MASTER'S THESIS
2015

Tuukka Haapaniemi

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Tuukka Haapaniemi

Maintainable architecture in project business analysis software

Master's Thesis
23.5.2016

Supervisor: Professor Eljas Soisalon-Soininen
Instructor: M.Sc. (Econ.) Jari Vanhanen

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Tuukka Haapaniemi		
Title:	Maintainable architecture in project business analysis software		
Date:	23.5.2016	**Pages:**	56
Professorship:	Software Systems	**Code:**	T-106
Supervisor:	Professor Eljas Soisalon-Soininen		
Instructor:	M.Sc. (Econ.) Jari Vanhanen		

This thesis is a case study of designing and implementing a domain centric, maintainable software architecture for Nova Servo Oy, a small software provider for project businesses. The company's Project BI product is a data warehouse solution, which integrates into the customers' existing software and utilizes the data produced in them. On top of this data warehouse, Nova Servo aims to build specific tools to aid the project business succeed in the form of analyzation, forecasting and other tools not currently available. These tools require an architecture, which should remain maintainable throughout the expected long lifetime of the software.

The traditionally used architecture is the layered architecture, which is based on the database, contains a middle business logic layer and has a user interface layer as the topmost layer. This presents problems with its database centricity. A new domain-centric architecture is proposed in the form of onion architecture, which leaves the database as well as other infrastructural concerns outside of the core of the application. The domain operations and rules are put to the center of this architecture. This reduces dependencies to the infrastructural concerns dramatically and makes the business logic completely independent. Several tools and frameworks facilitating the onion architecture are presented.

During this thesis process, a purchase item planning software was created using the designed architecture. We found the resulting software to be easy to modify and extend without breaking or having to change the existing parts of the software and as such conforming to the maintainability requirement. Some parts of the architecture could be simplified and altered to further increase maintainability, but this is left for future work.

Keywords:	software architecture, layered architecture, software maintainability, onion architecture, domain-driven design
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan tutkinto-ohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Tuukka Haapaniemi

Työn nimi:
Ylläpidettävä arkkitehtuuri projektibisneksen analysointisovelluksessa

Päiväys:	23.5.2016	**Sivumäärä:**	56
Professuuri:	Ohjelmistojärjestelmät	**Koodi:**	T-106
Valvoja:	Professori Eljas Soisalon-Soininen		
Ohjaaja:	KTM Jari Vanhanen		

Tässä diplomityössä esitellään Nova Servo Oy:lle suunniteltu ja toteutettu domain-keskeinen ylläpidettävä sovellusarkkitehtuuri. Nova Servo Oy on pieni ohjelmistotuottaja, joka tekee sovelluksia projektiliiketoiminnan tarpeisiin. Yrityksen Project BI-tuote on tietovarastoratkaisu, joka hyödyntää asiakkaiden olemassa olevilla sovelluksilla tuotettua tietoa. Tämän tietovarastoratkaisun päälle rakennetaan projektiliiketoimintaan erikoistuneita sovelluksia, joilla liiketoiminnan analysointi, ennustaminen ja muut projektimuotoisuuden erityispiirteet hoituvat auttaen asiakasyrityksiä toimimaan tehokkaasti. Näitä sovelluksia varten tarvitaan arkkitehtuuri, joka pysyy ylläpidettävänä koko sovelluksen pitkäksi oletetun elinkaaren ajan.

Perinteisesti sovellusarkkitehtuuri perustuu monikerrosarkkitehtuuriin, joka rakentuu tietokannan päälle, sisältää välikerroksena liiketoimintalogiikkakerroksen ja ylimmäisenä käyttöliittymäkerroksen. Tästä arkkitehtuurista syntyy ongelmia sen tietokantakeskeisyyden vuoksi. Uusi, liiketoimintalogiikkakeskeinen arkkitehtuuri esitellään sipuliarkkitehtuurin muodossa. Tämä arkkitehtuuri jättää tietokannan ja muut infrastruktuurilliset asiat sovelluksen ytimen ulkopuolelle. Liiketoiminnan operaatiot ja säännöt sijoitetaan arkkitehtuurin ytimeen. Tällä tavalla saadaan vähennettyä sovelluksen kytköksiä infrastruktuuriin ja pidettyä liiketoimintalogiikka täysin itsenäisenä. Työssä esitellään myös useita työkaluja, joiden avulla tällaisen arkkitehtuurin rakentaminen helpottuu.

Tämän diplomityön tekemisen yhteydessä rakennettiin hankintatehtävien hallintasovellus esiteltyä arkkitehtuuria käyttäen. Kehityksen aikana sovellus todettiin helposti muokattavaksi ja laajennettavaksi, eikä muutokset ja laajennukset rikkoneet tai vaatineet muutoksia olemassa oleviin osioihin. Täten ylläpidettävyyden vaatimukseen saatiin vastattua tehokkaasti. Osaa arkkitehtuurista olisi mahdollista vielä yksinkertaista ja muokata edelleen helpommin ylläpidettäväksi, mutta sen toteutus vaatii jatkokehitystä ja lisätutkimuksia.

Avainsanat:	Ohjelmistoarkkitehtuuri, kerrosarkkitehtuuri, ohjelmiston ylläpidettävyys, sipuliarkkitehtuuri, domain-keskeinen suunnittelu
Kieli:	Englanti

Acknowledgements

I would like to thank Nova Servo Oy for the opportunity to work on this interesting topic and complete this thesis with a concrete problem. I would also like to thank my supervisor Eljas Soisalon-Soininen for the insightful comments and critique and especially the debates regarding the structure of the thesis. Furthermore, I would like to thank my instructor Jari Vanhanen for helping me keep the focus correct. I would like to thank my parents for believing in me all my life. Most of all I would like to thank my beloved fiancée Janni, who encouraged and pushed me forward to complete the thesis. During the making of this thesis, she gave birth to our firstborn son, Eetu, whom I also want to thank for bringing such joy to our lives.

Table of contents

Abbreviations

API	Application Programming Interface
BBoM	Big Ball of Mud
CQRS	Command Query Responsibility Segregation
CQS	Command Query Separation
CRUD	Create, Read, Update, Delete
DDD	Domain Driven Design
DI	Dependency Injection
DIP	Dependency Inversion Principle
DTO	Data Transfer Object
ETL	Extract, Transform, Load
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol over SSL
IoC	Inversion of Control
JSON	JavaScript Object Notation
MVC	Model View Controller
RDBMS	Relational DataBase Management System
REST	Representational State Transfer
SaaS	Software as a Service
SSL	Secure Sockets Layer
UI	User Interface
WCF	Windows Communication Framework

Chapter 1

Introduction

Even though software maintenance takes more effort than any other phase in software engineering, it seldom gets respective amount of attention [1]. Software architecture is described by Shaw and Garlan as “the overall structure of the software and the ways in which that structure provides conceptual integrity for a system” [2]. As such it is one of the key aspects that enables or inhibits software maintenance. The most commonly used architecture is “The Big Ball of Mud”, or BBoM, as described by Foote and Yoder [3]. Such architecture is seldom chosen, but it usually evolves over time without the needed thought given to it. This leads to more and more complex software which is hard to understand and even harder to change. In this thesis, however, we ignore such unstructured architectures, and concentrate on the more robust ones. Even those pose challenges concerning maintainability and that is what we want to address. Unless the architecture is explicitly chosen, expressly communicated and strictly adhered to, there is a risk of architectural erosion over time, which further reduces maintainability [4].

Software architecture and design often reflects the bottom up design style most comfortable to technical software developers. This commonly leads to an architecture centered on the database which in turn creates unnecessary coupling of the layers, thus making the upper layers’ data structures both look like and change according to the database and its changes [5].

Problem domain

Nova Servo Oy is a software company working in and for the project industry. Project business is in many ways similar to product business, but some key differences set it apart. Because of these differences, the same tools, such as

Enterprise Resource Planning software (ERP), do not quite match the requirements of project business. This usually forces project business users to adapt their core work to the ways of the software, rather than the business needs directing the way the software should support the business itself. Nova Servo Oy aims to provide these project specific, tailored tools to the project business.

The aim of this Master's Thesis is to give Nova Servo Oy an architectural design proposition that will lend itself to all requirements defined in the later chapters. The scope is narrowed to project management software on the web platform, though the resulting application may be applicable in other platforms as well. We will discuss the different architectures and their effects on developing and maintaining software as well as tools to implement them. Usually the architectural decisions are made based on the organization of teams that are doing the development, as it is also a way of distributing work amongst different teams and developers [6]. In Nova Servo Oy, however, the team is currently very small, but growth is anticipated. Thus, the architecture should be simple enough to merit itself with the small team, but robust enough to endure even with a development team of dozens. Hence the architecture cannot be designed based on current situation alone and the future organizational structure being unknown, it cannot be utilized, either. The product under development is planned to be centric to the business of the clients, and thus the expected lifetime of the product is between seven and ten years, which in turn poses stress on maintainability, which consequently is one of the key requirements of the architecture.

Structure of the thesis

After this first introductory chapter the second chapter defines the basic building blocks used in common software architectures as well as the commonly used architectures, especially the layered architecture. A revised version of the traditional layered architecture, called onion architecture, is presented as a modern, more maintainable choice. Further, the two architectural patterns are compared and the key differences emphasized.

The third chapter presents programming patterns that support and enable efficient use of the onion architecture. In the fourth chapter, the problem domain is presented in more detail and the research questions are defined. In the fifth chapter the design and implementation is covered from both the research point of

view as well as from the proposed new architecture’s point of view. The new architecture is thoroughly presented and then the practical applications of the new architectural model are considered.

The Sixth chapter provides an in-depth analysis and evaluation of the new architectural model and describes how it relates to the research questions. The results are reviewed in light of the literature on the subject and the quality of research is assessed. The final chapter gives the conclusion to this Master’s thesis.

Chapter 2

Software Architectures

Software architecture can be defined in a multitude of ways. Traditionally it can be seen as “the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.” [6] A resulting piece of software does not necessarily need an explicit architecture, though it can be argued, that implicitly one always emerges [3]. The usefulness of architecture comes from the software design process itself [1]. It gives the developers the tools needed to analyze the design, consider alternatives and reduce the risks of the software development.

As different architectures and architectural styles are abundant, it is not possible or feasible to describe them all. This thesis focuses on the different forms of layered architectures. As maintainability is a major concern in a long lived computer software, that is also discussed.

2.1. Maintainability

Maintainability is one of the most important qualities of software development [7]. A proven principle to make software more maintainable is to follow the SOLID-principles [8]. These principles have become a de facto standard of good software architecture traits. The principles have also been proven to reduce coupling and increase cohesion [9]. The SOLID-principles consist of five different principles, from which the SOLID-acronym is derived: The Single Responsibility Principle, The Open Closed Principle, The Liskov Substitution Principle, The Interface Segregation Principle and the Dependency Inversion Principle. These principles are discussed in the following subsections. Adhering to

the SOLID principles gave Harmeet et al. [9] a 69 % reduction in coupling and an increase of cohesion of 29 % when compared to a sample program design without using the principles.

Single Responsibility Principle (SRP)

The single responsibility principle states “A class should have only one reason to change.” [10] If a class has more than one reasons to change, the responsibilities of the class can be seen as diversified. This leads to a problem: A change in the area of one responsibility will affect the area of another, often completely orthogonal responsibility. In a worst-case scenario, that change could break the other responsibility altogether. To avoid this, a class with multiple responsibilities should be refactored to smaller classes, each with its own responsibility.

Open Closed Principle (OCP)

This principle states, “A module should be open for extension but closed for modification.” [8] In effect, this means that a module should be designed so that extending it and thus changing its operation is possible without changing the module itself. This makes the module open for extension. Getting a module to be closed for modification requires the module to be isolated from the implicit implementations, and instead depend on an abstraction, which in turn can be extended into multiple implementations, as described before. This way the module is closed for modification in the event of a change to a single implementation or an addition of a new one.

Liskov Substitution Principle (LSP)

The Liskov Substitution Principle states “Subclasses should be substitutable for their base classes.” [8] This is seemingly easy to achieve, as a compiler of an object-oriented language will take care of this: a class inheriting from a base class will automatically inherit all its properties and functions and as such, the inheriting class can be passed to a function or a constructor accepting the base class as a parameter. The problematic violations of the LSP are subtler: A class inheriting the base class can disregard a parameter in an overloaded method, thus not explicitly honoring the contract of the base class. This can lead to very complicated and hard to find problems, as the user of the base class cannot, and should not, know all the details of all the inheriting classes. Even more so, when

the inheritance hierarchy can be multiple levels deep. Put succinctly by Martin [8], “derived methods should expect no more and provide no less”.

Violations of the LSP are generally detected so late in the development, that a complete redesign of that portion is not feasible. To circumvent the design flaw, generally an if/else structure is created to work differently with different types of parameters. This leads straight to the problems of the Open Closed Principle, as a new extension of the base class would require a change in the method, thus violating the OCP. In essence, a violation of the LSP is a latent violation of the OCP.

Interface Segregation Principle (ISP)

The Interface Segregation Principle states “Many client specific interfaces are better than one general purpose interface” [8]. This means that a class being used by multiple client classes should be split into client specific interfaces, all of which the class would implement, rather than have a big common interface for all the clients to use. The reasoning is that this way a client will only depend on those methods they actually need, and it will not be coupled unnecessarily to the methods it is not interested in. As an additional advantage, a change into a requirement of one client will only affect that client’s interface, and not break the other clients. Of course, there can be many clients of a similar type, that share the requirements of an interface, and these should share the interface, too. The distinction is applicable when the clients have varied requirements through the interface.

Dependency Inversion Principle (DIP)

In procedural design dependencies flow from the top to the bottom [8]. The higher level objects depend on the lower level objects, that contain the details of the implementation. This means that the higher level objects are tightly coupled to the details of the implementation, when they really should work on a more abstract level of detail. This can be achieved by adhering to the Dependency Inversion Principle (DIP), which states that one should “Depend upon abstractions. Do not depend upon concretions” [8]. When an abstract class or an interface is introduced in between the implementation and the higher level component, the dependency can be inverted to flow from both of these towards the abstraction. This is

required, but not enough [11]. The ownership of the abstraction is very important in this principle. If the implementing class owns the abstraction, any changes to the implementing class might change the abstraction and cause a rippling effect upwards to the dependent class, which is fundamentally wrong. The abstraction must be owned by the higher-level component. This way, when the higher level component changes, it may change the abstraction it owns and the rippling effect will be down to the implementing classes as it should be. This completely decouples the higher-level component from the implementation details of the concrete class. As a bonus, this also allows us to better confirm to the Open Closed Principle by providing these abstractions as the extension points of the architecture.

2.2. Layered architectures

The most common software architecture, Big Ball of Mud, aside, Layered Architecture is the most used [3]. This is a higher-level pattern, where layers are a means of separating concerns to disparate logical containers [3, 12]. By strictly defining and adhering to the boundaries between the layers, other layers can be shielded from the changes of one layer. A good measure of a successful layering is to detect the change rate of software components on different layers: if the components on the same layer change at the same rate, the layering is most likely successful [3].

Layers should not be confused with tiers [12]. A multitier application is one whose layers are spread out onto separate physical servers from an infrastructure viewpoint. Contrary, a multilayered application can be deployed on a single physical server, or for instance, a workstation, thus being single tier. A traditional client-server architecture consisting of a web server, and a web browser as the client. Together these form a two-tier application [13].

Most common layers used are Presentation Layer, Application Layer, Business Layer and Data access Layer, or variations thereof [14]. The number of layers also varies from two in the simplest programs to half a dozen or more in the more complex enterprise solutions.

The following sections introduce the traditional layered architecture with its strengths and weaknesses and a variation of it called the onion architecture and their comparison.

2.3. Traditional layered architecture

The traditional layered architecture is the one most developers are familiar with. The basic idea is to rigorously separate the components of the software on named layers starting from the Presentation Layer or User Interface (UI) Layer, going through Business Layer all the way to Data Access Layer which is commonly the lowest layer communicating with i.e. a database or a web service [12, 15]. The amount of layers and their names varies based on the particular software and its architectural specifics.

The defining characteristic of this architecture is that each layer is coupled only to layers below it with two distinctive styles: A Strict Layers Architecture, where coupling is only allowed to the layer directly below and a Relaxed Layers Architecture, where coupling is allowed to any layer below [15].

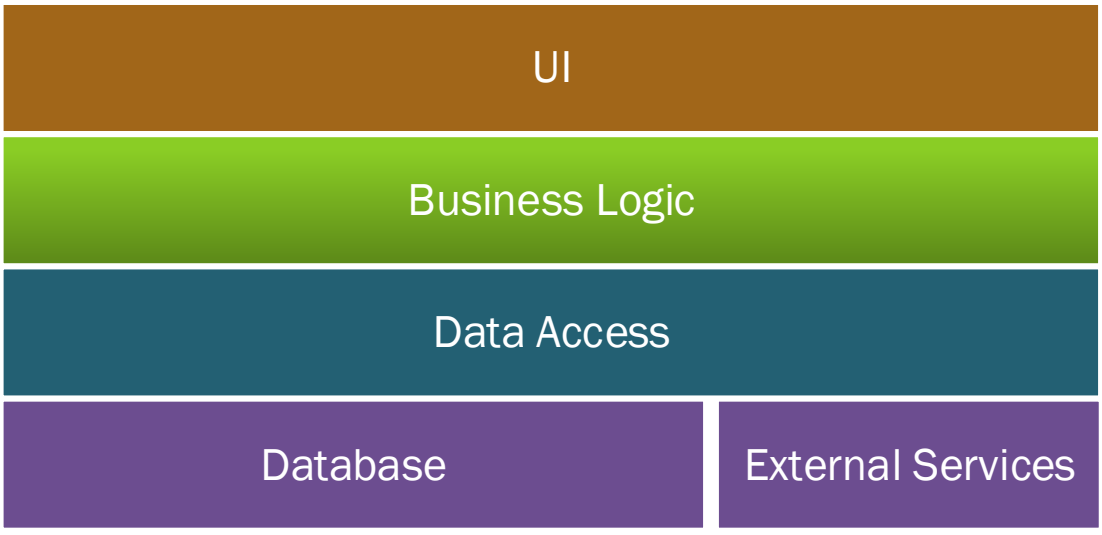

Figure 1- Layered architecture represented as boxes on top of each other

Presentation layer

The presentation layer or User Interface (UI) layer is the layer the user sees and interacts with [16]. Here the user interface components are defined, their positioning, functions texts, images and icons and all other visual elements. The interactivity that happens when a user interacts with the UI is also defined here. If a specific pattern for the user interface is used, such as the Model–View–Controller –pattern, the implementations of all those components also resides here in the presentation layer.

This layer uses the underlying layers either explicitly or implicitly to fill the user interface with the actual data of the application for the user to view. Likewise, when a user edits the data either directly in a CRUD-style or indirectly via tasks, the changed data or operations are conveyed through this layer into the lower layers. Ideally, this layer contains no application logic, as that should be the responsibility of the lower layers.

Business layer

This layer is the most important layer of the software [16]. It is intended to describe the business critical rules and methods of the problem domain. These rules act on the data of the application based on the predefined rules that are defining and critical to the business in question. Alongside they ensure the data consistency and validity.

Optionally this layer may contain an application façade that provides a simplified interface to the clients of this layer. This component is usually in charge of the unit of work, which dictates the transaction boundaries of the operations.

Data access layer

The lowest layer encapsulates infrastructural services accessing for instance the database or web services [16]. From the application perspective, the database and web services can both be seen as storage mediums. The function of this layer is to abstract the technical details of accessing the storage mediums away from the higher layers. This includes the adapters needed to connect and communicate with the storages as well as the data model used in the storage, which often differs substantially from the model most usable to the application due to data normalization.

Challenges

The problem with layered architecture is that everything is eventually coupled to the data access layer and if that changes, the changes are generally reflected on all the other layers, which is a clear sign of unnecessary coupling [15]. Looking from another perspective, we can see that the architecture is data-centric: Everything is built on and around the database. An example is the addition of a new field in the database, which needs to be shown in the user interface. To get the value from the database to the user interface the value has to be added to all the other layers in between. Minor changes such as the change of a field or table in the database can of course be shielded away with the traditional layered architecture, as that will only affect the mappings in the data access layer. The real problem is still the unnecessary coupling that this architecture creates. The most critical problem is the coupling of the business logic layer to the infrastructure: Business logic needs to know about infrastructural concerns such as the database, and it gets more than one responsibility, inherently violating the Single Responsibility Principle.

Testability is also a big challenge in a traditional layered architecture. As the business logic layer is dependent on the data access layer, the data access layer has to be completely mocked in order to be able to test the business logic rules. This is a major caveat as the business rules are the ones that should be thoroughly tested. Being the difficult discipline that mocking is, the testing becomes harder than it should.

The traditional layered architecture can also be depicted with concentric circles as in Figure 2. This representation highlights the data-centricity of this approach and connects this architecture to the architecture of the next paragraph. The dependencies in this drawing are always to the center. For instance, with strict layering, UI can only access business logic, whereas in relaxed layering UI can access both business logic and data access. Business logic can only access data access and data access is completely independent in both layering styles.

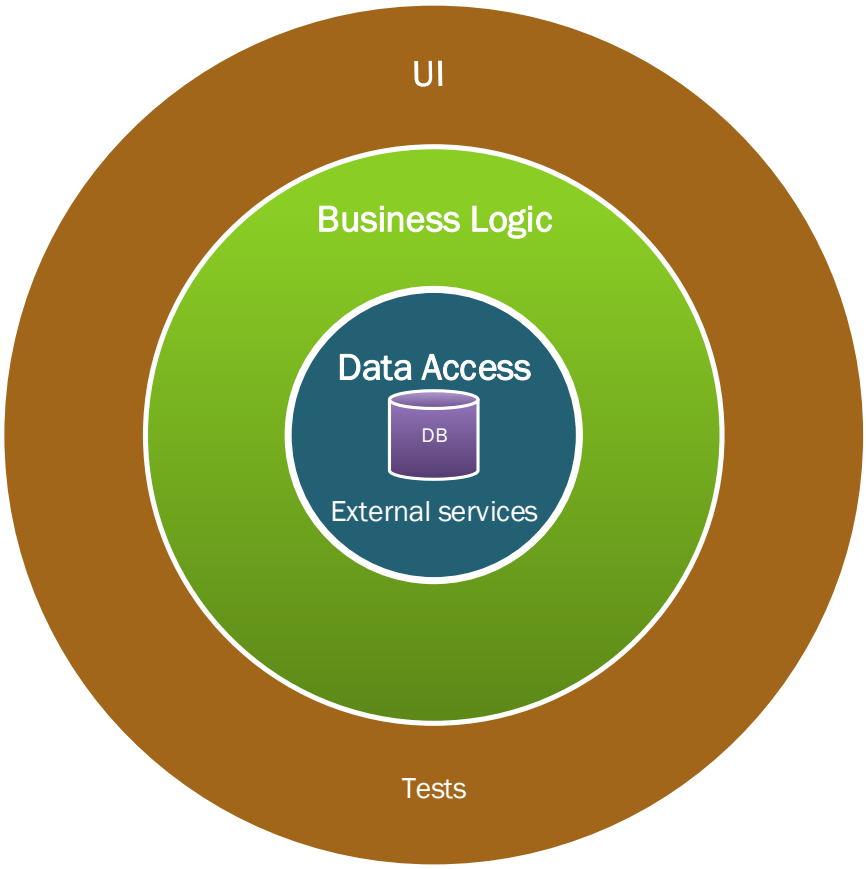

Figure 2 - Traditional Layered Architecture depicted with concentric circles

2.4. Onion architecture

Jeffrey Palermo has introduced a new architectural style called Onion Architecture [5]. Palermo stresses that the architecture and the methods themselves are not new but that the onion architecture as a concept and term is. Palermo coined the term to allow experts to discuss the possibilities and problems of the paradigm better. This architecture is commonly misinterpreted as a synonym for Ports and Adapters or Hexagonal architectures, which are synonymous between themselves, but actually a superset of the onion architecture [17].

Onion architecture is very much a result of the Dependency Inversion Principle (DIP) described in chapter 2.1 brought into the system architecture level. In the

onion architecture the dependencies are reversed so that the database and code used to access it, very much an implementation detail, is now outmost as seen in Figure 3 as opposed to it being at the very center in the traditional layered architecture. As in the traditional layered architecture, the dependencies are from the outer layers to the inner layers but never from the inner layers to the outer layers. Thus, the domain layer and the application interface layer become independent of the UI or the data access, or more generally, the outmost ring of the onion. The result of this is that when there is a change to these components, UI, Database, Web Services, messaging infrastructure etc., and as these are the components changing most often, the changes are not reflected on the core domain in any way [5].

Palermo defines the key tenets of onion architecture as follows:

1. The application is built around an independent object model
2. Inner layers define interfaces, outer layers implement interfaces
3. Direction of coupling is toward the center
4. All application core code can be compiled and run separate from infrastructure

The first and last tenet are heavily related. The object model in the core of the application must be independent of any infrastructural concerns. It is also the starting point of the application development. It is where the most value can be found and the area distincting the application from other applications of the same field [15].

By the second and third tenet, an interface definition must reside in a layer inner to where the implementing class resides. The interface should also be based on the business needs, not the technical aspects. Defining these interfaces first alongside the core is also a great method to keep the technical aspects from affecting the design of the interfaces and the core as a whole.

In the onion architecture, the domain is easily tested. The domain in the core of the architecture contains all the business logic and business rules. As the direction of dependency is towards the center, the domain is independent of any implementation details or frameworks, like the data access. Because of that, the

domain and its business rules can be easily tested with no need for mocking or similar tools.

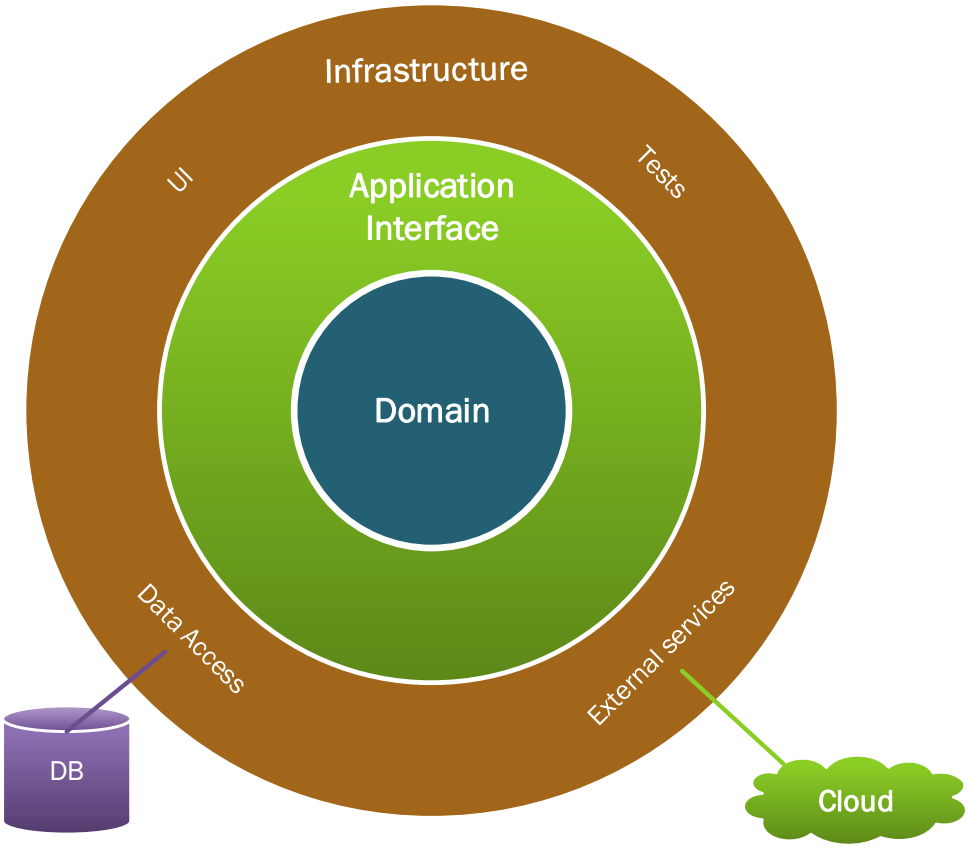

Figure 3 - Onion Architecture depicted with concentric circles

2.5. Key differences in layered architecture and Onion architecture

Eric Evans writes his view of the software layers concerning Domain Driven Design, introduced in chapter 3.1 as follows:

> "Isolate the expression of the domain model and the business logic, and eliminate any dependency on infrastructure, user interface, or even application logic that is not business logic. Partition a complex program into layers. Develop a design within each layer that is cohesive and that depends only on the layers below." [14]

Looking back at Figure 2 we can see that the business layer is between the UI layer and the data access layer, and coupled to the data access layer. Even the UI layer is coupled to the data access layer, but only transitively [5]. As data access is a form of infrastructure, this is in clear contrast with what Evans calls elimination of dependency on infrastructure.

As we contrast the findings of the traditional layered architecture to the offerings of the onion architecture, we can see that the onion architecture adheres to Evans' principles, as the domain layer is strictly independent. It depends only on itself and not on any changeable infrastructure or UI. The core of the domain depends on the interfaces defined in the core itself, and the implementers of those interfaces depend on the core. Thus, the implementations are free to change as much and as often as required with no effect on the core.

Regarding testability, things are quite a lot simpler in an onion architecture than in the traditional layered architecture. The traditional layered architecture needed to have the lower layers mocked in order to test the most crucial business components, whereas in the onion architecture the domain is independent by design, and as such, no mocking is needed. This is quite a big advantage for the onion architecture. Testing can also easily be done on multiple levels: The domain can be tested on lower level unit tests, whereas the application interface and even the infrastructure can be tested on more complete integration tests.

Chapter 3

Programming patterns for Onion architecture

Onion architecture is a way of structuring code. It does not necessarily need any specific patterns or tools to implement, but certain patterns make implementing and maintaining it much easier. As maintainability is high on the requirements list, any pattern, tool or framework that can potentially increase it is welcome. Thus, these are looked into in this chapter.

3.1. Dependency Injection Pattern

For this style of architecture to work, we need to employ a special design pattern called Dependency Injection (DI) Pattern. This differs from Dependency Inversion Principle introduced in section 2.1 in that the DI pattern is a concrete technical way of supplying the dependencies to the callers, whereas the Dependency Inversion Principle is more of a philosophical perspective and answers to the question: who is in charge of the abstraction of the implementation.

The basic idea in DI pattern is that dependencies, i.e. classes and components, required by the caller, are not instantiated within the caller, because this creates unnecessary coupling by tightly coupling the caller to the dependency [18]. The dependency cannot be replaced without modifications to the caller's code. Instead, the caller should only depend on the abstraction of the implementation: an interface or an abstract base class. With DI, the work of initializing the dependent class, being any class conforming to the abstraction, can be externalized from the caller. That is, the dependency can be injected form the outside.

The dependency is injected into the caller in the form of the abstraction by the class of method calling the caller, i.e. A calling B which depends on C, where A initiates an implementation of an abstraction C and provides it to B to be used. There are three forms of injection available in the common object oriented languages: Constructor Injection, Property Injection and Method Injection. In Constructor Injection, the constructor has a parameter of the type of the abstraction. Similarly, in Property Injection, a class has a publicly settable property of the type of the abstraction, and in Method Injection, the method to be called has an extra parameter of the type of the abstraction.

The most robust and flexible way to implement the dependency inversion principle is to use a Dependency Injection container specifically designed for this task. It is possible to implement DI pattern without any third party tools by injecting the dependencies manually all the way from the higher levels. This style leads to very long hierarchies of dependencies injected from one level to the other and is often cumbersome, leaving the maintenance cost quite high, especially considering that most DI-containers are completely free. There are many containers available and most provide similar basic features, so choice is much a question of preference. Some more advanced scenarios may have such specific requirements, that not all containers can fulfill them.

Using a DI container, the dependencies are usually injected by Constructor Injection, although other forms can be used. With Constructor Injection, the DI container can be left in charge of the instantiation of the whole object graph. On the highest level, the first dependency has to be resolved manually, or through some special bootstrapping. After this, when the dependency has its own dependencies and so forth, the DI container will resolve the dependencies based on its configuration. Adding another dependency to a class in any point of the object graph is extremely simple: just add a new argument to the constructor and a new configuration to the startup or bootstrapped class telling which implementation to use. Then, the DI container will take care of instantiating the implementing class, no matter how deep in the object graph it resides.

3.2. Domain-driven design (DDD)

Software design and architecture is commonly based on the data of the problem domain entities, as they are stored in a relational database [1]. Database design has been highlighted as the basic starting point, as the database is the hardest to change. This leads to a data-centric design, which is natural from a developer's standpoint but not necessarily from the architectural or especially the usability point of view. A more modern approach highlights the problem domain where the user works and the tasks associated with it [15]. There are complete frameworks designed to address this problem, such as Business Driven Development but their use falls outside of this thesis [19].

According to Eric Evans, Domain-driven design (DDD) is a philosophy [14]. He presents design practices, techniques and principles aimed at creating sustainable software in complex domains. The reason why the domain is so essential in software development is that software relates to some interest or activity of its users. This activity or interest is the domain, the core of the application, the defining characteristic that makes the software do something useful or interesting. The domain can represent the physical world, such as workers creating new ships on a shipyard, or something intangible, such as measuring monetary currants and values of the same ships construction. If the domain and its rules are not well understood, the software may end up doing something contrary to the customer's needs and essentially fail the whole project. Thus, the domain is of the utmost importance.

For the software to be successfully constructed, the domain understanding alone is not sufficient [14]. The domain and its rules must be clearly conveyed to the developers of that software. This requires the presence of that domain's experts in the development. The usual depiction of a software development project without DDD begins with a requirements definition phase from which requirement documents are created by the domain experts [1]. These are then forwarded to the development team, which has no domain experts of their own or seldom even direct communication channels to the customer. This creates an unavoidable interruption in the flow of information of the domain, especially when the domain expert uses terms of the business, but developers often end up using their own, more generic terms, that are more suitable for the development phase. The created

software can end up being unusable for the customer, because the real need is not conveyed all the way to the development team.

Domain-driven design advocates a way of working in which the domain experts are a central part of the development team [14]. Together with the developers, they bring the focus on the core domain, define clear boundaries on different bounded contexts and develop a ubiquitous language, so that the team can talk with the same terms throughout the project with little or no confusion of the terms' meanings.

Focus on the core domain

According to Evans "In designing a large system, there are so many contributing components, all complicated and all absolutely necessary to success, that the essence of the domain model, the real business asset, can be obscured and neglected" [14]. The core domain should clearly be identified and understood by everyone in the team. To make the core domain an asset for the software and for the client, the most skilled developers together with the most skilled domain experts should create it. The core domain also often contains the business rules that change most often and contain most complex logic. If this logic is not correctly understood, it becomes hard to change and maintain. If a software loses its ability to conform to the business needs in its core, it is not likely to succeed.

Domain model

A domain model is a tool used to present and structure the knowledge of the domain [14]. The model represents the real world entities and rules of the domain. For the domain model to be of most use, it is not necessary or even beneficial for it to be in a 1:1 relation to the real world. A simplification is generally preferred. This simplified model can put the correct weight on the most critical parts of the domain, essentially focusing it on the domains core problem. The domain model is not a diagram of just the knowledge of the domain. "It is a rigorously organized and selective abstraction of that knowledge", which can be conveyed by a diagram, by written text, by a multitude of development tasks or by the resulting code of the software [14].

Ubiquitous language

The domain experts seldom understand the technical terms developers tend to use [14]. Likewise, the developers might not be familiar with the domain terms the domain expert uses when discussing the software with the customer and when relaying the customer's needs to the developers. This creates a disparity of terms and languages, which easily leads to miscommunication during the development phase. This might be fatal for the software, as developers end up creating something else that the customers or the domain expert intended. To remove this risk and to reduce the effort of the conversations, where everything has to be "translated", a common language should be defined. One that both the domain experts and the developers learn to use in speech, in diagrams and especially in code. The code is always the most current and most important aspect of the domain, being the end result of the whole process, so it should reflect the language as well. This language is called ubiquitous language.

Bounded context

Enterprise software is usually very large and requires a large team or more than one team to develop and maintain. Trying to model such a big software with a single model is not practical [14]. Making a distinction between two separate pieces of software is quite easy: they are developed separately and when they are integrated, the models are different and the data moving in between needs to be converted from one systems format to the other. Making a similar distinction within a single software is more challenging. Code reuse is a very common method, which used thoughtlessly may produce one model used in different parts of a software for different purposes.

Consider an order in an e-commerce site. The viewpoint to the order is very different from the client's side compared to that from the supplier's side. The client is interested in the placing of the order, the price of the order and the delivery date of the order. The supplier on the other hand will be hard pressed to fulfill the order by thinking about the stocked amount, the shelf of the items in the storehouse and the wrapping and the postage of the package. These are very different views, or contexts, to the same order requiring different business logic and rules to complete. A single model will bloat and become cumbersome to use. The model will most likely have to compromise for the sake of the other context.

If the given example were strictly and explicitly split into two separate and clearly bounded contexts, the problem would fade away. The customer side of the e-commerce would have its own model with its own logic, and the order fulfillment side would have its own with different logic. Compromises are no longer necessary. Two distinct bounded contexts are created and both contain an order model, which applies within that bounded context. The order model has no value in the conjoining context. When these bounded contexts are integrated, they exchange messages regarding the same physical order, but both apply their own rules to the order model inside their contexts.

The ubiquitous language can and should be defined per context [14]. Building on the order example, both the client and the supplier can be thought of as having to send the order in the end. When the client sends the order, the supplier will get it to fulfill. When the supplier sends the order, the items are sent through a shipping company to the client. The same verb, “send”, has two different meanings. This is an important separation, and the bounded contexts give clear boundaries within which the ubiquitous language is defined and used, which in turn gives more power to the model. A new term is not needed just for the sake of distincting the sending in the two different contexts. More concisely, “Model expressions, like any other phrase, only have meaning in context.” [20]

Maintainability

Domain-driven design is also a very powerful tool in increasing software maintainability [7]. A software designed with DDD has an increased clarity in the required business rules. Design starting with the domain also forces the implementations to adhere to the domain specification. Likewise, DDD provides a framework with which the business logic is easily placed in the correct classes and components, which decreases the duplication of logic. Infrastructural concerns are also left out of the domain, which enables them to change and evolve without affecting the core domain.

3.3. Task based development

Quite commonly, software is built as views to the database tables and the data is manipulated though editing, creating and deleting the rows through these views.

Naturally, there are multiple tables relating to a single view via foreign key relations depicted in the UI as drop down list or similar, but the general idea remains. These kinds of user interfaces are referred to as CRUD-UIs, the acronym being composed of the database operation terms: Create, Read, Update and Delete [21]. In many cases, the resulting user interfaces meet the simpler requirements sufficiently. A good example is an administration screen, where data of a more static nature is managed. There seldom is complex business rules behind the data manipulation and as such, the CRUD style user interface suffices. This style, however, is also common in the more complex scenarios.

Usually a more complex operation becomes hard to understand and the usability greatly suffers if implemented in CRUD style. Complex workflows will especially become burdensome and error prone as the users need to know and remember which entities to edit, how, and in which order [21]. The possibility of bad data forming in the database is significant. The greatest problem still is the loss of the user intention. Looking at the database after a user has completed a series of updates, we cannot tell what the user intended to do and why. We only see the end result, i.e. the resulting state of the data. This will not allow us to capture the tasks users complete and hone the processes and the software based on the usage data. Another problem yet is that the lost intent, that is, not knowing the verb the user wanted to act with, will not allow us to use all the tools of domain-driven design [21]. This all changes when we introduce Task Based Development.

A task is “An activity which when undertaken results in a change of state in a given domain and satisfies a main goal” [22]. When using task-based development, we aim to identify the different goals the user wants to achieve and the tasks required to achieve them. This makes the user’s intent more explicit and hence complements domain driven design.

A task-based UI, also known as an Inductive UI, will concentrate on the user’s point of view and disregards the database structure. In CRUD-style development it is common to see a seemingly one operation split into many when executed with software, as discussed before. A task-based UI looks different and collects all the elements needed for that single operation into a single view, which explicitly describes the user’s intent and gathers all necessary data concerning that operation.

In the inner layers of the architecture, the task-based development also shapes the classes and methods [21]. Here, instead of the simple Data Transfer Object (DTO) containing the changed row's values being given as a parameter to the CRUD style operations, a more powerful command object is given as a parameter to a new operation, which corresponds to the actual intent of the user. This command object encompasses all the data from the task based UI necessary to complete the operation, but nothing more. As a side note, the intent of the user can be stored and the usage of the software can be analyzed with the stored data. The greater advantage, however, comes from the shape of the objects handling the command, that is, the user's intention. As the intent, and thus the command is very domain specific, and expressive in their own right, the command handler has all the necessary data it needs to apply all necessary business rules of the domain and complete the command. This also tracks back to the DDD, as generally the command and the domain specific use case correlate strongly.

3.4. Command and Query Responsibility Segregation (CQRS)

Business software is useless without data. There are some scenarios, where a piece of software only needs to read data, but not write. Even rarer are the scenarios where only writing of data is required, but not reading. Usually software needs to both read and write data to give some meaningful business value. In this scenario, the same object is often used to both read the data and write the modified data back [21]. With simple requirements this suffices, but the more complex the requirements and the more components the software needs, for instance APIs for external use, the more the objects responsibilities become twofold, as the requirements for reading are different from the requirements of writing. This also clearly violates the single responsibility principle.

The segregation of the read and write requirements affects three different aspects of the program design [23]. First, the data transfer objects (DTO) that are used to transfer the data across software boundaries, be it public interfaces between layers or APIs for external use, are usually different for read and for write. When writing, the referenced entities are only required to be presented by their identifiers. Often when reading, a name of the entity will also be needed for

presentation to the user, and to save one round trip, it is included in the response DTO of the read call. Trying to combine these two objects into one will result in an easily misinterpreted DTO, rising questions such as what will happen in the write, if a referenced entity is passed with an identity-name pair different to what is persisted? The second aspect is the architecture and especially the complexity of the software. When reading, performance is usually more critical than when writing, particularly as reading often affects multiple records and writing seldom more than one. Reading is also usually free of extensive validation and enforcement of rules, business or otherwise, whereas writing has to enforce them. Thus, it becomes very cumbersome to try to use the same architecture and the same objects for both reading and writing the same entity. The third aspect is the comprehensibility of the software. If a single function will both read data and write data, the caller will have to resort to the documentation to understand what side effects to the state of the data the read operation will cause, provided such documentation exists.

To address these concerns, Meyer has proposed a solution where the operations are clearly separated and their purposes defined as follows: “Functions produce results yet don’t affect state” and “Procedures explicitly affect state but don’t provide a result” [23]. In Meyer’s solution the read operations, the functions, and the write operations, the procedures, are separated into different programmatic functions within the same object. This addresses the comprehensibility aspect introduced earlier. The caller of a function can rely on the fact that the function will produce a result, but will not have any side effects, that is, will not alter state. A procedure, on the other hand, will always alter state, but never produce any results. The signature of the programmatic function will clearly convey, which is in question.

Taking CQS a step forward, more recently, a term Command and Query Responsibility Segregation (CQRS) was coined by Young: “Command and Query Responsibility Segregation uses the same definition of Commands and Queries that Meyer used and maintains the viewpoint that they should be pure. The fundamental difference is that in CQRS objects are split into two objects, one containing the Commands one containing the Queries.” [21] Here, commands are essentially same as Meyer’s procedures and queries same as functions. This addresses the rest of the problematic aspects noted earlier. The software

architecture can be defined separately for both sides. The query side, that is, the read side, can be designed for effective reading of multiple records, whereas the command side, that is, the write side, can be more complex, perhaps done with domain driven design principles and enforce the domain rules with a more intricate design.

CQRS is often associated with Event Sourcing, which is an architecture, where instead of storing the current state of the objects, all the commands executed are stored. As a command is executed, a separate, read storage is updated eventually to represent the current status of the system. Many articles claim, that CQRS requires event sourcing, but that is not the case. As Bogard states, “event sourcing is a completely orthogonal concept to CQRS” [24].

Chapter 4

Project business analysis software

Project business is inherently different from traditional trade or product business [25]. In project business, the success of the business is measured not only on the company level, but also on the project level. Building a project portfolio and learning from the successes or failures of past projects enables the business to become more competitive and profitable, or at the very least, avoid bankruptcy. Many complete ERP solutions offer very good tools for the businesses working in the traditional trade and manufacturing, but they often lack the intricacies of project business. Projects either are completely absent, or added as a later feature, thus not really accounted for in each required aspect. Software solutions specifically aimed at project business are scarce.

4.1. Description of the problem domain

Nova Servo Oy functions in project business domain. Our clientele is almost solely corporations, which complete the tasks given to them by their clients as projects. The specificities of project style working are also the core functions in which we at Nova Servo Oy excel and from which we strive to create the best possible software to aid our clients in completing their projects as effectively as possible.

Our main product, Project BI, is a tool for the project managers and the management of the company. The main functions of the product are analyzation, reporting and forecasting of the customers' project data. Certain project specific tools not currently available on the market in sufficient quality or extensiveness, such as purchase item planning, are also under development.

Project BI is based on a data warehouse database, which is loaded with the data of clients existing software solutions as a repeatable ETL process through software integration. The main advantages of this approach are twofold. First, the clients need not change the software they currently use to run their business, be it ERP, sales, cost calculation or any other area of business. The client does not need to adapt their processes to the new software and the users can use the existing software, the one they already know how to use effectively. As the tool is targeted for the project managers and the company management, they are the sole roles required to learn to use new software. Second, all the data produced by the business are collected at one location, the data warehouse, in a uniform format. Thus, analyzing the data becomes easy. The uniform format and an externalized data warehouse also facilitate the company to change any or all of their data generating tools, including the ERP, and still maintain full history and comparability of data even across the disparate systems. The data warehouse solution is, by DDD terms, the core domain of our product.

The project specific tools, e.g. purchase item planning, are designed and developed as modularized components. They are separated from the data warehouse solution into separate domains with regards to DDD. They are considered supporting domains, as they are something the clientele requires and wants, but are not the core of the business of Nova Servo. Should an existing product be found suitable to replace a supporting domain, we would rather integrate into that and concentrate on the core domain. These subdomains are also modeled as their own bounded contexts. Ubiquitous language of the domain is valid in that bounded context only, and the interfaces between the contexts are clearly defined.

4.2. Requirements

Project BI product is intended for business use and as such, businesses adopting it desire to use it for quite some time. The main requirements are, of course, the functional requirements, those, that offer value to our customers [26]. That is, provide them with a software that has all the necessary functions.

Non-functional requirements are those that do not directly add value to the customer, but that are required for the software to function, to be secure and to

remain usable for the remainder of the desired lifetime of the software [26]. Of the non-functional requirements, some are not at all critical in Project BI. While performance or scaling cannot be neglected, they are not something that require huge effort in this kind of software. Amount of users is very limited and mediocre response times are sufficient. Usability is already quite a lot more important as the topic of the software is very complicated, the software should not further complicate things, but rather help the user navigate in the masses of data. Security in business applications is always critical, though not on the level required in, for instance, banking applications. One non-functional requirement still rises above these all: maintainability. Maintainability and its subarea, adaptability, is an aspect through which all of the other, both non-functional and functional requirements, are easier to fulfill after the initial launch of the product. If the software is unmaintainable, the security is hard to patch, usability problems difficult to address and new features expensive to develop. Foote and Yoder define adaptability as one of the key requirements for a successful, long living software:

> "A system that can cope readily with a wide range of requirements, will, all other things being equal, have an advantage over one that cannot. Such a system can allow unexpected requirements to be met with little or no reengineering, and allow its more skilled customers to rapidly address novel challenges." [3]

Thus, maintainability is our main requirement and the focal point of this thesis. This combined with the proposed onion architecture leads to the research question of the thesis:

- Does onion architecture promote software maintainability?

The research method chosen for this thesis is a literacy review of the proposed components with which an architectural model is created following the empirical analysis of the proposed architecture through using it in a software project and evaluating the resulting prototype with respect to the literacy review.

Chapter 5

Design and implementation

This chapter provides insight into the technological choices made before starting the development of the Project BI product. The combined new architectural model based on the onion architecture is presented in detail and the resulting application of it to the software is described.

5.1. Technology stack

In the beginning of this thesis, some of the technology stack was predefined by the choices the company made in the beginning of the development of the Project BI software. Namely, Microsoft SQL Server was chosen as the Relational Database Management System (RDBMS) on which the software would run and which would supply the Data Warehousing functionality. Also, based on the expertise of the team, Microsoft .NET was chosen as the main software framework.

As the software is to be sold on a Software as a Service (SaaS) principle, the natural selection for the client is of course a web browser. Usability was one big requirement: The domain itself is complex enough to take a great deal of the users' comprehension. Thus, the software should be as easy to use and as logical as possible to really aid, not hinder, the users in their project business needs. This led to the realization, that a modern UI framework would lend itself very well to support the creation of a usable, domain and task centric software UI. A widely used, well-supported AngularJS was chosen. AngularJS 2 was also considered, but as of the start of this project, it was still in beta phase and the release date was not set. We decided not to take a chance on a pre-production quality software and so settled upon the v1 of the framework.

After deciding on AngularJS, we pondered on the question of how to serve the HTML pages to the browser. One common approach is to use ASP.NET MVC (Model View Controller) to serve the pages with their base data and enhance them with asynchronous operations with AngularJS. As we would have to provide a comprehensive API for the AngularJS in any case to support the design and the asynchrony of the UI, an MVC framework would not give much advantage over simply serving static HTML pages. Static HTML pages, on the other hand, would make it possible to cache the pages in browser or in the server and get a better first response time to the user, after which Angular would kick in to proceed with loading of the data through the API one UI element at a time in a concurrent fashion. This was agreed to be the better and leaner approach, as MVC framework adds another layer of complexity to an already complex architecture.

With the UI layer decided upon, the next major question is the API the UI will use and what framework to use there. The obvious choices for a .NET specialized team are either a WCF service or an ASP.NET Web API service [27, 28]. The former provides more options for the communication protocols and in some situations can be quite a lot more efficient. The downside is the complexity of configuration and the initial cost of implementation. ASP.NET Web API, on the other hand, is very easy to implement and provides a standardized way of creating RESTful or REST-like services [29]. Communication protocol is always HTTP or HTTPS but the payload can be customized to be either JSON, XML or binary (BSON) as well as a custom implementation. Out of the box interoperability with different JavaScript libraries, such as Angular JS, mobile clients and other diverse clients is also a strong point of ASP.NET Web API: HTTP, HTTPS and JSON are widely supported. This turned the decision in favor of the latter.

The Dependency Inversion principle discussed in chapter 3.1 also requires a tool, namely a dependency injection container, to make its use efficient and to not have to spend time on creating one from scratch. The most often referenced options in the articles and blogs talking of DDD, Onion architecture or dependency injection in general are StructureMap, SimpleInjector, Unity and Autofac. These are all quite established and well up to the speed of new framework versions, especially Unity, since it’s developed by Microsoft, who also develops the .NET framework. Performance wise there are not much differences between the candidates and the results vary from test to test [30, 31]. SimpleInjector is closest to a one-man

project, and hence it was dropped. From the remaining three containers, two aspects put Autofac before the others: The integration packages to integrate the container into either ASP.NET MVC or ASP.NET Web API are both provided by the same company that created Autofac [30]. Also, the child lifetime container support in Autofac is something the others cannot provide, which could be essential regarding the scoping and thus, the lifetimes of the instantiated objects [32].

Other tools and frameworks used in the prototype are not of particular interest for this thesis, so they will not be discussed.

5.2. Combined new architectural model

This section describes the proposed new architectural model and the design patterns and decisions made to support it. The architecture was based on the common teachings in Aalto University of Technology, several books, articles, blog posts, podcasts and videos of industry leaders in software architecture as well as the experience of the development team. The underlying architectural models of the solution are presented in the preceding chapters.

The resulting architecture model is depicted in Figure 4. The separate layers are decoupled from each other and the decoupling is enforced by giving each layer their own .NET project and controlling their references. Each .NET project produces their own separate assembly. Assembly references are means of one assembly utilizing the referenced assembly's public interfaces and classes.

Assemblies required for the application to run are introduced, beginning from the center. First, the core assembly contains no references and as such is self-contained. Second, next outwards from the core is the application layer, which references the core assembly. The third layer from the center is the infrastructure layer, which references and uses the application interface layer. It also has a reference to the core layer to be able to utilize the value objects defined there. Contrary to the onion architecture model presented by Palermo [5], in this model the user interface has been moved from the infrastructure layer to an even outer layer. This is because from the viewpoint of the application core, the Web API functions as the client of the core through the application interface. This Web API

can then be called from a variety of clients, be it internet browsers running the product's UI or a dedicated mobile software, and thus it deserves its own layer together with database and the external services. This layer, called adapters layer, references the infrastructure layer loosely coupled with HTTP through the Web API, so there is no assembly reference there.

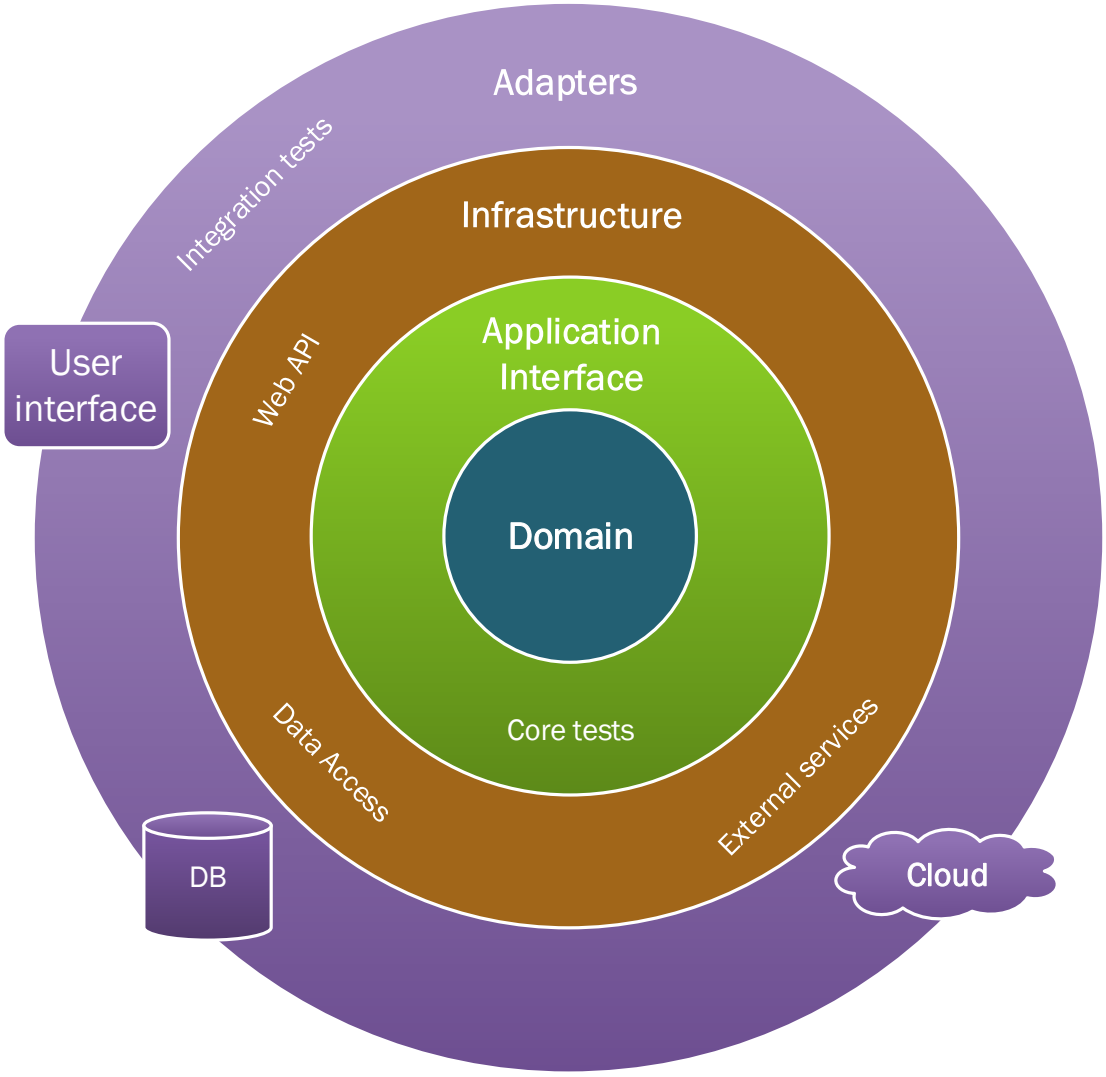

Figure 4 - Resulting architecture, top-level view

In Figure 4 the test assemblies used to make sure the software does what it is supposed to, are also depicted. In the application interface layer, there are the core tests, separated to their own assembly. Similarly, to the application interface assembly, this also references the core. Another test assembly is the integration

test assembly, which resides at the outmost layer together with the UI. Just as the UI, this assembly also references the Web API through loose coupling by HTTP protocol. This provides us with the best opportunity to test the software automatically as a complete whole, only leaving out the user interface. Automatic user interface testing is left outside the scope of this thesis.

The core assembly

Building on the idea of the Onion architecture described in chapter 2.4, the architecture and the development begun from the core of the application to decouple the business concerns from the infrastructural concerns. This assembly contains all the aggregate roots that take care of adhering to the business rules with the help of accompanying entity classes. The core assembly is the only assembly living on the core layer, as depicted in Figure 4.

One concrete example of such business rules lies within the purchase item planning domain. There, a work item can be designated to be outsourced to a contractor. Then, for some reason, such as saving costs, the work item is decided to complete with internal work force. This causes a change in the financial aspect: where before there was just one sum dedicated to the contractor to complete the work item, now the new cost is different and divided between the salaries and the social costs. The method used to calculate the amount of the social costs varies per company, so that method is the business rule that we have to adhere to. In this scenario, the default calculation method was to book a fixed percentage of the salaries to the social costs. This business rule lives inside an entity referenced by the purchase task aggregate root in the core assembly and it is responsible for the correct calculation of the new costs.

As this entity is completely independent, it is naturally easy to test via unit tests. This is one key aspect in promoting maintainability. If we factor in the need to make to software multitenant, that is, same software caters to the needs of multiple clients whose calculation methods differ, it becomes essential. Instead of implementing the different calculations with ‘if’ and ‘switch’ statements within the same entity, we can create one entity for each calculation method and have these entities implement the same interface, thus avoiding the big ball of mud that easily emerges from above depicted scenario. The correct entity for each tenant is then chosen via configuration at runtime with dependency injection. This way, the

entities can be tested separately and the tests remain focused and concise. Thus, when executing the calculation through the aggregate root via the chosen entity at runtime, we can be assured that the calculations are correct. The single responsibility principle remains well adhered to.

Application interface assembly

The application interface assembly is, like the core, the only operational assembly living on the application interface layer, as depicted in Figure 4. A core test assembly also resides on this layer, but when executing the program, it is naturally omitted. The main responsibility of the application interface assembly is to package the aggregate roots' operations into easily usable units and also to manage the scope of the unit of work. This means creating the necessary database connections and transactions, retrieving the aggregate roots, calling the domain operations on those aggregate roots, saving the changed state back to the persistence medium and committing the changes, and finally disposing of the used resources. This is also the layer where most of the infrastructure layers' interfaces are defined as this is the only way for the application service layer can use the services provided by the infrastructure layer due to the reference direction being from the outside to the inside. This layer defines the abstractions of the infrastructure layer, namely the interfaces on this layer. The actual implementations of these interfaces reside on the infrastructure layer. The correct implementations are chosen and provided by the dependency injection container at runtime.

On this layer, dedicated service classes have been crafted for each aggregate root to take care of the aforementioned responsibilities. These are called application services. These classes take the dependencies through constructor injection. From the perspective of the application service, the dependencies are those interfaces defined on this same layer, but the actual classes injected to the constructor are the implementing classes chosen by dependency injection container based on the dependency configuration.

There are two special groups of dependencies that need further introduction. As the architecture is split to queries and commands based on Command and Query Responsibility Segregation (CQRS), the split starts from here, from the application services. The application services function as façades for their callers,

and as such, they were not split per CQRS ideas. The classes called by the application services, that is, their dependencies, on the other hand, are split and have either read or write responsibilities, but not both. These are called query handlers and command handlers respectively. They live on the infrastructure layer, and will be described in more detail in the following subsection. The interfaces for the handlers as well as the command and query objects containing the necessary information to execute the commands and queries, lie here, as do the generic interfaces that all the query handler interfaces extend, and similarly so for the command handlers. These generic interfaces make it easier to join the implementations to the abstractions, as generally only one implementation exists for each command or query handler.

The model classes that the application service takes as parameters and returns are likewise defined on this layer. The domain objects in the core are kept pure and when necessary, they are mapped to and from the application service layer's respective objects.

Infrastructure assembly

Infrastructure assembly is on the infrastructure layer and ties the in-memory operations of the core and application service to the persistence mediums, specifically the databases and external services. This assembly also houses the public API in the form of WebAPI controllers, that the user interface and potential external applications use and as such, it is the entry point of the application back end. As such, this is the place where the dependency injection container is configured and initialized. Here, all the interfaces are given their concrete implementation classes by configuration and with that information, the container can instantiate and supply the correct implementation when that or its interface is requested.

The query and command handlers mentioned in the previous subsection also live here, in the infrastructure assembly. Each handler is responsible for executing one specific query or command. This is the strictest way of adhering to the single responsibility principle. The amount of classes surely gets large with this design, but each query and command handler is very simple and the name denotes perfectly what that class does. They are also easy to test as they only tie in to the

database against which the query or command is run. Another possibility would have been to combine the query operations of the same entity or entity group into one larger query service, and command side similarly. In time this would make the said services bloat with more and more methods, some of which share similarities and some of which are completely different due to different fetching strategies needed to keep the software performant. Instead, query reuse in the multiple query handlers is achieved by using extension methods on the database models [33]. This gives a bonus advantage in the form of making the queries more readable and thus getting them to adhere as much as possible to the ubiquitous language of the domain.

The query handlers access the database via Entity Framework (EF) data context and the database entity objects, with which the data can be queried easily and effectively. The query handlers have no other dependencies other than the EF and they access the database directly using the EF object relational mapper (ORM). As each query is executed in its own query handler, each query can be fine-tuned to use distinct fetching strategies and to return just the data needed at that specific query.

The command handlers can be split into two categories: those that do simple CRUD-style insert, update and delete operations on simple, tabular data, and those that execute complex domain operations via DDD-principles. The CRUD-style command handlers are similar to the query handlers in that they use the same EF context directly and do not have other dependencies. They execute the commands to the database altering its state with a straightforward manner. On the DDD-style of command handlers, on the other hand, the structure is much more complex, though the command handler itself is quite simple. The command handler has a dependency to the core of the application. The responsibility of the command handler is to retrieve the aggregate root, defined in the core, through a separate EF data context, which contains mapping configuration to hydrate the aggregate root and its entities from the database. Then, the command handler calls the correct method of the aggregate root to fulfill the domain operation. When the operation completes, the command handler calls save on the EF data context. As the aggregate root is mapped to the database and as the EF framework tracks the state of the objects, the EF ORM can execute the necessary inserts, updates and deletes that are caused by the aggregate roots state changes.

The DDD-side of the command handlers does not require any converters, as the EF generates the domain objects through the configured mappings. The CRUD-side of the command handlers as well as the query handlers require an additional converter to convert the database representation to the appropriate DTO-object, which can be passed through the API. These converters also reside in this assembly.

To facilitate unit testing and to ensure that all operations done within a single command receive the same acting username and the exact same timestamp, irrespective of the duration of the command, two helper classes are introduced in this assembly to provide the current acting username and the current timestamp. These are injected through dependency injection into the command and query handlers, for them to use where needed.

This assembly houses all the adapters required to communicate with external services. Where the semantics are same, but different service endpoints are needed, a common interface or base class will be used to contain the common logic, and then the endpoint specific adapter will extend that to work with the external services' specificities.

If at some point required, this layer would also be the place for any other applications using the same logic, such as a windows service for up keeping or a console application for a maintenance tasks. From this layer, they can use the same application services on the application interface layer and reuse the existing, tested code base.

User interface assembly

As the logic of the application resides in the inner layers of the application, the user interface is left very thin. It is hard to exempt it from all logic, as, for instance, data validation is a common requirement for an application with usability as one of the main targets. Still, most of the code and logic here is purely presentation focused. The UI calls the infrastructure layers API through HTTP(S), gets the data required for showing and when the user acts on the UI, relays the command to the WebAPI.

General flow of the application

As a query call comes from the UI through to the API in the infrastructure assembly, it is first passed to the underlying application service assembly, from where a query is passed on to the implementing class of that query. That implementing class lives in the infrastructure layer, so the only responsibility of the application service is to keep a concise façade and single entry point per resource to operate on and open a database connection that the query, or in some cases, multiple queries, can use. This is depicted in Figure 5 as the leftmost line crossing slightly into the application interface layer, but staying completely out of the domain.

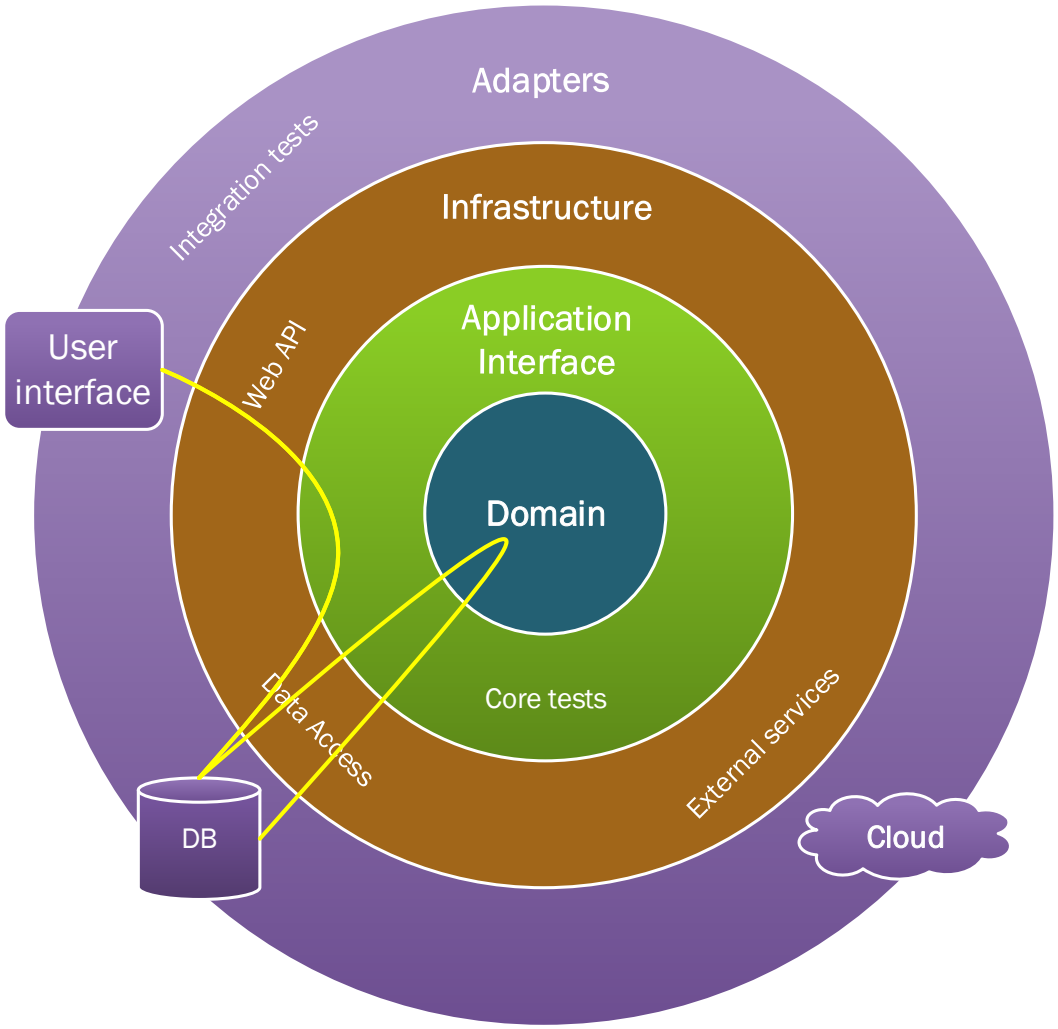

Figure 5 - Resulting architecture with the flow of method calls

A CRUD-style command is handled just like the query, although the database objects are first loaded into memory in the data access, then edited in the command handler and subsequently saved back to the database. The DDD-style command, on the other hand, is different. The command handler first loads the aggregate root into memory. This is similar to what the query handler does, but using a different data context and different entities. The aggregate root is on the core layer, so when the command handler calls the aggregate root's domain method, it is essentially skipping the application interface layer and calling directly into the core layer. This is correct, as per onion architectures principles layer skipping is permitted as long as the direction is towards the center. Once the aggregate root has completed all the changes, essentially changing the in-memory representation of the aggregate root and its contained entities, the infrastructure layer relays those changes back to the database with the help of Entity Framework's change tracking.

Chapter 6

Analysis and evaluation

As a result of this thesis, the Purchase item planning software, built with the architecture described, is now in beta testing on the client awaiting a few integrations and additional features to get to production. In this chapter, the first subchapter mirrors the resulting application to the requirements and an analysis of the produced application is given in respect to them. The development of the software created multiple ideas on how to improve and simplify the architecture. These ideas are described in the latter subchapter.

6.1. Results of the application

The feedback from the client thus far has been very positive. We have captured the need with the DDD principles well and managed to turn those into good requirements. From those requirements, we managed to create a good task driven UI, that really guides the user and captures the user's intent in the most domain centric operations. The usability of the resulting software received appraisal. The problem domain is quite complex, but utilizing task-based and DDD principles allowed us to simplify the user experience. Naturally, not all operations of the software are task driven or implemented with DDD principles, as lighter bounded contexts, such as administration views do not contain domain rules.

Regarding the research question, "Does onion architecture promote software maintainability?" unfortunately, a thorough and complete answer cannot yet be given. The real measure of software maintainability can only be seen after years

of maintaining and growing the software. How easy is it to add new features? How much regression is introduced with the new features or bug fixes? How swift can the bug fixing be? Will the code be regarded as “legacy code” in a few years, into which no one wants to delve, or will it stand against the effects of time? As the thesis cannot await years, the results have to be based on other factors.

The software testing was split into two different aspects: unit testing and integration testing. The domain operations in the core were unit tested thoroughly, as it was easily achieved with them being independent of other layers of software. These are also, as noted before, the most important aspects of the software and therefore great emphasis has to be put on testing those operations. The other layers were not methodically unit tested. Instead, the full stack, minus the UI, was tested as a whole with integration testing that called the same layer the UI calls, namely the WebAPI. This gives the best coverage and maintains a proper granularity, so that the tests do not succumb to testing implementation details, but instead test complete operations. This way, refactoring even larger portions of the code base will not require altering dozens of tests. Instead, the integration tests proved invaluable in trying out and developing the architecture, as the complete operation could be tested before and after the refactoring and with tests passing afterwards, adequate certainty was received that the software continues to work properly.

The tests were written with Test Driven Development style, where the test was written first and the implementation only afterwards, thus thinking of the test and the requirement foremost and implementation only later, when the interface and usage of the new operation is already defined. The testing was done utilizing Specflow, which is a framework for writing tests in business terms and in natural language. This allowed the tests to be created by nontechnical domain experts together with the developers. Tests were written in a way that they are independent of the environment in which they are run, so they can be run on any developers’ machine and also, in the Continuous Integration (CI) server. As they run the whole stack, Katana, which is a .NET implementation of the OWIN-

standard, is utilized in running the WebAPI in memory and the testing can be done with the same components, as when the software is hosted on the actual server. To be able to use the database in memory as well to facilitate the CI-testing, the database is instantiated into a LocalDB instance, which is a lightweight implementation of the Microsoft SQL Server that can be run completely in the memory.

With these testing tools in place, the structure of the application architecture was easy to modify and new ideas could be tested with confidence. This also gave indication of the maintainability of the architecture. A new feature almost never changed the behavior of the existing features. Bug fixes were easy to implement and seldom affected more than one class, which indicates that the single responsibility principle was well applied. The most important factor of the onion architecture held true: changes to the database had no effect on the code outside of the infrastructure layer. The same applied on external services. Even changing a file service to another had no effect on the inner layers of the architecture. The domain, that is, the core and application layers, remained completely unaffected on both accounts. This is a major aspect in making the software maintainable, as the domain truly is independent from the infrastructural concerns.

6.2. Further studies

Although the architecture of the application seems quite good and answers well to the given requirements, there are still flaws that would warrant further studies.

The greatest problem evident from working with the architecture lies within the Application Service layer. These service classes that are called by the clients, be it an API controller or a console application, are in controversy with the SOLID-principle [9]. As the application service is the entry point to the inner application logic, it has the responsibility of defining the boundaries of the unit of work, or transactional boundaries, of the operations. This is logically correct and works well both from the performance as well as integrity point of view. The problem lies in the structure of the application service classes. They are modeled after the

logical entities of the domain and as such contain both the queries and commands required regarding those entities. This creates a maintainability problem, as these classes have multiple reasons to change and as such, it breaks the Single-responsibility Principle of the SOLID-principles. Addition of a new query or command to that entity will also add a new method to the application service. It is also in violation of the Interface Segregation principle, as the clients of the application service usually only need to call one of its methods, but they are implicitly dependent of all of its methods. At this stage of the development of the ProjectBI-software, this was found acceptable and did not warrant further research.

Another problem with the same application service classes lies in the unit of work pattern and the return values of the commands. Ideally, a command can be seen as a “fire and forget” type, where when the command is launched, the call returns immediately and the command is left to be executed in the background. In this scenario, database generated values, such as identity-columns, become a problem. Easiest solution is to modify the command paradigm to also return a value, but this makes the code more obscure and defeats some of the purpose of the CQRS. A better approach is to generate the identities on client, as a GUID for instance, and rely that to the server and on to the database with the command. This would have required a more extensive redesign of the database, and was at this stage cut from the feature list. The easier, value returning route was chosen instead.

Another problem in an architecture such as this, is the abundance of dependencies injected into the constructor of any class utilizing more than one query or command handler. The problem points are again the application services, which act as the endpoints of a given resource, such as a “tender”. All methods regarding tenders are grouped into the same application service. As each query and command that needs to be launched, also needs to have a respective handler instantiated to be able to call it, the constructor has all those handlers as parameters. Of course, the DI container handles the actual instantiation, but still maintaining all those parameters and their respective class variables is a burden. One possible solution to this is to use a service locator, which is a single dependency injected into the constructor, which can be used to find the correct implementation of the query or command handler in runtime. The downside, and the reason this was not applied, is that these kinds of solutions hide coupling. It is

no longer obvious, which controller or other component is dependent on which query or command handler. Thus, we decided against using a pattern such as this and rather work with the numerous, but explicit dependencies.

Chapter 7

Conclusion

The intent of the thesis was to find out what traits make software maintainable, how they can be used in software development and finally construct a software architecture promoting maintainability in a complex problem domain. During the literature review phase, several tools were found that can be utilized in building the maintainable architecture as well as tools to validate it. One by one these tools came together to form a coherent architecture. Some of the tools, such as domain-driven design, affected the requirements gathering phase already, which had a positive effect on the clear requirements. The positive effect can be seen in the feedback of the client testing the software: our understanding of the client's needs was better than expected.

The main idea of the architecture presented in this thesis emerged already before the beginning of the thesis work. This thesis provided a good way to document and reason about the choices made in building the architecture. During the writing of the thesis, the architecture matured and was altered multiple times, both from the emerging revelations of the actual implementation as well as the findings and ideas from constructing the thesis.

Unfortunately, the timeframe of writing a thesis does not allow us to monitor and measure the maintainability of the resulting architecture where it really matters: during the maintenance phase of the software lifecycle. The resulting software is still under development, so changes to the software and its architecture are still being done. Instead, the evaluation was carried out by gathering input from the customers using the prototype software as well as from the design team creating the software. The results were quite positive and encouraging. The usability received praise, much thanks to the domain-driven design method and its combination to the task-based user interface design. Maintainability was found to

be good on the measures applicable in the pre-maintenance phase, such as the five SOLID-principles, and the observed effect of bug fixes and new features to the existing code base. Room for improvement in the architecture was still left and many ideas were left for further studies.

References

1. PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach.* McGraw-Hill Higher Education, 2005. ISBN 9780073019338.

2. SHAW, Mary; and GARLAN, David. Springer Berlin Heidelberg, 01/01, 1995. *Formulations and Formalisms in Software Architecture*, pp. 307-323. ISBN 978-3-540-60105-0.

3. FOOTE, Brian; and YODER, Joseph. Big Ball of Mud. *Pattern Languages of Program Design*, 1997, vol. 4. pp. 654-692.

4. JANSEN, Anton; and BOSCH, Jan. *Software Architecture as a Set of Architectural Design Decisions.* IEEE, 2005.

5. PALERMO, Jeffrey. *The Onion Architecture.* 2008 [cited 10.1.2016]. Available from: <http://jeffreypalermo.com/blog/the-onion-architecture-part-1/>.

6. BASS, L.; CLEMENTS, P.and KAZMAN, R. *Software Architecture in Practice.* Pearson Education, 2012. ISBN 9780132942782.

7. NILSSON, Jimmy. *Applying Domain-Driven Design and Patterns: With Examples in C# and .NET.* Pearson Education, 2006. ISBN 9780132797498.

8. MARTIN, Robert C. Design Principles and Design Patterns. *Object Mentor*, 2000, vol. 1. pp. 34.

9. SINGH, Harmeet; and HASSAN, Syed Imtiyaz. Effect of SOLID Design Principles on Quality of Software: An Empirical Assessment.

10. MARTIN, R. C.; and MARTIN, M. *Agile Principles, Patterns, and Practices in C\#.* Pearson Education, 2006. ISBN 9780132797146.

11. BAILEY, Derick. *S.O.L.I.D. Software Development, One Step at a Time.* Jan./Feb., 2010 [cited 13.2.2016]. Available from <http://www.codemag.com/article/1001061>.

12. BUSCHMANN, F. *Pattern-Oriented Software Architecture, A System of Patterns.* Wiley, 1996. ISBN 9780471958697.

13. GALLAUGHER, John; and RAMANATHAN, Suresh. *The Critical Choice of Client Server Architecture: A Comparison of Two and Three Tier Systems*, 1996 [cited 15.1.2016]. Available from <https://www2.bc.edu/~gallaugh/research/ism95/cccsa.html>.

14. EVANS, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional, 2004. ISBN 9780321125217.

15. VERNON, Vaughn. *Implementing Domain-Driven Design.* Pearson Education, 2013. ISBN 9780133039887.

16. Microsoft Patterns & Practices Team. *Microsoft Application Architecture Guide, 2nd Edition.* Microsoft Press, 2009. ISBN 9780735642799.

17. WALDRON, Wade. *Domain Driven Design through Onion Architecture.* Sep, 2014 [cited 10.1.2016]. Available from <http://boldradius.com/blog-post/VCRYijIAAHhescgX/domain-driven-design-through-onion-architecture>.

18. FOWLER, Martin. Inversion of Control Containers and the Dependency Injection Pattern, 2004 [cited 25.2.2016]. Available from <http://martinfowler.com/articles/injection.html>.

19. Amaba, Ben,PhD, CPIM. Business Driven Development. *IIE Annual Conference.Proceedings*, 2006. pp. 1-6 ABI/INFORM Complete, ProQuest Central.

20. EVANS, Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries.* Dog Ear Publishing, 2014. ISBN 9781457501197.

21. CQRS Documents by Greg Young. Nov, 2010 [cited 5.3.2016]. Available from <https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf>.

22. DUKE, D. J.; and PUERTA, A. *Design, Specification and Verification of Interactive Systems '99: Proceedings of the Eurographics Workshop in Braga, Portugal, June 2–4, 1999.* Springer Vienna, 2012. ISBN 9783709168158.

23. MEYER, Bertrand. *Object Oriented Software Construction.* Prentice Hall, Inc., 1988. ISBN 9780136291558.

24. BOGARD, Jimmy. *Busting some CQRS Myths.* 22.8.2012 [cited 5.3.2016]. Available from: <https://lostechies.com/jimmybogard/2012/08/22/busting-some-cqrs-myths/>.

25. ARTTO, Karlos A.; and WIKSTRÖM, Kim. What is Project Business?. *International Journal of Project Management*, 2005, vol. 23, no. 5. pp. 343. ISSN 0263-7863.

26. CHUNG, L., et al. *Non-Functional Requirements in Software Engineering.* Springer US, 2012. ISBN 9781461552697.

27. *What is Windows Communication Foundation.* Microsoft Developer Network. 2015 [cited 5.12.2015]. Available from: <https://msdn.microsoft.com/en-us/library/ms731082(v=vs.110).aspx>.

28. *ASP.NET Web API.* Microsoft Developer Network. [cited 5.12.2015]. Available from: <https://msdn.microsoft.com/en-us/library/hh833994(v=vs.108).aspx>.

29. BLOCK, G., et al. *Designing Evolvable Web APIs with ASP.NET.* O'Reilly Media, 2014. ISBN 9781449337902.

30. ZHOU, Jacky. *Do You Know the Best Dependency Injection Container?* 1.9.2015 [cited 5.12.2015]. Available from: <https://rules.ssw.com.au/do-you-know-the-best-dependency-injection-container-(aka-don%E2%80%99t-waste-days-evaluating-ioc-containers)>.

31. MANN, Nathanael. *IoC Battle in 2015 Results: Using Ninject – Think again!* 28.1.2015 [cited 5.12.2015]. Available from: <http://cardinalcore.co.uk/2015/01/28/ioc-battle-in-2015-results-using-ninject-think-again/>.

32. BLUMHARDT, Nicholas. *An Autofac Lifetime Primer*. Jan. 2011 [cited 5.12.2015]. Available from: <http://nblumhardt.com/2011/01/an-autofac-lifetime-primer/>.

33. CHARBENEAU, Edward. *Giving Clarity to LINQ Queries by Extending Expressions.* 15.11.2013, 2013 [cited 26.3.2016]. Available from: <https://www.simple-talk.com/dotnet/.net-framework/giving-clarity-to-linq-queries-by-extending-expressions/>.