HARMONIZING NORMS WITH A SOFTWARE DESIGN

Master Thesis

Sopo Esiava
4256069
MSc Management of Technology

***Graduation Committee**:*
Chair: Prof. dr. ir. Marijn Janssen
First Supervisor: Dr. Huib Aldewereld
Second Supervisor: Dr. Neelke Doorn



September, 2014

ACKNOWLEDGMENTS

As this master thesis marks the end of my studies at Delft University of Technology, I would like to use this opportunity and thank everyone who has influenced my two-year journey. Those last two years have not been easy. In fact, it has been the most challenging, stressful, and at the same time - the most inspiring, productive, and exciting journey. Journey, which has completely changed my life and has prepared me for new challenges and opportunities.

I owe a very special debt to the ex president of Georgia – Mikheil Saakashvili, for initiating a scholarship program for Georgian students abroad, willing to pursue master's degree. I would like to thank Mr. Saakashvili for giving me a chance to be one of those students, and for fully sponsoring my studies at TU Delft.

I must express special gratitude to my graduation committee for being way more supportive then I could ever imagine. I would like to thank them for all their feedback, support, and encouragement, for being very understanding, and shortly, for making it possible for me to finalize my thesis and to graduate. I would like to acknowledge the selfless help and support of my supervisor – Dr. Huib Aldewereld. His guidance and support encouraged me to be motivated and to keep working on the thesis. I would like to thank him for all his time that he has spent on reading my thesis, on giving me direct comments, and further motivation. I had times when I felt lost in my own thesis, and I was ready to give up, but Dr. Aldewereld always managed to encourage and motivate me. I would like to thank him for arranging meetings (lots of meetings) with me to discuss my progress, whenever I asked for it, despite his very busy schedule. Afterall, more than once, he has used his lunch hours to make sure that I kept working on my thesis.

I want to thank my parents and my sisters for always being there for me, for supporting me in every way, and for being proud of me no matter what. I want to especially thank my mother for being a woman I always wished I could be. I want to thank her for her endless love and encouragement. If not her, I would not be where I am right now, and I will be forever thankful for this. Together with my family, I want to thank Jack and Debbie Grinnell for playing a huge role in shaping my personality. I want to thank them for everything they have done for me, and for being as important in my life as my family is. I am honored to be able to refer to Jack and Debbie as to my parents, and to be treated as their own daughter.

I owe a very special thank you to my fellow group mate Kaisar Siregar - who, unfortunately, is no longer with us, but…. is always with us. From the very first day we met, he has always been there for me and I really wish I could share this moment with him.

Among my friends at TU Delft, I would like to especially thank Oscar Diaz-Cardiel, Nikos Beliokas, and Panos Liaros for never leaving me alone and for always being there for me. I want to thank them for encouraging me to always keep my head up and to never give up chasing my dreams. I want to thank them for making me laugh even when I was crying, and simply, for being my friends.

I want to thank Mariam Kata Qimeridze for motivating me to start my studies at TU Delft, and for sharing every single minute of my journey. I want to thank her for believing in me even when I did not believe in myself. I want to thank her for being part of my life, forever.

Finally, I owe a very special thank you to a very special person in my life – Stephan van Es. I would like to thank him for giving me a reason to wake up every morning, for dealing with my mood swings during my stressful days, and for always holding me so I would not fall. I also want to thank him for changing my perspective about the Netherlands and for making me fall in love with this country. I will be forever grateful for everything he has done (and keeps doing) for me, and I owe him a very special thank you for all the memories that we share together.

Sopo Esiava
September, 2014

Table of Contents

MANAGEMENT SUMMARY **5**

1. INTRODUCTION **6**

1.1. Research Scope and Research Objective **8**
1.2. Core Concepts **10**
1.3. Research Approach **11**

2. THEORETICAL BACKGROUND **14**

2.1. Value Sensitive Design (VSD) **14**
2.1.1. VSD-Integrative Tripartite Methodology **15**
2.1.2. Values Hierarchy **16**
2.2. Critical Design **18**
2.3. Approaches to Software Development Process **20**
2.3.1. Waterfall Model **21**
2.3.2. Agile Software Development Methods **22**
2.3.3. Agent-Oriented Software Engineering (AOSE) **25**
2.3.4. Service-Oriented Architecture (SOA) **30**
2.4. Critical Reflection on Theoretical Background **33**

3. THE MODEL DEVELOPMENT **35**

3.1. The Conceptual Model **35**
3.2. The Model for Incorporating a Concrete Set of Norms with a Software Design Process **38**

4. CASE STUDY **44**

5. CONCLUSION **60**

6. LIMITATIONS AND FUTURE WORK **62**

REFERENCES **64**

MANAGEMENT SUMMARY

In today's highly technology-based environment designers are becoming increasingly aware of values and norms that are embodied in technical systems. However, values and norms are only implicitly involved in a design process and generally, are addressed as an afterthought. Practical problem at stake is that there is no practical model, or guidelines for incorporating values and norms into a technology design. In regard to this problem, the presented master thesis develops a model for incorporating a concrete set of norms into a software design, explicitly during the software development/design process. The development of this model is motivated by the theories on Value Sensitive Design and Critical Design. Based on the characteristics of different approaches to a software development process, this research suggests applying the developed model to a software development process managed by one of the Agile Software Development Methods.

The thesis is structured as follows: chapter 1 will introduce the research problem at stake, and will discuss the research objective, research scope, research questions, and research approach. In chapter 2, theories related to this research will be discussed and analyzed, based on which, the model will be developed in chapter 3. The latter will be followed by the Case Study, where the developed model will be applied to an example case of software development process within the domain of organ donation. The conclusions and limitations (along with the recommendations for the future work) will be discussed in chapter 4 and chapter 5, respectively.

Keywords: *Values, Norms, Critical Design, Value Sensitive Design, VSSD framework, Software Design, Agile Software Development Methods.*

1. INTRODUCTION

The concept "value", in a narrow sense, refers to the economic worth of an object, but in the broader sense "value" is defined as "the principles or standards of a person or society, the personal or societal judgment of what is valuable and important in life" *(Simpson and Weiner, 1989)*. In this thesis, the term "value" will be used with its broader meaning. Friedman and her colleague researchers have developed a list of the "values of ethical import" that are often implicated in system design: human welfare, ownership and property, privacy, freedom from bias, universal usability, trust, autonomy, informed consent, accountability, courtesy, identity, calmness, environmental sustainability. There is a longstanding interest in designing information systems that support such human values. Values of ethical import have already been addressed by number of different researchers in the last twenty years *(Friedman et al, 2006)*. However, there is limited research focused on producing practical methodologies for systematically incorporating ethical values into a design process. Even though designers and engineers have become increasingly aware of social and ethical values that are embodied in technical systems, putting the values into practice explicitly during the design process is not straightforward. Especially in today's highly technology-based environment, which involves various stakeholders and therefore multi-actor complexity, many designers struggle in finding a balance between their own, users', stakeholders', and surrounding culture's values *(Flanagan et al, 2005)*.

Highly promising development in this area is the emergence of Critical Design *(Dunne, 2006, Dunne and Raby, 2001, Gaver et al., 2004)*. Critical Design theory has been used in design-related areas as a practice of reflecting on values, norms, and beliefs that are incorporated in design and artifacts *(Dunne and Raby, 2001)*. In other words, Critical Design is a way to get both users and designers to reflect on their own practice and to critically question values (and norms) reproduced in design and in design practice *(Bardzell, J. and Bardzell, S., 2013)*. The world we live in today is very complex. Society constantly moves on. From the beginning of 21st century we are experiencing complex technological, political, economic, and social changes. By challenging assumptions and preconceptions about the role technology (and design) plays in everyday life, Critical Design makes us think. It raises our awareness, provokes action, and sparks debate. By doing so, Critical Design remains relevant to the complex changes that we are experiencing in 21st century.

Designing a technology that accounts for human values and norms is important for organizations, especially for the ones that are highly technology based. In order to better understand this importance, let us consider an example of supply chain management. Supply chain is about managing a network of several interconnected businesses that are involved in a process of delivering a product to an end customer. Throughout this process, flow of information from one stage to another plays a crucial role because it ensures effective decision-making at the planning and execution stages. When it comes to the flow of information, no other tool or resource can do it as quickly and accurately as a technology. IT-enabled infrastructure capabilities not only help organizations achieve higher efficiency, but also reduce cycle time, ensure delivery of goods and services in timely manner and improve overall supply chain process *(Fouda, 2012)*.

With every passing year, managing supply chain becomes more complex. Organizations have to cope with different regulations, and norms throughout the process. Besides, in today's highly technology based

environment, customers are familiar with online ordering, with the ability to track shipments to a timely and expected delivery date. Therefore, customers are more demanding, more aware, and less patient. So, let us assume that an organization aims to develop a software for improving its supply chain visibility and business execution. The question is, how would this organization design and develop a software that accounts for abstract values and norms?

There is apparently a need for a theoretical and methodological framework with which to discuss the value dimensions of design work. One of the efforts to provide such framework is Value Sensitive Design (VSD) – 'a theoretically grounded approach to the design of technology that accounts for human values in a principled and comprehensive manner throughout the design process' *(Friedman et al, 2006).*

The Value Sensitive Design methodology can be applied in various domains of information systems design and engineering. The focus of this thesis, however, is on software engineering (software design process, in particular). Society is facing a remarkable progress towards the next generation Internet. In a highly diverse and rapidly changing environment, software services and applications provide a "sound backbone" for the next generation Internet by enabling the effective integration of new applications and information technologies *(Aldewereld et al, 2013).* However, in the mid nineties, 'researchers (such as: Winograd, Friedman, Perry, Shneiderman, Nissenbaum) showed that software could easily come to contain biases, arbitrary assumptions, and peculiar worldviews of makers, which could affect users in various ways' *(van den Hoven, 2007).* When designing and developing the software, developers and designers have to make various design decisions capturing key design issues and the rationale behind chosen solutions. Decisions made during the software design process have an impact on non-functional characteristics of software, such as software quality attributes *(Aldewereld et al., 2013).* In "Design for Values in Software Development", H. Aldewereld and his colleagues elaborate on the Value Sensitive Software Development (VSSD) framework "as a basis to specify and analyse complex adaptive systems from a value sensitive perspective". The VSSD framework is based on the ethical principles from the Value Sensitive Design.

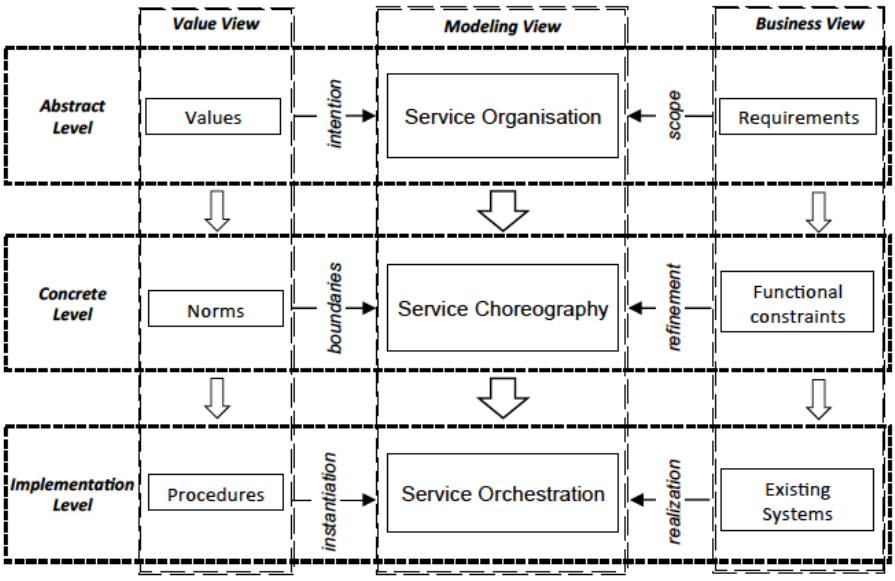


Figure _1: Value Sensitive Software Development (VSSD) Framework

(Aldewereld et al., 2013)

As we can see on Figure_1 above, the Value Sensitive Software Development (VSSD) framework is considered from three complementary views: (1) the *Values View* – specifies mechanisms of social order in terms of values, norms and procedures. (2) the *Business View* – exposes the contextual aspects of the domain, based on existing systems and services. (3) the *Modeling View* – exposes the architectural design of the system, based on value decisions and domain characteristics. In order to make the translations between the different elements of the three complementary views explicit, those above mentioned views are explored on the following levels of abstraction: Abstract level, Concrete level, and Implementation level. Considering the above mentioned views at three different abstraction levels facilitates the analysis and design of systems by providing the means to link abstract concepts to its implementation counterparts. The distinction between the levels of abstraction is made in order to capture all elements at their native level of abstraction, and to make the translations between the different elements explicit *(Aldewereld et al., 2013).*

While the vertical translations within the views of VSSD framework are well understood, the relations between the views are still missing. Values, in general, are expressed on a high-level of abstraction (making values particularly difficult to implement in the software systems) and practical problem at stake is that currently there is no approach, or a procedure, which would guide the implementation of the values in software systems explicitly during the software design process *(Aldewereld et al., 2013).*

1.1. RESEARCH SCOPE AND RESEARCH OBJECTIVE

In the introduction of this thesis, the importance of incorporating values explicitly into a software design process was discussed. It was also mentioned that because of the high level of abstraction of ethical values, the incorporation of values into a design process is not straightforward.

Values of ethical import are relevant for evaluating the worth or goodness of certain options, or certain decision during a software design process (or any systems design process, in general). However, values do not directly imply restrictions or prescriptions for actions and decisions during a design process. As already mentioned above, values are general and abstract.

Norms, on the other hand, are deontic, which means that norms express duty or obligation and are related to ethical concepts. Norms are deontic because they articulate restrictions and prescriptions for an action *(van de Poel, 2013)*. "The norms will provide guidance to determine whether certain patterns of behavior are legal or acceptable within the given contexts. Once the norms are understood, captured and represented, this will serve as a basis for programming intelligent agents to perform many regular activities" *(Liu et al., 2001).*

Let us go back to the Value Sensitive Software Development (VSSD) framework. Specific model components on the VSSD framework are specified on the concrete level, based on the domain analysis and value design processes (Aldewereld et al., 2013) and the Service Choreography on this level is the

blueprint which describes the software design requirements and the ways of obtaining the desired results. As mentioned in introduction, practical problem at stake is that currently there is no approach, or a procedure, which would guide the implementation of the values in software systems explicitly during the software design process (Aldewereld et al., 2013).

Focusing on the links between the three levels of the VSSD framework is outside the scope of this research, and this thesis will focus on the link between the Norms and the Service Choreography on the Concrete Level of the VSSD framework. In other words, within the scope of this research is to make it clear how Norms can be translated into the software design requirements.

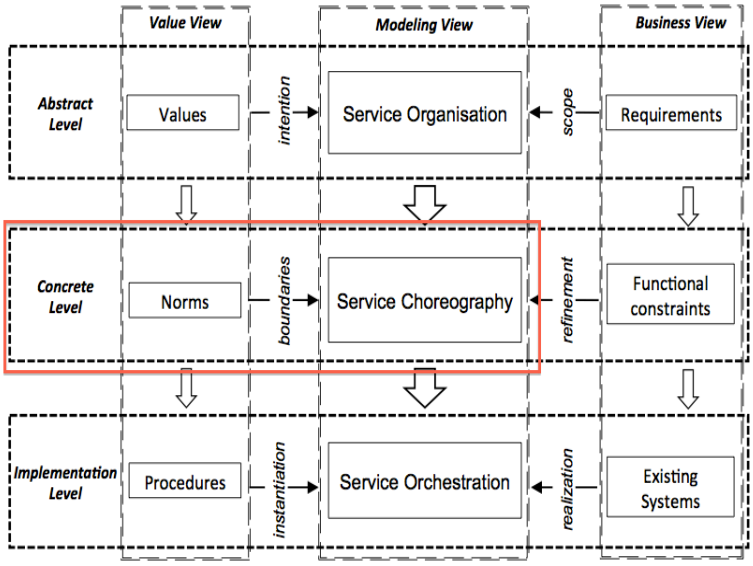


Figure _2: Value Sensitive Software Development (VSSD) Framework

(Aldewereld et al., 2013)

In order to incorporate values (which are general and abstract) into a software design process, first step is to translate general values into a set of more concrete norms *(van de Poel, 2013)*. Translation of abstract values into a concrete set of norms is outside the scope of this research, because the focus of this research is on the process of incorporating the concrete set of norms (already translated from the abstract values) into a software design process. The objective of the proposed research is ***to recommend a model for incorporating a concrete set of norms into a software design, explicitly during the software design process.***

Based on the above discussed research objective, the following main research question is formulated:

How can a concrete set of norms be incorporated into a software design process?

This research aims to answer the main research question by recommending a specific model, which will guide the incorporation of norms with a software design process. For developing this specific model, the research aims to examine theories on Value Sensitive Design and Critical Design, as well as various approaches to a software development process. It has to be identified how those relevant theories can be

helpful for developing a model which will be used for incorporating norms with a software design. Therefore, the following three sub-questions (for the main research question) are formulated:

- ***How can the theory on Value Sensitive Design contribute to this research?***
- ***How can the theory on Critical Design contribute to this research?***
- ***What are the various approaches to a software development process and how can they be helpful for this research?***

Relevant findings will be reviewed and presented during discussing the theoretical background of this research in Chapter 2. In other words, answers to the sub research questions will be used as a guide for finding the answer to the main research question.

1.2. CORE CONCEPTS

The objective of this research, as already discussed, is to develop a model for integrating a concrete set of norms into a software design. Based on this research objective, norms and software development process are core concepts of this research (because the model to be developed will integrate *norms* into a software design explicitly during the *software development process*). Those core concepts, along with brief explanations of what makes them core for the research, are discussed below:

Norms

Norms are formal and informal rules that might not be formally and explicitly defined, but have an affect on the software design process (and on the final product). During the software development process, norms will provide guidance for developers whether certain patterns of behavior are "legal or acceptable within the given context" *(Liu et al., 2001)*. Norms usually promote some underlying goals and values, hence modeling norms needs to support the representation of these values (and value preferences) *(Bench-Capon, 2002)*. Once captured, represented, and understood, concrete norms (translated from the general and abstract values) will serve as a basis for developing and designing software *(Liu et al., 2001)*.

In "Normative Reasoning and Consequence" *(Broersen et al., 2013)* Broersen and colleagues elaborate on two types of norms: *constitutive norms* and *regulative norms. Constitutive norms* make possible institutional actions such as the making of contracts, issuing of fines, decreeing of divorces. In other words, *constitutive norms* "tell us what counts as what for a given institution" *(Broersen et al., 2013). Regulative norms,* on the other hand, describe obligations, prohibitions, and permissions. For example, the following premise: "bikes count as vehicles" would be a constitutive norm, while the premise: "vehicles are not allowed to access public parks, therefore, bikes are not allowed to access public parks" would be a regulative norm *(Broersen et al., 2013)*.

Since *regulative norms* do not just tell us what counts as what, but they prescribe obligations, prohibitions, and permissions, this research, in order to develop a model for integrating the concrete set of norms into the software design process, will focus on translating *regulative norms* into the software design requirements during the software development process.

Software Development Process

Software development process is the set of tools, methods, and practices used to produce a software product" *(Humphrey, 1989)*.
Number of approaches to software development process is large *(MacCormack et al., 2003)*. Traditional approaches to software development (waterfall model, for example) assume that a complete set of software design requirements can be anticipated early, and eliminating change can reduce the cost of developing software. However, in today's rapidly changing environment, eliminating change early means business failure. External environmental changes cause critical variations in software development process. Besides, market's expectations for software have grown over years. Market demands (and expects) innovative, high-quality software that meets its needs. Agile Methods for software development are a response to expectations to today's market. Main strategy of Agile Methods is to reduce the cost of change throughout a project *(Highsmith and Cockburn, 2001)*. Erickson et al. define agility as follows: 'agility means to strip away as much of the heaviness, commonly associated with the traditional software-development methodologies, as possible to promote quick response to changing environments, changes in user requirements, accelerated project deadlines and the like' *(Erickson et al., 2005)*.
Various approaches to software development process are reviewed below in chapter 2.

1.3. RESEARCH APPROACH

The proposed research is design oriented, since the objective of this research is to develop a specific model for incorporating a concrete set of norms into a software design process. The design cycle for developing the model within the scope of this research consists of five phases (Figure_3):

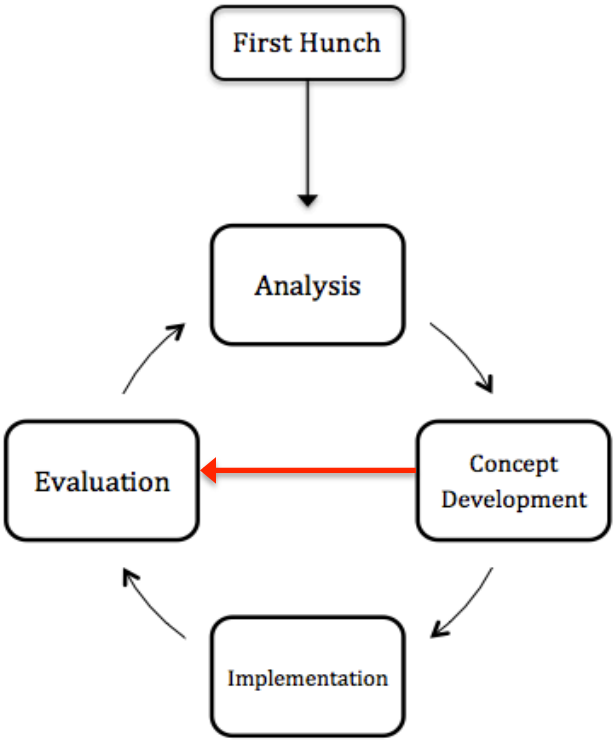


Figure_3: Research Approach

The very first phase of the cycle is the First Hunch, where the initiative for developing the model appears. The phase of the first hunch aims to define set of goals to be realized with the developed model and in case of the proposed research, this goal is to incorporate norms with the software design process. After the Analysis phase (which is the next phase of the design cycle) comes the Concept Development phase, during which the model should be developed (based on the results from two previous phases). Concept Development is followed by the Implementation phase, where the model developed during the Concept Development is put into a practice (in a real life context, preferably) and the final phase of the design cycle for this research is the Evaluation. Evaluation phase aims to check whether the developed model satisfies the goals defined in the first phase of the cycle, and whether the developed model satisfies the expectations of the designer (and other stakeholders) *(Verschuren and Hartog, 2005)*.

Above described design process is iterative, which means that "the designer continuously goes back and forth between the several stages (at least mentally), looking what repercussions a decision in one stage has for earlier as well as for later stages" *(Verschuren and Hartog, 2005)*. Please note that due to the time

available for this research, the Implementation Phase will fall outside the scope of this study and instead of implementing the developed model in a real life context, the Concept Development Phase will be followed by the Evaluation Phase, during which a specific example case (from Dutch Transplant Association, 2006) will be used to show how developed model can be used in practice. The model itself will be developed based on the theoretical background and examination of theories and practices that are relevant to the objective of the proposed research. Even though the proposed research methodology raises a question of validity of the research, the result – recommended model, will be a step ahead for the future research in this particular domain.

2. THEORETICAL BACKGROUND

In the following chapter, theories relevant to the process of incorporating values and norms into a design process are examined. By doing so, this chapter provides theoretical background for this research, which later will be used to develop a model for incorporating a concrete set of norms into a software design.

First, theory on Value Sensitive Design (VSD) is reviewed, as it focuses on the necessity of incorporating values into technology, explicitly during the technology design process, and provides guidelines (or at least claims to) on how to do so. Next, theory on Critical Design is summarized. Critical Design, which focuses on questioning and critically reflecting on norms throughout a design process, can be applied in various design processes, however, this thesis will focus on how ideas behind Critical Design can contribute to the process of incorporating norms into a software design process. As already mentioned in introduction of this paper, number of approaches to a software design/development process varies. It was also discussed that the actual process of incorporating abstract values into a design process is not simple, as it involves the translation of values into norms, and then - norms into design requirements. Therefore, after the review of Critical Design, different approaches to and models of software development process are discussed, in order to examine which method creates room for flexibility, necessary for incorporating norms into software design process. Critical reflections on the reviewed literature are discussed at the end of the chapter.

2.1. VALUE SENSITIVE DESIGN (VSD)

Value Sensitive Design is relevant to this research because, as mentioned in the introduction of this chapter, VSD focuses on incorporating values into technology design. In this section, the VSD approach will be further examined in order to see how it can be helpful for the research, and what are the guidelines that the VSD approach provides for incorporating values and norms into a design process.

Value Sensitive Design (VSD) is a theoretically grounded approach to the design of technology that accounts for human values in a comprehensive and principled manner throughout the design process *(Friedman et al., 2002)*.
Early interest in computer technology, values, and design emerged in the works of Wiener (1954), Weizenbaum (1972), and others *(Friedman et al., 2006)*. Supporting human values through system design has emerged within such approaches as: Computer Ethics, Social Informatics, Participatory Design, and Computer-Supported Cooperative Work. Computer Ethics advances our understanding of values that lie at the intersection of technology and human lives *(Bynum, 2001, Johnson and Miller, 1997, Nissenbaum, 1999),* however, Computer Ethics focuses on a single value at a time and remains too divorced from technical implementations *(Friedman and Kahn, 2002)*. Informatics provides socio-technical analyses of deployed technologies *(Johnsons, 2000, Kling et al., 1998, Kling and Star, 1998, Orlikowsi and Iacono, 2001, Sawyer and Rosenbaum, 2000)* but it says little about the design process itself *(Friedman et al.,*

2002). Participatory Design substantively embeds democratic values into its practice *(Bjerknes and Bratteteig, 1995, Bødker, 1987, Carroll and Rosson, 2006),* however, when applied in diverse contexts, it may not provide enough guidance *(Friedman et al., 2002)*. Computer-Supported Cooperative Work has been successful in the design of new technologies to help people collaborate effectively in the workplace *(Fuchs, 1999, Galegher et al., 1990, Grudin, 1988, Olson and Teasley, 1996),* but this approach narrowly frames its domain and its emphasis on the value of cooperation *(Friedman et al., 2002)*.

Value Sensitive Design shares and adopts some interests and techniques from the above-discussed approaches, but as Friedman and her colleagues argue, VSD also brings forward a constellation of following features:

- *Value Sensitive Design seeks to proactively influence the design of technology early in and throughout the design process.*
- *It offers a methodology that employs conceptual empirical, and technical investigations, applied iteratively.*
- *VSD identifies and focuses on (1) direct stakeholders – the ones directly interacting with the system, and (2) indirect stakeholders – the ones who are affected by the use of the system;*
- *VSD's interactional position holds that "technology's actual use depends on the goals of the people interacting with it". (Friedman et al., 2006).*

To better understand the approach of Value Sensitive Design, the following section 2.1.1. summarizes a VSD methodology and how it can be deployed in practice.

2.1.1. VSD – INTEGRATIVE TRIPARTITE METHODOLOGY

As mentioned above, Value Sensitive Design builds on an iterative methodology that integrates conceptual, empirical, and technical investigations. Each of those investigations are separately discussed below:

Conceptual Investigations

Friedman and colleagues define conceptual investigations as philosophically informed analyses of the central constructs and issues under investigation. During the design process, conceptual investigation aims to answer questions such as: what are values? How are moral values supported (or diminished) by particular technology design? Whose values should be supported in the design process? and etc.

In other words, by conceptual investigations VSD offers a philosophically informed working conceptualization of specific values. This is an important step in the design process, since working conceptualizations of specific values clarify fundamental issues raised by the specific design process at hand. Conceptual investigations offer thoughtful consideration of how stakeholders (direct stakeholders, as well as indirect stakeholders) might be socially impacted by technological designs *(Friedman et al., 2002).*

Empirical Investigations

By empirical investigations, Value Sensitive Design investigates the human context in which the technical artifact (or system) is situated. Besides, in order to evaluate the success of a particular design, empirical investigations are often needed.

To summarize shortly, empirical investigations, as Friedman and her colleagues explain, "encompass any human activity that can be observed, measured, or documented" *(Friedman et al., 2002)*. Questions that empirical investigations aim to answer include (but are not limited to): how do direct/indirect stakeholders apprehend individual values in the interactive context? How do stakeholders prioritize individual values? How do stakeholders prioritize competing values in design trade-offs? What are the differences between what people say (espoused practice) and what people do (actual practice)? How can designers bring values into consideration? etc. *(Friedman et al., 2002)*.

Technical Investigations

One of the aims of Value Sensitive Design is to focus on how existing technological properties and underlying mechanisms support or hinder human values. This is done during the technical investigations. Technical investigations also involve the proactive design of systems to support values that usually are identified during the conceptual investigation.

Technical investigations involve technological as well as empirical activities. Those activities are also involved in empirical investigations. Even though technical and empirical investigations may seem similar, there is a difference between the two. Empirical investigations focus on the people who configure, use, or are otherwise affected by the technology. In other words, empirical investigations focus on direct and indirect stakeholders, while technical investigations' main focus is on the technology itself *(Friedman et al., 2002)*.

2.1.2. VALUES HIERARCHY

Value Sensitive Design emphasizes on the importance of translating abstract values into design requirements; however, VSD methodology does not clarify how this translation can actually be made in practice. Few research has been done on translation of values into design requirements, and in this contribution, van de Poel introduces the notion of Values Hierarchy *(van de Poel, 2013),* components of which are arranged from general values to specific design requirements.

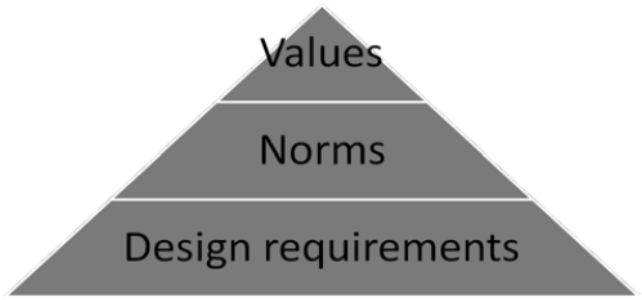


Figure _4: Values Hierarchy

(van de Poel, 2013)

As Figure_4 demonstrates, the upper layer of the Values Hierarchy consists of abstract values, the most concrete layer consists of specific design requirements, and norms represent an intermediate layer in the hierarchy *(van de Poel, 2013)*.

Two relations hold values hierarchy together: (1) *specification* – a relation by which higher level elements are translated into lower level elements (top-

down approach - abstract values are translated into more concrete set of norms, which in turn are translated into more specific design requirements) and (2) *for the sake of* – a relation by which design requirements are connected to underlying norms and values (bottom-up approach) *(van de Poel, 2013)*.

In "Translating Values into Design Requirements", van de Poel considers the design of chicken husbandry systems, as an example of using Values Hierarchy for translating values into design requirements. Chicken husbandry system makes it possible to produce eggs in an economically efficient and factory-like way. Important values in the design of battery cages have been identified using Value Sensitive Design's integrative tripartite methodology (conceptual, empirical, and technical investigations). General and abstract values then have been translated into more concrete set of norms and these norms have been translated into more concrete design requirements *(van de Poel, 2013)*. Figure_5 below demonstrates the partial values hierarchy for the design of a specific type of chicken husbandry systems, however, since the process of translating abstract values into more concrete set of norms is outside of the scope of this research, above discussed approaches to values hierarchy and translation of values into concrete set of norms will not be further discussed here.

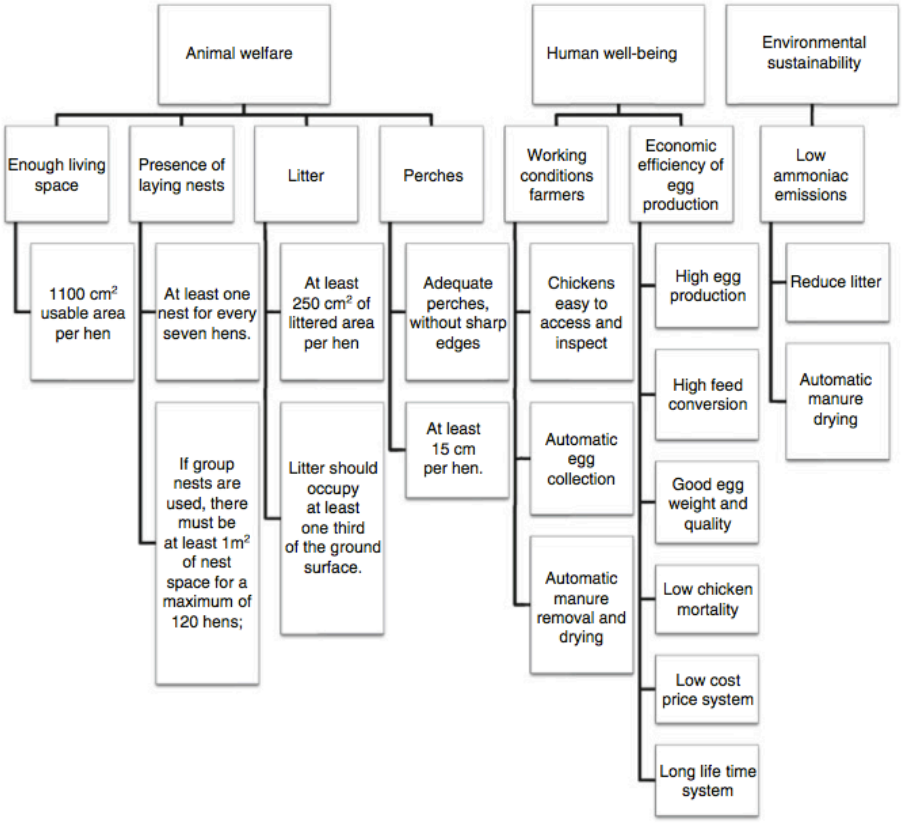


Figure _5: A Partial Values Hierarchy for the Design of Chicken Husbandry Systems *(the design requirements for animal welfare are based on EU Council Directive 1999/74/EC).*

(van de Poel, 2013)

While discussing the VSD approach, it was revealed that Value Sensitive Design focuses more on the process of translating abstract values into a concrete set of norms, and focuses less on the process of translating norms into design requirements. Even though the VSD approach does not provide guidelines on how to incorporate norms into a design process, it emphasizes on the importance of involving stakeholders in a design process. This feature of Value Sensitive Design approach will later be used as the motivation for developing the model for incorporating a concrete set of norms into a software design process. As the VSD approach does not make a norm implementation process clear, perhaps, the developed model will contribute to the theory of Value Sensitive Design. Before that, however, this chapter moves on to discuss the theory on Critical Design in the following section.

2.2. CRITICAL DESIGN

In the introduction of the Theoretical Background it was mentioned that Critical Design focuses on questioning and critically reflecting on norms throughout a design process. Questioning norms is outside the scope of this research because the focus of this research is on a set of norms that is concrete and ready to be incorporated into a software design. So why is the theory on Critical Design relevant to this research? Because Value Sensitive Design emphasizes on the importance of involving stakeholders in a technology design process but VSD does not make it clear how this should be done. As Critical Design focuses on not taking norms for granted, it involves bringing stakeholders to a discussion together with designers, in order to reflect critically on norms, therefore it is interesting how the idea behind Critical Design can be helpful for involving stakeholders in a process of software design.

Critical Design originates from and is influenced by Critical Theory. Critical Theory offers concepts and research on how to understand and reason about norms. Despite the fact that Critical Theory also offers concepts and research on how norms are incorporated in artifacts and design in society, it says nothing about how to create artifacts that in themselves question and challenge norms and normative understandings, which is Critical Design's main goal *(Faber et al.)*.

Over the past two decades there has been an increasing interest in design in Human-Computer Interaction (HCI) and how technology can improve the current state of human existence. Emergence of Critical Design is a highly promising development in this area *(Bardzell et al., 2012),* because by reflecting critically on values and norms embedded in technology, Critical Design gives designers an ability to instill critical thought, which, in turn, gives designers an opportunity to broaden their perspectives and to widen the range of possible choices and alternative futures in a design process *(Faber et al., Bardzell, J. and Bardzell, S., 2013)*.

The concept of Critical Design was developed in the field of interior and interaction design by designers Dunne and Raby *(Dunne and Raby, 2001)* and they described the notion of critical design as a possible way to get consumers to become more critical related to their everyday life and consumption of artifacts *(Faber et al.)*. In "Hertzian Tales: Electronic Products, Aesthetic Experience, and Critical Design" Dunne

defines critical design as an approach to "producing conceptual electronic products that encourage complex and meaningful reflection on inhabitation of a ubiquitous, dematerializing, and intelligent environment: a form of social research to integrate critical aesthetic experience with everyday life […] I hope in my approach I have retained the popular appeal of industrial design while using it to seduce the viewer into the world of ideas rather than objects. Industrial design locates its object in a mental space concerned with identity, desire, and fantasy and shaped by media […] Again, I hope this remains intact but is subverted to challenge the aesthetic values of both consumers and designers" *(Dunne, 2006)*.

Critical Design, as an area of research, has been used in Human-Computer Interaction (HCI) as a way to get both users and designers to reflect on their own practice and to critically question norms and values that are reproduced in design process *(Bardzell, J. and Bardzell, S., 2013)*. The purpose of Critical Design is to get users to understand that their needs are constructed by society *(Bardzell, J. and Bardzell, S., 2013)*. Besides, Critical Design challenges designers and users to envision and to demand design products that reflect "a more challenging view of human needs and experience" *(Bardzell, J. and Bardzell, S., 2013)*.

Critical Design focuses on inscribing alternative values in designs, and by doing so, it cultivates critical attitudes among users and designers alike, creating demand for and supporting the professional emergence of alternative design features *(Bardzell, J. and Bardzell, S., 2013)*.

Different researchers have been using Critical Design Theory in design/production process and they have created practical examples that show how critical design can be best deployed in practice. For example, Faber et al. used critical design perspectives in a video production project, to develop a pair of physical glasses that they chose to name as goggles. During the design process of goggles, Faber and colleagues developed three different questions representing: *(1) defamiliarization, (2) provocation,* and *(3) evaluation* of the goggles.

Questions associated with the *defamiliarization* of goggles were formulated in order "to help the designer when applying them [goggles] in specific design decisions to be able to reflect on what basis decisions are made" *(Faber et al.)*. By asking questions, such as: how has the specific decision resided? which circumstances does the decision rest upon?, the basic idea of defamiliarization was to distance designer from decisions in order to see how decisions are based on values and norms. Prerequisite for defamiliarization to work was to choose a specific design decision applied to goggles *(Faber et al.)*. Questions associated with *provocation* of goggles were used to create reflections about the design, to open up for alternative future directions, and to discover new ways to edit on both on a creative and norm-critical level *(Faber et al.)*. Questions associated with *evaluation* of goggles were actually based on Value Sensitive Design methods, and were used in order to better justify the decisions taken before they were performed. The idea of evaluation was to identify conflicts of values and norms (and their consequences) that could occur between different stakeholders, and how to discuss these conflicts *(Faber et al.)*.

During the design process of goggles, Faber et al. noticed that discussions that emerged out of the critical questions asked during the design, led to further understanding and new ideas by examining the different types of values and norms inherent in the design process *(Faber et al.)*.

In "Critical Design and Critical Theory: The Challenge of Designing for provocation" Bardzell and colleagues argue that in order to visualize the normative notion of design and design processes, designers

and other actors involved in the design process should start by identifying and questioning the norms during the design process itself. In this way critical design becomes prominent and useful in design processes *(Bardzell et al., 2012)*.

Even though questioning norms is outside the scope of this research, Critical Design can be helpful for involving stakeholders in a software design process. By using the idea behind Critical Design, stakeholders can be brought to a discussion together with software developers and designers, but instead of questioning norms, they will have critical reflection on the way norms are translated into software design requirements. How and why Critical Design can contribute to a model for incorporating norms with a software design will be further discussed in Chapter 3.

2.3. APPROACHES TO A SOFTWARE DEVELOPMENT PROCESS

In order to incorporate a concrete set of norms into a software design, the model developed within this research has to be in harmony with a software development process. Software development process, as defined by a software engineer Humphrey, "refers to the set of tools, methods, and practices used to produce a software product" *(Humphrey, 1989)*.

Over the past 40 years software development processes have received a great attention. A large number of software development practices and models have been proposed *(MacCormack et al., 2003)* with the primary functions to determine the order of the stages/phases involved in software development process and to establish the transition criteria for progressing from one stage/phase to the next one *(Boehm, 1988)*.

To see which software development approach can be harmonized with the model that this research aims to develop, number of different approaches will be discussed as follows:
Section 2.3.1 will summarize one of the traditional software development methodologies – the waterfall model, as it is the most commonly used approach to a software development process. Then Agile Software Development Methods will be discussed, because the agile methods focus on the collaboration between software developers and stakeholders. The Agile methods also focus on being flexible with changing design requirements (whenever necessary) throughout a software design process. As previously discussed, the model developed within this research aims to use Critical Design for critically reflecting on norm translation process. This means that during a design process, depending on stakeholders' reflection, there might be a need to make changes in software design requirements. Therefore the Agile Software Development Methods are of particular interest for this thesis.

In section 2.3.3 Agent-Oriented Software Engineering (AOSE) will be discussed, because by developing agent based systems, AOSE claims to be an effective approach to a large-scale software engineering problem. Figuring out how effective AOSE really is falls outside the scope of this research, however, it is

interesting to see if this approach to a software development process can be in harmony with the model developed within this research.

Discussion about Service-Oriented Architecture (SOA) will follow afterwards, in section 2.3.4. This approach to a software development process focuses on organizing a portfolio of software applications into an interconnected set of services. SOA claims to be able to adapt quickly to different external environments. It is also claimed that using the SOA approach to a software development process lowers software development and management costs. It is interesting to see if Service-Oriented Architecture creates a room for Critical Design, or if other characteristics of this approach can contribute to norm implementation.

2.3.1. WATERFALL MODEL

Early attempts at standardizing a formal software development model culminated in what is now known as the waterfall model *(MacCormack et al., 2003)*. Formally, the waterfall model of software development was first described in the article: "Managing the Development of Large Software Systems" by W. Royce, in 1970. Though W. Royce did not use the term "waterfall" in the article, he presented the model as a sequential design process in which progress is seen as flowing steadily downwards through the following phases:

1. Software requirements specification (resulting in the product requirements document);
2. Software design (resulting in the software architecture);
3. Implementation (or coding) (resulting in the actual software product);
4. Integration;
5. Testing and debugging;
6. Installation;
7. Software Maintenance.

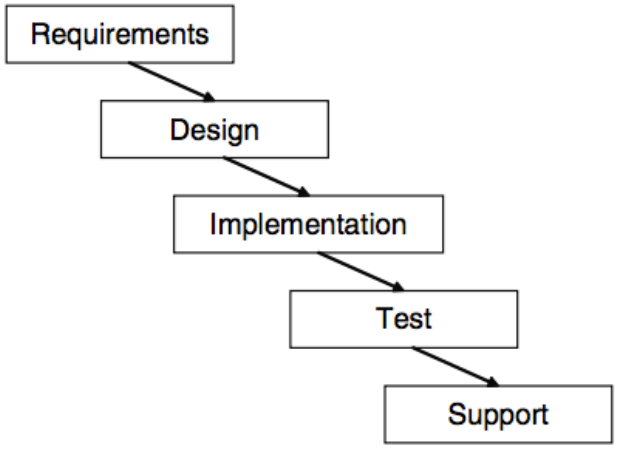


Figure_6: Waterfall Model

During the software development process, moving from one phase of waterfall model to another phase should happen only when the preceding phase is reviewed and verified *(Royce, 1970)*.
The waterfall model "emphasizes the need to develop comprehensive and detailed specifications up front and thereafter to execute efficiently according to these specifications" *(MacCormack et al., 2003)*. In other words, before beginning software development, the waterfall model stresses the need to complete software specifications. The milestones of the waterfall model (the phases listed above) let different companies, government organizations, and/or standard groups establish a set of interlocking regulations, specifications, and standards that cover a full set of software project needs *(Boehm, 1996)*.
Besides waterfall model, other traditional software development methods include Spiral model *(Boehm, 1988)* and Win-Win Spiral model *(Boehm, 1996)*. Both of those models have been evolved based on various refinements of the waterfall model.

2.3.2. AGILE SOFTWARE DEVELOPMENT METHODS

Agile software development methods focus on two concepts: (1) unforgiving honesty of working code, and (2) the effectiveness of people working together with goodwill. In February 2011, the Manifesto for Agile Software Development was signed *(Highsmith and Cockburn, 2001)*. Principles behind the Agile Manifesto include *(from (Lindstrom and Jeffries, 2004))*:

- To satisfy customers through early and continuous delivery of valuable software;
- To welcome changing requirements (even late in software development process) and to process change for the customer's competitive advantage;
- To deliver working software frequently (with a preference to shorter time scales);
- To promote sustainable development;
- To focus on technical excellence and good design, hence to enhance agility;
- To emerge architectures, requirements, and designs from self-organized teams;

Agile methods focus on customer collaboration, which means that all parties involved in the software development process (sponsors, customers, users, developers) are on the same team during the software development process *(Highsmith and Cockburn, 2001)*.

According to Agile Manifesto*, Agile software development methods focus on four core values:

1. Individuals and interactions over processes and tools;
2. Working software over comprehensive documentation;
3. Customer collaboration over contract negotiation;
4. Responding to change over following a plan.

* www.agilemanifesto.org

[2] Mentioned articles can be found under the *"Law on Disposal of the Dead"* and *"Law on Organ*

As Williams and Cockburn state, Agile development is 'about feedback and change'; Agile methods are developed to 'embrace, rather than reject, higher rates of change' *(Williams and Cockburn, 2003).*

Extreme Programming

Extreme Programming, one of the most widely used Agile methodologies, is a software development approach that advocates rapid iterations, rigorously tested code and working closely with end users *(Beck and Andres, 2004, Beck and Fowler, 2001)*. Extreme Programming is based on the following four principles: simplicity, communication, feedback, and courage. In Extreme Programming, every contributor to the software development process is a member of the whole team. Central to the team is a customer (one or more business representatives who sit with the software development team and work with them daily). By having customers work with software developers onsite, extreme programming removes communication barriers between customers and developers *(Cao et al., 2004)*. Teams use a simple form of planning and tracking to decide what steps to take next in the process of developing software. Software is produced in a series of small, fully integrated releases that pass all the tests that customer has defined *(Lindstrom and Jeffries, 2004).*

Extreme programming does not specify management practices. Instead, extreme programming simplifies management by empowering the customer and software developer to make most of the decisions regarding the software that is being developed *(Lindstrom and Jeffries, 2004).*

There are twelve core practices that define extreme programming. The practices can be described as a cycle of activities *(Lindstrom and Jeffries, 2004):*

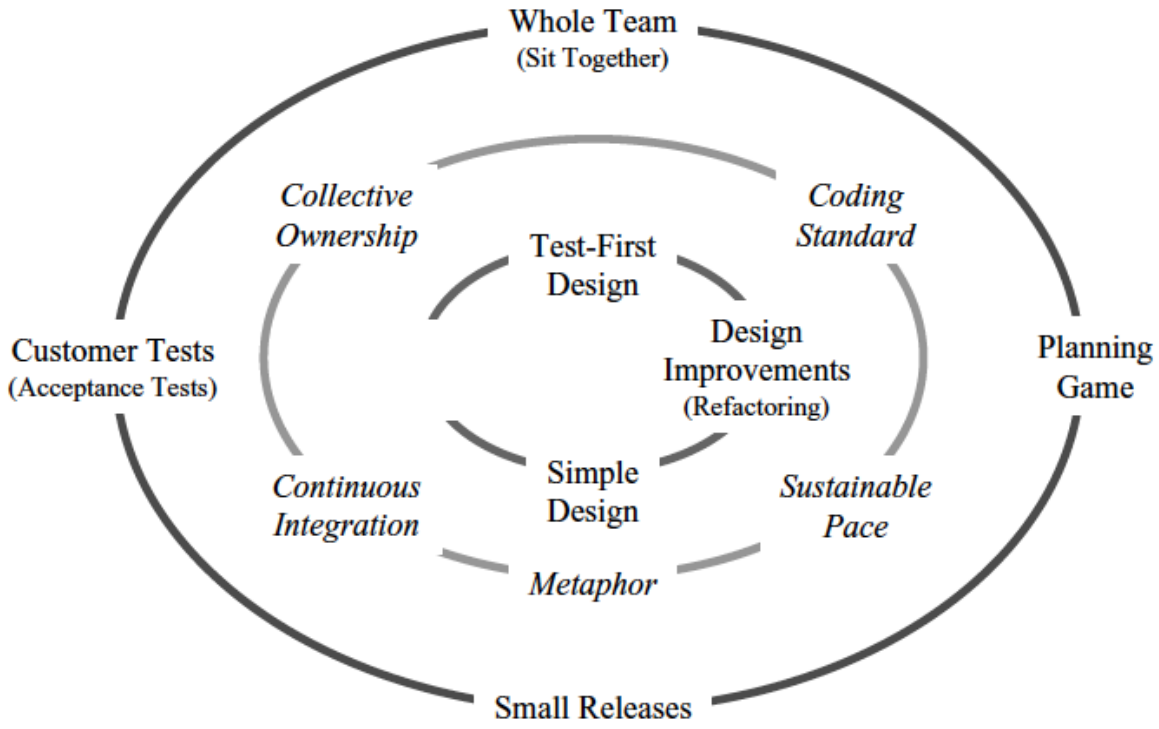


Figure_7: Practices in Extreme Programming

(Lindstrom and Jeffries, 2004)

The inner cycle (Test-First Design, Design Improvements, and Simple Design) is a tight cycle of the programmers/software developers. The middle cycle on Figure_7 shows practices (Coding Standard, Sustainable Pace, Metaphor, Continuous Integration, Collective Ownership) that help the team communicate and coordinate the delivery of quality software. The outer loop (Whole Team, Planning Game, Small Releases, Customer Tests) describes the planning cycle that occurs between customers and software developers/programmers *(Lindstrom and Jeffries, 2004)*.

By being explicit in its principles (simplicity, communication, feedback, and courage) and practices (Figure_7), extreme programming gives guidance on (1) what to do (the practices) during the software development process, and (2) how to react when the practices are not sufficient enough. Out of several Agile Methods for software development, extreme programming is the only method that combines software development practices with principles used to guide the team during software development process. While being specific on practices, other Agile Methods do not articulate a set of such principles *(Lindstrom and Jeffries, 2004)*.

Besides extreme programming, the Agile Software Development Methods also include: scrum, feature-driven development, crystal, lean software development, and dynamic software development methods (DSDM). The Agile Methods for software development process vary, but they share the common characteristics, such as: iterative development, working in frequent consultation with customers, and having small and frequent releases *(Cao et al., 2004)*. By emphasizing on "rationalized, engineering-based approach" *(Dybå, 2000, Nerur et al., 2005)*, the Agile Software Development methods can be seen as a reaction to plan-based (traditional) software development methods *(Dybå and Dingsøyr, 2008)*. While traditional software development methods focus on extensive planning, and codified processes *(Boehm, 2002)*, the Agile Methods rely on "people and their creativity rather than on processes" *(Dybå, 2000, Nerur et al., 2005)*.

2.3.3. AGENT-ORIENTED SOFTWARE ENGINEERING (AOSE)

Multiagent systems (MASs) have recently emerged as a technology to face the complexity of a variety of today's ICT scenarios. MASs represent a general-purpose paradigm for software development *(Jennings, 2001, Zambonelli and Parunak, 2004)*. Agent-based computing promotes designing and developing applications in terms of agents (autonomous software entities), situated in an environment. Those agents can flexibly achieve their goals by interacting with one another in terms of high-level protocols and languages *(Zambonelli and Omicini, 2004)*.
Agent-Oriented Software Engineering (AOSE) is a way of approaching large-scale software engineering problem and developing agent based systems. In AOSE applications are written as agents. As Wooldridge and Jennings define, an agent is "an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives" *(Wooldridge and Jennings, 1995)*. In this definition, flexible action refers to the reactive response in timely fashion to the environmental change and proactive act in anticipation of future goals, while autonomous action refers to the control over internal state and over own behavior *(Mahar and Bhatia, 2012)*.

The main purpose of Agent-Oriented Software Engineering is to create methodologies/tools for enabling inexpensive development and maintenance of agent-based software. In order to measure the various attributes of AOSE, agents, as software entities, are required to have the following characteristics:

- *Autonomy* – agents have their own will;
- *Sociality* – agents are able to interact with each other;
- *Reactivity* – agents respond to stimulus;
- *Proactivity* – agents initiate initiative;
- *Mobility* – agents can move around;
- *Veracity* – agents are truthful;

- *Benevolence* – agents do what they are told to do;
- *Rationality* – agents perform an optimal manner to achieve goals

(Jennings and Wooldridge, 1998).

Features and concepts of Agent-Oriented Software Engineering differentiate AOSE from Object-Oriented Software Engineering (OOSE). From the traditional object-oriented perspective on software engineering, an object, unlike an agent, is neither autonomous nor proactive (its internal activity can be solicited only by service requests coming from an external thread of control). In object applications there is no explicit modeling of external environment and everything is modeled in terms of objects. Interactions between objects are simply an expression of inter-dependencies, and concepts such as society or roles do not make any sense *(Zambonelli and Omicini, 2004, Bass, 2007, Shaw et al., 1995, Shaw and Garlan, 1996)*. On the other hand, from the agent-oriented perspective on software engineering, agents focus on the autonomy of oneself and the interaction among different agents. Agents depend upon their environment and counterparts in performing their tasks and goals through collaboration and interaction *(Mahar and Bhatia, 2012).*

As represented on Figure_8, Agent-Oriented Software Engineering has three major phases:

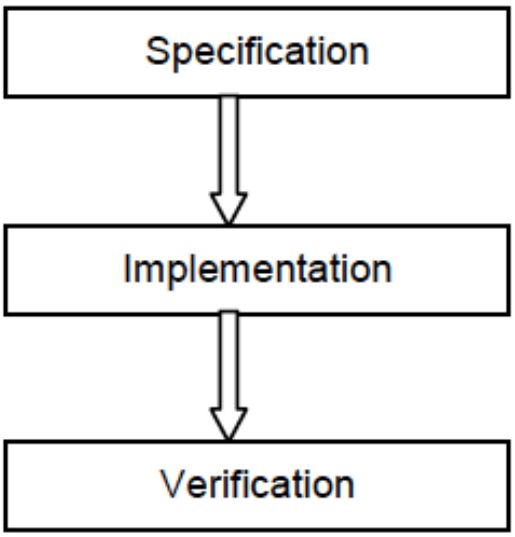


Figure_8: Agent-Oriented Software Engineering Phases

(Mahar and Bhatia, 2012)

(1) Specification – where it is defined what kinds of properties must an agent be capable of representing.
(2) Implementation – refers to moving from abstract specification to concrete computational system. In other words, a system is implemented with respect to specification done beforehand.
(3) Verification – the process of verifying that the developed system is correct with respect to the original specification *(Mahar and Bhatia, 2012).*

Two of the Agent-Oriented Software Engineering methodologies, Tropos and Gaia, are discussed below:

Tropos

The Tropos Methodology is intended to support analysis and design activities in the software development process, from application domain analysis down to the system implementation *(Bresciani et al., 2004).* Five main development phases of the Tropos Methodology are:
Early requirements – concerned with the understanding of a problem by studying an existing organizational setting;
Late requirements – where the system-to-be is described within its operational environment, along with relevant functions and qualities;
Architectural design – defines system's global architecture in terms of sub-systems, interconnected through data and control flows;
Detailed design – concerned with defining each architectural component in further detail (in terms of inputs, outputs, control, and other relevant information);
Implementation – where the actual implementation of the system is carried out *(Mylopoulos et al., 2002).*

Following modeling activities contribute to the acquisition of a first early requirement model, to its refinement and to its evolution into subsequent models:

- *Actor modeling* – consists of identifying and analyzing actors of the environment, as well as system's actors and agents.
- *Dependency modeling* – consists of identifying actors, which depend on one another for goals to be achieved.
- *Goal modeling* – covers the analysis of actor goals, conducted from the point of view of the actor.
- *Plan modeling* – an analysis technique complementary to goal modeling.
- *Capability modeling* – starts at the end of the architectural design and further specifies each agent's capabilities, which then are coded during the implementation phase *(Bresciani et al., 2004).*

By covering very early phases of requirement analysis (as discussed above), Tropos aims for a deeper understanding of the environment where the software must operate, and of the kind of interactions that should occur between software and human agents (evaluating agent-oriented software). However, the Tropos Methodology is not intended for any type of software. For example, for system software or embedded software, the operating environment of the system-to-be is an engineering artifact with no identifiable stakeholders, hence using the Tropos Methodology in this case would not be appropriate. The Tropos Methodology is also not suitable for sophisticated software agents, which require advanced reasoning mechanisms for plans, goals and negotiations *(Bresciani et al., 2004).*

Gaia

The Gaia Methodology is intended to allow an analyst to go systematically from a statement of requirements to a design that is sufficiently detailed and can be implemented directly. When applying Gaia, the analyst moves from abstract concepts to concrete concepts. "Each successive move introduces greater implementation bias, and shrinks the space of possible systems that could be implemented to satisfy the original requirements statement" *(Wooldridge et al., 2000)*.

The Gaia process starts with the *analysis phase*, the aim of which is to collect and organize the specification, which is the basis for the design of the computational organization. Output of the analysis phase consists of: an environmental model, a preliminary roles model, a preliminary interactions model, and a set of organizational rules. This output is then exploited by the *design phase (Zambonelli et al., 2003)*. Analysis and design phases can be thought of as a process of developing detailed models of the system-to-be *(Wooldridge et al., 2000)*.

Design phase is decomposed into an *architectural design* and a *detailed design* phases. The architectural design phase includes:

- *Definition of the system's organizational structure;*
- *Completion of the preliminary role and interaction models.*

The detailed design phase covers:

- *Definition of the agent model;*
- *Definition of the services model;*

(Zambonelli et al., 2003).

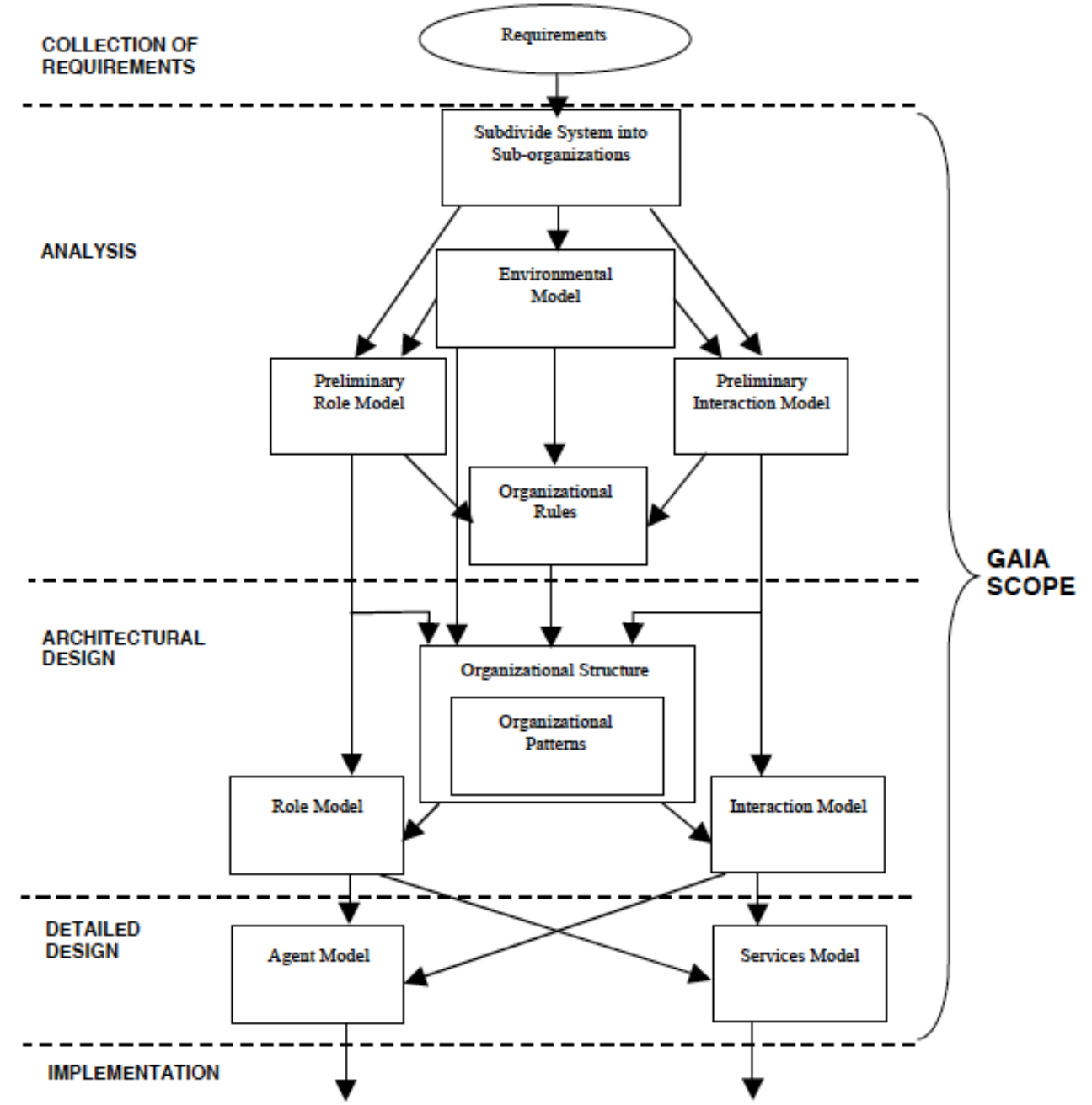


Figure_9: Models of the Gaia Methodology and their Relations in the Gaia Process

(Zambonelli et al., 2003)

The main concepts of Gaia can be divided into two categories: *(1) abstract entities* and *(2) concrete entities*. Abstract entities are used during analysis phase to conceptualize the system, but those entities do not necessarily have any direct realization within the system. Concrete entities, in contrast, are those entities used within the design process. Concrete entities have direct counterparts in the run-time system *(Wooldridge et al., 2000)*.

As discussed above, The Gaia Methodology proposes various modeling techniques, however, it does not directly deal with those techniques and does not commit to specific techniques for modeling. Also, the outcome of the Gaia process is a detailed but technology-neutral specification that should be implemented using an appropriate agent-programming framework.

Basic assumption in the Gaia Methodology is that the analysis phase can rely on the output produced by an early requirements engineering phase and this methodology of Agent-Oriented Software Engineering does not explicitly deal with the activities of early requirements engineering (such as: analyzing characteristics to be exhibited and goals to be achieved by the system-to-be, as they emerge from the needs of the stakeholders and from the specific operational environment) *(Zambonelli et al., 2003)*.

2.3.4. SERVICE-ORIENTED ARCHITECTURE (SOA)

Service-Oriented Architecture (SOA) is an architectural approach of organizing a portfolio of software applications into an interconnected set of services. Services are computational elements that support rapid, low-cost composition of distributed applications. Services provide information distributor for wide range of computing devices and software platforms *(Papazoglou, 2003)*.

With its architectural style, SOA defines an interaction model between three primary parties:
(1) Service provider – publishes a service description and provides the implementation for the service;
(2) Service consumer (or service client) – finds the service description in a service registry and binds and invokes the service;
(3) Service broker – provides and maintains the service registry *(Arsanjani, 2004)*.

A service may exhibit characteristics of service provider, as well as of service client. Usually, a service is a business function implemented in software, and wrapped with a formal documented interface that is well known. It is also well known where the service can be found (not only by actors who designed the service, but also by stakeholders who want to access and use it). Above-mentioned formal documented interface is an interface that provides the mechanism by which service communicates with applications and other services. Interfaces that client of the service expects, and interfaces that must be provided by the environment into which the service is composed, must be explicitly described by service specification *(Papazoglou, 2003)*.

The process of Service-Oriented Architecture involves three general steps: *(1) service identification, (2) component specification, and (3) service realization* (Figure_9).

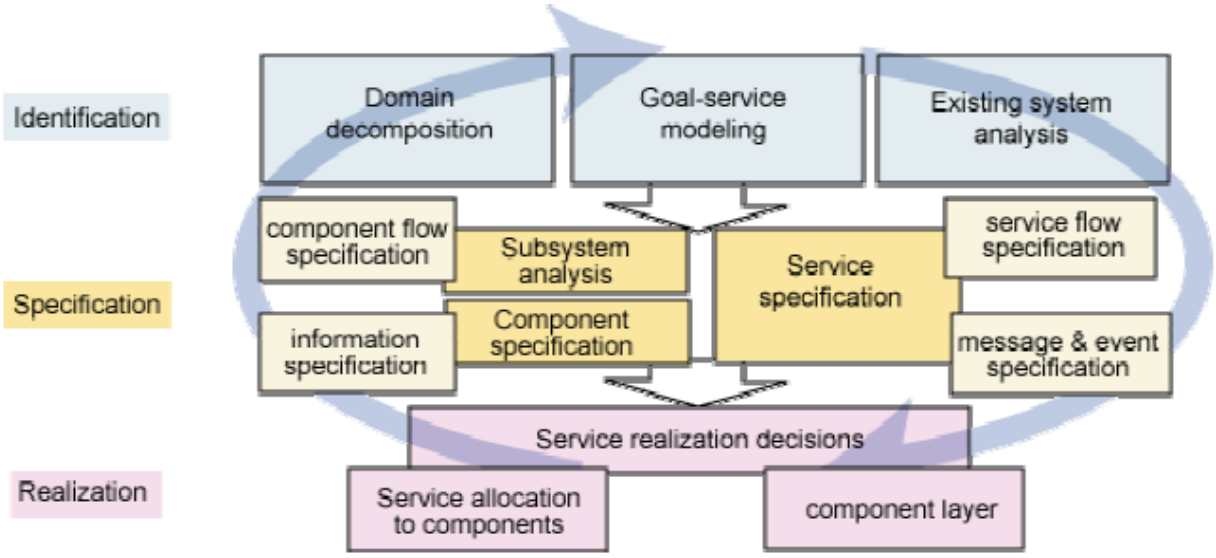


Figure_10: Process of Service-Oriented Architecture

(Arsanjani, 2004)

Service specification consists of:

- Combination of top-down, bottom-up, and middle-out techniques of domain decomposition;
- Existing asset analysis,
- Goal-service modeling

In the top-down view, a blueprint of business use cases provides the specification for business services. In the bottom-up view, or in existing system analysis, existing systems are analyzed and selected for providing solutions to the implementation of underlying service functionality that supports the business process. The middle-out view consists of goal service modeling to validate other services not captured by other service identification approaches (either top-down or bottom-up). Goal-service modeling ties services to goals and sub-goals, key performance indicators, and metrics *(Arsanjani, 2004)*.

Data, rules, services, configurable profile, and variations are details of the components that implement services. Those details of components are specified under *component specification*. Messaging and events specifications, as well as management definition, also occur at this step *(Arsanjani, 2004)*.

Under *service realization*, decisions are made as to which legacy system module will be used to realize a given service and which services will be built from the ground-up *(Arsanjani, 2004).*

Service-oriented architecture has three architectural perspectives:
(1) Application architecture – consumes services from one or more service providers and integrates them into the business processes;
(2) Service architecture – provides a bridge between implementations and the consuming applications;
(3) Component architecture – describes environments supporting the implemented applications, business objects and their implementations *(Sprott and Wilkes, 2004).*

For a *service consumer*, the SOA process must be organized in a way that only the service interface matters, and there must be no dependence upon knowledge of the service implementation. Service consumers are focused on *application architecture* and services used. They are not focused on the detail of the *component architecture*. Therefore, developers only need to know where the service is, what it does, and how it can be used. *Service providers*, on the other hand, have a different set of concerns. The providers are focused on the *component architecture* and *service architecture*, but not on the *application architecture*. Providers are not interested in every detail of the *consuming application*. Service providers need to develop and deliver a service that can be used by service consumers in a completely separate process. Focus of attention for service providers is on the interface that defines the required parameters of the service and the nature of the result *(Channabasavaiah et al., 2004, Sprott and Wilkes, 2004).*

In order to create a common understanding about services, priorities, responsibilities, etc., formal negotiated agreement exists between a service provider and a service consumer. This formal agreement, or contract, is called a Service Level Agreement (SLA) *(Nan et al., 2006).*

A Service Level Agreement builds a business relationship between service provider and service consumer via making promises at a specified service level and giving the compensation for failing to fulfill the promise *(Zhang et al., 2003).* In other words, Service Level Agreement establishes an actual link between requester (consumer) and provider of a service. The aim of an SLA is to guarantee the service consumer functional and non-functional aspects of the required service. Functional aspects of the required service may include the scope of the service, optional services, etc., while non-functional aspects of the required service might be service level objectives *(Raibulet and Massarelli, 2008).*

The service provider and the service consumer are *signatory parties* to an SLA. They are responsible for all obligations, and the ultimate beneficiary of obligations. In some cases, signatory parties can sponsor third parties in order to support the enactment of the contracts. Those third parties are referred to as *supporting parties (Ludwig et al., 2003).*

A Service Level Agreement comprises the following parts:
(1) Parties – this part describes parties involved in the management of a service. Eventhough together with *signatory parties*, *supporting parties* are also included in this part of contract, the relationship of a supporting party to its sponsors *(signatory party)* is not within the scope of the SLA.
(2) Service Definitions – describes the service properties on which obligations are defined.
(3) Obligations – define the service level that is guaranteed with respect to the SLA parameters defined in the *Service Definition* section. Also, the *Obligations* part represents promises to perform actions under particular conditions *(Ludwig et al., 2003).*

While an SLA contains one *Parties* and one *Obligations* section, it may contain multiple *Service Definitions*.

2.4. CRITICAL REFLECTION ON THEORETICAL BACKGROUND

Now that the theories relevant to this research have been identified and discussed in the previous sections of this chapter, this section will focus on reflecting on those theories in order to analyze how those theories can (or perhaps, can not) contribute to the objective of this research and how the ideas behind the relevant theories can shape the development of the model for integrating the norms with a software design/development process.

In chapter 2 we discussed theories on Value Sensitive Design, Critical Design, and various approaches to a software development process. From this discussion we can conclude the following:

- Value Sensitive Design (VSD) focuses more on translating abstract values into a concrete set of norms, and it focuses less on the process of translating norms into the design requirements;
- The main idea behind Critical Design is to critically question the values and norms that are incorporated into a product;
- Agent-Oriented Software Engineering (AOSE) – focusing on the autonomous software entities (agents), and Service-Oriented Architecture (SOA) – focusing on the services as the computational elements, have the same top-down approach to the software development process, as does the traditional Waterfall Model. The Agile Software Development Methods, on the other hand, focus on the flexibility of the processes and phases involved in the software design/development process.

Theory on the Value Sensitive Design tries to explain how the VSD approach can be used to design technology that accounts for human values. Conceptual, empirical, and technical investigations within Value Sensitive Design focus on value conceptualization, on the effects of values on direct and indirect stakeholders, and how technology supports or hinders values. With those investigations VSD claims to be able to translate abstract values into a concrete set of norms, however, it is still vague how the translation is actually done (or should be done) in practice. Besides, while focusing on value translations, VSD shows less focus on the process of translating norms into functional design requirements, and says nothing about questioning norms throughout the design process.

On the other hand, the purpose of Critical Design is not to satisfy the users' needs, but rather "to get them to understand that their needs are constructed by society" *(Bardzell, J. and Bardzell, S., 2013)*. Even though literature on Critical Design does not provide a specific approach with which Critical Design can be deployed in practice, different practical examples show that the main idea behind Critical Design is to question and critically reflect on the values and norms that are incorporated in the product, throughout the

design process. Critical Design "cannot be reduced to a simple recipe" and its success depends on the experience and expert judgment of the designers (and users) who undertake it *(Faber et al.)*. As Dunne and Raby define, critical design "is simply about not taking things for granted, to question and look beneath the surface" *(Frayling, 1993)*. Questioning the values and norms throughout the design process is outside the scope of this research, however, it is interesting how Critical Design can contribute to the process of integrating the concrete set of norms into a software development process, perhaps by critically reflecting on the design requirements translated from a norm.

Reflecting critically on the design requirements translated from the concrete set of norms would require the involvement of software designers/developers, as well as of the stakeholders interested in and affected by the developed software product. So, the process of translating norms into the software design requirements and integrating those norms into the software design process, involving designers' and stakeholders' critical reflection on the design requirements, requires flexibility. This is because after asking critical questions and after reflecting critically on the design requirements, there is a possibility that those design requirements will not be acceptable for all the stakeholders, therefore, it will be necessary to find the alternative translation(s) for the norm. If during the software development process, there is not enough flexibility for going back to the previous phase in order to make changes in the design requirements, how will it be possible to implement the alternative norm translations? Approach to the software design process has to be flexible enough to give to the Critical Design for reflecting critically on the design requirements translated from a norm. Out of several approaches to the software design process, this research finds the Agile Software Development Methods as the most suitable for the norm implementation process, because traditional software development approaches (Waterfall Model, for example), are based on sequential design process, in which software is developed in phases flowing steadily downwards, which means that moving from one phase to another is only possible when the previous phase is completed. This approach limits the flexibility necessary for implementing the norm translation process, as well as the process of critically reflecting on norms. Even though Agent-Oriented Software Engineering approach to software design process focuses on reactive response to environmental changes, the process of developing and designing software involves phases which again, as in case of Waterfall Model, flow steadily downwards. Other limitations of AOSE methods (Tropos and Gaia, in particular) where discussed earlier in this chapter. Service-Oriented Architecture also involves three downward flowing steps, which again, makes it possible to move from one phase to another, only when the preceding phase is completed and verified. For creating a common understanding about provided services, priorities, responsibilities between the service provider and the service consumer, Service-Level Agreements are used. However, SLAs say nothing about how norms should be incorporated into a design process, or how norms are translated into functional design requirements. Compared to software development methods discussed in this chapter, Agile methods are more flexible, as they focus on the effectiveness of people working together, and most importantly, Agile methods are open to changing design requirements if and whenever necessary throughout the software design process, which creates a room for flexibility necessary for getting critical reflection from designers/developers and stakeholders on norm translation, during a software design process.

After discussing and analyzing the theories relevant to this research, we can assume that the three sub-questions for the main research question (formulated in the previous chapter) are answered. This section summarized how theories on Value Sensitive Design and Critical Design can contribute to the development of the model for integrating a concrete set of norms into a software design. It was also summarized which approach to a software development process can best contribute to this research. Now

we can use the answers to those sub-questions to develop a model for integrating a concrete set of norms into a software design process. The development of this model is discussed in the following chapter.

3. THE MODEL DEVELOPMENT

The following chapter focuses on the model that was developed (within this research) for harmonizing norms with the software development process. In section 3.1 below, the conceptual model for this research is discussed, which is followed by demonstrating the actual model for integrating a concrete set of norms with a software development process in section 3.2 and the steps involved in the process, as suggested by the model, are discussed as well. The chapter ends with the concluding remarks about the model.

3.1. THE CONCEPTUAL MODEL

The objective of this research is to develop a model for integrating a concrete set of norms into a software design explicitly during the software development process. Based on the theoretical background and analyzed theories, first, the conceptual model was developed, in order to identify core concepts related to this research and the relationship between those concepts:

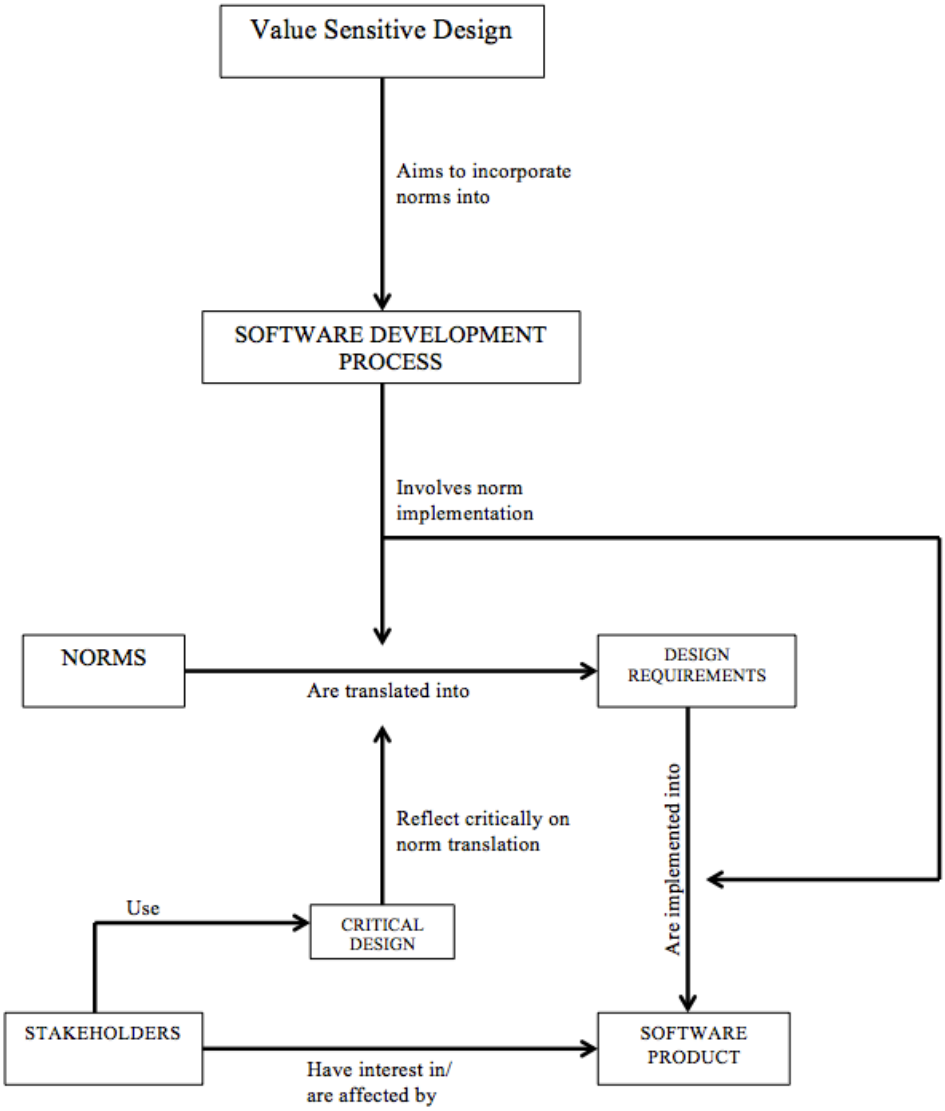


Figure_11: The Conceptual Model

As it was discussed during chapter 2 (Theoretical Background), Value Sensitive Design aims to incorporate values and norms into a technology design. While focusing on translating abstract values into a concrete set of norms, the VSD methodology shows less focus on the process of integrating norms into a design. However, Value Sensitive Design, as demonstrated on the conceptual model, is one of the core concepts of this research. The reason for this is that this thesis aims to contribute to the VSD methodology by recommending a model for incorporating a concrete set of norms into a software design. We already know from the Theoretical Background that the VSD methodology suggests involving stakeholders in the design process, because technology's use depends on the goals of its stakeholders. So, besides contributing to the VSD methodology, this thesis uses VSD's features as the motivation for the model development. In other words, the model developed within this research will focus on involving stakeholders in the software development process.
Then again, the VSD methodology does not make it clear how exactly stakeholders should be involved in the design process. This is why the conceptual model involves the Critical Design. To be more specific, this research uses the idea behind Critical Design in order to make sure that stakeholders are involved in a software development process.
The idea behind Critical Design is to continuously question norms before implementing them. As the objective of this research is to incorporate concrete set of norms into a software design, questioning norms would fall outside the scope. In other words, this research does not aim to make any changes in the concrete set of norms. In contrast, it aims to implement this concrete set of norms into a software product explicitly during the software design/development process. So what is the role of Critical Design in this scenario and how can it involve stakeholders throughout the software development process?

As demonstrated on the conceptual model, software development process involves translating norms into software design requirements, and then implementing those design requirements into a final software product. The role of Critical Design in this process is not to question norms, but to question the way those norms are translated into design requirements. So, Critical Design will be used to get a critical reflection on the norm translation. By using the concept of Critical Design, the model developed within this research will make sure that stakeholders interested in or affected by the developed software are involved in the software design process. Besides, having critical reflection on norm translation guarantees that the design requirements are not taken for granted, and that the norms as well as the goals of stakeholders are addressed throughout the software development process.

The conceptual model developed in this section summarizes how the theories identified in the Theoretical Background can contribute to the development of the model for integrating a concrete set of norms into a software design. This conceptual model can be seen as the backbone of the model discussed in the following section.

3.2. THE MODEL FOR INCORPORATING A CONCRETE SET OF NORMS WITH A SOFTWARE DESIGN PROCESS

Now that it has been explained in the previous section how the theories identified in the Theoretical Background relate to each other, and how they can contribute to the model that this research aims to develop, discussion can move on to the model for incorporating a concrete set of norms into the software design process. Figure_2 below demonstrates this model:

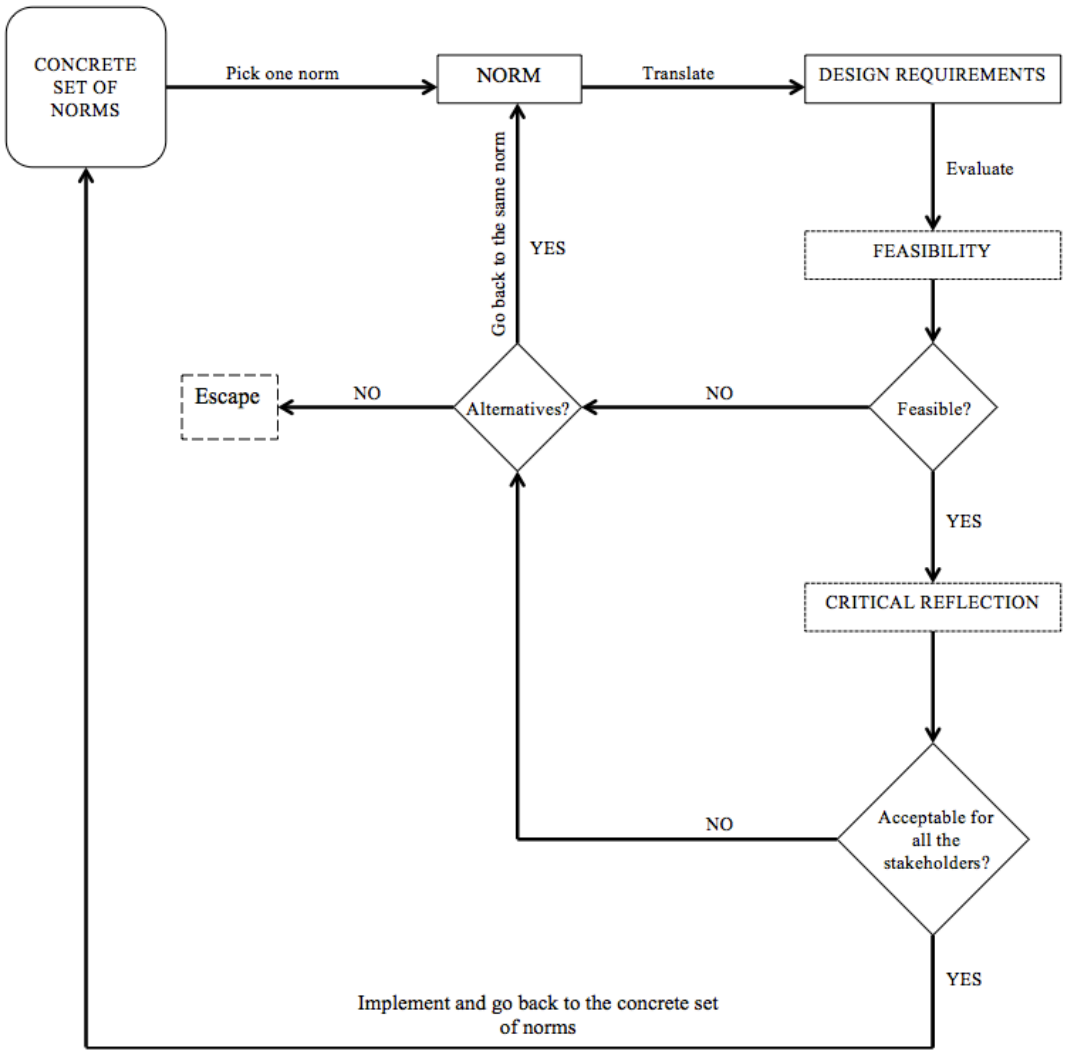


Figure_12: The Model for Incorporating a Concrete Set of Norms into a Software Design Process

The steps involved in the above-demonstrated model, are discussed as follows:

(1) Pick one norm from the concrete set of norms

As demonstrated on Figure_12 above, the model suggests to start the process of harmonizing norms with software development process by picking out one specific norm from the concrete set of norms. It is supposed that this concrete set of norms is already translated from the abstract values. However, the process of value translation, or which values were translated into norms, is outside the scope of this research, and therefore the model on Figure_12 focuses on the concrete set of norms at hand, and how this set of norms can be integrated with software development process. Each norm in the concrete set of norms is translated separately into design requirements. In the end, sets of design requirements (one set per each norm) can be implemented into software.

(2) Translate the norm into design requirements

Before implementing, the meaning of the norm has to be clarified by means of a formal representation. This representation has to make clear which actions and situations are accepted and which are not accepted. Such a representation of a norm helps us to capture the precise meaning of the norm and to reason about it *(Aldewereld, 2009)*. This aspect of the norm is traditionally captured by deontic logic *(Meyer and Wieringa, 1994)*. Deontic logic is the formal tool needed in the design and specification of normative systems, where normative systems are understood to be sets of agents (human or artificial) whose interactions "can be regarded as norm-governed" *(Carmo and Jones, 2002)*.

After picking one specific norm from the concrete set of norms, the next step, as recommended by the model, is to translate this norm into the functional software design requirements. This translation is done using deontic logic. Deontic logic is concerned with the logical analysis of such normative notions as obligation, permission, right and prohibition.

Although its origins lie in systematic legal and moral philosophy, deontic logic has attracted the interest of researchers in other areas, particularly computer science. How norms are translated into design requirements using deontic logic is a separate field of research, therefore it will not be discussed here in details. Number of different sources and research work, for example "*Deontic logic in computer science: normative system specification"* by Meyer and Wieringa *(Meyer and Wieringa, 1994)*, can be referred for detailed explanation of the use of and approaches to deontic logic.

For specifying and tranlsating a norm into design requirements, in this thesis, the following format of deontic logic is used:

```
if <condition>
then <agent>
is <deontic operator>
to <action>
```

In this format, <*deontic operator*> refers to an obligation, prohibition, or permission that the agent is responsible for.

The following example is from the case study that follows after this chapter, and this example demonstrates how deontic logic translates a norm into functional design requirements:
Let us consider that from the concrete set of norms, as a first step, the following norm was picked for translation:

An organ may not be removed when suspicion of a non-natural death exists, until it has been established that the district attorney grants the permission as mentioned in article 76, second item, of the Law on disposal of the dead.

Next step is to translate the above stated norm into the functional design requirements, using deontic logic. So, if there is a suspicion about the non-natural death (this is the condition) of the patient (*y*), then the doctor (*d*) is *prohibited* (deontic operator) to remove this patient's organ (action), unless the district attorney (*u*) gives the permission to do so. Hence, the norm is translated into the following design requirements:

```
if <suspicion(non_natural_death(y))>
then <d>
is <prohibitted>
to <remove_organ(y)>
unless <grants(u, permission, organ_removal(y))>
```

Once the norm is translated into functional design requirements, the process of integrating norms into software development process can move on to the next step.

(3) Evaluate the feasibility of design requirements
After translating the specific norm into the functional design requirements, the next step, as suggested by the model, is to evaluate the feasibility of these design requirements. As already discussed above, the process of integrating norms with software development process starts with picking out one specific norm from the concrete set of norms, and then starting the process of translation. Which means that each and every norm in the concrete set of norms is translated separately, hence, after the translation process, there is a set of design requirements for each norm. Those design requirements should not come in conflict with each other, otherwise, it will not be possible for the developed software to work as one whole system, or perhaps, it will not even be possible to implement design requirements.

In each design requirements an agent has an obligation, permission, or prohibition to perform or to not perform specific tasks in specific conditions. By the conflict between design requirements, it is meant that the responsibilities of an agent are not in harmony with each other. For example, there might be a conflict between design requirements in two of those following cases:

(1) If in the design requirements for one norm, an agent has an obligation to perform task *a* (is <obliged> to <a>), and in design requirements for another norm this same agent is prohibited to perform task *a* (is <prohibited> to <a>), and if this is happening in one time frame, we are facing a conflict between the design requirements, because, simply put, the agent can not have an obligation to perform a task, and at the same time, be prohibited to perform this task.

(2) If in design requirements for one norm, an agent has an obligation to perform task *a* (is <obliged> to <a>) and at the same time, in design requirements for another norm, the same agent has an obligation to perform task *b* (is <obliged> to <b>), there is a chance that performing task a might mean not performing task b, in which case, again, we are facing a conflict between the design requirements.

Evaluating the feasibility of the design requirements for one particular norm, involves making sure that these particular design requirements are in harmony with the design requirements of another norm. To automatically check for the conflicts between design requirements, Model checking techniques and tools *(Bé et al., 2010)* can be used. Model checking is an approach for the formal verification of design requirements for software. It automatically provides complete proofs of correctness, or explains, via counter-examples, why a system is not correct. Researchers have proposed a number of different model checking techniques and tools. Although each of these methods uses a different computational model and a different notion of verification, they all rely on decision algorithms that explicitly represent a state space, using a list or table that grows in proportion to the number of states *(Burch et al., 1990)*.

As mentioned above, model checking techniques and tools could be used during feasibility evaluation for automatically checking for conflicts between the design requirements. However, model checking techniques and tools will not be further discussed here. Those interested in understanding how different model checking techniques and tools work, can refer to number of different sources available focusing on this specific topic. For example, *"Systems and Software Verification: Model-Checking Techniques and Tools"* by Bérard and colleague researchers *(Bérard et al., 2010)*.

If design requirements are not feasible, the model suggests finding alternative ways for translating the same norm into the functional design requirements. In other words, in case of not feasible design requirements, the process goes back to the same norm and goes on a loop until design requirements are feasible. In which case, the next step suggested by the model follows.

But what happens if during the process of feasibility evaluation, there are no alternative ways found for translating norms into design requirements? In this case, as demonstrated on Figure_2, the process moves on to the "Escape" box, meaning that whatever happens after there are no alternative ways for norm translation, is outside of the scope of the model. Possible reason why there are no feasible design requirements for a specific norm, could be related to the translation of other norms from the concrete set of norms, which would require making changes in other sets of design requirements, or making changes in the concrete set of norms (such as making changes in a specific norm, or dropping a norm completely). In any case, this process falls outside of the scope of this research, therefore it is not demonstrated on the model and is not discussed further.

(4) Critical reflection on design requirements

After the design requirements are evaluated as feasible, the next step is critical reflection on design requirements. This is when Critical Design steps into the process. In today's highly-technology based environments, the process of integrating norms into a software development process involves not only software developers/designers, but also stakeholders who are affected by and have an interest in the developed software. In order to address norms explicitly during the design process and not only as an afterthought, stakeholders should also participate in the process of harmonizing norms with software

development. As the model suggests, once design requirements are feasible, stakeholders, together with software designers, should have a discussion, during which they should critically reflect on the design requirements, in order to make sure that the way the norm was translated into the design requirements is acceptable for everyone involved in and affected by the software development process. By critical reflection, software designers and stakeholders should be able to envision and demand a challenging view of users' needs and experiences and the focus of critical reflection should be on understanding that software's users' needs are constructed by society.

After critical reflection, if design requirements are acceptable for everyone involved in the process, those requirements then can be implemented and as the model demonstrates, a new norm can be picked out for translation from the concrete set of norms. However, if the requirements are not acceptable for everyone involved, then the model, again, as in case of not feasible requirements (discussed above), suggests finding alternative ways for translating the same norm into the design requirements. If such alternative way is found, the process can go back to the same norm, and start the alternative translation. Then the process goes on a loop until design requirements for this specific norm are feasible and acceptable for everyone involved in the process. But what happens if there are no alternative ways for norm translation found? Perhaps stakeholders could agree on one specific feasible alternative based on majority of votes? This would be very context specific and would depend on the scenario. In case of no alternative norm translations found it might be necessary to make changes to the design requirements translated from one of the previous norms, but those design requirements would already be implemented and whatever could be a solution for this problem, is outside the scope of the model developed within this research. The reason of not being able to find alternative norm translations, could be related to the way values were translated into the concrete set of norms, in which case Critical Design's focus and critical reflection would have to shift to value translation process, which is outside of the scope of this research. Therefore, as Figure_2 demonstrates, the process in case of no alternatives found, moves on to the "Escape" box, indicating that whatever steps should be taken in such case, falls outside the scope of the model.

Now that all the steps involved in the model for harmonizing norms with software development process have been discussed, it is clear that the process of norm implementation involves going back and making changes in design requirements if and whenever necessary. This requires the flexibility of software development process. Unlike the software development approaches that one way or another use waterfall structure, Agile Software Development Methods are very flexible with going back and making changes in design requirements whenever necessary. By having different tasks assigned to different groups of software developers, Agile Software Development Methods also make it possible to work on number of different norm translations at the same time, which is efficient when having a set of norms.

Another advantage of using Agile Software Development Methods is that instead of finding the best solution first and then implementing it, like waterfall approaches to software development process suggest, Agile Methods focus on first finding the solution which is feasible and acceptable at the time, but can be easily changed if and when necessary. The model developed within this research suggests to first develop the feasible design requirements and look for alternative norm translations only when the requirements are not feasible, or when they are feasible but are not acceptable for all the stakeholders involved in the process, and Agile Software Development Methods are flexible with going back and making changes to design requirements.

The model involves Critical Design in software development process by having the stakeholders critically reflect on design requirements, before implementing them. As discussed above, critical reflection on design requirements that are translated from a norm, is done only after those requirements are evaluated as feasible. The reason for this is that reflecting critically on design requirements that might not even be feasible for implementation, would not make sense., because it would be a waste of time. Also, Critical Design is "expensive" in a sense that it involves bringing together software developers and other stakeholders for a discussion about design requirements, so Critical Design should be used carefully. Besides, the purpose of involving Critical Design in software development process is to question design requirements and if necessary, to find alternative ways to translate norms into design requirements. In order to find an alternative norm translation, first, there must be feasible design requirements at hand to question and critically reflect on.

The model which was developed as the objective of this research, and which was in details discussed above, should contribute to the Value Sensitive Design. The VSD, as already mentioned, aims to incorporate norms and values into the technology design, however it focuses less on norm translation process and says nothing about how norms should be incorporated into the design process. The above-discussed model, on the other hand, suggests specific steps to be followed for incorporating the concrete set of norms into software development process. By involving the Critical Design in the process of harmonizing norms with software development, the model makes sure that the software development process involves not only software developers, but stakeholders as well and by having them critically reflect on design requirements, the same model makes sure that instead of being addressed as an afterthought, norms are addressed explicitly during and throughout the software development process.

4. CASE STUDY

The model for integrating a concrete set of norms into a software development process was discussed in the previous chapter. This model, in the following case study, is applied to the process of norm implementation during the software development process for the domain of organ donation. To be more specific, it is assumed that within the organ donation domain, an information system is developed in order to help doctors take right steps before getting the consent for extracting an organ from recently deceased patient. The case used in this chapter is from the medical domain [Dutch Transplant Association, 2006] and is of particular interest because the medical domain of organ transplantations is governed by a lot of restricting rules and regulations specified by the law *(Aldewereld, 2009)*, hence the process of designing the software for this domain, involves the implementation of various norms, which need to be addressed throughout the design process.

The case study starts with presenting the concrete set of norms. Next, we demonstrate how the model developed in the previous chapter should be applied to a real case of software development process. Each norm, one by one from this concrete set will be translated into design requirements. Design requirements will then be evaluated to make sure that the design requirements translated from a particular norm are in harmony with the design requirements translated from other norms.

Next step, as suggested by the model, should be the critical reflection on the design requirements. It will not be possible to actually have full critical reflection on the design requirements in this case study, because the Critical Design requires a discussion process among software developers and stakeholders. For this particular case there is not enough information about the stakeholders (or their interests and expectations related to the developed software). Even if we had such information, without having at least some medical background or knowledge, it would be impossible to imagine how stakeholders would reflect critically on the design requirements or what kinds of critical questions would they ask. Because of the reasons described in this paragraph, critical reflection on the design requirements will be skipped in the case study. However, after evaluating the feasibility of the design requirements translated from the first norm, an example critical reflection will try to demonstrate how the ideas behind the Critical Design could be used to have critical reflection on the design requirements. Afterwards, throughout the case study, after evaluating the design requirements translated from a specific norm, it will be assumed that those design requirements are acceptable for the stakeholders and critical reflection will be skipped.

Integrating the Concrete Set of Norms into the Software Development Process for Organ Donation Domain

Below (Figure_13) are listed the norms (specified by the Dutch law on Organ Transplants) which need to be incorporated into the information system being developed within the organ donation domain. Doctors will then use the developed system in order to take the right steps for getting a permission to extract an organ from a recently deceased patient. The norms in the concrete set below, are numbered randomly, since there is no required order for the norms to be implemented.

Concrete Set of Norms[2]	
Norm 1	The register can be consulted under the authority of a doctor during day and night whenever that is necessary considering the intended removal of an organ.
Norm 2	Before an organ is removed, death is certified by a doctor who may not be involved in the removal or transplantation of the organ.
Norm 3	If the doctor, in cases other than those meant with article 2, has a suspicion of the patient's non-natural death and therefore is not able to issue a death certificate, he should notify the municipal coroner immediately.
Norm 4	An organ may not be removed when suspicion of a non-natural death exists, until it has been established that the district attorney grans the permission as mentioned in article 76, second item, of the Law on disposal of the dead.
Norm 5	The doctor who determines the death makes sure that organs supposedly coming available for implantation are announced to an organ centre.
Norm 6	The one who determines the death [of a patient] takes care of checking whether a statement of intent as mentioned by article 9 and article 10 is present. If no statement of intent mentioned by article 9 and article 10 is present the doctor mentioned in the first item takes, in accordance with the protocol, care of consulting the person or persons [the relatives of the deceased patient] mentioned by article 11.

Figure_13: Concrete set of norms to be implemented for the organ donation system

(Dutch Transplant Association, 2006)

[2] Mentioned articles can be found under the *"Law on Disposal of the Dead"* and *"Law on Organ Donation"* at wetten.overheid.nl

Now that we have the concrete set of norms to be implemented during the software development process for organ donation, we can start the norm implementation process by using the model developed within this thesis. The first step of doing so, as discussed during model development, is to pick one norm from the concrete set. According to the model, it does not matter with which norm the implementation process will start, and as already mentioned, there is no required order for norm translation for this specific case of organ donation.

Pick a Norm:

Norm 1: The register can be consulted under the authority of a doctor during day and night whenever that is necessary considering the intended removal of an organ.

The next step is to translate this norm into the functional design requirements (using deontic logic).

Translate the Norm into the Design Requirements:

As discussed in section 3.2., the following format of the deontic logic will be used for translating norms into the design requirements:

```
if <condition>
then <agent>
is <deontic operator>
to <action>
```

As Norm 1 states, person under the authority of doctor has *the permission* to consult the register if there is an intended organ removal.

agent – person under the authority of doctor ("d"), denoted by "x";
condition – doctor has an intension to remove an organ from the deceased patient y;
deontic operator – x has a permission to perform an action;
action – to consult the donor register;

So, the design requirements translated from Norm 1 are:

```
if <intended (d, organ_removal(y)) and
      (x is under_authority(d))>
then <x>
is <permitted>
to <consult(register)>
```

Evaluate the Design Requirements:

Since Norm 1 is the first norm to be translated into the design requirements, at this moment there are no other design requirements yet, therefore there is no need to evaluate if the design requirements from Norm 1 are in harmony with the design requirements from other norms. However, the feasibility of Norm 1 will later be checked in any case, because the design requirements for the other norms from the concrete set have to be in harmony with design requirements translated from the other norms as well.

The next step in the process is the Critical Design. As discussed in the beginning of this chapter, critical reflection on the design requirements will be skipped in this case study. However, in order to demonstrate how critical reflection could change the design requirements, an example will be considered for Norm 1.

Critical Reflection on the Design Requirements:

As Norm 1 states, in case of intended organ removal, person under the authority of the doctor has the permission to consult the donor register. After this norm was translated into the design requirements, one critical question arose: what if, for example in case of an emergency need of an organ, or emergency surgery, there is a necessity for the doctor himself to consult the register because there is no time for an authorized person to do so. The design requirements for Norm 1 do not make it clear if the doctor himself has the same permission as the authorized person does, in case of intended organ removal. In other words, those design requirements from Norm 1 would not be acceptable for everyone involved in the discussion during Critical Design.

As suggested by the model, the next step in this case, is to find alternative translations from the same norm to the design requirements. Let us assume, that stakeholders and software developers involved in the discussion during the Critical Design process, found a possible alternative, which is giving the doctor a permission to consult the register in case of intended organ removal, as well. So now, the process goes back to Norm 1, in order to perform alternative translation:

```
if <intended (d, organ_removal(y)) and
      (x is (d) or x is under_authority(d)>
then <x>
is <permitted>
to <consult(register)>
```

Alternative design requirements for Norm 1 now gives the doctor as well as the person under the authority, the permission to consult the register. After this alternative translation is evaluated as feasible, let us assume that it is also acceptable for everyone involved in the discussion during the critical reflection, which means that this norm can now be implemented and the process of incorporating the concrete set of norms into the software development process can move on to the next norm from the concrete set.

Pick a Norm:
Norm 2: Before an organ is removed, death is certified by a doctor who may not be involved in the removal or transplantation of the organ.

Translate the Norm into the Design Requirements:
According to Norm 2, the doctor who certifies the death of a patient, is *prohibited* to be involved in the removal or transplantation of the organ.

agent – the doctor who certifies the death of a patient, denoted by "d";
condition –doctor certifies the death of the patient y;
deontic operator – doctor is prohibited to perform an action;
action – to remove the organ from the patient y;

Translated design requirements for Norm 2 are:

```
if <certify_death(d,y)>
then <d>
is <prohibited>
to<remove_organ(y)>
```

Evaluate the Design Requirements:
These design requirements are in harmony with the design requirements translated from Norm 1, because as Norm 1 states, in case of intended organ removal, agent d has a permission to consult the donor register any time, and according to Norm 2, agent d is prohibited to remove an organ, if the death of the patient was certified by the same agent d. There is no conflict detected between those two actions, because consulting the donor register has nothing to do with performing the organ removal. Meaning, even if the doctor who certified the patient's death is prohibited to perform the organ removal, this same doctor still can consult the donor register if and whenever necessary.

As already discussed, the critical reflection will be skipped. So, assuming that the above stated design requirements are acceptable for stakeholders, the translation of another norm from the concrete set can start.

Pick a Norm:

Norm 3: If the doctor, in cases other than those meant with article 2, has a suspicion of the patient's non-natural death and therefore is not able to issue a death certificate, he should notify the municipal coroner immediately.

Translate the Norm into the Design Requirements:

agent – the doctor, denoted by "d";
condition –there is a suspicion of patient y's non-natural death;
deontic operator – doctor has an obligation to perform an action;
action – to notify the municipal coroner, denoted by "k";

```
if <suspicion(non_natural_death(y))>
then <d>
is <obliged>
to <notify(k)>
```

Evaluate the Design Requirements:

First, let us see if the design requirements translated from Norm 3 are in harmony with the design requirements translated from Norm 1. According to Norm 3, if there is a suspicion of patient's non-natural death, then the doctor has an obligation to notify the municipal coroner. This obligation does not come into a conflict with the permission that the doctor (or authorized person) has according to Norm 1, because consulting the donor register would not interfere with the obligation to inform the municipal coroner, as those are two different scenarios.

Now, let us check if there is any conflict between the design requirements translated from Norm 3 and Norm 2. The prohibition to perform the organ removal as stated by Norm 2, will not interfere with the obligation to inform the municipal coroner in case of non-natural death suspicion, because even though the agent d (doctor) is the same in case of Norm 2 and Norm 3, the condition stated by Norm 2 (if d certifies y's death) is different from the condition stated by Norm 3 (if there is a suspicion of y's non-natural death). So, there is no conflict detected between the design requirements translated from Norm 2 and Norm 3.

It can be now assumed that the design requirements translated from Norm 3 are feasible and ready to be implemented.

Pick a Norm:

Norm 4: An organ may not be removed when suspicion of a non-natural death exists, until it has been established that the district attorney grants the permission as mentioned in article 76, second item, of the Law on disposal of the dead.

Translate the Norm into the Design Requirements:

If there is a suspicion of a patient's non-natural death, then the district attorney has to grant the permission of organ removal from the above-mentioned patient before the organ is removed. Otherwise, the doctor is *prohibited* to remove the organ.

agent – the doctor, denoted by "d";
condition –there is a suspicion of patient y's non-natural death;
deontic operator – doctor is prohibited to perform an action;
action – to remove the organ from the patient y, unless the district attorney denoted by "u" grants the permission of removing the organ.

```
if <suspicion(non_natural_death(y))>
then <d>
is <prohibited>
to <remove_organ(y)>
unless <grants(u, permission, remove_organ(y))>
```

Evaluate the Design Requirements:

In the design requirements translated from Norm 4, the agent d is prohibited to perform organ removal if there is a suspicion of the patient's non-natural death (unless the district attorney grants a permission to do so). The action that the doctor (d) is prohibited to perform, as stated by Norm 4, is not related to the action of consulting the donor register, as stated by Norm 1. Therefore, the design requirements translated from Norm 4 are in harmony with the design requirements translated from Norm 1.

The design requirements translated from Norm 2 and Norm 4, both prohibit the doctor to perform the organ removal. However, the condition in case of Norm 2 (doctor certifies the death of the patient) is different from the condition in case of Norm 4 (there is a suspicion of the patient's non-natural death). Hence, the design requirements translated from Norm 4 are in harmony with the design requirements translated from Norm 2.

Now let us see if the design requirements translated from Norm 4 are in harmony with the design requirements translated from Norm 3, as well. The condition that must hold in case of those two norms, is

the same – there is a suspicion of the patient's non-natural death. If this condition holds, Norm 3 gives the doctor an obligation to inform the municipal coroner, and Norm 4 prohibits the doctor two perform the organ removal (unless the permission to do so is granted by the district attorney). Actions that the agent (doctor) is obliged and prohibited to perform according to Norm 3 and Norm 4, are different and do not come in conflict with each other. Therefore, the design requirements translated from Norm 4, are in harmony with the design requirements translated from Norm 3, as well.

Norm 4 was translated into the design requirements which then were evaluated as feasible, therefore, if it is assumed that those design requirements are acceptable for the stakeholders, Norm 4 is ready to be implemented and another norm can be picked for the translation.

Pick a Norm:

Norm 5: The doctor who determines the death makes sure that organs supposedly coming available for implantation are announced to an organ centre.

Translate the Norm into the Design Requirements:

In case of the possible organ donation, the doctor, after certifying the death of the patient, has *an obligation* to announce the organ of the deceased patient to an organ centre:

agent – the doctor, denoted by "d";
condition –doctor has certified the death of a patient y, and an organ of patient y is possibly available for the donation;
deontic operator – doctor has an obligation to perform an action;
action – to announce the organ of the patient y to the organ center. The organ center is denoted by "c";

```
if <certify_death(d,y) and available(organ(y))>
then <d>
is <obliged>
to <announce(d,organ(y),c)>
```

Evaluate the Design Requirements:

According to the design requirements translated from Norm 5, after certifying the death of the patient, in case of the available organ for possible donation, the doctor has an obligation to announce this organ to an organ centre. The obligation to perform this action is not in conflict with the permission to consult the donor register, granted to the doctor by Norm 1, because the process of consulting the register is different from the process of announcing the available organ to the organ centre, and those two processes do not interfere with each other. So, the design requirements translated from Norm 5 are in harmony with the design requirements translated from Norm 1.

Norm 2 states that the doctor, who certifies the death of the patient, is prohibited to perform the organ removal for this deceased patient. This same doctor, according to Norm 5, is obliged to announce the organ (possibly available for the donation) to the organ centre. Being prohibited to perform the organ removal (Norm 2) does not stand in a way of announcing the available organ to the organ centre (Norm 5). Meaning that, if the doctor certifies the death of the patient, he cannot perform the organ removal (because design requirements translated from Norm 2 prohibit the doctor from doing so), however, this does not mean that this doctor cannot fulfill the obligation of announcing the available organ to the organ centre. Therefore, the design requirements translated from Norm 5 are in harmony with the design requirements translated from Norm 2.

The design requirements translated from Norm 5 are also in harmony with the design requirements translated from Norm 3, because Norm 3 obliges the doctor to report to the local coroner if there is a suspicion of the patient's non-natural death, and Norm 5 obliges the doctor to announce the organs available for the donation to the organ centre after certifying the death of the patient. So, those two obligations, as well as the conditions stated by Norm 3 and Norm 5, are completely different and do not interfere with each other.

Norm 4 prohibits the doctor to perform the organ removal if there is a suspicion of the patient's non-natural death. Performing the organ-removal (no matter if the doctor is obliged, permitted, or prohibited to do so) cannot interfere with announcing an organ to the organ centre (Norm 5), because those two are different actions, taking place in two different conditions. So the design requirements translated from Norm 5 are also in harmony with the design requirements translated from Norm 4. Thus, the design requirements translated from Norm 5 are feasible.

Pick a Norm:

Norm 6: The one who determines the death [of a patient] takes care of checking whether a statement of intent as mentioned by article 9 and article 10 is present. If no statement of intent mentioned by article 9

and article 10 is present the doctor mentioned in the first item takes, in accordance with the protocol, care of consulting the person or persons [the relatives of the deceased patient] mentioned by article 11.

Translate the Norm into the Design Requirements:

The doctor who certifies the death of the patient, has an *obligation* to check the statement of intent for this patient in the registry:

agent – the doctor, denoted by "d";
condition –doctor has certified the death of a patient y;
deontic operator – doctor has an obligation to perform an action;
action – to check the statement of intent, which is denoted by "z";

after checking the statement of intent:

agent – the doctor, denoted by "d";
condition –doctor has certified the death of a patient y and d has checked for the statement of intent z, and there is no statement of intent for the patient y;
deontic operator – doctor has an obligation to perform an action;
action – to consult the relatives of the patient y;

```
if <certify_death(d,y)>
then <d>
is <obliged>
to <check (z)>
if <certify_death(d,y) and check(d,z) and no_statement_of_intent(y)>
then <d>
is <obliged>
to <consult_relatives(y)>
```

Evaluate the Design Requirements:

Norm 6 states that the doctor who certifies the death of the deceased patient is obliged to check for this patient's statement of intent for organ donation, and if necessary, the doctor is also obliged to consult the patient's relatives. The design requirements translated from this norm are not in conflict with the design requirements translated from Norm 1, because Norm 1 gives the doctor (or authorized person) the permission to consult the donor register whenever necessary, and in order to check for the statement of intent (Norm 6), the doctor has to consult the donor register. So the design requirements translated from Norm 6 are indeed in harmony with the design requirements translated from Norm 1.

Norm 2 prohibits the doctor who certifies the death of the patient to perform the organ-removal from this patient, but Norm 6 obliges this doctor (same agent d) to check for the deceased patient's statement of intent, after certifying the patient's death. Simply put, according to Norm 2 and Norm 6, the doctor who certifies the death of the patient has to check for the statement of intent for the organ removal, or if necessary, has to consult the patient's relatives for the permission, but this doctor can not perform the actual organ removal. It is clear that the prohibition and obligations of the doctor stated by Norm 2 and Norm 6 do not come into conflict with each other, therefore, the design requirements translated from those two norms are in harmony with each other.

Norm 3 obliges the doctor to notify the municipal coroner, in case there is a suspicion of the recently deceased patient's non-natural death and Norm 4 prohibits this same doctor to perform the organ removal unless the district attorney grants the permission to do so. This obligation and prohibition do not come in conflict with the doctor's obligation to check for the statement of intent (as stated by Norm 6), because the latter happens if the patient's death is certified first. In other words, the design requirements translated from Norm 6 are in harmony with the design requirements translated from Norm 3 and from Norm 4 as well.

As Norm 5 states, the doctor who certifies the death of the patient has an obligation to inform the organ centre, in case there is an organ of the deceased patient available for the donation. Being obliged to announce an organ available for the possible donation to the organ centre does not come in conflict with the obligation to check for the deceased patient's statement of intent and if necessary to consult the relatives (Norm 6), therefore, the design requirements translated from Norm 6 are in harmony with the design requirements translated from Norm 5.
We can assume that the design requirements translated from Norm 6 are feasible and ready to be implemented.

Now that all six norms from the concrete set of norms have been translated into the design requirements and have been evaluated as feasible, based on those six norms that were introduced in this case study, we can actually map the steps and procedures necessary for a doctor (d) to follow in order to get the permission for the organ donation of the recently deceased patient. By doing so, it is possible to sketch the protocol governed by the concrete set of norms. This protocol can then be used as the guideline for the actions that need to be taken by the doctor (d) in order to see when it is possible to make an organ donation.

Based on the design requirements translated from the concrete set of norms in this case study, the actions involved in the organ donation process is demonstrated on the protocol below (Figure_14):

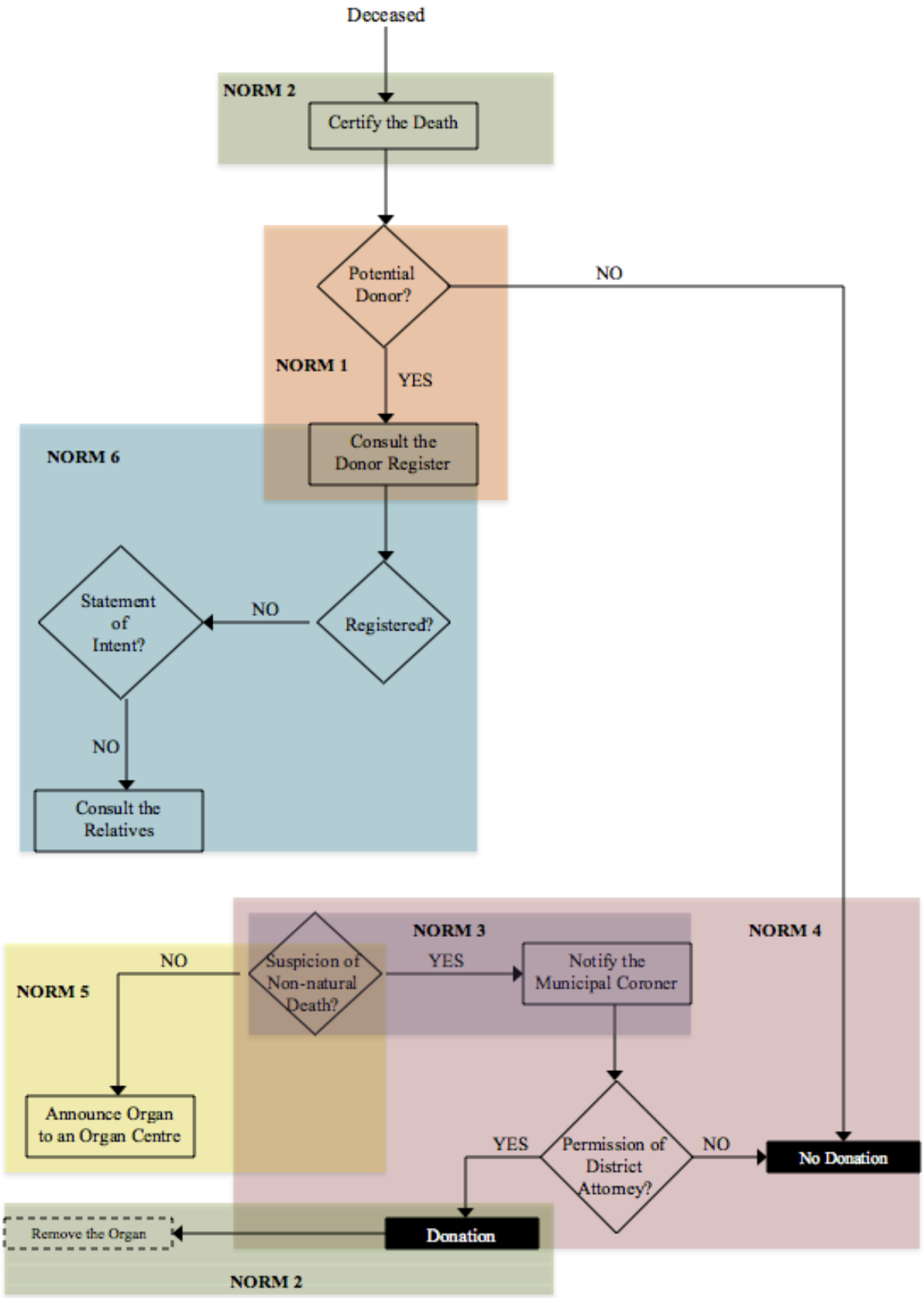


Figure_14: The Protocol for The Case Study

The actions represented on this protocol, are governed by the norms that were translated into the design requirements in this case study. However, the concrete set of norms in this case study included only six norms. In reality, the number of norms to be translated is much bigger than six, because, as already mentioned, the domain of organ donation involves the implementation of large number of norms. The norms not available for this case study, could not be interpreted on the protocol on Figure_14, therefore, Figure_14 does not represent what the actual protocol for an organ donation system looks like.

Figure_15 below demonstrates the simplified version of the actual protocol created [by Dutch Transplant Association, 2006] in order to help doctors do what is needed before getting a consent to perform organ removal from a recently deceased patient. This protocol, unlike the protocol described above, includes all steps that are needed, specified by law and medical regulations, and claims that no violations will occur if followed precisely.

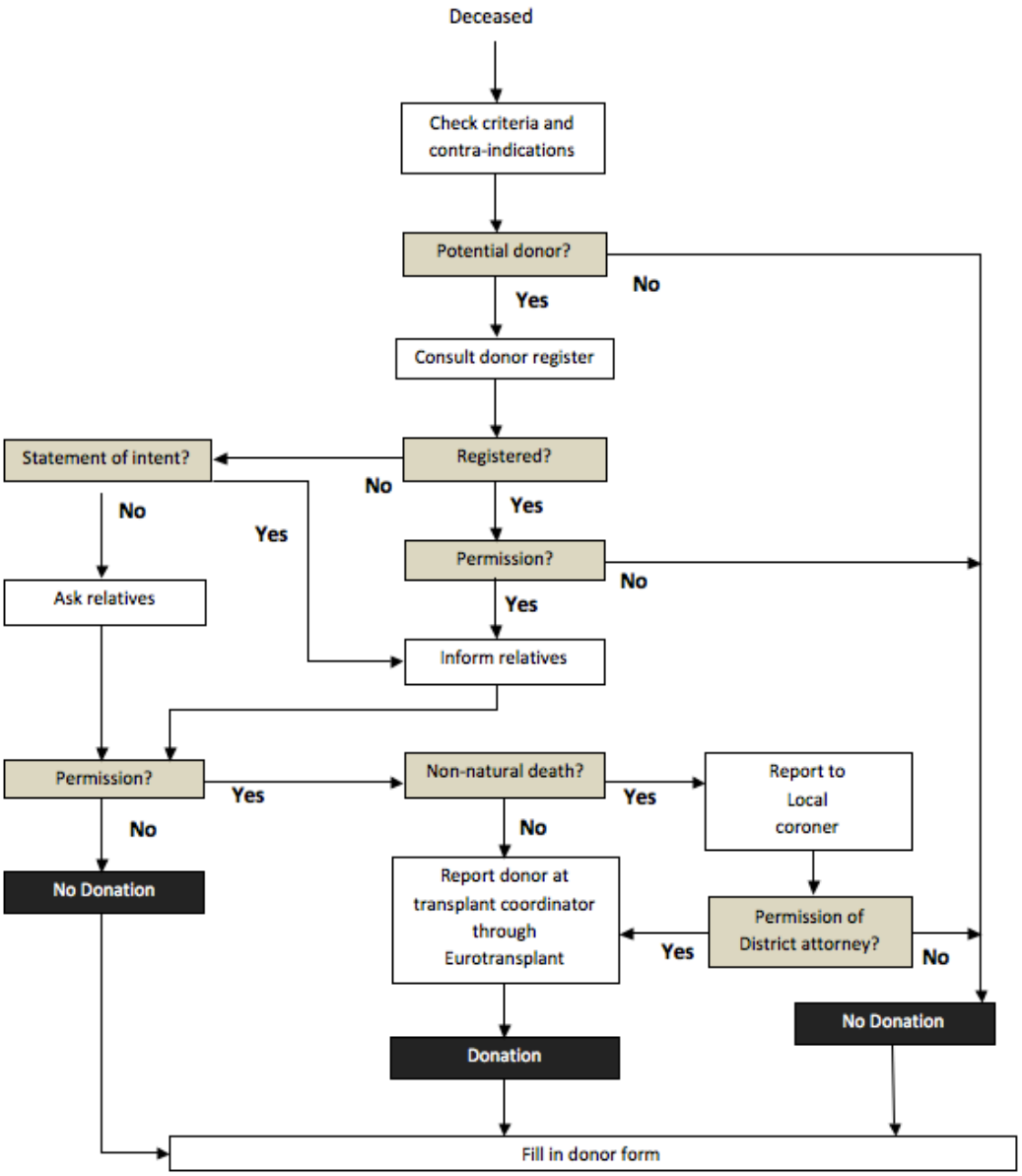


Figure_15: The Protocol for The Organ Donation System

If we label the actions involved on the original protocol with the six norms that where previously translated into the design requirements, then we will actually be able to compare the original protocol with the protocol sketched within this case study. Figure_16 below shows the side-side comparison of those two protocols. The protocol on the left side of the Figure_16 is the protocol created within this case study, while the protocol on the right side – is the original protocol labeled with the norms:

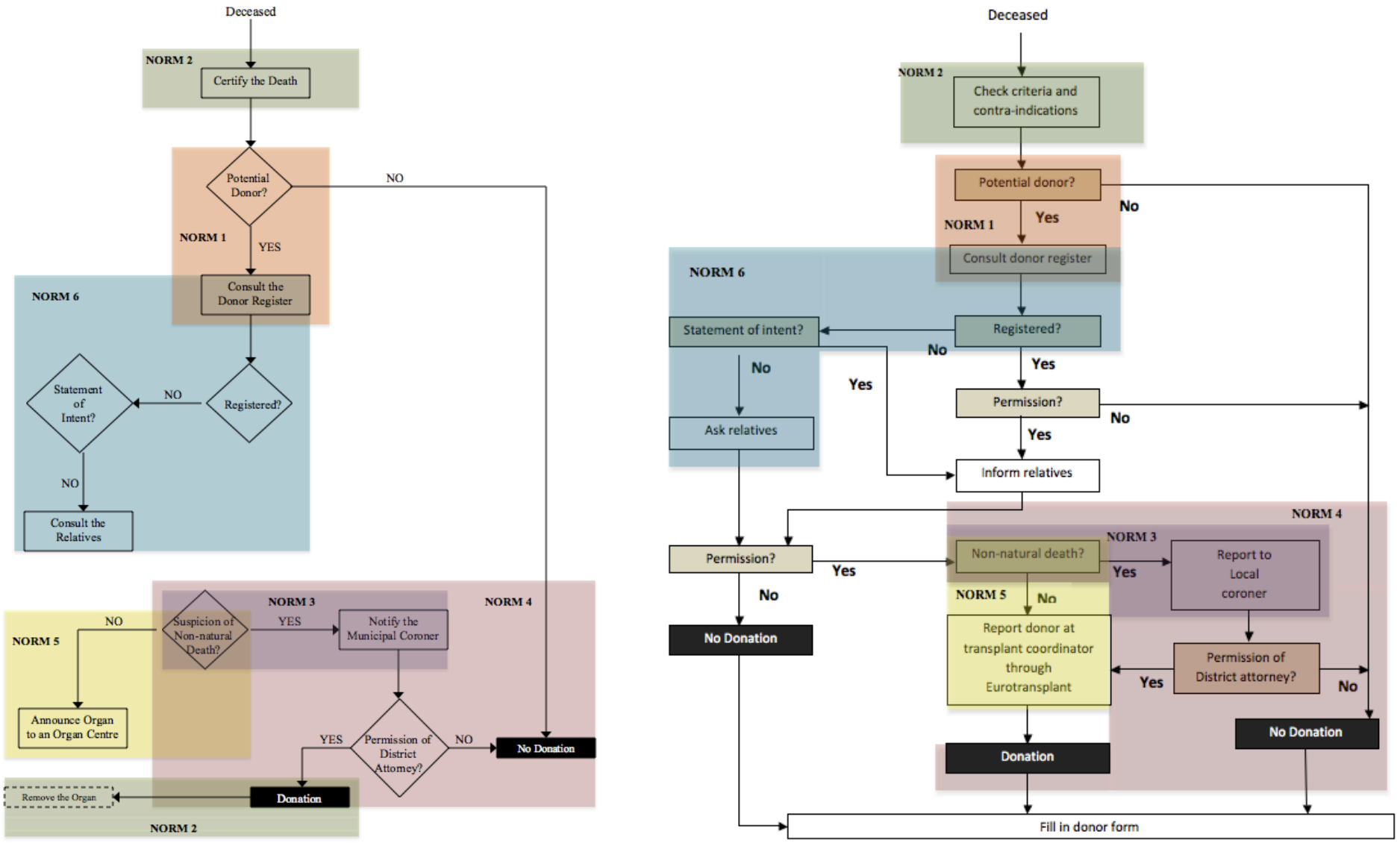


Figure_16: Side_side Comparison of Two Protocols

The side-side comparison of two protocols on Figure_16 demonstrates that the protocol created based on the design requirements translated from the norms available in the case study, is accurate, because the actions governed by the concrete set of norms actually matches the actions represented on the original protocol. As discussed above, the reason why the protocol sketched within this case study does not demonstrate all the actions involved in the process of getting consent for organ removal (like on the actual protocol) is because the number of the norms that are implemented in the system within the organ donation domain is actually more than what was available for this case study. However, in case of Norm 2, as you can see, on the original protocol the action of removing an organ (after having the permission of organ donation) is not demonstrated, while on the left protocol it is included. The reason for this is that as Norm 2 states, the doctor who certifies the death of a patient may not be involved in the removal of an organ. In other words, certifying the death and removing an organ from the deceased patient is not done by the same doctor, therefore the action of the organ removal on the left protocol is represented with the dashed box. The actions governed by Norm 2 on the left protocol match the actions governed by the same norm on the right protocol, with the difference that the left protocol gives more information based on the design requirements translated from Norm 2.

This case study was built around the concrete set of six norms and was used in order to show how the model developed within this research can be applied to a software development process. However, the validity of the model – which was applied to only one case - is not guaranteed. Further limitations of this research will be discussed in the final chapter (6.Limitations and Future Work) of this thesis. Before that, the following chapter will draw conclusions from this thesis.

5. CONCLUSION

As discussed in the introduction chapter of this thesis, there is a limited research focused on producing practical methodologies for systematically incorporating norms into a technology design process. Practical problem at stake is that there are no guidelines for addressing, translating, or implementing norms throughout the design process. In regard with this problem, this research aimed to develop the model for incorporating a concrete set of norms into a software design process.

In order to answer the main research question, first the answer to the research sub-question was found, by identifying and analyzing theories relevant to this research. This was done in the Theoretical Background chapter, after which, the identified theories, and the ideas behind those theories where used to develop the above-mentioned model.

Influenced by the ideas behind the theory on Value Sensitive Design (VSD), during the process of norm implementation, the model involves stakeholders in the software design process. However, VSD does not provide guidelines on how exactly stakeholders should be involved in a design process. The model uses Critical Design to do so. To be more specific, Critical Design brings stakeholders for a discussion together with software designers, in order to reflect critically on software design requirements. If those requirements are not acceptable for everyone involved in the design process, alternative design requirements are found. So in this case, the design process goes back to the same norm to find an alternative norm translation. By bringing stakeholders and software designers together for a discussion, and for critical reflections, Critical Design creates collaboration between stakeholders and designers, and therefore, stakeholders are involved in a software development/design throughout the design process.

As discussed during the model development, the model for incorporating a concrete set of norms into a software design process suggests that if and whenever necessary, it should be possible to go back to the norm translation process and to find alternative ways to translate the norm into the design requirements. Going back and making changes to design requirements requires flexible approach to a software development process. By flexible approach it is meant that the approach should allow going back and making changes to design requirements whenever necessary. Traditional approaches to software development, for example the Waterfall Model, focus on a sequential design process: flowing steadily downwards through phases during a software development process. Those approaches do not consider that there might be a need to make changes in software design requirements at any point during the software design process. Therefore, Waterfall Model is not flexible for going back and making changes to design requirements. Then there are other software development methods, Agent-Oriented Software Engineering (AOSE), and Service-Oriented Architecture (SOA), for example. Those methods, even though they are not as old fashioned as the Waterfall Model is, are somehow influenced by the waterfall top-down approach. In other words, the methods mentioned in this paragraph (and discussed in chapter 2) do not offer flexibility for making changes in design requirements. This flexibility is necessary for the model to be effective, because, as already discussed, the model suggests that if and whenever necessary, it should be possible to go back to a norm and find an alternative translation for it (in other words, change design requirements). Besides, neither the traditional approaches to the software development process, neither the AOSE, nor SOA focus on involving stakeholders in the process of designing a software. And the importance of bringing stakeholders and software designers together during the design process, was

already discussed above. On the other hand, one of the characteristics of the Agile Software Development Methods is flexibility. The Agile Software Development Methods are flexible in a sense that those methods consider going back and making changes to software design requirements, if and whenever it is necessary. Also, the Agile Software Development Methods focus on a collaboration between software designers and stakeholders, and on the involvement of stakeholders in a software design process. For the reasons discussed in this paragraph, applying the model developed in this thesis to the software development process, is recommended when this process is managed by one of the Agile Software Development Methods.

Scientific Contribution

The model developed within this research makes scientific contribution to the theory on Value Sensitive Design. As already discussed, Value Sensitive Design does not make it clear how norms should be incorporated into a technology design. The model developed within this research, on the other hand, can be seen as a guideline for incorporating a concrete set of norms into a software design process. Besides, theory on Value Sensitive Design focuses on the importance of involving stakeholders in a design process, but again, VSD does not explain how exactly should this be done. The model for incorporating a concrete set of norms into a software design process suggests using Critical Design for involving stakeholders in a software design process. How exactly Critical Design involves stakeholders in a process throughout the software development, was already discussed.

Besides involving stakeholders in a design process, by using Critical Design in a software development process, the aim of the model developed in this thesis is to make sure that instead of being addressed as an afterthought, norms are addressed throughout a software design process. And this is done by critically reflecting on design requirements translated from a norm, in order to find alternative translations if and whenever necessary. As discussed in chapter 2 (Theoretical Background), the main idea behind Critical Design is to question values and norms throughout a design process. However, within the scope of this research, the developed model suggests using Critical Design for having a critical reflection on software design requirements translated from a specific norm. By doing so, the model makes sure that instead of being addressed as an afterthought, norms are being addressed throughout a design process. So, this research makes scientific contribution to the theory on Critical Design by suggesting that addressing norms throughout a design process can be done not only by questioning those norms, but also, by critically reflecting on design requirements translated from those norms.

In the introduction chapter of this thesis it was discussed that the link between Norms and Service Choreography on the concrete level of the VSSD framework was not clear. By developing a model for incorporating a concrete set of norms into a software design, this research also contributes to the VSSD framework (to the link between Norms and Service Choreography, in particular).

In chapter 4 (Case Study) the model developed within this research was applied to the example case of a software development process within a domain of organ donation. Applying the model to the example case demonstrated how the model works and how can it be applied to a real case of a software development process.

In conclusion, by developing the model for integrating a concrete set of norms into a software development process, this thesis answered the main research question formulated in the introduction. However, the limitations of the model, and the recommendations for the future work, are discussed in the following chapter.

6. LIMITATIONS AND FUTURE WORK

The biggest limitation that this research faces is the methodological limitation. As discussed in chapter 1 *(1.3. Research Approach)* due to the limited time available for the master thesis, the Implementation Phase of the research fell outside the scope of the study, which means that the model developed during the Concept Development Phase was evaluated during the example case study, instead of first being implemented and put into a practice in a real life context. Because the Implementation Phase did not take place during the research, the validity and soundness of the model right now is under a question mark. Even though the model claimed to be effective when applied to the example case in chapter 4, we can not be 100% sure what would have happened in a real life context, especially that the Critical Design step recommended by the model was (partly) skipped. It would not have been possible otherwise, because to know the actual outcome of the critical reflection on the design requirements, the model has to be put in practice in a real case of the software development process, and the real software developers and real stakeholders have to be involved in the process. Perhaps it is possible to assume what the critical reflection could be for a particular case, however, not actually having the discussion (involving number of software developers and stakeholders) soaked with the critical reflection, would then question the soundness of the assumption.

The limitations discussed in the previous paragraph could be overcome in the future, by applying the model to a real software development process managed by one of the Agile Software Development Methods. This statement leads us to another limitation of the research: the model suggests that once a norm is translated into the design requirements, it should be possible to go back to the same norm and make changes to the design requirements, whenever necessary. In the process the model also involves Critical Design, which means that during a software development process it should be possible to bring together software developers and stakeholders into a discussion for a critical reflection on design requirements. Unlike other approaches to a software development process, the Agile Software Development Methods are flexible with making changes to design requirements, and focus on involving stakeholders in a process of software development. Therefore it is argued that the model developed within this research perfectly fits with a software development process managed by one of the Agile Software Development Methods. But how can the model developed within this research be generalized? How can a concrete set of norms be integrated into a software development process managed by an approach other than the Agile Methods? Those questions could be a motivation for the future research.

Another limitation that the model faces is that the model does not show what should be done if there is no alternative norm translation found in case of not feasible or not acceptable design requirements ("Escape" box on the model, Figure_12). As discussed in model development chapter, what should happen instead of the "Escape" box on the model, is outside the scope of this model. Therefore, perhaps, the limitation discussed in this paragraph could be overcome by extending the scope of this research in the future.

Despite the limitations discussed above, by developing the model for integrating a concrete set of norms into a software design process, this master thesis makes a scientific contribution to the theories on Value Sensitive Design (VSD), Critical Design, and also to the Value Sensitive Software Development (VSSD) framework. Scientific contributions of this research are discussed in chapter 5.

Besides the scientific contributions, this master thesis will be helpful for high-tech companies. Companies in today's highly technology based environment (which also involves number of different stakeholders with different values, interests, and expectations) are facing the problem of addressing norms during a design process and norms are usually addressed as an afterthought because before this master thesis, there was no specific guideline, framework, or model for integrating norms into a design process. By applying the model developed within this research to a software development process, high-tech companies will be able to involve stakeholders, together with designers, in a software design process. Besides, the model developed within this research makes sure that software design requirements are acceptable for everyone involved in a design process. Overall, by using the model developed within this research, high-tech companies can address and incorporate norms into a software design throughout a software design process.

REFERENCES:

Aldewereld, H. (2007). Autonomy vs. conformity: An institutional perspective on norms and protocols. *PhD thesis, Universiteit Utrecht.*

Aldewereld, H., Álvarez-Napagao, S., Dignum, F., & Vázquez-Salceda, J. (2010, May). Making norms concrete. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1* (pp. 807-814). International Foundation for Autonomous Agents and Multiagent Systems.

Aldewereld, H., Dignum, V., and Tan Y. -H (2015). Design for Values in Software Development. In, van den Hoven, J., Vermaas, P., van de Poel, I. (eds.) *Handbook of Ethics, Values, and Technological Design.* Springer.

Andrighetto, G., Castelfranchi, C., Mayor, E., McBreen, J., López-Sánchez, M., & Parsons, S. (2013). (Social) Norm Dynamics. In G. Andrighetto, G. Governatori, P. Noriega, & L. W. van der Torre (Eds.), *Normative Multi-Agent Systems* (pp. 135-170).

Arsanjani, A. (2004). Service-oriented modeling and architecture. *IBM developer works*, 1-15.

Bardzell, J., & Bardzell, S. (2013, April). What is critical about critical design?. In Grinter, R. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 3297-3306). ACM.

Bardzell, S., Bardzell, J., Forlizzi, J., Zimmerman, J., & Antanitis, J. (2012, June). Critical design and critical theory: the challenge of designing for provocation. In *Proceedings of the Designing Interactive Systems Conference*(pp. 288-297). ACM.

Bass, L. (2007). *Software architecture in practice*. Pearson Education India.

Beck, K., & Andres, C. (2004). *Extreme programming explained: embrace change*. Addison-Wesley Professional.

Beck, K., & Fowler, M. (2001). *Planning extreme programming*. Addison-Wesley Professional.

Bench-Capon, T. J. M. (2002). The missing link revisited: The role of teleology in representing legal argument. *Artificial Intelligence and Law*, *10*(1-3), 79-94.

Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., & Schnoebelen, P. (2010). *Systems and software verification: model-checking techniques and tools*. Springer Publishing Company, Incorporated.

Bjerknes, G., & Bratteteig, T. (1995). User participation and democracy: a discussion of Scandinavian research on system development. *Scandinavian Journal of information systems*, *7*(1), 1.

Bødker, S. (1987). Through the interface-A human activity approach to user interface design. *DAIMI Report Series*, *16*(224).

Boehm, B. W. (1988). A spiral model of software development and enhancement. *Computer*, *21*(5), 61-72.

Boehm, B. W. (1996). Anchoring the software process. *Software, IEEE*, *13*(4), 73-82.

Boehm, B. (2002). Get ready for agile methods, with care. *Computer*, *35*(1), 64-69.

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., & Mylopoulos, J. (2004). Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, *8*(3), 203-236.

Broersen, J., Cranefield, S., Elrakaiby, Y., Gabbay, D., Grossi, D., Lorini, E., Xavier, P. & Schwarzentruber, F. (2013). Normative reasoning and consequences. *Dagstuhl Follow-Ups*, *4*.

Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., & Hwang, L. J. (1990, June). Symbolic model checking: 10 20 states and beyond. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e* (pp. 428-439). IEEE.

Budd, R. W., Thorp, R. K., & Donohew, L. (1967). *Content analysis of communications.* Macmillan.

Bynum, T. W. (2001). Computer ethics: Its birth and its future. *Ethics and Information Technology*, *3*(2), 109-112.

Cao, L., Mohan, K., Xu, P., & Ramesh, B. (2004, January). How extreme does extreme programming have to be? Adapting XP practices to large-scale projects. In *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on* (pp. 10-pp). IEEE.

Carmo, J., & Jones, A. J. (2002). Deontic logic and contrary-to-duties. In *Handbook of philosophical logic* (pp. 265-343). Springer Netherlands.

Carroll, J. M., & Rosson, M. B. (2006). Dimensions of participation in information systems design. *Human-Computer Interaction and Management Information Systems: Foundations P. Zhang and D. Galletta (eds.). Armonk, NY: ME Sharp Inc*, 337-354.

Cavanagh, S. (1997). Content analysis: concepts, methods and applications. *Nurse researcher*, *4*(3), 5-13.

Channabasavaiah, K., Holley, K., & Tuggle, E. (2003). Migrating to a service-oriented architecture. *IBM DeveloperWorks*, *16*.

Cockburn, A. (2004). *Crystal clear: a human-powered methodology for small teams*. Pearson Education.

Dittrich, Y., John, M., Singer, J., & Tessem, B. (2007). For the special issue on qualitative software engineering research. *Information and software technology*,*49*(6), 531-539.

Dunne, A. (2006). *Hertzian tales: Electronic products, aesthetic experience, and critical design*. MIT Press.

Dunne, A., & Raby, F. (2001). *Design noir: The secret life of electronic objects*. Springer.

Dybå, T. (2000). Improvisation in Small Software Organizations: Implications for Software Process Improvement. *IEEE Software*, *17*(5), 82-87.

Dybå, T., & Dingsøyr, T. (2008). Empirical studies of agile software development: A systematic review. *Information and software technology*, *50*(9), 833-859.

Erickson, J., Lyytinen, K., & Siau, K. (2005). Agile modeling, agile software development, and extreme programming: the state of research. *Journal of Database Management (JDM)*, *16*(4), 88-100.

Faber, A., Alexandersson, V., & Lundmark, S. (2014). Critical design goggles: Explorative use of critical design perspectives in a video production project. In, *SIDER'14 Royal Institute of Technology, KTH,* Stockholm, Sweden.

Flanagan, M., Howe, D. C., & Nissenbaum, H. (2005, April). Values at play: design tradeoffs in socially-oriented game design. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 751-760). ACM.

Frayling, C. (1993). *Research in art and design*. London: Royal College of Art.

Friedman, B., & Kahn Jr, P. H. (2002, January). Human values, ethics, and design. In *The human-computer interaction handbook* (pp. 1177-1201). L. Erlbaum Associates Inc..

Friedman, B., Kahn Jr, P. H., & Borning, A. (2006). Value sensitive design and information systems. *Human-computer interaction in management information systems: Foundations*, *5*, 348-372.

Friedman, B., Kahn, P., & Borning, A. (2002). Value sensitive design: Theory and methods. *University of Washington technical report*.

Fuchs, L. (1999, January). AREA: a cross-application notification service for groupware. In *ECSCW'99* (pp. 61-80). Springer Netherlands.

Galegher, J., Kraut, R.E., and Egido, C. (Eds.) (1990). *Intellectual Teamwork: Social and Technological Foundations of Cooperative Work.* Lawrence Erlbaum Associates, Hillsdale, NJ.

Gaver, W. W., Bowers, J., Boucher, A., Gellerson, H., Pennington, S., Schmidt, A., Steed, A., Villars, N. & Walker, B. (2004, April). The drift table: designing for ludic engagement. In *CHI'04 extended abstracts on Human factors in computing systems* (pp. 885-900). ACM.

Grudin, J. (1988, January). Why CSCW applications fail: problems in the design and evaluation of organizational interfaces. In *Proceedings of the 1988 ACM conference on Computer-supported cooperative work* (pp. 85-93). ACM.

Highsmith, J., & Cockburn, A. (2001). Agile software development: The business of innovation. *Computer*, *34*(9), 120-127.

Hsieh, H. F., & Shannon, S. E. (2005). Three approaches to qualitative content analysis. *Qualitative health research*, *15*(9), 1277-1288.

Humphrey, W. S. (1989). *Managing the Software Process*. Addison-Wesley Professional.

Johnson, E. H. (2000). Getting Beyond the Simple Assumptions of Organizational Impact. *Bulletin of the American Society for Information Science and Technology*, *26*(3), 18-19.

Jennings, N. R. (2001). An agent-based approach for building complex software systems. *Communications of the ACM*, *44*(4), 35-41.

Jennings, N., & Wooldridge, M. J. (Eds.). (1998). *Agent technology: foundations, applications, and markets*. Springer.

Johnson, D.G. and Miller, K. (1997). Ethical issues for computer scientists and engineers. In: Tucker, A.B. JR. (ED. –in-Chief), *The Computer Science and Engineering Handbook*. CRC Press, pp. 16-26.

Kling, R., Rosenbaum, H., & Hert, C. (1998). Social informatics in information science: an introduction. *Journal of the American Society for Information Science*, *49*(12), 1047-1052.

Kling, R. and Star, S.L. (1998). Human centered systems in the perspective of organizational and social informatics. *Computers and Society*, 28(1), 22-29.

Le Dantec, C. A., Poole, E. S., & Wyche, S. P. (2009, April). Values as lived experience: evolving value sensitive design in support of value discovery. In *Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 1141-1150). ACM.

Liu, K., Sun, L., Dix, A., & Narasipuram, M. (2001). Norm-based agency for designing collaborative information systems. *Information Systems Journal*,*11*(3), 229-247.

Lindkvist, K. (1981). Approaches to textual analysis. *Advances in content analysis*, *9*, 23-42.

Lindstrom, L., & Jeffries, R. (2004). Extreme programming and agile software development methodologies. *Information Systems Management*, *21*(3), 41-52.

Ludwig, H., Keller, A., Dan, A., King, R., & Franck, R. (2003). A service level agreement language for dynamic electronic services. *Electronic Commerce Research*, *3*(1-2), 43-59.

MacCormack, A., Crandall, B., Kemerer, C. F., & Cusumano, M. (2003). Trade-offs between productivity and quality in selecting software development practices. *IEEE Software*, *20*(5), 78-85.

Mahar, S., & Bhatia, P. K. (2012). Agent Oriented Software Development. *International Journal of Software Engineering Research and Practices*, *1*(4), 17-23.

Mayring, P. (2000, June). Qualitative content analysis. In *Forum: Qualitative social research* (Vol. 1, No. 2).

McTavish, D. G., & Pirro, E. B. (1990). Contextual content analysis. *Quality and Quantity*, *24*(3), 245-265.

Meyer, J. J. C., & Wieringa, R. J. (1994). *Deontic logic in computer science: normative system specification*. John Wiley and Sons Ltd..

Mylopoulos, J., Kolp, M., & Giorgini, P. (2002). Agent-oriented software development. In *Methods and Applications of Artificial Intelligence* (pp. 3-17). Springer Berlin Heidelberg.

Nan, Z., Qiu, X. S., & Meng, L. M. (2006, October). A SLA-based service process management approach for SOA. In *Communications and Networking in China, 2006. ChinaCom'06. First International Conference on* (pp. 1-6). IEEE.

Nerur, S., Mahapatra, R., & Mangalaraj, G. (2005). Challenges of migrating to agile methodologies. *Communications of the ACM*, *48*(5), 72-78.

Nissenbaum, H. (1999). Can trust be secured online? A theoretical perspective. *Etica e Politca*, 2 (electronic journal).

Olson, J. S., & Teasley, S. (1996, November). Groupware in the wild: lessons learned from a year of virtual collocation. In *Proceedings of the 1996 ACM conference on Computer supported cooperative work* (pp. 419-427). ACM.

Orlikowsi, W.J. and Iacono, C.S. (2001). Research commentary: desperately seeking the "IT" in IT research: a call to theorizing the IT artifact. *Information Systems Research*, 12(2).

Palmer, S. R., & Felsing, M. (2001). *A practical guide to feature-driven development*. Pearson Education.

Papazoglou, M. P. (2003, December). Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on* (pp. 3-12). IEEE.

Poppendieck, M., & Poppendieck, T. (2003). *Lean software development: An agile toolkit*. Addison-Wesley Professional.

Raibulet, C., & Massarelli, M. (2008, September). Managing Non-functional Aspects in SOA through SLA. In *Database and Expert Systems Application, 2008. DEXA'08. 19th International Workshop on* (pp. 701-705). IEEE.

Rosengren, K. E. (1981). Advances in Scandinavia content analysis: An introduction. *Advances in content analysis*, 9-19.

Royce, W. W. (1970, August). Managing the development of large software systems. In *proceedings of IEEE WESCON* (Vol. 26, No. 8).

Sawyer, S., & Rosenbaum, H. (2000). Social informatics in the information sciences: Current activities and emerging directions. *Informing science*, *3*(2), 89-89.

Schwaber, K., & Beedle, M. (2001). *Agilè Software Development with Scrum.* Prentice Hall.

Sengers, P., Boehner, K., David, S., & Kaye, J. J. (2005, August). Reflective design. In *Proceedings of the 4th decennial conference on Critical computing: between sense and sensibility* (pp. 49-58). ACM.

Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., & Zelesnik, G. (1995). Abstractions for software architecture and tools to support them.*Software Engineering, IEEE Transactions on*, *21*(4), 314-335.

Shaw, M., & Garlan, D. (1996). *Software architecture: perspectives on an emerging discipline* (Vol. 1, p. 12). Englewood Cliffs: Prentice Hall.

Simpson, J.A. and Weiner, E.S.C. (Eds.) (1989) "Value, n." *Oxford English Dictionary*. Clarendon Press, Oxford; Oxford University Press, 30 may 2003.

Sprott, D., & Wilkes, L. (2004). Understanding service-oriented architecture. *The Architecture Journal*, *1*(1), 10-17.

Tesch, R. (1990). *Qualitative Research: Analysis Types and Software Tools.* Routledge.

Van den Hoven, J. (2007). ICT and value sensitive design. In *The information society: Innovation, legitimacy, ethics and democracy in honor of Professor Jacques Berleur SJ* (pp. 67-72). Springer US.

Van de Poel, I. (2013). Translating values into design requirements. In *Philosophy and Engineering: Reflections on Practice, Principles and Process*(pp. 253-266). Springer Netherlands.

Verschuren, P., & Hartog, R. (2005). Evaluation in design-oriented research. *Quality and Quantity*, *39*(6), 733-762.

Weber, R. P. (Ed.). (1990). *Basic content analysis* (No. 49). Sage.

Williams, L., & Cockburn, A. (2003). Agile software development: it's about feedback and change. *IEEE Computer Society, 36 (6)*.

Wooldridge, M., & Jennings, N. R. (1995). Intelligent agents: Theory and practice. *The knowledge engineering review*, *10*(02), 115-152.

Wooldridge, M., Jennings, N. R., & Kinny, D. (2000). The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, *3*(3), 285-312.

Zambonelli, F., Jennings, N. R., & Wooldridge, M. (2003). Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *12*(3), 317-370.

Zambonelli, F., & Omicini, A. (2004). Challenges and research directions in agent-oriented software engineering. *Autonomous Agents and Multi-Agent Systems*, *9*(3), 253-283.

Zhang R.Y., Qiu X.S. and Meng L.M. "SLA representation method and application", *Journal of BUPT*, volume 26, pp. 13-17, 2003.

Zambonelli, F., & Parunak, H. V. D. (2004). 'Toward a change of paradigm in computer science and software engineering: A synthesis. *Knowl. Eng. Rev*, *18*.