Improving software flexibility in a smart business network

Designing an ontology-driven software architecture for the Internet Business Learning Community

Bas Truren

January, 2010

Author: S.J.M. Truren
Student number: 1204106
Program: Technische Bestuurskunde

Graduation committee
Chair: W.A.G.A. Bouwman
First supervisor: J. van den Berg
Second supervisor: H.J. Honig
External supervisor: P.J.A.M. Bessems

Delft University of Technology
Faculty of Technology, Policy and Management
Section Information and Communication Technology
Jaffalaan 5
2628 BX, Delft
The Netherlands

Internet Business Learning Community
TU/e Multimedia Paviljoen
Horsten 1
5612 EX, Eindhoven
The Netherlands

Abstract

Because of the rapid changes in the environment in which businesses operate, businesses have to become more flexible. The smart business network has emerged as a new organisational paradigm that promises more flexibility. A smart business network is a network of multiple actors that is supported by an open digital platform. Flexibility at the business level requires flexibility at the information technology level and alignment between the business level and the information technology level. The research described in the thesis focuses on the flexibility of the open digital platform at the information technology level. This open digital platform is the software system that supports the business processes of the actors of the smart business network.

The objective is to design a flexible software system for a smart business network. The research is performed with an existing smart business network as the case study: the Internet Learning Community (IBLC). The thesis introduces an ontology-driven software architecture and a software engineering approach that is complementary to that architecture. The ontology-driven architecture is based upon the notion that the different actors in a smart business network may have different perceptions upon the business domain. An actor's view upon the business domain is formalized in an ontology. The ontology consists of the terms and their relations used by the actor. The ontology is linked to the software components by models, which describe the individual components of the software system. The models and the source code of the components are defined using the same terms and relations as defined by the ontology.

Preface

This thesis is the finalization of my part-time study at the faculty of Technology, Policy and Management of the Delft University of Technology. In the daytime I'm a self employed information technology consultant. As an information technology consultant learning new technology and gaining new insight is part of the job. A couple of years ago I noticed that the knowledge about the new technology is useful to me for about five years. And although the lifespan of the gained insight is longer, I felt that I needed something more, something lasting. I started the part-time study to gain information that would broaden my horizons and to gain knowledge that would last. At that time my primarily interest was gaining knowledge about decision making and strategic behaviour. I regularly encounter situations in which both subjects are important and I found it difficult to learn about them from experience or books only. For my master thesis I wanted the subject to be a combination of my daytime work and my part-time study. During a discussion with Wilfred Kuijpers I realized the importance of the change that the Internet Business Learning Community (IBLC) is going through. The IBLC is a client I regularly work for. Reading about similar organisations led me to the subject of smart business networks. A publication about smart business networks led me to Jan van den Berg of the TU delft.

I would like to thank my supervisors at the TU Delft: Jan van den Berg, Harry Bouwman and Job Honig, for their guidance, feedback and advice. I would also like to thank Paul Bessems of the IBLC for his guidance and advice. I thank Wilfred Kuijpers and Ron Kompier of the IBLC for the stimulating discussions and support over the years.

I would like to thank Kim, without her unconditional support the part-time study and this work would not have been completed.

Bas Truren

The Hague, December 2009

Table of Contents

List of Figures ... 5
List of Tables ... 6
1 The research project ... 7
1.1 Introduction ... 7
1.2 Background ... 7
1.3 Research problem ... 7
1.4 Research objective ... 8
1.5 Research methodology and thesis outline ... 9
2 Smart business networks ... 11
2.1 Introduction ... 11
2.2 The emergence of smart business networks ... 11
2.3 Business processes in a smart business network ... 13
2.4 Introducing multi-actor complexity ... 15
2.5 The Internet Business Learning Community ... 17
2.6 Conclusion ... 19
3 Software flexibility ... 20
3.1 Introduction ... 20
3.2 A workable definition of software flexibility ... 20
3.3 Relevant software management theories ... 23
3.4 Relevant software-engineering theories ... 25
3.5 Conclusion ... 28
4 Towards an ontology-driven software architecture ... 30
4.1 Introduction ... 30
4.2 Software architecture ... 30
4.3 The ontology ... 31
4.4 Linking the ontology to the software ... 32
4.5 Multiple ontologies: versions and mappings ... 36
4.6 Comparison to an existing architecture ... 37
4.7 Conclusion ... 38
5 Towards a software engineering approach ... 39
5.1 Introduction ... 39
5.2 Specification: gathering functional requirements ... 39

5.3 Realization: constructing the software system ... 40
5.4 Technical characteristics ... 41
5.5 Conclusion ... 44
6 Testing the flexibility of the software system ... 46
6.1 Introduction ... 46
6.2 Business drivers for change ... 46
6.3 Information technology drivers for change ... 48
6.4 Technical evaluation ... 49
6.5 Conclusion ... 51
7 Conclusions and recommendations ... 52
7.1 Conclusions ... 52
7.2 Relevance to other smart business networks ... 53
7.3 Reflection ... 54
7.4 Recommendations for further research ... 54
7.5 Recommendations for the IBLC ... 55
References ... 57
Appendix A The process diagrams ... 64
Appendix B The ontology ... 67
Appendix C The use case model ... 70
Appendix D The object oriented model ... 71
Appendix E The interface model ... 73
Appendix F The relational model ... 75
Appendix G From object oriented model to the relational model ... 78

List of Figures

Figure 1: Thesis outline and applied methodologies. 10
Figure 2: Alignment between business and information technology. 14
Figure 3: The relation between a business domain, the actor and the conceptualization. 14
Figure 4: Framework for producing knowledge in a multi-actor setting (van Riet, 2003). 16
Figure 5: Core elements for process design (de Bruijn and ten Heuvelhof, 2002). 17
Figure 6: A schematic abstraction of a network of organisations. 18
Figure 7: Abstract representation of the open digital platform. 20
Figure 8: Levels of interoperability (Janssen and Scholl, 2007). 25
Figure 9: Quality indicators and instruments for the open digital platform. 29
Figure 10: The relation between a business domain, the actors and their conceptualizations. 31
Figure 11: Relation between the conceptualization and the ontology. 32
Figure 12: Service layers and the dependency between services (Erl, 2008). 33
Figure 13: The layered software architecture (Evans, 2004) within a service. 34
Figure 14: Synthesis of the service layers and the layered software architecture. 34
Figure 15: Dependencies between the ontology and the layers. 35
Figure 16: Dependencies between the layers. 43
Figure 17: Drivers for change in the service and software layers. 46
Figure 18: Dependency analysis of the layers of the Bookings service. 50
Figure 19: Dependencies between the ontology and the layers. 53

List of Tables

Table 1: Definitions of business flexibility. 11
Table 2: Definitions of a business network. 12
Table 3: Overview of the business requirements. 19
Table 4: Definitions of software flexibility. 21
Table 5: Overview of the quality indicators. 23
Table 6: Overview of the instruments. 29
Table 7: Overview of the guidelines. 45

1 The research project

1.1 Introduction

This chapter introduces the research project and the work performed. It starts with background information about the company for which the research is performed: the Internet Business Learning Community (IBLC). The second section describes the context of the research problem and demarcates the problem to the issue of flexible software. Section three introduces the research objective and states the research questions. Section four describes the research methodologies used to answer those questions. The last section provides an overview of the chapters of this thesis. It links the chapters to the applied methodologies and to the discovered answers.

1.2 Background

The IBLC provides professional services in the field of human resources services. It is located in the multimedia pavilion of the TU Eindhoven. Its expertise is to provide information and implement processes in the field of corporate training and development and in the field of recruitment and assessment. The core product of the IBLC is a course catalogue system. The catalogue contains information about thousands of corporate training courses in the Netherlands. The system is maintained by both employees of the IBLC and of its supplying partners. The supplying partners include training companies like: NCOI, ISBW, LOI, Oracle and Schouten & Nelissen. The system is also used by both the employees of the IBLC and its buying partners to search for courses and to register for an appropriate course. The buying partners include companies like: Philips, Aegon, Atos Origin, the province of Utrecht and the Ministry of Foreign Affairs.

According to Bessems (2009), the CEO of the IBLC, the human resource market is changing rapidly. He mentions the following changes: there is more pressure on the labour market by population ageing, there is more recruitment via social networks and there is a decrease in the time span an employee works for the same organization. The IBLC expects this perceived market changes to result in higher recruitment and training costs for organizations. The IBLC's solution is to increase the innovation in the human resource market. It therefore formulated its new mission statement as: "deploying people smarter". That mission is realized by enabling organizations and individuals to efficiently work together and by providing smart services (Bessems, 2009). To realize its mission IBLC is gradually transforming itself from an intermediary in the human resource market into a central hub within a business network. As an intermediary the IBLC operated between the suppliers and the buyers. As a network hub it enables the suppliers and the buyers to collaborate, by providing business services and an information system.

1.3 Research problem

This research project takes place within the context of the transformation of the IBLC. The most important organisational transformations are almost completed. The transformation of the information system however, is just beginning. There are two structural differences between the current information system and the new system that is needed to support the role of the IBLC in the extended business network. First, the system currently in use is primarily designed to support the processes of the IBLC. The desired system should also support the processes of the other actors in the business network. This is a focus shift from a single-actor to a multi-actor system. The new system should support and integrate a variety of future actors. Second, the current system supports monolithic products. The desired system has to support composite products, consisting of

components from different suppliers. The relation between a supplier and a product changes from a hierarchical structure to a network structure, in which multiple suppliers are jointly responsible for the supplied product.

These two structural changes have several implications for the design of the new system. Those include the need for new functionality, the redesign of the security features, the possibility of scalability problems and possible issues with various other software aspects. The new functionality is specified for the most part. However there are uncertainties about how to provide the flexibility needed by the new system. These uncertainties are twofold: first, there is a knowledge gap with respect to the realization of the software. Second, there is inherent uncertainty in the system: the requirements placed upon the system by future actors, future products and future services are partly unknown. The security, the scalability and the other implications are considered, as far as the IBLC is concerned and within the context of this thesis, to be solvable by applying existing techniques available within the IBLC and with existing information technology techniques.

The research project is designed using the guidelines provided by Verschuren en Doorewaard (1999). In their classification of project types, this project is a practice-oriented design project. Before such a project can be started a problem finding project and a diagnosis project should be finished. In the case of the IBLC, the problem finding is performed by Bessems (2009) and the diagnosis by Gerbranda (2008)[1].

1.4 Research objective

Before the objective is stated two definitions are needed. In chapter two of this thesis the transformation of the IBLC is shown to resemble the transformation into a smart business network. A definition of Van Heck and Vervest (2008) is used: a smart business network uses an open digital platform to do business with anyone, anywhere, anytime, despite different business processes and computer systems to gain results that normal business network are not be able to achieve. A broad definition for the flexibility of software is used: the ease with which a system can adjust to changing circumstances and demands (Egyedi and Verwater-Lukszo, 2005). Referring to those definitions and to the research problem in section two, the objective of this research is stated as follows:

The objective of the research is to design a flexible software system for a smart business network.

In this thesis the proposed solution is to identify and incorporate knowledge from scientific research into the architecture of the software. The structure of the information system shifts from a single-actor to a multi-actor system and from a hierarchical structure to a network structure. This indicates that multi-actor theory might be useful. Furthermore, a definition for flexible software is used to identify useful software-engineering theories. The applicability of the theories is tested by designing and testing a software system for the business processes of the IBLC. In this research project the IBLC is used as a case study. From the research objective the main research question is derived:

1. *How can this software system support:*
    a. *The processes of the actors of the smart business network?*
    b. *The handling of composite products and services, consisting of components from different suppliers?*

[1] The years of publication suggest an inverse order of problem finding and diagnosis. This is not the case: the work of Bessems is published at a later date.

The sub questions are defined as follows:

2. *Which business requirements does a smart business network place upon the software system?*
3. *Which instruments might increase the flexibility of the software system?*
4. *What kind of architecture can be used?*
5. *How can the software system be realized using the architecture?*
6. *How does the software system respond to change?*

1.5 Research methodology and thesis outline

The applied methodology varies per phase of the research. This section describes the research methodologies in chronological order. The outline of the thesis reflects the methodology and the phases of the project. Although the phases are described in chronological order, the actual work is performed in a few iterations. This approach enables the possibility to use new insight, gained from a previous iteration, in any phase of the current iteration. Hopefully this decreases the chance of missing important information and enhances the coherence of the phases. It also enables the possibility to focus on the most important guidelines in the first iterations and to shift the focus to less important guidelines in later iterations (Figure 1).

Chapter two: Smart business networks
In chapter two provides an introduction to smart business networks as an organisational structure. The chapter addresses the second research question. The business requirements a smart business network places upon a software system are discussed. A literature study conducted provides the information and the resources for an overview of the need for flexibility in a business networks. The main corpus of literature is: scientific books, articles and theses. This research is followed by interviews to describe the IBLC.

Chapter three: Software flexibility
Chapter three provides a workable definition of software flexibility useable within this research project. The definition is used to identify the quality indicators for the software system. The definition and the indicators are derived from scientific literature. This chapter addresses the third research question: the definition and the indicators are used to search for relevant instruments to influence the flexibility of the system. These instruments are derived from software management and software engineering theories.

Chapter four: Towards an ontology-driven software architecture
Chapter five provides an overview of the ontology-driven software architecture. The architecture acknowledges the existence of multiple actors and uses an ontology to represent an actor's view upon the domain. The architecture links the ontology to the models that are needed to specify and realize the software system. The chapter addresses the fourth research question.

Chapter five: Towards a software engineering approach
In chapter five the architecture is applied to the software system of the case study. Applying the architecture to the case study reveals whether the architecture is successfully applied and reveals the guidelines for applying the architecture. The chapter addresses research question five.

Chapter six: Testing the software system

In chapter six the software system is tested against the forces of changes. The flexibility of the architecture is tested using four scenarios: two scenarios with a business-driven change and two scenarios with an information technology-driven change. The software system is also evaluated using modern software analysis techniques. This chapter addresses the sixth research question.

Chapter seven: Conclusions and recommendations

This chapter describes the main conclusions of the research. The individual answers to the research questions are discussed in the conclusion of the corresponding chapter. The recommendations address the use of the guidelines by the IBLC. Furthermore this chapter provides an overview of the recommendations for future research.

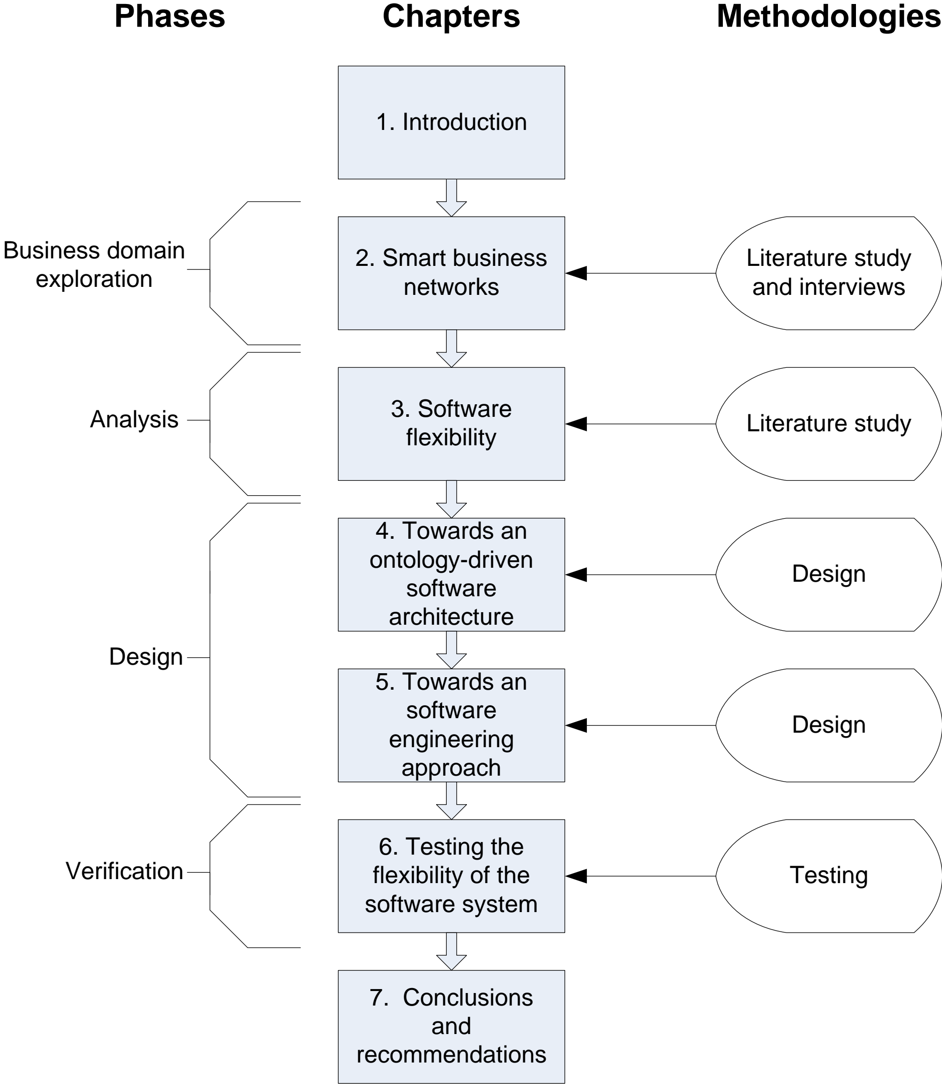


Figure 1: Thesis outline and applied methodologies.

2 Smart business networks

2.1 Introduction

This chapter provides an introduction to smart business networks as an organisational structure. The specific business requirements that a smart business network places upon a software system are discussed. Next the complexity introduced by a multi-actor system is described. The chapter ends with an example of a smart business network: the Internet Business Learning Community (IBLC). In this thesis the IBLC is used as a case study. However, the results of the research should not be specific to the IBLC, they should be usable in business networks in general.

2.2 The emergence of smart business networks

The environment in which businesses operate is changing rapidly. Oosterhout et al. (2008) identify two primary forces that change the business environment: *technology innovation* and *long-term public policy shifts*. The technology innovations enable organisations to become more competitive. Technological innovations also require organisations to keep up with competitors. This leads to an innovation race, which changes the business environment (Navarra and Cornford, 2008). The policy shifts include, for example, the increase in liberalization and the trade agreements that enable globalization. Liberalization increased competence in markets that were primarily monopolistic or oligopolistic. Globalization increased the competition across national borders. According to Christopher (2008) *the biggest change is the shift of focus towards speed*. That is the speed of innovation, the speed of producing and the speed of delivering. He mentions the shortening product life cycles to create a shorter time-to-market, the just-in-time practices in manufacturing to reduce inventory levels and the increased flexibility in the delivery requirements of the customers. In order to increase their competitive advantage in this complex and rapidly changing environment, organisations have to become more flexible (Christopher, 2008). According to Phillips and Wright (2008) organizations are finding that their ability to respond to unpredicted changes in the business environment is an important factor for survival.

In scientific literature multiple definitions can be found for the term business flexibility[2]. The table below provides an overview of some of those definitions. In this thesis MacKinnon et al.'s (2008) definition for business flexibility is used. Their definition contains one aspect that is important when addressing the flexibility of business: there is the notion of "the firm's deliberately crafted ability" which acknowledges the assumption that flexibility is not always an emergent characteristic of an organisation, but rather a desired feature that often requires effort.

Author	Definition of business flexibility
Sanchez (1995)	Strategic flexibility is the firm's abilities to respond to various demands from dynamic competitive environments.
Lau (1996)	Strategic flexibility refers to a firm's ability to respond to uncertainties by adjusting its objectives with the support of its superior knowledge and capabilities.
MacKinnon et al. (2008)	Strategic flexibility is the firm's deliberately crafted ability to recognize, assess, and act to mitigate threats and exploit opportunities in a dynamically competitive environment.

Table 1: Definitions of business flexibility.

[2] Flexibility is sometimes referred to as adaptability (Oxford, 1996) or agility (Ribeiro, et al., 2009).

In order to enable businesses to become more flexible a new organisational paradigm has emerged: the business network. From a traditional point of view independent organisations are competing with other businesses in order to survive. But organisations not only compete with each other, they also need to cooperate (Christopher, 2008). In a business network the organisation is part of a network of interdependent organisations, competing against other business networks. The model describing the value creation in a business networks is called: *value network* (Stabell and Fjeldstad, 1998). A value network is more complex and dynamic than a value chain model (van de Kar, 2005). Porter (1985) introduced the concept of value chains: describing an organisation as a chain of value-adding activities through which the product flows from supply side to demand side. According to Porter (1985) the organisation derives its competitive advantage from the efficiency of the organisation and the performance of these activities. Underperforming activities or activities that do not add to the competitive advantage are candidates for outsourcing. By outsourcing an activity to another organisation the business establishes a relationship with that organisation and thereby marks the beginning of a network of businesses. Some successful business networks are companies like: Cisco, Dell, E-bay and Schwab (Hacki and Lighton, 2001; Christopher, 2005). There is an important difference between the value chain and the value network: in a value chain the specification of products start at the supply side, where in a value networks they may start with the customer's requirements at the demand side (Delporte-Vermeiren, 2003). Therefore, a value network not only supports a *supply-driven market* in which the suppliers push predefined products or services into the value chain, but also a *demand-driven market* in which the customer pulls the products or services from the value network (Hadaya and Cassivi, 2008). The customer may combine the products or services offered by the suppliers into the product or service required. In this situation, the value chains are dynamically and temporarily constructed within the value network to provide the customer with the desired combination of products and services. These dynamics of a value network increases the complexity of the business processes of the demanding and the supplying actors.

Author	Definition of a business network
Thorelli (1986)	Networks are two or more organisations involved in long-term relationships.
Miles & Snow (1992)	In a dynamic network, numerous firms are operating at each of the points on the value chain, ready to be pulled together for a particular customer order and then disassembled to become part of another temporary alignment.
Hoogeweegen (1997)	A business network involves a large number of actors contributing to providing service offerings triggered by actual demand based on their core capabilities.
Christopher (1998)	The network of connected and interdependent organizations mutually and cooperatively working together to control, manage and improve the flow of materials and information from suppliers to end-users.
van Heck and Vervest (2008)	A smart business network uses an open digital platform to do business with anyone, anywhere, anytime, despite different business processes and computer systems to gain results that normal business network are not be able to achieve.

Table 2: Definitions of a business network.

A short historical overview of definitions for business networks is given in the table above. The overview starts with a definition that allows for a network that exists of two organisations. The overview ends with a definition that incorporates the use of information technology. In this thesis the broad definition of van Heck and Vervest (2008) is used as the definition for a business network because it explicitly mentions the use of information technology as an enabler for a business

network. The definition fits well with the previously given definition for business flexibility, because the use of a digital platform can be used to craft flexibility. The adverb “smart” is added to indicate that business network supported by an open digital platform are able to generate exceptional or “smart” results (van Heck and Vervest, 2008). The definitions given above result in the following characteristics of a smart business network:

- it consists of multiple actors
- it is capable of delivering composite products and services
- it is supported by an open digital platform

The actors are the businesses of the network. The constellation of actors may vary over time, because business join and leave the network. The composite products and services are products and services combined by the customer. They may consist of components provided by different suppliers. The open digital platform links the information systems of the individual actors and enables the communication between the business processes of those actors.

2.3 Business processes in a smart business network

An established smart business networks might not only enable outsourcing whole activities, like in Porter’s value chain, it is able to outsource individual business processes or even sub processes. As a result these processes run across different business domains and are dynamically linked in rapidly changing constellations (Heck and Ververst, 2008). This dynamic environment requires the business processes to be flexible. The flexibility at business level requires:

- flexibility at information technology level
- alignment between the business level and the information technology level

The flexibility at information technology level requires a flexible open digital platform that supports the business processes of the business network. The flexibility at information technology level is addressed in the next chapter. The flexible open digital platform is discussed in chapters four, five and six. In this section the alignment between the business level and the information technology level is discussed from a business architecture perspective (Versteeg and Bouwman, 2006). The business architecture perspective is suited because:

- it realizes a clear alignment between the business level and the information technology level
- it uses a business domain as the main element of analysis
- it is meant to be used across multiple businesses

Business architecture (Figure 2) links the business processes at the business level to the open digital platform at the information technology level. The business processes are defined using business functions and business objects. Examples of business functions are: booking a course or shipping a product. The business functions are defined at a high granularity level: the individual steps are not described. Examples of business objects are: course description or booking information. The business objects are also described using high granularity level: no detailed data definitions are included. The business processes are decomposed into tasks. The business functions and objects are decomposed into information technology functions and objects. Functional decomposition is used to decompose the business functions into information technology functions. Data decomposition is

used to decompose the business objects into data objects. These information technology functions and data objects are provided by the open digital platform of the smart business network.

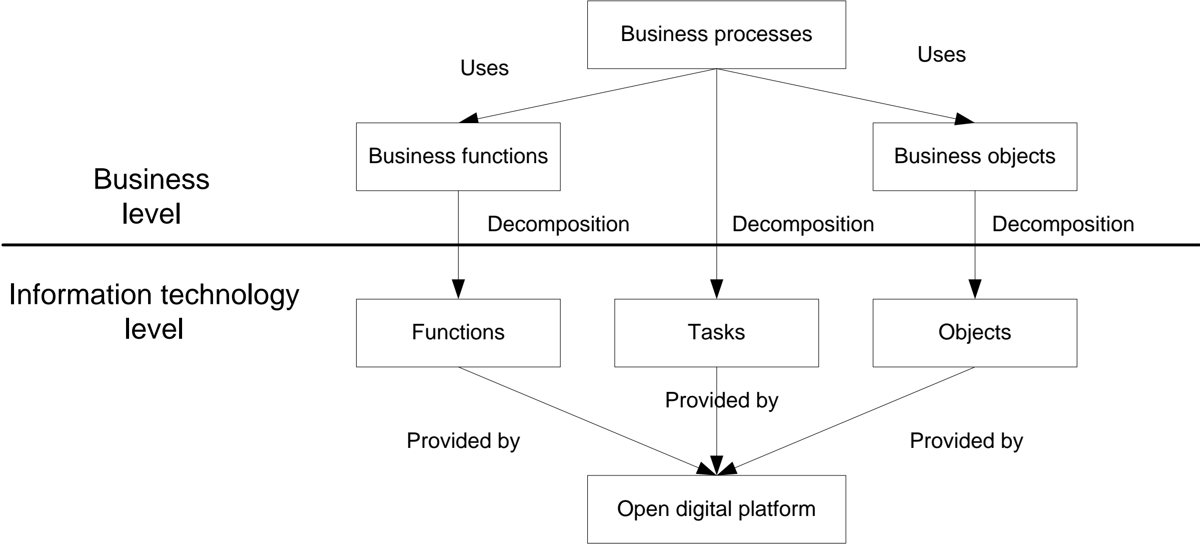


Figure 2: Alignment between business and information technology.

In a multi-actor environment the actors may have a different perception of the business domain and its processes. Each actor may have different definition of the business functions and objects within the domain. Therefore it is important to explicitly deal with the differences amongst the actors. Within this thesis the concept of a conceptualization[3] is used (Figure 3). The conceptualization represents the view of the actor upon the business domain. It contains the actor's definitions of the business functions and the business objects. When dealing with multiple actors there might be different perceptions of the domain, subsequently there may be multiple conceptualizations.

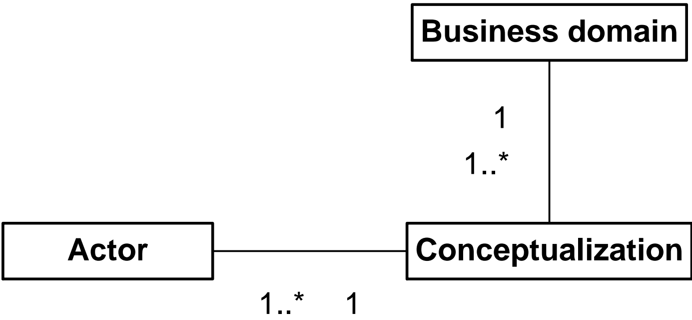


Figure 3: The relation between a business domain, the actor and the conceptualization[4].

In a smart business network the actors need to connect to the network efficiently: their business processes need to be compatible and the information needs to be portable (Ververst and Zheng, 2008). This situation requires a reasonable amount of consensus amongst the actors about the

[3] This concept is further elaborated in chapter five.
[4] The diagram is constructed using the Unified Modeling Language (Booch et al., 2005).

definitions of business functions and objects. If this consensus is reached, the actors may share the same conceptualization. However, this consensus might not always be reached. And even if consensus is reached and their definitions comply with an agreed upon standard, interoperability is not guaranteed Egyedi (2007). Therefore the conceptualizations should:

- allow for variations in the definitions of the business functions
- allow for variations in the definitions of the business objects

These variations should be *small enough to preserve the efficiency* and *large enough to allow for differences amongst the actors*.

2.4 Introducing multi-actor complexity

Reaching consensus amongst the actors about the business processes and information takes time and requires understanding of the characteristics of a business network. De Bruijn and ten Heuvelhof (2004) identify four major characteristics. First, the network is pluriform: there are many actors and they vary in size, market power, product portfolio and other organizational characteristics. In a pluriform network different actors might respond differently to a proposed solution. Nevertheless, providing custom-made solutions for the individual actors is an enormous task and most likely not viable. Implementing many custom made solutions takes time and thereby reduces the flexibility of the network. An increase in consensus among the actors might reduce the task at hand. Consensus amongst the most important actors will probably be sufficient. If they support a solution and the solution is successfully implemented, the other actors may follow the example and implement the solution as well. Second, most of the actors in a network can be characterised as closed with respect to incentives or solutions. This closedness means the actors are focused on their "core values", values deeply rooted in the organization. This focus makes it difficult to reach consensus about a solution: each actor will try to make the fit between the actor's focus and the solution as comfortable as possible. Increased negotiation amongst the actors takes time and thereby decreases the flexibility of the network. However, once the solution fits, the relationship might be a long one. Within a business network the actors have at least a partial common interest, which might be a key to overcome the closedness. Third, the actors in a network are interdependent. They become dependent upon each other with respect to, for example: knowledge, money and information. This set of dependencies might be very complex. This complexity makes it hard to estimate the effectiveness of proposed solutions and to determine which actors to involve. This decreases the speed of decision making and thereby decreases the flexibility of the network. However, the more dependencies in the network the more opportunities there exist within the network. Fourth, the network is dynamic. The constellation of actors involved changes over time. Actors leave the network, new actors join. The existing actors may also change: their focus might shift, preferences change and certain objectives become important or unimportant. The characteristics of networks indicate that the flexibility of the network is an ongoing complex issue. There are many barriers and opportunities and it's not easy to find or negotiate the right equilibrium.

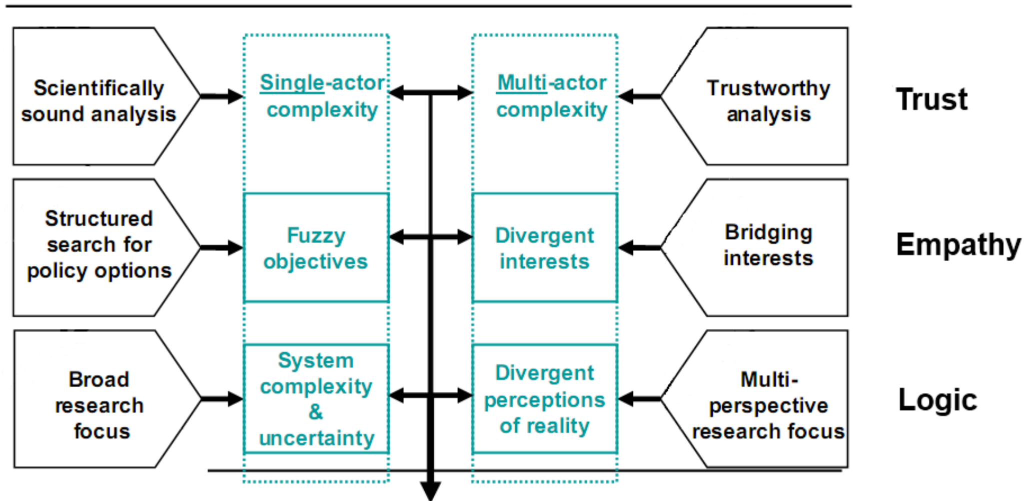


Figure 4: Framework for producing knowledge in a multi-actor setting (van Riet, 2003).

The actors in the network require knowledge for the negotiation and decision making processes. Decision-making requires useful knowledge, whether in a single actor environment or in a multi actor environment. In a multi-actor environment more requirements that determine the usefulness of the information must be fulfilled (van Riet, 2003). According to Van Riet (2003) acquiring useful knowledge in a multi-actor setting requires trust, but also empathy and logic (Figure 4). The term trust addresses the quality of the information and the independency of the analysis. It contains requirements like giving the stakeholders a voice and including analysts they trust. Trust also includes using the correct data, methods and models and making them accessible for all the stakeholders. Exchanging the information may also reduce misunderstandings. The trust requirements are about being transparent. The empathy requirements address the consensus about the solutions and respecting the interests of the stakeholders. It addresses requirements like the clarifying of fuzzy or contradictory objectives and systematically searching for solutions. Empathy also includes taking a broad scope to maximize the benefits and minimize the losses and identifying irresolvable differences. The empathy requirements are about creating win-win situations. The term logic addresses the rationality and completeness of the analysis. It contains requirements like including all relevant parameters and dealing with uncertainties. Logic also means including all relevant features for the stakeholders and identifying diverging views on assumptions. The logic requirements are about taking a multi-perspective focus.

Besides the requirements for the production of useful knowledge there are requirements for the decision making process. If the knowledge is uncontested by the actors but at the same they contest the decision making process, then the decisions probably will not be accepted by all actors. De Bruijn and ten Heuvelhof (2002) identified four core elements for the design of successful processes (Figure 5). The openness of the process indicates that the process and its management should be transparent and that all actors may place items on the agenda and all decisions about those items should be multilateral. The protection of core values of the involved actors is needed in an open process to keep the actors committed to the process. The decision making process should protect the main interests of the actors. The actors should commit to the process and the result of the process should not harm their core values. The actors should have an exit option which they can use if they feel that they cannot commit to the process and the result of the decision making. Careful

negotiations take time. Nevertheless the process should have sufficient speed because the actors involved are in need of decision. To increase the speed of decision making the process should stimulate cooperation between the actors and the actors should have commitment power. The process should have substance or meaning, otherwise the decisions are not useful. The risk exists that the actors agree upon the obvious or that the actors negotiate an agreement which contains contradicting compromises or an agreement that nobody really wants. In those cases no real decision has been made, the items will probably reappear on the agenda.

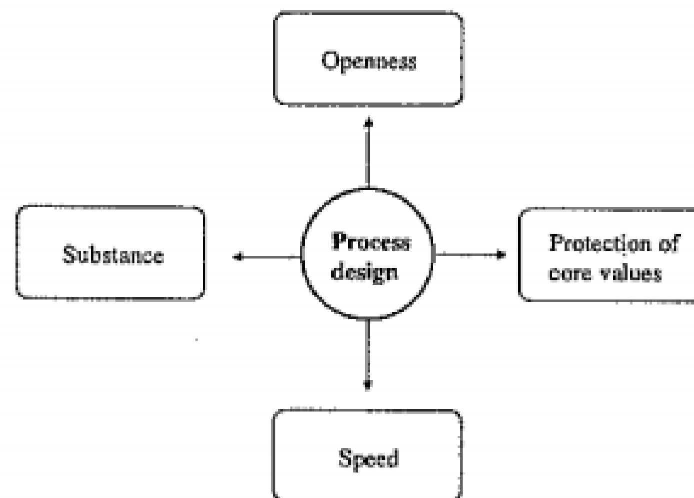


Figure 5: Core elements for process design (de Bruijn and ten Heuvelhof, 2002).

2.5 The Internet Business Learning Community

In this thesis the Internet Business Learning Community (IBLC) is used as a case study. The business environment of the IBLC is the human resource market. This market is changing rapidly, there is increased complexity and competition (Roehling et al., 2000) and the speed of these changes is high (Bessems, 2009). The IBLC is increasing the innovation in the human resource market in order to cope with these changes. The IBLC is currently transforming itself into a central hub within a business network. Before the characteristics of the transition are described, a short history of the IBLC is provided as background information.

The IBLC started almost fifteen years ago as a physical shop. The shop provided telephone and face-to-face advice to anyone interested in corporate training and development courses. To that end a large database containing course information was designed and maintained. The first transition was an addition to the shop in the form of an electronic marketplace. An electronic marketplace provides the business, its customers and its suppliers the ability to interact over the internet and perform transactions (Holzmüller and Schlüchter, 2002). The core of the marketplace consisted of the database, an internet application to access the database and various processes to support the transactions of bookings. During that phase IBLC could be described as a click-and-mortar (Adelaar et al., 2004), operating as both a physical shop and an internet shop. During the next transition the physical shop was closed and the company focused on the internet, positioning itself between the suppliers of courses and the customers. Those customers were mainly human resource departments in need for a solution to manage the training of the company's employees. The suppliers were mainly the training institutes operating in the Netherlands. The IBLC utilizes the internet as the medium for

collaboration, in which each member maintains and verifies their information. This reduces the burden to maintain a central database and it also increases the currency and accuracy of the information. Currently the network consists of about two hundred organizations.

As described above, the position of the IBLC shifted from an intermediate between a customer and a supplier to a central hub within a network with many customers and suppliers. This shift has consequences for the operations of the IBLC and for the system supporting the operations. The system currently in use is primarily designed to support the processes of the IBLC. The desired system should also support the processes of the partners of the community. This is a focus shift from a single-actor system to a multi-actor network, which increases the complexity of the system. A network consists of nodes and arcs, in a business network the organisations are the nodes and their mutual relations are the arcs (Figure 6). The manifestation of the relations can be the exchange of information, the flow of products or the provisioning of services. The complexity is also increased by the diverse nature of the dependencies, for example: products, services, knowledge and money.

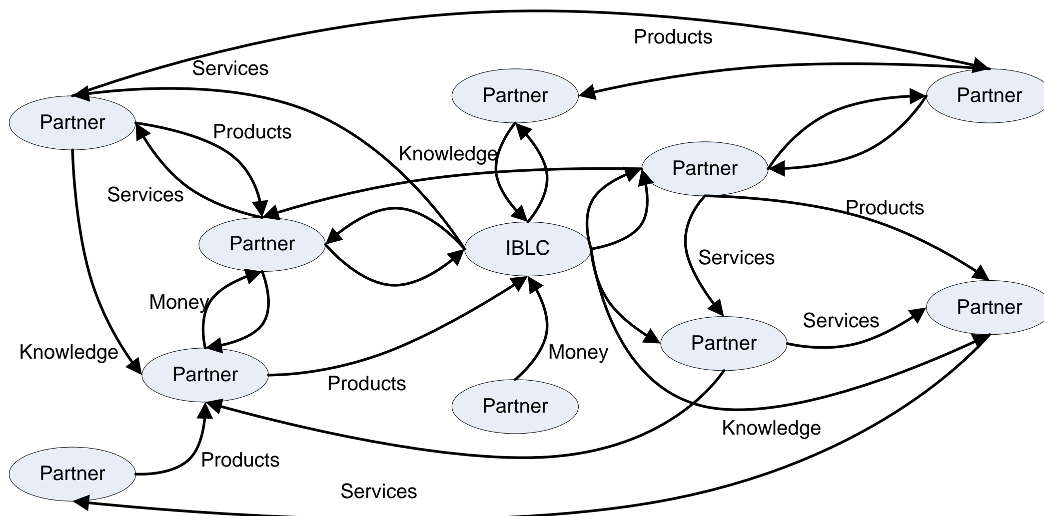


Figure 6: A schematic abstraction of a network of organisations.

Bessems (2009) expects the human resource market to shift more towards a demand-driven market, in which the customer pulls the products or services from the network. This shift requires the network and its supporting system to support composite products. A composite product is a product that the customer can construct on demand by assembling components from different suppliers. The relation between a product and its suppliers changes a hierarchical structure to a network structure. In the former situation, one supplier owned the product. In the later, many suppliers together own parts of the product. Within the network the IBLC takes most of the initiatives for innovation, it provides services to other actors and it supplies the software. The organizational unit IBLC Infrastructure is responsible for most of the software used by the IBLC. The supplying partners include training companies like: NCOI, ISBW, LOI, Oracle and Schouten & Nelissen. The buying partners include companies like: Philips, Aegon, Atos Origin, the province of Utrecht and the Ministry of Foreign Affairs. Other partners include companies like: Daywize, Workwide and Yiller.com. These companies provide all kinds of service related to human resource management.

2.6 Conclusion

Organisations have to become more flexible because of the rapid changes in the environment in which they operate. The smart business network emerged as a new organisational paradigm to enable business to become more flexible. In a smart business network the organisation is part of a network of interdependent organisations, competing against other business networks. The open digital platform used by those networks enable them to do business with anyone, anywhere, anytime, despite different business processes and computer systems. Within a smart business network, the customers may combine the products or services offered by the suppliers into the product or service required.

Business requirements for the open digital platform of a smart business network
It should support the conceptualizations of multiple actors: • allow for variations in the definitions of the business functions • allow for variations in the definitions of the business objects
It should support the delivery of: • composite products • composite services

Table 3: Overview of the business requirements.

Flexibility at business level requires flexibility at information technology level and alignment between the business level and the information technology level. The alignment between the business level and the information technology level is discussed from a business architecture perspective. Business architecture links the business processes, defined using business functions and objects, within a business domain to the open digital platform at the information technology level. In a multi-actor environment the actors may have a different perception of the business domain and its processes. A conceptualization is used to explicitly deal with the differences amongst the actors. The conceptualizations should allow for variations in the definitions of the business functions and objects. These variations should be small enough to preserve the efficiency and large enough to allow for differences amongst the actors.

3 Software flexibility

3.1 Introduction

Business and information technology alignment enables the effective and efficient use of information systems by the business (Henderson and Venkatraman, 1993). If the alignment between the business level and the information technology level is sub optimal there is a gap between what the business requires and the information system delivers. Increasing the flexibility of software may decrease the alignment gap (Tallon, 2008). If the software is not flexible then the business processes supported by the software are not easy to change. But what is software flexibility? And what instruments affect this flexibility?

To answer those questions an abstract model of the system is used (Figure 7). The abstract model in the figure below relates the actors in the business network to the software system that supports the processes of the network. In chapter one the objective of the research is stated as designing a flexible software system for a smart business network. In chapter two this objective is related to two specific business requirements: support multiple conceptualizations and support composite products and services. The external factors of the model are the changes in business environment and technology innovation (Chapter two). This chapter provides a workable definition of software flexibility useable within this research project. The definition is used to identify the quality indicators for the software system. The definition and the indicators are derived from studying scientific literature in the field of software engineering. The definition and the indicators are used to search for relevant instruments to influence the flexibility of the system. These instruments are derived from software management and software engineering theories.

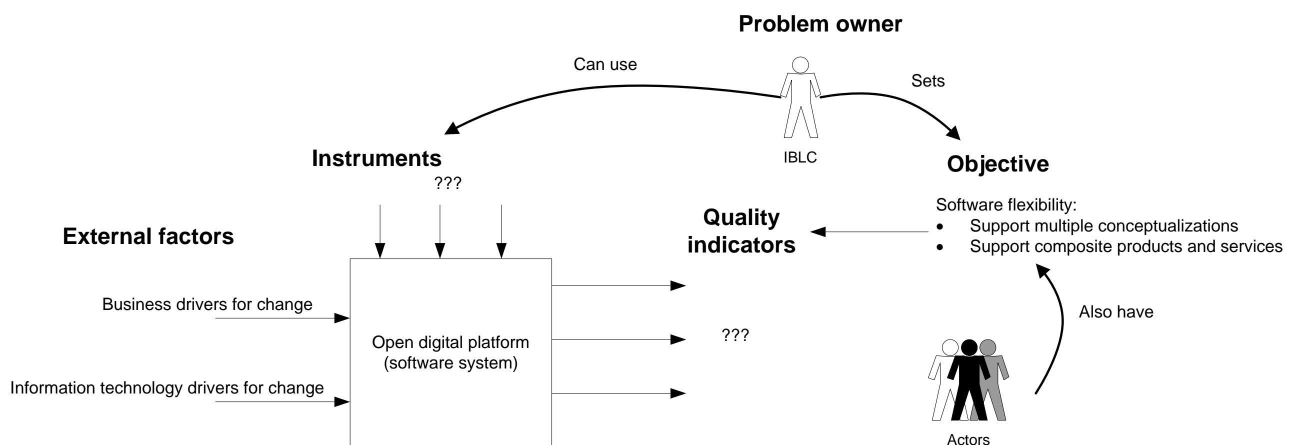


Figure 7: Abstract representation of the open digital platform.

3.2 A workable definition of software flexibility

DeLone and McLean (1992) categorize the flexibility of an information system as an indicator for the quality of that system. They examined and categorized quality indicators for software systems, like: reliability, accuracy, usability and many more. They found that most of them are "fairly straightforward", but at the same time they can be interrelated and interdependent. This interrelatedness and interdependence makes defining software flexibility not easy. However, there is

reasonable consensus in literature about a broad definition: the ease with which a system can adjust to change (Table 4). In this thesis the definition by Egyedi and Verwater-Lukszo (2005) is used: flexibility is the ease with which a system adjusts to changing circumstances and demands. This definition corresponds well with the previously given definition of business flexibility, since it mentions the source of the changes: circumstances and demands. This corresponds well with the dynamically competitive environment in which the business operates. Another advantage of this definition is that does not mention modification. There are more ways to adjust a system then just modification. Some systems, for example, can be adjusted by tuning the parameters of the system, without applying any changes to the system's structure.

Author	Definition of software flexibility
Barbacci, et al. (1995)	The ease of adapting software to new requirements
Nelson and Nelson (1997)	The ability to adapt to both incremental and revolutionary change in the business or business process with minimal penalty to current time, effort, cost, or performance.
Egyedi and Verwater-Lukszo (2005)	Flexibility is the ease with which a system can adjust to changing circumstances and demands.
Stojanović (2005)	Flexibility enables easy modification to meet changing needs.
Wang, et al. (2008)	Flexibility allows a product to be modified rapidly and cost-effectively for new needs.
Philips and Wright (2008)	Flexibility is the ability to adjust e-business processes to customer preferences.

Table 4: Definitions of software flexibility.

The problem with the definition of flexibility is that it is too broad and implicit. It affects almost all aspects of a software system. Therefore a literature study is performed to decompose the definition of flexibility into a set of quality indicators for software systems. These quality indicators are the indicators to estimate the effectiveness of the instruments (Figure 7). Because of the interrelated and interdependent nature of the indicators, numerous authors use them in slightly different ways. For this thesis the most suitable definitions are selected with respect to the context of a smart business network. Suitable can be described as to minimize the dependency between the indicators. This decomposition is by no means complete and the definitions are rather abstract. However, the study yields a workable set of quality indicators (Table 5).

Versatility and agility

In their research to create an assessment tool to measure the flexibility and profitability of business networks, Delporte-Vermeiren (2003) defined flexibility of a business network as a combination of the versatility and the agility of that network. The versatility is defined as a measure for the number of different products or services the network can deliver. The versatility increases as the number of different products increases. And the agility is defined as a measure of the throughput time of the network[5]. The faster a network can produce a product the lower the throughput time. The throughput time is a combination of the time it takes to link the actors in the network and the time it takes for the actors to deliver their part of the product or service. The research of Delporte-Vermeiren (2003) is focused on the economic aspects of the business networks, whereas this thesis is focused on the software architectural aspects.

[5] Other definitions are also common in scientific literature (Ribeiro, 2008). In this thesis agility is defined according to Delporte-Vermeiren(2003).

Connectivity

In their book about design of service systems, Van de Kar and Verbraeck (2007) use flexibility to describe the ease with which new actors can be added to the system. The same concept is used in the previously given definition for smart business networks to describe the digital platform: "it must be open and enable the businesses in the network to do business with anyone, anywhere, anytime". This concept is called connectivity and it works at the physical, syntactic and semantic level (Uschold and Gruninger, 2008). Connectivity and the previous defined agility both address the connection between an actor and the network. The main difference is that agility concerns the speed of the link with an existing actor and connectivity concerns the ease to connect new actors.

Interoperability

The previously given definition for smart business networks indicates the existence of another indicator. The phrase "despite different business processes and computer systems" describes interoperability. Jansen and Scholl (2004) define interoperability as the ability of diverse systems and organizations to work together. Traditionally software interoperability deals with the problems of multiple software components written in different programming languages to communicate and interact with one another (Wileden and Kaplan, 1999). It addresses problems like portability: the reuse of complete applications on new platforms (Mooney, 1995) and vendor lock-in: the customers buying choices are tied to an original purchase for a related product (Shah et al., 2008). Some of the problems created by the lack of interoperability are mitigated by using vendor independent standards (Shah et al., 2008) and virtualization (Crosby and Brown, 2006).

Reusability

The definition of software flexibility states that flexibility is the ease with which a system can adjust to change. If a software system supports a business process, then ideally that system should be useable even if the business process changes. And if the software system is not entirely useable, it should be easy to adjust. This quality indicator is called reusability. Jacobson et al. (1997) define reusability as the future use or repeated use of software, designed for use outside of its original context. Reusable software can not only respond to changes in the original business process, but it is also usable in other business processes, than the process it was originally specified for. Reusability of software speeds up the time to market, increases the overall quality and makes it cheaper to produce (Jacobson et al., 1997).

If a software system is not easy to reuse, it should be easy to adjust its configuration and if it is not easy to adjust, it should be easy to modify. The effects of the agility, the versatility, the connectivity, the interoperability and reusability of software are felt even without modification of the software system. With respect to the software system, they can be called external quality indicators (Barbacci et al., 1995). When the software system needs to be modified, internal quality indicators are also important to determine the flexibility.

Reusability is not only an external quality indicator; it's also relevant when the system is modified. If during the modification of the system, parts of the system can be reused, the modification takes less time. The less time it takes to modify a system, the more flexible the system. The external reusability is defined at system level. The internal reusability can be defined at a technical level and it concerns artefacts like: code modules, documentation, test data, requirements, design, code, architectures and whole systems or subsystems that can be integrated along with bought and built software components or parts into a new or existing software product (Rine, 1995).

Maintainability
If the software is not reusable it should be easily modified. This aspect of software is called maintainability. Maintainability is not restricted to code; it addresses the same artefacts as reusability (Broy et al., 2006). Like the other quality indicators maintainability is not clearly defined in literature or practice. Broy et al. (2006) concluded that there is little consensus about what it actually is and how it can be achieved. In this thesis the definition by Barbacci (1995) is used: Maintainability is the ease with which a system can undergo repair and evolution. Maintainability is, for example, increased by clear requirements, good documentation, good test data, clear source code and a good architecture. In general, reducing the complexity of software will increase the maintainability (Swanson, 1999).

Quality indicators	Definition of the quality indicators
Agility	The speed at which the business network handles transactions between the actors.
Versatility	The ability of the business network to process a greater number of different products.
Connectivity	The ease with which new actors can be added to the business network.
Interoperability	The ease of diverse systems and organizations to work together.
Reusability (external)	Future use or repeated use of software, designed for use outside of its original context.
Reusability (internal)	Reusable software components or parts are code modules, documentation, test data, requirements, design, code, architectures and whole systems or subsystems that can be integrated along with bought and built software components or parts into a new or existing software product.
Maintainability	The ease with which a system can undergo repair and evolution.

Table 5: Overview of the quality indicators.

Within this set of quality indicators (Table 5), the overlap in definitions mainly concerns the concept of interoperability. Interoperability directly affects the flexibility as it increases the easy with which systems are able to work together. At a technical level an increase in interoperability decreases the need for conversions and adaptations, thereby increasing the speed of the transactions in the network and thus the agility of the network. If the interoperability of the system increases, it takes less effort to connect new actors. Thus interoperability positively affects connectivity. Also at a technical level interoperability is an enabler for reusability. Interoperability works at different levels, thereby affecting different quality indicators.

3.3 Relevant software management theories

In the previous section the quality indicators are identified. In this section and the next section the instruments that influence the indicators are discovered. The instruments are not discussed individually; they are marked in *italic* within the text. An overview of the instruments is provided in Table 6.

The decision making process described in the previous chapter can be used for decision making in a complex dynamic multi-actor setting. However many new software systems are developed in projects (Collaris and Dekker, 2006) and project management requires a relative stable project environment (de Bruijn and ten Heuvelhof, 2002). De Bruijn en ten Heuvelhof (2002) define the decision making process within a project as hierarchical, linear and structured. However, many software projects are performed in complex technological environments. To deal with the complexity of the technology, the software development projects *use a software development methodology* like

rapid application development (Martin, 1991), rational unified process (Kruchten, 1998), dynamic systems development methodology (Stapleton, 1997), agile software development (Beck et al., 2001) or one of the many others. The methodology used within a project deals primarily with the technical complexities of the construction of software and the changing requirements of a single hierarchically superordinate actor. These software development methodologies are not per se usable in a multi-actor setting.

Software development in a smart business network is an ongoing process. Traditionally new software systems are created in projects and taken into production by the information technology department (Collaris and Dekker, 2006). This department maintains the software and the systems on which the software runs (Rozanski and Woods, 2005). The maintenance of the software is usually performed by other teams than the teams that originally constructed the software (Berns, 1984). Those teams may differ in interpretation of the software. They may even have different training and apply different software development methodologies between maintenance tasks. The complexity and evolution of the software systems that are developed and maintained in such a piecemeal fashion is difficult to manage, which may result in systems that lack flexibility (Egyedi and Verwater-Lukszo, 2005). Maintainability of a system may increase by *applying a software architecture* (Sangwan and Ros, 2008). Using software architecture limits the number of choices a team has to make (Rozanski and Woods, 2005). The applied architecture provides answers to the question: How should the software system be built? Maintainability of a system may also increase by *designing a domain model* (Evans, 2004). The domain model answers the question: What should be built? A good architectural description and a good domain model can be effectively and consistently communicated to the developers and the maintainers (Rozanski and Woods, 2005; Evans, 2002). Furthermore the domain model can increase the shared understanding between business domain specialists and information technology specialists, which might increase the alignment between the business level and the information technology level (Barn, 2009).

The quality of the communication within projects is an important factor for the success of the project (Boonzaaier and van Loggerenberg, 2006). Miscommunication or the lack of communication is also one of the reasons for the misalignment between the business and the information technology (Byrd et al., 2006). In many cases the language spoken is different at the business level and the information technology level. Frequent communication and the exchange of knowledge might reduce miscommunication (Thompson and Ang, 1999). Sharing a common language facilitates the exchange of knowledge (van Onna and Koning, 2003). Such a common languages contain well defined concepts that are understood at the business level and at the information technology level. Those concepts and their relations are called *the common vocabulary*.

In a smart business network a governing authority might reduce the multi-actor complexity. If the organization adheres to the requirements as described in section 3.2, the organization may create a relatively stable environment to run a project. The governing organization publishes decisions and standards negotiated and agreed upon by the stakeholders of the smart business network. The more support there is for the decisions and the standards, the less turbulence in the environments of the software development projects. However, this situation is similar to the waterfall approach of software development (Royce, 1970). In the waterfall model the decisions, standards and models are defined before construction starts. The waterfall model lacks flexibility (Collaris and Dekker, 2006) and in many situations an agile approach to software development might be more successful (Kroll

and MacIsaac, 2006). One of the key aspects of agile software development is the position of the customer. The costumer is an integral part of the project team. However, in a multi-actor environment there are many customers. Integrating a representative of each customer in the project team brings the multi-actor complexity back into the project team. It is important to *keep the multi-actor complexity out of the software development team*.

3.4 Relevant software-engineering theories

The quality indicators are used to identify relevant instruments to control the flexibility of the software system. For each indicator a qualitative analysis amongst the vast amount of software-engineering theories is performed. In this section a short description of the instruments is given per indicator. This might suggest a direct relationship between the instrument and the indicator; however in reality the relationship is indirect and complex. As a result there is some overlap in the descriptions of the instruments.

Interoperability

The quality indicator interoperability indicates the ease of diverse systems and organizations to work together. Janssen and Scholl (2007) define four levels of interoperability (Figure 8). At the *technical level* the interoperability of the infrastructure is defined. It is governed by standards addressing, for example: messaging, security and transactions (Cabrera and Kurt, 2005). At the *syntactical level* the structure of the messages and the services are described. The standards for the syntax level define, for example, the names and the data types for the information exchange. There are many standards for the technical and the syntax levels (W3C, 2009) and there is an organization concerned with the interoperability of the standards at these levels (WS-I, 2009). At the *semantic level* the interpretation of the messages is described. The semantic level defines the exchange of information, where the syntactic level defines the exchange of data. There also exist many standards for the semantic level (Colomb, 2007). The pragmatic level denotes the interoperability at the business level of the organizations involved.

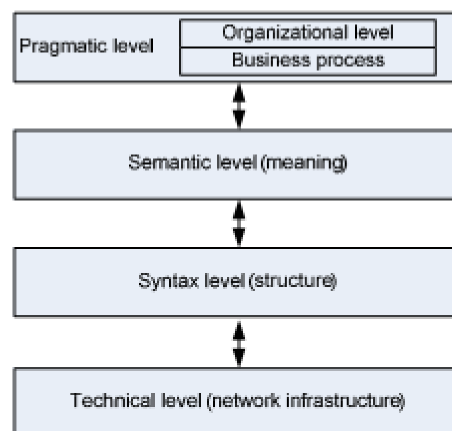


Figure 8: Levels of interoperability (Janssen and Scholl, 2007).

Agility

The quality indicator agility indicates the speed at which the actors of the business network perform transactions. The communication between the systems of the actors should be fast and easy. The speed is affected by the time it takes to link the actors in the network and the time it takes for the actors to deliver their part of the product or service. This indicator is influenced by another quality

indicator: interoperability. The more a network is designed for interoperability, the less data conversions need to take place and therefore the faster the pace of communication in the network. These conversions might be needed between standards at any of the four levels of interoperability (Figure 8). Even if systems are standard-compliant, interoperability is not guaranteed. Egyedi (2007) found that the problem is usually the way the specification is realized in code. These variations in the realization of the code are caused by flaws in the standards and the standardization process. Including realization concerns in the standardization process might reduce the variation in the software, thereby increasing the interoperability of software based on the same specification (Egyedi, 2007). Furthermore, using domain-specific specifications might make specification matching easier (Kratz, 2003). Beside interoperability, the agility is also affected by the time it takes for the actors to deliver their part of the product or service. If it takes the system too long to respond, the system might be overloaded. If a system is overloaded the scalability of the software system becomes an important aspect. Using *virtual environments and machines* increases the scalability of software at the cost of its slower performance due to the high degree of hardware abstraction (Kazi et al., 2000). The reduced performance is compensated by the possibility of running the software on more powerful hardware if the performance is poor. The application of these techniques makes it easier to design scalable and portable software. Less design effort is needed as there are less considerations and decisions to be made. If it takes the system to long to respond the system might even be broken. If a system is broken the reliability of software becomes an important aspect. The reliability of software is influenced by *the quality and the amount of tests* that are performed on the software (Glass, 2008). The test cases performed can range from testing small units of code to complete and overall tests for compliance with the requirements. Furthermore, to test from a reliability point of view, test cases to test for failure free operation of the software system in a specific operational environment can be used (Kundu et al., 2008).

Versatility

The quality indicator versatility indicates the ability of the business network to process a number of different products and services. The indicator becomes important if many slightly different products or services are defined by the actors of the network and when many versions of the products or services need to be supported over time. It is important to explicitly design for variation and indicate that the software is prepared for a certain type of predictable changes (Bachman and Bass, 2001). Versatility can be incorporated in the design of the system in three different ways. First, by using services that can be rearranged. According to Estublier and Vega (2005) experience has shown that variations of products can be successfully supported in systems where domain-specific components can be assembled in different ways. *Reusing existing services* by rearranging them is relatively easy to do if the services are implemented using interoperability standards. The main difficulty is to predict and incorporate future arrangement of the services into the design of the system. Second, versatility can be incorporated in the design by providing runtime configuration. Each request for the service specifies which configuration is applicable for handling the request. The configuration contains a value for each variation point in the software. A variation point is a part of the system that can vary (Koning et al., 2007). The challenge lies in predicting which variation points to incorporate in the design. Third, new products or services can be supported by modifying the source code of the existing services. Compared to the first two ways of increasing the versatility, modification is probably more expensive. It takes time and sometimes even a small project to modify existing

software. During the design the software can be prepared for change by trying to encapsulate the predicted changes in separate components (Bachmann and Bass, 2001).

Connectivity

The quality indicator connectivity indicates the ease with which new actors can be added to the business network. This requires the system to be able to add new systems easily. This indicator is influenced by another quality indicator: interoperability. The more a network is designed for interoperability the easier it is to add new actors. However, if the standards for interoperability are difficult to understand a new actor might find it hard to connect. Therefore *well known and well understood standards should be preferred* (W3C, 2009; WS-I, 2009). On the technical and the syntactical level these standards are commonly used and well integrated in existing commercial development environments (Löwy, 2007; Hansen, 2007). On a semantic level the standards are available, for example: OIL (Fensel et al., 2001), DAML (Darpa, 2009) and OWL (Gašević et al., 2009). However, they are still hard to apply in the design of commercial applications (Wessel and Moller, 2009). *An ontology can be used to enable communication at the semantic level* (De Nicola et al., 2009). It is a formal way to represent domain knowledge (Hartung, 2009) and consists of semantic structures, terms and their relations valid within that domain (De Nicola et al., 2009). The use of an ontology provides two advantages. First, communication at the semantic level enables the future possibility to connect actors to the network without human intervention (Uschold and Gruninger, 2004). Removing human intervention and enabling software to autonomously connect makes it easier to connect new actors to the network (Tweedale and Jain, 2009). Second, besides the use of an ontology in machine to machine communication an ontology can be used in human to human communication. Ontologies may be used as a formal method to communicate intent to the actors involved. It may be used as the common vocabulary, understood and spoken by the actors.

Reusability

The quality indicator reusability indicates the future use or repeated use of software, designed for use outside of its original context. *The basic unit of reuse is a software component.* Such a component is a software product that has been specifically engineered to be reusable (Jacobson, 1997). A component is self contained, is clearly defined, has a clear interface and has a clear reuse status (Sametinger, 1997). By breaking the software up into components, the components can be arranged into a software system (Szyperski, 2002). Rearranging the components yields a system with different behaviour. The important aspect in the design process of a component based system is to minimize the dependency between components and to discover the right size of the components. In a service-oriented architecture the main unit of reuse is a web service (Yang and Papazoglou, 2002). In a service-oriented architecture software systems are dynamically composed by discovering, matching and integrating pre-developed services. These services function independently of each other and communicate by sending standardised messages through standardised interfaces (Janssen, 2005). Reusability of already available resources is believed to be a cost saving factor. Therefore it should be supported and promoted by the system.

Maintainability

The quality indicator maintainability indicates the ease with which a system can undergo repair and evolution. The maintenance of software systems is expensive and often seen as a necessary evil (Swanson, 1999). Therefore the system should be easy to modify. The more difficult it is to understand a software system, the more difficult it is to maintain it (Berns, 1984). The difficulty to

understand a software system is influenced by many factors: unclear requirements, poor design, unstructured coding, insufficient testing, and incomplete documentation (Huang and Lai, 2003). To clarify requirements a *common vocabulary* might be used by all actors involved (Evans, 2004). The common vocabulary is used to discuss requirements, to model the system and to build the software. This common frame of reference might reduce miscommunication (Thompson and Ang, 1999). Requirements are also called unclear if they are not defined at the start of the software development project, or if they shift during the project. The shift *in* requirements should be traceable throughout the software lifecycle. Versioning of the requirements may be used to track the changing requirements. The *iterative development approach* is commonly used as a software development methodology in situations with changing requirements (Martin, 1991; Kruchten, 1998; Stapleton, 1997; Beck et al., 2001). The software is developed in sequential iterations. During each iteration functionality is added and changes, as a result of a new or changed requirement, are applied. This approach enables the developers to neatly integrate the code changes due to unclear requirements in the existing code. This might result in a more consistent code structure. The realization of software can be improved by *using a software architecture* (Rozanski and Woods, 2005). The quality of the design may also be increased by *applying design patterns* (Gamma et al., 1995). These design patterns are well known and well described (Gamma et al, 1995; Carey et al. 2000; Fowler, 2002). However to understand them and apply them correctly is not always easy. A technique called *refactoring* (Fowler, 2000) can be applied if the patterns are not used where they should. Refactoring is a structured way to increase the quality of the design. In small controlled and tested steps the code is reorganised. To test the changes to the code refactoring uses a technique called *unit tests* (Fowler, 2000). Unit tests are used to automatically test the code of the software system. The use of unit tests also increases the maintainability of a software system (Beck, 2003). After each modification to the code, the unit tests should be run and the modified code must pass all the tests to ensure correct operation of the system. To increase the structure of the code, *coding standards* can be used (McConnell, 1993). These coding standards describe the use of variable names, code structure and guidelines (Cwalina and Abrams, 2005) for many other micro decisions that are made during the construction of software (Hunt and Thomas, 1999). Some modern software development environments can automatically enforce the use of coding standards (Levinson and Nelson, 2006).

3.5 Conclusion

Business flexibility requires flexibility in the software systems that support the business processes. There are many definitions for software flexibility, in this thesis it is defined as the ease with which a system can adjust to changing circumstances and demands. It is possible to decompose the definition into a slightly overlapping set of quality indicators: agility, versatility, connectivity, interoperability, reusability and maintainability (Figure 9).

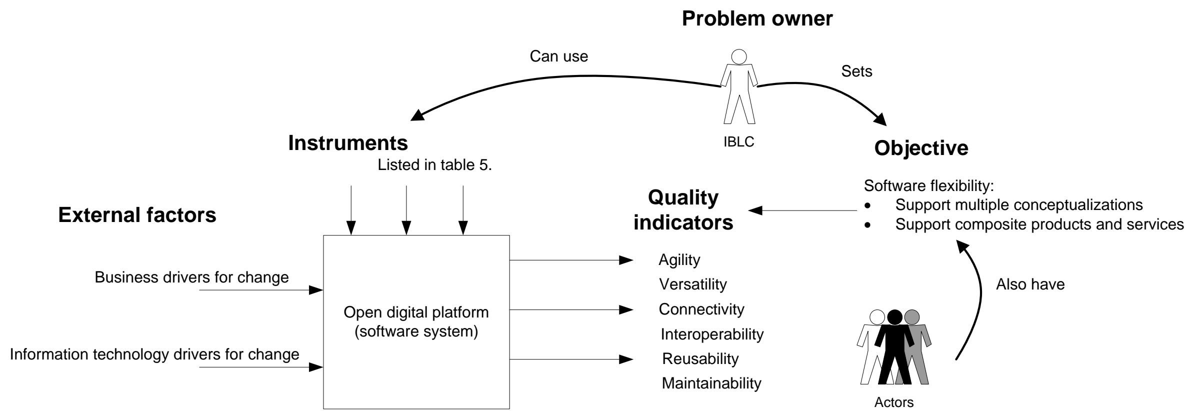


Figure 9: Quality indicators and instruments for the open digital platform.

Many software systems are realized in projects. Project management requires a relatively stable project environment. The software development processes used within a project deal primarily with the technical complexities and the changing requirements of a single hierarchically super-ordinate actor. These software development processes are not per se usable in a multi-actor setting.

The quality indicators are used to identify relevant instruments to influence the flexibility of the software system. The indicators are used as a starting point to find instruments that might influence them. The relationship between the instrument and the indicator is complex: applying a specific instrument might affect more than one indicator. An overview of the instruments is presented in Table 6. The words between round brackets are possible implementations of the instrument, explicitly mentioned in this chapter.

Instruments to increase the flexibility of the software system[6]	Primarily related quality indicator(s)
1. apply a software architecture	Maintainability
2. define a common vocabulary (ontology)	Interoperability
3. design a domain model	Maintainability
4. use a software development methodology (iterative)	All indicators
5. keep the multi-actor complexity out of the software development team	All indicators
6. trace the requirements throughout the software lifecycle (versioning)	Maintainability
7. use and reuse software components (services)	Reusability and versatility
8. use well known technical, syntactical and semantic standards	Connectivity
9. use virtual environments and machines	Agility
10. apply design patterns, refactoring, tests and coding standards	Maintainability

Table 6: Overview of the instruments.

[6] The instruments are not ordered to priority. The numbers are for referencing only.

4 Towards an ontology-driven software architecture

4.1 Introduction

In this chapter the ontology-driven software architecture is introduced. The architecture describes the models that are needed to specify and realize the software system for the smart business network. The architecture acknowledges the existence of multiple actors and uses an ontology to represent an actor's view upon the domain. Each actor may have its own view or a group of actors may share a common view.

4.2 Software architecture

In chapter two, business architecture is used to link the business processes of a smart business network to the software system that supports those processes. Business architecture is an architectural framework at enterprise level. Enterprise architecture addresses many aspects of the enterprise, for example: the business aspects, the information aspects, the application aspects and the technology aspects (The open group, 2009). Whereas software architecture addresses only one aspect: the organization of the software system. The ISO/IEC standard 42010[7] (ISO/IEC, 2007) defines software architecture as: "the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution". The software system of the open digital platform provides the tasks, the functions and the objects that are needed to support the business processes of the smart business network (Figure 2). Key aspect of the business architecture approach is the linkage between the processes of the business and the software system that support these processes (Versteeg and Bouwman, 2006). To keep this linkage clear, the software architecture for the open digital platform should support the link between:

- The business processes and information technology tasks.
- The business functions and the information technology functions.
- The business objects and the information technology objects.

In literature on information technology many definitions of software architecture can be found (Kande, 2000). The differences are noticeable in the vocabulary used, the documentation techniques, the guidelines and best practices (Hofmeister et al., 2006). In general, software architecture describes:

- The views, models or blueprints for the system. For example: RUP 4+1 architecture (Kruchten, 1995), ISO/IEC standard 42010 (ISO/IEC, 2007) or the viewpoint catalogue (Rozanski and Woods, 2005).
- The distribution of the components of the system. For example: Client/Server (Fielding, 2000), multi tier (Malveau and Mowbray, 2001) or services (Erl, 2008).
- The driver or central source of reference for the system. For example: attribute based (Bass et al., 2003), event-driven (Michelson, 2006) or model-driven (Gašević et al., 2009).

[7] Technical identical to standard: IEEE 1471-2000.

The view, model or blue print represents a structural aspect of the architecture (Rozanski and Woods, 2005). Separating the aspects of the software system into multiple views allows for focussing upon a specific aspect of the system. Dijkstra (1974) calls this design principle the "separation of concerns". The views also allow for the separation of the functional and non functional requirements of the system (Kruchten, 1995). The distribution of a software system concerns the way the components of the system are partitioned over the available hardware. The driver or central source of reference represents the overall architectural style or method of design.

Given the discussion above, the architecture for the open digital platform can be specified twofold. First, the architecture should link the business processes, functions and objects to the information technology tasks, functions and objects. Second the architecture should use a domain model (instrument three[8]) as the blue print, use services (instrument seven) for the distribution of the system and use an ontology (instrument two) as the main driver.

4.3 The ontology

In a multi-actor environment the actors may have a different perception of the business domain and its processes (chapter two, section three). Within this thesis the concept of a conceptualization is used (Figure 10). The conceptualization represents the view of the actor upon the business domain. Within this thesis the conceptualization is the starting point for the specification of the software. That is: each specification of the software belongs to one domain and one actor or group of actors. The conceptualization can be defined by constructing an ontology for that conceptualization (Figure 11). An ontology is an explicit and formal specification of a conceptualization (Gruber, 1993).

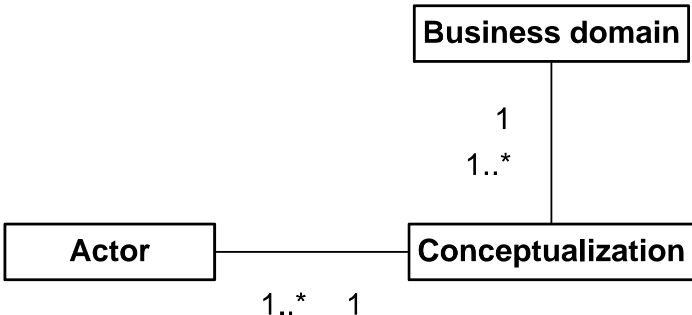


Figure 10: The relation between a business domain, the actors and their conceptualizations.

An ontology consists of a list of terms and their relations. The terms are classes of objects and can be classified in a hierarchical structure. This classification is called a taxonomy. The relations within a taxonomy are called: subtype and supertype relationships, or parent-child relationships. The relations outside the taxonomy can specify logical relations between objects. These relations include: properties, value restrictions and disjoint statements. The ontology determines the scope of the conceptualization. Everything expressed in the ontology is in scope. Everything else is out of scope. When dealing with multiple actors and multiple conceptualizations, there might be multiple ontologies for a specific domain. Translating an ontology into another ontology is called mapping ontologies (Antoniou and van Harmelen 2004). These mappings are important when two or more

[8] See chapter three, table six.

actors, using different ontologies, want to exchange information. The mappings enable the communication between those ontologies, therefore they enable the communication between the conceptualizations and thus between the actors.

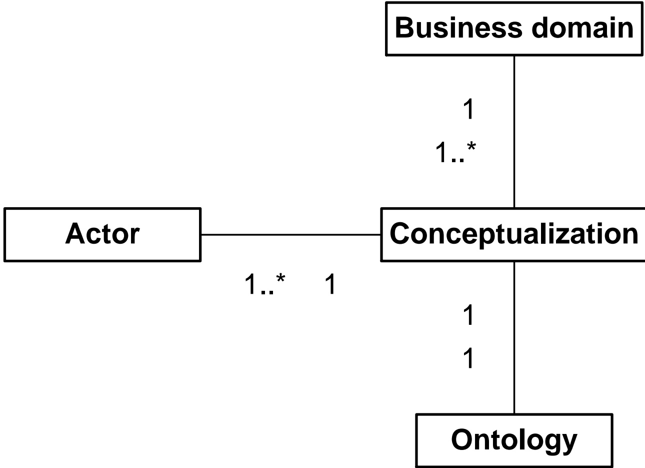


Figure 11: Relation between the conceptualization and the ontology.

4.4 Linking the ontology to the software

The design of flexible software for a smart business network is approached from a synthesis of two main theories of software engineering: service oriented architecture (Erl, 2008) and object oriented software development (Fowler, 2002; Evans, 2004; Larman, 2005). The service oriented architecture theory describes how functionality and data are decomposed and partitioned across a set of services. The object oriented software development theory describes how a service can be designed.

In a service oriented architecture the functionality of a software system is partitioned over a set of services. The main motivation for this partitioning is the promise of an increased reusability of a service in particular and an increased flexibility of the system as a whole (Erl, 2008). Partitioning functionality over services might also reduce the complexity of the individual services. However the overall complexity of the system might increase. Erl (2008) defines three layers of abstraction to decrease the number responsibilities of any given service. The less responsibility a service has the less complex it is. Each service belongs to one service layer only (Figure 12). The task service layer contains services that directly support the process of the business. Examples of task services are: searching for a specific course, registering a trainee for a course or invoicing a customer. The services of the task service layer are related to the business architecture described in chapter two. The services in the task service layer are the services that are responsible for providing the tasks that make up the business process. The task services implement the information technology functions and use the information technology objects (Figure 2). The responsibilities and the size of each service are determined by the decomposition performed to decompose the business process into information technology tasks. The entity service layer contains services that are responsible for the retrieval and storage of entities. Examples of common entities are: Account, Customer, Booking or Course. The entities encapsulate the information that is needed by the business. These services also implement the business logic that is specific to the entity. The services of the entity service layer are related to the business architecture described in chapter three. The services in the entity service layer are the services that are responsible for providing the information technology objects (Figure

2). The responsibilities and the size of each service are determined by the data decomposition performed to decompose the business objects into information technology objects. The utility service layer contains services that support the services in the task or entity service layer. These utility services do not represent business logic. They encapsulate functionality like: logging, event notification or exception handling. This kind of functionality is needed by most services. Isolating the functionality into a set of special services enables the reuse of functionality. And more importantly it reduces the complexity of the services in the task and entity services layers, by moving common functionality out of those services and into the services of the utility layer. However, it increases the dependency between the services and utility service layer.

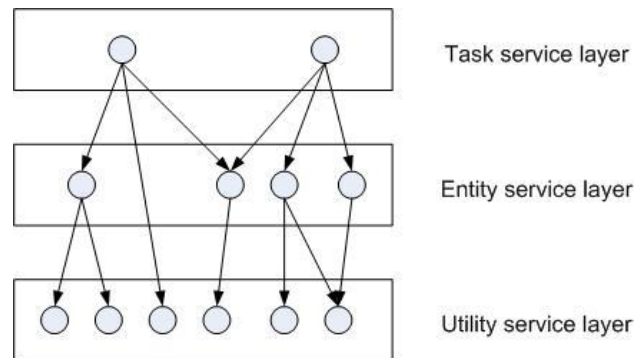


Figure 12: Service layers and the dependency between services (Erl, 2008).

The concept of layering also applies to the design of an individual service. According to Fowler (2002) layering is the most common way to handle the complexity of software systems. Each layer has its own responsibility and functionality. Evans (2004) identifies an architecture of four layers (Figure 13). A service can be designed using those four layers. The presentation layer is responsible for communication with external actors or external systems. When communicating with an actor, the layer may present a graphical user interface. When communicating with another system the layer presents an application programming interface. The presentation layer may contain multiple presentations of the same information. Therefore it is able to simultaneously serve multiple actors and systems with different needs. It also supports multiple versions of the same interface. The application layer is responsible for the coordination of tasks and the delegation of work. This layer contains definitions for the workflow. It is also responsible for translating the domain objects into presentation objects and vice versa. The domain layer represents the concepts defined by the ontology. The ontology represents the actor's view on the domain and defines the scope of the system. Therefore, this layer is the most important layer of the system. The infrastructure layer provides technical services to the other layers. These services may provide access to: the database systems, the file systems, the E-mail servers or the web services. The infrastructure layer hides the technical complexity of accessing those external systems. Therefore the infrastructure layer provides the domain layer with a consistent and easy way to access the information contained within and the services offered by these external systems. Note that in this architecture there are two layers communicating with external systems: the presentation layer and the infrastructure layer. The difference is that the presentation layer responds to requests from external systems and the infrastructure layer places requests upon external systems.

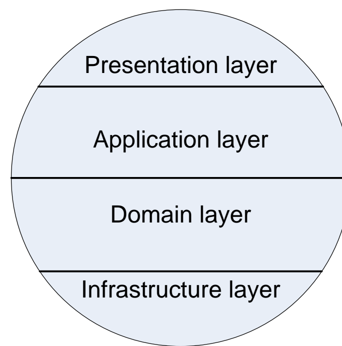


Figure 13: The layered software architecture (Evans, 2004) within a service.

Combining the service layers from the service oriented architecture with the layered software architecture from the object oriented software development results in the specification for the architecture for the prototype (Figure 14). The functionality of the system is partitioned over services and also partitioned within a service. Each service belongs to a specific service layer and consists of software layers.

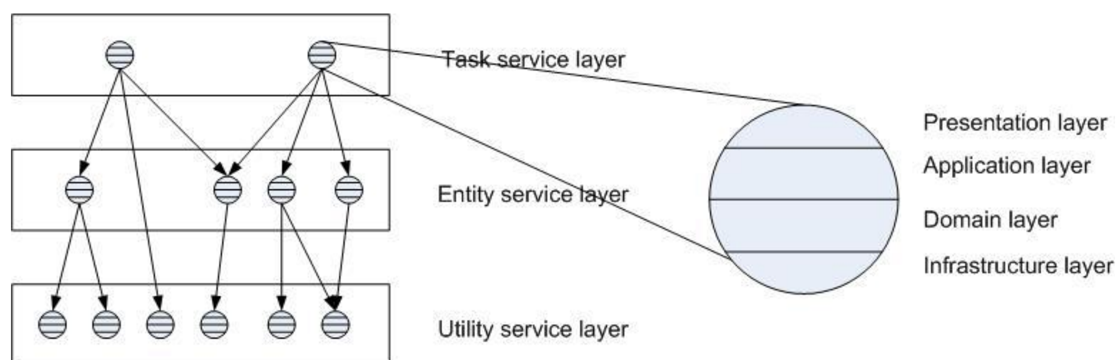


Figure 14: Synthesis of the service layers and the layered software architecture.

In this thesis the ontology is linked to the layers by using models, which specify the layers. Four models are identified: the interface model, the use case model, the object oriented model and the relational model (Figure 15). The ontology describes terms and their relations. It provides the vocabulary that can be used in the specification of the models. All models should be valid with respect to the definitions in the ontology. If a model requires a redefinition of a term or uses a term in an unsupported relation, the ontology should be modified and all other models should be revalidated.

The behaviour of the system is described in the use case model. A use case model is a description of the interactions between the actor and the system (Cockburn, 2000). In the form of scenarios the action of an actor is described in a textual format. The responses of the system are also described in text. The text should be understandable to the actors (business level) and to the team that realizes the system (information technology level). The use case model is the contract that links the business to the information technology. It captures and contains the requirements placed upon the system by the business. A disadvantage of using textual descriptions of requirements is the risk of

misinterpretation. To reduce this risk, the ontology is used as the vocabulary used in the textual descriptions of the scenarios. This way of formalizing the scenarios will hopefully result in less misunderstandings and misinterpretations of the specifications.

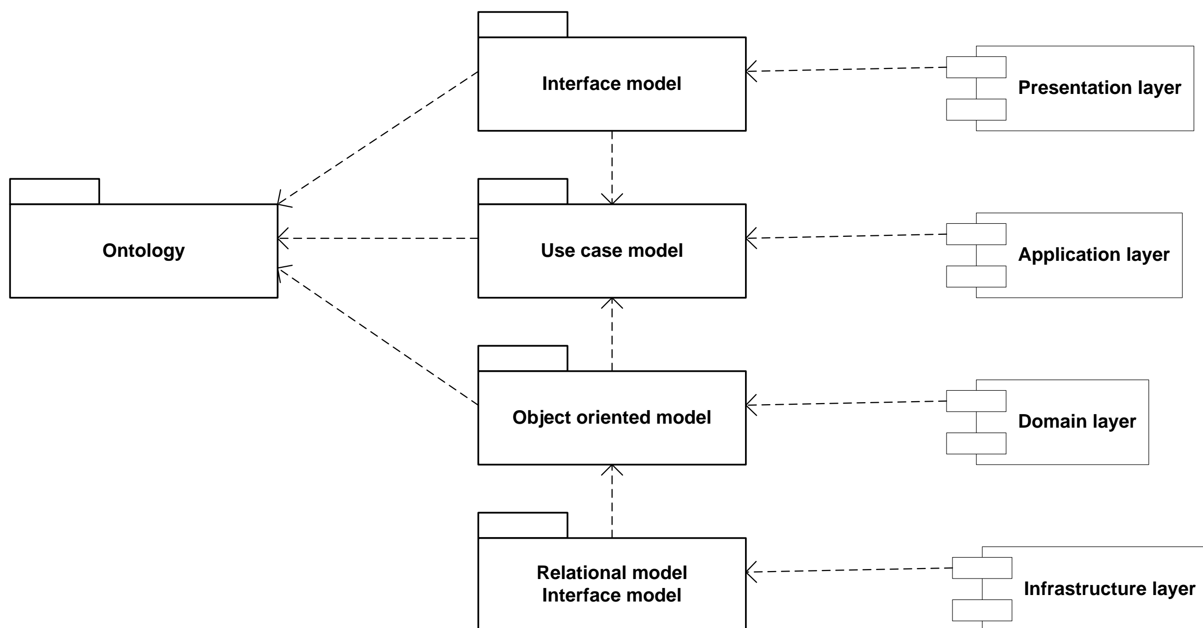


Figure 15: Dependencies between the ontology and the layers.

An interface model presents a possible view upon the system. Each view does not represent the system as a whole, but rather a part of the system. Which part of the system is represented and how large the part is, is determined by the task at hand. If for example the actor wants to update an account, the interface for updating an account is limited in scope to the information contained within an account. The behaviour provided by the interface is described in the use case model. The information supported by the interface is defined using a subset of the ontology. A user of the interface, who is not familiar with the whole system, only sees a part of the system and might get confused about the definitions. The ontology may be consulted for an overview of the terms defined in the interface and their overall relations. Furthermore, as the system evolves, the interface might be based upon a previous version of the ontology. In this case the interface exists for backwards compatibility reasons. This allows for the situation where existing users keep using the old interface, while new users use the new interface. In this case the interface model contains a mapping between the old ontology and the new ontology.

The object oriented model for the domain layer is the heart of the software (Evans, 2004). This layer is the most important layer of the system. It consists of the ontology and the behaviour of the system expressed in software code. The ontology terms and their relations are transformed into code objects. This transformation should be as straightforward as possible: the names of the code object are the same as the names of the terms in the ontology. And the relations between the code objects are the same of the relations between the terms of the objects. This way of working keeps the ontology and the object oriented model aligned. It makes the communication between the business level and the information technology level easier: what is possible in the ontology is also possible in the code and what's impossible in the code is also impossible in the ontology. Furthermore the

behaviour of the system, as expressed in the use cases, is also transformed in code objects. This transformation is slightly less straightforward than the transformation of the ontology into the object oriented model. However if the code objects follow the specifications in the scenarios, an alignment between the use case model and the object oriented model can be realized.

The infrastructure layer is defined by two models: the relation model and the interface model. The relation model specifies the storage of the information in a relational database. This model defines all the information that is stored locally to the service. The interface model of the infrastructure layer states the services upon which the service depends. It defines all the information that is external to the service. The relational model is derived from the object oriented model. Rules governing this derivation belong solely to the information technology level. They should not affect the object oriented model. Changes in the ontology result in changes in the object oriented model. These changes in the object oriented model result in changes in the relational model. The primary responsibility of services of the task layer (Figure 12) is to support business processes. They provide this functionality by (re)using other services, mostly of the entity layer. The communication with the other services is defined in the interface models of those services. Therefore the interface model is of importance. The primary responsibility of the services of the entity layer is to manage the life cycle of entities. Those services use a relational database management system to store the entities. Therefore for these services the relation model will be of more importance, as the service portfolio of an organization increases in size. That is, as more services become available for reuse. Less information might be stored locally to a service and more might be reused from existing or new services. It is important that the effects of this change are restricted to the infrastructure layer. The other layers of the service should not be affected by this change.

4.5 Multiple ontologies: versions and mappings

An actor's conceptualization of a domain may change over time. A change in an actor's conceptualization results in a change in the ontology. In order to track the changes to an ontology, each ontology is assigned a version number. The number of versions within the system is an indicator for the dynamics of the domain. If there are many versions per ontology, the number of changes is high and therefore the domain is more dynamic. The system should be able to support multiple versions of an ontology of an actor, because the other actors might depend on a specific version of an ontology. When the new version becomes available not all actors will be willing or able to switch to the new version immediately. Therefore the new version and the old version should be able to coexist. The ontology-driven software architecture offers several ways to support the multiple versions of an ontology. If the changes between the old version and the new version are small, the new version may be realized within the existing services. When the change occurs at the syntactical level (Figure 8, section 4.2), the change can be isolated to the interface layer of a service. If the change occurs at the semantic level, all four layers of the service might be affected. If the behaviour described in the use case model changes, the change affects the application and probably the presentation level. And when the changes between the old version and the new version are large, new services are realized. These new services may co-exist with the old services.

In a smart business network multiple conceptualizations can exist at the same time. The different actors may have a different perception the domain. Therefore multiple ontologies can be constructed. Each ontology describes the view upon the domain by an actor or a group of actors. To enable the different actors to communicate, the ontologies need to be translated. Translating an

ontology into another ontology is called mapping ontologies (Antoniou and van Harmelen 2004). Mapping the ontologies requires effort at the information technology level. Reducing the number of mappings within a system is important to reduce the amount of effort needed. Ideally there are no mappings: all actors involved share the same ontology. In this situation there is consensus between the actors about a dominant conceptualization. The ontology of the dominant conceptualization is the de facto standard for the domain. The number of mappings within the system is an indicator for the amount of consensus amongst the actors. If there are many mappings, there are many ontologies and thus the consensus is low. If there are few mappings, there are few ontologies and therefore the consensus is high. Within the system multiple ontologies and the mappings between them can exist at one point in time. The ontology-driven software architecture offers several ways to support the mappings between the ontology. If the mapping between the two ontologies is small, the mapping may be realized within an existing service. When the change occurs at the syntactical level (Figure 8 , section 4.2), the change can be isolated to the infrastructure layer of the calling service or the interface layer of the called services. If the change occurs at the semantic level, the infrastructure layer and probably the domain layer of the calling service are affected. And when the mapping between the two ontologies is large, a special mapping service is realized. This service is responsible for the two way translation of the ontologies.

4.6 Comparison to an existing architecture

The concrete software architecture discussed in this chapter is compared to the RUP 4+1 architecture. The RUP 4+1 architecture is a software architecture that is well-known and documented (Kruchten, 1995; Hofmeister et al., 2006). It is introduced by Kruchten to include the concerns of the stakeholders of a software system into the architecture and design. The architecture identifies four distinct views: The logical view, the process view, the development view and the deployment view. Each view addresses the concerns of a specific group of stakeholders. The stakeholders are grouped by the role they play during the software development process: end-users, programmers, system integrators and system engineers. The fifth view is a generic view: the scenario view. This view addresses concerns that are common to all stakeholders.

There are similarities between the RUP 4+1 architecture and the ontology-driven software architecture. The logical view of the RUP 4+1 architecture addresses the concerns of the end-users. It primarily deals with the object-oriented decomposition. The ontology-driven architecture also uses an object oriented model. It is used to define the domain layer of the services. This model is primarily defined by the ontology. The process view of the RUP 4+1 architecture addresses the concerns of the system integrators. It primarily deals with the process decomposition. The ontology-driven architecture does not explicitly offer a method for process decomposition. However, it can be used together with business architecture as the methodology for process decomposition. The architecture uses task and entity services to realize the information technology tasks, functions and objects in code. The development view of the RUP 4+1 architecture addresses the concerns of the programmers. It primarily deals with the decomposition of the system into components. The ontology-driven architecture uses services as the components of the software system. These services are further decomposed into individual layers. The physical view of the RUP 4+1 architecture addresses the concerns of the system engineers. It primarily deals with the deployment of the software on the available hardware. This view is not explicitly available in the ontology-driven architecture. The view is of lesser importance because of the use of virtual environments and machines (instrument nine). The scenario view of the RUP 4+1 architecture show how the other for

views work together. The scenario view defines a use cases and describes the interactions between objects and between processes (Kruchten, 1995). This is similar to the use of the use case model by the ontology-driven software architecture.

The main difference between the ontology-driven software architecture and the RUP 4+1 is the role of the end-users. Kruchten (1995) introduced the use of views in software architecture to address the notion that architecture does often not address the concerns of all the stakeholders. In RUP 4+1 the logical view addresses the concerns of the end-user. In the ontology-driven software architecture there are multiple and different end-users, therefore there may be multiple ontologies and thus multiple logical views. Compared to the RUP 4+1 the ontology-driven software architecture focuses on the role of the end-user and their views upon the business domain.

4.7 Conclusion

The architecture addresses the problems of multi-actor complexity and technical complexity. The multi-actor complexity is addressed by defining a conceptualization. This conceptualization is defined as a demarcation of the business domain by an actor. It represents the view of the actor on the domain. The conceptualization is formally defined using an ontology.

The technical complexity is addressed by applying partitioning. The functionality of system is partitioned over different services. The services are divided over three layers: The task service layer, the entity service layer and utility service layer. The task service layer contains services that directly support the process of the business. The entity service layer contains services that are responsible for the retrieval and storage of entities. The utility service layer contains services that support the services in the task or entity service layer. The services in the task service layer are the services that are responsible for providing the tasks that make up the business process. The task services implement the information technology functions and use the information technology objects. The services in the entity service layer are the services that are responsible for providing the information technology objects.

Each service is internally also divided in layers. Each layer has its own responsibility and functionality. The four layered architecture of Evans (2004) consists of: the presentation layer, the application layer, the domain layer and the infrastructure layer. The presentation layer may contain multiple presentations of the same information. The application layer is responsible for the coordination of tasks. The domain layer represents the concepts defined by the ontology. The infrastructure layer provides technical services to the other layers.

The ontology is linked to the layers by using models. Four models are identified: the interface model, the use case model, the object oriented model and the relational model. The ontology defines the vocabulary that can be used in the construction of the models. The behaviour of the system is described in the use case model. An interface model presents a view upon the system. The object oriented model for the domain layer consists of the ontology and the behaviour of the system expressed in software code. The relation model specifies the storage of the information in a relational database. This model derived from the object oriented model.

5 Towards a software engineering approach

5.1 Introduction

In the previous chapter the architecture is introduced and the required models are described. This chapter describes the specification and realization of the models. In other words: the architecture is applied to the software system the case study. During this phase some guidelines are discovered that can be applied complementary to the architecture. Applying the architecture to the case study reveals:

- whether the architecture is successfully applied
- the guidelines that complement the architecture

This chapter starts with a description of the gathering of the functional requirements for the software system of the case study. These functional requirements are described using the ontology and the use case model. The ontology and the use case model define *what* the system should do. The use case model is one of the four models that connect the ontology to the software system (Figure 15). The other three models are: the interface model, the object oriented model and the relational model. These models describe *how* the system should work. Therefore they are not part of the specification of the system; they are part of the realization of the system.

5.2 Specification: gathering functional requirements

The functional requirements are gathered using a process consisting of three steps. First, the domain is demarcated by describing the part of the business processes that needs to be supported by the software system. Second, the ontology for the conceptualization is constructed. The ontology defines the vocabulary for the business processes and the software system. And last, the use case model is written. It describes the behaviour of the system within the business processes. The description of the business processes and the use case model should use the vocabulary defined by the ontology. However, there is an inconsistency: the ontology is defined in step two, only after the business processes are described. Using an iterative process solves this problem. In the early iterations the important terms and relations are defined. In subsequent iterations the vocabulary is refined, until all the documents describing the functional requirements are consistent.

A conceptualization belongs to an actor and a represents the actor's view upon the domain. The conceptualization belongs to the IBLC which operates in the domain of human resource management. The scope of the software system is demarcated by choosing one business process of the IBLC: the process to search and book a course. This process is well known and accepted within the business network: there is consensus amongst a variety of actors and the process is proven at the business level and at the information technology level. Over the past years this process is already correctly decomposed into business functions and business objects (chapter two). Therefore business process reengineering or further decomposition is not needed. This allows for the focus to be on the uncertainty described in chapter one: the knowledge gap with respect to the realization of the software. The business process is described using the structured analysis and design technique (SADT) (Ross, 1985). This technique is a top-down functional decomposition of the process into sub processes. The process is documented graphically using process diagrams. Appendix A contains the

process diagrams for the software system. The diagrams are constructed using historical data and the existing knowledge about the well known process.

The ontology is described using the web ontology language (OWL) as defined by the World Wide Web Consortium (W3C). This standard is widely used and it is supported by software for the creation of ontologies (Gašević et al., 2009). The standard consists of three sublanguages (Antoniou and van Harmelen 2004). For the construction of the ontology the web ontology language for description logic (OWL-DL) is used. This is a pragmatic choose: the standard is not too complex and supports the use of reasoning. The reasoning feature allows for automatically inference of the taxonomy. The reasoning feature can also be used during development to test the consistency and internal validity of the ontology. The ontology is constructed using a huge set of excel and word files containing the vocabulary and definitions written by Paul Bessems (CEO of IBLC). Appendix B contains an example of the ontology. To create support for the vocabulary within the community, the IBLC has a created its own wiki[9]. This wiki is comparable to Wikipedia and allows for collaboration amongst the actors in the network.

The vocabulary defined in the ontology is used in the use case model to describe the behaviour of the system. Appendix C contains an example of a use case specification. The use cases are described focusing only on the “happy flow” (Beckers et al., 2007). The happy flow of a model is a deliberate simplification of the model. The desired behaviour is modelled; all disturbances or possible error situations are left out. What remains is the basic flow of actions: the steps needed if everything is as expected. Happy flows are useful to express the overall intent, as the model does not get cluttered with details.

5.3 Realization: constructing the software system

The first step in the realization of the system is to identify the services needed. Starting point for the identification is the business process that needs to be supported. The business process diagrams are available in appendix A. The software system is demarcated to the “search and book course” process, which is process A1. Examining the information needed for the process reveals the entity services needed. The entity services are responsible for the retrieval and storage of entities. The information items needed by the process are: course, trainee, person, booking and manager. The relations between those items can be found in the ontology (appendix B). The three entity services to start with are: the contact service, the course service and the booking service. Examining the sub processes of process A1 reveals the task services. The task services directly support the process of the business. Only one task service is identified: the sub process “book course”. The other two sub processes “Search and view course information” and “authorize booking” are data retrieval and data update actions. They are therefore implemented in the entity services.

For each entity service an object oriented model is defined. This object oriented model is derived from the ontology. The transformation of the ontology into an object oriented model is a manual process. Ontologies are closely related to object-oriented models (Siricharoen, 2007). The main difference between an ontology and the object oriented model is the fact that ontologies allow for inference. An automatic reasoner can infer the types of classes an object belongs to. In an object oriented model the type of an object determines which class the object belongs to. Furthermore an ontology allows for multiple inheritance. This is not supported by some object oriented languages.

[9] The wiki is publicly available at http://www.wikilexis.nl.

Therefore the class inheritance structure of the object oriented model is constructed similar to the *specified* taxonomy of the ontology. This taxonomy is the hierarchical structure of classes as *specified* during construction of the ontology (Appendix D). On the other hand, the *inferred* taxonomy is the structure of classes as constructed by the automatic reasoner. The inferred structure may contain multiple inheritance. In the object oriented model, the multiple inheritance is simulated using interfaces. Another important difference between the ontology and the object oriented model is the use of properties. In an ontology the properties may be part of an inheritance structure. Properties may inherit from other properties. In an object oriented model the properties are part of the class definition. A class may inherit properties, but properties may not inherit from each other. Therefore in the object oriented model for the services each property is defined independent from the other properties.

The interface model is derived from the ontology and the use case model (Appendix E). The ontology specifies the vocabulary to use in the interface definitions. The use case model specifies the behavior supported by the interface definition. The combination of the vocabulary and the behavior of the interface provide the contract for the service. Both the interface model and the object oriented model are derived from the combination of the ontology and the use case model. However, the object oriented model consists of a complete representation of the ontology and all the business rules of the use case model. Whereas, the interface model is a subset of the ontology and it contains no business rules at all. Each interface definition is a view upon the system, for a specific purpose.

The relational model is used to define the way the objects of the object oriented model are stored in a relational database management system. It is therefore not directly derived from the ontology, but from the object oriented model. Whenever the object oriented model changes: the relation model also changes. To store objects in a relational database three things are needed. First, a relational structure is defined to store the data. Second, mappings are defined to transform the data of the object oriented structure into a relational structure. Third, queries to create, read, update and delete the data in the relational database are needed. The relational structure is defined using the well known normalization rules (Codd, 1970; Date, 1975). The model can be found in appendix F. The software system uses an existing object-relational mapper. The mapping between the object oriented structure and the relation structure is specified using the specifications of this mapper. Examples of the mapping files are presented in appendix G. The mappings are constructed using the guidelines and examples of Kuaté et al. (2009). The main advantage of the use of an object-relational mapper is the fact that the mapper is responsible for the generation of the operations for the relational database: create, read, update and delete queries. Therefore these queries are not specified, they are created automatically.

5.4 Technical characteristics

This chapter describes the technical characteristics of the software system. The most important software engineering decisions are described. The text marked in *italic* contains the guidelines that are applicable for software engineering within a ontology-driven software architecture.

Businesses have the choice between two main virtual platforms for building services for a service oriented architecture: the Java Enterprise Edition platform and the Microsoft .Net platform. The Internet Business Learning Community uses the Microsoft .Net platform for its systems. Therefore the platform of choice for the software system is the Microsoft .Net platform.

The vocabulary of the ontology is used in the source code, at all levels. The xml documents that describe the interface definitions use the vocabulary for the names of the elements and attributes. The source code files use the vocabulary for the names of the classes and properties. The database files use the vocabulary for the names of the tables and the columns. The consistency of the naming allows for easy traceability, while keeping the concepts technically separated. Combining two ontologies in one service might result in duplicated naming. For example: if both actors of an ontology use the term "course" and both define it differently, then the term course would be used in the source code for two classes. The compiler will not allow for that. Namespaces are used to resolve this naming conflict: the ontologies (Figure 11) are each assigned its own namespace. This allows for the same concept from the ontology to be used multiple times, each in its own namespace. These *namespaces are used to uniquify the names* in the xml documents, the source code files and the database files[10].

The services of the task service layer use the services of the entity service layer to support a range of products. The software system contains one task service: AuthorizeBooking. This service is implemented *using a web service business process execution language engine* (WS-BPEL). The WS-BPEL is a standard for realizing interactions with services. The specification uses a higher level of abstraction than the object oriented language used for the entity services. The WS-BPEL engine used for the software system is the Microsoft BizTalk server.

The interface layer is realized using two interoperability standards. The WS-I Basic Profile 1.1[11] (WS-I, 2006) is provided for basic operations and supported by a large constellation of vendors. The *WS-* suite of standards* (Cabrera and Kurt, 2005) is provided for more advanced situations. These standards are not as widely adopted as the *WS-I Basic Profile standard*. However they support more advanced features like: reliable messaging, security and transactions (Cabera and Kurt, 2005). By supporting both standards no trade-off between interoperability and functionality is needed. In chapter seven the effects of a technology change in the interface layer are described. It describes another way interacting with services using the representational state transfer (Fielding, 2000). This type of data transfer is more a principle then a standard. However providing a third way of interacting with the services will increase the interoperability of the system.

The application layer of a service is responsible for the data transformations between this service and the services that depend upon it. An alternative would be to make the interface layer responsible for these data transformations. However, if the emerging of a new technology for data exchange requires the addition of a new interface layer, then the new interface layer is required to perform the same data transformations. In this situation the same data transformations would exist twice in the system: once in the old layer and one in the new layer. Therefore: *making the application layer responsible for these data transformations* enables the swift replacement or addition of the technology used in the interface layer, without the penalty of duplicated code. The infrastructure layer of a service is responsible for the data transformations between this service and the services upon which it depends and for the data transformations between this service and the relational database management system.

[10] The relational database management system does not support the use of namespaces. However the use of namespaces may be simulated by using multiple users in the same database. Each user corresponds to a specific namespace.
[11] At this point in time version 1.1 is the latest finalized WS-I Basic Profile version.

The domain layer is the heart of a service: it contains the ontology expressed in source code. The domain layer depends on no other layers (Figure 16). In many software architectures each layer has a dependency on one or more of the lower layers. However *reversing the dependency between the domain and the infrastructure layer* makes the domain layer independent of the infrastructure layer. This inversion of dependency technique (Martin, 2006) enables a swift switch in technology used for the storage of entities. A service may store entities locally or depend upon another service for the storage. As more services become available in a system, less information is stored locally. In such a situation the infrastructure layer changes. Because of the inversion of the dependency between the domain and infrastructure layer, no changes are needed in the domain layer. The presentation layer determines when to start a transaction and when to commit or rollback that transaction. The layer is therefore allowed to have a dependency upon the infrastructure layer (Figure 16).

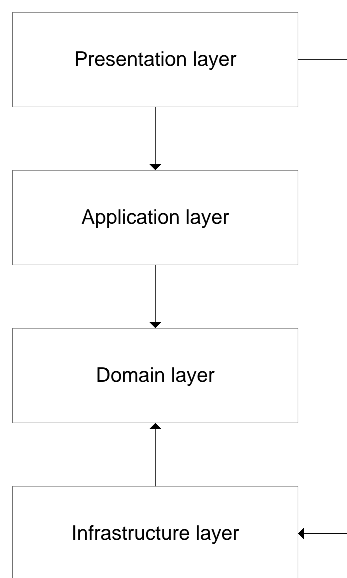


Figure 16: Dependencies between the layers.

The software system is realised using various design patterns. To illustrate the use of patterns, a pattern will be briefly described for each layer (Figure 16). The presentation layer uses the open session in view pattern (Kuaté, et al. 2009). When the presentation layer receives a request, it opens a session with the database. When the request is processed by the service, the presentation layer closes the session. The application layer uses the façade pattern (Gamma, et al., 1995). The façade pattern provides a unified way to access the functionality of the application layer and domain layer. The unified interface keeps the code on the presentation layer clear because one interface is used, instead many different interfaces from various sources. The domain layer uses the abstract factory pattern (Gamma, et al., 1995). The abstract factory pattern provides a way to create related objects. The infrastructure layer uses the lazy load pattern (Fowler, 2002). When the domain layer uses an entity, only the information need is loaded from the database. It is an alternative to loading the complete object graph into memory on first use. This is expensive with respect to the resources used. Another technique would be to explicitly state what needs to be loaded. However, this is expensive with respect to development time.

Design for change is realized using the architecture, the layering and the dependency injection. Also the artefacts of the software system are arranged in a clear structure. The services and the individual

layers are easily recognizable in this structure. Furthermore the five S.O.L.I.D. principles as defined by Martin (2006) are used. The single responsibility principle (S) states that a class should have one reason to change. The open closed principle (O) states that the class should be extendable without modifying it. The Liskov substitution principle (L) states that derived classes must be substitutable for their base classes. The interface segregation principle (I) states that interfaces should be fine grained and client specific. The dependency inversion principle (D) states that classes should depend on abstractions and not on concretions.

The software system is designed using the Microsoft .Net platform. Therefore, *the standards used by the designers of the platform* are the main coding standards used for the software system. They are published in the book by Cwalina and Abrams (2005). An important aspect of these coding standards is that they may be enforced and analyzed using an automated tool[12]. This makes it easier to audit and control the quality of the code. Most of these coding standards are technology dependent: they are specific for the .Net framework. A good source for generic coding standards is a book by Martin (2009). This book presents the rules for object oriented programming and explains the rationale behind them. However these rules are not automatically enforced. But modern integrated development environments enable tools to assist the development of clean and consistent code[13]. These tools analyze the code in real time and provide feedback and suggest changes. Some of the standards provided by Martin can be checked and reported by these tools.

The software is testable and some unit tests and integration tests are provided. However these tests do not adequately cover the source code of the software. They are provided for reference usage only. The existing test setup and structure can easily be used to extend the tests to a full coverage of the source code. The application tests are used to test across the layer boundaries. The integration tests are used to test across the service boundary.

All the services of the software system are configurable. All the resource upon which they depend can be configured using configuration files. The configurability and the architecture allow for a range of deployment and scalability options.

5.5 Conclusion

This chapter describes the models for the specification and realization of the system: the use case model, the interface model, the object oriented model and the relational model. The models are realized using the architecture described in the previous and the guidelines for the realization described in this chapter (Table 7).

The functional requirements for the software system are specified using the ontology and the use case model. The interface model, the object oriented model and the relational model are not part of the functional specification; they are part of the realization of the software. The functional requirements are gathered using an iterative process consisting of three steps. First, the conceptualization is demarcated by describing the relevant business processes. Next, the ontology for the conceptualization is constructed. Last, the use case model is written.

---

[12] The tool FxCop (www.microsoft.com) analyzes the compiled code for violations of the code standards and reports them.

[13] Visual Studio may be expanded by the tools: ReSharper (www.jetbrains.com) or CodeRush (www.devex.com).

The Microsoft .Net platform is chosen for the platform for building the services of the system, because it is the platform of choice of the Internet Business Learning Community. However most of the architecture, the techniques, the guidelines and the technical characteristics are not restricted to this platform. The services of the system are identified using the business process specification. The software system consists of tree entity services and one task service. The entity services are realized in Microsoft .Net. The task service is realized in Microsoft .Net, using BizTalk server.

The object oriented model is derived from the ontology. This transformation from ontology to object oriented model is a manual process. The defined taxonomy of the ontology is used as the class structure in the object oriented model. The inferred structure is simulated using interfaces. In chapter eight an alternative method for the support of inferred class is described. The interface model is also derived from the ontology. The object oriented model consists of a complete representation of the ontology and all the business rules. The interface model is a subset of the ontology. The relational model is derived from the object oriented model. It defines the way the objects of the object oriented model are stored in a relational database management system. The relational structure is defined using normalization rules. The mapping between the object oriented structure and the relation structure is specified using the specifications of an existing mapper.

Guidelines complementary to the ontology-driven software architecture.
1. use the vocabulary of the ontology in the code
2. use namespace to uniquify names
3. realize services from the task service layer using a WS-BPEL engine
4. use the WS-I Basic Profile standard for operability use the WS-* suite of standards for advanced situations
5. make the application layer responsible for the data transformations
6. reverse the dependency between the domain and the infrastructure layer
7. design for change (S.O.L.I.D. principles)
8. use the same standards as the designers of the development environment
9. make the services configurable
10. use the vocabulary of the ontology in the code

Table 7: Overview of the guidelines.

6 Testing the flexibility of the software system

6.1 Introduction

In this chapter the software system is evaluated against the forces of changes (Figure 17). Two scenarios using the business drivers for change are discussed in section two. These drivers may be the result of the introduction of a new actor or of new functionality. Two scenarios using the information technology drivers for change are discussed in section three. These drivers may be the result of emerging new technologies. The scenarios are chosen in corporation with the IBLC to represent realistic changes. Third, the software system is evaluated using modern software analysis techniques.

6.2 Business drivers for change

The business drivers for change describe the required changes to the software system due to changes at the business level. The evaluation of these changes shows the traceability of changes from the business level to the information technology level. At the information technology level, these changes can affect the service layers of the architecture and they can affect the layers within a particular service (Figure 17). The changes affect the service layers when new services are introduced or existing services are removed. The changes affect the layers within a service when services are modified. The smallest change possible at the business level is a change in the properties of one of the classes in an ontology. This change is discussed in scenario one. The largest change at business level is the replacement of the ontology by a new ontology. This might happen when the old ontology no longer represents the actor's new conceptualization of the business domain. In that case the change that existing service are reusable is probably small. Scenario two describes a scenario in between the smallest and largest possible change.

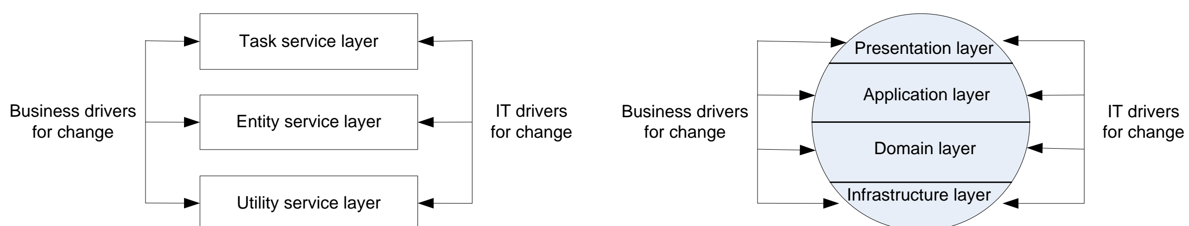


Figure 17: Drivers for change in the service and software layers.

The business drivers for change may be caused by an actor's change in view of the domain. If an actor changes its view of the domain, the conceptualization changes (Figure 11). The business drivers for change may also be caused by an actor's change in scope of the system. The conceptualization demarcates the domain in scope (chapter five, section two). If an actor changes the scope of the conceptualization, the system changes in size. Therefore, the business drivers for change cause changes in the conceptualization and subsequently in the ontology (Figure 11). The ontology and the use case model together specify the functionality of the system. By applying the changes to the ontology and use case, before changing the models or the source code, the systems remains clearly defined.

In this section the software system is evaluated against changes in the ontology and the use case model. In the ideal situation the effects of these changes can be restricted to a single layer, without affecting any other layer. The effects of the changes will be described using the smallest unit of change. Henry and Wake (1988) use a single line of code to track the changes to a software system. The single line of code is useable as the smallest unit of change in their study. However, the single line of code is too fine grained for communicational purposes. Clear communication between the business level and the information technology level is important for a good business and information technology alignment and therefore for business flexibility (chapter three). The smallest unit of change needs to be small enough to describe the effects of the change without unwanted side effects. A side effect due to a change is a condition that leads the system to an erroneous state or a state that violates the original semantics (Kagdi and Maletic, 2006). The smallest unit of change also needs to be large enough to have clear semantic meaning at the business level and the information technology level. In the web ontology language (OWL) the smallest units are the classes, the object properties and the data properties[14]. The smallest unit of change in the ontology is one single change to a class, an object property or a data property. The addition, modification or removal of a class affects the defined and inferred taxonomy. The addition, modification or removal of a property affects the class and possibly the inferred taxonomy. If a class is added and properties are added to the class, then this can be described a series of smallest units of change.

Scenario one concerns a business-driven change: the addition of a data property to an existing class. The software system contains information about courses. An existing actor in the network wants to record the duration of a course. The duration is expressed in days and provides information about the time it takes to take the course. This change at the business level corresponds to a single change at the information technology level: in the ontology a new data property "courseDuration" is defined and added to the course class. No change is needed in the use case models because the data property does not affect any behaviour. A change in the ontology might affect the interface model and the object oriented model (Figure 15). In this case both the interface model and the object relation model are changed: the course class is enhanced with a new property: "duration"[15]. Therefore both the interface layer and the domain layer are changed. The application layer is responsible for the transformation of the information between the domain layer and the interface layer. This layer is also changed. The information in the new property of the course class in the domain layer needs to be stored. The relation model is changed: a column named "duration" is added to the course table. This business-driven change of an addition of a data property is a very small change. Never the less, it affects all layers of the existing product service[16]. However the changes in each layer are very small and transparently traceable.

Scenario two concerns a business-driven change: the addition of a new product to the system. In the software system, a person can search for courses. These courses are instructor-led classroom trainings. A customer wants to offer computer based training (CBT) to its employees. The characteristics and the semantics of the computer based trainings differ from the instructor-led

[14] The instance or individual is another unit of change. In the specification of the ontology, no individuals are used.

[15] The property is named duration and not courseDuration because it always is referenced through the course class. In the OWL ontology the properties can belong to many different classes, hence the course prefix. In the C# language a property always belongs to a single class.

[16] The unit test should be modified prior to applying the changes to the layers (Beck, 2003).

courses. Therefore a new service is added to the entity service layer. This entity service is responsible for the life cycle of the computer base training products. This change at the business level corresponds to a multitude of changes at the information technology level. However each of these changes can be described using a series of smallest units of change. First, a class is added to the taxonomy of the ontology. Second, the appropriate data properties are defined and added to the new class. These are properties like: cbtMaterial[17], cbtDuration[18] or cbtLevel[19]. Third, the required object properties are defined and added to the new class. These are properties like: mayBeProceededByProduct or isSimilarToProduct. The entity service for the computer based training products is a new service. Therefore a new use case model is specified. However, his use case model is similar to the use case model of the entity service for the course bookings. The use case model of the existing course entity service can be used as an example. This is also possible for the interface model, the object oriented model and the relational model.

6.3 Information technology drivers for change

The information technology drivers for change describe the changes to the software system due to changes at the information technology level (Figure 17). These changes may be the result of emerging new technologies. The main challenge is to apply the new technology successfully, without causing any change in the existing functionality. In the ideal situation the effects of these changes can be restricted to a single layer, without affecting any of the other layers. To evaluate the software system against information technology drivers for change, two technology changes are used.

Scenario three concerns an information technology driver for change: the growing popularity of an existing interoperability technology. The new interoperability technology is an extension to the existing technologies. The change is not a replacement but rather an extension of existing technology. In this case a new actor joins the business network and the actor does not want to commit to the WS-I Basic Profile or the WS-* suite of standards (chapter six). The actor wants to use the representational state transfer (REST) technique (Fielding, 2000). For this change the existing ontology and use case model are reused. No change is needed. The presentation layer is changed to support the representational state transfer technique. No changes are needed for the application layer or the object oriented layer. This change is local to the presentation layer[20].

Scenario four concerns an information technology driver for change: the use of a different relational database system. In this case the changes are restricted to the infrastructure layer. The object relational mapper hides many of the vendor specific capabilities of relational database systems and many of the differences in the query language dialects. If the new relational database system is supported by the object relation mapper, the change to a different database system has a low impact on the system. The object relational mapper[21] used can switch between different relational database systems by changing the configuration of the infrastructure layer.

---

[17] Examples of values for the property cbtMaterial are: CD-Rom, DVD, DVD-Rom or online.

[18] Examples of values for the property cbtDuration are: four hours, eight hours, sixteen hours or forty hours.

[19] Examples of values for the property cbtLevel are: beginner, intermediate or advanced.

[20] The presentation layer of the contacts service is modified to support the representational state transfer technique. The other service may be modified in the same manner.

[21] The object relational mapper is NHibernate. It supports the relational databases management systems of the main vendors, like: Microsoft SQL Server, Oracle database server, IBM DB2, MySql, Firebird and PostgreSQL.

6.4 Technical evaluation

The technical evaluation of the software system addresses the use of coding standards, the dependencies between the services and the layers, the interoperability standards and the deployment options.

The main coding standards used during the realization are the standards used by the designers of the .Net platform. These standards are evaluated using a static code analysis tool. The tool defines rules for each standard and examines the compiled code. For each contravention of a rule a warning or an error[22] is given. The software passes all rules, except for two[23]. The first rule is CA1020: Consider merging the types with another namespace. The coding standards state that each namespace must contain enough types to justify the existence of the namespace. This rule is broken because the prototype has limited functionality. Therefore each namespace only contains a few types. This rule can safely be ignored, because in a real situation more functionality is specified and more types are realized in the namespaces. The second rule is CA1024: Change a method to a property if appropriate. This rule is breached because the repository pattern uses methods that have a name, start with "get" and take no parameters. The repository uses these names for the methods to keep the semantics within a repository consistent. Therefore this rule can safely be ignored for the repository interfaces and classes.

The services need to be loosely coupled to promote reuse: the system should be designed with minimal dependencies between the services. The dependencies are minimized by the layered architecture that prescribes the use of contracts in the presentation layer. Each service only depends upon the contract of other services, not on the way the other services are realized. The dependencies between the layers within a service can be measured using design quality metrics. Two metrics are of importance: abstractness and instable[24] (Martin, 1994). The abstractness is defined as the ratio of the number of abstract types to the number of types in a layer. An abstractness value of zero indicates a completely concrete layer, where an abstractness value of one indicates a completely abstract layer. The instability is defined as the ratio of efferent coupling (defined further on) to total coupling. It measures the layers easiness to change. An instability value of zero indicates a layer that is difficult to modify, where one indicates a layer that is easier to modify. The instability (I) of a layer is calculated as follows:

$$I = \frac{Ce}{Ce + Ca}$$

The efferent coupling (Ce) is the number of types within this layer that depend on types outside this layer. The afferent coupling (Ca) is the number of types outside this package that depend on types within this package. Layers that have an acceptable balance between abstractness and instability are maintainable. This balance is the dotted line shown in the figure below. This figure shows the result of the dependency analysis of the Bookings service. All the layers of the service have an acceptable deviation of the dotted line. They are located within the green area[25]. This means that there are no

[22]The static code analyzer is configurable: deviations from the rules can be ignored, treated in a warning or treated as an error.

[23] The projects containing the test code are excluded from the static code analysis.

[24] In English the term instable has a negative connotation. As a metric the term only refers to the resilience to change. An instable package is easier to change than a stable package.

[25] Monochrome: The green area is the light area near the dotted line.

unmanageable dependencies. Analysis of the other services yields similar results. Although instability has a negative connotation, it is actually a good characteristic if it is co-occurred with a low level of abstractness. Martin (1995) state that the two most desirable layers are either: abstract and stable or concrete and instable.

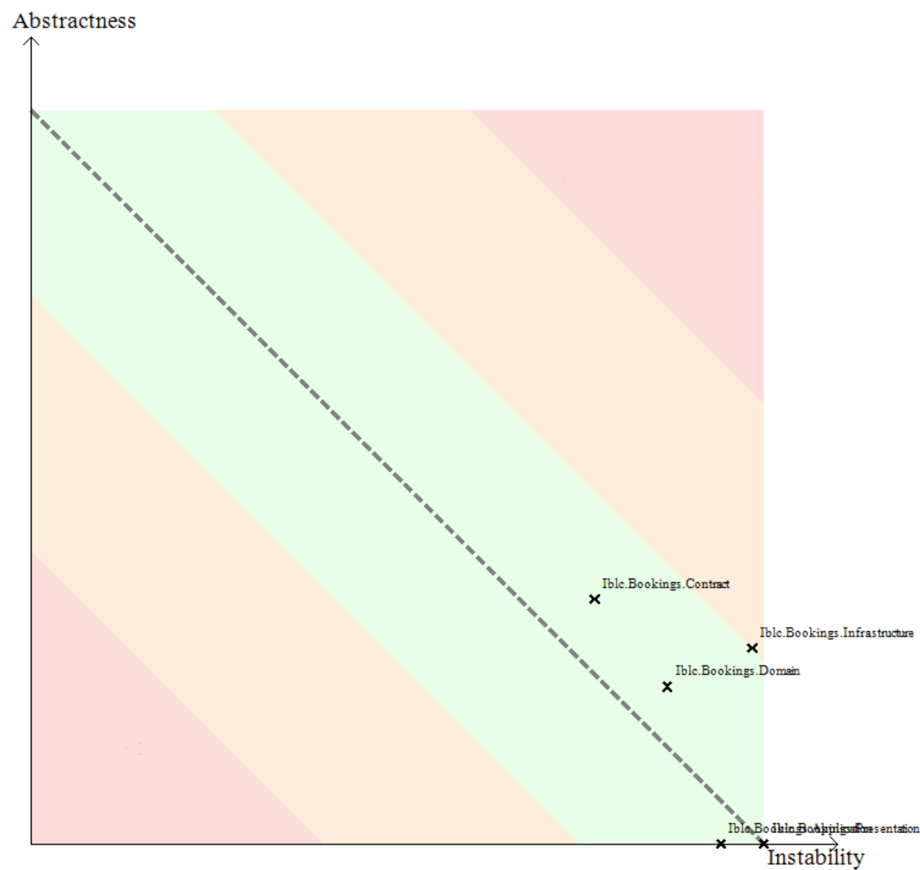


Figure 18: Dependency analysis of the layers of the Bookings service.[26]

The services of the proof concept are testable using unit tests, application tests and integration tests (chapter six, section four). All the mentioned tests are constructed within the .Net framework. They do not test the interoperability of the software. To test the interoperability of the services a java based test application[27] is used. The evaluation is performed manually: the request messages are sent to the individual services and the response messages are compared to the expected messages. In all cases the expected response message is received. The services are successfully evaluated against the WS-I Basic Profile.

The services of the software system are independently deployable. All the resources upon which a service depends are configurable. These two characteristics provide a range of deployment options. Scalability is provided by the deployment options. For example: in the smallest deployment option all services and the databases are deployed on a single server. As the load on this server increases an extra server may be used: the database server is moved to new server. In a three server setting, the

[26] The analysis is performed using the NDepend tool, freely available for academic research at http://www.ndepend.com.

[27] The application is called soapUI and is publicly available at: http://www.soaui.org.

BizTalk server may be moved to a dedicated server. Furthermore, each service may share a server with other services or be deployed on its own server. And last, a server may be deployed on a cluster of two or more servers to increase the performance. No code changes are required.

6.5 Conclusion

The software system is evaluated using cases that require changes to the system. Two cases concern business drivers for change. Those changes are caused by the introduction of a new actor or of new functionality. Changes to the ontology are used as the smallest unit of change. The changes to the ontology are large enough to have clear semantic meaning at both the business level and the information technology level. These changes are traceable from the business level to the source code. The traceability is provided by the four models defined in chapter five. In the first evaluation case a small change in the ontology affect all the layers of the existing service. However the change in each layer is very small and transparently traceable. Due to the loosely coupled services, the change is restricted to the affected service only. The smallest unit of change may be used to estimate the cost of changes: if many lines of source code are needed to realize a change in an ontology component, then that change is expensive. The second evaluation case showed how a series of smallest unit of change may be used to define a large change.

Two other evaluation cases concern the information technology drivers for change. These changes are the result of emerging new technologies. The main challenge is to apply this new technology successfully, without causing any change in the existing functionality. In the case of the new interoperability technology the changes are neatly applied to the presentation layer only. No changes are applied to the other layers. The second case evaluated the change due to the use of a different database server. This change is handled in the infrastructure layer. Both evaluation cases result in the ideal situation: isolate change to a single layer.

The software system is evaluated using modern software analysis techniques. The coding standards are evaluated using a static code analysis tool. The software system passes all rules, except for two. Both these rules can be safely ignored. The dependencies between the layers of the services are analysed using software metrics. All the layers of the services have an acceptable deviation of the acceptable balance between abstractness and instability. They are maintainable. The vendor independent interoperability standard, the WS-I Basic Profile, is manually tested using a java based application. In all cases the request messages results in the expected result message. The scalability of the system is described using the deployment options of the services. The services may be deployed in various ways to provide the scalability needed. Configuration changes are needed, but no code changes.

7 Conclusions and recommendations

In this chapter the main conclusions are drawn about the research. These conclusions are based upon the work as a whole. The recommendations are twofold: the recommendations for further research are based upon the knowledge gaps encountered during the research and the recommendations for the Internet Business Learning Community (IBLC) are based upon the research and the conclusions.

7.1 Conclusions

Both parts of the first research question are answered bellow. Research questions two and three are answered in the conclusion of chapters two and three. The main result of the research provides answers to research questions four, five and six.

The first part of the first research question addresses the change of the software system from a single-actor system to a multi-actor system. The second part addresses the support of composite products, consisting of components from different suppliers. How are the two structural differences changes handled? The first change is the shift to a multi-actor system. Each actor has its own view on the domain and upon the system that supports their processes within that domain. To decrease the multi actor complexity it is important to try to minimize the number of views. At the business level: use policies and define processes to try to create consensus amongst the actors about the view on the domain. If consensus cannot be reached amongst all actors or on all subjects, there will remain multiple views. At information technology level: the architecture supports multiple views by explicitly defining the actor's view upon the domain for which the system is build. A view upon the domain is formalized in an ontology. The ontology is used to define the models of the system.

The architecture supports the use of composite products by separating the definition of a product from its use. This separation allows for a product to be combined into a composite product. The architecture defines an entity service layer. The services in this layer define the products or product components. The task service layer contains services that use the products. The services in the task service layer may freely combine products into a composite product. Although the architecture supports the definition of composite products, the software system does not support the joint responsibility of the suppliers. When a customer buys a composite product, the buy is governed by a contract and possibly delivering conditions. When a product is delivered by one supplier, the supplier's delivery conditions are applied. If the product is a composite product, consisting of products from multiple suppliers, there is no easy way to define which conditions to apply. The architecture allows designers to build rules that validate composite products. For example a rule might state that only one course instructor may be part of the composite product. However, neither the software system nor any of the standards used, support the automatic integration of the contracts and delivery conditions of the suppliers of the product. The IBLC mitigates this problem at the business level by using a brokering company.

The main result of the research is the definition and use of an architecture that places the ontology in a central position. The use of *"the language of the business"* as the starting point of the design of a software system is not a novel approach within the information technology. Novel is the explicit definition of this language for multiple actors and linking the ontology to different services and layers of the software system (Figure 19). An actor's view upon the business domain is formalized in an ontology. The architecture of the software system supports multiple versions of ontologies and the

mappings between them. An ontology may belong to one actor or a group of actors. Therefore the system supports the change from a single-actor system to a multi-actor system. The architecture of the software system supports the use of composite products, consisting of components from different suppliers. An important aspect of the architecture is the acknowledgement of the dynamics within a smart business network and the divergent views upon the business domain by the multiple actors of the network. The ontology based architecture supports multiple versions of ontologies and mappings between ontologies (chapter four). Furthermore, the changes between the versions and the changes handled within these mappings can be described in smallest units of change (chapter six). The effects of these smallest units of change can be traced through the models of the architecture into the changes to the services and layers of the software system. Using the same the names and definitions in the ontology and the code of software system, allows for the changes to be tracked back from the components to the ontology. This two-way traceability makes the services and layers easier to understand, reuse and maintain.

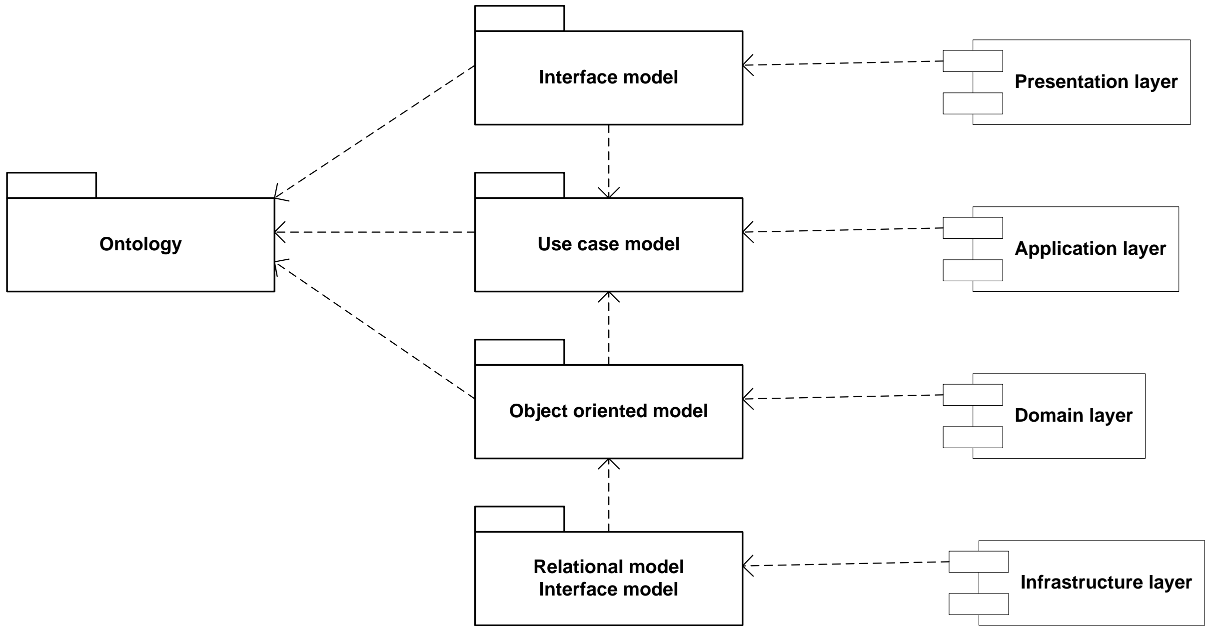


Figure 19: Dependencies between the ontology and the layers.

7.2 Relevance to other smart business networks

This section discusses the relevance of the theories, the instruments, the architecture and the guidelines to other smart business networks. The research of this thesis is performed using a single case study. More studies are needed to validate and improve the architecture of the software system. The case study is focused on flexibility; other important architectural aspects are not included. These aspects include quality indicators like: performance and security. In a real-life production environment these aspects may not be ignored. The multi-actor theories (chapter two), the software management and engineering theories (chapter three) are not specific to the case of the IBLC. They may be used by other smart business networks. The definition software flexibility as used within this research may be used by other smart businesses. The definition is constructed using software quality indicators that are interrelated and have overlap (chapter three, section two). There might not be consensus about their exact definitions or relevance in other smart business networks. Altering the definitions and relevance might affect the instruments (Table 6). These instruments are

used as a basis for the architecture and the software engineering approach. The use of an ontology is also relevant to other business networks. The functional requirements for the software system are of little use to other small business networks (chapter five). However, the way they are gathered and specified can easily be used by other networks, because existing and well-known modelling techniques are used. The software system is realized using the Microsoft .Net framework. Other smart business networks might use different technology stacks. Without a change in the theories, requirements and guidelines the software system could just as well be realized within a java environment. The guidelines (Table 7) are technology independent.

7.3 Reflection

The research focuses on two aspects of software engineering: the flexibility of a software system as a quality indicator and the software architecture of a software system. In chapter three, the flexibility of a software indicator as a quality indicator is investigated. Within software engineering the quality indicators are ambiguously defined. Using a different approach then the one taken in chapter three might result in a different set of quality indicators or in a different definition for a specific quality indicator. The approach taken in chapter three is a qualitative analysis. A quantitative analysis, similar to Bonnet (2009), might embed the instruments better in existing software engineering research. The software architecture is linked to a business architecture using an ontology. The software architecture is described using models. This link between the business architecture and the software architecture can be improved by describing the software architecture using an existing enterprise architecture framework, like for example TOGAF (The open group, 2009). In retrospect the he software architecture could have been described within such an enterprise architecture, instead of next to a business architecture. This might make the link between a business architecture and the ontology-driven business architecture stronger and might result in more formal definitions for the software architecture.

Two important research fields are deliberately not addressed in this thesis. First, the way the business processes, functions and objects are decomposed into information technology tasks, functions and objects. The case study uses a process that is well known and well defined. Therefore the tasks, functions and objects were known on forehand. For other processes, that are less well known, the decomposition is an important issue. The software architecture supports refactoring: if the decomposition of the process, functions or objects changes, the source code of the software can be changed in controllable way. Second, the middleware (Erl, 2008) required to connect the individual services of the software architecture is not addressed. The middleware connects the services and enables communication between them. The interface and the infrastructure layers of the services are flexible: the service may support various middleware technologies.

7.4 Recommendations for further research

Future research can increase the link between a business architecture and the ontology-driven software architecture. Thereto the software architecture can be described within an existing enterprise architecture framework. The software architecture is applied to one case study. More case studies are needed to gain further insight. These extra case studies may also be used to identify more guidelines for the software engineering approach (chapter five).

Ideally all the actors involved share the same conceptualization. In that case the multi-actor complexity is reduced to single actor complexity. This is probably hard to achieve. However, reducing

the number of divergent views and thus the number of ontologies is possible. Further research is needed to design and evaluate successful governance policies in a smart business network.

The cost of the construction and maintenance of the software system was not part of the research. In the ontology based architecture the alignment between the names and definitions in the ontology and the source code is very important. Keeping the ontology, the models and the code aligned takes time. The maintenance cost may be reduced because the software system is easier to understand, reuse and maintain. With respect to other architectures, it is uncertain which kind of systems is cheaper to construct and maintain by using an ontology based architecture. Further research might reveal which characteristics of a software system play an important role. The cost of the construction and maintenance of software systems is hard to predict. The smallest unit of change (chapter six) may be used as an indicator for the estimation of the cost of the realization of a particular change. However, more insight in the relation between a smallest unit of change and the cost of a software system is needed.

The object oriented model links the ontology to the domain layer. The C# language is used as the object oriented programming language. The transformation of the terms and their relations of the ontology into the C# language are limited (chapter five). For example there is no support for type inference. Further research might yield a language that is better suited to represent the terms and the relations of the ontology within the domain layer of the service.

7.5 Recommendations for the IBLC

The architecture of the software system supports multiple ontologies. However adding ontologies to the system makes the system more complex and more expensive to construct and maintain. Therefore it’s important to minimize the number of ontologies in the system. Policies and processes need to be defined to create consensus amongst the actors to decrease the number of divergent views upon the business domain. These policies and process should use the theories addressed in sections two and three of chapter three.

The source code is neat and clean realization of the happy flow of the booking process. The software system may be extended to become the open digital platform. With respect to the booking process, more details about the business objects need to be included. With respect to other processes: the source code may be the basis or template for the construction of the open digital platform that will support the business processes of the Internet Business Learning Community.

The software system does not support the automatic integration of the contracts and delivery conditions of the suppliers of the composite product. However architecture allows designers to build rules that validate composite products. Therefore the software system may be extended to include some form of automatic integration.

The existing software development methodologies are not per se useable in a multi-actor network (chapter three). They deal primarily with the technical complexity, not with multi-actor complexity. It is important to keep the discussions between the actors about the specification of the software out of the software development process.

The applicability of code generation is not investigated in the research. All code of the system is constructed manually. However parts of the software system are suited to be generated based upon

predefined specifications. For example: code generation could speed up the construction of the relational model, by generating the model based upon the object model or code generation could speed up the construction of the transformations needed in the application layer (chapter four).

References

Adelaar, T., H. Bouwman, et al. (2004). "Enhancing customer value through click-and-mortar e-commerce: implications for geographical market reach and customer type." Telematics and Informatics 21(2): 167-182.

Antoniou, G. and F. Van Harmelen (2004). A Semantic Web Primer. The MIT Press, Cambridge.

Bachmann, F. And L. Bass (2001). Managing Variability in Software Architectures. SSR'01, May 18-20, 2001, Toronto.

Barbacci, M.R., M.H. Klein, T.H. Longstaff and C.B. Weinstock (1995). Quality Attributes, Technical Report CMU/SEI-95-TR-021. Carnegie Mellon University, Pittsburgh.

Barn, B.S. (2009). On the Evaluation of Reference Models for Software Engineering Practice. ISEC'09, February 23–26, 2009. ACM.

Bass, L., P. Clements, R. Kazman (2003). Software Architecture in Practice. Addison-Wesley, Reading.

Beck, K. (2003). Test-Driven Development: by Example. Addison-Wesley, Boston.

Beckers, J.M.J., W.P.M.H. Heemels, B.H.M. Bukkems and G.J. Muller (2007). Effective industrial modeling for high-tech systems: The example of Happy Flow. Seventeenth Annual International Symposium of the International Council On Systems Engineering (INCOSE).

Berns, G.M. (1984). Assessing Software Maintainability. Communications of the ACM, January 1984 Volume 27 Number 1.

Bessems, P. (2009). IBLC Group Business plan 2009 – 2012. Deploying people smarter in a community economy. IBLC, Eindhoven.

Bonnet, M. (2009). Measuring the Effectiveness of Enterprise Architecture Implementation. Master Thesis. Faculty of Technology, Policy and Management, TU Delft.

Boonzaaier, J. and J.J van Loggerenberg (2006). Implementation of a Project Office to Improve on Project Delivery and Performance: A Case Study. Proceedings of SAICSIT. pp. 206 –217.

Broy, M., F. Deissenboeck and M. Pizka (2006). Demystifying Maintainability. WoSQ '06: Proceedings of the 2006 international workshop on Software quality.

Bruijn, de J.A. and E.F. ten Heuvelhof (2002). Processmanagement, Over processontwerp en besluitvorming, 2[de] herziende druk. Academic Service, Den Haag.

Bruijn, de J.A. and E.F. ten Heuvelhof (2004). Management in netwerken. Lemma, Utrecht.

Booch, G., J. Rumbaugh and Jacobson, I. (2005). Unified Modeling Language User Guide. Second edition. Addison-Wesley, New York.

Byrd, T.A., B.R. Lewis and R.W. Bryan (2006). The leveraging influence of strategic alignment on IT investment: An empirical examination. Information & Management 43. pp. 308–321.

Cabrera L.F. and C.Kurt (2005). Web Services Architecture and its Specifications: Essentials for Understanding WS-*. Microsoft Press, Redmond.

Christopher M.G. (1998 and 2005), Logistics and supply chain management: strategies for reducing costs and improving services. Pitman Publishing, London.

Cockburn, A. (2000). Writing effective Use Cases. Addison-Wesley, Boston.

Cockburn, A. (2001). Agile Software Development. Addison-Wesley, Boston.

Codd, E.F. (1970). A Relational Model of Data for Large Shared Data Banks. Communications of the ACM, Vol. 13, No. 6.

Colomb, R.M. (2007). Ontology and the Semantic Web. IOS Press, Amsterdam.

Crosby, S. and D. Brown (2006). The Virtualization Reality. ACM Queue, december. pp. 34–41.

Cwalina K. and B. Abrams (2005). Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries. Addison-Wesley, Boston.

Darpa (2009). The DARPA Agent Markup Language. http://www.daml.org/index.html.

Date, C.J. (1975). An Introduction to Database Systems. Addison-Wesley, Reading.

DeLone, W.H. and E.R. McLean (1992). Information systems success: the quest for the dependent variable. Information Systems Research. pp. 60–95.

Delporte-Vermeiren, D.J.E. (2003). Improving the flexibility and profitability of ICT-enabled business network: an assessment method and tool. ERIM, Rotterdam.

Dijkstra, E. W. (1982). Selected Writings on Computing: A Personal Perspective. Springer-Verlag, Berlin. pp. 60-66.

Egyedi, T. M. and Z. Verwater-Lukszo (2005). Which standards' characteristics increase system flexibility? Comparing IT and batch processing infrastructures. Technology in Society 27. pp. 347–362.

Egyedi, T. M. (2007). Standard-compliant, but incompatible?! Computer Standards & Interfaces 29. pp. 605–613.

Erl, T. (2008). SOA: principles of service design. Pearson Education Inc., Boston.

Estublier, J. and G. Vega (2005). Reuse and Variability in Large Software Applications. ESEC-FSE'05, September 5–9, 2005, Lisbon.

Evans, R. (2004). Domain-Driven Design. Tackling Complexity in the Heart of Software. Addison-Wesley, Boston.

Fensel, D., F. Van Harmelen, I. Horrocks, D.L. McGuinness and P.F. Patel-Schneider. OIL: An Ontology Infrastructure for the Semantic Web. IEEE Intelligent Systems 16. pp. 38-45.

Fielding, R.T. (2000). Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine.

Fowler, M. (2000). Refactoring. Improving the Design of Existing Code. Addison-Wesley, Boston.

Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley, Boston.

Gamma E., R. Helm, R. Johnson and J. Vlissides (1995). Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston.

Gašević, D., D. Djurić and V. Devedžić (2009). Model Driven Engineering and Ontology Development. Second edition. Springer-Verlag, Berlin.

Gerbranda, S. (2008). Managing the Young and Ambitious. A Strategic and Financial Perspective on the Deployment of a New Generation Employees. Master Thesis. Tilburg University, Financial Management.

Glass, R. (2008). A Ancient (but Still Valid?) Look at the Classification of testing. IEEE Software, November/December. pp 111-112.

Gruber, T.R. (1993). A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition, 5. pp. 199-220.

Hadaya, P. and L. Cassivi (2007). The role of joint collaboration planning actions in a demand-driven supply chain. Industrial Management & Data Systems, Vol. 107 No. 7. pp. 954-978.

Hacki, R. and Lighton, J. (2001). The future of the networked company. The McKinsey Quarterly. pp. 26-39.

Hansen, M.D. (2007). SOA Using Java Web Services. Prentice Hall, Upper Saddle River.

Heck, E. Van, P. Vervest and K. Preiss (2006). Smart Business Networks, a new Business Paradigm. SBNi, Rotterdam.

Heck, E. van and P. Vervest (2008). Smart Business Networks: How the Network Wins. In: Vervest, P.H.M., et al. Smart Business Networks. A new Business Paradigm. SBNi, Rotterdam. pp. 15-29.

Henderson, J.C. and N. Venkatraman, 1993. Strategic alignment: leveraging information technology for transforming organizations. IBM Systems Journal 32.

Holzmüller, H.H. and J. Schlüchter (2002). Delphi study about the future of B2B marketplaces in Germany. Electronic Commerce Research and Applications 1. pp. 2–19.

Huang, S.-J. and R. Lai (2003). Measuring the Maintainability of a Communication Protocol Based on Its Formal Specification. IEEE Transactions on Software Engineering, Vol. 29, No. 4.

Hunt, A and D. Thomas (1999). The pragmatic programmer. Addison Wesley, Boston.

Jacobson, I., M. Griss and P. Johnsson (1997). Software Reuse. Architecture, Process and Organisation for Business Success. Addison-Wesley, New York.

Janssen, M. (2005). Engineering a Software-Oriented Architecture in E-Government. In Stojanovic, Z. And A. Dahanayke (2005). Service-Oriented Software Engineering. Challenges and Practices. Idea Group, Hershey.

Janssen, M. and H.J. Scholl (2007). Interoperability for Electronic Governance. ICEGOV2007, December 10-13, 2007, Macao.

Kagdi, H. and J.L. Maletic (2006). Software-change Prediction: Estimated and Actual. Second International IEEE Workshop on Software Evolvability, 24-24 Sept. 2006. pp. 38–43.

Kar, E. van de (2004). Designing Mobile Information Services. An Approach for Organisations in a Value Network. Phd Thesis, TU Delft.

Kar, E. van de and A. Verbraeck (2007). Designing Mobile Service Systems. IOS Press, Amsterdam.

Kazi, I.H., H. H. Chen, B. Stanley and D.J. Lilja (2000). Techniques for Obtaining High Performance in Java Programs. ACM Computing Surveys, Vol. 32, No. 3. pp. 213–240.

Koning, M., Chang. Sun, M. Sinnema and P. Avgeriou (2009). VxBPEL: Supporting variability for Web services in BPEL. Information and Software Technology 51. pp. 258–269.

Kroll, P. and B. MacIsaac (2006). Agility and Discipline made easy. Practices from OpenUP and RUP. Adisson-Wesley, Upper Saddle River.

Kratz, B. (2003). Empirical Research on the Relationship between Functionality and Interfaces of Software Components. Master Thesis. Tilburg University, Department of Information Systems and Management.

Kruchten P. (1991). The Rational Unified Process: An Introduction. Addison-Wesley Longman, Boston.

Kruchten P. (1995). Architectural Blueprints—The “4+1” View Model of Software Architecture. IEEE Software 12 (6). pp. 42-50.

Kundu, D., Sarma M. and D. Samanta (2008). A Novel Approach to System Testing and Reliability Assessment Using Use Case Model. ISEC’08, February 19-22, Hyderabad, India.

Kuaté, P.H., T. Harris, C. Bauer and G. King (2009). NHibernate in Action. Manning, Greenwich.

Larman, C (2005). Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design and Iterative Development. Third edition. Pearson Education Inc., Upper Saddle River.

Levinson, J. and D. Nelson. Pro Visual Studio 2005 Team System. Apress, New York.

Löwy, J. (2007). Programming WCF Services. O’Reilly, Sebastopol.

Malveau, R. and T. J. Mowbray (2001). Software Architect Bootcamp. Prentice Hall, Upper Saddle River.

Martin, R.C. (1994). OO Design Quality Metrics. An Analysis of Dependencies. Available online: http://www.objectmentor.com/resources/articles/oodmetrc.pdf.

Martin, R.C. (2006). Agile Principles, Patterns, and Practices. Pearson Education, Boston.

Martin, R.C. (2009). Clean Code. A handbook of Agile Software Craftsmanship. Pearson Education, Boston.

Martin, J. (1991). Rapid Application Development. Macmillan Publishing Company, New York.

McConnell, S. (1993). Code Complete. A Practical Handbook of Software Construction. Microsoft Press, Redmond.

Michelson, B. M. (2006). Event-Driven Architecture Overview. Patricia Seybold Group, Boston.

Navarra, D. D. and T. Cornford (2008). Globalization, networks, and governance: Researching global ICT programs. Government Information Quarterly. pp. 35–41.

Nelson, K. M. and H. J.Nelson (1997). Technology Flexibility: Conceptualization, Validation, and Measurement. IEEE Computer Society, Washington.

Nicola, A. De, Michele M. and R. Navigli (2009). A software engineering approach to ontology building. Information Systems 34. pp. 258-275.

Onna, van M. and A. Koning (2003). The little Prince 2. A practical guide to project management. Third edition. Ten Hagen Stam Publishers, Den Haag.

Oosterhout, M. Van, E. van Heck, E. Waarts and J. van Hillegersberg (2008). Business Agility Implications for Smart Business Networks. In: Ververst, P. , E. Van Heck and K. Preiss (2008). Smart Business Networks a new Business Paradigm. SBNi, Rotterdam.

Oxford (1996). The Oxford Dictionary and Thesaurus, American Edition. Oxford University Press, New York.

Phillips, P.A. and Wright, C. (2008). E-business's impact on organizational flexibility. Journal of Business Research, doi:10.1016/j.jbusres.2008.09.014.

Porter M.E. (1985), Competitive Advantage: Creating and Sustaining Superior Performance, The free Press, New York.

Ribeiro, L., J. Barata and A. Colombo (2009). Supporting agile supply chains using a service-oriented shop floor. Engineering Applications of Artificial Intelligence. Elsevier.

Rine, D.C. (1995). Success factors for software reuse that are applicable across domains and businesses. SAC '97: Proceedings of the 1997 ACM symposium on applied computing.

Roehling, M.V., M.A. Cavanaugh, L.M. Moynihan and W.R. Boswell (2000). The Nature of The new Employment Relationship: A Content Analysis of the Practitioner and Academic Literatures. Human Resource Management, 39 (4). pp. 305–320.

Ross, D.T. (1985). Applications and Extensions of SADT. IEEE Computer 18. pp. 25-34.

Royce, W. (1970). Managing the Development of Large Software Systems. Proceedings of IEEE WESCON 26 (August). pp. 1-9.

Rozansky, N. And E. Woods (2005). Software Systems Architecture. Working with stakeholders using viewpoints and perspectives. Pearson Education, Upper Saddle River.

Sametinger, J. (1997). Software Engineering with Reusable Components. Springer-Verlag, Berlin.

Sanchez, R. (1995). Strategic Flexibility in Product Competition. Strategic Management Journal. pp. 135-159.

Shah, R., J. Kesan, and A. Kennis (2008). Implementing Open Standards: A Case Study of the Massachusetts Open Formats Policy. The proceedings of the 9th Annual International Digital Government Research Conference.

Siricharoen, W.V. (2007). Ontologies and Object models in Object Oriented Software Engineering. IAENG International Journal of Computer Science, Vol. 33, No. 1.

Stabell, C.B. and Ø.D. Fjeldstad. Configuring Value for Competitive advantage: on Chains, Shops, and Networks. Strategic Management Journal, Vol. 19. pp. 413–437.

Stapleton, J. (1997). DSDM: The method in Practice. Addison Wesley Longman, New York .

Swanson, E.B. (1999). IS "Maintainability": Should It Reduce the Maintenance Effort? The DATA BASE for Advances in Information Systems - Winter 1999 (Vol. 30, No. 1).

Szyperski, C, D. Gruntz and S. Murer (2002). Component Software. Beyond Object-Oriented Programming. Pearson Education, Edinburgh.

Tallon, P.P., 2008. A Process-Oriented Perspective on the Alignment of Information Technology and Business Strategy. Journal of Management Information Systems / Winter 2007–8, Vol. 24, No. 3, pp. 227–268.

The Open Group (2009). TOGAF Version 9. The Open Group Architecture Framework (TOGAF). The Open Group, San Francisco.

Thompson, S.H.T. and J.S.K. Ang (1999). Critical success factors in the alignment of IS plans with business plans. International Journal of Information Management 19. pp. 173-185.

Tweedale, J. and Jain, L. (2009). The Evolution of Intelligent Agents within the World Wide Web. In: Nguyen, N.T. and L.C. Jain (2009). Intelligent Agents in the Evolution of Web and Applications. Springer-Verlag. Berlin.

Uschold, M. and M. Gruninger (2004). Ontologies and Semantics for Seamless Connectivity. SIGMOD Record, Vol. 33, No. 4. pp. 58 - 64.

Verschuren, P. and H. Doorewaard (1999). Designing a Research Project. Lemma, Utrecht.

Versteeg, G. And H. Bouwman (2006). Business architecture: A new paradigm to relate business strategy to ICT. Information systems frontiers 8. pp. 91-102.

Vervest, P.H.M. and L. Zheng, (2008). The Network Experience – New Value from Smart Business Networks. In: Ververst, P. , E. Van Heck and K. Preiss (2008). Smart Business Networks a new Business Paradigm. SBNi, Rotterdam.

W3C (2009). World Wide Web Consortium. http://www.w3.org.

Wang, E. T. G., P. H. Ju, et al. (2008). The effects of change control and management review on software flexibility and project performance. Information & Management. pp. 438-443.

Wessel, M. and R. Möller (2009). Flexible software architectures for ontology-based information systems. Journal of Applied Logic 7. pp. 75–99.

WS-I (2006). Basic Profile Version 1.1. http://www.ws-i.org/Profiles/BasicProfile-1.1.html

WS-I (2009). Web Services Interoperability Organization. http://www.ws-i.org.

Yang, J. and M.P. Papazoglou (2002). Web Component: A Substrate for Web Service Reuse and Composition. CAISE 2002, LNCS 2348. pp. 21–36.

Appendix A The process diagrams

This appendix contains the process diagrams for the software system. The process diagrams are described using the structured analysis and design technique (SADT). The basic element of the diagram is a process (figure A1). Each process has a function name and a function number. Process A0 is the main process, this process is decomposed into sub processes A1 to A3. An arrow from the left to the rectangle represents the input to the process an arrow from the rectangle to the right represents the output of the process. Arrows from the top to the rectangle represent the control of the process. Arrows from the bottom to the rectangle represent a mechanism, that is: people and systems.

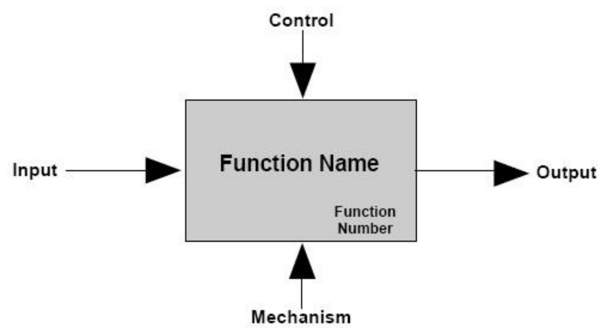


Figure A1: The basic SADT element.

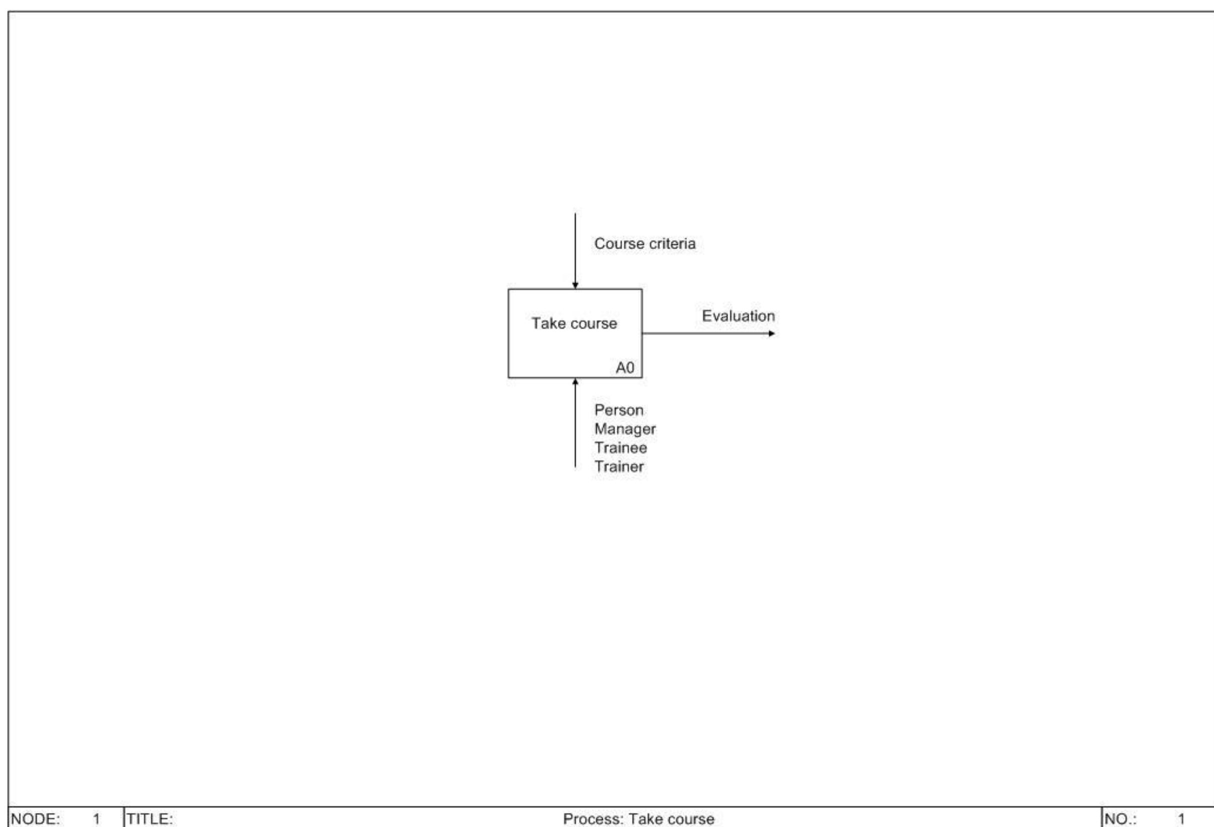


Figure A2: The top level process diagram.

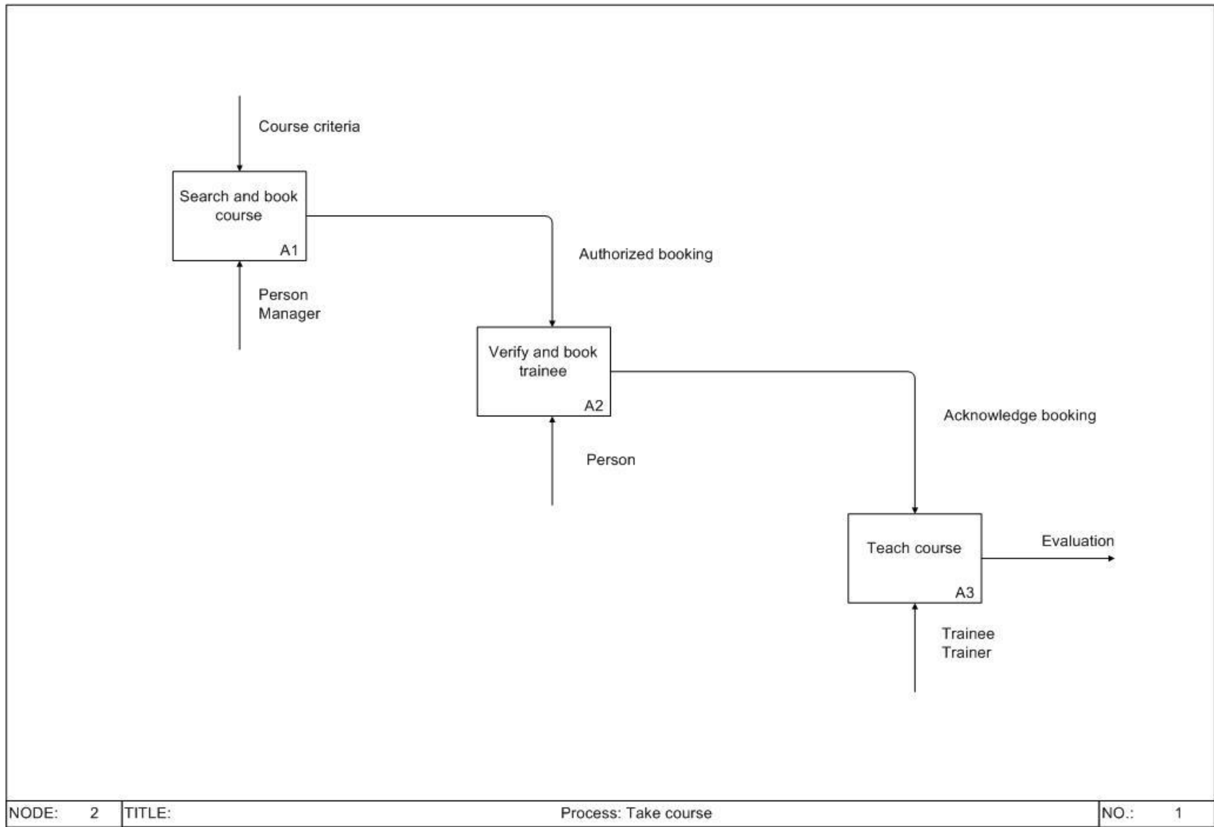


Figure A3: The first decomposition of the process diagram.

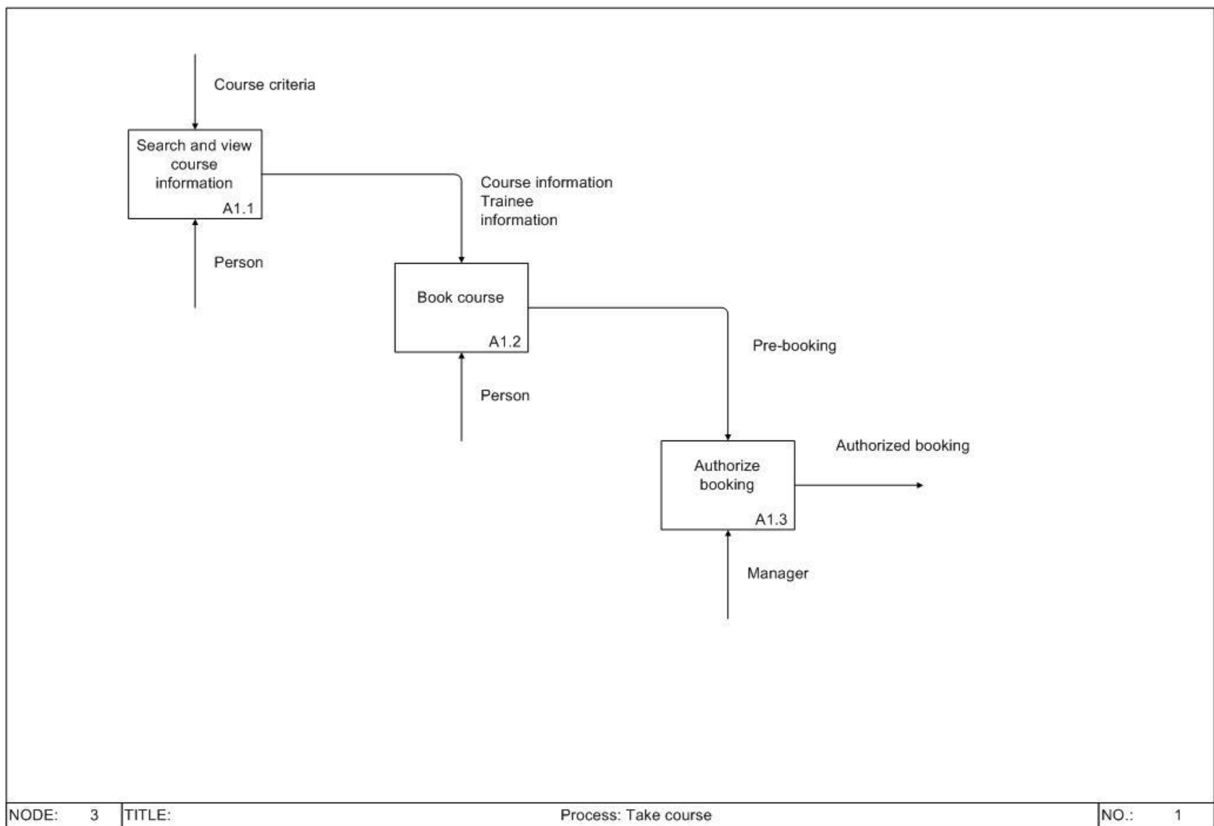


Figure A4: The decomposition of level A1.

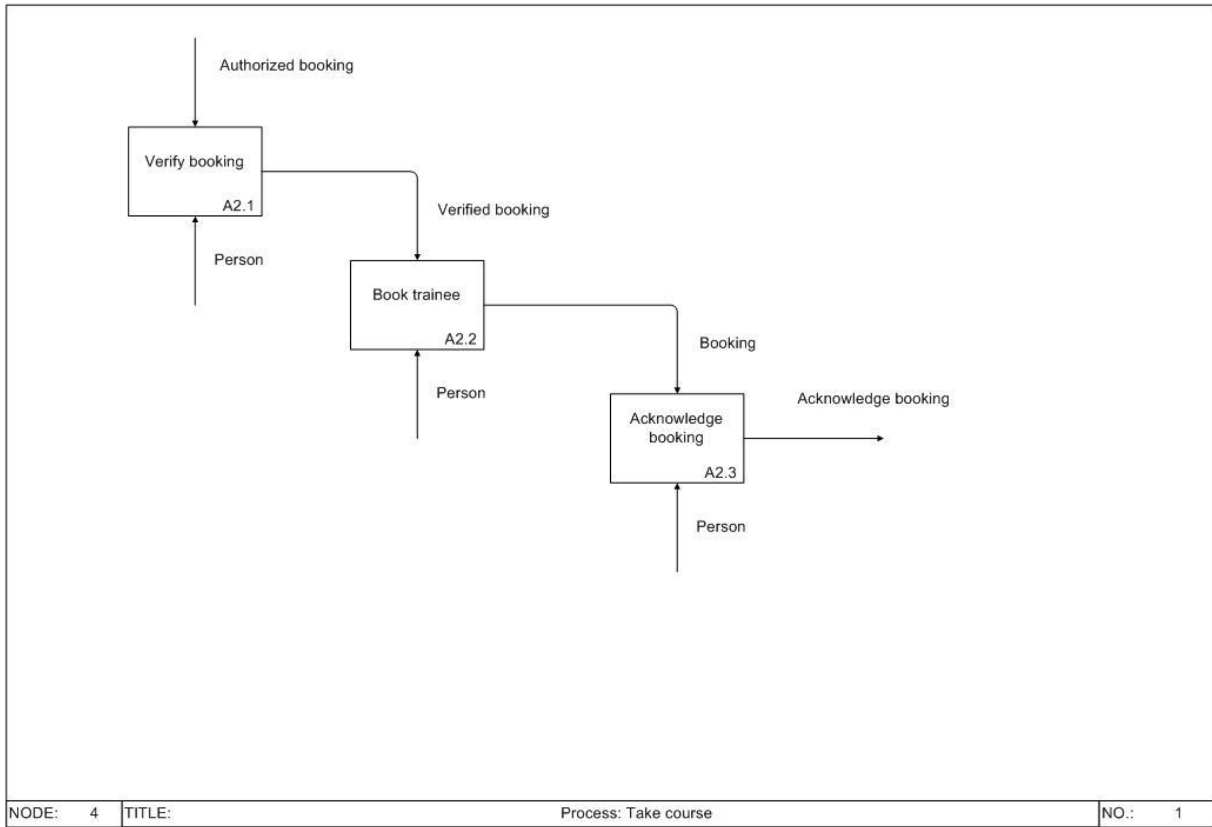


Figure A5: The decomposition of level A2.

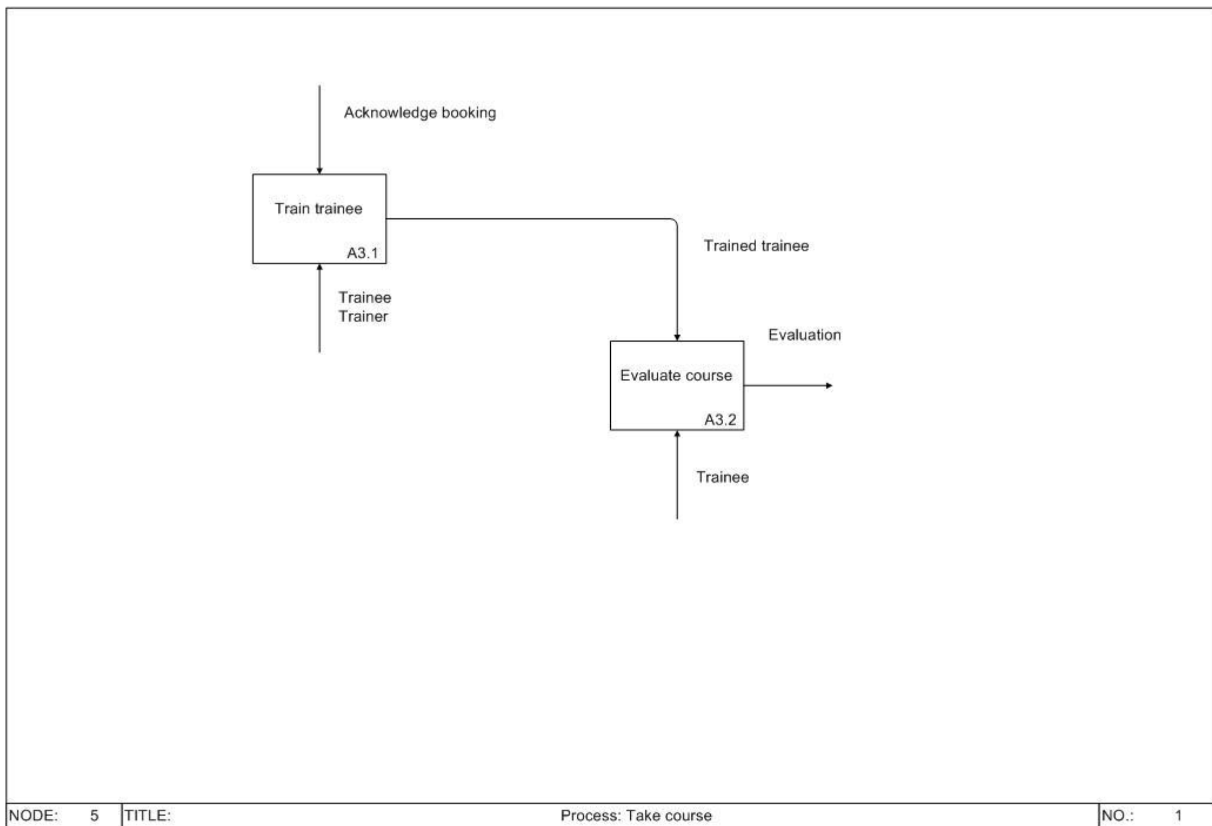


Figure A6: The decomposition of level A3.

Appendix B The ontology

This appendix contains examples of the ontology. The ontology is described using snippets of the ontology. The complete ontology is stored in an xml file that is a little over a thousand lines long. The xml file is a valid web ontology language for description logics (OWL-DL).

The ontology is constructed using a huge set of Microsoft Excel and Word files containing the vocabulary and definitions written by Paul Bessems (CEO of IBLC). The ontology is constructed using Protégé 4.0[28], an open-source ontology editor and knowledge-base framework. Figure B1 is a screenshot of Protégé showing the Booking class.

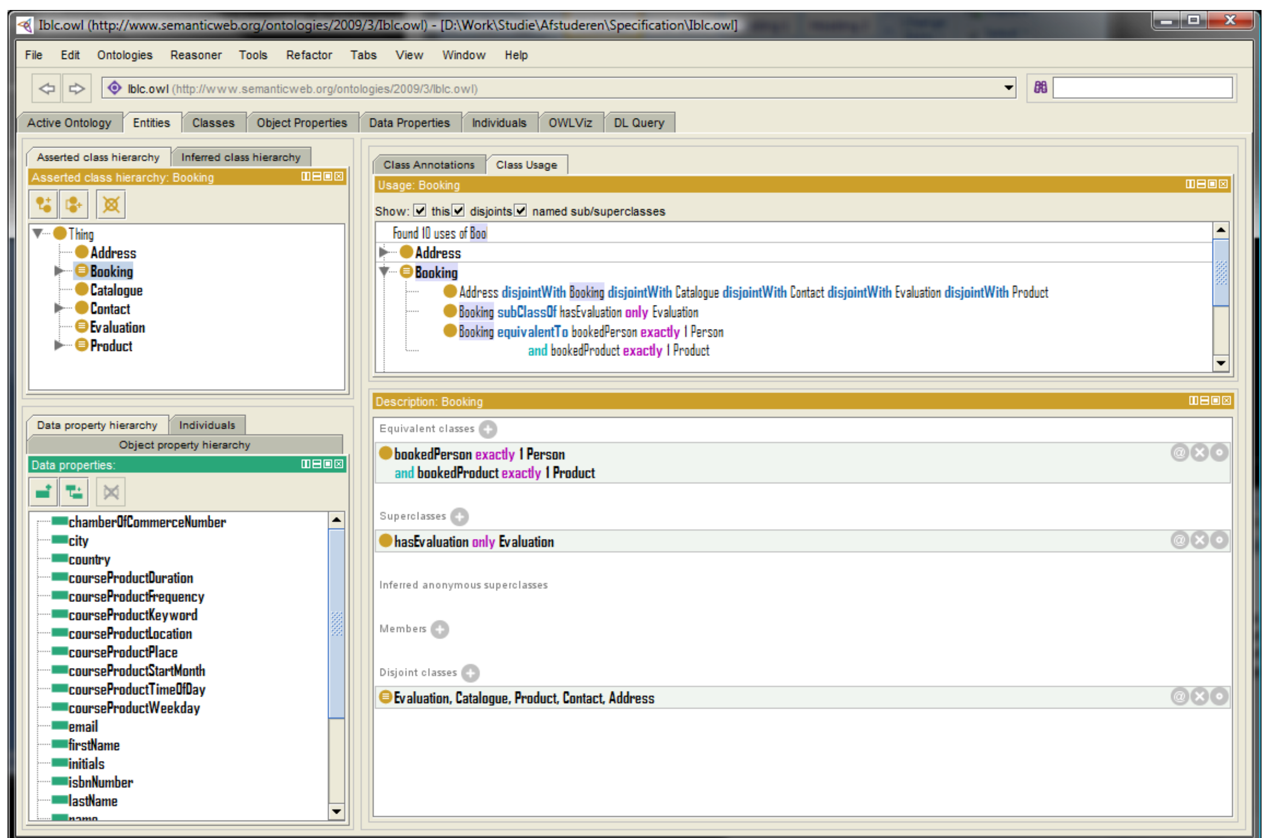


Figure B1: Screenshot of Protégé showing the Booking class.

The core of the ontology is the taxonomy. The taxonomy describes the classes of the ontology and their relations. The taxonomy of the ontology can be found in figure B2. As the relations between the classes in a taxonomy are inheritance relations, Protégé qualifies them as a "is-a" relation.

[28] Protégé is freely available on the internet at: http://protege.stanford.edu.

Figure B2: Taxonomy of the ontology.

The following xml snippets describe the class *Booking* defined within the ontology. The *Booking* class has an object property called *bookedPerson*. Object properties relate objects to other objects. The *bookedPerson* property relates objects of the class *Booking* to objects of the class *Person*. The cardinality restriction on the property states that for an object of the class *Booking* it is required to have a relation to exactly one object of the class *Person*. The *bookedProduct* property states something similar for the class *Product*. Therefore, this definition of the *Booking* class defines a booking as a combination of exactly one person and exactly one product. The *hasEvaluation* property relates objects of the class *Booking* to objects of the class *Evaluation*. An object of the class *Booking* is not required to have a relation with one object of the class *Evaluation*. However it may have relations with more than one object of the class *Evaluation*.

The OWL specification is build upon the RDF and RDFS specifications. The xml root element defines the namespaces for rdf, rdfs and owl:

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
```

The example of the owl definition of the class *Booking* is as follows:

```
<owl:Class rdf:about="#Booking">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
```

```
        <owl:Restriction>
          <owl:onProperty rdf:resource="#bookedPerson"/>
          <owl:onClass rdf:resource="#Person"/>
          <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#bookedProduct"/>
          <owl:onClass rdf:resource="#Product"/>
          <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasEvaluation"/>
      <owl:allValuesFrom rdf:resource="#Evaluation"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Appendix C The use case model

This appendix contains an example of the use case model. The template for the description of the use case specifications is defined by Cockburn (2001). The use cases are described focusing only on the happy flow. The happy flow of a model is a deliberate simplification of the model (Beckers et al., 2007). The desired behaviour is modelled; all disturbances or possible error situations are left out.

Use case #	1. Get a list of suppliers by name.	
Goal in context	This use case enables the actor to search for a specific supplier. The actor supplies a part of the name of the supplier. The service returns a list of suppliers that match the name.	
Scope and level	The use case is defined for the context of the actor IBLC and the domain human resource management. It specifies part of the functionality of the contacts service. The contacts service is part of the entity service layer.	
Preconditions	There are no preconditions	
Success end condition	A list of suppliers is returned to the actor. If no suppliers are available, an empty list is returned.	
Failed end condition	None	
Primary	The product service and the booking service of the IBLC.	
Secondary actors	The future services of the system.	
Trigger	The use case starts when the contact service receives a valid request.	
Description	Step	Action
	1	The name of the requested suppliers is retrieved from the request message.
	2	The name from the request message is match with the names of the suppliers. A match is found if the name from the request message is the same as a part of the name of the supplier.
	3	A list of found suppliers is returned. The list contains the identifier, the name, the E-mail address and the address of the website of each supplier.
Extensions		None
Sub-variations		If no suppliers are found: return an empty list.

Table C1: Use case specification for the contacts service.

Appendix D The object oriented model

This appendix contains examples of the relational model. The object oriented model is realized in C# code.

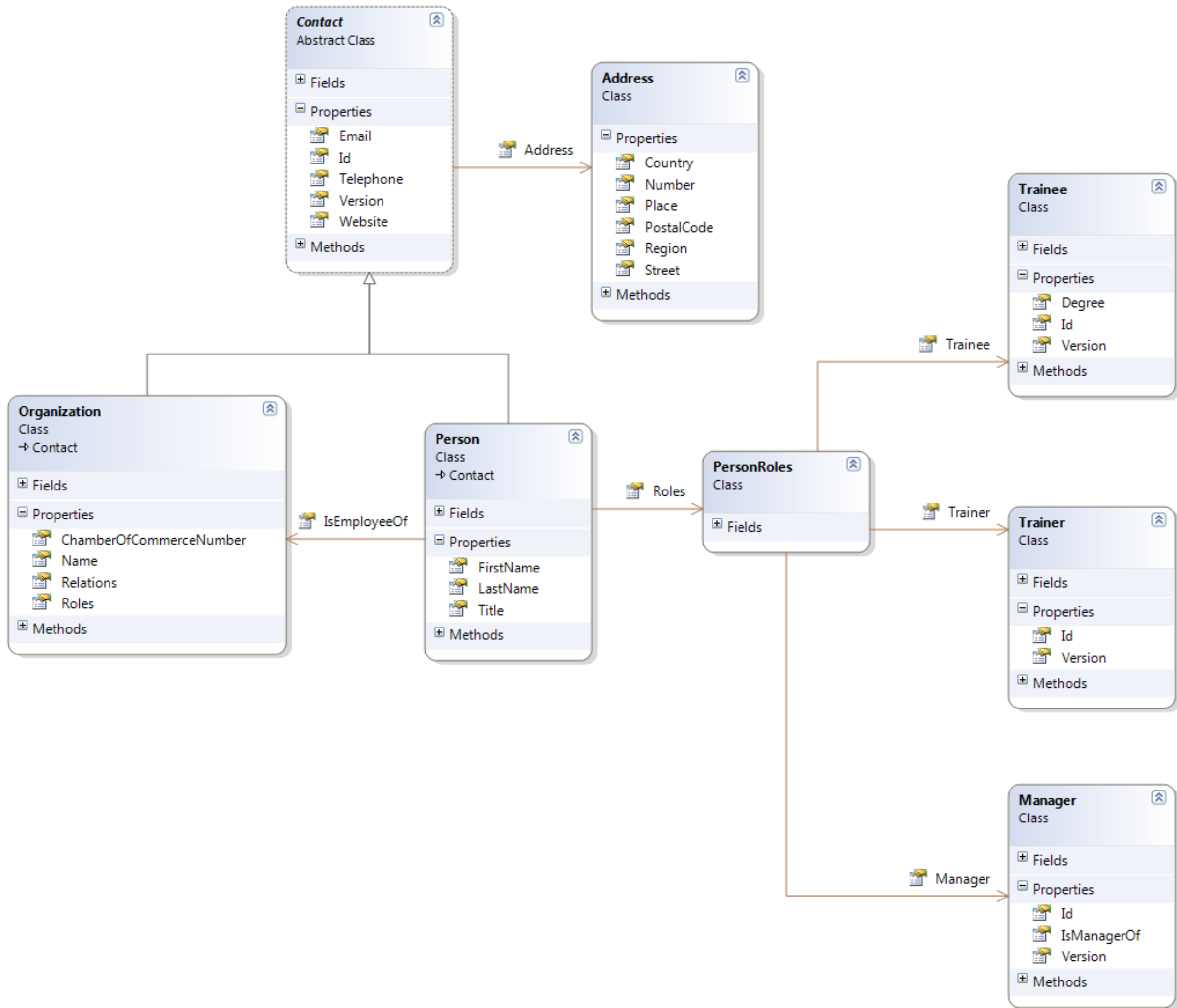


Figure D1: Part of the object oriented model of the contact service.

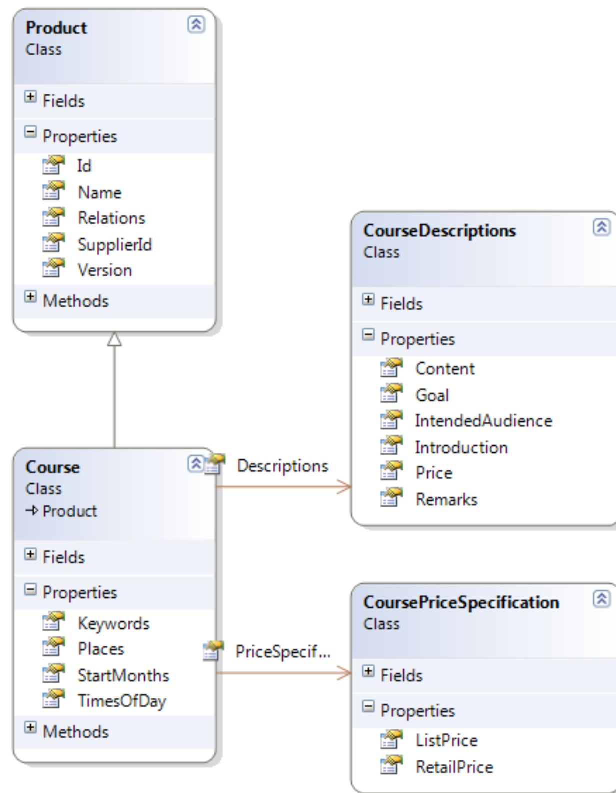


Figure D2: Part of the object oriented model of the product service.

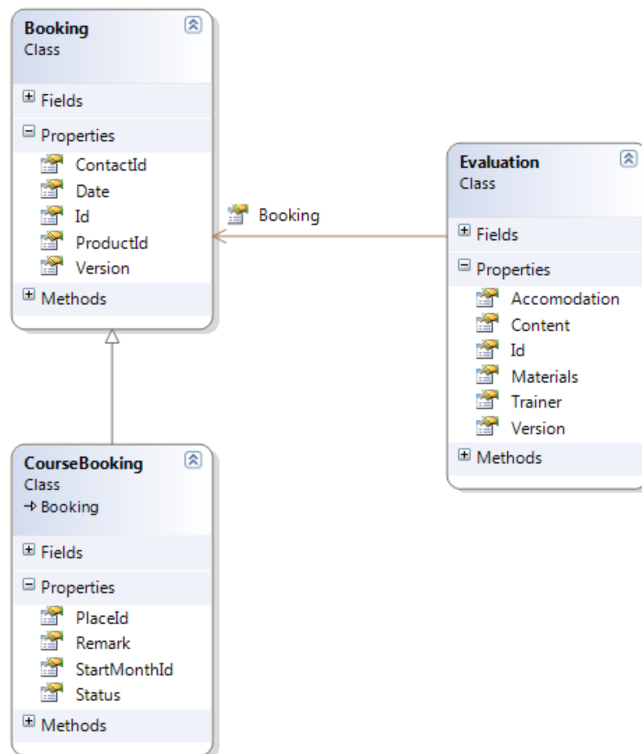


Figure D2: Part of the object oriented model of the booking service.

Appendix E The interface model

This appendix contains the interface model. The interface definition of the contact service is used as an example. This interface contains an operation named *GetSuppliers*. The method accepts a request message containing a part of the name of a supplier. The operation returns a response message containing a list of suppliers whose name matches the name in the request message. The list of suppliers contains the *Id*, *Name*, *Email* and *Website* of each supplier.

The snippet of the web services description language (WSDL) that defines the operation *GetSuppliers*:

```
<wsdl:message name="ISupplierService_GetSuppliers_InputMessage">
  <wsdl:part name="parameters" element="tns:GetSuppliers" />
</wsdl:message>
<wsdl:message name="ISupplierService_GetSuppliers_OutputMessage">
  <wsdl:part name="parameters" element="tns:GetSuppliersResponse" />
</wsdl:message>
<wsdl:portType name="ISupplierService">
  <wsdl:operation name="GetSuppliers">
    <wsdl:input wsaw:Action="http://tempuri.org/ISupplierService/GetSuppliers"
message="tns:ISupplierService_GetSuppliers_InputMessage" />
    <wsdl:output wsaw:Action="http://tempuri.org/ISupplierService/GetSuppliersResponse"
message="tns:ISupplierService_GetSuppliers_OutputMessage" />
  </wsdl:operation>
</wsdl:portType>
```

The snippet that defines the input message for the *GetSuppliers***:**

```
<xs:element name="GetSuppliers">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" name="name" nillable="true" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The snippet that defines the output message for the *GetSuppliers***:**

```
<xs:element name="GetSuppliersResponse">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="Supplier" nillable="true"
type="tns:Supplier" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Supplier">
    <xs:sequence>
      <xs:element minOccurs="0" name="Email" nillable="true" type="xs:string" />
      <xs:element minOccurs="0" name="Id" type="xs:long" />
      <xs:element minOccurs="0" name="Name" nillable="true" type="xs:string" />
      <xs:element minOccurs="0" name="Website" nillable="true" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The definition of a supplier in the interface model differs significantly from the definition of a supplier in the object oriented model (figure E1). The class structure in the object oriented model is replaced by a single data type. The data type contains just the necessary information, nothing more, nothing less. This makes connecting with the services easier.

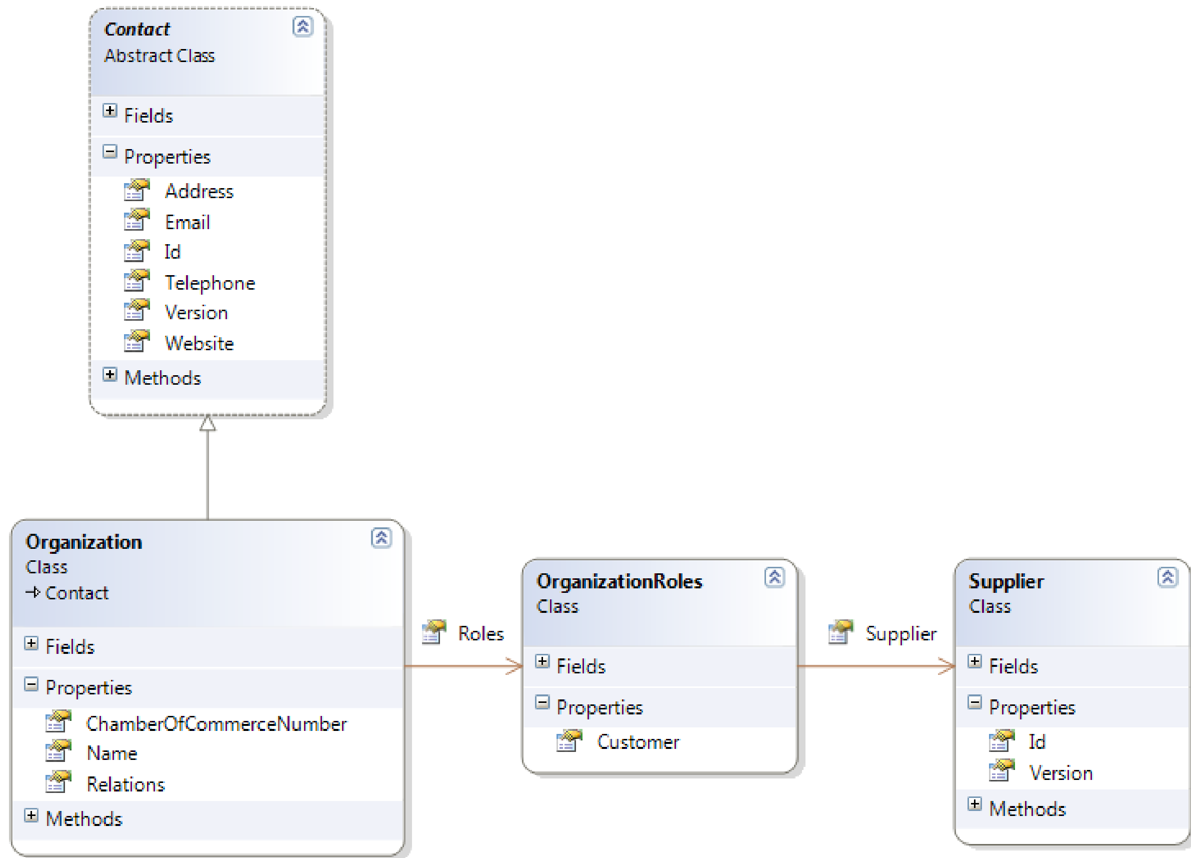


Figure E1: The supplier classes in the object oriented model.

Appendix F The relational model

This appendix contains the relational model. The relational model is defined using structured query language (SQL) statements. The complete definition is stored in a set of files.

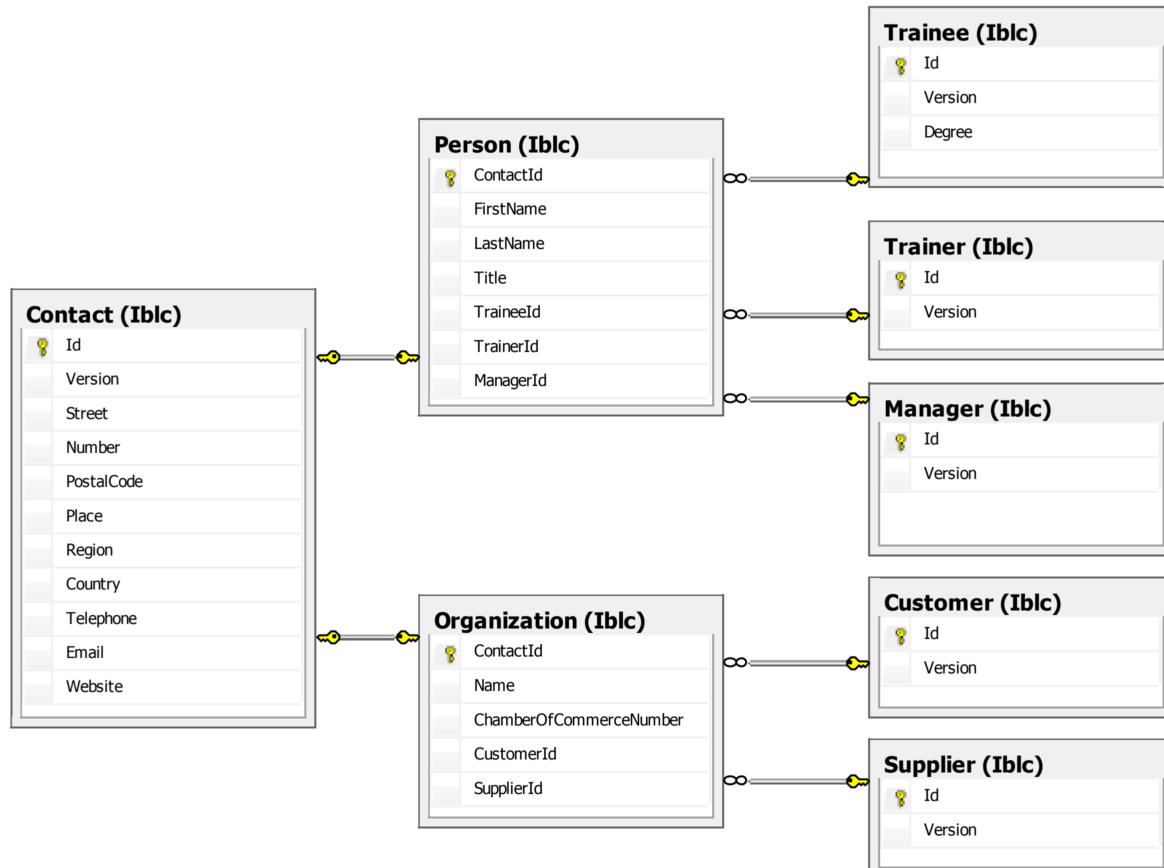


Figure F1. Relational model for the contact service.

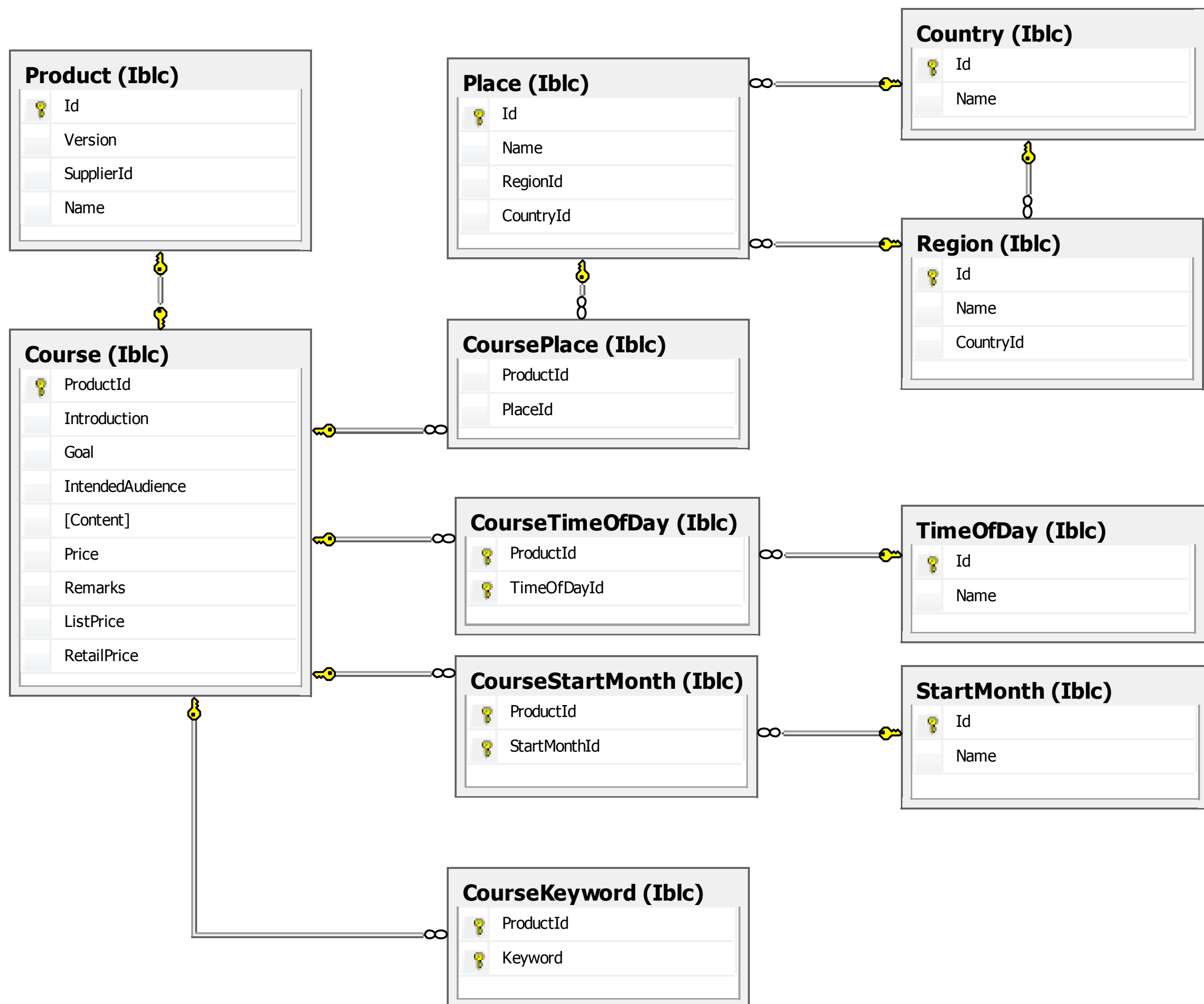


Figure F2: Relational model for the product service.

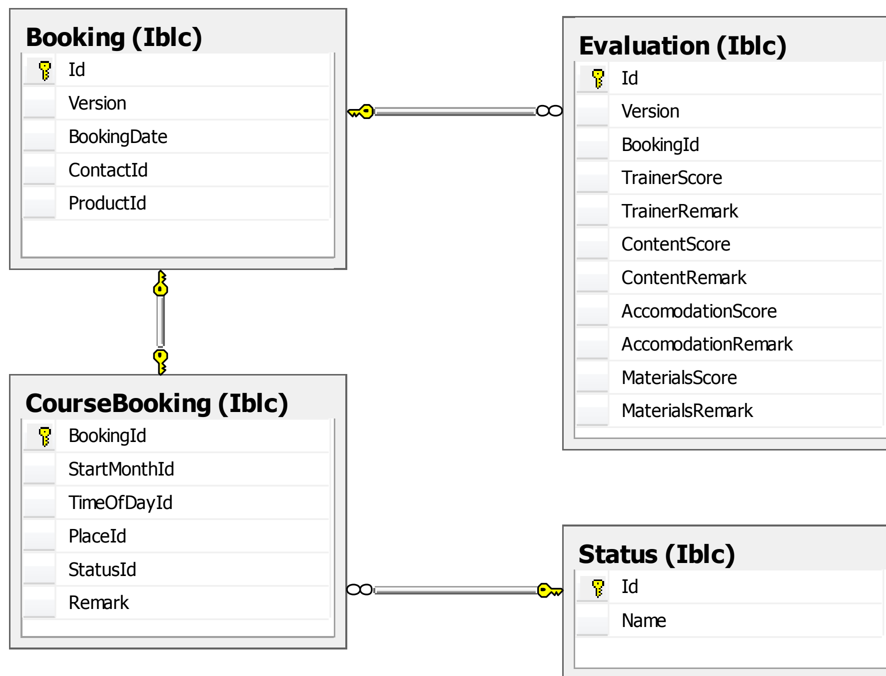
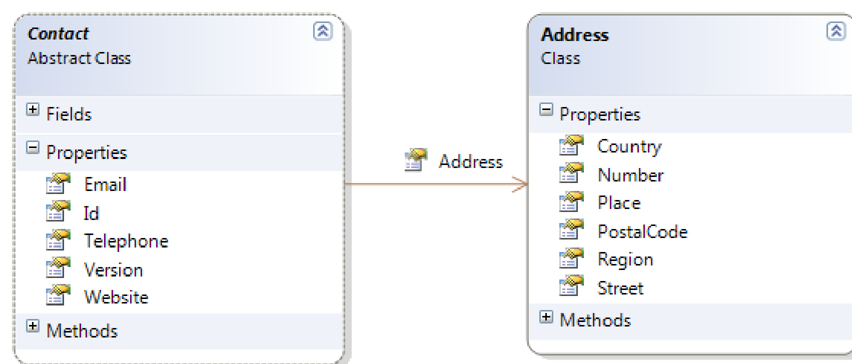


Figure F3: Relational model for the booking service.

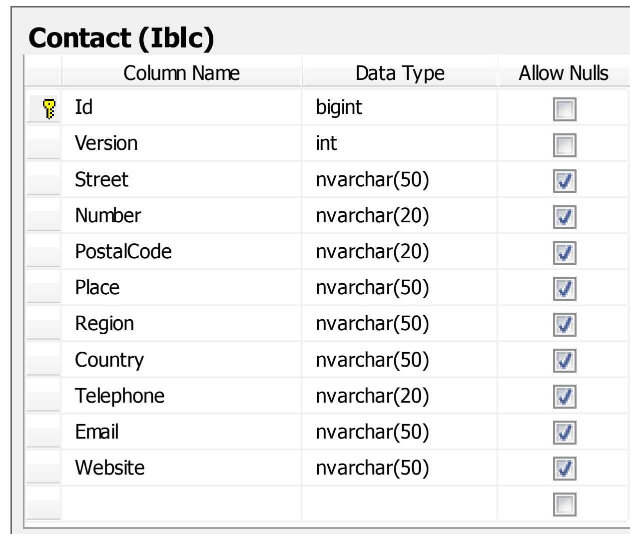
Appendix G From object oriented model to the relational model

This appendix contains an example of the mappings between the object oriented model and the relational model. The object oriented model is expressed in C#. The relational model is defined using structured query language (SQL) statements. The mapping between the classes of the object oriented model and the tables of the relational model are defined in xml files. These files are constructed according to the specifications of the NHibernate object relational mapper (Kuaté, 2009).

The following xml snippets describe the mapping of the classes *Contact* and *Address* (figure G1) to the table *Contact (figure G2)*. The xml element named *class* has an attribute called *name*. This attribute contains the fully qualified name of the C# class that is mapped. The same element also has an attribute called *table*. That attribute contains the name of the database table. The attribute *name* of the element *id* corresponds to the name of the C# property that contains the identifier of the class. The attribute *column* of the element *id* corresponds to the name of the database column that contains the primary key of the table. The element *generator* defines how unique identifiers and primary keys are handled. The value *identity* of the attribute *class* denotes that the identifiers are generated by the database server. The element *version* is used to track the updates of the information. Each time the row in the table is update the version is incremented by one. The properties with the names *Telephone*, *Email* and *Website* are properties of the C# class *Contact* and stored in the table *Contact*. The properties named: *Street*, *Number*, *PostalCode*, *Place*, *Region* and *Country* are properties of the C# class *Address. These* properties are also stored in the table *Contact*. Therefore this mapping maps two C# classes *Contact* and *Address* to one table *Contact*. The object relational mapper supports numerous other ways of mapping classes to tables. Some of those are very powerful and can be used very complex mappings.



FigureG1: The part of the object oriented model to be mapped.



Contact (Iblc)

	Column Name	Data Type	Allow Nulls
🔑	Id	bigint	☐
	Version	int	☐
	Street	nvarchar(50)	☑
	Number	nvarchar(20)	☑
	PostalCode	nvarchar(20)	☑
	Place	nvarchar(50)	☑
	Region	nvarchar(50)	☑
	Country	nvarchar(50)	☑
	Telephone	nvarchar(20)	☑
	Email	nvarchar(50)	☑
	Website	nvarchar(50)	☑
			☐

Figure G2: The part of the relational model to be mapped.

The example of the definition of the mapping is as follows:

```
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" assembly="Iblc.Contacts.Domain" auto-
import="false" default-access="field.camelcase">
  <class name="Iblc.Contacts.Domain.Entities.Contact" table="Contact">
    <id name="Id" column="Id" unsaved-value="0" >
      <generator class="identity"/>
    </id>
    <version name="Version" column="Version"/>

    <component name="Address" class="Iblc.Contacts.Domain.ValueObjects.Address">
      <property name="Street" column="Street" access="property"/>
      <property name="Number" column="Number" access="property"/>
      <property name="PostalCode" column="PostalCode" access="property"/>
      <property name="Place" column="Place" access="property"/>
      <property name="Region" column="Region" access="property"/>
      <property name="Country" column="Country" access="property"/>
    </component>

    <property name="Telephone" column="Telephone"/>
    <property name="Email" column="Email"/>
    <property name="Website" column="Website"/>
  </class>
</hibernate-mapping>
```