fulfil their responsibility to other components. A component is independent from the context in which it is used to provide functionality. At the lower-level software architecture (e.g. programming language level) components may be abstract to elements such as modules, packages, classes, objects, or even a set of related functions or methods. Connectors realize the communication, cooperation, and interaction between components. Their main responsibility is to describe relationships between the components. The system achieves certain qualities based on the composition of its components and connectors, which form the architecture. However, as it was presented in previous section, the meaning of software architecture extends far beyond the definition of components and relationships between them.

The term component is used as a high-level design element. Components differ from other 'objects' from the level of abstraction they concern [13]. Components are fundamental architectural building blocks, whereas 'objects' are runtime entities of a lower-level design. Unlike components, 'objects' have an identity; they are arranged into hierarchies according to their inheritance relationships.

To summarize, software architecture definitions are in concern with:

- major (high-level) components,
- component behaviour,
- decomposition into certain structures,
- interactions between components through connectors.

These principals set the fundamental system structure. Architecture definitions do not define however what a component is. It is presented though as a software element.

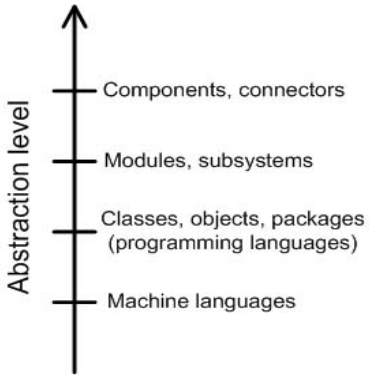

Figure 2 - Software elements at different abstraction levels

2.4 Architecture Description Languages

Architectural Description Languages (ADLs) are formal, modelling languages for describing the software architecture. They are represented by a formal notation (semantics) and also using graphical representations that corresponds to the textual notation. Natural language (written text) is also a modelling language. A number of ADLs have been introduced for modelling architectures to provide extensive means in modelling capabilities and tool support. The architecture description should provide input for the analysis of quality attributes. However, those methodologies are strong in representing functionality but tend to be weak in representing quality requirements. Architectural Description Languages that are concerned with object orientation can though represent, but not directly, some of the quality attributes such as maintainability, understandability, and reusability. An ADL presents architecture in one view only. Since the architecture is not a simple flat view of component and connectors, multiple views are used to understand the architecture comprehensively. Multiviewed approach to software architecture is required for managing the complexity of designing and developing software systems. Multiple views started becoming popular in software architecture with the development of modelling concepts and notations of the *Unified Modelling Language* (UML).

2.5 Views

2.5.1 Introduction

Multiple views provide representations of the software architecture that can be used to guide its construction. A view provides a useful vehicle for communicating the architecture to different stakeholders. They provide a multiviewpoint framework for software architecture. They also manage complexity, i.e. multiple views enable decomposition of the designed architecture. The view models address a static structure of the architecture, dynamic aspect, physical layout, and also the development of the system. An architect is responsible is to decide which view should be used to describe the software architecture. The main purpose

of using views from the point of this research is that different views exhibit different quality attributes important during software architecture design and evaluation.

Bass et al. in [2] underline that software architecture defines the overall system's *structure*[1] since software systems exhibit many structures. The most common and useful structures (views) are: *module*, *conceptual* (logical), *process* (coordination) and *physical* software structures. Each of them helps to exhibit different quality attributes, and that is why it is important to mention about different types of software structures. Each structure is an abstraction of the system with respect to different criteria. Moreover, each structure may use different notation (description language), including its own signification of system components and relationships among them. Structure(s) included in software architecture are not visible to the system's end user.

According to the definition of software architecture in terms of components, connectors and relationships between them, a view is a set of specified components and connectors that describe a software architecture. Each view has its own definition of these architectural elements.

2.5.2 RM-ODP

Another approach considering architectural issues is the *Reference Model for Open Distributed Processing* (RM-ODP) described in [19] by the International Standards Organization (ISO). RM-ODP is a formal standard that serves guidance how to describe distributed object-oriented software architectures. The model is quite general, and therefore it is used in various application domains. The model defines a practise for software architectures that investigate the properties of distributed software systems, i.e. provides a framework for the development of distributed processing. The RM-ODP introduces the concept of a *viewpoint* to reveal a certain set of system concerns. There are five essential viewpoints that serve a comprehensive model for a single software architecture. These viewpoint are:

1. *Enterprise viewpoint* defines the system in terms of business requirements, system objectives, policies, and purpose. It is directed towards user needs.
2. *Information viewpoint* deals with the information structure and objects. It is an activity when elements of the system are modelled.
3. *Computational viewpoint* handles decomposition of the system into objects and their interfaces and behaviours. This viewpoint supports dynamic behaviours that are specified by the information viewpoint. It uses logical partitioning of the distributed systems independently of an environment.
4. *Engineering viewpoint* defines the relationships between the distributed objects and presents methods of supporting behaviours between these objects.
5. *Technology viewpoint* decompose the system into software and hardware components. It identifies possible technical structures.

Each of these perspective is object oriented, and provides a model for the system from the given viewpoints. The first three viewpoints define software architecture making the distributed computing transparent. Usually, the Unified Modelling Language is used as formal notations for describing each of the software architecture viewpoints. The RM-ODP viewpoints provide a separation of architectural issues that divide the software architecture into functionality and the distributed computing aspects.

[1] The term *structure* by Bass et al. [2] is used synonymously with *view*.

2.5.3 The "4+1" view model

The literature provides several *view models* that consist of a number of architectural views. Each view reveals different aspects of the software architecture. The *"4+1" View Model* described in [22] by Philippe Kruchten focuses on describing object-oriented systems. The model is composed of five main views (perspectives):

1. *Logical view* addresses the functionality; it is a object model of the design.
2. *Process view* depicts the concurrency and synchronization aspects of the design.
3. *Physical view* presents how the software is combined onto the hardware and reflects its distributed aspects.
4. *Development view* captures the static organization of the software in its execution environment.

The fifth view (+1) provides *scenarios* or use cases that tie the other four views together and help to validate the design in the other views. Each view has its own particular notations and may use different patterns that guide their composition, and therefore allow multiple styles in one software system.

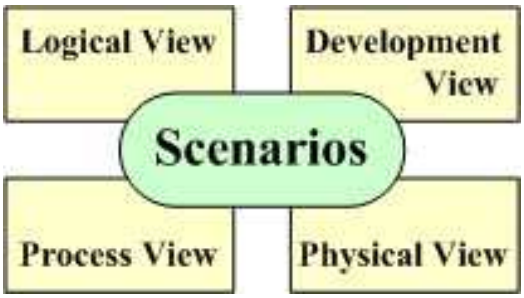

Figure 3 - The "4+1" view model

2.5.4 Hofmeister et al. design method

Another important model to understand the architectural issues facing designers presented in [16] by Hofmeister et al. depicts four views that address different engineering concerns. *Conceptual view* pays attention to appropriate decomposition of the system without delving into details. However, it handles some global system properties such as performance, maintainability or dependability. It also makes architecture available to different stakeholders (end-users, developers, project managers, marketing, etc.). *Module view* maps and controls system's functionality. It addresses how the solutions of conceptual architecture can be realized in today's software platforms and technologies. *Execution view* is often used for distributed or concurrent systems. It maps the components (with the functionality included inside) onto the processes and platform elements of the physical system. Last, but not least, the *code view*. It describes the organisation of the architecture elements are mapped into the implementation.

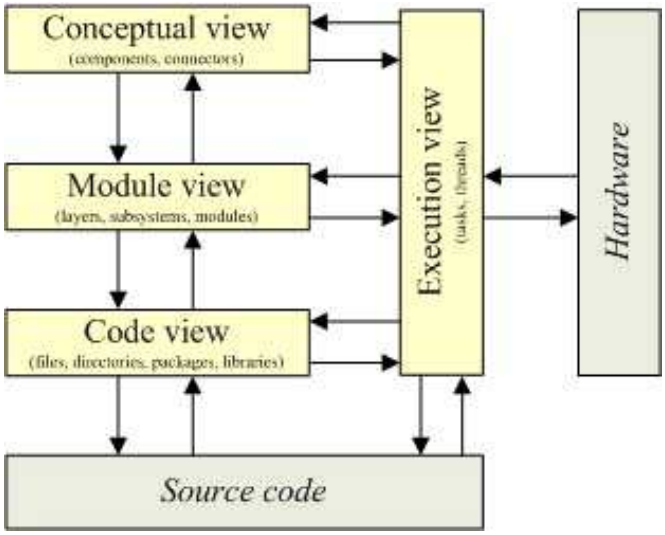

Figure 4 – Hofmeister et al. view model

2.5.5 Summary and remarks

The Hofmeister et al. four view model is quite similar to Kruchten's. The logical view resembles conceptual view, process view resembles execution view, etc. The important this is that views are used to express different aspects of the architecture using an appropriate way. Views that best fit the situation should be used. One way to select the view set is to use previous experience and look at similar architecture solutions that used views. An architect can also focus at the stakeholders needs, and complexity of the system. However, from the goal of this research, looking at system's quality attributes can help to decide which views provide the information relevant to deal with them. Architectural patterns presented in [9] by Buschmann et al. as the software architecture descriptions represent the conceptual view by Hofmeister et al. [16] and logical view by Kruchten [22].

2.6 Software requirements

Software requirements, defined during the early stages of a system development as a result of requirements specification, consist of *functional* and *quality requirements*. Both of these are important because they provide basis for all of the software architecture design activities. The requirement specification is used as an input for architectural design. Once the requirements are set, a software architect initiates the design and other technical work follows: development, testing, implementation. Software architecture design is about converting the requirements into software architecture that fulfils these requirements. The meaning of software architecture design is discussed in detail in **Chapter Four**.

Functional requirements are defined by Sommerville in [28, p. 100] as *"statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations"*. Functional requirements describe what the system must do. They are also called *behavioural* or *operational requirements* because they specify the system's possible inputs and outputs (interactions with between the system and its external world), including their behavioural relationships among them. Typically, a functional requirement is implemented in the system as one or more components or modules that fulfil some part of the application functionality.

"While developers were used in the past to concentrating on providing the stated functional properties for software, today non-functional properties are becoming increasingly important" [9 p. 389]. Bass et al. in [2] state accurately that architecture addresses a lot more than just functional requirements. Nevertheless, these requirements put constrains how functional requirements are ought to be implemented. Unlike the functional requirement describing *'what'* the system will do, a quality requirement describes *'how'* it will do it. The definition, concept and nature of quality requirements are presented in **Chapter Three**.

2.7 Styles and patterns in Software Architecture

The literature such as [7] distinguishes between *architectural styles* and *patterns*. Styles are a categorization of systems and patterns exhibit general solutions to common, recurring problems. Also, patterns tend to be more detailed than styles. However, they are often synonymously termed as they provide a common usage and vocabulary. That is why the term *'pattern'* will be used in the following and further discussion.

Software architecture design consists of activities needed to specify a solution to balance the fulfilment of the requirements. In order to properly design the architecture an architect should know how particular design problems are solved and to be able to compare and discuss different candidate choices. Since the size and complexity of software systems continuously increase, experienced software designers and engineers use of certain, predefined ways of organizing software elements, because of the properties these structures provide. Patterns are one of these approaches to designing software architectures.

Patterns are in general an essential tool in software architecture design that support the development, maintenance and evolution of large-scale systems. providing documented and communication proven design solutions to recurring problems, that also ensure a problem context, not only the specific results they propose. Patterns are recognized for many uses such as common design vocabulary, documentation and learning aid as well as their solution trade-offs.

Their role has become important in describing software architecture due to the influence on software quality. Not only patterns are used to fulfil functional requirements of a system, but also, what is important to this research, help to address quality attributes corresponding to quality requirements. Literature sources, such as [2][7][9][16], prove that quality attributes are affected by decomposition of components and their responsibility. Different arrangements of components affect different quality attributes of the designed architecture without affecting the system's functionality. This fact brings software architecture design in suitable ways for the purpose of this paper. However, patterns help to address only development quality requirements. It is almost impossible to evaluate the operational quality requirements at the architectural level. Descriptions, categories and detailed concepts of patterns are further described in **section 4.2**.

2.8 Summary and remarks

The most important concept of this chapter in terms of this research is that certain infrastructures of software architecture elements, i.e. components and connectors, cover to some degree several quality requirements. Patterns are recognized as topologies of such elements, and hence they are a great 'tool' for addressing quality requirements at the architectural level.

Systems are built to satisfy their requirements. Software architecture design determines whether the software architecture has fulfilled system requirements. There is still lack of knowledge and what matters the most – little practical guidance on how to manage the design activity. There is a lack of precise design methods that guide software architecture for quality. Usually design means taking steps to provide the system with its expected functionality. However, a number of different attributes, properties, or qualities are of interest during software architecture design. These attributes are of crucial importance because they constrain quality requirements, which in turn constrain the design and development of software architecture. **Chapter Four** presents a detailed concept of software architecture design and related issues.

It is obvious that the same requirement specification given to two different architects will produce two different architectures. The question is – how can we determine which one of them produced better architecture? There is no place for statements like good or bad architecture. Those candidates are less or more suitable for the stated purpose. One way to check whether the requirements were addressed, and assuming that all of them were covered, the next step is to measure *'how well'* these requirements are fulfilled.

Patterns can be very useful. On the other hand, if misunderstood, they can lead to disastrous solutions. The most important is whether a patterns fits the design and is the most applicable choice among others. Large-scale software systems will incorporate many patterns in their design as it is almost impossible to describe a large system with a single pattern. This leads to an observation about the way the quality requirements are concerned to be fulfilled or not by architectural means. The activity of measuring whether and to what degree quality requirements are coved by the architecture candidates is called software architecture evaluation (see **section 4.3** for details).

Chapter Three – Quality Requirements

3.1 Software quality

This part should be introduced by the definition and meaning of *software quality*. Before dealing with quality, one has to realize what it really is to be able to control it. Understanding clearly the concept of quality in software makes it easier to be aware of its importance in software development. An increasing number of standards in the field of software quality emphasize the need for its observation and measurement. Literature also proves that the notion of software architecture has achieved the appropriate level for dealing with quality. Mostly it is due to many researches in quality attributes field ([2][3][7][9][16][17][18][29][30][31][32] and many others). It is also recognized that architecture sets the software quality boundaries of the resulting system. This thesis underlines that the functionality of a system itself does not guarantee required software quality. Also, achieving quality is not simply checking if requirements are met; it includes specifying the measures and criteria to demonstrate their level of achievement. That is why quality requirements are extremely important and their role is investigated in this research. However, software quality assurance is an afterthought in most designs.

Quality is subjective due to the lack of formalism and consensus in definition. Quality is defined in [20, p. 20] *as "the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs"*, whereas software quality definitions concern conformance to requirements. A good definition is presented in [18] in the following way: *"the degree to which a system, component, or process meets customer or user needs or expectations"*, and *"ability of the system to satisfy its functional, nonfunctional, implied, and specified requirements"* introduced in [1]. Software quality is often sacrificed in order to keep low development costs and project on schedule. The quality aspect can be attributed to process and product quality.

The challenge of software development is to ensure that software has the right quality levels. More efforts are concentrating on ensuring that the quality is addressed at the architecture design level before the system is actually implemented. The best way is to measure the level of quality using quality models described in **section 3.3** as they propose a more structured, fixed, and what is important quantitative view on software quality. In general, a quality model depicts how composition of particular quality characteristics and their relationships affect the total software quality.

Architectural decisions have a great impact on the final software quality. It is possible to tell whether the most suitable architectural decisions have been made during the design without having the system developed and deployed if the system exhibits its required quality attributes [2]. That is why it is necessary to evaluate how a software architecture meets its quality related issues at the software architecture level. **Section 4.3** focuses on the importance of software architecture evaluation as a method of identifying potential risks and verifying that the quality requirements have been addressed during the design.

One of the goals of this research is to reveal the importance of quality requirements in software architecture and to ensure they are always taken into account during a design. Paying attention to them will bring nothing but benefits and increased software quality.

Therefore, this general discussion of the software quality was presented right before one of the "core" concepts of this thesis, i.e. quality requirements explained in detail in the following section. Also, many researchers have proposed their own categorization of software quality, which resulted in proposing several quality models described in **section 3.3**.

3.2 Quality Requirements

3.2.1 Introduction

It is worth to mention at the beginning that there is no one-standard, universal definition of quality requirements. Also, different people use different terminologies. Quality requirements are also recognized in literature as *non-functional requirements*, *non-behavioural requirements*, *system properties* or *constrains*. Bass et al. [2] underline that terms of these requirements that consider the lack of functionality is an inappropriate term. A number of literature sources, and what are important world standards use the term quality requirements, and therefore it will be used in this paper. Their definition has to be customized in order to be properly used [10]. Hence, a definition, meaning, their categories and other relevant information from chosen literature sources will be presented.

3.2.2 Definition and concept

Different from a functional requirement (FR), a quality requirement (QR) defined in [17] as *"a requirement that a software attribute be present in software to satisfy a contract, standard, specification, or other formally imposed document"* is a requirement that does not concern functionality. As the name suggest, they are concerned with the quality delivered by the system. They *"place restrictions on the product being developed and the development process, and they specify external constrains that the product must meet"* [26, p. 187]. In other words, quality requirements determine constrains on the functionality.

Quality requirements determine the overall qualities, attributes, or properties of a software system. Functional requirements describe '*what*' a system is expected to do, whereas quality requirements put constrains or restrictions on '*how*' these functional requirements are ought to be implemented. This means they are more above the functionality, i.e. system services, capabilities and behaviour. In consequence, functional requirements may need to be sacrificed in order to be able to address quality requirements [28].

Quality requirements may affect either one part of an application (concern one functional requirements abstraction of a system) or the system as a whole. To understand their importance it is worth to mention that some functional requirements may need to be sacrificed in order to meet the system quality requirements, and in result – the product goals. Furthermore, the lack of a system service (functional requirement) may degree the system usability, while not covering a quality requirement could make the system totally useless [28].

3.2.3 Quality Attributes

A software system has many characteristics such as maintainability, reliability and usability. The quality of each of these characteristics determine the total software quality. Each characteristic can be specified as an property (attribute) of the system. A quality attribute is *"a characteristic of software, or a generic term applying to quality factors, quality subfactors, or metric values"* [17]. The previous section defined quality requirements.

In other words, it is a measurable or observable property of a system that has some qualitative or quantitative value. Measurable means that a metric is given on how to verify that the architecture addresses the quality attributes. For example performance is a quality attribute. Helpful for the introduction of quality attributes (also referred as *qualities*, or *"-ilities")* is a definition of a quality requirement as *"specification of the acceptable values of a quality attribute that must be present in the system"* [1]. Quality requirements put constraints on a quality attributes. They are usually specific values, a scope, or ranges of values for quality attributes. *"Quality requirements that can not be quantified can not be controlled either"* [8, p. 77]. It means that in order to be able to satisfy quality requirements and generally – the quality of a software system, quality attributes have to be quantified. Having a requirement that system shall handle a specified amount of connections concurrently, then that is a requirement on quality attribute represented quantitatively, i.e. a quality requirement. Analogically, the response time shall be less than a time unit, is another example of a constraint put on performance. The same attribute, two different quality requirements.

3.2.4 Quality Attribute impact

Architecture design decisions have proven to impact certain quality attributes, which are not mutually exclusive – they often affect each other positively or negatively. *"Non-functional properties may contradict as well as complement each other"* [9, p. 410]. Some quality attributes strengthen (positive impact) each other like flexibility and maintainability, safety and security, or maintainability and portability. Bass et al. mentions that *"no quality can be maximized in a system without sacrificing some other quality or qualities"* [2, p. 75]. In other words, as previously stated, some quality attributes may hinder others (negative impact). Some relationships are greatly proved by the literature. For example: "*The benefit of exchangeability comes at the price of increased programming effort and possibly decreased run-time performance"* [9, p. 49]. Other negative interdependencies include similarly maintainability and efficiency, security and usability, security and performance, etc. The quality attribute relationships are not "set in a stone", i.e. the impact may be stronger or weaker depending on attributes and their design context.

Since the quality attributes are interdependent, the design is a difficult task. In worst case, every design decision impacts multiple quality attributes negatively. Each architecture candidate has to be evaluated to check its impact on desired quality attributes. Further design decision may neglect previous ones. If determined that attributes are in conflict, it is important to find an architecture that provides an appropriate compromise. "*When specifying non-functional requirements for a software architecture, you need explicitly consider the interdependencies and trade-offs that exist between them*" [9, p.410]. A good design balances all the quality attributes, usually according to their *prioritization* as well as *trade-offs* (**sections 3.2.6** and **3.2.7** respectively).

To summarise, three types of quality attributes relationships are identified:

- **passive** impact – a quality attribute does not influence the other,
- **positive** impact – high value on a quality attribute determines a high value on the other,
- **negative** impact – high value on a quality attribute determines a low value on the other.

Figure 4 from [25] illustrates how identified quality attributes influence each other. However, quality attributes are not specified in terms whether they strengthen or hinder each

other. The illustration indicates only that a relationship between a pair of attributes exist – one depends on another. High impact between attributes is presented with a light circle. Dark circle describes low relationship, whereas a blank field denotes passive influence (no dependencies).

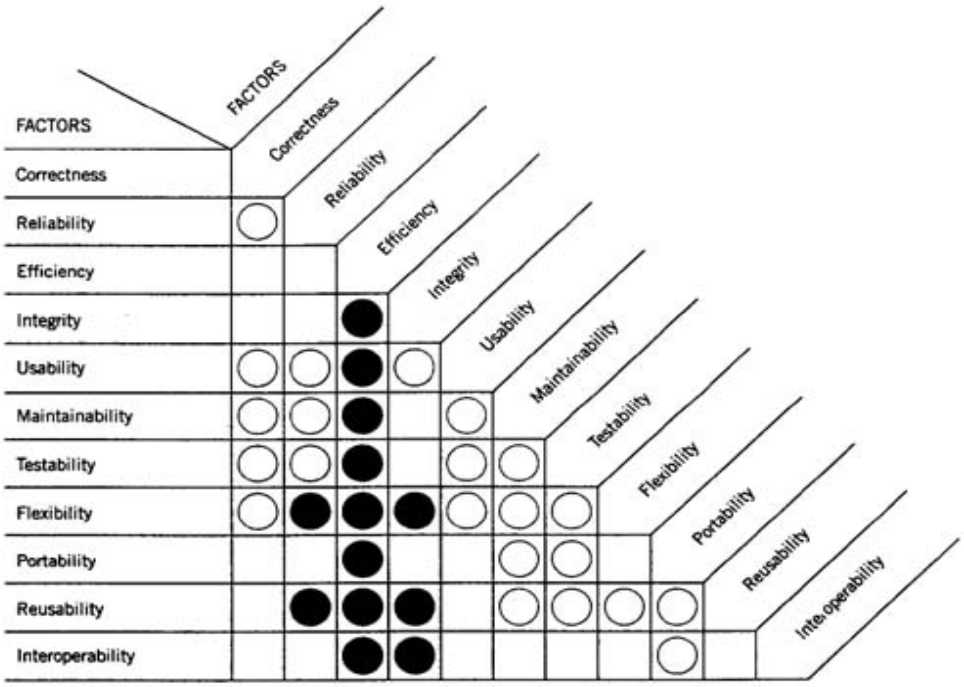

Figure 4 - Quality attribute impact and relationships [27]

3.2.5 Quality requirements categories

Bosch in [7] categorized quality requirements as *development* or *operational*. Development quality requirements are those qualities relevant from a developer point of view, from the software engineering perspective, e.g. maintainability, demonstrability, extensibility, flexibility, reusability, portability, etc. Maintainability, for example, is more important to developers, because it enables the chance of a system to make changes, fixing bugs, and further development of the system. On the opposite, there are operational quality requirements important from the user perspective, because they are noticeable and measurable during the system's runtime (*"qualities of the system in operation"* [7, p. 27]) like availability, efficiency, flexibility, performance, security, usability, etc. Users see the performance more important, as it affects the usability of the system. Bass et al. [2] used similar categorisation against which the designed system can be measured – attributes *observable via execution* and those *not observable via execution*. Some qualities, such as flexibility or understandability, are important from both perspectives and therefore, could be classified into both categories depending on the quality model is used. A good point to mention is that quality of development quality requirements is inherently difficult to measure. Performance and reliability (operational quality requirements) may be measured to certain degree using numeric criteria (e.g. by executing the system), but attributes like maintainability are almost impossible to measure without previously stating what these qualities mean, especially to different groups of stakeholders.

Kotonya and Sommerville in [27] classify quality requirements[2] into three major groups: *product requirements* (reliability, usability, etc.), *process requirements* (delivery, implementation, standards) and *external requirements* (economic constrains, legal constrains, interoperability). Product requirements are similar to Bosch's operational requirements; they specify product behaviour.

Quality characteristics and associated metrics (**section 4.3.3**) are used to defining quality requirements.

3.2.6 Prioritization

Functional requirements usually have an associated priority: required, preferred or optional. Why not do that with quality requirements? It is a significant task to prioritize quality requirements too. Prioritizing quality requirements is crucial since not all of them are created by equal means. Moreover, different quality attributes are not of equal importance.

In order to balance between the specified quality requirements a priority has to be assigned to each of them to indicate how important they are. Stakeholders are commonly responsible for establishing priorities. Different stakeholders have varying interest and thus prioritize quality attributes in a different way. If they decide that all requirements are equally essential, the harder it will be to achieve an effective balance. It is highly recommended to establish a preference of one quality requirement against another (others) in case of conflict.

Customers and developers must settle on an agreement on requirements prioritization. One prioritization scale may not be enough, while sometimes different stakeholders need different scales. Developers will not know what is important to the customers, and of course, customers cannot specify the cost, effort, time needed and technical difficulty associated with some quality requirements. Especially that quality requirements are often invisible to customers. Once quality requirements are specified and classified, they have to be decided on which must to be implemented and which ones could be rejected if there should be a shortage of budget, time or in case of technical difficulties.

As a result of prioritization activity, quality requirements will be weighed according to their importance. Priority is a function that provides values necessary for comparing quality requirements, and from the position of this research it enables to select the most appropriate software architecture among alternatives with similar properties.

3.2.7 Trade-offs

Trade-offs are about analyzing quality requirements possibilities with regards to how well a software architecture meets each of these requirements, and reasons about their possible conflicts. This often enables further quality requirements refinement and according to quality attribute impact (**section 3.2.4**) this might exhibit new conflicts. Besides prioritization, a good way of balancing quality requirements are their trade-offs. Software architecture design involves a series of trade-off decisions among quality requirements to obtain a compromise design which best meets these requirements. It is important to make trade-offs early in software architecture design because such decisions are hard and expensive to be implemented in further stages.

Designs almost always require trade-offs between competing design choices to meet quality requirements. Large-scale software systems often do not fulfil all of their quality requirements, but select the most suitable architectural solutions using trade-off mechanism

[2] Gerald Kotonya and Ian Sommerville in [27] use the term *non-functional requirements* similarly, but not equivalent to *quality requirements*.

with respect to significant parameters between different quality requirements. By significant parameters two things are meant that can be used on a qualitative and quantitative basis:

- priorities between different quality requirements discussed in previous section,
- positive and negative impact of quality attributes on each other.

These factors are used during the early design phases, and of course, during the software architecture design.

Based on little personal experience in software architecture design an example of trade-offs analysis is presented. Without any automated support and hence – based mostly on architects knowledge and intuition two candidate architectural structures were chosen during the design activity. These were described in terms of their benefits and liabilities so that in future persons in interests can look why certain solution was accepted and the others rejected. Finally, the rationale for the final decision between potential solutions was described one of the architectural designs proposed for this system. **Figure 5** illustrates a trade-off analysis used in the provided example.

1. Trade-off analysis

1.1. Structure A

Benefits:	Liabilities:
+	-
+	-
+	-

1.2 Structure B

Benefits:	Liabilities:
+	-
+	-
+	-

1.3. Rationale for the chosen solution

Figure 5 - An example trade-off analysis method

3.2.8 Quality Requirements in practise

Developers are constantly under pressure to deliver the software product on time and on budget. In result, projects lack in quality requirements as they tend to focus only on delivering functionality. Insufficient time and effort are spent on the quality requirement-related activities associated with the design of software architecture. As it will be presented, a better approach is to invest additional time on eliciting, gathering, analysis and generally handling quality requirements. It will benefit in the final software product quality. Thus, the total quality of a system is ultimately determined by the quality of each requirement. By leaving them unstated, the software system lacks in quality and in worst case – lead to a series of failures in software development and afterwards during the system usage. However, these types of requirements often are neglected. Many software requirements specifications

(also called software requirements documents), being an official statement of what is required, are either full of badly written (quality) requirements or do not specify them at all. Most applications lack in these areas that are not concerned with functionality. This is often a result of the system's complexity and badly specified needs. If they are specified at all, they are of poor quality, i.e. incomplete, inconsistent, ambiguous, or incorrect. Their completeness means that all quality requirements should be defined. Consistency means that their definitions should not contradict each other. Quality requirement is unambiguous when it cannot be interpreted in more than one way. Correctness means that it should accurately reveal system needs.

Software architecture notations should be capable of stating quality requirements. None of the studied addresses quality aspects of the architecture. UML use-case models are used to present the functionality of a system expressed by functional requirements. Quality requirements, on the other hand, are often described below them in supplementary text or as footnotes. **Table 3** illustrates such examples. Notations should have the ability to visualise quality requirements, or at least support their estimations while a difficult task is to present graphically the above example.

Another problem of quality requirements in requirements specifications is how to specify the notion of software quality. Much attention should be paid on its understanding and so that all participants share the same meaning of the quality aspects. Everyone has to agree on how quality requirements have to be quantified and, in consequence measured if they address the specified level of quality.

On the opposite of functional requirements, quality requirements are often hard to specify, test and verify. Design methodologies are strong in expressing functionality but tend to be weak when it comes to quality requirements. There is little precise guidance available on how to elicit and specify quality requirements. Main reasons include misunderstanding of their importance, their mutual dependencies, inadequate languages or inappropriate formalism of expression, and many more. Those are the common reasons why they are afterwards addressed subjectively. This results in architectural solutions that badly address quality requirements. Mostly because of the inherent difficulty in designing for quality requirements, which is provoked by the lack of documented patterns and their benefits and liabilities for certain quality attributes that guide the design for quality requirements. However, an important step towards designing with quality requirements is Bosch's design method [7] depicted in **section 4.4.2**.

3.2.9 Summary and remarks

Quality requirements does not concern functionality. As the name suggest, they are concerned with the quality delivered by the system. They *"place restrictions on the product being developed and the development process, and they specify external constrains that the product must meet"* [27, p. 187]. In other words, quality requirements determine constrains on the functionality.

The goal of this thesis is not to give guidance how to elicit and specify quality requirements, but to investigate their role in software architecture design. Nevertheless, one has to be aware that badly written requirements are useless for their further analysis. Quality requirements specify system attributes, such as maintainability, reliability and safety. They are a result of putting constraints on one or more of these attributes. Attention to requirements is crucial for quality. By leaving certain quality requirements not covered, the system lacks in required quality level.

Many of the quality requirements cannot be measured or calculated before the system is actually implemented, and therefore difficult to validate. Yet they are hard to deal with

since they often tend to interact with each other, having positive or negative influence. However, during the design phase, much of the quality aspects of a system can be addressed. During software architecture design such requirements need to be prioritized and balanced in design tradeoffs when architects have to decide upon the selection of a particular software architecture solution.

Probably the most difficult activity during software architecture design is the transformation from requirements, especially quality requirements into the particular structural or behavioural aspects of software architecture due to lack of methodological and technological available support. Hence, this paper is an attempt of bridging the gap between quality requirements and software architecture.

3.3 Quality Models

3.3.1 Introduction

Similarly to quality requirements, which have no one-standard definition, there is no one-complete, universal list of quality attributes. However, many taxonomies and standards were published to define quality attributes such as IEEE, ISO, and ANSI.

The terms and definitions around quality present rather its qualitative view. Quality models are used to reveal a structurized, and what is important – quantitative view on quality. Their intention is to capture quality in a model since the total quality consist of the composition of particular characteristics. A quality model sets a standard taxonomy for quality attributes and relationships among them. It studies aspects of software systems which relate to the notion of software quality. It also serves a framework for quality attributes within which to analyze requirements and design decisions. There are several well-known quality models such as McCall's (1977), Boehm's (1978), FURPS/FURPS+, and ISO/IEC 9126. The ISO/IEC standard will be detailed described, as the one this research refers to. Two first are briefly mentioned due to the fact that ISO/IEC 9126 [20] was based on the McCall's and Boehm's models. FURPS/FURPS+ is presented as it is a relatively recent quality model proposal, and resembles in its structural manner the other mentioned models. The main difference between them is the classification and definition of quality attributes, as well as the depth of hierarchy and a different total number of characteristics.

3.3.2 McCall's Quality Model

Jim McCall and his colleagues in [26] organized the software product quality into three categories: *product operation*, *product revision*, and *product transition*, where to each category a set of quality characteristics is associated. *Product operation* focuses on qualities important from user perspective (operational characteristics). It contains correctness, reliability, efficiency, integrity, and usability. *Product revision* includes maintainability, testability, and flexibility. These characteristics describe the ability of a system to make changes. *Product transition* presents the software adaptability to new environments. It contains portability, reusability, and interoperability.

Figure 5 presents high-level quality attributes, termed *quality factors* in this model. McCall distinguished also a second level quality attributes, termed *quality criteria*, which describe the internal view of the software, from the developer perspective. The model also depicts *metrics* that are defined and used to provide a scale and method for characteristics measurement.

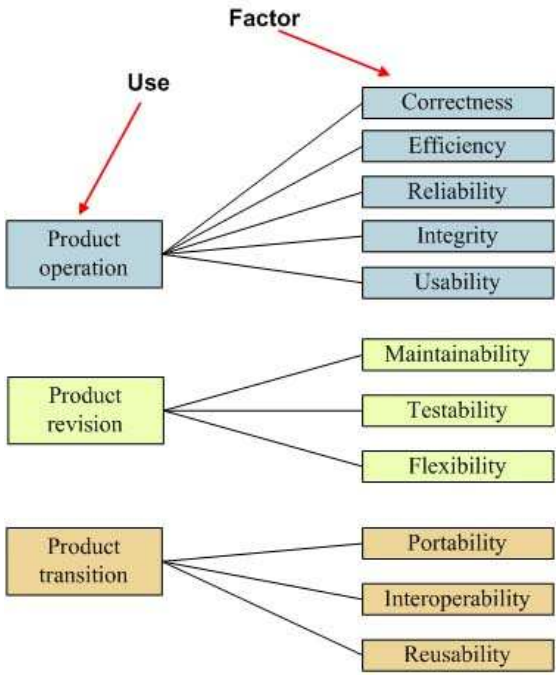

Figure 6 - McCall software quality model divided in three types of quality characteristics

3.3.3 Boehm's Quality Model

Barry Boehm in [6] presented similar approach sharing a common subset with the McCall's model and identifying additional quality attributes. It also presents a model based on hierarchical dependencies among attributes, structured around high-level characteristics, intermediate level characteristics, primitive characteristics (metrics). McCall's quality model was basically focused on the measurement of the high-level attributes (quality factors), whereas Boehm's model considers a wider set of characteristics. Of course, each characteristic of both models set the boundaries of the overall quality level.

3.3.4 FURPS/FURPS+

The FURPS model used by Unified Process is similarly structured as the previous two described models. It provides five following categories of quality attributes:

- *Functionality* – A set of attributes characterizing feature sets, accuracy, interoperability, and security.
- *Usability* – Attributes that depict the usage effort. They include understandability, operability, user documentation, and other human factors.
- *Reliability* – Characteristics that involve fault tolerance, recoverability, predictability, accuracy, and Mean Time Between Failure (MTBF).
- *Performance* – Attributes that consider response and processing time, level of performance in comparison to the amount of resources used, efficiency, and availability.
- *Supportability* – Characteristics that include the effort needed to incorporate new requirements and to make modifications. It also concerns configurability, serviceability, installability, and localizability.

FURPS acronym is named after first letters of each above category. Later the FURPS model was extended by IBM Rational Software into FURPS+ which defines additional quality requirements categories: implementation requirements (constraints on tools, programming languages, and hardware), interface requirements (interaction with external systems), operations requirements (constrains on administration and management), packaging requirements (constraints on system delivery), and legal requirements (licences, law regulations). The FURPS categories are divided into two different types: functional (F) and non-functional (URPS). As it was stated earlier quality requirements are also referred as non-functional requirements such as in FURPS. However, what is interesting that the model defines quality requirements as non-functional requirements, which are grouped into "URPS" categories. Additional non-functional requirements are called constrains or pseudo requirements.

3.3.5 ISO/IEC 9126 Quality Model

Like any other quality model, ISO/IEC 9126 serves a useful tool for quality requirement engineering as well as quality evaluation. Its quality characteristics and associated metrics define a framework for specifying quality requirements, and for trade-offs between software product capabilities. ISO/IEC 9126 quality model enables software product quality to be specified and evaluated from different perspectives. It can be used by different groups of stakeholders, i.e. architects, developers, and testers responsible for dealing with software product quality. It is structured basically like the two, above mentioned models. However, it includes also the functionality as one of the quality characteristics. Functionality is concerned with *'what'* the software does to meet stated and implied needs, whereas the other characteristics are concerned with *'when'* and *'how'* it fulfils these needs. Also, differently from McCall and Boehm, the model identifies both *internal* and *external quality* characteristics. This research deals with internal quality, as the software quality is measured and evaluated by quality attributes. ISO/IEC 9126 also differs from previous models in having a one-to-one hierarchy where each subcharacteristic relates to only one characteristic. Each quality characteristic may be broken down into subcharacteristics, which can also be broken down. **Figure 7** depicts the top-level characteristics with their general meaning.

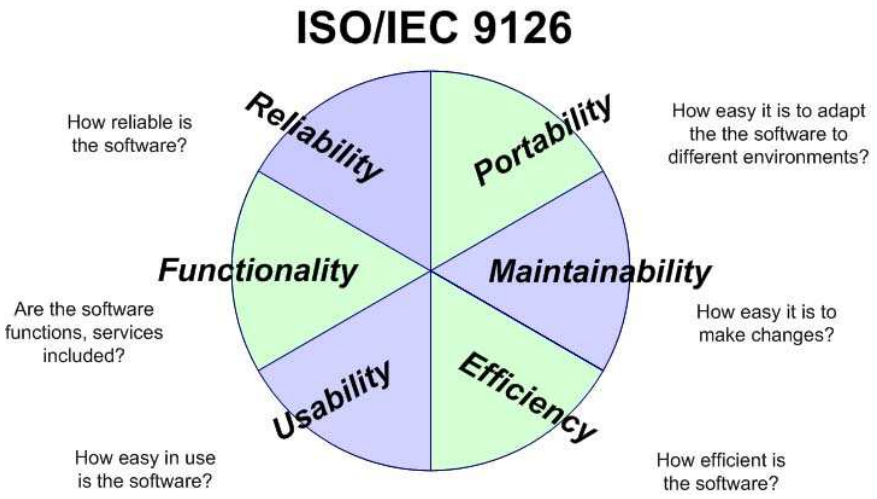

Figure 7 - ISO/IEC 9126 six main software quality characteristics

The model defines three types of software product quality:

- *quality in use* (software product used in a specific environment and context from the user's perspective),
- *external quality* (executable software product),
- and *internal quality* (software product during development).

That is why software quality requirements are defined here as external quality requirements, that specify the level of required quality from the external view, and internal quality requirements which specify the required level of quality from the internal view of the product. ISO/IEC 9126 consists of six internal and external quality characteristics namely: functionality, reliability, usability, maintainability, efficiency, and portability. Each of these is divided into several quality attributes or subcharacteristics, e.g. reliability is composed of maturity, fault tolerance, and recoverability.

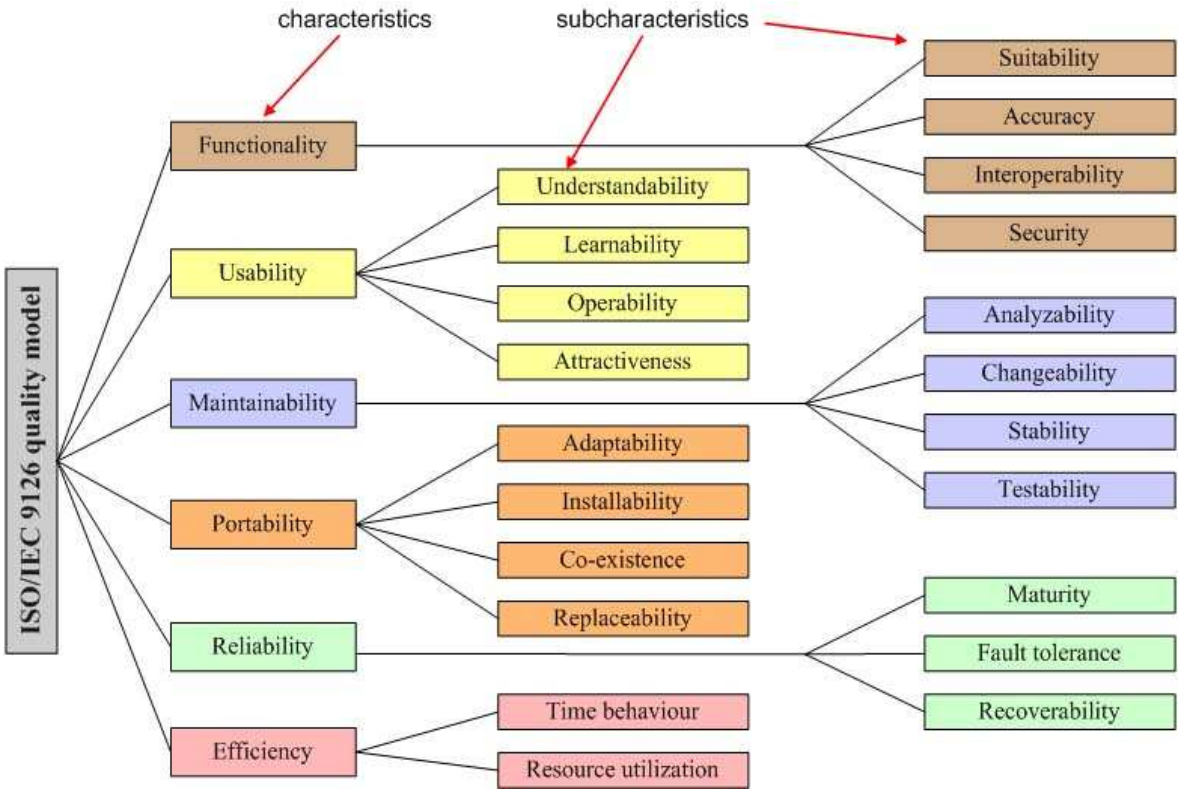

Figure 8 - ISO/IEC 9126 quality model for external and internal quality

Figure 8 presents the hierarchical structure of ISO/IEC 9126 quality model. These subcharacteristics are measured through metrics described in the following section. The top-level characteristics are defined as externally observable features for each software system. Different software products imply its characteristics to be considered of different importance than others.

There are *compliance* subcharacteristics in every of the six main characteristics that were neither listed above in **Table 1** nor in **Figure 8** Compliance means in general to adhere to standards, conventions or regulations in laws concerning the high level and "fellow" attributes at the same level. Adhering to compliance for a top-level characteristic means that the subcharacteristics are considered.

Characteristics	Subcharacteristics	Meaning
1. Functionality		A set of attributes that relate to the capability to provide functions used under specified conditions. The functions are those that satisfy stated or implied needs.
	Suitability	The capability to provide an appropriate set of functions for specified tasks and user objectives. Suitability also affects operability.
	Accuracy	The capability to provide the required or agreed results or effects with the needed degree of precision.
	Interoperability	The capability to interact with one or more specified systems.
	Security	The capability to protect information and data so that unauthorised persons or systems cannot read or modify them.
2. Reliability		A set of attributes that relate to the capability of a software to maintain its level of performance under stated conditions for a certain time period.
	Maturity	The capability to avoid failure as a result of faults in the software.
	Fault tolerance	The capability to maintain a specified level of performance in cases of software faults or of infringement of its specified interface.
	Recoverability	The capability to re-establish a specified level of performance and recover the data directly affected in the case of a failure.
3. Usability		A set of attributes that relate to the capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions. Some aspects of functionality, reliability and efficiency may also affect usability.
	Understandability	The capability to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use.
	Learnability	User's efforts for learning the software product.
	Operability	The capability to enable the user to operate and control the software product. Suitability, changeability, adaptability and installability may affect operability.
	Attractiveness	The capability to make the software more attractive to the user, such as the use of colour and the nature of the graphical design.
4. Maintainability		A set of attributes that relate to efforts needed to make specified modifications. Modifications include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.
	Analyzability	The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.
	Changeability	The capability to enable a specified modification to be implemented.
	Stability	The capability to avoid unexpected effects from modifications of the software.
	Testability	The capability to enable modified software to be validated.
5. Efficiency		A set of attributes that relate to the capability to provide appropriate performance, relative to the amount of resources used, under stated conditions.
	Time behaviour	The capability to provide appropriate response and processing times and throughput rates when performing a software product function, under stated conditions.
	Resource utilization	The capability to use appropriate amounts and types of resources when the software performs its function under stated conditions. Human resources are here excluded.
6. Portability		A set of attributes that relate to the ability of a software product to be transferred from one organisational, hardware or software environment to another.

	Adaptability	The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered.
	Installability	Efforts needed to install the software product in a specified environment.
	Co-existence	The capability of the software product to co-exist with other independent software in a common environment sharing common resources.
	Replaceability	The capability of the software product to be used in place of another specified software product for the same purpose in the same environment. Replaceability includes upgrading.

Table 1 - Quality attribute glossary (descriptions)

Table 1 serves a glossary for the quality attributes used in this research. Each attribute corresponds to a capability of a software product to provide a certain quality, which definitions were introduced above.

3.3.6 ISO/IEC 9126 metrics

ID	Name, contains
ISO/IEC 9126-1	Software Engineering – Product quality, Part 1: Quality model
ISO/IEC 9126-2	Software Engineering – Product quality, Part 2: External quality metrics
ISO/IEC 9126-3	Software Engineering – Product quality, Part 3: Internal quality metrics
ISO/IEC 9126-4	Software Engineering – Product quality, Part 4: Quality in use metrics

Table 2 - List of ISO/IEC 9126 standards

It is not the intention of this section to give an explanation what metrics are, as they were introduced in **section 2.5.3.6**, but to briefly describe the metrics used by the ISO 9126 quality model. ISO/IEC 9126 contains four parts under the general title “Software engineering — Product Quality”. First part defines a quality model for a software product. The second, third and fourth part suggest metrics that define quantitative scale and measurement method, which can be used for measuring quality attributes depicted by the first part: external, internal, and quality in use metrics respectively. External metrics are used in a n executable software product. Different from external, internal metrics do not rely on software execution. They are applicable in a software product during development. Quality in use metrics are used when the final software product is executed only in real conditions.

Quality attribute	Metrics
Functionality	- number of functions suitable for performing tasks - degree to which the functions meet user objectives - security inspections
Reliability	- mean-time-to-failure - probability of failure - rate of failure, availability - number of detected faults - breakdowns occurrence - repair time, time to restart after failure
Usability	- training time - number of interface functions - input and output data items - tutorials, demonstrations - user observations

Maintainability	- failure occurrence after change - number of components requiring modifications - time for identifying operations that cause failures
Efficiency	- transactions/sec - time to complete a task - response time - screen refresh time
Portability	- number of target environments - time to adapt to a new environment - number of components affected by switching environments - installation time

Table 3 - Example metrics

Table 3 presents several ISO/IEC 9126 metrics. These examples illustrate how metrics can be used to verify whether the required quality attributes are fulfilled. For instance, a time behaviour metric aims to measure a) what is the time taken to complete a certain task; b) how long does it take before a system response to a certain operation? That means they can be used during the software architecture evaluation (see **section 4.3**).

3.3.7 Summary and remarks

There are many quality models that suggest ways of dealing with its quality attributes. Presented models are similar in the idea that software quality is decomposed in a number of high level characteristics, which are further decomposed in a number of subcharacteristics (attributes). Metrics (discussed in **section 4.3.3**) are a scale and a method of measuring these subcharacteristics.

Models differ from each other in how software quality is decomposed, i.e. the number of hierarchical levels and the total number of characteristics. McCall's [26] divided it in 11 factors, Boehm's [6] into 7 factors, whereas ISO/IEC 9126 [20] consists of total 21 characteristics arranged in 6 main areas. Sometimes a high level factor from one model is a subfactor according to another.

Literature, such as [6][20][26], provides a useful tool for discussing, planning, and rating the software quality. Each model depicts how its quality characteristics contribute to the whole product quality. It is not easy to estimate which model is of the best quality. Standards are published by a number of agencies such as ANSI (American National Standards Institute), IEEE (Institute of Electrical and Electronics Engineers) and ISO (International Standards Organization). They are developed to provide high-level, systematic and global guidance as they often abstract from detailed descriptions. The ISO/IEC 9126 [20] aims to provide a rational and systematic approach to dealing with quality attributes. It has been chosen as the most suitable for several reasons. The ISO/IEC 9126 standard serves a complete set of metrics for evaluating software quality and contains attributes which other models lack in. Therefore, it is commonly used by the industry. The model also serves solutions independent of technology and situation. ISO/IEC 9126 level of abstraction enables to its general usage and applicability. When composing a requirements document, the appropriate model properties can be filled in for situation at hand. Specifying all characteristics is not a guarantee for accuracy and completeness. On the other hand, the presented model provides little guidance on what should be measured and how the results should be used in the architecture evaluation as it strongly depends on the context and purpose of its use.

Although quality models describe quality attributes, their definitions are often recognized as ambiguous and measuring the amount of quality still remains a difficult task. Software quality evaluation techniques allow measuring several of the quality attributes, but there is still lack of precise methods that could be preformed in straightforward way. A quality model is a useful tool since it brings ideas of measuring quality, but nevertheless it does not depict clearly defined methods.

Chapter Four – Architectural Design and Evaluation

4.1 Introduction

This research investigates the role of quality requirements in software architecture design, but what exactly does that mean? Software architecture and its relevant issues were defined and discussed in **Chapter Two**. Afterwards, software quality requirements were presented in terms of their impact on software architecture in **Chapter Three**. Understanding the terms *software architecture* and *quality requirements* with the analysis of their relationships is the first of the several interests of this thesis. This chapter deals with the wide concepts of architectural design an evaluation.

First of all, there is a difference between the terms *architecture* and *design* which are often used as synonyms. It is said that architecture is design, but not all design is architecture. A related misunderstanding is about the software architecture elements which are named *architectural* and *design* elements. Of course, their usage depends of the level abstraction they concern. This research distinguishes between these terms and recommends similar approach. Therefore, it is assumed that:

1. A software architecture is an artefact; it comprises the highest level description of a system structure.
2. Design (architectural design or software architecture design) stands for an activity that results in a software architecture. Also, design consists of a set of decisions made by the software architect to ensure that the system meets its functional and quality requirements.

However, *architecture* and *design* are termed interchangeably in practise. Often, the amount of detail is insufficient to characterize the differences [13] and the software architecture is seen as a tool that deals with the design and implementation of a structure of the system at highest abstraction level [2]. At the same time it is regarded as one of the most important artefacts. In consequence, a solution is required. For example two phases can be distinguished: architectural design and detailed design with respect to the abstraction level. Another possible solution can distinguish between architectural modelling and architectural design. Either way, but software engineering should mark a clear boundary between the varying degrees of abstraction to avoid pointless confusions.

> *"Software design is the activity performed by a software developer that results in the software architecture of a system. It is concerned with specifying the components of a software system and the relationships between them given functional and not-functional properties"* [9, p. 390]

To summarise, in this thesis a (architectural) design is termed similarly to the above Buschmann et al. definition as a process/activity that involves (design) decisions to ensure the fulfilment of (software) requirements and results in a artefact called software architecture.

Software architecture design involves [23]:

- domain analysis and understanding the requirements,
- designing an architecture to provide architectural solutions in order to meet requirements and desired qualities,
- allocating the requirements into components and connections,
- providing a description of an architecture,
- architecture evaluation with respect to the requirements,
- documenting the architecture with a rationale to design decisions.

A number of mature design methods exist. These provide a series of steps for designing a software architecture. In other words, design methods are ways of representing a software architecture, usually with the help of views. A view is a description of a whole system from the perspective of a related set of concerns. **Chapter Two** presents three following software architecture design methods:

- Reference Model for Open Distributed Processing (RM-ODP),
- The "4+1" view model,
- Hofmeister et al. design method,
- *QASAR - Bosch design method.*

The last one – QASAR is especially marked in italic as it is the method chosen for further analysis. It is described in detail in **sections 4.3.3** and **4.4.2**.

Software architecture design consists of a set of decisions made by the architect to ensure that the system meets its software requirements. The decisions made early in the design process determine greatly the desired quality attributes. These fundamental architectural choices are the hardest to be further changed. Therefore, they are the most significant and require special attention. This includes the usage of certain acknowledged architectural design techniques, i.e. patterns. Patterns are a proved instrument for describing software architectures since patterns represent a solution to a number of design problems. Furthermore, patterns are an essential tool in software architecture design due to the fact they address quality attributes corresponding to the quality requirements of a system. Hence, patterns are categorised in this thesis as means for architectural design.

Software architecture design must in its process have an activity to estimate whether the design result, i.e. software architecture, is capable of fulfilling software requirements. Unfortunately, in practise requirements specifications often lack in quality requirements required for an architectural design and the evaluation. Several methods for evaluating software architectures have been proposed in the literature [7][13] in order to assist design methods the achievement quality requirements. Moreover, Bosch [7] introduced Architecture assessments are performed in one or more development stages. A number of the assessment methods focus on analyzing a single quality attribute. The concept of architecture evaluation, the available techniques and a detailed discussion are presented in **section 4.3**.

Architecture views divide the architecture into parts where each of them describes the system from a different perspective and focus on those aspects that address the concerns of stakeholders. This research has two goals of using views:

- different views exhibit different quality attributes important during software architecture design and evaluation,
- patterns provide support in designing view models, and in composing views based on them.

4.2 Patterns

4.2.1 Definitions and categories

In practice, architectures are usually not developed from scratch. The usage of patterns is an important tool for building high-quality software architectures [9]. Patterns have been briefly discussed in **section 2.7**. This part brings patterns closer to the practical area of this research – the recommendation framework. Buschmann et al. [9] give the following description of a pattern:

> *"A **pattern** for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate"* [9, p. 8]

The terms related to software architecture including components, connectors, and relationships among them are introduced in **section 2.1.3.** and will not be reminded here.

Patterns are divided further in [9] with respect to their range of scale and abstraction into three main categories: *architectural patterns*, *design patterns* and *idioms*. **Figure 9** illustrates Buschmann et al. [9] categories. Architectural patterns define overall structuring principles. They define templates for concrete software architectures providing system-wide organization schemes that refer to the system as a whole. Their description and detailed concept is presented further in **section 4.2.4**.

> *"A **design pattern** provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context."* [9, p. 13]

Table 4 illustrates the difference between architectural patterns and design patterns. Although the idea originates from the confusion between *architecture* and *design* as synonyms, the concept remains the same – the level of abstraction is the difference. Design patterns are medium-scale patterns that regard several smaller architectural units in contrast to architectural patterns. They provide structures for decomposing complex services or components being independent of particular programming language or programming paradigm as it is in case of idioms. The fundamental structure of a software architecture is not affected by design patterns. They rather have strong influence on the architecture of a subsystem or a component. There are eight design patterns introduced in [9]: *whole-part*, *master-slave*, *proxy*, *command processor*, *view handler*, *forwarder-receiver*, *client-dispatcher-server*, and *publisher-subscriber*.

> *"An **idiom** is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language."* [9, p. 14]

Idioms represent the lowest-level patterns. They deal with the implementation matters of particular design issues, specific to programming languages. Sometimes idioms useful for one programming language does not find usage in another. They can also directly address the concrete implementation of certain design patterns. Idioms demonstrate competent use of programming language features such as memory management in C++. Therefore, idioms also are recognized as means for teaching a programming language and a communication tool among developers.

4.2.2 Why Patterns?

This paper focuses on software architecture in the context of patterns for several reasons. These among others previously stated in **section 2.7** are listed below:

- Patterns document existing, well-proven software architecture design experience.
- Patterns provide a common vocabulary and understanding for design principles among different types of stakeholders.
- Patterns are proven means for documenting software architectures structures and rationale for design decisions.
- Patterns help to build and manage complex and heterogeneous software architectures.

These are a general rationale why patterns are used to address the software architecture design activity in this research. However, following reasons deserve more attention in discussion.

Software architecture evaluation is performed to measure quality attributes, so these can be compared to the quality requirements. If one or more of those requirements are not fulfilled, the architecture needs changing in order to improve its quality attributes. Bosch emphasizes in [7, p. 116] that *"with each architectural style*[3] *there is an associated fitness for the quality attributes"* and that is why the choice of the most suitable architectural pattern depends on the system's quality requirements.

If a quality attribute is not covered, there are two types of change – either change the software architecture context or change the architecture itself [7]. Of course, assuming that the context or the requirement specification could not be changed, the architecture is subjected to new design decisions such as the architecture transformation discussed in **section 4.3.3**. One of the method of transformation mentioned was imposing an architectural pattern. This is an excellent example how patterns fit in the software architecture design and bridge the gap from the requirements to design.

Buschmann et al [9] and Bosch [7], present styles and patterns in terms of quality attributes. Those literature depict both, positive and negative quality attribute impact so that patterns alternatives reveal its strengths and weaknesses. Most quality attributes assessments regarding patterns will base on these sources.

Patterns applied late in the development cycle involve more costs. They are used to shape the architecture at the very beginning of the design. Hence, patterns as an approach to designing software architecture should be considered in the first place to find the most suitable architecture.

[3] Jan Bosch in [7] uses the term *architectural style* similarly for *architectural pattern* described by Buschmann et al. in [9].

4.2.3 Why Architectural Patterns?

Buschmann et al. [9] architectural patterns are chosen as they are specified in a context that allows for the practical investigation how quality requirements that impact the software architecture design. They were selected among other pattern categories for following reasons.

A difficult task is to analyze the designed architecture that has not yet been written down. Patterns that comprise the architectural description have a significant impact on the ability to analyze an architecture for certain quality attributes. These pattern are relevant for the analysis, while they provide knowledge for addressing certain quality aspects that include specific quality requirements. In some cases, a specific model is necessary for a set of quality attributes. Software architecture design ensures that requirements, and especially quality requirements, are fulfilled. One way to do that is to start the design process with a pattern that addresses certain quality attributes.

Idioms depict language-specific implementation issues and this paper is an attempt to focus on the software architecture at high abstraction level. That is why they were rejected in first place. The architecture in this research should deal with its structure from the scratch, i.e. from the very beginning when the fundamental decisions are taken. Different from design patterns which address parts of the architecture, i.e. subsystems or components of a system, architectural patterns affect the whole fundamental structure. Architectural patterns represent the highest-level patterns in system hierarchy as they specify the primary organization scheme for the software architecture. As every following development activity is governed by this structure these kind of patterns have the most significant contribution to the shape of architecture. That is why they were chosen as a method for software architecture design in this research. However, software engineers should be familiar in understanding other patterns beyond those categorized as architectural.

4.2.4 Architectural Patterns

An important concept in the area of software architecture are architectural patterns. These, similarly to what Bosch recognizes as *architectural styles* [7], Buschmann et al. describes as architectural patterns due to the fact that these both pattern types have similar, significant usage, i.e. they set the overall system, the fundamental structure. Buschmann et al. [9] presents a description of this concept in a following way:

> *"**Architectural patterns** express fundamental structural organization schemas for software systems. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them"*
>
> [9, p. 25]

Architectural patterns define some overall structuring principles. They capture fundamental, system-wide structural organizations of software systems. Their primary task is to provide descriptions of subsystems, define their responsibilities, and specify how they interact with each other to solve a particular design problem. By dealing with quality attributes an architectural pattern helps to decide if software architecture fulfils quality requirements. An architect should however be familiar with other patterns well beyond those so called architectural. **Figure 9** illustrates architectural patterns and their categories.

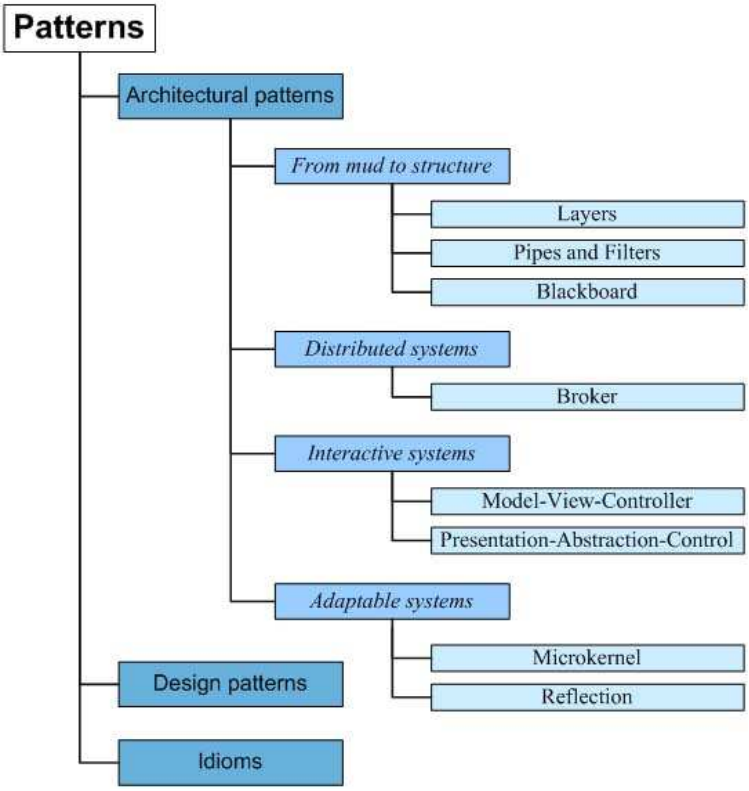

Figure 9 - Buschmann et al. [9] pattern categories and subcategories

4.2.5 Architectural Pattern categories

> *"Different architectural patterns imply different consequences, even if they address the same or very similar problems"* [9, p. 27]

Buschmann et al. [9] depicts eight architectural patterns sorted in related groups: layers, pipes and filters, blackboard, broker, model-view-controller, presentation-abstraction-control, microkernel, and reflection. Each architectural pattern helps to achieve a specific global system properties. Architectural patterns that help to support similar properties occur in categories. These categories are illustrated in **Figure 9** and classified into four categories as follows. Patterns in *from mud to structure* category support a controlled decomposition of an overall system task into cooperating subtasks. *Distributed systems* ensure the complete infrastructure for distributed applications. Patterns from *interactive systems* relate to structures with interaction between humans and the system. Finally, *adaptable systems* support extension and their adaptation to evolving technology of software applications. Some of these patterns belong to more than one problem category. Pipes and filters, for example, can be seen as a pattern to deal with decomposition of a system, and as a pattern to ensure distribution aspects, eventually both.

4.2.6 Summary and remarks

Reuse provides with already gained quality and sometimes even labour savings through architectural reuse in the notion of patterns. *"The most appropriate style for a system depends primarily on its quality requirements"* [7, p.37]. The problem is to evaluate it along

the alternatives to discover which one of them applies best to the requirements provided. Pattern, or a combination of patterns have a tendency to repeat in similar types of applications. That is why it is also worth to look at other software applications which had had similar requirements.

The relative importance of the various quality attributes depends on the nature of the intended system. This paper examines the quality attributes of the ISO/IEC 9126 [20] quality model for quality attributes that are of particular interest to architectural patterns discussed by Buschmann et al. in [9]. Quality attributes should be considered during all phases: design, implementation, and deployment. Not all of them presented in [20] can be addressed by the architectural design. These are mostly developer-oriented quality attributes due to the fact that it can not be measured how architectural patterns influence e.g. usability or attractiveness. It will be clearly stated when and why an architectural pattern has a passive influence on quality attribute(s).

Architectural patterns allow to reason about the high-level design of a system before it is implemented. As they are applied early in the design activity, they represent the first approach in achieving system's quality requirements via architectural means. The choice of an architectural pattern depends primary on the quality attributes that are to cover: *"Another objective of patterns is to build software systems with predictable non-functional properties"* [9, p. 392]. They started becoming an important approach to sharing design knowledge by sharing an architecture description that favours or hinders certain quality attributes.

Mapping quality attributes to architectural patterns is not an easy task and there is no automated way to do this. The solution is to have quality attributes as an input, and as output get a software architecture based on architectural patterns that fulfil quality requirements. This results in an architectural patterns as a method for achieving software quality. This should relatively bridge the gap from the quality requirements to software architecture design.

The purpose of software architecture design is to create a system that meets its functional and quality requirements. The structure the architecture of is much related to what that system has to do. This is the reason why systems with similar requirements have also a common software architecture. This leads to reuse the existing, well-proven design knowledge, and what is important from the point of this thesis:

- document existing software architecture experience to common design problems,
- use similar specification of quality requirements – which structures and to what degrees cover quality attributes.

> *"The earlier in the life cycle reuse is applied,*
> *the greater benefit that can be achieved"* [2]

Reuse is a great *"tool"* for systems with similar requirements, useful to shape patterns, that impacts software architecture design in terms of addressing quality requirements. The chosen topology of components and their relationships (chosen architectural pattern) must conform to those requirements, not the other way around. Therefore, a classification of architectural patterns should be based on the quality attributes induced by those patterns in order to be useful.

> *"Your pattern selection should be further influenced by your application's non-functional requirements, such as changeability or reliability"* [9, p. 27]

Design activity is about having a way of choosing suitable software architecture. Architectural patterns are methods of achieving quality attributes corresponding to quality requirements posed on the system. Different architecture patterns address various quality attributes and to different degrees. Therefore, the role of desired quality requirements has tremendous meaning because it affects the selection of the most appropriate architectural pattern during the design of software architecture.

4.3 Software Architecture Evaluation

4.3.1. Evaluation theory

While quality models may be neglected as a tool for achieving the required levels of quality, there is no doubt about the importance of measuring quality by existing, systematic methods. Although, there are many that can be used to improve the quality of a system, this paper focuses on one, the most important technique, which is *architecture evaluation*. Architecture evaluation is the process of measuring or analyzing how well the architecture addresses quality requirements of the system. The main goal is to check whether the quality attributes of the system meet quality requirements, and specify those that lack. The evaluation improves the potential software quality of the system before it is implemented. It is performed when the design of the architecture is not good enough, i.e. when it does not meet quality requirements. The assessment results are an important feedback to the design process to be able to improve the architecture. A number of methods have been developed to evaluate quality related issues of software architecture at the design level. These methods serve several alternative designs or their variations. Assessment provides the tool for comparing and eliminating design alternatives, thus reducing the potential solution area. In consequence, the design after evaluation should be of higher quality because it improves design knowledge. Design knowledge defines documented and communication proven design solutions to recurring problems, i.e. patterns chosen as a best practise for existing knowledge.

The design process may choose many solution paths. Of course, the design process itself runs more effectively since there are less solutions to be discovered. Concentrating on a system quality attributes, for example, specifies a number of paths and hence – its branches. Each of the required quality attributes probably needs a different solution. This increases the number of potential solutions to choose from. An architect should concurrently consider several patterns in order to select a suitable solution. However, too many design alternatives may cause a negative effect on the process. Hence, software architecture design should be combined with evaluation of its quality requirements as the most optimized tool for solution searching.

Architecture evaluation allows to measure and observe the quality attributes of software architecture design after the design activity has been accomplished or at the specified level for analysis. Assessing quality is important due to the fact that the lack of quality is expensive, and may cause a system to be totally useless.

4.3.2 Aims of assessment

Bosch [7] identified three approaches to software architecture evaluation: *qualitative assessment, quantitative assessment, theoretical maximum or minimum*. The first is used to compare two candidate architectures, which results in an *'boolean'* answer stating which architecture is more suited for investigated quality attribute (e.g. architecture A is more situated for performance than architecture B). The second is used where quality attributes are

given in numbers. Quantitative metrics help to measure, if or to what degree, a system satisfies a quality requirement (response time, maintenance cost, transactions per time unit, etc.). Quantified quality assessment results can be compared to the pre-set quality goals for the overall quality. Quantitative interpretations could be graded, e.g. by the impact of quality attributes on software architecture. Grading can use a scale presenting the impact such as {-2, -1, 0, +1, +2} which determines respectively *high negative*, *negative*, *passive*, *positive*, and *high positive* impact of a software architecture structure on a particular quality attribute. The third goal is about determining the gap between present and the theoretical maximum or minimum level for a certain quality attribute of evaluated architecture.

4.3.3 Techniques for Architectural Assessment

Introduction

The studied techniques for architectural assessment allow to make qualitative and some quantitative statements at certain level of accuracy about quality attributes at the software architecture level. This means that quality attributes that can be addressed by the architectural design can be evaluated or measured against the software architecture [7]. Of course, analysis applies to the system at design level and is limited to those quality attributes that can be verified using an architectural description. This means that not all, but many relevant quality attributes can be evaluated before the application will be implemented.

This part discusses several techniques for software architecture evaluation that were found interesting or useful for the purpose of this research.

Scenario-based assessment

Bass et al. in [2] emphasizes that quality attributes have meaning only within a context and this led to adopt scenarios as they are a way of describing the mentioned quality attribute(s) context. *Scenario* is defined as *"a brief description of a single interaction of a stakeholder with a system"* [2, p. 192]. In other words, a scenario is an instance of a use case (specified set of steps performed by a user on the system). It helps to compare and contrast candidate architectures to validate their required quality. Using scenarios makes quality requirements more concrete to the architect.

A usage scenario (used in a usage profile) focuses on the system under a typical usage. It can be used during the assessment of an operational quality attribute(s). A change scenario (used in a change profile) describes a modification or a general change to a system. Each scenario focuses on one quality attribute, and often several scenarios are created that stresses the same quality attribute, but from different perspectives. Scenarios can be used to evaluate quality attributes such as maintainability, changeability, etc.

Scenario based assessment is a structured and formalized technique to assess design decisions. A great example of its usage provides PerOlof Bengtsson in his *"Architecture-Level Modifiability Analysis"* [4].

Simulation

Scenario based assessment is static in that no executable model is used. *Simulation* uses the software system's context, its environment at a certain abstraction level to execute the model. The system behaviour is used to predict the software quality attributes. Prototypes are similar but in the opposite of simulation they are used to execute an intended part of the architecture. Once, a context and a high-level software architecture implementation is

available, scenarios can be used to evaluate relevant quality attributes. Simulation of the architectural design not only evaluates quality attributes, but also functional aspects of the design [7].

Mathematical modelling

Mathematical modelling is an alternative to simulation because they are both appropriate for dealing with operational quality attributes. Mathematical modelling allows for static evaluation of architectural design models. It makes make predictions about the potential qualities of a resulting product using a variety of metrics. Mathematical models are unique for each quality attribute [7].

Experience-based evaluation

The liability of evaluation methods is that they provide subjective and qualitative assessment. Nowadays the most common way to create an architectural structure is to rely on objective reasoning based on previous experiences and logical argumentation. Experts (experienced software architects) provide their own valuable insights and suggestions that proved to be efficient and help to avoid bad design decisions. This approach is different from previously described in that the evaluation process is less explicit and formal. It is based on subjective factors such as intuition, feelings, and rational thinking. This kind of analysis process usually starts with a feeling that something does or does not fit [7]. “Good” or “bad” designs have a large tendency to bring negative results, but this approach should never be underestimated as it is still the most popular one used in industry.

Experience-based means reuse of existing knowledge. Hence, patterns introduced in **section 2.7** facilitate widespread reuse of software architecture from early phases in the development lifecycle. Patterns help to capture existing, well-proven experience and promote good practices in solving design problems. They are built upon collective experience of software architects and engineers. Best designers do not invent new solutions distinct from existing ones. They rather tend to look at solutions in similar projects and reuse the essence of the previous solutions into the new project [9]. Therefore, patterns are commonly used in software architecture design, not only because of the experience-based knowledge, but also for constructing software architectures that address certain quality requirements.

Metrics

A metric is a “the defined measurement method and the measurement scale” [20, p. 20]. Software metrics are proposed not only during the architecture evaluation, but also during an early architectural design. They intend to measure and assure system’s quality.

Software quality metrics are divided into three categories: *process metrics*, *product metrics*, and *project metrics*. Process metrics present guidance how to improve development and maintenance of a software product. Product metrics are used for describing the characteristics. Project metrics specify the project characteristics and execution.

Metrics can are used to find, measure, and monitor quality attributes that are prone to problems. Metrics serve quantitative interpretations such as number of transactions per time unit. In order to use them, software architecture has to provide an level of details. Otherwise there is no data to perform the measurement on. However, most of the metrics deal with the measurement of already implemented systems rather than based on the results of early development phases.

Though, many methods and external metrics of evaluation exist in the literature, but they tend to focus on one quality attribute and ignore the others. Also, many software quality communities focus on specific quality attributes and analysis techniques. There is a community, for example, that focuses on software system performance software metrics. Again the same example by PerOlof Bengtsson [4], where he concentrates on evaluating the architecture modifiability characteristics. Sometimes, it is not even possible to evaluate a software design to understand a single quality attribute. As it was inducted in **section 3.2.4**, some of them influence each other (positive or negative impact), such as the mentioned modifiability and performance. Hence, even isolating quality for evaluation can be a difficult task.

SAAM

Scenario-Based Architecture Analysis Method (SAAM) [2] specifies how well an architectural design responds to the demands placed on it by a set of scenarios. SAAM relies on a description of candidate architectures that identify relevant components and connections and the overall system behaviour to gain a more complete understanding of them. Competing architectures are compared against each other using similar scenarios with assigned weights of their relative importance. SAAM produces a set of metrics for each scenario. These scenarios are evaluated by investigating which architectural elements are affected by them.

The method consist of six steps:

1. Scenario development.
2. Architecture description.
3. Classification of scenarios.
4. Individual evaluation of indirect scenarios.
5. Assessment of scenario interaction.
6. Overall Evaluation.

ATAM

Architecture Trade-Off Analysis Method (ATAM) [12] developed from SAAM is a comprehensive way to evaluate a software architecture. It reveals how well an architecture satisfies particular quality attribute goals. ATAM explicitly address interactions between multiple quality attributes, and recognizes trade-offs between them. It uses scenarios for that purpose. ATAM requires several different architectural views: dynamic, system, and the source view. ATAM's specialities are: modifiability, performance, availability, and security.

Bosch architecture assessment

Jan Bosch in [7] proposed architecture evaluation as a part of the design process. The method consists of three main phases: functionality-based design, quality attribute assessment, and architecture transformation. This section outlines the two last phases in context of the architecture evaluation. The overall method description is presented in detail in **section 4.4.2**.
Bosch identified the following techniques for assessing quality requirements:

- scenario-based assessment,
- simulation,
- mathematical modelling,

- experience-based evaluation.

Besides these, Bosch also mentions about metrics that are concerned with quantifying various aspects of software. However, most metrics approaches perform measurements on implemented systems instead of software architecture at design level. Each technique may have different assessment goals:

- qualitative (relative) assessment,
- quantitative (absolute) assessment,
- assessment of theoretical maximum or minimum.

The techniques and their approaches have already been defined in detail in previous sections. Once, the quality attributes have been assessed, the outcomes are compared to the expected values in requirements specification. The goal of the quality attribute assessment is to evaluate the potential of a software architecture to ensure the fulfilment of its quality requirements. If one or more quality requirements are unsatisfied, the architecture is transformed to cover the missing requirements. If all estimated quality attributes are satisfactory, the architectural design process is completed. Bosch identified five following transformation methods:

- imposing an architectural style,
- imposing an architectural pattern,
- applying a design pattern,
- converting quality requirements into system's functionality,
- distributing requirements.

Architecture transformations are illustrated in **Figure 10** where they are distinguished by their scope of architecture changes and the transformation type. Transformations change the structure of the architecture; they affect quality attributes, but not the functionality. In consideration of **section 3.2.4** where quality attribute relationships have been investigated, each transformation (so called quality attribute optimizing solution) improving one or more quality attributes may affect others negatively.

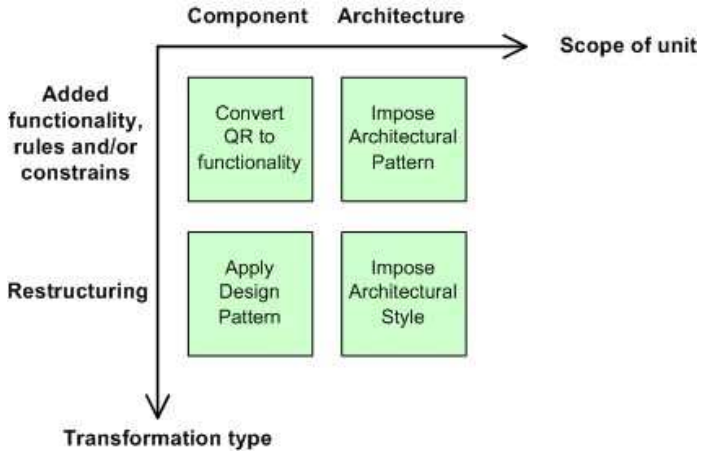

Figure 10 - Architecture transformation categories

4.3.4 Summary and remarks

> *"People often want to analyze software architectures with respect to quality attributes expressed using words such as maintainability, security, performance, reliability, and so forth"* [2, p. 191]

There are methods and external metrics to use depending on what quality attribute is to evaluate. An activity of analyzing a system's architecture in order to understand its quality attributes described by Bass et al. is called the architecture evaluation. In order to check whether software architecture fulfils its requirements, it needs to be assessed. It is impossible however, to verify explicitly if all quality attributes of the final system were addressed based on the design itself.

Evaluation is about how to assess the system in terms of the quality requirements. The basic goal is to evaluate the potential of the candidate architectures (which ones should be rejected), with respect to qualities and their important during the design phase. Candidate architectures in this research, i.e. candidate architectural patterns will be evaluated with respect to a set of desired quality attributes, and to uncover recommendations for best design practices.

Architecture's main goal is the system quality. Therefore it should be analysed and evaluated early, when early design decisions are taken to achieve the system's quality attributes. Evaluation should be performed before the problem arise, i.e. at the very beginning parallel to development process. However, studied software architecture evaluation methods are ought to be performed right after the design was complete [7].

Evaluation, done properly, will definitely lead to increased quality not matter at what stage of architecture design it is performed. However, quality cannot be completely assessed during architectural design. It only *"assesses the ability of the architecture to support the desired qualities"* [2, p. 191]. Software quality cannot be added to the architecture – design process should constantly take into account its importance. Problems found early are easier and less expensive to correct.

When evaluating quality requirements, a general purpose evaluation technique can not be used for all quality attributes. Different quality attributes must be evaluated using different techniques since, as it was presented, techniques tend to specialise on revealing weaknesses of one, or small number of attributes.

One of the greatest problems of software architecture design is the lack of quantitative methods for evaluating quality requirements of proposed designs. Nevertheless, architecture evaluations help to detect and hence, reduce the risk of having a system with insufficient quality. The measurement of quality attributes depicts whether quality requirements will be fulfilled by the architecture or not.

4.4 Quality requirement-oriented design method

4.4.1 Introduction

In recent years, software architecture design and its importance is widely recognized in software engineering. As indicated earlier, a critical issue during the design and construction of a software system is to fulfil its requirements. Software requirements are divided into functional and quality requirements (**section 2.6**). A traditional approach is to develop a software architecture that results from a set of design activities that provides desired functionality. In general, functional requirements determine what the system does. The design activity should however address both of these requirements. A key task that is a difficult challenge for architects is the transformation from quality requirements into software architecture. Despite little progress, this are of the design remains relatively immature.

While the functionality of a system is handled by a number of well-proven development methodologies, the quality objectives still require effective support. Quality requirements in current practice are covered by ad hoc decisions that often rely on experienced-based evaluation (**section 4.3.3**) of architects. This gap intensifies a need for tools and techniques that support a systematic achievement of quality requirements. Since their significant importance is proved in many literature sources (**section 4.1**), the design should handle them in the first place. This part proposes an approach that covers quality-related issues in software architecture design at first.

4.4.2 Bosch design method in context

Quality Attribute-oriented Software ARchitecture design method (QASAR) [7] has already been mentioned in this paper among other evaluation techniques since it enables the software architecture assessment. However, it is also a mature design method with incorporated practise for addressing quality requirements that provided inspiration for the proposed method. Hence, it is presented here.
QASAR consist of three main steps:

- *functional-oriented design*,
- *quality attribute assessment*,
- *architecture transformation*.

It starts with software architecture design based on functional requirements. During this step quality requirements do not receive any particular attention. The next step evaluates quality attributes in the system using a number of scenario profiles either complete or selected. The transformation, being the last step, is performed when the architecture is not good enough – does not meet quality requirements. Transformation, i.e. modification and evaluation, is conducted iteratively with respect to the quality requirements until the evaluation shows that they are satisfactory fulfilled by the proposed architecture. The method do not focus on a single quality attribute, but provides an instrument for the assessment and reasoning about a set of desired attributed. The main disadvantage is that the method deals with functional requirements at first. Large amount of resources have been used without verifying whether the architecture fulfils quality requirements. **Figure 11** presents the QASAR activities.

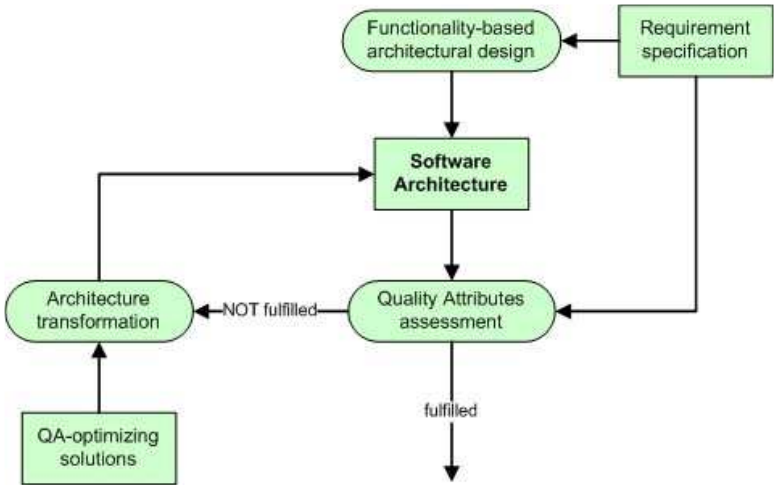

Figure 11 - Quality Attribute-oriented Software ARchitecture design method

As it was indicated in **Chapter Two**, several architecture design methods have been proposed. Examples of these that are discussed in this thesis include the "4+1" view model [22], Hofmeister et al. design method [16], Reference Model for Open Distributed Processing [19] and Bosch design method [7]. The methods differ on where they put their focus during the architectural design. Many of these pay little attention to support explicitly the development towards the quality requirements. These design methods vary with the number of contained activities, notations, design aims and objectives, etc. However, all of them have one thing in common, i.e. requirements specification is used on input and a software architecture is given on output. This means that architectural design can be viewed as a function [7]. Despite this fact, design methods are not an automated process and much effort is involved.

Bosch indicates that several other authors in their conventional object-oriented design methods pay little attention to quality requirements and the architecture focuses on achieving the desired functionality of a system. To sum up, there is a lack in software architecture design methods that which explicitly support the development towards quality requirements. Hence, this quality requirement-oriented design method is proposed.

4.4.3 Method activities

The introduced model is a quality requirement-oriented software architecture design method. It iteratively assesses the degree up to which the provided architecture supports the quality requirements. There are four activities, i.e. quality requirements-oriented design, quality requirements evaluation, functional requirement-oriented design and functional requirements evaluation. The proposed approach uses two artefacts, i.e. requirements specification and the software architecture. It is assumed that the requirements specification is divided into functional and quality requirements, so that each design orientation uses corresponding requirements category. **Figure 12** illustrates the proposed design method.

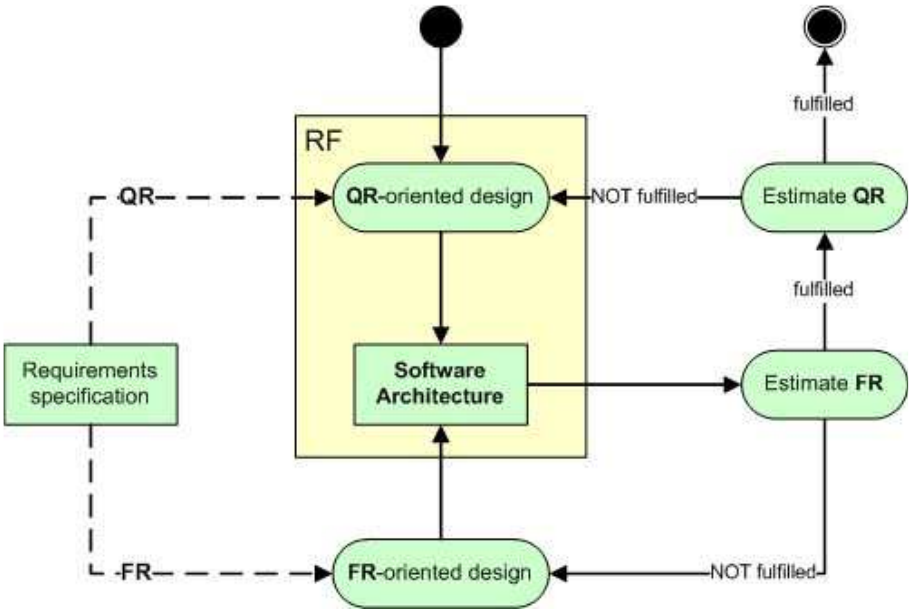

Figure 12 - Quality requirement-oriented design method

The objective of this model is to design a software architecture that targets both requirements types of a system. In opposite QASAR, this design process starts with a design of a preliminary version of the software architecture based on quality requirements specified in requirements specification. Then, the software architecture is evaluated in terms of functional requirements that were not considered in the previous step. The estimates are compared to the expected results from the requirements specification. If these indicate that results are not met by the current architecture, the functional requirement-oriented design is performed until the architecture fulfils its functional requirements. Quality requirements are not considered at this stage. Then, the functionality design is further evaluated, but this time the quality requirements are the subject of assessment. If the outcomes determine that the architecture does not met its required quality requirements, the design towards the missing quality requirements starts iteratively again. Otherwise, the software architecture design is considered as completed. Since the last evaluation considers quality requirements, this ensures that every functional modification of the architecture is evaluated later with respect to the quality requirements. The proposed design method differs also as the QASAR architecture transformation is not a separate activity, but a part of the quality requirement-oriented design.

4.4.4 Method example

The following example is given in order to illustrate the intention and usability of the proposed method. As it is proved in **Chapter Three**, quality requirements are crucial to a software system. Therefore, if possible they should be addressed as early as possible by the design activities – in software architecture design. This research discusses the concept of the high-level design before committing into detailed design. Dealing with quality requirements at the very beginning helps to fulfil several quality attributes, for example maintainability. Maintainability is recognized as the ease to modify, improve, correct, replace or adapt to a changed environment a software system or component. The level of maintainability depends how strong components are connected with each other. Encapsulation of these components is a method of achieving maintainability as they include modularization strategies that reduce

the effort of modifying the system. The idea of the proposed method is to deal with the desired quality attributes of a system at first, i.e. before functional requirements are addressed. Decomposing the system into a certain number of components addresses the quality requirements regarding maintainability, before taking into account the functional requirements. Afterwards, the functionality of a system can be placed on these components that fulfil the desired quality attribute which in this example is maintainability. However, a number of quality attributes can be situated before the functional requirements are handled such as:

- efficiency – "volume and complexity of intercomponent communication and coordination, especially if the components are physically distributed processes" [2, p. 32]),
- security and safety – determine a component for system authorization, error checking, data encryption, etc.),
- reliability – component(s) replication, implies redundancy strategies),
- portability – similarly to maintainability – encapsulate a component that can be replaced in order to migrate to a different environment),
- flexibility – making the system configurable
- fault tolerance – component(s) for exception handling.

In general, these quality requirements strategies involve of ensuring the existence or a specified order (decomposition) of one or more components (mechanisms) that fulfil desired attributes of a system. It is also worth to mention that all of these abstract from the system functional requirements. Therefore, the proposed method find its usage in software architecture design activities.

4.4.5 Benefits and liabilities

This section serves a rationale on benefits and liabilities of both Bosch and proposed design methods regarding relevant architectural issues.

Bosch himself describes the two main liabilities of his approach. These liabilities also concern the proposed approach. They are listed to present similarities with Bosch method and to realize the required model rework. Firstly, not all quality attributes can be evaluated until the system is put in operation. Secondly, if these attributes (categorised as operational) are measured when the design is completed, much rework has to be done in order to include the modifications. Incorporating these modifications result generally in total resource usage. From personal findings, the changes may also not bring expected outcomes, and even decrease the level of achieving quality attributes. As it was indicated in **section 3.2.4**, attributes may influence each other positively or negatively. Though, implementing modifications to a quality attribute may hinder others. Even if the fulfilment of an attribute is achieved, such procedure may result in total lower software quality.

Moreover, Bosch indicates that having a functionality-based design may result in needless design efforts as the system was directed towards functional requirements and lacks in quality requirements. Bosch method is more modification-prone since in most cases the first iteration requires transformation. Therefore, the proposed method benefits from the start, even when there is no conception of exactly which functional requirements should be supported and in which way. The early quality requirements-oriented design allows for the achievement of global quality attributes that constrain the architecture on the highest abstraction level. This design in the first step ought to decompose the system into top-level components similarly to Bosch architectural styles [7] and Buschmann et al. [9] architectural patterns. These styles and patterns are used to deal with a structure of an architecture from the

scratch, when the fundamental decisions are taken. Since they specify the primary organization scheme for an software architecture, the quality requirements can already be addressed, even before the functionality. This is significant because it allows to:
a) save much effort on and reduce the number of quality requirement evaluations,
b) avoid modifications caused by the lack of quality requirements during the functionality-oriented design, especially during the first iteration.

QASAR does not consider a situation when the architectural transformation affects functional requirements. Bosch says that the functionality is stable so that the idea behind the transformation is that a system has exact functions before and after a transformation. The differences occur only in quality properties of an architecture. However, incorporating quality requirements may influence negatively the specification of functional requirements. Also, functional requirements may have to be sacrificed in order to meet the quality requirements, especially when these quality requirements impact the whole architecture. Even though a quality attribute estimation resulted in a transformation required to fulfil the missing quality requirements, QASAR considers the functional requirements to be covered. The proposed approach takes into account such a possibility.

The proposed approach also considers an activity which evaluates the functionality-oriented design. It estimates whether and to what extent the functional requirements are covered, i.e. are the required system functions, services available. Such procedure allows for monitoring the development progress and handles the situation of functional requirements variability. Moreover, if a close collaboration between the two design types is considered, functional requirement-oriented design may depend on quality requirement-oriented design. This means the iterations of functional requirement-oriented may include a design that addresses a part of the functionality. Then, this activity is intermitted to fulfil the quality requirements of the obtained functionality part and to ensure some quality-related background issues for further functionality-based design.

The model also benefits in having similar development progress towards both types of requirements. Bosch concentrating on functionality leaves quality requirements completely aside. The proposed method balances more or less the achievement of both requirements at a time. This may help to control the progress and to improve the workflow understanding.

4.4.6 Summary and remarks

The challenge in software architecture design is to develop a system that fulfils its requirements. Traditionally, design methods concentrate on the desired functionality which alone is not sufficient to achieve the right quality level. It is therefore necessary to get an early quality requirements achievement of the resulting software. Bosch's approach is a good example of a design method that covers quality related issues, but the functional requirements are considered at first. Maybe an approach of dealing with quality requirements from the very beginning would in consequence result in higher software quality levels compared to the resources used in architectural design. The proposed model serves such a solution.

It is not the subject of the presented model to assess whether the existing design methods are better or worse, but to consider a possible software architecture design guided at first towards quality requirements. As the proposed model attempts to bring quality requirements closer to software architecture, it serves a solution to the research investigated in this thesis.

The proposed model activities are left to the architect to be conducted in a fashion that seems appropriate. It only emphasizes the order of design activities which contribute to the fulfilment requirements categories. The main challenge facing the proposed model was to

find an optimal balance among typical design activities in order to make a software architecture address at best its quality requirements.

The proposed design method similarly to QASAR can be seen as a function. It takes a requirements specification as an argument and results in a software architecture. Neither the proposed or QASAR design method is an automated process. They both require much efforts and creativity from the architects. To summarise, there is a lack in support for design towards quality requirements. The following chapters present the recommendation framework, i.e. an automated approach to fulfilling quality requirement via architectural means.

Chapter Five – Empirical approach to Recommendation Framework preparation

5.1 Study design

5.1.1 Empirical research

This chapter focuses the methodology used in this research. Since one of this thesis objectives is to improve the way that quality requirements are handled in software architecture design, an empirical method seems to be the a suitable solution. It is chosen for achieving this goal due to the fact that it reveals how others deal with such problem in practice. To gain more knowledge from society, there have been developed a variety of methods of that enable information collection and analysis. There are two commonly used aims of assessment described in **section 2.5.2**. differentiated on the research result type: qualitative and quantitative. Quantitative assessment requires a construction of a set of numerical information on large groups of people through questions with a suitable scale for data measurement. The subject matter of a research and its expected results influence the method chosen for the assessment. Particular methods are often associated with particular research types. However, all the techniques do not support an rational scale. Hence, their results remain subjective. The background knowledge presented in **Chapter Two – Four** is introduced according to a number of literature sources that investigated the role of quality requirements in software architecture design and related issues. However, the research in this thesis is also conducted in an empirical manner as it uses questionnaire for observing and gaining required data for the following chapters of this research.

This research consists of seven main steps listed as follows:

1. Identify the research problem and purpose.
2. Design a questionnaire.
3. Collect the research data.
4. Analyze and interpret the results; make conclusions.
5. Specify recommendation framework concepts; indicate the usage steps.
6. Present a usage example.
7. Verify the validity of the proposed solution.

Chapter One determined in detail the first step of this research. Step two is described in **section 5.1.3**, whereas analysis and results presented in **section 5.2** are responsible for the fourth step. The recommendation framework, its relevant context, and a usage model are pinpointed in **Chapter Six**. The last two research steps are presented in **Chapter Seven,** where a practical usage example and its further validation are described.
Moreover, **Chapter Eight** summarizes the research outcomes and draws conclusions. A possible future work is listed in terms of the given results.

5.1.2 Aims and objectives

The study presented in this thesis is carried out to gather the required data for the proposed recommendation framework. There are two main literature concepts used by the framework: *architectural patterns* and *quality attributes*. Although architectural patterns are described by Buschmann et al. [9] and some of these by Bosch [7] is such way that their influence on several quality attributes is investigated, but it does not provide enough data to build the framework. Firstly, the patterns impact on quality attributes is presented in a qualitative way. Therefore, the relationships between them could not be compared equally. This makes the impact immeasurable among the patterns and even among the attributes. Secondly, not all of the patterns pinpoint the influence on the same attributes. The literature investigates certain quality attributes for each of the patterns leaving others unmentioned. Thirdly, neither Bosch [7] nor Buschmann et al. [9] use the same quality model (ISO/IEC 9126 [20]) for specifying quality attributes. Though, a number of translations or assumptions had to be made. These reasons prove why the qualitative data from the literature cannot be used by the framework.

Questionnaires are commonly used since they allow to reach a large number of people who could be interviewed. In the case of this research, it is achieved via e-mail since sending out a questionnaire is an easy way of finding persons that would potentially participate. Having a large sample of people involved and the results generalization ensure the questionnaire's validity. The results rely on experience-based evaluation (introduced in **section 2.5.3**) of the interviewed persons. Questionnaire outcomes are easily exposed for analysis as they contain fixed questions that provide quantitative answers. This is why a empirical questionnaire technique is chosen for gathering the required data for the proposed recommendation framework. It ensures that answers coming from different background and approach of interviewees, and their further generalization enable the frameworks general usage applicability and validity.

To sum up, the objective of this study is to collect and prepare the data required to build the recommendation framework by examining students and experts' viewpoints and priorities with respect to their knowledge and experience. As an instrument for examining interviewed persons' viewpoints and priorities a questionnaire with quantitative questions is prepared that concerns the impact of each architectural pattern introduced by Buschmann et al. [9] on ISO/IEC 9126 [20] quality attributes.

5.1.3 Questionnaire design

This section describes the outline of the questionnaire. The designed questionnaire consist of three parts. **Part A**, right after a sort questionnaire introduction lists the benefits of answering the questions in order to encourage people in participation. By using a questionnaire technique, additional information regarding interviewees' current knowledge can be gained. Hence, the second part (**Part B**) serves a pre-study – an introduction to the interviewed person. It collects background information about whether he/she has any, or in what degree, knowledge and previous experience in the area of software architecture, quality requirements, and architectural patterns. **Part B** was added to the questionnaire due to the threats to the validity and reliability of the study. In order to avoid that, if someone marks the answers that prove his/hers unawareness to the subject, these answers will not be taken into account in the final study results.

Part C starts with a short introduction followed by eight questions. Each of these main questions contain six subquestions. The introduction of **Part C** serves a reminder of two main concepts used in this part, i.e. architectural patterns by Buschmann et al. [9] and quality

attributes categorised in ISO/IEC 9126 [20]. This procedure helps to clarify what the questions are about so that every interviewed person shares the same meaning and understands them properly without any ambiguities. It also provided a pre-stated taxonomy for the used terms. These "reminder" materials were prepared due to the fact that before the questionnaire was really distributed to respondents, a pilot study was carried out to verify whether questions used in the final version of the questionnaire will be capable of being clearly answered. This treatment prepared the questions to be more understandable as an interviewed person can take a look the definitions before an answer is given.

Part C contains eight main questions due to the fact that Buschmann et al. [9] classified eight patterns as architectural. They were chosen as software architecture descriptions in this research for reasons which were presented in section **4.2.2** and **4.2.3**. The first of total six subquestions in each of the eight architectural patterns questions is concerned with an interviewed person's knowledge about the pattern. A five-level scale is given to measure the degree how a recipient is familiar with a particular pattern. The following five subquestions contain five ISO/IEC 9126 [20] characteristics (except functionality) and their equivalent subcharacteristics. Similarly to the first subquestion, the remaining subquestions are represented by a five-level scale, although this one investigates how every architectural pattern impacts each of the ISO/IEC 9126 [20] quality model characteristics. A legend for this purpose is given in the introduction to **Part C** and it indicates the specification of possible choices.

A scale with greater number of possible options would increase the study validity and in consequence the recommendation framework's correctness in finding the most suitable architectural pattern. However, a five-level scale seemed to be the most appropriate as it depicts reasonable dependencies between quality attributes concerning an architectural pattern. The more answers a question has, the more difficult it is. A possible wider scope of choices could have been prepared, but it would discourage participants and increase unnecessarily the difficulty of the questionnaire, and therefore making its final results useless for the research. Nevertheless, it limits the opportunity for more precise statistical analysis.

See the **Appendix 2** for the attached questionnaire.

5.1.4 Summary and remarks

A research methodology is chosen approximately for the purpose of this thesis. This study addresses empirically a few research question stated in the **Chapter One**. A questionnaire technique chosen for the reasons described in the introduction of this chapter helps to collect all the data required for further analysis and considerations. In general, it gathers the information on how the idea of quality requirements is recognized in practise. Problems are identified by observing and measuring how organizations develop software with respect to quality requirements. The domain of this research investigates the concepts related to: quality requirements, quality attributes, software architecture, architectural design and assessment, and patterns. The results may in consequence help to improve the way these concerns are handled in practical software engineering.

A questionnaire is prepared as the data collection instrument as it is the easiest way a large number of participants with different background experience can be gathered and brought under analysis. This will help in research results generalization as they come from different sources, i.e. from people with different approaches towards dealing with quality related issues in software architecture design.

Although some knowledge is mandatory for the validity of the results, the questionnaire itself is designed as easy as possible so that every interviewed person is willing to answer the questions and generally finds it simple to handle.

The preparation for the questionnaire is minimal. It depends mainly on booking time with participants and preparing printing materials (questionnaires, ISO/IEC 9126 [20] quality attributes glossary, and basic Buschmann et al. [9] architectural patterns description, given as reminders). The material (originally prepared in English) was not translated into different languages, although interviewed persons have different nationalities (Polish, Swedish, and others), and some of them may be unable to understand the questionnaire and the provided "reminder" part completely.

Conducting an empirical research based on personal experiences and general observations brings threats to the results validity. Despite the fact tat a number of preventing steps are taken to decrease them, they are almost impossible to be completely avoided. Various means have been applied in order to increase the questionnaire's reliability for the research. These methods are refined as a result of feedback from the pilot material given to several students of the fifth year of computer science (software engineering specialization) at the Wroclaw University of Technology. In is not the intention of this section to depict the validity threats since the following chapter describes not only the potential ones, but also those that could not have been avoided during the design of the study.

5.2 Analysis and results

5.2.1 Introduction

This part presents the research results and findings. It was aimed to understand the role of quality requirements in software architecture structures. Data was collected by means of a quantitative questionnaire. The participants were given as much time as they need to fill out and return the questionnaire with answers. The questionnaire will be analysed in order to present statistical outcomes. Finally, important findings during the study will be described.

5.2.2 Research domain

The research presented in this thesis was carried out among two groups of interviewed persons. The questionnaire was given to:

- students of the fifth (last) year of computer science, software engineering specialization at the Wroclaw University of Technology,
- Experts from industry with background knowledge about practical software engineering and application design.

The age of the target groups of people is not relevant. The knowledge and experience are the significant factors that matter. Since the questionnaire respondents have significant approach differences, their results are obvious to differ from each other. Students have academic knowledge on architectural patterns gained from the mandatory courses attended at the University. Hence, their knowledge is rather theoretical since the lack of practical industry experience. Different from students, the group of experts who participated in the research consists of people working in large industry companies:

- Advanced Digital Broadcast (www.adbglobal.com),
- Comarch (www.comarch.com),
- Osmosys Technologies (www.osmosys.tv),
- Siemens (www.siemens.pl),
- Silicon & Software Systems (www.s3group.com),
- SMTSoftware (www.smtsoftware.com),
- TETA (www.teta.com.pl).

Their answers were based on experience gained during practical large-scale projects development. In consequence, their answers should ensure the final results reliability. However, investigating students should also bring positive results as they are familiar with information from the literature. Nevertheless, the questionnaire outcomes of these two interviewed groups are be presented separately and in total. Hence, interesting conclusions are described.

High sampling is important for validity and allows for results generalization. The sample size is 13 experts and 38 students, giving in total 51 questionnaires that were collected, but 43 in total were used for measurement as the **Part B** proved that the interviewed persons were not reliable to take them into account.

5.2.3 Questionnaire results

Before a statistical analysis is given, this section presents a general discussion on the questionnaire results.

Some of the respondents did not answer all the questions provided. These questionnaires are not disqualified to keep as many data sources as possible. Missing answers are not taken into account; only complete cases are used as the basis for result analysis.

In many cases students tend to answer in the scope of [-1, 0, +1] close to zero which corresponds to passive impact. These results suggest as if students are either not sure about the answer (lack of knowledge) or the answers are "protectively" marked. This is observed not only when the question about familiarity with a pattern proved that their knowledge here is rather weak, but also when an option corresponding to high familiarity is chosen. On the other hand, experts are not as "shy" as students. Their choices tend to be rather "courageous" as they often mark answers from the whole available scope [-2, -1, 0, +1, +2]. This may slightly suggest that experts were surer in their choices, and their answers should be more correct compared to students.

5.2.4 Data analysis

This section investigates the questionnaire results. The results involve subjects that their knowledge proved applicability for the research. Collected data was analyzed through statistical analysis and involved some interpretations.

- **Part B results:**

Questions one and two proved that all interviewed persons taken into account for analysis participated in software architecture design. Though, several questionnaires that were rejected indicated that students have not taken part of an architectural design yet. **Table 4** depicts the total number of subjects with their average of attended designs. This points the first difference between the interviewed groups, where students and experts differ from each other with the number of architectural designs.

Subject	Participants	Number of designs	Designs per subject
Students	31	74	2.39
Experts	12	104	8.67
Total	43	178	4.14

Table 4 - Participation in software architecture designs

Also, respondents were asked to grade their knowledge about quality requirements, and patterns in general. While the previous table served concrete values for measuring the level of experience, **Table 5** calculates a subjective assessment of respondents knowledge about the basic terms used further in the questionnaire. Nevertheless, it also proves the predominance of experts in this research.

Subject	QRs	Patterns
Students	5.29	6.74
Experts	7.33	8.17
Total	5.86	7.14

Table 5 - Average knowledge of quality requirements and patterns

Part B was prepared to verify whether interviewed persons have some background knowledge to use their answers as a source of information. Section x. indicated that several respondents were excluded for not being applicable for this research. Moreover, the comparison of students and expert's values indicate the differences between academic and industrial knowledge in the area of quality requirements and software architecture. However, these results are subjective interpretations. Moreover, the student group consists of total 31 interviewed persons. High sampling allows for results generalization and therefore answers given by students in Part C will also be used for the purpose of this research.

- **Part C results:**

Respondents were asked to grade their familiarity with each architectural pattern before the answers concerning the quality attributes impact were stated. In some cases, the relationships between a pattern and quality attributes were identified, but the familiarity question was left unmarked. **Table 6** presents the summary results for all architectural patterns, where they are sorted with respect to their total familiarity. Whereas a green box indicates the highest value, the lowest value within a group of subjects is represented in yellow. High accuracy of results is expected among patterns with high familiarity level as they are commonly known and recognized by respondents. Familiarity results also determine pattern usage popularity. Hence, these results may be used as answers for questions about usability since popularity as general applicability suggests pattern usability.

The pattern familiarity may be multiplied by the values of the relationship between a certain pattern and its quality attributes relationships values. Such procedure ought to increase the results reliability since dominant values are given by subjects with high familiarity. For example, a subject's familiarity with a layered pattern is '4' corresponding to 'good' knowledge. Then, each relationship value of every quality attribute in multiplied by '4' so that if someone has much knowledge about a pattern, his answers will be significant. Nevertheless, this procedure was not performed as it would bring even more subjective outcomes.

Pattern	Students	Experts	Total
MVC	4.36	3.89	4.23
Layers	4.13	4.15	4.14
Pipes and filters	3.85	3.93	3.88
Microkernel	3.35	4.20	3.82
PAC	3.74	3.67	3.71
Broker	3.41	3.72	3.52
Blackboard	3.22	3.76	3.46
Reflection	2.38	3.45	2.94

Table 6 - Subjects familiarity with architectural patters

The following **Table 8** presents the final research results, i.e. data that is further used by the recommendation framework. These results are a total average of students and experts answers. Relationships between architectural patterns and quality attributes were calculated as explained in the following example:

36 persons identified the relationship of a **layered** architectural pattern and **maturity** quality attribute. These answers are:

Answer	Subjects per answer
-2	6
-1	12
0	12
+1	5
+2	1

= 36 persons in total

$$\text{Layers}_{[\text{Maturity}]} = \frac{(-2)*6+(-1)*12+0*12+1*5+2*1}{36} \approx -0.47$$

Naturally, provided answers were within the scope of $-2 \le x \le 2$, where **x** is the given answer. Although, these results could have been divided by 2 for facilitation, the original values are presented since it enables a comparison to the answers proposed in the questionnaire.

5.2.5 Validity and threats

Limitations determine potential weaknesses of this research and decrease results validity. These are as follows:

- Small sample size:
  - Lack of practical knowledge from the student community.
  - Small number of experts involved.
- Small assessment scale due to the number of patterns (8) and attributes (16). Hence, 48 assessments had to be conducted.

In this section, threats to the validity of the study are presented together with steps taken to avoid them. During an empirical research the concern of validity and threats is of

high importance and needs proper consideration. In order to avoid threats in the research performed in this thesis, the questionnaire itself was carefully designed thorough precise examination towards the prior objectives for realization of the study. However, precise questionnaire design does not guarantee that all of them will be omitted. Some of them remained, but their scope was reduced due to several efforts described below.

Construct validity, concerned with the design of questionnaire, targets in the lack of exact perception shared among respondents of the terms and definitions used in the questions. For example, as stated in **section 2.3.1** quality requirements are often referred as non-functional requirements, non-behavioural requirements, system properties or constrains. Quality attributes are also recognized under different terms such as qualities, "-ilities", characteristics, or factors. Moreover, patterns categorised as architectural [9] may not be recognized between other types of patterns. This is why descriptions of quality attributes from ISO/IEC 9126 [20] and architectural patterns from Buschmann et al. [9] were included with the questionnaire as an attempt of avoiding this construct validity.

The results can be exposed for generalization as they come from different groups of interviewed people. Students provide the academic knowledge, whereas the experts share their industrial, practical knowledge and experience. These two approaches in the subjected population of data balance the input data so that the total results should be balanced and therefore – reliable.

As it was indicated in **section 5.2.3** there were cases when questions were left unanswered. Participants were told not to mark answers that they have little idea about. This procedure prevents to some extent the fabrication of data due to the fact that if a subject is not reliable in certain questions, these will not be taken into account for analysis. The unmarked questions were not used in final results and their presence had no influence the total validity of the research. However, unanswered fields decrease the particular question's validity since a smaller part of the population was taken into account during generalization.

5.3 Conclusions and findings

The empirical research was carried out to collect the data for the recommendation framework described in detail in the following **Chapter Six**. The questionnaire in this thesis considered two types of people involved: students and experts. Their answers were analyzed in context of both, academia and industry viewpoints. Table x. illustrates the final and expected research results that provide the required data for the proposed framework.

Experts representing the industry viewpoint shared their experience from 7 large companies, where the number of employees goes beyond 350 (in case of Advanced Digital Broadcast). Size is relevant to the study since it determines the scale of company's designs. Also, the number of employees bears on the elaborated and years of experience, and therefore – position in industry. In consequence, the larger company is, the more reliable information should it provide.

Students representing the academic viewpoint have little practical experience and shared their knowledge gathered from courses they attended at university. Nevertheless, both perspectives answers were used to prepare the final results through statistical analyses, which involved some interpretations. These were determined by threats to the study validity caused in general by human estimations. These decrease the accuracy of measured results and therefore the precision of the recommendation framework to identify the most suitable architectural pattern for a set of quality attributes in question.

	Reliability			Usability			Maintainability				Efficiency		Portability			
	Maturity	**Fault tolerance**	**Recoverability**	**Understandability**	**Learnability**	**Operability**	**Analysability**	**Changeability**	**Stability**	**Testability**	**Time behaviour**	**Resource utilisation**	**Adaptability**	**Installability**	**Co-existence**	**Replaceability**
Layers	-0.47	-0.54	-0.62	1.58	1.47	1.24	1.80	1.84	1.63	1.72	-1.67	-1.75	1.64	1.79	1.72	1.65
Pipes & Filters	-1.23	-1.60	0.81	-0.20	-0.21	-0.26	1.75	1.73	1.55	1.71	1.84	1.83	1.47	1.35	1.41	1.50
Blackboard	1.01	1.22	0.69	-0.34	-0.26	-0.31	0.61	-0.24	0.52	0.67	0.06	-0.12	-1.33	-1.33	-1.28	-1.04
Broker	-1.39	-1.45	-1.74	0.12	0.33	0.49	1.30	1.39	1.39	1.25	-0.52	-0.29	1.51	1.42	1.56	1.69
MVC	0.86	0.85	1.06	1.81	1.75	1.68	1.23	1.16	1.20	1.37	-1.49	-1.34	-0.66	-0.97	-0.58	-0.43
PAC	0.69	1.13	0.87	1.26	1.19	1.22	0.99	1.03	1.25	1.08	0.10	0.16	-1.13	-1.24	-1.07	-0.60
Microkernel	1.36	1.43	1.18	-1.50	-1.41	-1.38	0.52	0.29	0.47	0.51	1.27	1.13	1.42	1.67	1.58	1.59
Reflection	0.30	-0.26	-0.47	-1.73	-1.54	-1.73	-0.42	-0.40	0.36	-0.37	-0.53	-1.07	1.38	1.41	1.23	1.17

Table 7 - Empirical research data for the Recommendation Framework

Chapter Six – Recommendation Framework

6.1 Introduction

The objective of the recommendation framework (RF) proposed in this thesis is to enable a quantified understanding of software architecture design in terms of quality requirements that constrain a software system. The proposed method provides automated support for deciding which of the specified architecture structures best suits the quality requirements in question. The context of the method, i.e. ISO/IEC 9126 quality attributes [20] and Buschmann et al. architectural patterns [9], are chosen for the reasons indicated in **sections 3.3.7** and **4.2.3** respectively. **Figure 13** illustrates the framework usage which is a step forward of the **Figure 1** that indicated the main aim of this thesis, i.e. bridge the gap between quality requirements and software architecture. The idea of such architectural design support originates from a lack of assistance during the achievement of quality requirements. **Section 4.4** discusses this gap in software engineering and introduces an approach of quality requirement-oriented support for architectural design. The proposed design model puts pressure on quality requirement-oriented design where the recommendation framework finds its usage. The framework is inspired by the work of Svahnberg et al. [30].

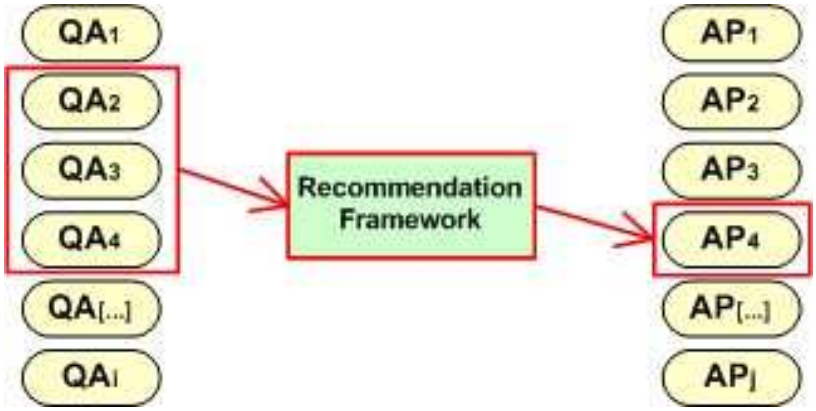

Figure 13 - An illustration of the Recommendation Framework usage

Despite the background that **Chapters Two – Four** provide, the proposed recommendation framework is also introduced in terms of several new aspects which are not mentioned earlier. Hence, the following several sections investigate important concepts that are indicated as a preparation for a clear framework understanding.

6.2 Background philosophy

Software architecture has become an important issue in software engineering. Quality requirements have also received increased attention, but there are still no effective solution how to handle them. Architectural design is traditionally performed as an informal activity. More precise and systematic approach to software architecture design is needed to improve

the architect ability to understand and analyze the design activity regarding the fulfilment of quality requirements. Insufficient research has been done in this area. The recommendation framework is an attempt of bridging the gap between quality requirements and software architecture through the architectural design process. State-of-the-art findings exhibit that Svahnberg and Wohlin [30][31] had done similar research. The work in this research is based on these papers.

One of the techniques in software architecture evaluation is experience-based evaluation described in section **2.5.3**. The concept of patterns is an outcome of the experience of architects and designers where their knowledge, understood intuitively, was documented. Software engineers have tent to reuse the same design solutions over and over again. Certain ways of structuring software elements proved to have good properties which were preferred in choosing certain solutions over alternative candidates. These design solutions were a result of repeated appliance of software element organizations on different levels of abstraction which generally covered different kinds of requirements. As the size and complexity of software systems increases it has become important to have ways of choosing appropriate software architecture structure to fulfil the system requirements. However, quality requirements are not often addressed adequately in existing solutions as functionality is. Architectural patterns serve not only the basic knowledge for good design paradigms but also an effective tool in analyzing quality attributes. By documenting the existing knowledge, discussions, and reasoning how a certain software elements combination influences certain quality attributes, patterns are a verified way in the achievement of quality requirements. This recommendation framework is a systematic approach of fulfilling quality requirements by architectural means with the use of Buschmann et al. [9] architectural patterns.

6.3 Support for design activity

"Design and programming are human activities;
forget that and all is lost"
- Bjarne Stroustrup

Software architecture design and evaluation are typical human activities. It takes a lot of knowledge that mainly comes from the previous experience to manage the whole design process. The activity itself, consist of a number of steps such as collecting, documenting, and managing the information relevant during the design. One of the most important part deals with making trade-offs among solution alternatives. Candidates are investigated carefully in order to choose the most suitable software architecture description for the desired requirements. The task remains difficult as it relies on experience-based estimations (**section 4.3.3**). Human assessment is tendentious, error-prone, and expensive. That is why there is a need of a reliable approach that could manage the architectural design and evaluation. However, software engineering still lacks in such automated procedures. The proposed recommendation framework is an attempt of bridging this gap. It is created to assist the designers fulfil the required quality requirements during the software architecture design.

Section 4.4 introduced a design method that focuses on quality requirements at first instead of functional requirements. It starts with a design of software architecture guided by quality requirements of the system. The model serves as basis for the use of the recommendation framework. Architectural patterns used by the framework are used when the fundamental decisions are taken. They affect the whole fundamental system structure different from other pattern categories described in [9] that address only single parts of the architecture. Therefore, the recommendation framework finds its usage in the first step of the

proposed design method during the quality requirement-oriented design. Since architectural patterns set the fundamental structure of a system, quality requirements can be covered before the functionality is. Based on the preliminary architecture established by the framework, an architect addresses the functional requirements with the assurance that a certain quality level is already achieved.

6.4 Requirements variability and management

In practise, requirements often change during the software development. Kotonya and Sommerville in [12] depict several reasons that cause requirements to change. These include: changes to the system environment, a better (growing) understanding of the customers needs, new requirements appear, and the existing ones need modifications. Maintaining the requirements variability becomes a critical part among other requirement engineering activities. It is therefore worth to mention about the changing requirements in the context of this design support. Failure to control and document the changing requirements may result in poorly specified, or in worst case, inappropriate quality attributes for a system to develop. Kotonya and Sommerville [12] also offer a number of solutions to minimize such difficulties.

The framework does not decrease its usability when the requirements are not static. Therefore, the concept of requirements management is not questioned. However, due to the fact that this research denotes the significant role of quality requirements in software architecture design it is also crucial to focus on effective quality requirements management. After all, requirements definition and analysis is the first stage of system development and should be carried on from the very beginning during the requirements elicitation. Nevertheless, requirements management is not an activity of the architectural design in particular, but the whole software development process. Though, it should rather be conducted in parallel to the design so that the requirements specification is always current. Effective requirements management benefits generally in increased customer satisfaction as the desired requirements are better prepared in the requirements specification used as a basis for further design.

In theory, requirements management is directed towards the maintenance of both software requirements categories. Unfortunately, as it was indicated, quality requirements are often weakly specified or even totally neglected in practice. The functionality is still a predominating issue in software development, also in the context of requirements management.

This section discusses in general the concepts of variability and management in the context of quality requirements. However, the main aim is to emphasize the role of quality requirements as a method for requirements variability. The way the system functional requirements are decomposed into a combination of components leads to high efforts in case of requirements changes [7]. This is because a requirement change may lead to at best a local change in a component, or in worse case – may cause a number of changes in the architecture. A proper architectural design decreases the scale of possible requirements variability. Additional efforts include increased costs, schedule, erosion, resource usage, etc. Maintainability is a quality attribute that affects the fundamental architecture. Having an appropriate, encapsulated structure, the system prevents from additional architecture modifications. Though, it should be relatively possible to incorporate requirements changes without unnecessary changes that impact other architectural parts.

To sum up, requirements management ensures that the input of the proposed framework, i.e. requirements specification, is up to date so that the system (quality) requirements represent the actual customer needs and expectations in such a way, they can be

used for the proposed design activity. In case of a possibility that requirements could change, there are quality requirement methods to prevent the efforts of unnecessary modifications. These methods should be considered by the requirements management activity which ensures the system delivers the expected outcomes, i.e. the solution architecture meets the actual requirements.

6.5 Method activities

The proposed recommendation framework consists of three activities as follows.

1. Identify the required quality attributes.
2. Perform quality attributes prioritization.
3. Calculate the choice of solution architecture.

These are described in detail as follows:

- **Step 1:**

  - **Input:**

    Quality requirements from the requirements specification.

  - **Output:**

    Quality attributes identified from quality requirements.

Chapter Two and **Three** indicated that it is crucial to ensure quality attributes of a system when designing software architecture. Therefore, they need to be clearly specified in order to be properly recognized and addressed by the architecture. The first step of the framework usage is concerned about identifying the quality attributes from quality requirements listed in the requirements specification. Typically, software architecture is likely to ensure the achievement of more than a single quality attribute. In many cases, several attributes are indicated from a single quality requirement. On the opposite, a particular attribute may be specified in a number of quality requirements. In case there are no quality requirements, or the existing ones are badly specified so that even a single attribute cannot be listed, there is no point in using the framework as there is no data given as input.

It is not the intention of this thesis to present guidance how to identify attributes from quality requirements. According to **section 3.2.8** there is no elaborate method of handling these requirements. Therefore, more efforts should be directed towards eliciting, specifying, testing and verifying quality requirements in software engineering. Several concluding remarks about little guidance and lack of methodologies in handling quality requirements are indicated in **Chapter Eight** among other future work objectives. Nevertheless, to emphasize the vague and impressive specification of quality requirements Bosch [7] proposes to define scenarios (see **section 4.3.3** for details).

This step is simple to describe, although not a trivial task in practical software development. The intention is that desired quality attributes of a system are listed so that their notability is understood by architects and they are capable of being used in further steps.

- **Step 2:**

  - **Input:**

    Quality attribute(s) towards which the system is designed.
    that ought to be met by the resulting software architecture.

  - **Output:**

    Quality attributes with relative weights of their importance.

The next step is to prioritize the outcomes of the previous step, i.e. the quality attributes towards which the system is designed. The concept and importance of prioritization has already been described in **section 3.2.6.** Nevertheless, it is worth to discuss why this activity is one of the method's steps. First of all, this step can be excluded and the framework will be still capable of giving design solutions. However, they may not be as precise as in the case of conducting prioritization. In such case, the priority value equals one. Having a set of quality attributes with relative weights of their importance ensures that the most desired attributes will be addressed in the first place. Usually stakeholders are responsible for conducting quality requirements prioritization, but it extremely important for a software architect to establish priorities between specified quality attributes himself because:
a) they often tend to impact each other negatively such as maintainability and performance, security and usability, or security and performance (see **section 3.2.4** for quality attribute relationships);
b) they are fulfilled in the architecture by different amounts of resources (means).
Therefore, it is highly required to perform the prioritization activity in case of these potential conflicts. Relative weights sort quality attributes in order of which they should be taken into account during the software architecture design. Priorities also serve basis when some quality attributes may need to be sacrificed in order to meet project schedule, budget, etc. In case of the recommendation framework, the irrelevant attributes should not be taken into account (excluded) in the usage steps.

Conducting prioritization includes subjective judgement which quality attributes are of higher importance than the others by assigning weights (values). The greater the differences among the values are, the more precise final results are. There is no need for prioritization when a single quality attributes is taken into account for the design. Typically software architecture is likely to ensure the achievement of a set of quality attributes to conduct prioritization on. The number of attributes can be reduced by:
a) grouping them into categories, each representing some aspect of the system requirements;
b) choosing a smaller set of quality attributes to focus on [30].

Prioritizing can be applied by several methods. The prioritization in Analytic Hierarchy Process (AHP for short) as described in [30] is based on pair-wise comparisons meaning that each quality attribute is compared to others. **Figure 4** serves as an example of AHP comparisons where each relationship between attributes is specified. However, AHP uses a wider scale of possible answers – a value in a set in favour of one of the attributes in each of pairs to compare. Svahnberg and Wohlin [31] used a comparison scale answered with a number between 1 to 9. **Figure 14** is an illustration of such approach.

QA_1	-9	-8	-7	-6	-5	-4	-3	-2	-1		+1	+2	+3	+4	+5	+6	+7	+8	+9	**QA_2**

Figure 14 - An example of AHP quality attribute comparison

A vector (called **PQA**) with assigned relative weights of importance for every quality attribute (**PQA_m**, where **m** is a number of an attribute) is a result of the prioritization. This procedure determines a precise evaluation of the priorities and dependencies among quality attributes in question. However, the number of AHP steps and mandatory comparisons increase the method's usage complexity. $m\left(\frac{m-1}{2}\right)$ comparisons have to be conducted, where **m** is the number of quality attributes in question. In case of this research, the total number of characteristics categorised by the ISO/IEC 9216 [20] which are applicable by the recommendation framework is 16. This means that "in worst case" 120 comparisons need to be performed among the set of quality attributes as shown in **Table 8**.

Number of QAs	1	2	3	4	5	10	16
Comparisons	0	1	3	6	10	45	120

Table 8 - AHP comparisons per number of quality attributes

Although, it is not likely that a system must cover all quality attributes investigated by this study, and each comparison is conducted very quickly [30], AHP prioritization would decrease the framework's simplicity and general applicability. Therefore, an easier method is proposed despite the fact there is slightly less chance of choosing the most suitable architectural pattern since more subjective weights (in opposite to AHP) are assigned to the desired quality attributes

Another prioritization technique that can be used is the 100-dolar test. It is a very straightforward method, where 100 imaginary units are distributed between the given quality attributes. The results are specified on a ratio scale with the assigned number of dollars. For example, 50 dollars are given to QA_1, 30 dollars are assigned to QA_2, and the rest goes to QA_3 that summarise to 100 dollars in total. A problem with this technique comparing to AHP is that there is no straightforward dependencies between a pair of attributes. However, the overall viewpoint on a set of quality attributes presents quite well the required order weights of their importance. This research suggest the following, much easier method for quality attributes prioritization similar to 100-dollar test:

1. Choose a scope of possible answers for the assessment.

$$1 \leq PQ_i \leq x$$

where:

PQ_i – is an importance weight of *i*-th quality attribute,
x – is the chosen top integer value of the scope.

2. Assign weights to the quality attributes within the specified scope.

Each quality attribute is given a value from the chosen scope. The wider scope of answers is chosen, the more precise the results should be, as it allows for presenting greater

dependencies among a set of attributes. A typical assessment ten-point scale should be sufficient.

3. Normalise the importance values.

The assigned values are normalised using the following formula to determine the relative "distances" between the adjacent values.

$$QAP_i = \frac{PQ_i}{\sum_{1}^{m} PQ_m}$$

where:

QAP_i – is a normalised weight of *i*-th quality attribute.

Quality Attribute	Weight
QA_1	QAP_1
QA_2	QAP_2
$QA_{[...]}$	$QAP_{[...]}$
QA_m	QAP_m
	Sum: 1

Table 9 - Quality attributes with their weights of importance

Table 9 presents the expected outcome of this step, i.e. a list of the quality attributes identified by the first step containing their normalised weights. To summarise, the purpose of this prioritization activity is to assign values to distinct prioritization quality attributes that allow establishing a relative order between the quality attributes within the desired set.

- **Step 3:**

  - **Input:**

    Quality attributes with their relative weights.

  - **Output:**

    The most suitable architectural pattern representing the best opportunity to deal with the given quality attributes.

This step makes use of the empirical research results described in **Chapter Four**. **Step 3** is the actual "heart" of this proposed design method where the recommendation framework decides on a recommended architecture. The "solution" determines a number from 1 to 8 that corresponds to Buschmann et al. [9] architectural patterns listed in **Table 8**. This table also contains the values required to calculate the results using the following formula:

$$\max_k \left(\sum_1^i \left(QAP_k \bullet QAV_{k,i} \right) \right) \Rightarrow solution = k$$

where:

j – is the number of architectural patterns – eight in total.
$QAV_{k,i}$ – is the i-th quality attribute value of the k-th architectural pattern from Table **8**.

6.6 Benefits and liabilities

The most important idea of the proposed recommendation framework is concerned with the achievement of quality requirements. It serves a systematic approach that addresses quality requirements via architectural means. Therefore, it is an automated assistance for the software architecture design activity and provides a ready-made decision rationale. The solution architectural structures consist of Buschmann et al. patterns [9] that enable a common understanding among architects. Hence, the results are capable of being clearly understood in terms of benefits and liabilities of each architectural pattern with respect to its quality attribute [30]. The use of the framework saves much effort that would have been spent on searching for potential architectural solutions that cover the desired quality requirements. Nevertheless, identified candidates would have been evaluated against each other to choose the best applicable solution. At it is defined in **section 4.3**, software architecture evaluation is a process of assessing whether architecture possesses the desired quality attributes. The recommendation framework usage covers and therefore may even replace the evaluation process. Similarly is in case of design trade-offs (**section 3.2.7**), where quality attributes are compared against each other to a) measure the relationships between attributes as some of them influence each other positively or negatively; b) in case some of the requirements have to be sacrificed in terms of others. The framework determines solutions that should cover the issue of quality attribute trade-offs. In general, benefits of the recommendation framework include savings in areas such as project schedule, labour, resources, and budget funds.

The recommendation framework has also liabilities. First of all, it does not address the desired quality attributes equally, i.e. the method suggest the best opportunity to ensure at best the given combination of quality attributes, so they are not achieved at the same level. Even though the prioritization step was especially added to the method to ensure that most important quality attributes are taken into consideration t first place, the procedure will still cover the required attributes by different means. This generally results in less precise outcomes if a large number of quality attributes is considered.

Moreover, the data required to build the framework is based on the interviewed persons subjective judgements. When applying empirical research the concern of validity and threats is important and needs special attention. For the questionnaire used in this thesis, the threats are mainly caused by small sampling and also a lack of knowledge in the area of this research. Therefore, the results do not ensure a proven level of accuracy.

To summarize:

+ addresses quality requirements
+ automated design support
+ saves much design efforts as it helps to manage the design activity.

+ tool for trade-offs
+ helps to understand the benefits and liabilities of each architectural pattern with respect to its quality attributes [30].
+ tool for evaluation
+ provides understanding for design decisions (rationale) as it uses patterns
– based on subjective judgements
– does not consider relationships between quality attributes
– small probe to provide reliable data that could be used in practise
– the more quality attributes are in question, the less precise the results are

6.7 Quality requirement-oriented and pattern-based design method

6.7.1 Introduction

Buschmann et al. [9] differentiated patters into three categories, i.e. architectural patterns, design patterns, and idioms. These patterns cover various ranges of scale and abstraction.

- **Architectural patterns** (AP) help to decompose a software system into global subsystems. They set the overall structuring principles to express a fundamental structural organization schema for a system. They specify the system-wide structural properties.
- **Design patterns** (DP) support the refinement of these subsystems, or of the relationships between them. They describe a commonly-recurring structure for components communication.
- **Idioms** help in implementing particular components aspects. They specify how to implement an architectural issue using the features of certain programming languages.

This introduction into patterns categories helps to understand the different abstraction level of patterns usage. Hence, they also address quality requirements at various levels. Patterns in general and Buschmann et al. [9] categories have already been described in detail in **section 4.2**. This section indicated certain relevant issues as a starting point for this design method's discussion.

Section 4.4 proposed a quality requirement-oriented design method, where quality requirements are at first taken into account during a software architecture design. The recommendation framework serves assistance for such an approach. It uses ISO/IEC 9126 quality attributes [20] on input, whereas on output it generates patterns which are mentioned above and categorised as architectural. However, it is indicated that these patterns establish only a fundamental architectural structure so quality requirements at top-abstraction level can be fulfilled. Architectural patterns help to address global quality requirements that affects architecture as a whole, e.g. maintainability, reusability, and flexibility is ensured by breaking a system into components. However, this is only a top-level decomposition such as in layered style, where the system in broken into several layers. Such procedure covers in some way quality requirement(s), but does not guarantee they are fulfilled entirely since the structure of lower-level components is not concerned with the exemplary attributes. Moreover, some quality requirements are not met by the top-abstraction level at all. Even

though a system is divided into certain subsystems structure, it is the responsibility of these lower components to capture the intended quality requirements. In consequence, quality requirements are left uncovered. Therefore, architectural patterns are not enough to address entirely the desired quality requirements. Buschmann et al. [9] categorised patterns with respect to levels of range and abstraction. Introduced patterns are proved to have a significant impact on the ability to analyze architecture for certain quality attributes. All these patterns are relevant for the design while they provide the knowledge to address quality requirements. Hence, the design based on architectural patterns, design patterns, and idioms can be combined as they all comprise the architectural description and have a significant impact on the ability to analyze architecture for certain quality attributes.

The proposed quality requirement-oriented and pattern-based design method originates from:

- the concept of quality requirement-oriented design method discussed in **section 4.4**,
- the idea of recommendation framework as a design assistance for fulfilling quality requirements,
- the use of various patterns as means for representing software architecture abstraction levels,
- the lack of a support for achieving quality requirements at all design levels.

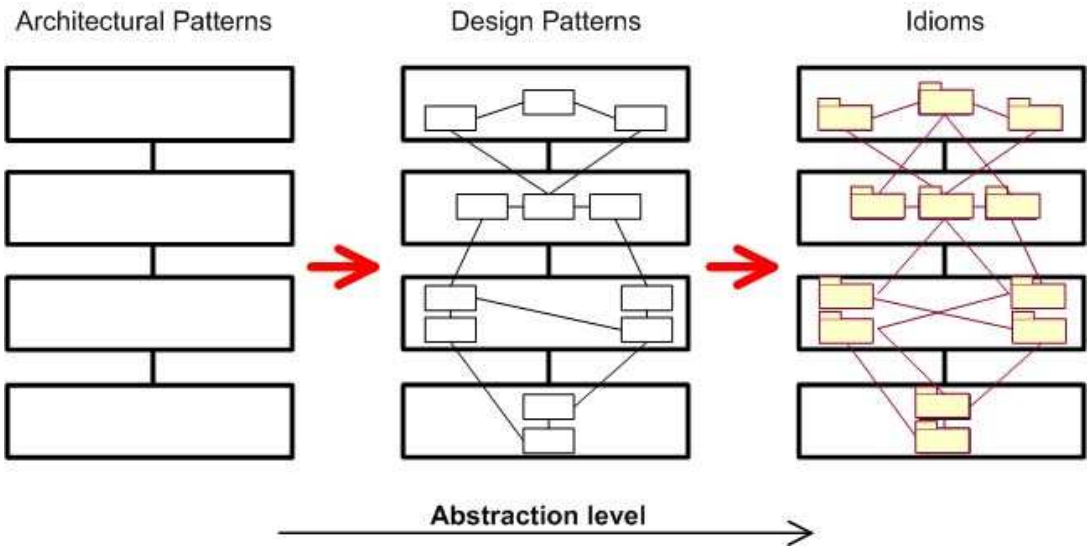

Figure 15 - An illustration of pattern categories at different abstraction levels

6.7.2 Top-down vs. bottom-up design approach

Top-down and bottom-up are strategies of designing software architecture. These are differentiated by the starting point of a design, i.e. a system abstraction level. Bottom-up design starts by defining the low-level components. Then moves up towards more and more complex subsystems using these already defined ones. These specified in detail, individual components are combined together to form larger parts until a complete system if consequently formed. On the other hand, top-down approach forms a fundamental structure without going into details. Then, each decomposed part of the system is refined by designing in more detail. Each created part may be refined again, defining a more detailed structure until the entire system is decomposed in sufficient detail.

Many architectural designs use a mixture of top-down and bottom-up design. Top-down approach is often conducted when the system is designed from the scratch, whereas bottom-up design is performed when a system uses Commercial off-the-shelf (COTS) components cause having predefined parts, a system needs to be designed towards from these parts towards the entire system. Bosch in [7] indicates that from his experience it is not feasible to start from details of a system and therefore recommends a top-down approach.

From the position of this thesis, the choice of design approach direction has no matter as both of these strategies seem to have equal influence during quality requirements fulfilment. Nevertheless, patterns from definition decompose a system into components. This means that are observed on a certain viewpoint and used to specify in detail further components and their responsibilities. Therefore, the proposed design method represents a top-down approach as indicates the abstraction level direction in **Figure 15**.

6.7.3 Method activities

The quality requirement-oriented and pattern-based design method is concerned (as the name suggests) with a design towards quality requirements by the means of patterns. As it is proved in **Chapter Three** quality requirements are hard to manage during an architectural design. The method provides a convenient design mechanism for software architecture description that allows for the fulfilment of quality requirements by applying patterns from [9]. The method starts from the highest abstraction level of patterns presented by Buschmann et al. (see **Figure 15**). The first step is an architectural pattern-based design. It results in a software architecture described in terms of architectural patterns that fulfil global quality requirements. Then the architecture is designed towards design patterns which results in a **refinement of software architecture** that handles particular quality requirements of this 'middle' abstraction level. The case is similar with idioms – architecture is **refined again** according to idiom patterns that fulfil the requirements specific to this 'low-level' design.

Each method part considers different design levels of abstraction. Each of these levels consist of similar steps that include:

1. Identify the problem of a given design.
2. Nominate a pattern that represents a solution of the given problem.
3. Evaluate the consequences of applying the pattern.
4. If the quality attributes in question are fulfilled satisfactory,
   go to a lower abstraction level. Otherwise, repeat these steps.

Identifying a problem is about specifying quality attributes relevant to a particular design level. The activity of nominating a solution pattern is performed to choose the most suitable pattern for the given attributes. The recommendation framework which is pinpointed as a yellow rectangle marked as RF in **Figure 16** represents this procedure in case of architectural patterns at the top-level design. The method is repeated until all the quality requirements have been included satisfactory for the specific architectural design and in conclusion – at all abstraction levels. Different software architectures are a refinement of the 'total' software architecture. It illustrates the same architecture in similar way the <<refinement>> stereotype describes UML elements.

To summarize the method:

- **Input:**
  QRs that ought to be met by the SA.

  - **AP stage outcomes:**
    SA that fulfils the top-design (global) level QRs.
  - **DP stage outcomes:**
    SA that fulfils the middle-design (component) level QRs.
  - **Idioms stage outcomes:**
    SA that fulfils the bottom-design (implementation) level QRs.

- **Output:**
  SA in description of patterns that fulfils all its QRs
  at every design (abstraction) level.

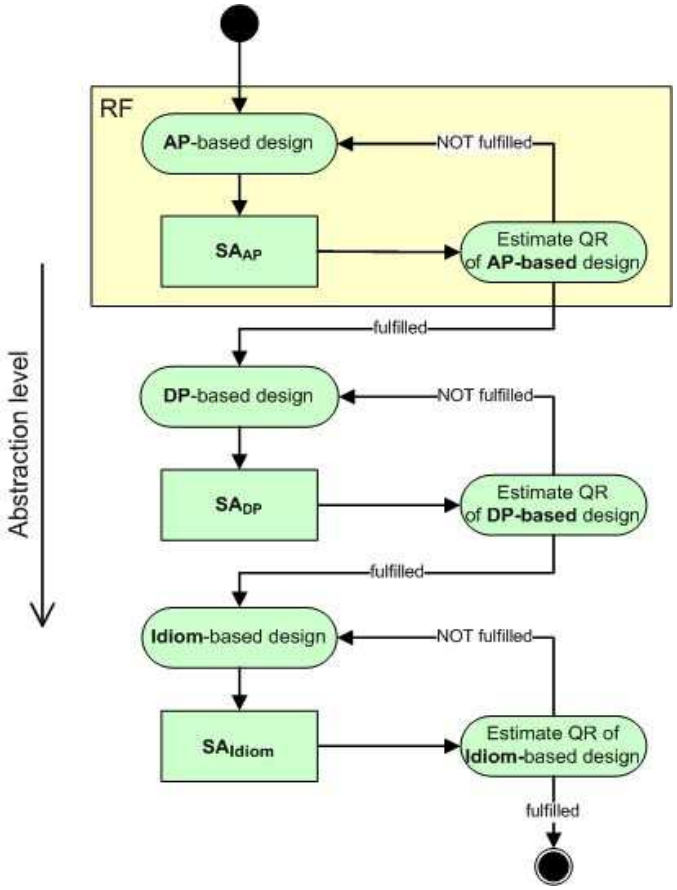

Figure 16 - Quality requirement-oriented and pattern-based design method

6.7.4 Method summary and conclusions

Due to the size and the complexity of most nowadays software applications, a single pattern usage in not enough. A number of or a combination of several pattern is required. Patterns are concerned with various ranges of scale and abstraction. They can be applied in

different stages of software architecture design as they help to address a variety of different design problems. From the perspective of this thesis aims – to address a variety of different quality requirements. Pattern appliance at different abstraction (design) levels fulfils quality requirements specific to these levels. *"The benefits of a set of related patterns is more than the sum of benefits of each individual pattern in a set"* [9, p. 381].

In other words, a combination of patterns may bring positive results as they exhibit different relationships with each other. To take advantage of such sets, patterns need to be organized into pattern systems, i.e. uniformly methods to handle a significant number of patterns in a convenient way [9]. Hence, the proposed design method could find its usage when a predefined assistance provided a search strategy to support such architectural design with patterns. Pattern systems ensure the guidance to support the software architecture design based on patterns. However, such method itself does not guarantee the fulfilment of quality requirements. Therefore, a tool for pattern selection is required and the recommendation framework for architectural patterns (marked as a yellow rectangle called RF in **Figure 16**) determines the first step in the proposed approach.

6.8 Summary and remarks

As it was indicated in **section 2.5**, software architecture is expressed with a combination of structural views of a system, where each view a represents a certain abstraction of the system with respect to different criteria. Software architecture should describe a high-level view of a system structure and therefore - abstract from implementation details. One of the main reasons (among other listed in **section 4.2.3**) why architectural patterns are chosen for the recommendation framework is that they decompose a software system into global subsystems representing the fundamental organization scheme of a system. However, each architectural view is a specific abstraction of the architecture, for a specific purpose. Different patterns denote different dependencies between components and structural properties. Therefore, software architectures cannot be based on a single architectural pattern. Often architecture must address a number of quality requirements that different patterns at different abstraction levels support. An architect has to combine several patterns to form a structure that covers all of the desired quality requirements. Therefore, the recommendation framework is only the first step in the quality requirement-oriented and pattern-based design method which satisfy all quality requirements at all relevant abstraction levels. Of course, the choice of architectural pattern(s) is a major, but an architect has to be also familiar with patterns far beyond these categorised as architectural. Buschmann et al. [9] is a good selection of patterns that presents issues and alerts of quality properties they affect. Important is also that these patterns are identified with respect to abstraction level they concern. Therefore, they are used in this research. Also, based on [9] **Chapter Seven** attempts to prove the recommendation framework validity.

Chapter Seven – Usage examples and validity

7.1 Introduction

The recommendation framework introduced in the previous chapter is worth nothing until its validity is proven. The validity in this research is concerned with the recommendation framework ability to provide expected results, i.e. choose the most suitable architectural pattern for a given set of quality attributes identified from quality requirements.

The framework results validity in terms of the interviewed persons subjective judgements is discussed in section **5.2.6**. Threats to the validity caused by conducting an empirical research is not the issue of this part. The point is to compare whether the literature sources reflect the outcomes of the framework.

One way to verify the validity is conduct comparisons with the literature. The literature focuses to some extent on architecture structures analysis against quality requirements these architectures support. A similar research has been done by Svahnberg and Wohlin [29] in which the literature is compared against the quantitative data from a similar empirical research [31]. The research conducted in [31] will be also compared as a good source for quantitative information. The literature view is based on a) Buschmann et al. [9] as the architectural patterns originate from this book and b) Bosch [7]. These are selected due to the fact the presented architectural structures are described in terms of their benefits and liabilities to fulfil quality attributes. However, there are some limitations of such comparison. These are as follows:

- the literature provides qualitative, whereas the framework quantitative information on architectural patterns,
- the RF uses 16 quality attributes for describing patterns, whereas the literature is concerned with only several that seemed important for a particular pattern,
- neither Buschmann et al. [9] nor Bosch [7] use the same categorisation of quality attributes investigated in this research.

These limitations decrease the potential ability to make the comparisons. Therefore, the recommendation framework results cannot be proved to be entirely reliable. The quantitative data of the framework is compared to the qualitative information found in the literature. This enables to verify only a certain extent to which the framework values (see **Table 8**) exhibit the information the literature provides. Therefore, some interpretations and translations have to be made in order to verify the validity.

7.2 Interpretations

Limitations determine potential weaknesses of the comparison. This section specifies the interpretations and translations have to be made in order to compare the qualitative information from the literature (Buschmann et al. [9] and Bosch [7]) with the quantitative data from the recommendation framework.

All architectural patterns used by recommendation framework are investigated in [9] as they were used from this selection. However, Bosch [7] mentions only about Layers, Pipes and filters, and Blackboard. Therefore, the comparison is based mainly on these.

The literature uses different taxonomy and categorisation of quality attributes. Neither Buschmann et al. [9] nor Bosch [7] use ISO/IEC 9126 [20] quality model for identifying quality attributes of patterns as it is in case of the recommendation framework. Therefore, in order to establish a common vocabulary it is assumed that performance characteristic in [9] and [7] is used in common with efficiency [20]. Also, flexibility and reusability are recognized as maintainability. However, one should keep in mind that performance is also observed as not only efficiency, but also reliability.

To complicate things even further, the empirical research was conducted to obtain the data required for a detailed design, i.e. to ensure the ability of addressing quality each subcharacteristic in sufficient detail. The literature considers whether a pattern influence positively or negatively a top-characteristic. Moreover, in some cases these are even unmentioned. To enable the comparison between the literature qualitative information, the following translations have to be made:

- value of a characteristic (from **Table 7**) is equal to the sum of its sub-characteristics,
- unmentioned characteristics or these identified in some cases as passive correspond to zero value.

7.3 Usage example

The main purpose of this chapter is to verify whether the framework is capable of choosing best (optimized) solution for a given set of quality attributes. However, it is also worth to demonstrate an example how to manage the steps involved in this design support. Therefore, besides the discussion on validity, some example calculations are presented at first in order to illustrate the method. Then, these resulting architectures are compared to what literature suggests in similar designs. The following examples take into account limitations and assumptions discussed in the previous section.

➢ **Step 1:**

Besides the functional requirements, the requirements specification consists of the quality requirements such as:

1. The system shall be capable of incorporating new requirements.
2. The system shall be adaptable to other environments.
3. The system shall be able to give a response no later than X seconds.
4. The system shall have less than Y hours downtime per Z months.

These quality requirements identify *maintainability*, *portability*, *efficiency*, and *reliability* respectively. Quality requirements put constraints on quality attributes. They are usually specific values, a scope, or ranges of values for quality attributes. Some example constraint values that may be placed on these requirements:

- number of required modifications to incorporate new system feature,
- total allowable time needed to switch the system into a different environment.

Section 3.3.6 presents examples of metrics that are used to constrain quality attributes.

The identification process is relevant for the software architecture design as it allows for concrete and precise definition of quality attributes. It is not the subject of this thesis to

investigate this activity. Also, for the purpose of this validity example, quality attributes are predefined; chosen as a starting point for further discussion. Their exemplary specification aims to present all required steps involved in this method and hence – ensure a good method understanding.
In order to illustrate the recommendation framework usage, there are two examples that are considered: **System A** and **System B**. The following table identifies their quality attributes from two requirements specifications.

System A	System B
• Usability • Maintainability • Portability	• Efficiency • Maintainability

➢ **Step 2:**

The second step conducts quality attributes prioritization.

1. Choose a scope of possible answers for the assessment.

.**System A**: $1 \leq PQ_i \leq 6$ **System B**: $1 \leq PQ_i \leq 6$

where:

PQ_i – is an importance weight of *i*-th quality attribute.

2. Assign weights to the quality attributes within the specified scope.

System A	System B
Usability **PQ_1** = 2 Maintainability **PQ_2** = 2 Portability **PQ_3** = 2	Efficiency **PQ_1** = 3 Maintainability **PQ_2** = 3

These examples are further analysed in terms of the framework validity. Therefore, the given quality attributes are assigned with the same weights in order to compare the data further with the qualitative information. Such procedure is also performed when the attributes in question are equal important for a design

2. Normalise the priority values.

System A	System B
Usability **QAP_1** = 0.33 Maintainability **QAP_2** = 0.33 Portability **QAP_3** = 0.33	Efficiency **QAP_1** = 0.50 Maintainability **QAP_2** = 0.50

- **Step 3:**

It is assumed that the value of a characteristic is a sum of its sub-characteristics values. For example: *Efficiency = (Time behaviour + Resource utilisation) /2*

Therefore, the calculation for the final results are:

The required calculations are only presented for the most suitable architectural pattern which best meets the given quality attributes.

System A final results:

$$\begin{aligned}\textbf{Layers}_{[\text{Usability, Maintainability, Portability}]} &= 0.33 * (1.58 + 1.47 + 1.24) \\ &+ 0.33 * (1.80 + 1.84 + 1.63 + 1.72) \\ &+ 0.33 * (1.64 + 1.79 + 1.72 + 1.65) \\ &\approx 5.97\end{aligned}$$

Table 10 lists architectural patterns in order of the ability to fulfil the usability, maintainability, and portability quality attributes. In consequence, the framework recommends the **System A** the Layered pattern to meet best the set of given attributes.

Pos.	Pattern	Score
1.	Layers	5.97
2.	Broker	4.11
3.	Pipes & Filters	3.89
4.	MVC	2.49
5.	PAC	1.31
6.	Microkernel	1.24
7.	Reflection	-0.21
8.	Blackboard	-1.43

Table 10 - RF results for usability, maintainability, and portability

Similarly with the results of the **System B**, where the framework identified the Pipes and filters pattern as the most suitable architectural pattern for a combination of efficiency and maintainability quality attributes. **Table 11** sorts architectural patterns towards their fulfilment of quality attributes identified for fulfilment during the design of **System B**.

System B final results:

$$\begin{aligned}\textbf{Pipes \& Filters}_{[\text{Efficiency, Maintainability}]} &= 0.5 * (1.84 + 1.83) \\ &+ 0.5 * (1.75 + 1.73 + 1.55 + 1.71) \\ &\approx 4.70\end{aligned}$$

Pos.	Pattern	Score
1.	Pipes & Filters	4.70
2.	Microkernel	4.33
3.	Broker	2.69
4.	Reflection	1.80
5.	Layers	1.69
6.	PAC	-1.89
7.	Blackboard	-2.52
8.	MVC	-2.74

Table 11 - RF results for efficiency and maintainability

7.4 Qualitative study

This section presents the literature view, i.e. what information the literature sources provide about the strengths and weaknesses of architectural patterns. Despite the limitations identified in **section 7.2** there are several following that prevent from a straightforward comparison.

Although some relationship between patterns in general and certain quality attributes is available, the impact is investigated in terms of benefits or liabilities that a pattern impose. Moreover, different quality attributes are investigated for different patterns – a number of quality attributes are left unmentioned. It is assumed that the identified relationships are:

- (+) positive – a pattern supports a quality attribute,
- (–) negative – a pattern does not support a quality attribute,
- (x) passive – a quality attribute is neither benefit nor liability of a pattern.

A passive influence is also given when the literature leaves the relationship unmentioned. This means, there is nothing extraordinary in relationship between a patter and a certain quality attribute. Although several relationships are listed, there is no order between them to define:

- how quality attributes affect each other within a particular patter,
- how quality attributes of a pattern influence quality attributes of other patterns.

This complicates the comparison even more. In consequence, it is not a trivial task to perform a relative comparison that is able to predict whether the research outcomes are one hundred percent valid. Nevertheless, it is possible to measure the recommendation framework reliability to some extent. The literature view is represented by Buschmann et al. [9] and Bosch [7] since these sources reveal the strengths and weaknesses of certain architectural patterns[4].

To ensure the comparison is based on the various literature views so that the results are compared to different and independent descriptions, two well-known sources are chosen. Buschmann et al. [9] and Jan Bosch [7] identify strengths and weaknesses of certain architectural patterns at the best of all investigated literature sources. Three most significant and recognized architectural patterns discussed in both [9] and [7] are used for this discussion: layers, pipes and filters, and blackboard.

[4] Jan Bosch in [7] uses the term *architectural style* synonymously to *architectural pattern*.

The first step is to investigate [9] and [7] to identify points of view on the specified architectural patterns. Tables x, x, and x illustrate in a convenient way benefits and liabilities of layers, pipes and filters, and blackboard respectively. These describe the quality attribute relationships relevant for the comparison discussion in the following section.

- **Layers**

Impact	QA	Explanation
+	Changeability	Individual layer implementations can be replaced by semantically-equivalent im0plementations without too great effort [9].
+	Maintainability	Changes often affect only one layer. Adapting affected without altering the remaining layers [9]. Layers have small dependencies on each other; changes are implemented affecting one or small number of components [7].
+	Portability	Changes of the hardware, the operating system, the window system, special data formats and so on often affect only one layer [9].
+	Testability	Test particular layers independently of other components [9].
+	Usability	Supports standardization. Defined and accepted levels of abstraction enable the development of standardized tasks and interfaces. This allows for using products from different vendors in different layers [9].
–	Efficiency	Layering increase high efficiency penalties since data is transferred through a number of intermediate layers [9]. Computation requires several switches of method context resulting in decreased efficiency [7].
x	Reliability	A high degree of reliability (error correction support) can be built into layers, for example by using checksums [9].

Table 12 - Quality attribute strengths and weaknesses of layers

As indicated in **Table 12**, reliability has passive impact. Buschmann et al. discuss its concept, but do not stand on a side. Buschmann et al. in their considerations mention also about sub-quality attributes (also called subcharacteristics) such as testability and changeability that both relate to maintainability, i.e. a top-level quality attribute (a characteristic). These are listed to emphasize the literature view, but are generalized to top-level quality attributes because of the different categorisation of quality attributes. Hence, this makes the straightforward comparison not reliable as these are not equal in meaning. For further discussion, quality attributes may be referred also as characteristics and subcharacteristics divided according to its level in the ISO/IEC 9126 [20] when necessary.

- **Pipes and filters**

Impact	QA	Explanation
+	Changeability	Filters allow for their easy exchange within a processing pipeline [9].
+	Efficiency	Allows for parallel data processing in a multiprocessor system or a network [9].

		Excellent units of concurrency allowing for parallel processing [7].
+	Maintainability	Allows creating new processing pipelines by rearranging, adding and removing filters [9].
–	Reliability	Error handling hard to address since pipelines components do not share any global state [9].
x	Fault tolerance	If a filter detects errors in its input data, the input may be ignored until some clearly marked separation occurs. This approach is useful when incorrect input data is possible and inaccurate results can be tolerated [9]

Table 13 - Quality attribute strengths and weaknesses of pipes and filters

There are several cases when the literature when the literature presents both sides of a given quality attribute. Either there are features that should be considered as benefits or liabilities depending on the particular architecture context and development. **Table 15** lists quality attributes of the investigating architectural patterns to presents their opposite impact sides. Buschmann et al. [9] list efficiency as a benefit of pipes and filters identifying advantages. On the other hand, there are four main concerns on efficiency gained by parallel processing as an illusion. These liabilities are concerned with possible effects and costs involved. Nevertheless, efficiency of pipes and filters used properly is a benefit. In some cases, the literature depicts advantages and disadvantages of an architectural pattern for a certain quality attribute and do not stand on a side.

- **Blackboard**

Impact	QA	Explanation
–	Resource utilization	Needless amount of computation is spent on behaviour not related to the application domain, e.g. roaming the blackboard or multiple components trying to process a data element [7].
–	Time behaviour	No explicitly defined control flow, so the computation is not performed in the optimal order [9].
+	Maintainability	Processing components are independent of each other. Hence, they can be added or removed, without changing other processing components [7]. Blackboard supports maintainability because the individual knowledge sources, the control algorithm and the central data structure are strictly separated [9].
+	Fault tolerance	Supports tolerance for noisy data and uncertain conclusions performed by the system [9].
–	Testability	Computations do not follow a deterministic algorithm [9].
–	Efficiency	Suffer from computational overheads in rejecting wrong hypothesis [9]. Does not support the use of a control strategy that exploits the potential parallelism of knowledge sources [9]. Needless amount of computation not related to a particular application domain [7]

Table 14 - Quality attribute strengths and weaknesses of blackboard

The quality attribute impact that the literature survey presents is only concerned about the development quality requirements, i.e. there that are observable during the system development. Buschmann et al. [9] and Bosch [7] investigate the potential of patterns from a developer point of view. Quality requirements concerned with the user needs and expectations that can be measured on a system in execution are neglected in their discussion. For example, maintainability is mentioned for every pattern, but usability is investigated only by Buschmann et al. in relationship with layers.

QA [Source], Pattern	As benefit	As liability
Reliability [7], Layers	A layer may contain functionality for dealing with faults that occur in other layers.	A failure in one layer may result in the failure of whole system.
Efficiency [9], Pipes and filters	Each filter in a pipeline consumes and produces data in parallel.	Cost and defects comparing to: - transferring data between filters comparing to computation by a single filter - defects in implementation - context switching between threats - synchronization of filters via pipes
Efficiency [7], Pipes and filters	Excellent units of concurrency that allow for parallel processing.	Every filter performs a small unit of computation for each unit of data.
Maintainability [7], Pipes and filters	Organization of pipes and filters allows for their reorganization, even during run-time. Changes allow for adding, changing, removing existing elements.	Changes affect several filters at a time, so the whole their arrangement needs to be modified. The majority of requirements changes affect more than one filter.
Maintainability [7], Blackboard	Independent processing components allow for adding, changing, removing the others.	Naïve design may lead to systems that are hard to maintain.
Reliability [7], Blackboard	The independence of processing components and the fact the control component iteratively activates the various components.	No central or explicit specification of the behaviour which makes hard for a system to identify that some responsibilities are not fulfilled.

Table 15 - Quality attributes from different viewpoints

The quality attribute impact on the following architectural patterns is described entirely from Buschmann et al. [9]:

- **Broker**

Impact	QA	Explanation
+	Portability	Hides operating system details and network system details from clients and servers by using indirection layers as APIs,

		proxies and bridges.
+	Maintainability	Allows for dynamic change, addition, deletion, and relocation of objects. Distributed services are encapsulated within objects.
–	Fault tolerance	Server or broker may fail during program execution and all of the applications that depend on the server or broker are unable to continue successfully.
–	Testability	Many components and many ways of their collaboration failure.
+	Testability	A client application developed from tested services is more robust and easier itself to test.
–	Efficiency	Low efficiency because of the indirection layers that enable the system to be portable, flexile and changeable.

Table 16 - Quality attribute strengths and weaknesses of broker

- **Model-View-Controller**

Impact	QA	Explanation
+	Usability	The model is separated from the user-interface components. Multiple views can therefore be implemented and used with a single model. At run-time, multiple views may be open at the same time, and views can be opened and closed dynamically.
–	Usability	Increased system complexity without gaining much flexibility.
+	Maintainability	Changes to the user interface are easy, and even possible at run-time. Change-propagation mechanism.
+	Portability	Pluggable views and controllers; allows to exchange the view and controller objects of a model. User interface objects can be substitutes at run-time.
–	Efficiency	Inefficiency of data access in view; views need multiple calls to obtain all its display data.

Table 17 - Quality attribute strengths and weaknesses of MVC

- **Presentation-Abstraction-Control**

Impact	QA	Explanation
+	Usability	The model is separated from the user-interface components. Multiple views can therefore be implemented and used with a single model. At run-time, multiple views may be open at the same time, and views can be opened and closed dynamically.
–	Usability	Increased system complexity because of the implementation of every semantic concept as its own PAC agent.
+	Maintainability	Different semantic concepts are represented by separate agents independent of other agents. Data model and user interface for each semantic concept or task within the application developed interdependently of other semantic concepts or tasks.

		Change within the presentation or abstraction components of a PAC agent do not affect other agents in the system.
+	Efficiency	PAC agents can be easily to different threads, processes, or machines.
–	Efficiency	The overhead in the communication between PAC agents when a top-level agent retrieves data from a bottom-level agent (all of them are involved).

Table 18 - Quality attribute strengths and weaknesses of PAC

- **Microkernel**

Impact	QA	Explanation
+	Portability	Migrating the microkernel to a new hardware environment only requires modifications to the hardware-dependent parts.
+	Maintainability	Copes with continuous hardware and software evolution. Implementing an additional view requires only adding a new, external server. Extending the system with additional capabilities only requires the addition or extension of internal servers.
+	Reliability	Allows to run the same server on different machines (replication). Failures are easy to hide from a user and do not affect the application (in such distributed systems).
–/+	Efficiency	If the functional core of the application platform is separated into a component with minimal memory size, and services that consume as little power as possible, Microkernel avoids performance problems.

Table 19 - Quality attribute strengths and weaknesses of Microkernel

- **Reflection**

Impact	QA	Explanation
–	Efficiency	Reflective software are usually slower that non-reflective systems caused by the complex relationship between the base level and the meta level. Inefficiency with extra processing with information retrieval, changing metaobjects, consistency checking, and the communication between the levels.
+	Maintainability	The metaobject protocol provides a safe and uniform mechanism for changing software. Metaobjects encapsulate every aspect of system behaviour. Supports changes of any kind of scale – even the fundamental aspects can be changed.
–	Maintainability	Modifications at the meta level may cause damage to the software or its environment (dangerous changes).
–	Usability	Increased number of components – the greater the number of aspects that are encapsulated at the meta level, the more metaobjects there are (system complexity).

Table 20 - Quality attribute strengths and weaknesses of Reflection

7.5 Comparative discussion

This section compares the qualitative study results to the quantitative recommendation framework data. In order to start and facilitate the discussion, **Table 21** lists the summarised literature survey outcomes next to the required framework values. The brackets indicate which side of the presented two possibilities is the final (resulting) one. It should be noted that the limitations described in **section 7.2** involve a certain amount of translations and interpretations. This procedure allows for comparing the qualitative and quantitative information together in a convenient way. Nevertheless, it reduces the number of quality attributes that can be compared. It should also be noted that values for characteristics are the average values of their subcharacteristics as follows:

For example concerning reliability:

$$Reliability = (Maturity + Fault\ tolerance + Recoverability)/3$$

Therefore, reliability for layers:

$$Reliability_{[Layers]} = [(-0.47) + (-0.54) + (-0.62)]/3 \approx -0,54$$

This procedure allows for the comparison. However, it does not take into account the quality attribute relationships described in **section 3.2.4**. Quality attributes may influence, i.e. strengthen or hinder each other. It is assumed for the purpose of this comparison that quality attributes in a group (subcharacteristics) strengthen each other more or less in the same way. Nevertheless, this assumption may not be correct in every case.

It is not a trivial task to compare quantitative values to qualitative information. Nevertheless, some conclusions can be pointed out. **Table 21** proves to some extent the recommendation framework validity. Surprisingly majority of the empirical values reflect the literature view, e.g. when the value is above zero, the literature mentioned that a pattern is fairly good at a quality attribute (positive impact). The values that are coloured in orange present strong-value disagreements.

Moreover, the examples presented in the previous section are based on quality attributes that the literature presented positive influence. Layers are proved by the literature to be good at maintainability [7][9], portability [9] and usability [9]. Therefore these quality attributes were used in the example. In the example results, the framework puts layers on the first place (see **Table 10**) according to a set of these quality attributes with equal weights of importance. The third place belongs to pipes and filters which are proven in the literature to have positive influence on maintainability [7][9]. However, the literature says nothing about portability and usability of pipes and filters.

The second example selected an architectural pattern based on maintainability and efficiency attributes. The results in **Table 12** indicated that the best opportunity for these quality attributes with equal importance values is the pipes and filters, followed by microkernel and broker not investigated in the literature survey. However, the first result reflects he literature view on positive aspects of maintainability [7][9] and efficiency [7][9].

AP	QA	Bosch	Buschmann	RF
Layers	Reliability	–/+	x	-0.54
	Maintainability	+	+	1.75
	Changeability (maintainability)	x	+	1.84
	Testability (maintainability)	x	+	1.72
	Efficiency	–	–	-1.71
	Portability	x	+	1.70
Pipes & filters	Reliability	–	–	-0.67
	Maintainability	–/(+)	+	1.69
	Efficiency	–/(+)	–/(+)	1.84
Blackboard	Reliability	–/+	+	0.97
	Maintainability	–/(+)	+	0.39
	Testability (maintainability)	x	–	0.67
	Efficiency	–	–	-0.03
	Resource utilization (efficiency)	x	–	-0.12
	Time behaviour (efficiency)	x	–	0.06
Broker	Portability		+	1,55
	Maintainability		+	1,33
	Fault tolerance (reliability)		–	-1,45
	Testability (maintainability)		–/+	1,25
	Efficiency		–	-0,41
MVC	Usability		–/+	1,75
	Maintainability		+	1,24
	Portability		+	-0,66
	Efficiency		–	-1,42
PAC	Usability		–/+	1,22
	Maintainability		+	1,09
	Efficiency		–/+	0,13
Microkernel	Portability		+	1,57
	Maintainability		+	0,45
	Reliability		+	1,32
	Efficiency		–/(+)	1,20
Reflection	Efficiency		–	-0,80
	Maintainability		–/(+)	-0,21
	Usability		–	-1,67

Table 21 - Summarised comparison values

Another way to verify the validity of the recommendation framework results is to compare the framework data against data from a related research conducted by Svahnberg and Wohlin [31]. Both of data is quantitative in nature so the differences can be measured in mathematical values. The original results of this research is attached in **Appendix 1**. In order

to make these comparisons, several preparations needed to be done. First of all, Svahnberg and Wohlin [31] used five of total eight patterns categorised by Buschmann et al. [9] as architectural. These are: blackboard, layers, microkernel, model-view-controller, and pipes & filters. Also, recommendation framework takes into account not only the top-level quality attributes (characteristics) from the ISO-IEC 9126 quality model [20], but also attributes from the lower level (subcharacteristics). The recommendation framework does not take into account the functionality characteristics because of the reasons described in **Chapter Five**. Naturally, the comparison will be limited to these factors. Moreover, the research in [31] is based on a framework that consists of two tables:

- Framework for Architecture Structures (FAS) which rates the ability of each architectural pattern to support for different quality attributes,
- Framework for Quality Attributes (FQA) that ranks which architectural pattern is best situated at each of the specified quality attributes.

Both of these framework are compared and for that purpose the data have to be once again normalised, but this time – without the functionality quality attribute. Then, the recommendation framework data is also normalised and gather in **Table 22** and **Table 23 that** corresponds FAS and FQA frameworks respectively. Pairs of quality attributes with similar values (similar results in both researches) are marked in orange colour.

	Microkernel		Blackboard		Layers		MVC		Pipes & Filters	
	Svahnberg	RF	Svahnberg	RF	Svahnberg	RF	Svahnberg	RF	Svahnberg	RF
Efficiency	0,183	0,244	0,214	0,201	0,074	0,023	0,063	0,049	0,257	0,273
Usability	0,120	0,043	0,187	0,173	0,334	0,272	0,118	0,317	0,096	0,126
Reliability	0,138	0,254	0,108	0,304	0,122	0,115	0,119	0,247	0,169	0,094
Maintainability	0,208	0,187	0,403	0,244	0,289	0,297	0,339	0,274	0,319	0,262
Portability	0,351	0,272	0,088	0,077	0,181	0,293	0,362	0,113	0,159	0,244

Table 22 – Quantitative research results comparison on FAS

		Microkernel	Blackboard	Layers	MVC	Pipes & Filters
Efficiency	**Svahnberg**	0,264	0,175	0,087	0,113	0,360
	RF	0,324	0,199	0,029	0,059	0,388
Usability	**Svahnberg**	0,914	0,113	0,250	0,408	0,137
	RF	0,051	0,151	0,306	0,334	0,158
Reliability	**Svahnberg**	0,126	0,142	0,318	0,190	0,224
	RF	0,277	0,248	0,121	0,244	0,111
Maintainability	**Svahnberg**	0,191	0,092	0,285	0,239	0,193
	RF	0,158	0,154	0,242	0,209	0,238
Portability	**Svahnberg**	0,112	0,069	0,426	0,139	0,225
	RF	0,279	0,059	0,289	0,105	0,268

Table 23 – Quantitative research results comparison on FQA

These basic examples prove to some extent the literature point of view, and hence – the recommendation framework validity. However, the respondents might have known these sources and what they say about architectural patterns in terms of their relationships with quality attributes. Nevertheless, architectural patterns used in the questionnaire are well-known patterns and their benefits and liabilities are commonly recognized in practise which reflects to some extent the recommendation framework (RF) validity.

7.6 Summary conclusions

Section 4.3.2 describes the two approaches of evaluation, i.e. qualitative and quantitative. The qualitative information is gathered from the literature, and the quantitative information is represented by the recommendation framework data. The disadvantage of the qualitative approach is that comparing the given architectural patterns for more that one quality attribute, the outcome is still 'boolean'. On the opposite, despite the quantitative data of the recommendation framework, there are still no means to identify the potential limitations for an architecture. Nevertheless, it provides better a reasoning background than the qualitative information.

A number of translations and assumptions had to be made. These decrease the ability to make appropriate comparisons of the literature view and the recommendation framework values. **Table 21** summarises the comparison of qualitative and quantitative information. The values prove to some extent the recommendations framework potential for choosing an architectural pattern that fulfils the desired quality attributes.

Chapter Eight – Summary and concluding remarks

8.1 Research summary

The total work presented in this paper is concerned with a number of research activities. In order to understand the role of quality requirements in software architecture design a number of related concepts were presented. The first of the objectives was to define the software architecture in terms of its relationship with quality requirements. **Chapter Two** discusses software architecture of a software system as a set of components of the system, their responsibilities and interactions. Their composition allows not for addressing functional requirements, but also is a method for an early fulfilment of quality requirements. Views are also described as they are used in software architecture to exhibit different quality requirements important during software architecture design and evaluation (**Chapter Four**). Therefore, a good architecture is important in order to achieve the desired quality requirements. Attention to quality requirements is also crucial for software quality. By leaving them unfulfilled, the system lacks in required quality level.

The second objective was to identify and classify quality requirements which influence the selection of software architectures. **Chapter Three** is a detailed description on quality requirements, where they are divided similarly by Bosch [7] and Bass et al. [2] into two categories: operational (observable via execution) and development (not observable via execution). The fulfilment of quality requirements (especially development-oriented) cannot be measured before the system is actually implemented. Yet, they are hard to deal with since they often tend to interact with each other, having positive or negative impact (**section 3.2.4**). However, during the design phase, much of the quality aspects of a system can be addressed. During software architecture design such requirements need to be prioritized (**section 3.2.6**) and balanced in design tradeoffs (**section 3.2.7**) when architects have to decide upon the selection of a particular software architecture solution.

Software architecture design activity (described in **Chapter Four**) takes the requirement specification that contains both functional and quality requirements as an input, and results in an artefact – a software architecture. In other words, it is about converting requirements into software architecture that fulfils these requirements. Software architecture design determines whether the software architecture has fulfilled its requirements, especially quality requirements. There is still lack of knowledge on what was proved to be the most important – little practical guidance on how to manage the design activity in terms of quality requirements. Usually architectural design means taking steps to provide the system with expected functionality. However, a number of different quality attributes are also of interest in software architecture. These attributes are of crucial importance because they constrain quality requirements, which in turn constrain the design and development of software architecture. These considerations in **Chapter Three** discuss the fourth objective of this paper, i.e. software architecture design as a method of achieving quality requirements. Furthermore, this objective is continued in the discussion on patters that provide an approach for developing software with predefined quality requirements, and hence – high-quality software architectures. They document existing design knowledge and help to choose the most suitable solution to recurring design problems. Patterns exist in various ranges of scale

and abstraction (**section 4.2.1**). Different patterns imply different design consequences, that includes the fulfilment of different quality requirements. This means that the different compositions of components and their specified responsibilities and interactions fulfil a number of quality attributes. However, the selection of architectural pattern(s) is only the first step during the software architectural design.

Chapters Two – Four are responsible for presenting the state-of-the-art analysis. The findings are described in terms of the objectives concerned with the main aims of this thesis. S**ection 8.2** summarizes the literature survey and indicates relevant issues that should be considered as recommendations. These concluding remarks emphasize the meaning and importance of ensuring quality requirements in architectural design.

8.2 Proposed solutions

Systems are built to satisfy their requirements. Software architecture design determines ensures that the fulfilment of system requirements by the software architecture. There is still lack of knowledge and what matters the most – little practical guidance on how to manage the design activity towards the achievement of quality requirements. Usually design means taking steps to provide the system with its expected functionality. However, as it was proved, to ensure the required level of software quality a system must fulfil also quality requirements. Therefore, this paper has proposed a *requirement-oriented design method* in **section 4.4**. It should be considered as an outcome of the literature review since these results revealed the gap between quality requirements and software architecture. Having reviewed important aspects of several software architecture design methods, little but not sufficient attention is paid to govern the design towards the fulfilment of quality requirements. The proposed method is inspired by an important breakthrough in this area, i.e. Bosch design method [7]. In opposite to Bosch, this design process starts with a design of a preliminary version of the software architecture based on quality requirements specified in requirements specification. The objective of this model is to design a software architecture that targets both requirements types of a system. In general, these quality requirements strategies involve of ensuring the existence or a specified order (decomposition) of one or more components (mechanisms) that fulfil desired attributes of a system. It is also worth to mention that all of these abstract from the system functional requirements. Therefore, the proposed method find its usage in software architecture design activities.

The method is aimed towards:

- design software architecture that targets both types of requirements
- fill the gap of quality requirements in software architecture design,
- ensure software high quality compared to the resources used in architectural design.

The benefits and general features of this approach in comparison to Bosch include:

- less modification-prone, i.e. avoids modifications caused by the lack of QRs during the functionality-oriented design,
- early fulfilment of the top abstraction level (global) QRs,
- addresses QRs before the core functionality is placed,
- saves efforts due to the smaller number of QRs evaluations
- considers FRs variability caused by QRs ensuring procedures,
- allows for sacrificing FRs in order to meet QRs,
- similar development progress – FRs & QRs are fulfilled comparatively.

Besides the conceptual design considerations, this research also presents a practical solution to the identified problems. Architectural patterns from [9] represent the highest-level patterns. They are used to specify the fundamental architectural structure. Every development activity that follows is governed by this structure [9]. The selection of an architectural pattern is greatly influenced by the quality requirements of a system. That is why they have been chosen for the proposed Recommendation Framework (**Chapter Six**) that is a systematic approach of fulfilling quality requirements during software architecture design stage. This helps to systematically guide selection among design alternatives. It selects the most suitable architectural pattern from a set of given quality attributes. The idea of such solution originates from the observation of certain architectural structures to predict the quality attributes they affect [30]. Although the literature [7] and [9] provides benefits and liabilities of a certain pattern, the qualitative nature of the information does not allow for equal measurements and comparisons. Hence, an empirical research was conducted to obtain the required data for the proposed method. Based on the questionnaire quantitative outcomes, an automated design support is created. **Chapter Seven** discusses the framework's validity. Despite the solution reliability, it is a step forward in designing towards quality requirements.

The recommendation framework uses patterns categorised as architectural besides design patterns and idioms also specified in [9]. These patterns are divided with respect to their range of scale and abstraction. Different quality requirements are addressed at different abstraction levels. The first proposed model is a quality *requirement-oriented design method*, where quality requirements are taken into account at first during a software architecture design. The recommendation framework is considered as a practical illustration of such an approach. However, it concentrates on architectural patterns that express the fundamental structure and hence, the quality requirements of the highest-abstraction level. These considerations and the lack of a support for achieving quality requirements at all design levels resulted in a third, last proposal method of this thesis – a *quality requirement-oriented and pattern-based design method* described in **section 6.7**. This method is proposed to deal with the quality requirements at all abstraction levels specified by patters in [9].

8.3 Conclusions

Software engineers used to provide systems that concentrate on the required system behaviour, i.e. functional requirements. Today the software market is full of applications with similar functionality, and the factor that differs an application from another is its quality level. As it was proved in this thesis, the level of quality depends greatly on the achievement of quality requirements. This means that the fulfilment of quality requirements set the boundaries for the total software quality of a system. Therefore, in order to select the best solution from comparable (similar) applications, one has to take into consideration how, and to what degree these applications fulfil the desired quality requirements. This should results in increased customer satisfaction.

As it was indicated in **Chapter Three** quality requirements may affect one part of an application or a system as a whole. To underline their importance it should be stated that functional requirements may need to be sacrificed in order to meet the system quality requirements, and in result – the product goals. This is because the lack of a system service (a functional requirement) may degree the system usability, while leaving a quality requirement unfulfilled can make the system totally useless [28].

The designed system itself has an impact on quality requirements. The larger the system is developed, the more crucial quality requirements of such system are. This means that the importance of quality requirements increases with the size of a designed system.

Similarly with complexity – the more complex system is, the more attention is paid to quality requirements and their fulfilment. Chapter Three presented how quality requirements are divided (development and operational). These categories also influence the software architecture design. First of all, operational-oriented quality requirements are in most cases impossible to incorporate at the software architecture level as they can be observed and measured during the system execution. What is extremely important that the cost of incorporating quality requirements into a system that has been developed absorbs a lot of resources such as schedule or budget costs. Therefore, it is important to ensure the fulfilment of quality requirements as soon as possible to avoid such resource penalties.

The research presented in this thesis benefits in a better understanding of quality requirement-related issues. It lays out quality requirements in such a way that they can govern architectural decisions and be used to evaluate the architecture. This thesis proposes a quantitative approach for achieving software product quality. This might be a step forward towards the systematization of design methods and quality requirement-oriented approach of architectural designs. The research focused on analysis of software architectures against one or more desired software qualities that ought to be achieved at the architectural level.

8.4 Concluding remarks

While the previous section summarised what has been done, this part indicates relevant concepts described in this thesis. These are the state-of-the-art findings that ought to be considered as recommendations. They are presented informally to increase attention and avoid potential misunderstanding in the area of quality requirements in software architecture design.

Software architecture design (**Chapter Two** and **Four**) is not concerned with the design of algorithms and data structures. It is commonly agreed that software architecture design of a system is:

- a high-level system design beyond the algorithms and data structures of the computation
- an overall organization of elements and their relations (components and connectors)
- a coherent combination of views that describe the system features from different perspectives
- a set of fundamental design decisions that establish a system topology and vocabulary.

Conclusion 1.	**Software architecture** is a structure that illuminates the top-level design decisions. It governs the system decomposition into interacting parts as results of **design** decisions towards the fulfilment of system requirements.

Conclusion 2.	The design process of achieving system requirements via architectural means is called **software architecture design** and results in an artefact called **software architecture**.

Software architecture is often separated into multiple views that present the system features from different perspectives. One view presents various quality requirements that may be invisible via others. Views reduce the complexity and help to make decisions about trade-offs. Views are represented by a number of notations such as Unified Modelling Language (UML) considered as one of Architecture Description Languages (ADLs).

The design process is not only governed by functional requirements, but also by quality requirements [2][7][9][16]. Functional requirements capture the intended behaviour of a system (services or tasks to perform), while the quality requirements impose constrains or restrictions on the software product and the development process [12].

Conclusion 3. **FRs** describe *what* a system does, whereas **QRs** put constrains on *how* these **FRs** are ought to be implemented.

Quality requirements affect either one part of an application or the system as a whole. In some cases, functional requirements have to be sacrificed in order to meet the quality requirements, and in result – the product goals. A lack of a functional requirement (a system service) may degree the system usability, while not covering a quality requirement could make the system totally useless [13].

Functional requirements are usually captured with UML's use-cases, analyzed by sequence diagrams, statecharts, etc. Quality requirements are often specified as a part of functional requirements, e.g. "The system shall be able to present a response message *no later than 3 seconds"*. In other words, quality requirements determine overall constrains on the functionality.

Conclusion 4. *Quality requirements = Functional requirements + Constrains*

A software system has many characteristics, e.g. maintainability, reliability usability. The quality of each of these characteristics determine the total software quality. Each characteristic can be specified as an attribute of the system if a metric is given to verify that the architecture addresses the quality.

Conclusion 5. **Quality attributes** are measurable or observable properties of a system that have some qualitative or quantitative value.

Quality requirements determine the quality attributes of a system by placing constraints. These are usually specific values, a scope, or ranges of values for quality attributes.

Conclusion 6. *Quality requirements = Quality attributes + Constrains*

Quality attributes are categorised into development and operational [7]. Development quality requirements are attributes relevant from a developer perspective, e.g. maintainability, portability. Operational quality requirements are noticeable and measurable on *the system in operation, e.g.* efficiency, security, usability.

It is important to understand the potential relationships between quality attributes, especially conflicts (**section 3.2.4**). It allows for monitoring how different quality attributes interact with each other. It is especially important to examine potential conflicts in order to minimize the inappropriate design decisions.

Conclusion 7. Quality attributes often *interact* each other *positively* or *negatively*.

One way to design a software system is to start with a pattern. Patterns describe a set of components, their relationships and the required constraints, the rationale for their cooperation, and the software qualities they provide. Buschmann et al. [9] categorized

patterns according to their level of abstraction. Architectural patterns are concerned with the fundamental structure of a system. Design patterns are concerned with smaller architectural units such as subsystems or components. In opposite to design patterns, idioms are language specific patterns that concern implementation matters of particular design issues.

Conclusion 8. Patterns *address* some quality attributes at *various* abstraction levels.

Software architecture design should determine whether the design result, i.e. software architecture, has fulfilled the (quality) requirements. Software architecture evaluation is performed to measure quality attributes, so these can be compared to the quality requirements from the requirements specification. The main purpose of architectural evaluations is to asses the quality attributes of a system during the design – without having a concrete system available. Different approaches for assessing quality requirements have been identified such as *scenarios, simulation,* and *mathematical modelling* [7].

Conclusion 9. *Architecture evaluation* measures how well the architecture addresses quality requirements of the system.

One of the major issues in software systems development today is quality. The notion of software architecture determines the level for dealing with software quality. This is because the overall quality of a system depends on the fulfilment of system requirements.

Conclusion 10. *Software quality* is governed by the level of quality requirements fulfilment.

Little, but not sufficient research has been done to design software architecture from quality requirements. Although there has been some interaction investigated by the literature [3][4][9][44], but the task still remains difficult.

8.5 Future work

The author may wish to consider possible further regarding the software architecture design in terms of quality-related issues. In particular, there are several aims of future work described as follows.

Unified Modelling Language (UML) is often used as an Architectural Description Language (ADL). Although it describes detailed design decision, it is also used to manifest the high-level software architecture design. However, UML does not specify the boundary between the designs at different abstraction levels. While UML emerged from object-oriented designs, it commonly supports various-level designs. Confusion exists since a software architect cannot distinguish between architectural information (high-level design decisions and artefacts), and other types of information.

The aim is to specify the difference between using UML for software architecture and the more common use of designing applications with UML. This should results in a proposal of an UML usage for high-level and detailed software design.

Software architecture design is worth nothing without requirements engineering – the first activity in software development life cycle. The path to fulfilling quality requirements begins with eliciting, verifying, documenting and generally – managing quality requirements.

As this paper indicated - insufficient time and effort are spent on the quality requirement-related activities associated with the gathering quality requirements for the use of architectural design. Many software requirements specifications (also called software requirements documents), are either full of badly written (quality) requirements or do not specify them at all. In order to meet them properly by the software architecture, they need to be precisely specified during requirements engineering. UML use-case models are commonly used to express functional requirements, whereas quality requirements are specified as footnotes or supplementary text. Further work also aims towards improvements in eliciting, analyzing and verifying quality requirements. This might result in extending an ADL for software architecture to deal with quality requirements. A possible target is the Unified Modelling Language (UML).

An important issue in this research is the concept of interdependencies among quality attributes (**section 3.2.4**). Quality attributes often influence each other, either strengthen (e.g. *security* and *safety*) or hinder (e.g. *efficiency* and *maintainability*). In order to address a set of desired quality attributes in software architecture design, their relationship has to be recognized. Future work aims also includes identifying, specifying and testing quality attributes interdependencies.

An empirical research was conducted to collect the data required for the recommendation framework (**Chapter Six**). The literature such as [7][9] provides some dependencies between quality attributes and software architecture structures, especially patterns. However, a number of quality attribute impact is not recognized. Future study aims also identifying these missing relationships.

References

[1] S. T. Albin, *The Art of Software Architecture: Design Methods and Techniques*, John Wiley and Sons, 2003.

[2] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Addison Wesley Longman, 1998.

[3] L. Bass, M. Klein, F. Bachmann, *Quality Attribute Design Primitives*, Technical Note CMU/SEI-2000-TN-017, 2000.

[4] P. Bengtsson, *Architecture-Level Modifiability Analysis*, Doctoral Dissertation Series No. 2002-2, Blekinge Institute of Technology, 2002.

[5] P. Bengtsson, *Software Architecture - Design and Evaluation*, Research Report 10/99, Blekinge Institute of Technology, 1999.

[6] B. Boehm, J. Brown, H. Kaspar, M. Lipow, G. McLeod, M. Merritt, *Characteristics of Software Quality*, North Holland, 1978.

[7] J. Bosch, *Design and Use of Software Architectures. Adopting and Evolving a Product-Line Approach*, Addison-Wesley, 2000.

[8] M. Bray, M. Ross, G. Staples, *Software Quality Management IV: Improving Quality*, Mechanical Engineering Publications, 1996.

[9] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture. A System of Patterns*, John Wiley and Sons, 1996.

[10] L. Chung, B. A. Nixon, E. Yu and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishing, 1999.

[11] L. M. Cysneiros, J.C.S.P Leite, *Non-Functional Requirements: From Elicitation to Conceptual Model*, IEEE Transactions on Software Engineering, May 2004.

[12] L. Dobrica, E. Niemela, *A Survey on Software Architecture Analysis Methods*, IEEE Transactions on Software Engineering, Vol. 28, No. 7, 2002.

[13] A. Eden, R. Kazman, *Architecture, Design, Implementation*, Proceedings of the 25th International Conference on. Software Engineering (ICSE 25), 2003.

[14] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns - Elements of Object-Oriented Software*, Addison-Wesley, 1994.

[15] D. Gross, E. Yu, *From Non-Functional Requirements to Design through Patterns*, Requirements Engineering, Vol. 6, No. 1, pp. 18-36, February 2001.

[16] C. Hofmeister, R. Nord, D. Soni, *Applied Software Architecture*, Addison-Wesley, 2000.

[17] Institute of Electrical and Electronics Engineers, IEEE Standard 1061-1998: *A Standard for a Software Quality Metrics Methodology*, New York, 1998.

[18] Institute of Electrical and Electronics Engineers, IEEE Standard 610.12-1990: *IEEE Standard Glossary of Software Engineering Terminology*, New York, 1990.

[19] International Standards Organization, *Reference Model for Open Distributed Processing*, International Standard 10746-1, ITU Recommendation X.901, 1996.

[20] International Standards Organization, *Software Engineering - Product Quality, Parts 1 – 4*, ISO/IEC 9126, 2001.

[21] R. Kazman, M. Klein, P. Clements, *ATAM: Method for Architecture Evaluation*, CMU/SEI-2000-TR-004, ADA382629, Software Engineering Institute, 2000.

[22] P. Kruchten, *The "4+1" View Model of Software Architecture*, IEEE Software 12 no. 6, pp. 42-50, 1995.

[23] P. Lalanda, S. Cherki, Object-oriented methods and software architecture, Proceedings of the ECOOP'98 on Object-Oriented Software Architectures, Blekinge Institute of Technology, 1998.

[24] L. Lundberg, M. Mattsson, C. Wohlin, *Software quality attributes and trade-offs*, Blekinge Institute of Technology, 2005.

[25] J.A. McCall, Quality Factors, *Encyclopaedia of Software Engineering*, John Wiley and Sons, 1994.

[26] J. A. McCall, P. K. Richards, G. F. Walters, *Factors in Software Quality*, Technical Report (RADC)-TR-77-369, NTIS, Volumes I, II, III, 1977

[27] I. Sommerville, G. Kotonya, *Requirements Engineering: Processes and Techniques*, John Wiley and Sons, 1998.

[28] I. Sommerville, *Software Engineering, 6th edition*, Addison Wesley, 2000.

[29] M. Svahnberg, C. Wohlin, *A Comparative Study of Quantitative and Qualitative Views of Software Architectures*, Proceedings of the 7th International Conference on Empirical Assessment in Software Engineering (EASE 2003).

[30] M. Svahnberg, C. Wohlin, L. Lundberg, M. Mattsson, *A Method for Understanding Quality Attributes in Software Architecture Structures*, Proceedings of the 14th International Conference on Software Engineering Decision Support, (SEKE 2002).

[31] M. Svahnberg, C. Wohlin, *An Investigation of a Method for Identifying a Software Architecture Candidate with respect to Quality Attributes*, Empirical Software Engineering, 10, 149–181, 2005.

[32] M. Svahnberg, C. Wohlin, *Consensus Building when Comparing Software Architectures*, Proceedings of the 4th International Conference on Product Focused Software Process Improvement (PROFES 2002).

[33] *UML 2.0 Superstructure – Final Adopted Specification*, OMG document ptc/03-08-02, 2002.

[34] K. E. Wiegers, *Software Requirements*, Microsoft Corporation, 2000.

Appendix 1

	Microkernel	Blackboard	Layered	Model-View-Controller	Pipes and Filters
Efficiency	0.161	0.145	0.0565	0.0557	0.218
Functionalit	0.119	0.321	0.237	0.115	0.151
Usability	0.106	0.127	0.255	0.104	0.0818
Reliability	0.122	0.0732	0.0930	0.105	0.144
Maintainabilit	0.183	0.273	0.221	0.300	0.271
Portabilit	0.309	0.0597	0.138	0.320	0.135

Table 24 - Framework for Architecture Structures (FAS) [31]

	Microkernel	Blackboard	Layered	Model-View-Controller	Pipes and Filters
Efficiency	0.264	0.175	0.0868	0.113	0.360
Functionalit	0.205	0.252	0.199	0.206	0.139
Usability	0.0914	0.113	0.250	0.408	0.137
Reliability	0.126	0.142	0.318	0.190	0.224
Maintainabilit	0.191	0.0921	0.285	0.239	0.193
Portabilit	0.112	0.0689	0.426	0.139	0.255

Table 25 - Framework for Quality Attributes (FQA) [31]

Appendix 2 – Questionnaire

Questionnaire

This is an **anonymous** questionnaire about the role of **Quality Requirements** (also referred as **Non-Functional Requirements**) in software architecture design. It will be very important and helpful for my research if you could spend a few minutes answering the questions below. Please complete the questionnaire honestly at the best you can.

A. Benefits of the research and your participation:

1. You will be given the results and conclusions.
2. The solution of this research may help in your future designs.
3. Opportunity to help in an academic experiment.
4. Your participation will provide results that may impact software engineering education.

B. General information

1. Have you taken participation in software architecture design?

□ Yes	□ No

2. **If yes**, specify more or less how many times:

3. How do you grade your knowledge about quality (non-functional) requirements?

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Any	Bad	Poor	Below average	Average	Above average	Good	Very good	Superb	Excellent

4. Do you take into account quality requirements (besides the functionality of a system) in your designs?

□ Yes	□ No

5. How do you grade your knowledge about patterns?

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Any	Bad	Poor	Below average	Average	Above average	Good	Very good	Superb	Excellent

C. Architectural Patterns and Quality Attributes

This research is an attempt of creating a recommendation framework. It shall provide an automated support in choosing the most suitable software architecture description for the given quality attributes. These attributes come from quality (non-functional) requirements that constrain a software system. **Your answers** will help to gather the required data, based on which the transformation from quality requirements into a architectural pattern will be created.

Reminders to:

1) Architectural Patterns: http://www.student.bth.se/~kkwn05/architectural_patterns.htm

2) Quality Attributes: http://www.student.bth.se/~kkwn05/quality_attributes.htm

available through the website.

Legend for the assessment:

Grade:	Explanation:
+2	**High positive** impact of a quality attribute on an architectural pattern.
+1	**Positive** impact of a quality attribute on an architectural pattern.
0	**Passive** impact (neither benefit nor liability).
-1	**Negative** impact of a quality attribute on an architectural pattern.
-2	**High negative** impact of a quality attribute on an architectural pattern.

Thank you for your cooperation!

1. Layers

1.1. How do you grade your familiarity with the **layered** pattern?

1	2	3	4	5
None	Poor	Average	Good	Excellent

1.2. Reliability

Maturity	-2	-1	0	+1	+2
Fault tolerance	-2	-1	0	+1	+2
Recoverability	-2	-1	0	+1	+2

1.3. Usability

Understandability	-2	-1	0	+1	+2
Learnability	-2	-1	0	+1	+2
Operability	-2	-1	0	+1	+2

1.4. Maintainability

Analysability	-2	-1	0	+1	+2
Changeability	-2	-1	0	+1	+2
Stability	-2	-1	0	+1	+2
Testability	-2	-1	0	+1	+2

1.5. Efficiency

Time behaviour	-2	-1	0	+1	+2
Resource utilisation	-2	-1	0	+1	+2

1.6. Portability

Adaptability	-2	-1	0	+1	+2
Installability	-2	-1	0	+1	+2
Co-existence	-2	-1	0	+1	+2
Replaceability	-2	-1	0	+1	+2

2. Pipes and Filters

2.1. How do you grade your familiarity with the **pipes and filters** pattern?

1	2	3	4	5
None	Poor	Average	Good	Excellent

2.2. Reliability

Maturity	-2	-1	0	+1	+2
Fault tolerance	-2	-1	0	+1	+2
Recoverability	-2	-1	0	+1	+2

2.3. Usability

Understandability	-2	-1	0	+1	+2
Learnability	-2	-1	0	+1	+2
Operability	-2	-1	0	+1	+2

2.4. Maintainability

Analysability	-2	-1	0	+1	+2
Changeability	-2	-1	0	+1	+2
Stability	-2	-1	0	+1	+2
Testability	-2	-1	0	+1	+2

2.5. Efficiency

Time behaviour	-2	-1	0	+1	+2
Resource utilisation	-2	-1	0	+1	+2

2.6. Portability

Adaptability	-2	-1	0	+1	+2
Installability	-2	-1	0	+1	+2
Co-existence	-2	-1	0	+1	+2
Replaceability	-2	-1	0	+1	+2

3. Blackboard

3.1. How do you grade your familiarity with the **blackboard** pattern?

1	2	3	4	5
None	Poor	Average	Good	Excellent

3.2. Reliability

Maturity	-2	-1	0	+1	+2
Fault tolerance	-2	-1	0	+1	+2
Recoverability	-2	-1	0	+1	+2

3.3. Usability

Understandability	-2	-1	0	+1	+2
Learnability	-2	-1	0	+1	+2
Operability	-2	-1	0	+1	+2

3.4. Maintainability

Analysability	-2	-1	0	+1	+2
Changeability	-2	-1	0	+1	+2
Stability	-2	-1	0	+1	+2
Testability	-2	-1	0	+1	+2

3.5. Efficiency

Time behaviour	-2	-1	0	+1	+2
Resource utilisation	-2	-1	0	+1	+2

3.6. Portability

Adaptability	-2	-1	0	+1	+2
Installability	-2	-1	0	+1	+2
Co-existence	-2	-1	0	+1	+2
Replaceability	-2	-1	0	+1	+2

4. Broker

4.1. How do you grade your familiarity with the **broker** pattern?

1	2	3	4	5
None	Poor	Average	Good	Excellent

4.2. Reliability

Maturity	-2	-1	0	+1	+2
Fault tolerance	-2	-1	0	+1	+2
Recoverability	-2	-1	0	+1	+2

4.3. Usability

Understandability	-2	-1	0	+1	+2
Learnability	-2	-1	0	+1	+2
Operability	-2	-1	0	+1	+2

4.4. Maintainability

Analysability	-2	-1	0	+1	+2
Changeability	-2	-1	0	+1	+2
Stability	-2	-1	0	+1	+2
Testability	-2	-1	0	+1	+2

4.5. Efficiency

Time behaviour	-2	-1	0	+1	+2
Resource utilisation	-2	-1	0	+1	+2

4.6. Portability

Adaptability	-2	-1	0	+1	+2
Installability	-2	-1	0	+1	+2
Co-existence	-2	-1	0	+1	+2
Replaceability	-2	-1	0	+1	+2

5. Model-View-Controller

5.1. How do you grade your familiarity with the **MVC** pattern?

1	2	3	4	5
None	Poor	Average	Good	Excellent

5.2. Reliability

Maturity	-2	-1	0	+1	+2
Fault tolerance	-2	-1	0	+1	+2
Recoverability	-2	-1	0	+1	+2

5.3. Usability

Understandability	-2	-1	0	+1	+2
Learnability	-2	-1	0	+1	+2
Operability	-2	-1	0	+1	+2

5.4. Maintainability

Analysability	-2	-1	0	+1	+2
Changeability	-2	-1	0	+1	+2
Stability	-2	-1	0	+1	+2
Testability	-2	-1	0	+1	+2

5.5. Efficiency

Time behaviour	-2	-1	0	+1	+2
Resource utilisation	-2	-1	0	+1	+2

5.6. Portability

Adaptability	-2	-1	0	+1	+2
Installability	-2	-1	0	+1	+2
Co-existence	-2	-1	0	+1	+2
Replaceability	-2	-1	0	+1	+2

6. Presentation-Abstraction-Control

6.1. How do you grade your familiarity with the **PAC** pattern?

1	2	3	4	5
None	Poor	Average	Good	Excellent

6.2. Reliability

Maturity	-2	-1	0	+1	+2
Fault tolerance	-2	-1	0	+1	+2
Recoverability	-2	-1	0	+1	+2

6.3. Usability

Understandability	-2	-1	0	+1	+2
Learnability	-2	-1	0	+1	+2
Operability	-2	-1	0	+1	+2

6.4. Maintainability

Analysability	-2	-1	0	+1	+2
Changeability	-2	-1	0	+1	+2
Stability	-2	-1	0	+1	+2
Testability	-2	-1	0	+1	+2

6.5. Efficiency

Time behaviour	-2	-1	0	+1	+2
Resource utilisation	-2	-1	0	+1	+2

6.6. Portability

Adaptability	-2	-1	0	+1	+2
Installability	-2	-1	0	+1	+2
Co-existence	-2	-1	0	+1	+2
Replaceability	-2	-1	0	+1	+2

7. Microkernel

7.1. How do you grade your familiarity with the **microkernel** pattern?

1	2	3	4	5
None	Poor	Average	Good	Excellent

7.2. Reliability

Maturity	-2	-1	0	+1	+2
Fault tolerance	-2	-1	0	+1	+2
Recoverability	-2	-1	0	+1	+2

7.3. Usability

Understandability	-2	-1	0	+1	+2
Learnability	-2	-1	0	+1	+2
Operability	-2	-1	0	+1	+2

7.4. Maintainability

Analysability	-2	-1	0	+1	+2
Changeability	-2	-1	0	+1	+2
Stability	-2	-1	0	+1	+2
Testability	-2	-1	0	+1	+2

7.5. Efficiency

Time behaviour	-2	-1	0	+1	+2
Resource utilisation	-2	-1	0	+1	+2

7.6. Portability

Adaptability	-2	-1	0	+1	+2
Installability	-2	-1	0	+1	+2
Co-existence	-2	-1	0	+1	+2
Replaceability	-2	-1	0	+1	+2

8. Reflection

8.1. How do you grade your familiarity with the **reflection** pattern?

1	2	3	4	5
None	Poor	Average	Good	Excellent

8.2. Reliability

Maturity	-2	-1	0	+1	+2
Fault tolerance	-2	-1	0	+1	+2
Recoverability	-2	-1	0	+1	+2

8.3. Usability

Understandability	-2	-1	0	+1	+2
Learnability	-2	-1	0	+1	+2
Operability	-2	-1	0	+1	+2

8.4. Maintainability

Analysability	-2	-1	0	+1	+2
Changeability	-2	-1	0	+1	+2
Stability	-2	-1	0	+1	+2
Testability	-2	-1	0	+1	+2

8.5. Efficiency

Time behaviour	-2	-1	0	+1	+2
Resource utilisation	-2	-1	0	+1	+2

8.6. Portability

Adaptability	-2	-1	0	+1	+2
Installability	-2	-1	0	+1	+2
Co-existence	-2	-1	0	+1	+2
Replaceability	-2	-1	0	+1	+2