Environment-Based Design of Software: an Agile Software Design Method

Alexandr Moroz

A Thesis in the

Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science (Quality Systems Engineering) at
Concordia Institute for Information Systems Engineering
Montreal, Quebec, Canada

March 2011

This is to certify that the thesis prepared

By: Alexandr Moroz

Entitled: Environment-Based Design of Software: an Agile Software Design Method

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Quality Systems Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examination committee

Dr. Z. Tian Chairman

Dr. S. Li Supervisor

Dr. Y. Zeng Supervisor

Dr. J. Bentahar Examiner

Dr. O. Ormandjieva External Examiner (CSE)

Approved by _____

Chair of the Department or Graduate Program Director

01 April 2011 _____

Dean of Faculty

ABSTRACT

Environment-Based Design of Software:

an Agile Software Design Method

Alexandr Moroz

The Environment-Based Design of Software (EBD-S) is a design method, representing the application of the Environment-Based Design (EBD) to agile software development. It compliments contemporary agile software development methods – Scrum and Feature-Driven Development (FDD) – by providing a light-weight and flexible framework for the architecture and design documentation, formalized design concept generation and effective system evolution control. Under the EBD-S umbrella, software requirements are categorized as functional, leading to the design of the system, and quality requirements, reflected in software architecture. EBD-S uses the component-bus-system-property approach for conflict identification and capturing the proto-architecture of the system in a graph structure. The design concept generation stage relies on a two-phase matrix-based problem decomposition approach, adjusted for non-binary dependency analysis, and using the heuristic partitioning analysis to find better design solutions. The change control mechanism of EBD-S permits effective monitoring and control of the software architecture evolution through the agile development cycle. The integration of EBD-S to the real-world Scrum development processes is demonstrated on the example of Telecom Expense Management software development. EBD-S application resulted in 25% project time saving due to more accurate estimations, higher code quality and lower error rate.

ACKNOWLEDGEMENTS

I would like to thank my supervisors Dr. Yong Zeng and Dr. Simon Li for their constant support, direction, and overwhelming patience, which enabled me to complete my thesis.

I acknowledge the work of Dr. Zeng, whose researches inspired me and outlined the direction of my explorations, and the constant guidance of Dr. Li, who helped me to structure the ideas.

On a special note I express my gratitude to my colleagues who made possible the application of the developed method to real-world project.

Last but not least, I thank my loving wife for her support and understanding throughout my studies.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Background and motivation of the research

Software systems of today are characterized by increasing complexity, distribution, heterogeneity and size. The software development tasks exhibit a high degree of variability and uncertainty.

A rationalized approach has dominated software development since its inception. Such an approach assumes that problems are fully specifiable, and that an optimal and predictable solution exists for every problem. It demands detailed capture and modeling of requirements, architecture and design early on, before significant effort is expended for system construction (Butler, Jones, Romanovsky, & Troubitsyna, 2006). Creating the interaction between software requirements, architecture and design is one of the most challenging problems in software engineering research. It requires not only elaboration of business requirements into flexible software architecture and design, but constant reconciliation of changes, introduced both in the requirements and the software system. Currently execution of this task is based mainly on the intuition and experience of engineers (Egyed & Grunbacher, 2002).

Appearance of the family of agile software methodologies in mid-90s (eXtreme Programming, Scrum, FDD, and others) addressed the high complexity of the software by introduction lightweight methods of fast software development, which “deal with unpredictability by relying on people and their creativity rather than on processes” (Nerur, Mahapatra, & Mangalaraj, 2005).

The nature of agile approaches differs from the traditional software development, and many attempts to bridge them were taken. Most of these attempts are based on the modification and formalization of the agile methods. Dyba and Dingsøyr showed in the study on the agile methods, that the formalized agile methods work in specific, narrow domains, and don't demonstrate real-world applicability in the wide variety of software development projects (Dyba & Dingsøyr, 2008). However, both traditional and agile software development methods are aimed to the same goal – facilitate and guide the software engineering process, and in many cases face the same problems.

The goal of software engineering process is to build a solution to an existing problem. To select or construct a solution to the problem, an engineer shall understand available options and existing limits. Thus, selection of basic technologies or creation of new ones is one of the most important problems which shall be solved in software development process. Next important step in software development is elaboration of system's principles, or selection of an approach to problem solving. This approach determines the structure of the software system, decomposition on the components and services, and the way how basic technologies are used.

Due to extremely high complexity of current software systems, solution of the problem leads to the modification of the initial concept (Zeng & Cheng, 1991). Change is an inevitable effect of the system development. Systems shall change in order to evolve; at the same time, the change can violate the architecture and design. System changes add an extra dimension to the complexity, which is especially true for the agile approaches, and change control in software engineering is frequently disconnected from the initial software concept.

Regardless of methodology used in development process, software engineers face four main challenges with concept development:

- Elaboration of adequate and feasible requirements
- Selection of system architecture
- Development of flexible software design
- Maintaining the requirements, architecture, design and code in concordance during the development lifecycle

With the experience in software requirements analysis and design elaboration domain, we have found out that we can take the advantage of the design theories, dealing with generation of design concepts. We can consider the requirements engineering as a design problem and use a design theory to generate several design concepts and select the best one. Taking this point of view, we can easily relate the software requirements with architecture and design in an unambiguous way to provide a method to control changes on any level of the system, from code to the requirements.

In this thesis we propose an approach to enhance agile software development methodologies, namely Scrum and FDD, with formal analytical toolset, aimed to address the main challenges of software engineering. The proposed approach, called Environment-Based Design of Software (EBD-S), is derived from Environment-Based Design (Zeng, 2004) and Non-binary Design Matrix for design concept elaboration and selection (Li, 2010), and uses Component-Bus-System-Property (CBSP) method (Medvidovic, Egyed, & Grunbacher, 2003) extinction to relate conceptual entities on the different levels to abstraction of the software system.

The Environment-Based Design provides a design model that is derived from the axiomatic theory of design modeling (Zeng, 2002). This model provides a unique view on the conceptual design problem:

- It defines the design problem in terms of the product environment rather than product functional structure;
- It generates design problems and solutions simultaneously, with the solutions affecting the perception of the problem.

The Design Structure Matrix (DSM) methodology emerged in early 1980s and demonstrated how graph theory can be used to analyze complex engineering projects (Steward, 1981). Steward showed how the sequence of design tasks could be represented as a network of interactions and how it can be mathematically analyzed as a system of equations. This representation of the design tasks allowed Steward to identify redundancies, inefficiencies, and other common problems analytically. DSM has been extended to the analysis of technical artifacts using the component-based DSM (Pimmler & Eppinger, 1994). We use the latest to compliment Environment-Based Design approach for generation and analysis of design solutions.

CBSP approach provides an intermediate model between requirements and architecture that helps to evolve the two models iteratively (Nuseibeh & Easterbrook, 2000). The intermediate CBSP model captures architectural decisions as an incomplete "proto-architecture" that prescribes further architectural development (Brandozzi & Perry, 2001). The CBSP approach also guides the selection of a suitable architectural style to be used as a basis for converting the proto-architectures into an actual implementation of

software system architecture. We use CBSP extinction to capture a software architecture view of the software design concept, generated by EBD and refined with help of DSM.

United under the umbrella of EBD-S, these approaches address the main problems of software concept development: they help to refine requirements, select the most appropriate architectural solution, build flexible design and maintain the control of the changes during the software development process.

1.2 Objectives

This thesis aims to provide an effective approach to the solution of software design problems, from requirements elaboration through architecture concept generation to detailed design development.

In the present thesis we plan to achieve the following objectives:

1) Introduce the Environment-Based Design method to software design problems.

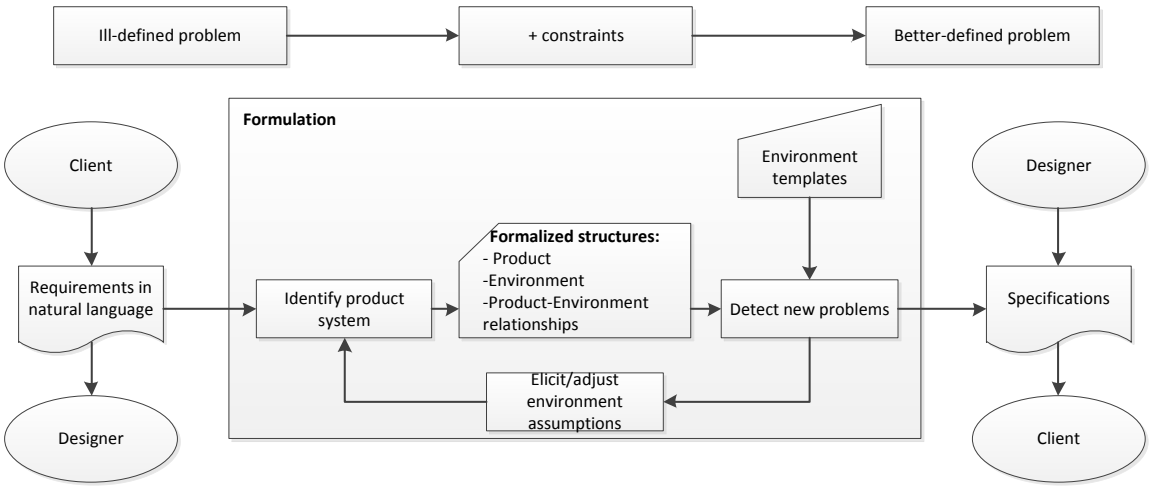

Figure 1 Problem formulation process in environment-based design

Software design problems, as well as product requirements, evolve along the design process (Chen & Zeng, 2006). Iterative formalization of the design problem, performed in the Environment-Based Design framework, allows adding more constraints to the domain to get a better definition of the problem for the next iteration. Requirements for software systems are well-known to be incomplete and ambiguous. EBD process helps to clarify the problem, define its scope and find applicable solutions.

2) Reinforce EBD method with Design Matrix application to generate and select the best design concepts.

Design Matrix (DM) is an effective tool to perform the analysis and management of complex systems. It helps to model, visualize and analyze the dependencies between the elements within the system (Li, 2010). Applied to the Environment-Based Design approach for software problems, it provides a way to derive suggestions for the best synthesis of the system.

3) Propose a method of relating software requirements, architecture and design within the concept for better change analysis and control.

Changes are inevitable during the software development process. They occur at any level of abstraction of the system – in the code, design, architecture and the requirements. In order to understand the impact of these changes we apply a graph-based method to link the relations between software requirements, architecture and design. It is based on CBSP (Component-Bus-System-Property) methodology (Medvidovic et al., 2003), adjusted for software problems. The graph compliments the DM, and allows agile management of the software changes during the whole software development lifecycle.

4) Construct a framework for solving the software design problems – Environment-Based Design of Software.

Software development process requires application of different best-practices for better control and effectiveness. We introduce a methodology for solving the design problems in agile software development, which relies on Environment-Based Design, reinforced with Design Structure Matrix problem analysis and CBSP change control. It is called Environment-Based Design of Software, and it provides guidance for the agile software developers from the beginning of software process to the end of its lifecycle.

5) Demonstrate the applicability of the Environment-Based Design of Software on a real-world business case.

Environment-Based Design of Software was elaborated not solely on the basis of scientific research – it was applied to the management of real software development processes, and was refined according to the observed results. Our aim is to demonstrate how EBD-S application helped to perform full development of an enterprise software solution, from concept generation to the integration and delivery, in a telecommunication expense management domain. The developed product relies on a large legacy system, but represents a completely new line of products, build with newest Rich Internet Application technologies. In Chapter 5 we give a description of the detailed process and solutions found, as well as of the control of deviations from the original design.

1.3 Challenges

Here we describe the main challenges that we faced during the development of the Environment-Based Design of Software approach:

1) Discover and define the main problems of software development process

One of the main challenges of the software development teams is to provide a clear and unambiguous method to understand the client's problem, find a solution and communicate it back to the client to verify it (Wiegers, 2003). The nature of agile software development implies that many solutions can be verified only after they are built (or prototyped). Concept refinement in software development is always a time-consuming process, and it is better to make and correct all the errors in assumptions on this stage. Usually the domain of possible solutions is quite wide, so it is hard to find the best fit with the scope of the problem (Chung, Nixon, Yu, & Mylopoulos, 2000).

Control of the software system evolution is another challenge that software developers face. Many unplanned changes happen to the system, and software developers strive to find a method to analyze the impact of the changes in the real time. There are automatic tools that can demonstrate the impact on two lowest levels of abstraction, code and UML design (Medvidovic et al., 2003). But the relation of these changes with architecture of the system and requirements is usually left to be determined by people. In small projects it is not an issue; but as soon as team size exceeds 2 developers, the issues of evolution control arise (Paetsch, Eberlein, & Maurer, 2003).

These problems of software process motivated us to build a scientific method for their resolution – EBD-S. This method is described in the Chapter 4 of the thesis.

2) Analyze the existing methods to solve these problems

To elaborate a new method to cope with the given problems it is important to understand the existing methods and analyze their pros and cons. We performed a comprehensive literature review of the software methodologies and best-practises in Chapter 2. It allowed us to focus on the problems which are not covered by existing approaches.

3) Enrich the Environment-Based Design Theory with Design Matrix approach

The idea of Environment-Based Design was developed by Dr. Yong Zeng in 2004 based on Axiomatic Theory of Design Modeling (Zeng, 2002). It includes three main stages: environment analysis, conflict identification and concept generation (Zeng, 2004). Design Matrix approach allows analyzing the system and verifying the applicability of the solution concepts to the real problem. Together these approaches form a unique tool for problem solving. We provided a literature review of design theories and their comparison to understand the advantages of the selected approach.

4) Build up a problem solving framework that can address different software concept generation problems

Environment-Based Design and Design Matrix approach form a powerful tool to generate software concepts and select the most adequate solution, as well as to manage the effects of change. However, they lack an ability to track the history of decisions and changes, which is an extremely important part of evolution control. To provide such ability, we reinforce the EBD-DM approach with a simple visual tool that captures and relate the requirements-architecture-design artefacts. This tool is called CBSP (Component-Bus-

System-Property) approach, and it complements the two presented methods in software process. In fact, this approach was added on the basis of feedback from real-world implementation of the EBD-DM. Together these three approaches form a solid basis for resolution of main software development challenges.

1.4 Contribution

The objective of this thesis is to elaborate an agile software design methodology that combines the Environment-Based Design method with Design Matrix decomposition approach for software architecture selection and CBSP approach to control the software evolution. The advantages of the proposed design method are the formalization of software concept generation, justification of the software architecture selection and traceability of the design decisions for the evolution control. The contribution of this thesis can be summarized as following:

- Many existing software methodologies focus on fast-adapting design and development approaches, but they sacrifice profound analysis for the speed of implementation. We propose to integrate lightweight and effective analytical methods to the software process, which allow planning and controlling the software architecture evolution in a long run. With EBD-S methodology we focused on finding the right balance between formalized analysis and rapid development that ensures high quality software and reduces the risk of architectural lock-in.
- The application of EBD-S approach in the real world software development project allowed us to tune some parts of it, and underlined the gaps between

theory and practice that we were able to close within next development iterations. The case study, described in Chapter 5, reflects this process.

- The Environment-Based Design and Design Matrix problem-solving had been proven and widely used in many engineering fields. This thesis extended the application of these methods to the software design and development process.

1.5 Organization of the thesis

The thesis is organized as follows:

Chapter 1: Introduction of the thesis. This chapter presents the motivation, scope, objectives, challenges of the research, contribution of the thesis and its organization.

Chapter 2: Literature review. This chapter defines the key aspects of the software architecture selection, described in existing researches

Chapter 3: Theoretical foundations review. It includes the review of the Environment-Based Design (EBD) methodology, of the Design Matrix problem decomposition and analysis approach, and CBSP methodology review.

Chapter 4: Environment-Based Design of Software (EBD-S) approach. This reusable model unites EDB and Design Matrix principles with CBSP approach for robust software architecture development and provides an instrument for software evolution control.

Chapter 5: Case Study. This chapter overviews the EBD-S approach by demonstrating its application in a real-world example.

Chapter 6: Conclusion. This chapter summarizes the thesis material, and provides thoughts on the future work.

2. Literature review

2.1 Overview

This chapter contains the review of the literature related to our research. This review encompasses two main aspects, which are the key elements of the research: system design theories in general and software process and design methodologies, focusing on the most popular agile software development methods.

2.2 System design theories

2.2.1 The nature of design theories

Design is "the use of scientific principles, technical information and imagination in the definition of a structure, machine or system to perform pre-specified functions with the maximum economy and efficiency" (Fielden, 1975). Design is central topic within engineering.

A design theory is a prescriptive theory based on theoretical underpinnings which says how a design process can be carried out in a way which is both effective and feasible. (Walls, Widmeyer, & Sawy, 1992).

The primary difference between scientific theories and design theories is in how they deal with goals. Goals are meaningless in natural science theories, social science theories may deal with goals as objects of study. The purpose of a design theory is to support the achievement of goals. Goal orientation is the key element required in a design theory which is missing in a science theory (Walls et al., 1992). The following statements characterize design theories:

(1) Design theories must deal with goals as contingencies. While goals are extrinsic to explanatory and predictive theories, they are intrinsic to a design theory.

(2) A design theory can never involve pure explanation or prediction. If it explains, it explains what properties an artifact should have. If it predicts, it predicts that an artifact will achieve its goals to the extent that is possesses prescribed by the theory.

(3) Design theories are prescriptive. They integrate the explanatory, predictive and normative aspects in "can" and "will" design paths that realize more effective design and use.

(4) Design theories are composite theories which encompass kernel theories from natural science, social science and mathematics.

(5) While explanatory theories tell "what is", predictive theories tell "what will be", and normative theories tell "what should be", design theories tell "how to / because".

(6) Design theories show how explanatory, predictive, or normative theories can be put to practical use.

(7) Design theories are theories of procedural rationality (Simon, 1996).

If it is to be a good theory (Nagel, 1961), a design theory must be subject to empirical refutation. An assertion that possession of a particular set of attributes will enable an artifact to meet its goals can be verified by building and testing the artifact. Prototype construction is a major aspect of design theory research.

2.2.2 A formal definition of a design theory

A design theory must have two aspects – one dealing with the product and another dealing with the process of design. These aspects cannot be independent, since the design process must yield the product to be designed (Suh, 1990).

The first component of a design theory dealing with the product of design is a set of meta-requirements which describe the class of goals to which the theory applies. The second component is a meta-design describing a class of artifacts hypothesized to meet the meta-requirements. The third component is a set of kernel theories from natural or social sciences, or from other design theories which govern design requirements. The final component is a set of testable design process hypotheses which can be used to verify whether the meta-design satisfies the meta-requirements.

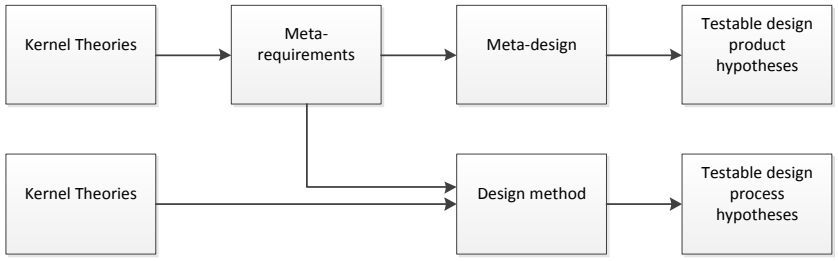

Figure 2 Components of a design theory according to Walls (2001)

The second aspect of a design theory deals with the design process. The first component of this aspect is a design method, which describes procedure for artifact construction. The second is a set of kernel theories governing the design process itself. These kernel theories may be different from those associated with the design product. The final component is a set of testable design process hypotheses which can be used to verify whether the design method results in an artifact which is consistent with meta-design (Walls et al., 2001).

2.2.3 Classification of design theories and methodologies

The field of Design Theory and Methodology (DTM) is a rich collection of findings and understandings resulting from studies on how we design (rather than what we design).

While perhaps the ultimate goal of the DTM research would be to obtain a general and abstract (thus universal) theory about design, there can be theories only general but still concrete or theories abstract but individual as an intermediate state of progress. Therefore, DTM can roughly be categorized into four categories along two axes; one is "concrete vs. abstract" and the other is "individual vs. general" (Tomiyama, Gu, Jin, Lutters, Kind, & Kimura, 2009).

Table 1 DTM categorization by Tomiyama et al.

	General	**Individual**
Abstract	*Design theories* (GDT, UDT)	*Math-based methods* (Axiomatic Design, Optimization, Taguchi Method)
Concrete	*Design methodologies* (Adaptable Design, Integrated Product Development, TRIZ, etc.) *Methodologies to achieve concrete goals* (Axiomatic Design, Design for X, DSM, FMEA, QFD, Total Design of Pugh) *Process methodologies* (DSM, Concurrent Engineering)	*Design methods*

- Concrete and Individual: By grouping records of individual design cases belonging to a specific product class and by extracting commonalities among them, we obtain ''design methods'' for this particular product class.
- Concrete and General: DTM in this category aims at concrete descriptions but applicable to a wide variety of products. This type of DTM can be obtained by generalizing design methods. This generalization is possible by focusing on particular characteristics common to different types of products. By focusing on functions, we

obtain so-called prescriptive design methodologies such as Pahl and Beitz (1988). Similarly, by focusing on various concrete design goals within design, we obtain DfX (Design for X). If we focus only on design process management, we obtain process technologies to control and manage product development processes, such as concurrent engineering.

- Abstract and Individual: By abstracting design methods, we obtain this type of DTM applicable (only) to a specific class of product design. Abstraction often takes a form of mathematics, meaning design solutions can be obtained algorithmically with computation. DTM in this category includes, for example, a variety of computational methods for optimization and engineering computation. Note that these computational methods do not include modeling systems (such as geometric modeling), because they are ‘‘modeling frameworks’’ rather than ‘‘design methods’’. However, some DTM methods describe design at such an abstract level that they are applicable to a certain class of design targeting specific goals (for instance, Taguchi method for quality design (Steward, 1981)).
- Abstract and General: Design Theories about design processes, activities, and knowledge. For example, General Design Theory (GDT) by Yoshikawa explains design as knowledge operations (set operations).

2.2.4 Generic design process

Design methodology begins with a design process model that can be used to develop product specifications. In all cases it is apparent that the development process is commonly regarded as a logical sequence of phases in which tasks are completed. Although differences exist in for instance the scope of the models and the use of

iterations, all models show a similar way of describing a progression through a sequence of events (Yoshikawa, 1981).

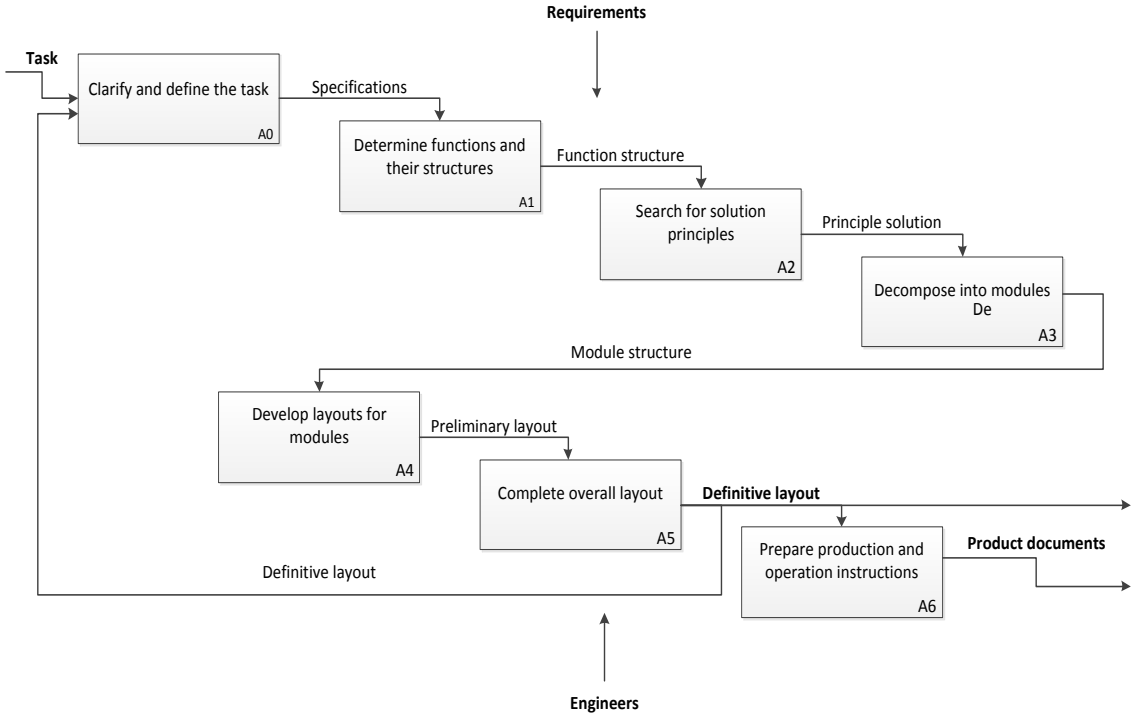

Figure 3 General Design Theory process (Yoshikawa, 1981)

2.2.5 Evaluation of major design theories and methodologies

Tomiyama et al. performed a deep analysis of the contemporary Design Theories and Methodologies in the following domains: research, education and industry (Tomiyama et al., 2009). Their findings are summarized in Table 2.

Table 2 DTM widely taught and widely used (Tomiyama et al., 2009)

	General	Individual
Abstract	*Design theories* – Widely taught	*Math-based methods* – widely taught and used
Concrete	*Design methodologies* – widely taught *Methodologies to achieve concrete goals* – widely taught and used *Process methodologies* – widely taught and used	*Design methods* – widely taught and used

Design theories (GDT, UDT) and design methodologies (Adaptable Design, Integrated Product Development, TRIZ) are widely taught, but rarely used in industry. They mostly focus on the embodiment design rather than on how to achieve concrete performance goals (cost, quality, time). For routine design, which represents the vast majority of the design cases in industries, these aspects are more important than innovation in functional design. However, increasingly industry started to realize the importance of innovative design and for this reason TRIZ as a method to enhance innovation capabilities is popular among industry (Tomiyama et al., 2009). In surveying various DTM, Tomiyama et al. found out that many of them do not reflect modern product development activities, especially lacking support of the following:

- Complex multi-disciplinary product development
- Further advances in digital and virtual engineering for better collaboration
- Globalization in product development

Among the most widely design methods, used in industry, only one found really wide adoption in industry – Axiomatic Design Theory. Generally it is applicable for all kinds of design activities, including complex system design; it has large number and wide range of examples to follow; and it can be an effective tool in analysis in addition to design activities.

2.2.6 Axiomatic design theory

Axiomatic design theory and method have been widely reported in CIRP (The International Academy for Production Engineering) community. Axiomatic design states the best design solution fulfills two axioms:

1. Maximum independence of the functional elements.
2. Minimum information content.

Compliance with the first axiom assures that designs will be adjustable, controllable and will avoid unintended consequences. Compliance with the second axiom assures that the design will be robust with a maximum probability of success. There are also theorems and corollaries associated with the axioms (Tomiyama et al., 2009).

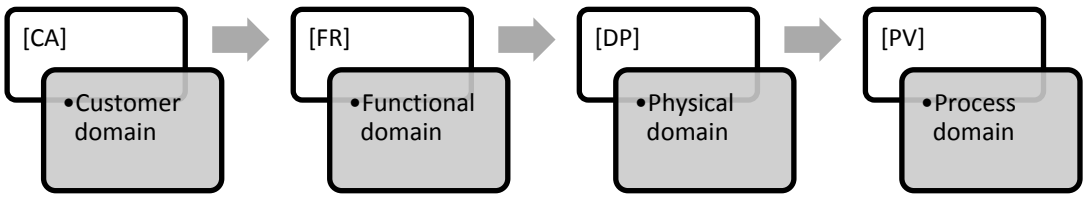

Figure 4 Four domains in axiomatic design

Application of Axiomatic Design consists of three elements each with two parts. The parts of the first element are the axioms. In order to apply the axioms systematically through the design, a structure for the design elements is required. The structure is the second element and its two parts are a horizontal decomposition into domains of customer, functional, physical and process domains as shown in Figure 4, and a vertical decomposition in a hierarchy from general to specific aspects of the design. The third element is the process. It is composed of zigzagging decomposition to create the design hierarchies in the domains from the top down by first developing the functional requirements (FRs) from the customer attributes (CAs) in the customer domain then selecting the Design Parameters (DPs) in the physical domain to satisfy the FRs and the corresponding Process Variables (PVs) in the process domain to create the DPs (Suh,

1990). In order to check for compliance with Axiom 1, the independence axiom, Suh defines a design matrix ([A]) which is used to display which DPs influence which FRs:

$$[FR] = [A][DP] \quad (1)$$

The desirable design is uncoupled where matrix is diagonal. If the matrix is triangular it is a decoupled design, and there is a fixed order of adjustment of the DPs to satisfy the FRs. Otherwise, the design is a coupled design which should be avoided.

Axiomatic design theory has been used in a wide range of industrial applications ranging from software design to products and manufacturing systems design (Tomiyama et al., 2009).

2.3 Software design methodologies

2.3.1 Software process

Software process is the term given to the organization and management of software development activities. Generic software development process shares the same principles with engineering process in all industries: from concept through design to the final product. It is iterative, as in the majority of the industries. Project management directs all the stages of the process.

Each stage produces certain outcome. To describe the generic software process, we need to differentiate its sub-processes by their outcomes, and group them in the structure. The software sub-process classification, presented below, is based on the Microsoft software development guideline (Wiegers, 2003):

- ***Requirements elicitation*** – it is the process of building the concept of the software. The main outcomes are Vision and Scope document and Use-Cases.
- ***Software specifications development*** – the specification of the design and architecture of the future product. Main deliverables are:
    - *Software architecture* – is the complex of basic technologies of the software solution, and a set of design patterns, united in the framework or core.
    - *Software design* – is the segmentation of the functionality by the components, modules or classes and their relationships to each other and the environment.
    - *Detailed software design* – represents the algorithms and data structures, which will be used by the developers during coding.
    - All the requirements (business and software) are usually united in one document, called *Software Requirements Specifications*. This document represents the deliverable of the analysis and design stages, and serves as a base for following stages.
- ***Software implementation*** – the process of coding performed according to the software requirements. Includes many iterative stages, internal quality verification and code refactoring. The main deliverable is the software product itself.
- ***Verification and validation*** – the process of internal quality verification; usually goes in parallel with software implementation process. The main goal is to eliminate the defects (non-conformities to the specifications) of the software. The deliverables are the verified software and the list of "known errors" – non-critical issues, which are not planned to be fixed in current version.

- ***Integration and delivery*** – creation of the final software package, automated installers or integration to the work environment. The deliverable is the final software product.
- ***Maintenance*** – supporting and troubleshooting the operations of the software. This activity is important when the developers need to get user's feedback and improve the product.

There are several approaches to software development. They give different recommendations for the length of iterations, order of the stages, and involvement of the team members. But all of them are in agreement that these processes are essential in software development.

2.3.2 Software architecture and design

Software architecture has emerged as a crucial part of the design process. It encompasses the structures of large software systems. The architectural view of a system is "abstract, distilling away details of implementation, algorithm, and data representation and concentrating on the behavior and interaction of "black box" elements" (Shaw & Garlan, 1996). Software architecture is developed as the first step toward designing a system that has a collection of desired properties. Shaw and Garlan defined what constitutes software architecture in more details:

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them (Shaw & Garlan, 1996).*

First, architecture defines the key software elements and embodies the information about how the elements relate to each other. This means that it specifically omits certain information about elements that does not pertain to their interaction. Thus, an architecture is foremost an abstraction of a system that suppresses details of elements that do not affect how they use, are used by, relate to, or interact with other elements.

Second, the definition makes clear that systems can and do comprise more than one structure and that no one structure can irrefutably claim to be the architecture. For example, all nontrivial projects are partitioned into implementation units. This is one kind of structure often used to describe a system. Other structures are much more focused on the way the elements interact with each other at runtime to carry out the system's function.

Third, the definition implies that every computing system with software has a software architecture because every system can be shown to comprise elements and the relations among them.

Fourth, the behavior of each element is part of the architecture insofar as that behavior can be observed or discerned from the point of view of another element. Such behavior is what allows elements to interact with each other, which is clearly part of the architecture.

A set of business and technical decisions define the software architecture. These business and technical decisions are strongly related on the environment in which the architecture is required to perform. In any development effort, the requirements make explicit some of the desired properties of the final system. Not all requirements are concerned directly with those properties; a development process or the use of a particular tool may be

mandated by them (Chen, Yao, Lin, Zeng, & Eberlein, 2007). Weigers identified the key influences to software architecture (Wiegers, 2003):

- *Business requirements*, expressed by stakeholders, is the main source of influence to architecture
- *Developing organization* frequently reshapes the architecture according to current investments in certain assets, long-term strategies, and organization structure
- *Technical environment* usually sets the limitations of the software system and defines the available selection of the basic technologies
- *Background and experience of architects* inevitably affects the architecture

2.3.3 Software design: commonality and variability

The question of what is the nature of software design is important for understanding its principles. Coplien – from his "Multi-Paradigm Design for C++" – provides an answer:

> *When we think abstractly, we emphasize what is common while suppressing detail. A good software abstraction requires that we understand the problem well enough in all of its breadth to know what is common across related items of interest and to know what details vary from item to item. The items of interest are collectively called a family, and families—rather than individual applications—are the scope of architecture and design. We can use the commonality/variability model regardless of whether family members are modules, classes, functions, processes or types; it works for any paradigm. Commonality and variability are at the heart of most design techniques.* (Coplien, 1999).

Finding the commonalities and variabilities within a system, and expressing them, forms the heart of design. Commonalities are often the parts that are difficult to explicitly identify, not because we don't recognize them, but because they're so easily and intuitively recognizable it is tough to spot them.

Variability can come in two basic forms, one of which is easy to recognize and the other much more difficult. Positive variability is when the variability occurs in the form of adding to the basic commonality. For example, an abstraction desired is that of a message, such as a SOAP message or e-mail. If we decide that a Message type has a header and body, and leave different kinds of messages to use that as the commonality, then a positive variability on this is a message that carries a particular value in its header, perhaps the date/time it was sent. This is usually easily captured in language constructs—in the object-oriented paradigm, for example, it is relatively trivial to create a Message subclass that adds the support for date/time sent.

Negative variability, however, is much trickier. As might be inferred, a negative variability removes or contradicts some facet of the commonality – a Message that has a header but doesn't have a body (such as an acknowledgement message used by the messaging infrastructure) is a form of negative variability. And capturing this in a language construct is problematic – most of object-oriented languages don't have a facility to remove a member declared in a base class.

Thus, the goal of the software design is to maintain the right level of abstraction by encompassing commonalities, supporting positive and avoiding negative variability (Neward, 2010).

2.3.4 Nature of change in software development

Software development process is an iterative activity. Stages of development are often interwoven and affect each other. Change is the key notion in the understanding of these interactions.

Change in the software development is the modification of some important aspects of the development, which cause the modification of expected result. Changes are inevitable in all the development processes. James F. Peters and Sheela Ramanna proposed following classification of the changes in the software development process: external and internal from the system perspective; planned and unplanned from the process perspective (Peters & Ramanna, 2003).

External changes are caused by the client's requests; *internal* ones are the result of the internal decisions and optimizations. The necessity of the change must be determined and analyzed before it goes to the implementation. Existing approach of determination is called risk-value-cost analysis and consists in determination of risks and costs of the tasks and activities with comparison of benefits and drawbacks.

Planned changes are usually reflected in all levels of the requirements, from business to detailed design, and approved by the software architect / designer. *Unplanned changes* are introduced by mistake or personal decision of the developer, and often cause the architecture or design decay.

Consistency in the software change control process permits to plan software evolution during the whole software life-cycle (Medvidovic et al., 2003).

2.3.5 Background of agile methodologies

In the last 30 years a large number of different approaches to software development have been introduced, of which only few have survived to be used today. The nature of software development results in the fact that traditional information systems development methodologies “are treated primarily as a necessary fiction to present an image of control or to provide a symbolic status” (Nandhakumar & Avison, 1999). More than that, several researchers and software practitioners in early 2000s agreed that traditional methods “provide normative guidance to utopian development situation” (Truex, 2000). As a result, industrial software developers have become skeptical about new solutions that are difficult to grasp and thus remain not used (Wiegers, 2003). This was the background for agile methodologies appearance.

Agile – software development methods attempt to offer once again an answer to the business community asking for lighter weight along with faster and nimbler software development processes. Agile proponents claim that the focal aspects of light and agile methods are simplicity and speed, as opposed to deep formalization and complexity of traditional design methodologies. The principles of agile development are expressed in Agile Software Development Manifesto published by a group of software practitioners and consultants in (Beck, 2001). The focal values honored by this manifesto are:

- Individuals and interactions over process and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Highsmith and Cockburn (2001) report that the changing environment in software business seriously affects the software development process. To satisfy the customers at the time of delivery has taken precedence over satisfying them at the moment of project initiation. That calls for procedures dealing with how to better handle inevitable changes throughout the software development cycle (Cockburn & Highsmith, 2001). It is claimed that agile methods are designed to:

- Produce the first delivery in weeks, to get rapid feedback
- Invent simple solutions, so there is less to change and making changes easier
- Improve design quality continually, making next iteration less costly
- Test constantly, for earlier and less expensive defect detection

All this forms the major difference between agile and traditional design. Agile methods assume largely emergent, rapidly changing requirements and agile design is worked out for current requirements. While traditional design methodologies, applied to software, work with knowable early and largely stable requirements, with architecture designed for current and foreseeable requirements.

2.3.6 Generic agile software process

Miller gives the following characteristics to agile software processes from the fast delivery point of view, which allow shortening the life-cycle of projects (Miller, 2001):

- Modularity on development process level
- Iterative with short cycles enabling fast verifications and corrections
- Time-bound with iteration cycles from one to four weeks
- Parsimony in development process removes all unnecessary activities

- Adaptive with possible emergent new risks
- Incremental process approach that allows functioning application building in small steps
- Convergent and incremental approach minimizes the risks
- People-oriented, agile process favor people over process and technology
- Collaborative and communicative working style

Since the agile software development principles differ from the traditional design approaches, its process is also very different. Figure 5 represents the high-level view on the agile software development process (Robertson & Robertson, 2007). All product requirements and user stories are collected under the name of Product Backlog. In each iteration, which usually lasts from 1 week to 1 month, a sprint backlog – sub-set of requirements – is selected for implementation. Then a very light-weight process of design-implementation-testing-documentation is performed. The result is a working application, with a small number of new (or updated) features.

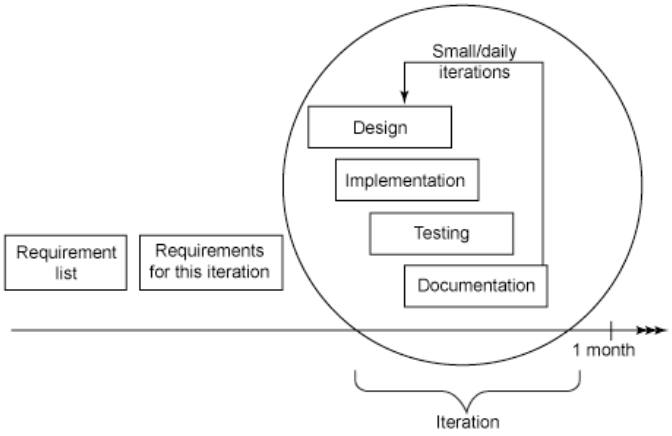

Figure 5 Generic agile software development process (Robertson & Robertson, 2007)

2.3.7 Agile software development challenges

The benefits associated with agile software development methods are obtainable only if these methods are correctly used in production process. While agile approaches concur with the traditional software development practice, they are not all suitable for all phases in the software development life-cycle – the results of the study (Abrahamsson, Salo, Ronkainen, & Warsta, 2002) are summarized in Figure 6.

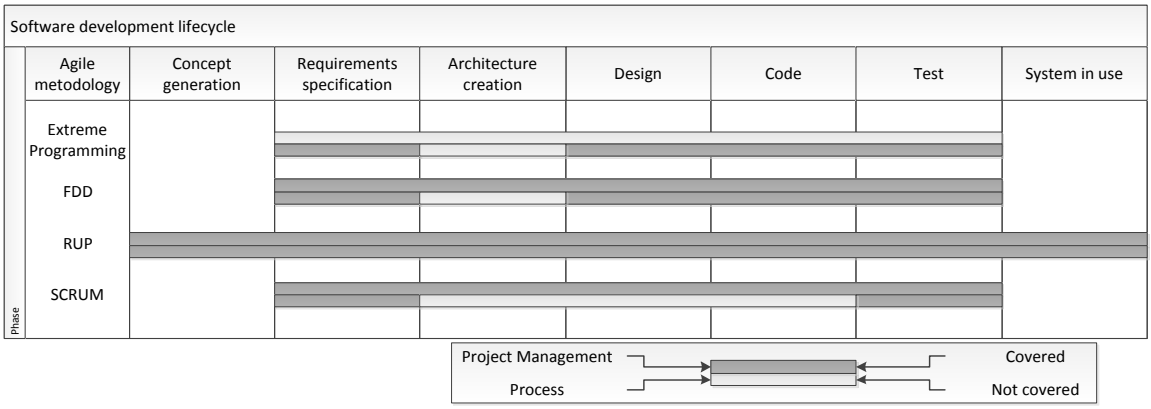

Figure 6 Software development life-cycle support

According to the study, only Rational Unified Process covers all the aspects of software development, both from project management and process viewpoints. Other popular agile approaches do not support concept generation stage, and do not cover change control during the system in use evolution. Process of software architecture creation is not determined in XP, FDD and Scrum, while the last lacks process description of design and coding stages as well (Abrahamsson et al., 2002).

Another type of challenges, generic to all approaches, is related to requirements engineering. Eliciting precise and comprehensive product requirements from customers is of critical importance for the success of product development (Wang & Zeng, 2008).

However, many agile methods advocate the development of code without waiting for formal requirements analysis and design phases. Based on constant feedback from the various stakeholders, requirements emerge throughout the development process. Most agile organizations shun formal documentation of specifications. Instead, they use simple techniques such as user stories to define high-level requirements and rely on the heavy communication with the customer. For projects that "can't achieve high-quality interaction, this approach poses risks such as requirements inadequately developed or, worse, wrong" (Lan & Ramesh, 2008).

The traditional requirements engineering process phases – elicitation, analysis, and validation are present in all agile processes. The techniques used vary in the different agile approaches and the phases are not as clearly separated as in the traditional RE process. They are also repeated iteratively which makes it harder to distinguish between the phases. More than that, continuous reprioritization of the requirements leads to instability. The techniques used in the agile development processes are sometimes described vaguely and the actual implementation is left to the developers. This is a result of the emphasis on highly skilled people: "good" developers will do the "right thing" (Paetsch et al., 2003). As all agile approaches include at least a minimum of documentation, it is the responsibility of the development team to ensure enough documentation is available for future maintenance. It either slows down the development, or leads to the lack of documentation (Lan & Ramesh, 2008).

3. Theoretical foundations review

3.1 Environment-Based Design

3.1.1 Introduction to Axiomatic Theory of Design Modeling

There are two basic approaches in representing the design problem: bottom-up and top-down. The first one is based on generalization of the design problem structure by analysis of engineering design activities and case studies. The top-down approach works the other way around – it tries to derive the design problem structure from high-level principles.

The axiomatic approach is one of the most important tools in top-down design problem representation. It addresses the general design models and problems, and lets the concrete design problem models to be deducted. It is based on the set of axioms, which are statements that are self-evident truths, and uses mathematical structure to consistently derive the invariant structure for design problem representation.

Axiomatic Theory of Design Modeling (Zeng, 2002) provides a logical tool for representing and reasoning about object structures. It uses three basic axioms: universe, object and relation. Axiomatic Theory of Design Modeling differs from set theory, where concrete and abstract objects are distinguished by set and element. In this theory the only abstract concept is the universe. Here are the definitions of these basic axioms:

[Definition 1] The universe (U) is the whole body of things and phenomena observed or postulated.

[Definition 2] An object (denoted by capital letters) is anything that can be observed or postulated in the universe.

[Definition 3] A relation (~) is an aspect or quality that connects two or more objects as being or belonging or working together or as being of the same kind. A relation can be a property that holds between an ordered pair of objects.

$$R = A \sim B, \qquad \exists A, B, \exists R \tag{2}$$

where A and B are objects, A ~ B is read as "A related to B", and R is the relation from A to B. It is important to note that relation is also an object. Based on definitions 1-3, the following axioms are introduced:

[Axiom 1] Everything in the universe is an object.

[Axiom 2] Every object in universe interacts with other objects.

The characteristics of relations play a critical role in the axiomatic theory of design modeling. We need to define a group of basic relations to capture the nature of object representation. They will be used to establish new types of relations in the theory. We need two basic relations – the corollaries of the theory:

[Corollary 1] Every object in the universe includes other objects.

$$A \supseteq B, \qquad \forall A \, \exists B \tag{3}$$

B is called a sub-object of A. The symbol $\supseteq$ is inclusion relation. The inclusion relation is transitive and idempotent but not commutative. Other operations such as $\subseteq$, =, $\cup$, and $\cap$ are also defined based on this corollary (Zeng, 2002).

[Corollary 2] Every object in the universe interacts with other objects.

$$C = A \otimes B, \qquad \forall A, B \; \exists C \tag{4}$$

C is called the interaction of A on B. The symbol $\otimes$ represents interaction relation. Interaction relation is idempotent but not transitive or associative.

Based on the above two corollaries, the structure operation is established. It provides the aggregation mechanism for representing the object evolution in the design process.

[Definition 4] Structure operation, denoted by $\oplus$, is defined by the union of an object and the relation of the object to itself.

$$\oplus O = O \cup (O \oplus O), \tag{5}$$

where $\oplus$O is the structure of object O. The structure operation provides the aggregation mechanism for representation of object evolution in the design process (Zeng, 2002).

3.1.2 Environment-Based Design process

The notion of Environment-Based Design was introduced by Dr. Yong Zeng in 2004. This design methodology is based on his Axiomatic Theory of Design Modeling. While traditional axiomatic design theories are based on the generic design process, the environment-based design process encompasses three domains: environment analysis, identification of conflicts and concept generation. These domains are processed iteratively and progressively to elaborate design requirements and design solutions. The generic EBD process is shown on the Figure 7 using IDEF0 notation.

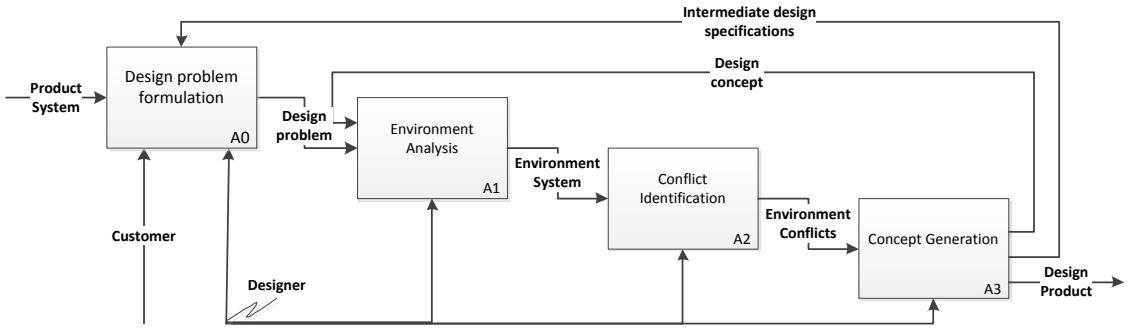

Figure 7 Environment-Based Design process

In order to understand the EBD process, we need to provide definitions for the main components of the EBD.

[Definition 5] A product system is the structure of an object (Ω) including both a product (S) and its environment (E).

The product can be a machine, a software package, a process, an idea, etc. Everything except the product itself can be seen as its environment. Let

$$\Omega = E \cup S, \qquad \forall E, S[E \cap S = \Phi], \tag{6}$$

where Φ is the object that is included in any object.

Based on the definition of structure operation, the product system ($\oplus\Omega$) can then be expanded as follows:

$$\oplus\Omega = \oplus(E \cup S) = (\oplus E) \cup (\oplus S) \cup (E \otimes S) \cup (S \otimes E), \tag{7}$$

where $\oplus E$ and $\oplus S$ are structures of the environment and product, respectively; $E \otimes S$ and $S \otimes E$ are the interactions between environment and product. A product system is illustrated in Figure 8.

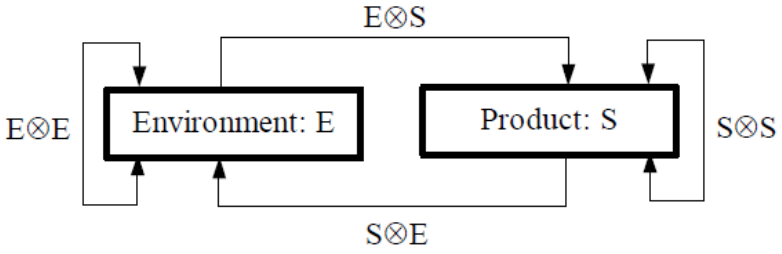

Figure 8 Product system (Zeng, 2004)

The definition of the product system gives a description how a product exists in the universe. The product system is composed of the product (object / collection of objects), environment (all other objects in the universe) and interactions between product and environment, between elements of the environment and between the elements of the product. It is important to separate the interactions between product and the environment, because they define the place and the behavior of the product in the system. Thus we introduce a new definition:

[Definition 6] Product boundary, denoted by B, is the collection of interactions between a product and its environment.

$$\mathrm{B} = (\mathrm{E}\otimes\mathrm{S})\cup(\mathrm{S}\otimes\mathrm{E}). \tag{8}$$

We can define two types of product boundaries: structural and physical. The structural boundary (B^s) is the shared physical structure between a product and its environment. The physical interactions include actions (B^a) of the environment on the product and responses (B^r) of the product to the environment. Therefore, product boundary can be represented as

$$\mathrm{B} = B^s \cup B^a \cup B^r, \qquad \forall B^s, B^a, B^r \tag{9}$$

Based on the definition of the product system, we can formally define a design problem.

3.1.3 EBD process: design problem formulation

[Definition 7] A design problem can be literally defined as a request to design something that meets a set of descriptions of the request. Based on the axiomatic theory of design modeling, both "something" and "descriptions of the request" can be seen as objects and can be further seen as product systems in the context of formulating design problem. Thus a design problem, denoted by P^d, can be formally represented as

$$\mathrm{P}^d = \lambda(\oplus\Omega_O, \oplus\Omega_S), \tag{10}$$

where $\oplus\Omega_O$ ($\Omega_O = \mathrm{E}_O \cup \mathrm{S}_O, \mathrm{E}_O \cap \mathrm{S}_O = \Phi$) can be seen as the descriptions of a request for the design, $\oplus\Omega_S$ ($\Omega_S = \mathrm{E}_S \cup \mathrm{S}_S, \mathrm{E}_S \cap \mathrm{S}_S = \Phi$) is something to be designed, and λ is the "inclusion" relation ($\supseteq$) implying that $\oplus\Omega_S$ will be a part of $\oplus\Omega_O$ so that the designed product will meet the descriptions of the design.

At the beginning of design process, $\oplus\Omega_S$ is unknown and $\oplus\Omega_O$ is the only thing defined. The true value of P^d is undetermined, which means the request is yet to be met.

According to (6) and (7), we have

$$\begin{aligned} \oplus\Omega_O &= (\oplus\mathrm{E}_O) \cup (\oplus\mathrm{S}_O) \cup \mathrm{B}_O \\ \oplus\Omega_S &= (\oplus\mathrm{E}_S) \cup (\oplus\mathrm{S}_S) \cup \mathrm{B}_S \end{aligned} \tag{11}$$

Since $E_i \cap S_j = \Phi, \forall i, j = 0, s$, we have

$$\mathrm{P}^d = \lambda(\oplus\mathrm{E}_O, \oplus E_S) \wedge \lambda(\oplus\mathrm{S}_O, \oplus S_S) \wedge \lambda(B_O, B_S) \tag{12}$$

where $\wedge$ denotes logical “and”.

Substitute (9) into (12), we have

$$\mathrm{P}^d = \lambda(\oplus\mathrm{E}_O, \oplus E_S) \wedge \lambda(\oplus\mathrm{S}_O, \oplus S_S) \wedge \lambda(B_0^s, B_S^s) \wedge \lambda(B_0^a, B_S^a) \wedge \lambda(B_0^r, B_S^r) \tag{13}$$

Equation (13) can be organized into three parts:

1) $\lambda(\oplus \mathrm{E}_O, \oplus E_S)$, which defines the requirements on the product environment
2) $\lambda(\oplus \mathrm{S}_O, \oplus S_S) \wedge \lambda(B_0^s, B_S^s)$, which denotes direct constraints on the product
3) $\lambda(B_0^a, B_S^a) \wedge \lambda(B_0^r, B_S^r)$, which defines direct constraints on actions/responses

Therefore, the following theorem is derived:

[Theorem 1] **Structure of Design Problem**. A design problem is implied in a product system and composed of three parts: the environment in which the designed product is expected to work, the requirements on product structure, and the requirements on performance of the design product.

Table 3 Structure of design problem

Design Problem: P^d	
Product Environment	$\lambda(\oplus \mathrm{E}_O, \oplus E_S)$
Performance Requirements	$\lambda(\oplus \mathrm{S}_O, \oplus S_S) \wedge \lambda(B_0^s, B_S^s)$
Structural Requirements	$\lambda(B_0^a, B_S^a) \wedge \lambda(B_0^r, B_S^r)$

In other words, the design problem is a problem about how to change the existing state of universe to a desired state.

3.1.4 EBD process: environment analysis

The foundation of the design problem is the environment of the designed product. We can state the following theorem:

[Theorem 2] **Source of Product Requirements**. All product requirements in a design problem are imposed by the product environment in which the product is expected to work.

A design problem can be formulated based on the product environment $\bigcup_{i=0}^{n} E^i$. Obviously, different ways to organize the components in product environment will lead to different formulations of product requirements. To formulate the design problem clearly, it is important to analyze all the aspects of the environment.

The detailed derivation and discussion can be found in Zeng (2004).

The key objective of Environment Analysis is to find all the key environment components for a design problem and the relationship between the environment components. The result of this analysis constitutes an environment system. To facilitate the analysis, an environment decomposition method was developed (Zeng & Gu, 2001).

There are different ways to decompose the product environment to sub-environments. According to its properties, the environment can be viewed as composed of natural environment E^n, build environment E^b and human environment E^h. According to the importance for the product, environment can be classified as close and remote (Zeng, 2004).

It is impossible to list all the environments of a product before decomposition and analysis. Thus, the first step in decomposition should be the done according to the relative importance. It allows focusing on the close environments only and eliminating the relatively unimportant, remote environments. On the next step we should decompose the close environment to nature, build and human environments. Later on it is possible to perform further decomposition to technology, manufacture, assembly, market environments, etc.

The structure of the product environment is shown on the Figure 9.

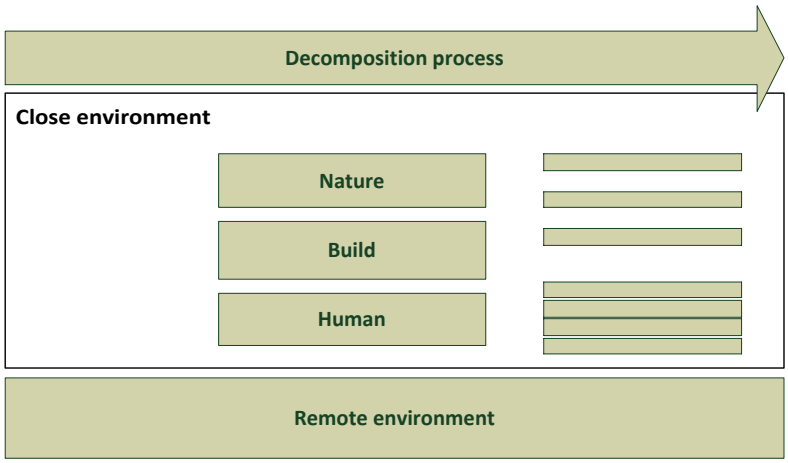

Figure 9 Structure of product environment

3.1.5 EBD process: conflict identification

One of the most important stages of the EBD process in the identification of key conflicts between environment components. We need to define a conflict. The Webster dictionary gives the following definition:

> *Conflict – competitive or opposing actions of incompatibles: antagonistic state or action (as of different ideas, interests, or persons).*

A conflict is composed of three basic elements: two competing objects and one resource object that the former two objects contend for (Yan & Zeng, 2009). By evaluating a conflict according to the five categories – relationship, data, interest, structural and value – one can begin to determine the causes of a conflict and design resolution strategies that will have a higher probability of success (Klein, 1991).

From a design point of view, a design concept is a composition of conflict resolutions. Environment decomposition allows finding out the conflicts between the product requirements and provides a basis for the conflict resolution (Yan & Zeng, 2009).

3.1.6 EBD process: concept generation

A concept is an approximate description of the technology, working principles and form of the product, which is sufficiently developed so that one can evaluate the principles that govern its behavior. The primary goal of the design concepts is to meet the requirements. Concepts must be iteratively refined in order to evaluate the technologies and implement them (Ulman, 1995).

The concept-generation process is the process of transformation from $\oplus\Omega_0$ to $\oplus\Omega_S$. It is demonstrated on the Figure 10.

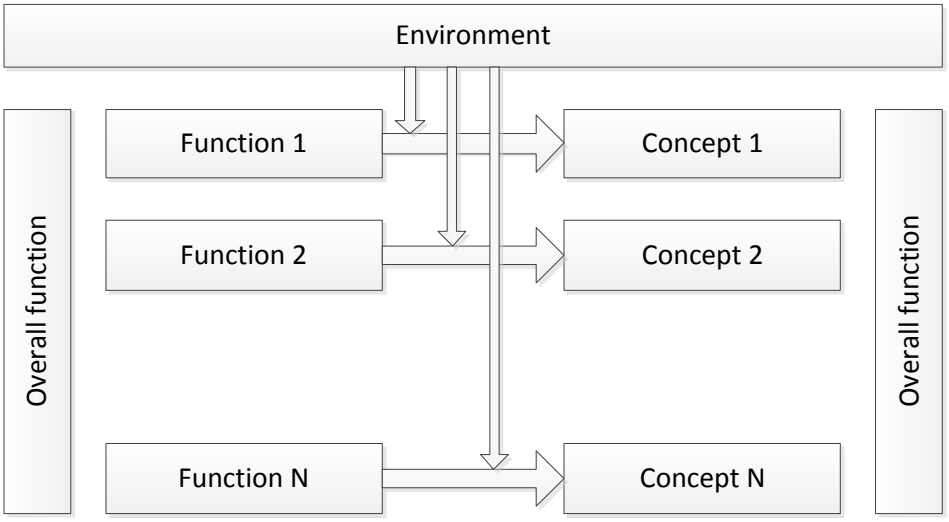

Figure 10 Concept generation process

3.1.7 EBD process: dynamics of the process

Design problem and product description evolve along the design process in EBD. Theorems 1 and 2 present a static structure of design problem. In this section we demonstrate the mechanism driving the evolution of the design. A generalized evolution process is shown on the Figure 11.

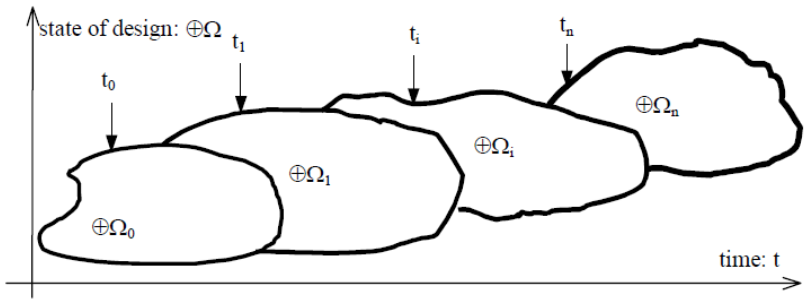

Figure 11 Evolution of product in the design process (Zeng 2004)

At each stage of the product evolution, B_i^j and B_{i+1}^j are defined as follows:

$$B_i^j = (E_j \otimes S_j) \cup (S_j \otimes E_j); \qquad (14)$$
$$B_{i+1}^j = (E_j \otimes S_{j+1}) \cup (S_{j+1} \otimes E_j).$$

At each stage of the evolution process, the design problem is defined by its current product system $\oplus\Omega_i$, which is called the state of the design. If P_i^d is the design problem at the i[th] stage of the design process, it can be represented as

$$P_i^d = K_i^e(\oplus\Omega_i), \qquad (15)$$

where K_i^e is evaluation operator responsible for identifying the conflicts between the current and desired states of design.

It can be seen from (14) that though the product environment does not change in most of the cases throughout the design process, the product-environment boundary B_i may be updated every time when the design solutions S_i are refined to S_{i+1}. As a result, the design

problem P_i^d will be updated as the design process progresses. This results in the zig-zag design process, as shown in Figure 12.

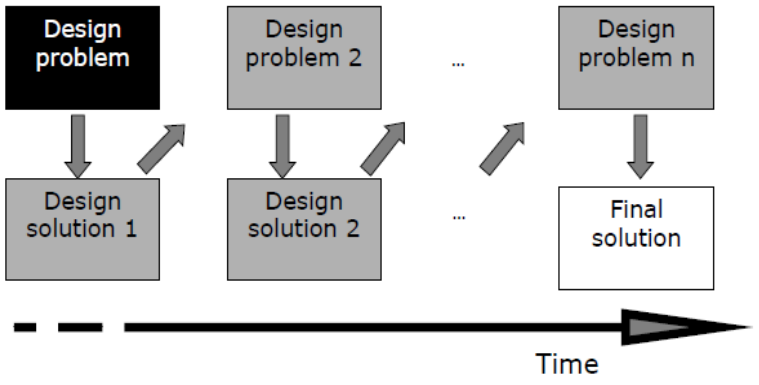

Figure 12 Zig-zag design process (Zeng 2004)

This can be stated as the following theorem:

[Theorem 3] Dynamic Structure of Design Problem. In the design process, design solutions to a design problem may change the original design problem, if the design solutions are different from their precedents, either by refinement or by alteration.

As can be seen in Figure 12, for each design problem P_i^d, there may exist design solutions S_i so that a new state of design $\oplus\Omega_{i+1}$ can be derived as follows:

$$\oplus\Omega_{i+1} = K_i^s(P_i^d), \tag{16}$$

where K_i^s is a synthesis operator responsible for generation of design concepts from a design problem. By substituting (15) into (16) we have:

$$\oplus\Omega_{i+1} = K_i^s K_i^e(\oplus\Omega_i),, \tag{17}$$

Equation (17) is called design governing equation. It underlines the design process and governs design activities. It defines dynamics of design. The basic concept behind this equation is the recursive logic of design (Zeng & Cheng, 1991) which states that design is

a recursive process in which the design solution and design problem interdependently evolve (Dorst & Cross, 2001), (Zeng & Cheng, 1991).

3.2 Design Matrix problem decomposition

3.2.1 Matrix-based decomposition of design problems

Matrix representation in product design and development can be classified into two formats: square matrix and rectangular matrix (RM). First one is often referred to as Design Structure Matrix (DSM), which rows and columns represent the same set of elements. Rectangular matrices capture the relations between different entities. In problem decomposition, the matrix's rows are labeled with design functions, and columns are labeled with design parameters. Thus the matrix entries show which parameters are required to achieve a specific function. This format is used in axiomatic design. In this context, problem decomposition is applied to divide the original complex problem represented in a matrix format into design sub-problems for a tractable design process.

Decomposition is a common and effective way to address the complexity of a design problem. In this context, matrix-based design decomposition is referred to the partitioning of a design problem that is represented in a matrix format. Particularly, the columns of this kind of matrix represent the parameters that describe the physical constituents and/or behavioral properties of a design, while the rows represent the constraint functions that define the correlations among these parameters. Then, each matrix entry indicates a dependency relationship of the corresponding row and column (Li, 2010).

One of the most widely used matrix decomposition methods is a two-phase method. The main feature of this method is the decoupling of the function of decomposition into two phases of analysis: dependency analysis and partitioning analysis. This methodical structure explicitly analyzes the coupling relationships between design elements to synthesize decomposition solutions. The original version of the two-phase method assumes the binary input matrix, which only captures the presence/absence type of dependency in a design problem. In design problem decomposition a non-binary matrix representation of relations is required.

3.2.2 Two-phase method overview

The two-phase decomposition method was proposed by Chen et al. (2004). This method is built upon the unique structure of a two-phase decomposition scheme that decouples the decomposition process into two functionally disjointed phases, each achieved by an autonomous algorithm. Figure 13 shows the high-level workflow of this method.

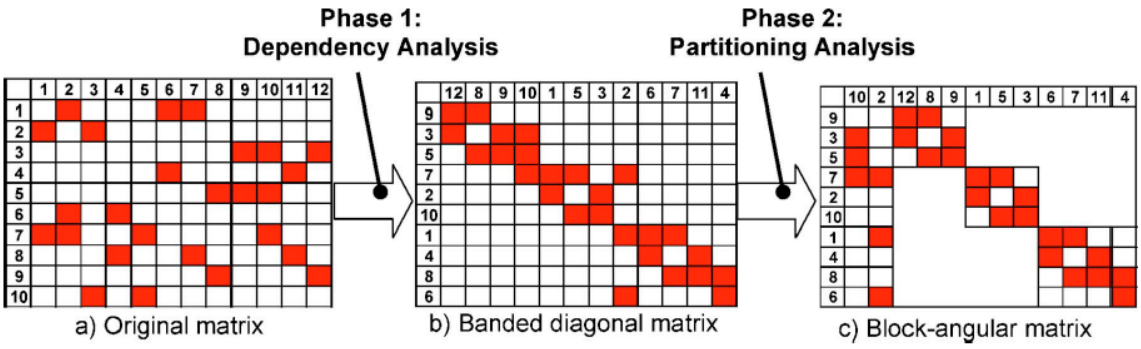

Figure 13 Two-phase method workflow (Li, 2010)

The input of the method is a rectangular matrix, which represents a system comprising two sets of elements, i.e., *n* column elements (design parameters) and *m* row elements (design functions). Then, each matrix entry exhibits a dependency relationship between

the corresponding row and column elements. The purpose of the two-phase method is to obtain a block-angular matrix, where the blocks represent subsystems and the interaction part represents the connection between subsystems, as shown in Figure 13 (c).

The two-phase method consists of two methodical components, which are labeled with Phase 1 and Phase 2. Phase 1 – dependency analysis – consists of two classes of algorithms:

1. *Cluster formation*. The coupling analysis is performed on rows and columns of the matrix. Obtained coupling information is used for hierarchical clustering analysis (HCA) to reallocate similar rows and columns close to each other and form clusters in a matrix. The formed clusters are often scattered, because the couplings between them are not explicitly considered.
2. *Cluster alignment*. Analysis of the couplings between clusters to bring similar clusters close to each other. The formed clusters will be aligned along the main diagonal direction, resulting in a banded diagonal matrix (Figure 13 (b)).

Phase 2 consists of the application of partitioning analysis to transform the banded diagonal matrix to a block-angular matrix (Figure 13 (c)). The following decomposition criteria are considered: number of blocks, size of blocks, and size of interactions. Partitioning analysis is designed to generate a set of decomposition solutions to satisfy the specified decomposition criteria (Li, 2010).

3.2.3 Non-binary dependency analysis overview

The two-phase method is limited in its usage due to the fact that its original version supports only binary input matrices, indicating presence/absence of the dependency. In many engineering problems the strength of dependency plays a very important role. Thus, a non-binary dependency analysis is a vital extension for the application scope of the two-phase method.

The dependency analysis of the two-phase method has its root with the hierarchical cluster analysis (HCA). The HCA researchers have developed numerous resemblance coefficients to address different types of classification, and the common coefficients are the distance coefficients, the association coefficients, and the correlation coefficients. In the context of matrix-based decomposition, it is assumed that the notion of coupling is relevant to similarity (resemblance coefficients).

Li in his work had selected the min/max formulation of Jaccard's resemblance coefficient to measure the coupling between rows and between columns. The formulation of couplings are given in equations (18) and (19).

$$r_{row\_ij} = \frac{\sum_{k=1}^{n} \min(m_{ik}, m_{jk})}{\sum_{k=1}^{n} \max(m_{ik}, m_{jk})}, i, j \in [1, m] \quad (18)$$

$$r_{col\_ij} = \frac{\sum_{k=1}^{m} \min(m_{ki}, m_{kj})}{\sum_{k=1}^{m} \max(m_{ki}, m_{kj})}, i, j \in [1, n] \quad (19)$$

where m_{ij} is a matrix entry of RM, and r_{row_ij} (r_{col_ij}) is the resulting coupling value between the i^{th} row (column) and the j^{th} column (row).

3.2.4 Overview of Phase 1

Similar to the binary dependency analysis, the non-binary one consists of the algorithms of cluster formation (CF) and cluster alignment (CA). To analyze non-binary information, only the cluster formation algorithm is modified from the binary version. The cluster alignment algorithm processes the same type of coupling information from the cluster formation algorithm and remains the same.

The following steps describe the non-binary cluster formation algorithm for columns. The same steps can be applied for the rows by transposing the matrix (Li, 2010).

1. Measure the coupling between columns using the min/max coefficient.
2. Construct the resemblance coefficient matrix (RCM) that indicated the coupling measure between every two columns.
3. Pick the column pairs that yield the highest coupling value to form a branch of the column tree. The column indices are shown as the leaves of the column tree.
4. Modify the resemblance coefficient matrix to represent the newly formed branch.
5. Repeat steps 3 and 4 until the resemblance coefficient matrix cannot be further reduced, and a complete column tree is formed. The index sequence of the formed column tree becomes the sequence to re-arrange the columns of the input matrix.

After applying the cluster formation algorithm for the input matrix, the cluster alignment (CA) algorithm is applied to align the formed clusters. For this step a Binary Tree Association (BTA) algorithm is used (Chen, Ding, & Li, 2005). This algorithm deals with the dependencies between the column tree and row tree.

The purpose of this algorithm is to arrange the branches of the row or column tree in an attempt to position the 1s elements along the main matrix diagonal. The step-by-step procedure of BTA is given below.

1. Divide the matrix into four parts based on the leaves of the branches B_{R1}, B_{R2}, B_{C1} and B_{C2}. Two lines, horizontal and vertical, are drawn to divide the matrix.
2. Calculate the number of 1s elements in each part using the formulation as

$$N_{kl} = \sum m_{ij} \ , i \in B_{Rk}, j \in B_{Cl} \tag{20}$$

   where N_{kl} is the number of 1s elements in Part *kl*.
3. Switch the branches B_{R1} and B_{R2} if $N_{12} + N_{21} > N_{11} + N_{22}$; otherwise, leave the tree intact.
4. Repeat steps 1-3 for the left and right branches of B_R until the tree leaf is reached. Figure 13 (b) illustrates the resulting matrix, obtained from applying BTA algorithm to the initial matrix (a).

3.2.5 Overview of Phase 2

The second phase of analysis implies the application of partitioning analysis to transform the banded diagonal matrix to a block-angular matrix as decomposition solutions of a matrix-based system. A concept of partition point is introduced to facilitate this function.

A partition point is an imaginary point that is placed on a banded diagonal matrix for two-block partitioning. The coordinates of partition point in the matrix are expressed as (row_i, col_i) and represent the position of horizontal and vertical partitioning lines. A partition point essentially divides a matrix into four parts. Based on the structure of the banded diagonal matrix, the diagonal parts will form the blocks (or subsystems), while

the nonzero elements in the off-diagonal parts will contribute to the interaction part. The placement of partition points becomes the essential step to determine the final decomposition solutions.

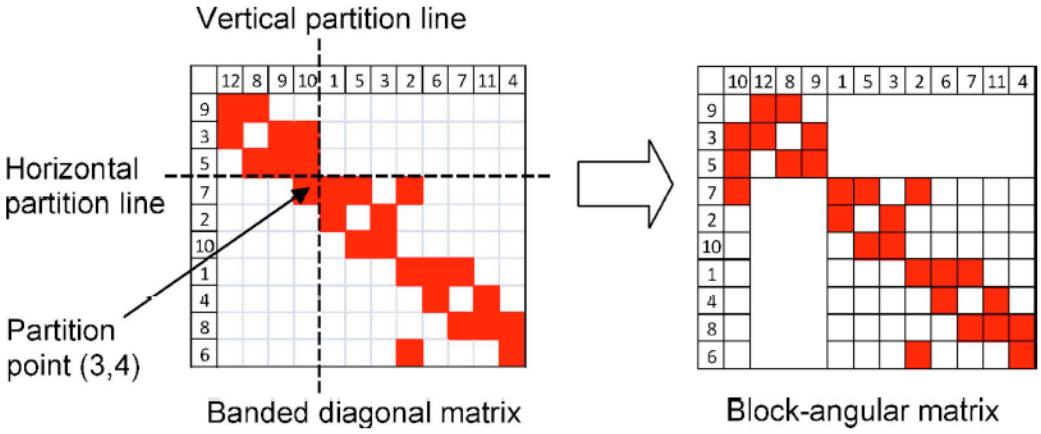

Figure 14 A sample diagonal matrix and partition lines (Li, 2010)

The number of possible decomposition combinations grows exponentially with the size of matrix. However, the engineers are looking usually for a single feasible solution which is reasonably good, instead of looking for all the feasible solutions. When decomposition criteria cannot be clearly specified, the engineers may want to identify several possible decomposition solutions for evaluation. The heuristic approach is developed for this case.

To estimate the quality of a decomposition solution, the matrix-based complexity metric is used. This metric approximates the complexity entailed in a block-angular matrix by inspecting the size of each block and the size of an interaction part.

The inputs of the heuristic partitioning analysis are the diagonal matrix and the resemblance coefficient matrices (RCM) from the dependency analysis. The step-by-step HPA algorithm is presented here:

1. *Step 1: Re-arrange the Rows and Columns of RCMs*. According to the row and column sequence of the diagonal matrix, the orders of the RCMs for rows and columns are re-arranged. Thus, the nonzero coupling values are clustered along the main diagonals, which indicates that the highly coupled rows and columns are placed close to each other.
2. *Step 2: Construct the Coupling-Partitioning Plots for Rows and Columns.* To construct a coupling-partitioning plot, we first place each partition line on the re-arranged RCM, which helps to identify the broken coupling values between two separate groups. These broken coupling values are added together and then divided by the total of the coupling values in the same RCM for normalization. The resulting normalized value is the broken coupling value that corresponds to the partition line, and it will be used for the plot.
3. *Step 3: Select the Partition Lines and Form Partition Points*. From the row and column coupling-partition plots, the partition lines that belong to the local minimum will be selected. If decomposition solutions with n_g blocks are desired, n_g-1 partition lines are required from the row and column coupling-partition plots, respectively, to form n_g-1 partition points. In addition, the selection of partition lines depends on some decomposition criteria. For instance, the size of blocks is measured via the number of rows and/or columns. Then, if the distance of the partition lines does not agree with the desirable block size, these partition lines will not be selected.

The heuristic partitioning analysis reveals how the coupling information can be utilized to expedite the process to obtain a decomposition solution. Through the coupling-partition

plots this coupling-driven approach provides a convenient way to explore different matrix-based structures (Li, 2010).

3.3 CBSP approach for requirements-architecture reconciliation

3.3.1 Introduction to CBSP approach

Understanding and supporting the interaction between software requirements and architectures remains one of the challenging problems in software engineering research (Nuseibeh, 2001). Evolving and elaborating system requirements into a viable software architecture satisfying those requirements is a difficult task, mainly based on intuition and experience. Similarly, little guidance is available for modeling and understanding the impact of architectural choices on the requirements (Egyed & Grunbacher, 2002). Software engineers face some critical challenges when trying to reconcile requirements and architectures:

- Requirements are usually captured informally in a natural language. On the other hand, entities in a software architecture specification are usually specified in a more formal manner causing a semantic gap (Medvidovic & Taylor, 2002).
- System properties described in non-functional requirements are commonly hard to specify in an architectural model (Egyed & Grunbacher, 2002)
- The iterative evolution of requirements and concurrent development of architectures demands that in the beginning architecture is based on incomplete requirements. More than that, certain requirements can only be understood after modeling or even partially implementing the system architecture (Nuseibeh, 2001).

- Mapping requirements and architecture, as well as maintaining the consistency and traceability between the two are complicated. A single requirement may address multiple architectural concerns and a single architectural element typically has numerous non-trivial relations to various requirements.
- Contemporary large-scale systems satisfy hundreds, even possibly thousands of requirements. It is difficult to identify and refine the architecturally relevant information contained in the requirements due to this scale.
- Requirements and the software architecture emerge in a process involving heterogeneous stakeholders with conflicting goals, expectations, and terminology. Supporting the different interests demands finding the right balance across these often divergent interests.

CBSP (Component-Bus-System-Property) approach provides an intermediate model between requirements and architecture that helps to evolve the two models iteratively (Nuseibeh, 2001). For example, a set of incomplete and quite general requirements captured as statements in a natural language might be available. The intermediate model then captures architectural decisions as an incomplete "proto-architecture" that prescribes further architectural development (Brandozzi & Perry, 2001). The CBSP approach also guides the selection of a suitable architectural style to be used as a basis for converting the proto-architectures into an actual implementation of software system architecture.

CBSP approach provides:

- a lightweight way of refining requirements using a small, extensible set of key architectural concepts

- mechanisms for “pruning” the number of relevant requirements, rendering the technique scalable by focusing on the architecturally most relevant set of artifacts
- involvement of key system stakeholders, allowing nontechnical personnel to see the impact of requirements on architectural decisions
- adjustable voting mechanisms to resolve conflicts and different perceptions among architects

Together, these benefits afford a high degree of control over refining large-scale system requirements into architectures.

3.3.2 CBSP taxonomy

The fundamental idea behind CBSP is that any software requirement may explicitly or implicitly contain information relevant to the software system architecture. It is frequently very hard to surface this information, as different stakeholders will perceive the same requirement in very different ways (Medvidovic et al., 2003). At the same time this architectural information is often essential in order to properly understand and satisfy requirements. CBSP supports the task of identifying architectural information contained in the requirements and explicating it in an intermediate model.

Each requirement is assessed for its relevance to the system architecture’s components (C), connections (buses), topology of the system or a particular subsystem, and their properties. Thus, each derived CBSP artifact explicates an architectural concern and represents an early architectural decision for the system.

There are six possible CBSP dimensions discussed below. They involve the basic architectural constructs (Medvidovic & Taylor, 2002) and, at the same time, reflect the simplicity of the CBSP approach.

1. *C* are model elements that describe or involve an individual Component in architecture. A requirement may be refined into CBSP model elements describing both processing components (C_p) and data components (C_d).
2. *B* are model elements that describe or imply a Bus (connector).
3. *S* are model elements that describe System-wide features or features pertinent to a large subset of the system's component and connections.
4. *CP* are model elements that describe or imply Component Properties.
5. *BP* are model elements that describe or imply Bus Properties.
6. *SP* are model elements that describe or imply System Properties.

A meta-model showing the different model elements relevant to CBSP is given in Figure 15. Requirements are related to architectural elements such as components or connectors via an intermediate CBSP model that acts as a bridge. Different subtypes of CBSP elements are used to represent different architectural dimensions listed in the CBSP taxonomy.

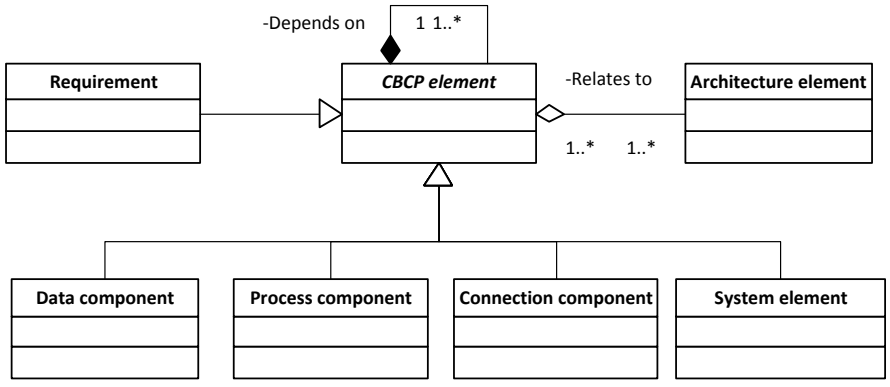

Figure 15 CBSP meta-model

3.3.3 CBSP process

This section discusses major steps of CBSP process, which can be generalized as shown in the Figure 16.

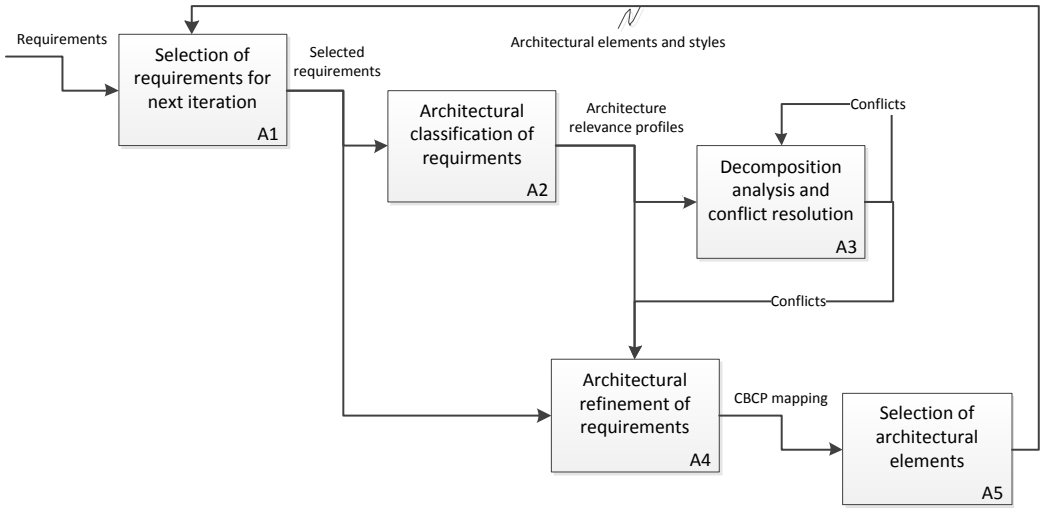

Figure 16 CBSP process

Step 1: Selection of requirements for next iteration.

To reduce the complexity of addressing large numbers of requirements, a team of architects applies the CBSP taxonomy to the most essential set of requirements in each iteration. The architects eliminate requirements considered unimportant or infeasible through stakeholder-based prioritization, thus arriving at a set of core requirements to be considered for the next level of refinement.

Step 2: Architectural classification of requirements.

Architect classifies the selected requirements using the CBSP taxonomy. Each requirements is assessed by the experts based on the requirement's relevance to the CBSP dimensions, using an ordinal scale (not=0; partially=1; largely=2; fully=3). For instance,

a requirement that is rated as partially relevant along the connector (B) dimension implies that it has some (partial) impact on one or more architectural connectors.

Step 3: Decomposition analysis and conflict resolution.

If multiple architects independently perform an architectural classification of requirements using CBSP, their findings may diverge since they may perceive the same statement differently. Revealing the reasons for diverging opinions is an important means of identifying misunderstandings, ambiguous requirements, tacit knowledge, and conflicting perceptions. The voting process is as a mechanism to reveal dissent among the architects and to reduce risks in requirements refinement.

Step 4: Architectural refinement of requirements.

In this activity the team of architects rephrases and splits requirements that exhibit overlapping CBSP properties and concerns. Each requirement passing the consensus threshold (concordance and at least largely relevant) may need to be refined or rephrased since it may be relevant to several architectural concerns. For instance, if a requirement is largely component relevant, fully bus relevant, and largely bus property relevant, then splitting it up into several architectural decisions using CBSP will increase clarity and precision.

Step 5: Selection of architectural elements.

At this point, requirements should have been refined and rephrased into CBSP model elements in such a manner that no stakeholder conflicts exist and all model elements are at least largely relevant to one of the six CBSP dimensions. Based on simple CBSP model elements, an architectural draft can be derived.

Architectural styles provide rules that exploit recurring structural and interaction patterns (referred to as "architectural patterns") across a class of applications and/or domains (Medvidovic, Rosenblum, & Taylor, 1999). A style guides the architectural design of a system, with the promise of desirable system qualities. At the same time, the rules guiding the selection and application of a style (or of specific architectural patterns suitable in that style) are typically semiformal at best, requiring significant human involvement.

The diagram on the Figure 17 shows the result of CBSP approach application for the software architecture problem formalization.

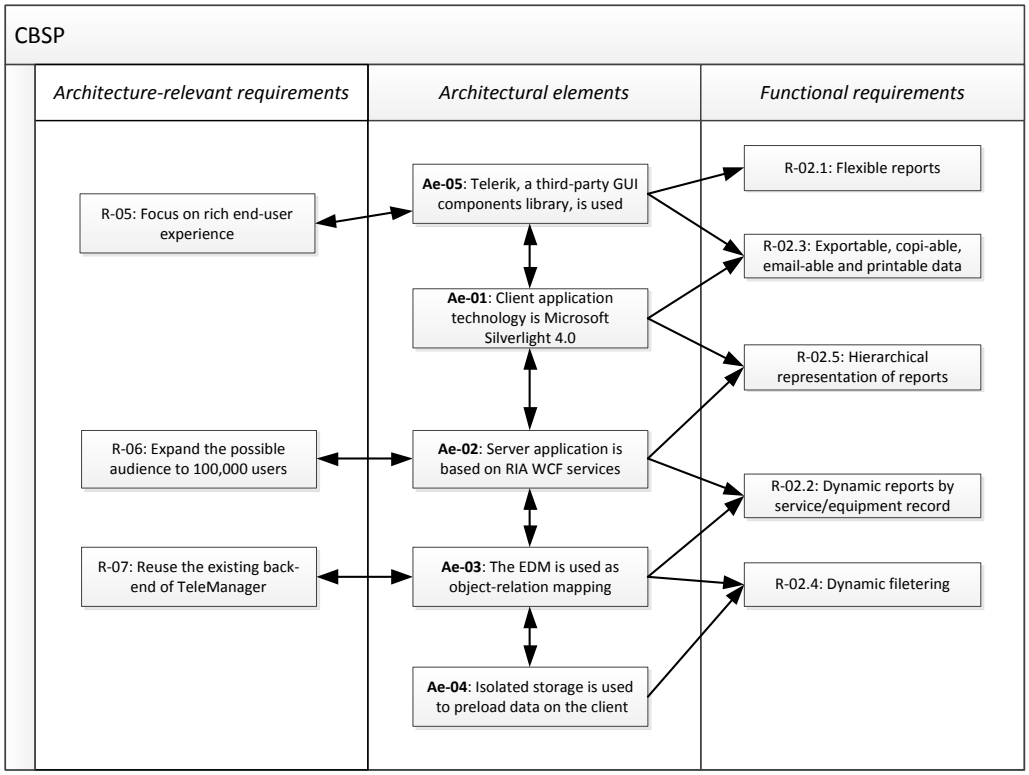

Figure 17 Example of relations between requirements and CBSP

4. Environment-Based Design of Software

4.1 Overview

This chapter describes the Environment-Based Design (EBD) approach for Agile Software Development. This approach represents the main contribution of this thesis.

4.2 EBD-S framework

The Environment-Based Design of Software (EBD-S) is an application of Environment-Based Design by Zeng (2004) to agile software architecture and design elaboration problem. It uses the generic process of EBD as a framework, and applies specific methods for conflict identification and concept generation.

The main goal of EBD-S is the application of formalized design approach to agile software development. In fact, Agile Manifesto states that working product is preferred to deeply developed design documentation (Beck, 2001). However, such an approach works better for small teams and projects (less than 1000 person/hours). When work synchronization between two or more agile teams is required, there is a need in well-elaborated design documentation (Paetsch et al., 2003).

Current approaches, addressing this problem – Feature-Driven Design (FDD) and Scrum, provide a generic recommendation to create a UML design documents in advance, and refine them iteratively. More than that, main idea of FDD is the development of the conceptual model before the code is written – and it requires an elaborated software design. These agile methodologies show the clear trend in software development – agile and traditional design methods merge together for better effectiveness. The EBD-S

approach is intended to work with two most recent and gaining popularity agile approaches – Scrum and Feature-Driver Design (FDD). Here we provide the theory of EBD-S application for Scrum. EBD-S-FDD approach differs in some aspects and is discussed on the basis of real examples in Chapter 5.

EBD for Agile Software Development compliments the Scrum process, and provides effective tools for requirements analysis, architecture creation and design concept generation. The EBD-S implementation does not require modification in Scrum process, it works with intermediate data only; it simplifies the adoption of the EBD-S methodology. This process is illustrated in Figure 18.

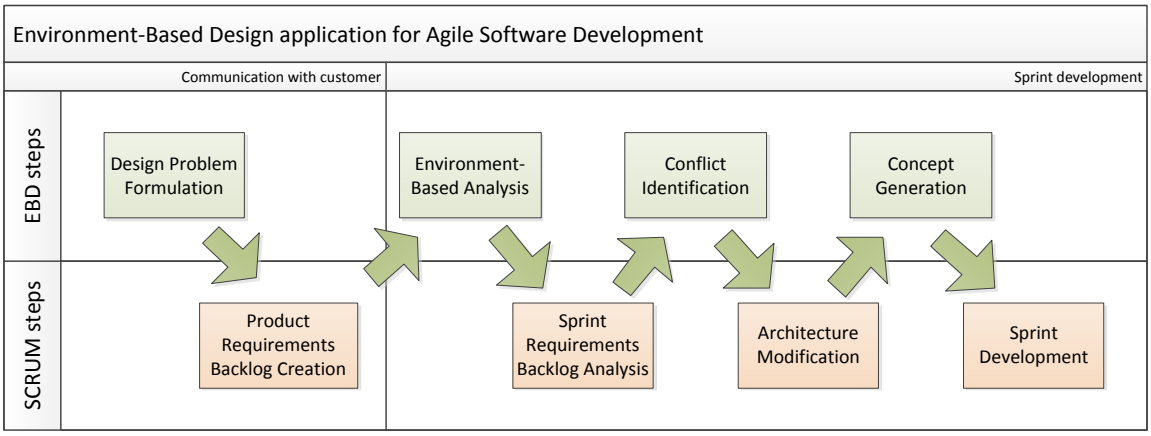

Figure 18 EBD - Scrum mapping

EBD-S implies use of specific analytical method on the each stage. In order to perform an effective conflict analysis, in our work we use CBSP methodology for requirements and architecture synchronization, adjusted for our needs.

To address the growing complexity of software systems, for concept generation we use a matrix-based problem decomposition approach, based on non-binary two-phase method, developed by Simon Li. The EBD-S framework is shown in Figure 19.

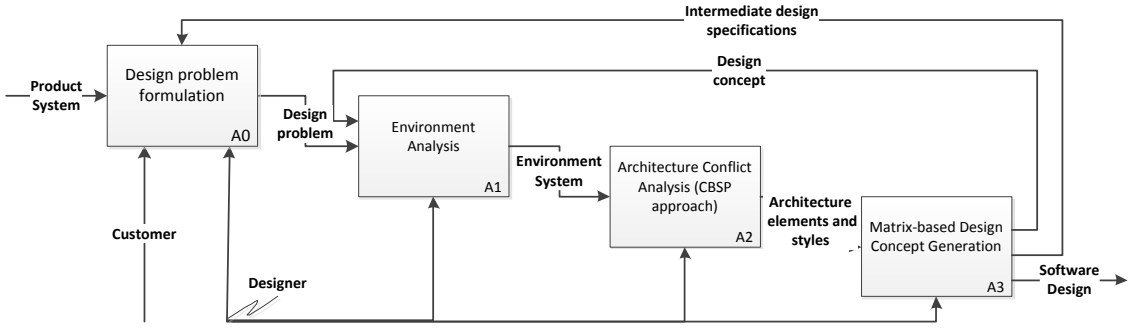

Figure 19 Environment-Based Design of Software framework

In the following sections we discuss the EBD-S stages in connection with agile software development methods.

4.3 EBD-S problem formulation

The problem formulation of EBD-S relies on the Theorem 1 of the Environment-Based Design – Structure of Design Problem:

> *A design problem is implied in a product system and composed of three parts: the environment in which the designed product is expected to work, the requirements on product structure, and the requirements on performance of the design product.*

Thus, the problem formulation stage of EBD-S addresses the understanding of the product software scope – $\lambda(\oplus \mathrm{E}_O, \oplus E_S)$, elaboration of quality software requirements – $\lambda(\oplus \mathrm{S}_O, \oplus S_S) \wedge \lambda(B_0^s, B_S^s)$, and functional software requirements – $\lambda(B_0^a, B_S^a) \wedge \lambda(B_0^r, B_S^r)$.

The software design problem is formulated in terms of the scope and performance (quality) / structural (functional) requirements (Zeng & Gu, 1999).

Functional requirements in system and requirements engineering define functions of software system or components, and describe them as sets of inputs, behaviours and outputs (Chen & Zeng, 2009). Functional requirements define *what* a system is supposed to accomplish. In agile software development process these requirements are usually captured in use cases. The implementation of functional requirements is described in *software system design*.

Quality (or non-functional) requirements specify the criteria that can be used to judge the operation of a system rather than specific behaviours. Quality requirements define *how* the software system is supposed to accomplish its mission. In agile software development process quality requirements are reflected in *software system architecture*.

As we can see, the EBD-S design problem can be formulated in terms of functional requirements, translated to system design, and quality requirements, reflected in system architecture. Both design and architecture shall be related to user requirements, expressed in natural language. To achieve that, EBD-S uses a graph-based model, based on CBSP approach.

4.4 EBD-S environment analysis

The environment analysis stage of EBD-S refines the software design problem by in-depth analysis of software product environment and identification of functional and quality requirements.

The software product environment can be represented as the union of the following four domains:

1. Software domain – technologies, development languages and platforms, existing software applications and communication protocols;
2. Hardware domain – the physical computers, networks and devices, which shall interact with the software;
3. Human interactions domain – people, directly or indirectly affected by the software product, and organizations.
4. Development domain – the people and organization, developing and maintaining the application, as well as the technologies used to facilitate the development.

Constraints, related to these domains, are marked as Cs, Ch, Ci, and Cd correspondingly. Requirements, communicated by customer, are marked as R.

Z. Y. Chen and Y. Zeng in 2006 classified product requirements based on product environment and identified 8 levels of requirements. Figure 20 illustrates this model.

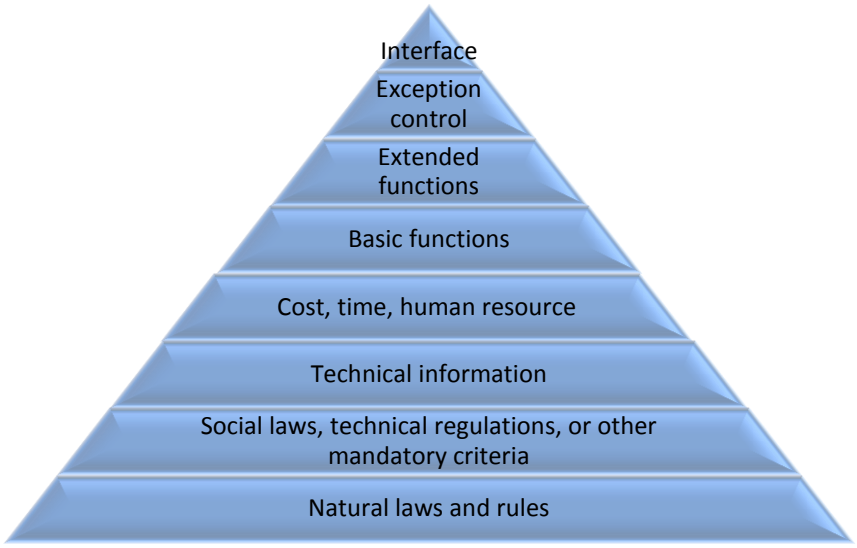

Figure 20 Levels of requirements, according to Chen and Zeng

Four requirements in the lower half of the pyramid represent the non-functional (quality) requirements; four upper requirements are functional (Chen & Zeng, 2006).

The environment analysis process model (Zeng & Gu, 2001) can be described as follows:

1) Extract one environment element from the environment set
2) Determine whether there is a piece of design knowledge mapping the extracted element to another action or response. If so, the product structure *s* attached to this knowledge, will be a component of architecture / design concepts
3) Add component *s* to the product structure S, and perform conflict analysis.
4) Form a new environment set and repeat the analysis, if necessary.

The environment analysis process allows determining the full body of product system, and to gather and classify the product requirements. It can be directly applied to the software design problem.

All elements from the close environment of the product system are analyzed with this algorithm, and the requirements are derived and classified. If the requirements expressed by a customer are insufficient, this analysis allows to identify the gap and to communicate it back to the client. Scrum development model approves the development process in the conditions of insufficient requirements. Scrum developers hope that a working prototype of the software will help to a much better feedback from the customer.

At this stage EBD-S compliments the Scrum model with the analysis tool, allowing to capture the missing requirements and (possibly) to re-focus the development process.

4.5 EBD-S architecture conflict analysis

4.5.1 Environment and conflict analysis process

The next stage of EBD-S – architecture conflict analysis – finds and deals with the contradictions in functional and quality requirements that are selected for current Scrum sprint.

As we determined in Section 4.3, quality requirements of the software system are reflected in the software architecture, which governs the design. This stage relies on the architecture analysis methods. We adapted the CBSP model for requirements and architecture reconciliation of Environment-Based Design of Software. The generic process of environment and analysis for software is shown in Figure 21.

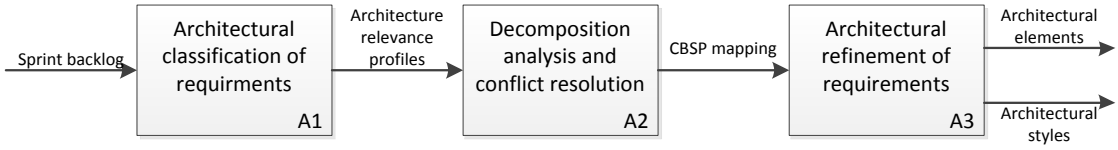

Figure 21 Architecture conflict analysis process

Each architecture conflict analysis step is discussed in more detail below. We use ETVX (Entry, Task, Verification, and eXit) (Radice, Roth, O'Hara, & Ciarfella, 1985) to document the steps. ETVX cells consist of four components:

1. Entry lists all items required for the execution of the task
2. Task describes what should be done, by whom, how, and when (this includes appropriate standards, procedures and responsibilities)
3. Verification/Validation describes all checks and controls that help to indicate if the task is being executed properly
4. eXit lists criteria which need to be satisfied before the task can be considered complete and the output(s) of the task itself

4.5.2 Architectural classification of requirements

The requirements, elicited at environment analysis stage, are classified using the CBSP taxonomy. Each requirements is assessed by the experts based on the requirement's relevance to the CBSP dimensions, using an ordinal scale (not=0; partially=1; largely=2; fully=3).

Table 4 EVTX for architectural classification of requirements

Architectural classification of requirements	
E	Set of requirements for next-level RDCP refinement RDCP taxonomy Voting tool
T	Architect classifies selected requirements using the CBSP taxonomy
V	Check selection of architect Check completeness of classification
X	Voting ballots Architectural relevance profiles for all requirements

A profile showing the aggregated architectural relevance is created for each requirement.

4.5.3 Decomposition analysis and conflict resolution

If multiple architects independently perform an architectural classification of requirements using CBSP, their findings may diverge since they may perceive the same statement differently. Revealing the reasons for diverging opinions is an important means of identifying misunderstandings, ambiguous requirements, tacit knowledge, and conflicting perceptions (Wang & Zeng, 2008). The voting process is as a mechanism to reveal dissent among the designers and to reduce risks in requirements refinement.

The measured consensus among the designers serves as a proxy for their mutual understanding of a requirement's meaning and their agreement on the architectural relevance of a requirement.

Table 5 ETVX for decomposition analysis and conflict resolution

Decomposition analysis and conflict resolution	
E	Voting ballots Architectural relevance profiles for all requirements
T	Designers discuss reasons for diverging opinions for low-consensus items Designers update requirements to address issues and ambiguities Designers exclude architecturally irrelevant requirements
V	Check dependencies among requirements to make sure critical requirements are not dropped
X	Issues and ambiguities Architecturally relevant requirements

The rules in Table 6 indicate how to proceed in different situations: in case of consensus among architects, the requirements are either accepted or rejected based on the voted degree of architectural relevance.

Table 6 Concordance / relevance matrix

	Relevance	
Concordance	>= Largely	< Largely
Agreement	Accept	Reject
Disagreement	Discuss and redefine	

We accept requirement as architecturally relevant if the mean of all stakeholders is at least "largely", otherwise the requirement is rejected. If the stakeholders cannot agree on the relevance of a requirement to the architecture, they further discuss it to reveal the reasons for the different opinions. This discussion process may also involve customers and other stakeholders to clarify a requirement and eases the subsequent step of refining it into one or more architectural dimensions.

4.5.4 Architectural refinement of requirements

In this activity the team of architects rephrases and splits requirements that exhibit overlapping CBSP properties and concerns (see Table 7). Each requirement passing the consensus threshold (concordance and at least largely relevant) may need to be refined or rephrased since it may be relevant to several architectural concerns. For instance, if a

requirement is largely component relevant, fully bus relevant, and largely bus property relevant, then splitting it up into several architectural decisions using CBSP will increase clarity and precision. During this process, a given CBSP artifact may appear multiple times as a by-product of different requirements.

Table 7 ETVX for architectural refinement of requirements

Architectural refinement of requirements	
E	Issues and ambiguities Architecturally relevant requirements
T	Architects rephrases and splits requirements that exhibit overlapping CBSP properties Architects eliminate redundancies
V	Check to make sure that redundancies are minimized
X	CBSP elements with dependencies Architectural styles

Along with CBSP elements and their interdependencies, the output of this step is a set of architectural styles. Architectural styles provide rules that exploit recurring structural and interaction patterns (referred to as "architectural patterns") across a class of applications and/or domains (Medvidovic et al., 1999). A style guides the architectural design of a system, with the promise of desirable system qualities.

4.5.5 Software architectural styles and proto-architecture

In EBD-S software architecture can be viewed as a set of limitations for the design, as "rules to follow" or "legacy code to use". Architecture implements the class of requirements. It is a "strategic design", which has a goal to fulfill quality requirements and gain advantage in a long-term prospect, but not in the context of the current project with the existing functional requirements (Perry & Wolf, 1992).

Software architecture in EBD-S is the result and the main component of the strategic planning process. This process is always continuous: the architecture must be corrected with the course of time, reflecting new concepts, risks, threats and possibilities. The iterative nature of EBD-S allows refining the architecture of the software system continuously.

Based on the dependencies among the elements in CBSP, the rules of the architectural style allow us to compose them into architecture. In other words, we select the style based on (1) the characteristics of the application domain and (2) the desired properties of the system, identified in the requirements negotiation and elaborated in the CBSP model. By considering the rules and heuristics of the selected style(s) the designers start converting the CBSP model elements into components, connectors, configurations, and data, with the desired properties. In other words, architectural style determines the set of possible software design solutions. To perform this job, a proto-architecture structure is used. Figure 22 illustrates this structure.

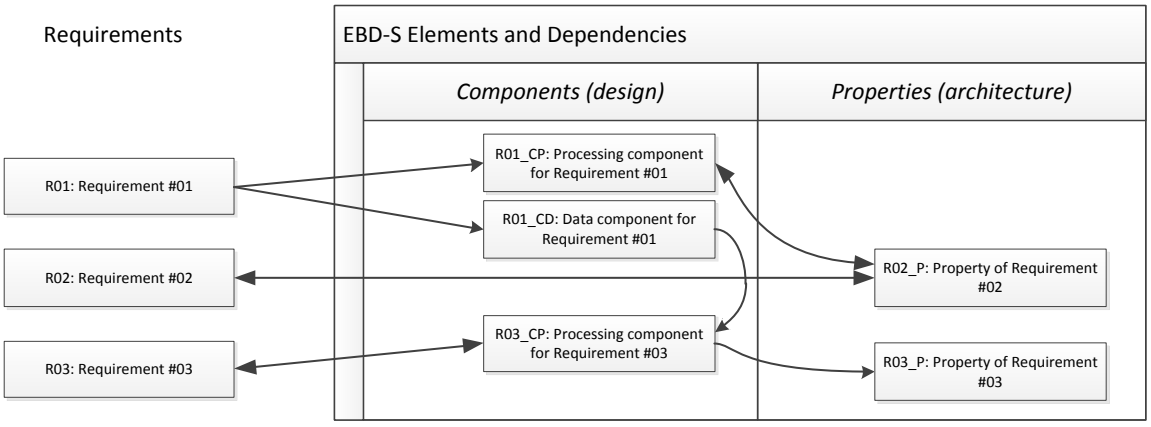

Figure 22 EBD-S requirements-architecture-design reconciliation

Design is derived from the existing functional requirements with the architectural limitations in mind. In some cases design elaboration can cause the architectural changes. The process of software design concept generation is discussed in the Section 4.6.

4.6 EBD-S design concept generation

The inputs of the design concept generation phase of EBD-S are architectural styles and elements, as well as previously developed design concepts (if any). The conflict identification step of the EBD-S prepares an architectural model of the system, which addresses the quality requirements.

To encompass the functional requirements in the same structure, we need to analyze them and find the candidate solutions – software design elements that address the requirements in question. Next we need to estimate the feasibility of our candidate solutions – it can be done by decomposition of the software design problem to sub-domains.

On this step we create a rectangular matrix, which rows represent the functional requirements and architectural elements, and columns represent design elements that address these requirements. The relation between requirements and architectural/design elements is given in non-binary format, on the scale of 0-3: 0 – no relation, 1 – weak relation, 2 – strong relation, 3 – fully coupled elements.

To generate design concepts, we apply a problem decomposition method, based on extended two-phase method (Li, 2010).

On the first phase we perform cluster formation and cluster alignment algorithms to transform the initial matrix to a banded diagonal matrix, representing the sub-systems of the software system.

On the second phase we apply heuristic partitioning analysis to convert the banded diagonal matrix, obtained on the first phase, into possible block-angular matrixes, which would represent decomposition solutions of the design problem. An example of such decomposition solutions is shown in Figure 23.

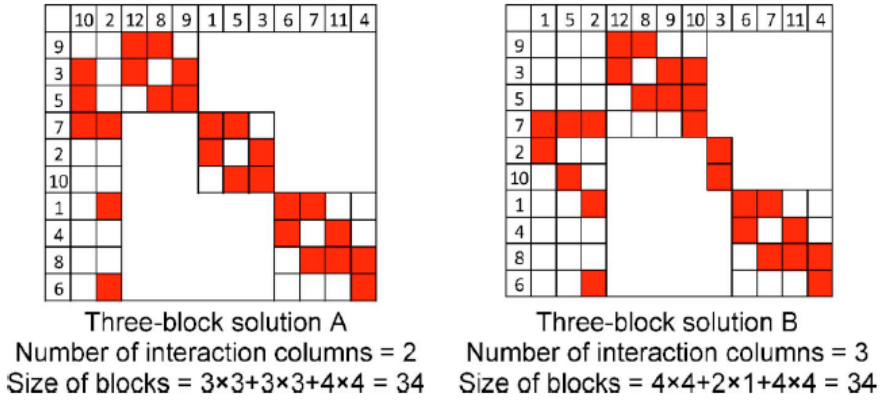

Figure 23 Comparison of two decomposition solutions (Li 2010)

The resulting matrix decomposition solutions will represent the alternatives of the system design. To estimate the quality of a decomposition solution, the matrix-based complexity metric is used. This metric approximates the complexity entailed in a block-angular matrix by inspecting the size of each block and the size of an interaction part (Li, 2010).

The smaller blocks with fewer interactions among them will have smaller complexity value, and are more feasible and easy-to-maintain after implementation. As per example in Figure 23, both of the solutions have the same number of blocks, but Solution A has 2 interaction columns against 3 in Solution B. Thus, Solution A is considered to be a better solution.

To finalize concept generation step of EBD-S, the selected solution is expressed in terms of architectural and design element, captured in CBSD architecture – it can be done automatically by translation of the resulting design matrix to the CBSP model graph. The results of the design concept generation step are:

- New or updated software design concept
- CPSB model, capturing the interaction between the elements
- New or updated design specifications, based on the design concept

In terms of EBD-S-Scrum application, these results are the basis of the implementation of the next sprint. After the sprint implementation, the resulting software along with the design can be:

- Transferred to the Environment-Based analysis for further refinement, which marks the inception of the new sprint in Scrum development; or
- Communicated back to the client to retrieve feedback and/or approval on the software development progress.

Thus, in this section we demonstrated how the EBD-S approach compliments the agile software development by providing a flexible framework for requirements analysis, architecture elaboration and design concept generation. The next section will cover the aspects of change control in EBD-S.

4.7 EBD-S change control mechanism

The nature of agile software development implies multiple changes in requirements and product, introduced on the iterative basis (Peters & Ramanna, 2003). Product backlog requirements are used to define “should-be” vision in the first iteration. After several iterations, the “to-be” design concept arrives, which take into consideration all the constraints and conditions of the software architecture.

According to James F. Peters, four main problems, associated with the unplanned changes in software development, are:

- Requirements non-conformance (requirements erosion)
- Architecture erosion
- Design erosion
- Code erosion

EBD-S, being a design method, addresses first three problems with the coupled requirements-architecture-design structure. All modifications, introduced to the system, are reflected in the CBSP proto-architecture and are transferred to the design concept generation stage. That allows to monitor and to control effectively the unplanned changes in the system. An example of this is shown in Figure 24.

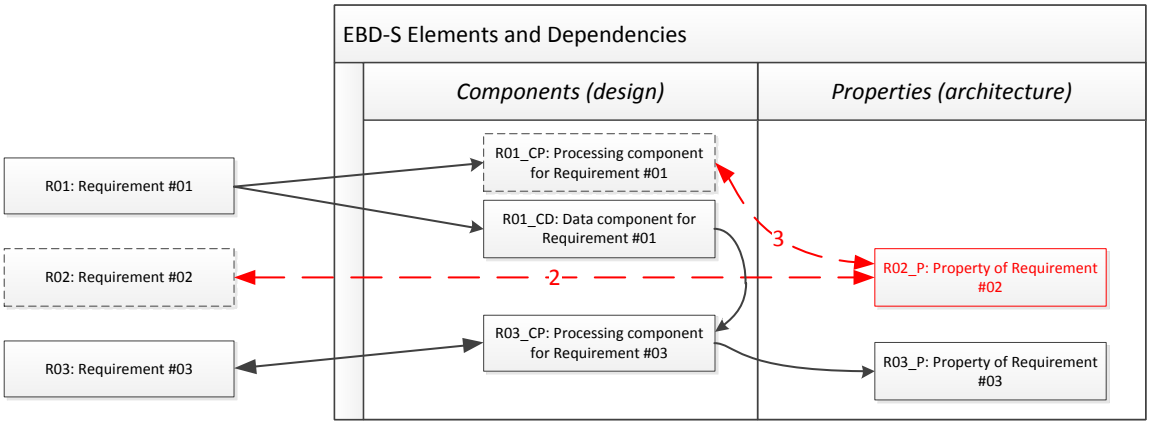

Figure 24 EBD-S impact analysis

Here a change in system property R02_P can cause potential changes in component R01_CP and requirement R02. The rest of elements can be impacted indirectly as well.

The strength of the relation (label of the arrow) demonstrates the intensity of the impact of the change.

5. EBD-S application for telecom expense management software development: Case Study

5.1 Introduction

The case study for the application of EBD-S method to Scrum software development process is based on the real-world example from telecom expense management (TEM) domain.

The TEM application used in the case study is developed by a Canadian company and is called TeleManager. This is an enterprise-level application, aimed to collect and analyze data about telecom expenses, maintain and track the inventory of telecom services and assets, and support telecom ordering processes within an organization. It manages the entire lifecycle of network services. The main functional areas of the TeleManager are shown in Figure 25.

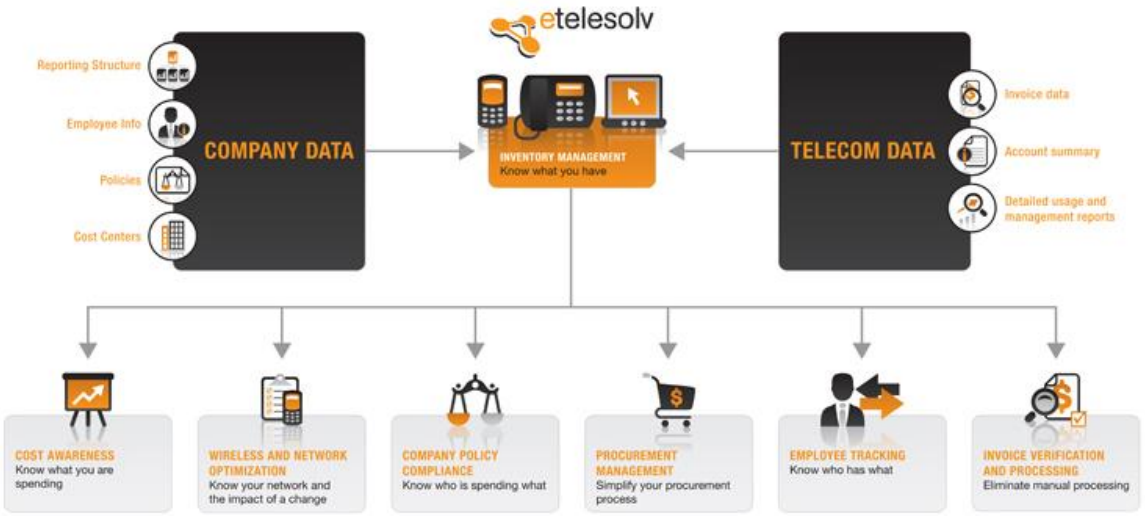

Figure 25 TeleManager functional domains

TeleManager is developed with help of agile methodology – Scrum, with elements of Feature-Driven Design. The methodology implies decomposition of the initial design scope to small sub-domains, which are developed in short iterations – sprints (around one month each). The result of each iteration is a set of new or updated features, which are added to the application framework.

Current customer base of TeleManager exceeds 25 clients. Company delivered customized versions of TeleManager to many of them, focusing on the specific customer requirements for each implementation.

The Scrum development approach clearly shows its strength in this situation – the product is customizable and projects are delivered in time. However, there are some drawbacks of the existing approach:

- While customer requirements are consistent within one client implementation, there is significant difference between the requirements of different clients. This difference is not documented, as Scrum approach focuses on the delivery of the working code, and the requirements analysis is done during the coding.
- Customer requirements are frequently modified during the iterations, and sometimes they are communicated to the developers after the iteration is over – the lack of pre-defined requirements specifications hinders the development.
- Architecture of the software application lacks unification; different parts were developed with no correspondence, which results in the difficulties in the evolution process: it is getting hard to encompass new technologies.

- Software design relies on the existing codebase and knowledge of the developers, with no proper documentation. It leads to the problems with analysis of the new functionality, which is performed empirically, and with the knowledge transfer.

The EBD-S methodology is developed as a complimentary design process to the Scrum/FDD development. It is aimed to address the stated issues of the Scrum/FDD approaches by introducing a formalized framework for requirements collection and analysis, architecture elaboration and design concept generation and selection.

The flexibility of EBD-S allows implementing it within a working process without interruption and step-by-step. EBD-S creates a certain overhead in the agile development process, but it is easy to calculate the time, dedicated to the EBD-S process, and estimate the effectiveness of the method by looking at overall development performance change. For this estimation several development iterations are required.

5.2 Structure of the case study

The presented case study is organized as following:

- The design task is formulated in terms of Product Requirements Backlog in Section 5.3
- The product environment description and decomposition extend the requirements analysis and provide the architectural classification of requirements in Section 5.4
- Architectural conflict analysis is demonstrated in Section 5.5 – it shows the process of software architecture establishment with help of CBSP model
- Section 5.6 illustrates the matrix-based non-binary analysis of the design problem within the framework of architectural constraints.

- Iterative application of the EBD-S is illustrated in Section 5.7
- Effectiveness of the EBD-S is calculated and discussed in Section 5.8

The structure of the case study represents the EBD-S process flow as it was implemented in the real-world software development process.

5.3 TeleManager Executive: design task formulation

TeleManager is built on the highly-customizable software platform, which allows reshaping the application for a specific client needs. However, the following factors started to play significant role with the growth of the software complexity:

1. As TeleManager represents the Software-as-a-Service (SaaS) model, the addition of new features imposes higher workload on the company servers;
2. End-user interactions within the existing model had very high latency, resulting in a poor user experience;
3. Clients dedicate significant human resources to work with TeleManager, since training is required for professional use of the application; a simplification of user interface will bring value to the clients;
4. Graphical user interface, available to the TeleManager technologies, is significantly behind the interface of desktop applications; and
5. Some client request in-house installations instead of Software-as-a-Service model, which is not supported by current technology.

The decision was taken to build a new application, which will use the TeleManager back-end and will provide access to the analytical, reporting and personal information. The

following design task was specified: build a customizable Rich Internet Application, named TeleManager Executive (TME), which will:

R-01. Provide interactive graphical representation of telecom costs;

R-02. Build reporting engine for telecom costs and services;

R-03. Provide real-time access to the personal and departmental telecom invoices;

R-04. Provide access to personal information and telecom service control center;

R-05. Focus on rich end-user experience, with interactive graphical part;

R-06. Expand the possible audience of the single instance to 100,000 users;

R-07. Reuse the existing back-end of the TeleManager;

According to the requirements, four main functional areas (FAs) of the TME are identified:

FA-1. Dashboard, for providing interactive summarized information about telecom costs and important system messages

FA-2. Reports, providing access to configurable financial and telecom service reporting engine

FA-3. Invoice, displaying personal or departmental telecom invoices

FA-4. Self-service, providing end-user access to telecom service information and configuration.

Each functional area is analyzed and decomposed further for identification of the design-relevant requirements. The following sections describe how the set of high-level requirements (R01-R07) was analyzed with help of Environment-Based Design of Software methodology.

5.4 TeleManager Executive: environment Analysis

5.4.1 Software environment

The evolutionary development of TeleManager software was relying on the stack of Microsoft web-technologies, focused around ASP.NET and Microsoft SQL Server. The legacy code is the key component of the software environment of the TeleManager.

The software part of the TME product environment was identified as shown in Figure 26.

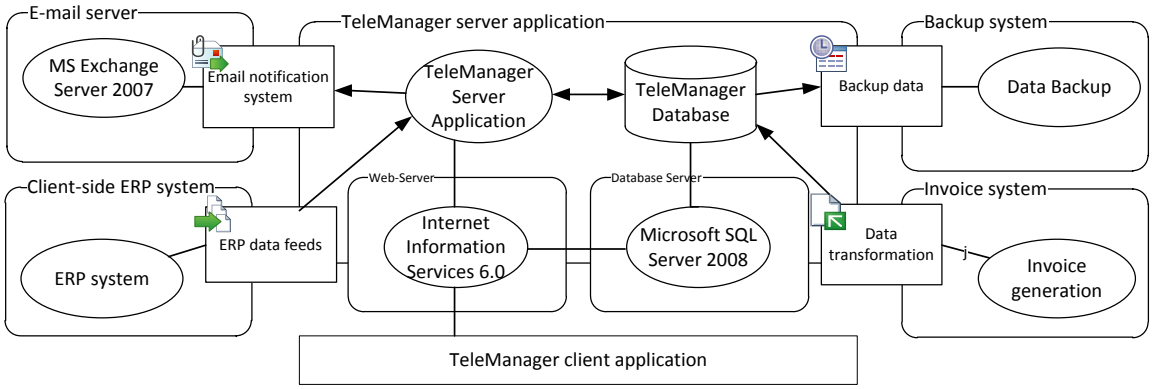

Figure 26 TME software environment

The software infrastructure shall be reused in the new TME system, thus the following elements of the software environment are identified:

1. Application server: Internet Information Services (IIS) 6.0 or later;
2. Database server: Microsoft SQL Server 2008 Enterprise Edition;
3. Email notification system: Microsoft Exchange Server 2007;
4. Back-up data server: Microsoft Enterprise Backup 2005;
5. Client-side enterprise systems:
    a. Oracle PeopleSoft HumanResource;
    b. SAP HR;

c. J.D. Edwards EnterpriseOne HR;

d. SecondNature HelpDesk;

6. Provider invoice generation systems:

    a. Bell billing portal;

    b. Telus invoicing;

    c. Rogers invoicing;

    d. Verizon billing;

7. Client application operating environment – Windows-based in-browser

The environment analysis allowed to identify the following software environment items to be in close product environment:

1. Application server: Internet Information Services (IIS) 6.0 or later;
2. Database server: Microsoft SQL Server 2008 Enterprise Edition;
3. Email notification system: Microsoft Exchange Server 2007;

The rest of environment items are not directly related to the TME application, and are considered to be in the remote environment. The following software constraints (Cs) were drafted after the analysis of the close software environment:

Table 8 TME software constraints

Code	Description
Cs-01	Application shall run under Microsoft IIS 6.0 or later
Cs-02	Application shall use the MS SQL 2008 database
Cs-03	Email notifications shall be sent through MS Exchange 2007
Cs-04	Client application shall run in web-browser under MS Windows 2000 or later

5.4.2 Hardware environment

The hardware environment of TeleManager is shown on the Figure 27.

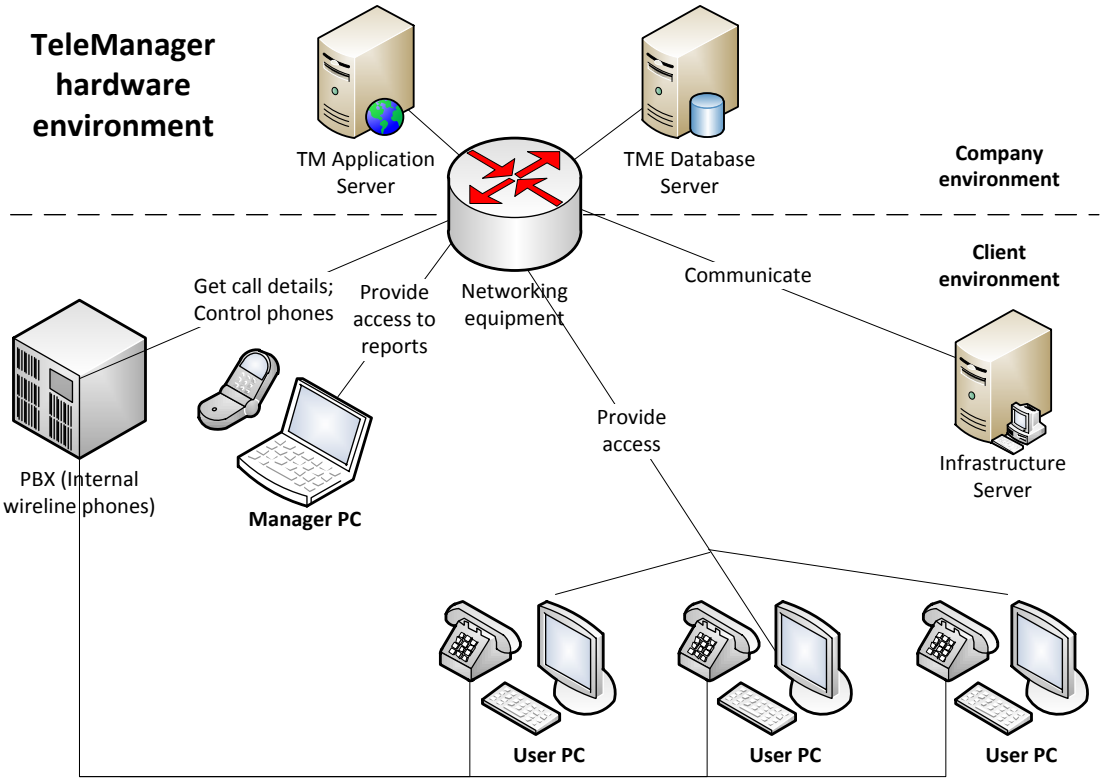

Figure 27 TME hardware environment

The hardware environment of the TeleManager application, formulated during the preliminary analysis, is the following:

1. Internal Application and Database servers – physical computers running the software and storing the database;
2. Customer PBX (one or many) – a telephone exchange, that controls the telephone system in the client's office (offices). TeleManager downloads call details from the PBX;
3. Telephones, connected to the PBX. TeleManager controls phone configurations;

4. Customer Infrastructure Server(s), communicating with the TeleManager;
5. User's and manager's computers;
6. Networks, relating the internal company servers with client environment.

The analysis of hardware environment shows that majority of the hardware items are located in the Remote Environment. In fact, TeleManager communicates with PBXs, telephones and Infrastructure Servers via software protocols; thus, there is no direct impact on the TeleManager from these elements. The only items rest in hardware environment of TeleManager application are:

1. Internal Application and Database servers – they affect the processing speed of server application and data volume available to be stored;
2. Networking equipment, defining the speed of communication between server and client applications, as well as between server and client's infrastructure;
3. Client computers, defining the processing speed and interface of client application.

The analysis of these environmental items, directly related to the product, resulted in the following list of hardware constraints (Ch), shown in Table 9.

Table 9 TME hardware constraints

Code	Description
Ch-01	Server application shall run on predefined hardware configuration*
Ch-02	Database storage is limited to 120GB of raw data
Ch-03	Network latency between server and client is 500-750 ms
Ch-04	Client computers have given minimal hardware configuration**

* In the project – Dell PowerEdge R410
** In the project – Dell Vostro 1015

5.4.3 Human interaction environment

The human domain of the TeleManager environment is represented by the following user roles and is shown in Figure 28:

1. Internal administrator – uploads the telecom invoices to the TeleManager database;
2. Customer support – resolves the customer issues by providing advice / configuration suggestions;
3. Financial department – receives the telecom invoices, uploaded to the system;
4. Manager – monitors the activity and the expenses of the employees, approves their requests;
5. Employee – receives personal telecom invoices, generated by TeleManager, and modifies personal telecom service through self-service portal;
6. TeleManager client administrator – controls the activity of employees.

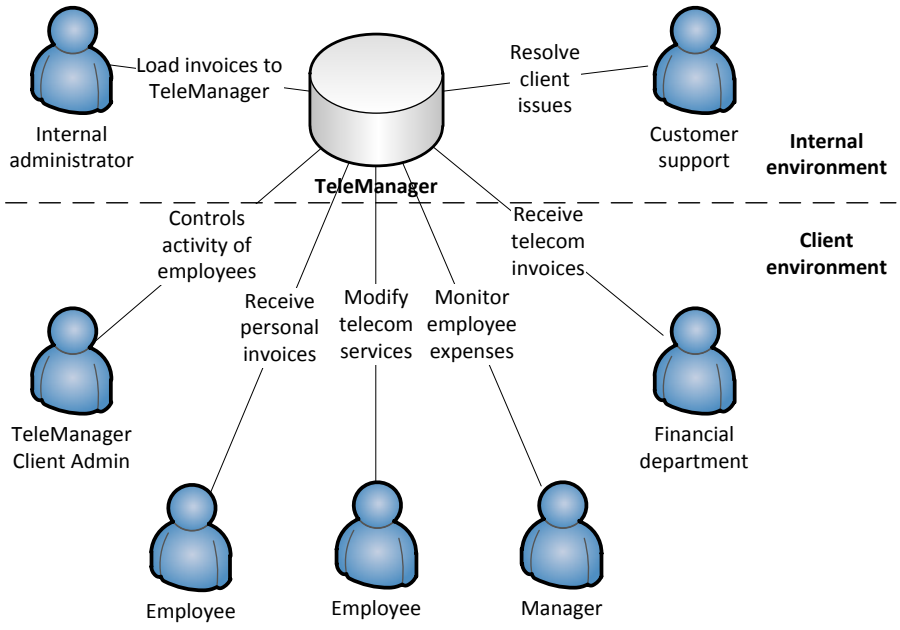

Figure 28 TME human interaction environment

Environmental analysis allows discovering that the following user roles are directly interacting with TME application:

1. Customer support – to review user activity and understand their requests;
2. Manager – to get access to the reporting information and review employee requests;
3. Employee – to view personal invoices and request modification of personal telecom services.

The rest of user roles in the given project belong to the remote product environment. Analysis of user roles allowed to define the list of human interaction constraints (Ci), presented in Table 10.

Table 10 TME human interaction constraints

Code	Description
Ci-01	Support up to 100,000 concurrent users
Ci-02	Display the data based on the user role
Ci-03	Follow the user actions in the system in real-time
Ci-04	Keep page update latency under 1 second
Ci-05	Support long transactions on the client side

5.4.4 Development environment

The development domain of the TME environment is represented by the following components:

- Development framework – Microsoft Visual Studio 2010 Premium
- Database – Microsoft SQL Server 2008
- Code sharing tool – Microsoft TeamFoundation Server 2010
- Collaboration tool – Microsoft Sharepoint 2010

The roles in Scrum process, used in TME development, are:

- Product Owner – decides what will be built and in which order
- Scrum Master – a facilitative team leader who ensures that the team adheres to its chosen process and removes blocking issues
- The Team – cross-functional team of 5 developers who perform the coding

Analysis of development environment technologies and process roles resulted in the list of environmental constraints, presented in Table 11.

Table 11 TME development constraints

Code	Description
Cd-01	System shall be highly maintainable (updates without service interruption)
Cd-02	System model shall be customizable (loose coupling of modules)
Cd-03	Unit testing shall be applied to minimum of 75% of code logic

5.5 TeleManager Executive: architectural conflict analysis

5.5.1 Requirements classification and architecture synthesis

At the next step the analysis of requirements and environmental constraint is performed. First, the architecturally relevant requirements are selected; then, the environmental constraints are added to the pool. At the next step the requirements and constraints are analyzed for the correspondence to the software architectural elements. The process of the requirements classification and architectural synthesis is shown schematically in Figure 29.

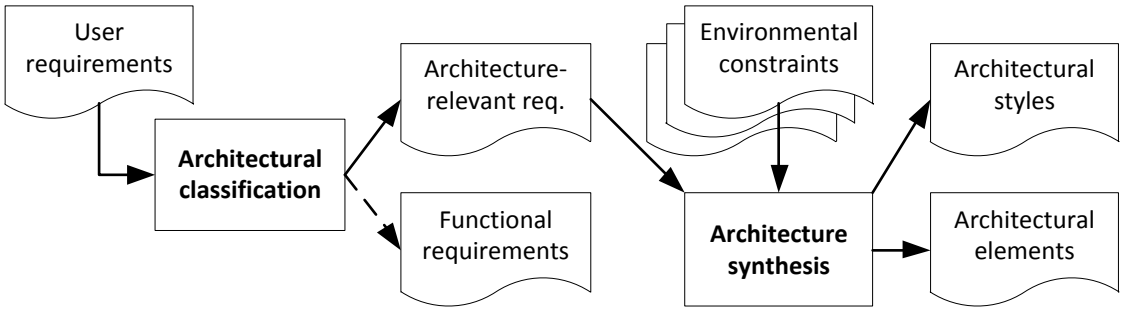

Figure 29 Requirements classification and architecture synthesis

The application of the requirements classification and architecture synthesis on the TME example is shown in the Table 12. The combination of architecture-relevant requirements and environmental constraints allow defining the architectural elements and patterns.

Table 12 TME architecture synthesis

Requirements	Environmental constraints	Architecture elements (Ae)	Architectural patterns (Ap)
R-05	Cs-04 Ch-04 Ci-03 Ci-04	**Ae-01:** Client application technology is Microsoft Silverlight 4.0	**Ap-01:** Analytical reporting
R-06	Ci-01 Ci-04 Ch-03 Cs-04	**Ae-02:** Server application is based on RIA WCF services	**Ap-02:** EAI/ESB
R-07	Ci-05 Cs-02	**Ae-03:** The Entity-Data-Model (EDM) is used as Object / Relation Mapping mechanism	**Ap-03:** TDS/OLTP
	Ch-03 Cs-01 Ci-05	**Ae-04:** Caching mechanism is used to preload data to the client application	
R-05	Cd-01 Ch-04	**Ae-05:** Telerik, a third-party graphical user interface components library is used	

The resulting list of architectural elements and styles is transferred to the next step of EBD-S – conflict identification, which is described in the following section.

5.5.2 Conflict identification and resolution

The analysis of the architectural elements and patterns allows to find and to resolve the possible conflicts within the system. Table 12 provides a solid basis for the software architectural analysis.

Full analysis of the interactions between requirements, environmental constraints and architectural elements and styles was performed. The determined architectural elements operate with each other with no conflicts; however, there is a difficulty in synthesis of the determined architectural patterns. Ap-01 (Analytical reporting) is not supported by the transactional data-store pattern, defined as Ap-3 (TDS/OLTP). Instead, Ap-3 provides access to a similar solution, called transactional reporting. Thus, it makes sense to replace Ap-1 with "transactional reporting" architectural pattern.

5.5.3 Architectural refinement of requirements

The architectural refinement of the requirements is the analytical review of the existing requirements and environmental constraints, which pursues the goal of model simplification and decomposition.

In the given example, total latency can be represented as sum of network delay and software delay. Thus, the architectural constraints Ch-03 (Network latency between server and client is 500-750 ms) and Ci-04 (Keep page update latency under 1 second) complement each other; as the result, a new software constraint is determined: Cs-05, shown in the Table 13. Cs-05 can replace both Ch-03 and Ci-04, which simplifies the architectural model of the system.

Table 13 Updated TME software environment constraints

Code	Description
Cs-01	Application shall run under Microsoft IIS 6.0 or later
Cs-02	Application shall use the MS SQL 2008 database
Cs-03	Email notifications shall be sent through MS Exchange 2007
Cs-04	Client application shall run in web-browser under MS Windows 2000 or later
Cs-05	**Software transaction delay shall not exceed 250 ms**

Next, the interconnections between the elements of the system are captured in the graph CBSP model, shown in Figure 30.

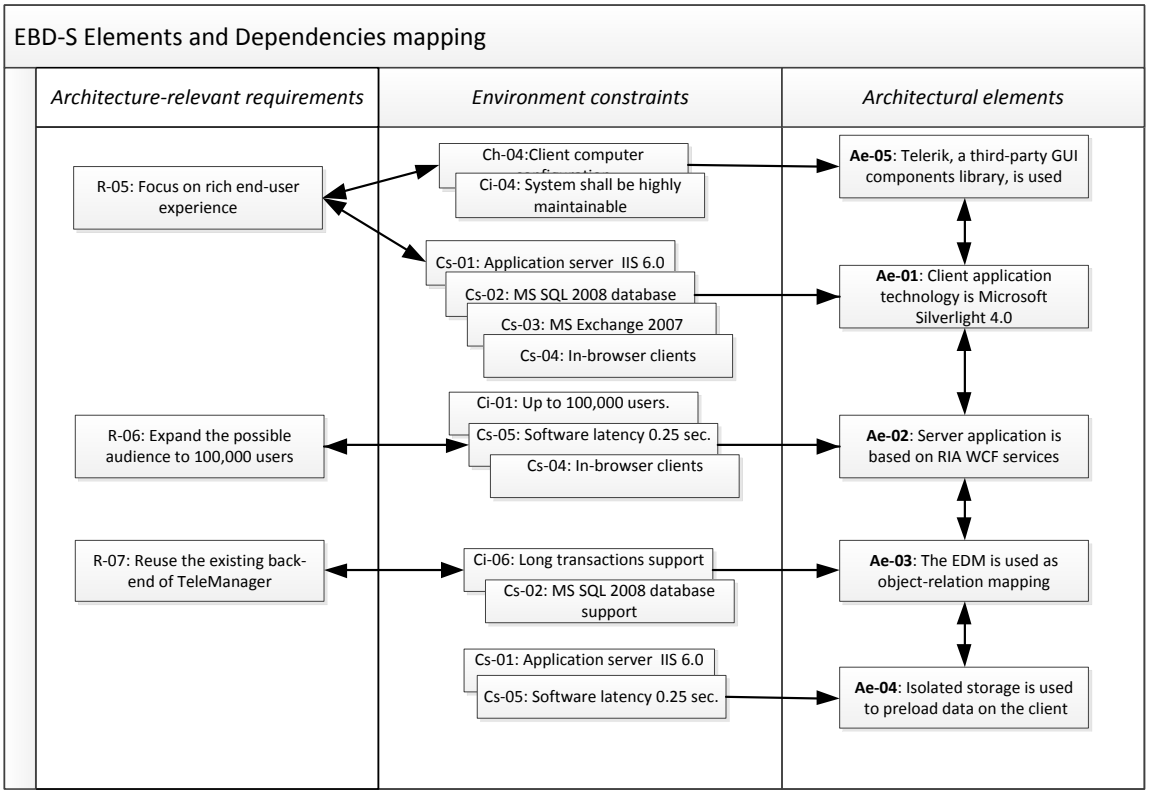

Figure 30 TME architectural mapping

At the same time, functional requirements are verified against the architectural model and decomposed according to the determined limitations. For example, the requirements associated with the R-02 (Build reporting engine) are shown in the Table 14.

Table 14 Requirements for the TME reporting engine

Item #	Requirement	Description
R-02.1	Flexible reports	Generation of reports by various filters should be easy to generate and should be reflective of the information they represent; Rather than a creating a module that tries to fit the information to it.
R-02.2	Report layout changes are dynamic by service / equipment record	Again, data or reports are generated with columns and titles that are dynamic. Example: *The client is not required to choose a series of filters based on a service.*
R-02.3	Reports / data is exportable, copiable, e-mailable and printable	Ensure that all data, including graphics are exportable, e-mailable and / or copiable to a clients' personal document.
R-02.4	Dynamic filtering	Data or reports, based on a pre-selected series of customer needs, should be filterable and hierarchical. Example: *Information displayed should be by overall category, service, etc* *Or* *Information displayed should be able to expand to the next level*
R-02.5	Information / data is expandable / hierarchical (one elements drops down to the next and the next).	The data or information should be expandable. Client can drill down to the next level or return backwards, as well he should be able to navigate between various information and types with simple icons and filtering.
R-02.6	Cost overview report	Shall reflect the consolidated information about cost distribution
R-02.7	Wireless cost report	Shall reflect the consolidated information about wireless-associated cost distribution
R-02.8	Cost comparison report	Shall reflect the cost overview and comparison by organization structure
R-02.9	Service types report	Shall reflect the information about service types in inventory and associated costs

The resulting CBSP model with the decomposed functional requirements is used in the next stage of EBD-S – design concept generation. This topic is covered in the next section of the case study.

5.6 TeleManager Executive: software concept generation

5.6.1 Design elements generation

CBSP model, built on the previous step, contains the refined architectural elements and functional requirements. This is a description of the problem to be solved. To generate a set of solutions to the design problem, software design elements shall be generated. Software design elements that address the requirements and architectural elements are proposed by the software development team members, based on their experience. The selected design elements are related to architectural elements / requirements.

The software concept generation phase of TME is based on the analysis of the EBD-S graph model, and involves the extraction of relationships between architecture / functional requirements and design elements. Figure 31 displays such a graph model, representing the TME v2.1 development.

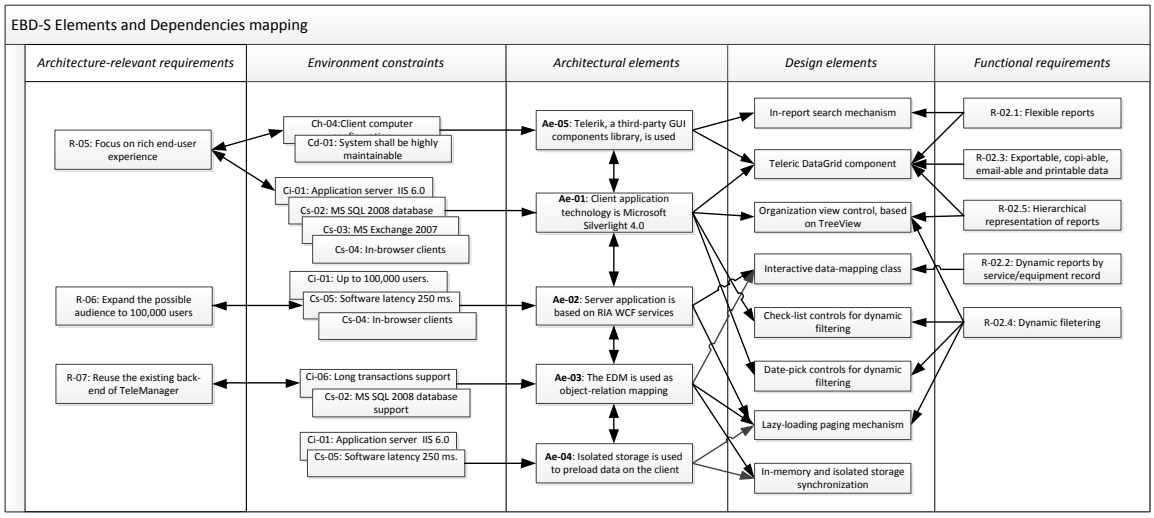

Figure 31 EBD-S elements and dependencies mapping

Analysis is performed with help of Design Matrix toolset. The correct representation of this model is done in the framework of non-binary Domain Mapping Matrix (DMM) – a rectangular matrix, that maps design elements to specific domains (architecture / functional requirements), and preserves the strength of the relation (0 / empty cell – no relation, 1 – weak relation, 2 – significant relation, 3 – very strong relation).

Architectural elements in EBD-S are considered to be technical domain requirements, and are associated with the DMM rows, as well as functional requirements. Design elements are mapped to the columns. The resulted matrix is shown in Figure 32.

	Search	Data-Grid	Org. View	Data-map	Check-list	Date-pick	Lazy loading	Cache
Ae-01		3	3		3	3		
Ae-02				3			2	
Ae-03				3			1	3
Ae-04							2	3
Ae-05	2	3						
R-02.1	2	2						
R-02.2				3				
R-02.3		3	3					
R-02.4			3		3	3		
R-02.5		3	3					
R-02.6	1	3					3	2
R-02.7	1	3					3	
R-02.8			2					
R-02.9	1	3					3	3

Figure 32 EBD-S software concept mapping matrix

Next the team needs to estimate the feasibility of the candidate solutions – it can be done by decomposition of the software design problem to sub-domains. The following section describes this process in details.

5.6.2 Concept development – decomposition

This software concept mapping matrix is decomposed with help of two-phase method for non-binary matrix decomposition, and the final solution is selected on the basis of resulting problem complexity.

On the first phase cluster formation and cluster alignment algorithms are applied to transform the initial matrix to a banded diagonal matrix, representing the sub-systems of the software system.

On the second phase we apply heuristic partitioning analysis to convert the banded diagonal matrix, obtained on the first phase, into possible block-angular matrixes, which would represent decomposition solutions of the design problem. These two phases are presented in Figure 33.

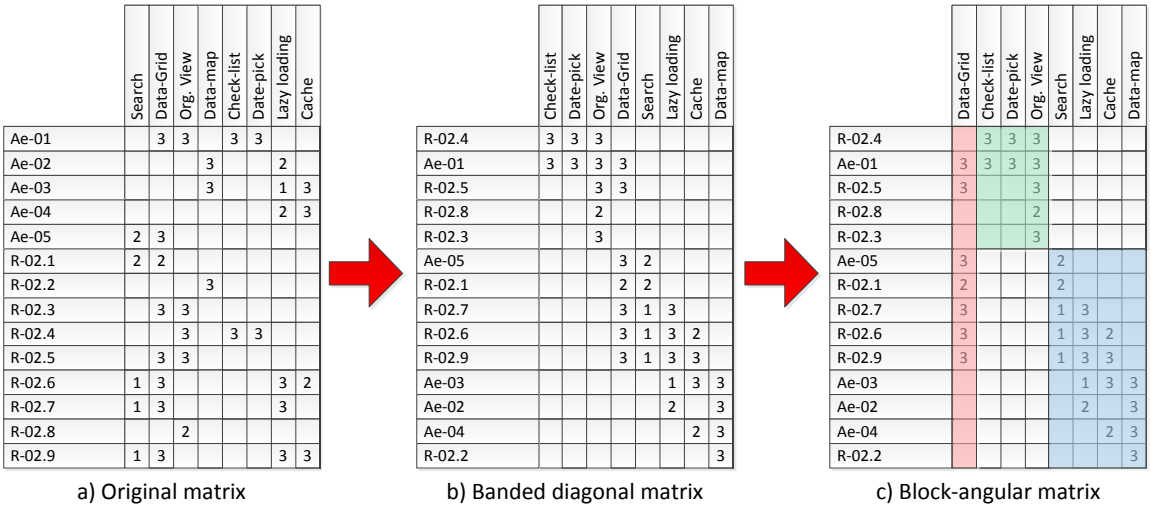

Figure 33 Software concept matrix decomposition

The part c) of the Figure 33 shows one of the resulting software concept solutions – one design element is common for the most of requirements, and there are two blocks of elements. If several acceptable solutions are generated by the two-phase algorithm, they

are analyzed and the most detailed decomposition is usually selected. The problem decomposition allows assigning independent parts of the problem to different development teams, working in parallel.

It resolves one of the most severe problems of agile development methods – low effectiveness in the teams, exceeding 5-7 members. The design problem can be split to several independent modules of approximately similar complexity, and the sub-teams can be formed to work on the modules.

In the TME v2.1 development two sub-teams were formed. They were working on two generated sub-problems after the architectural framework with Data-Grid connectivity (common to each sub-problem) had been developed.

5.7 TeleManager Executive: change and evolution control

The change control mechanism of EBD-S is based on the change impact analysis, performed with help of EBD-S graph model.

In the EBD-S abstract model there are three possible sources of unplanned changes:

- Source code
- Software model (architecture and design)
- Requirements

The strength of the impact is different on these levels, and usually is expressed in 10x costs growth per level. That is, software model changes cost ten times more than source code changes, and requirements changes cost ten times more than software model changes.

The EBD-S approach employs graphical impact analysis of the changes on any of these levels. Figure 34 shows an example of such analysis.

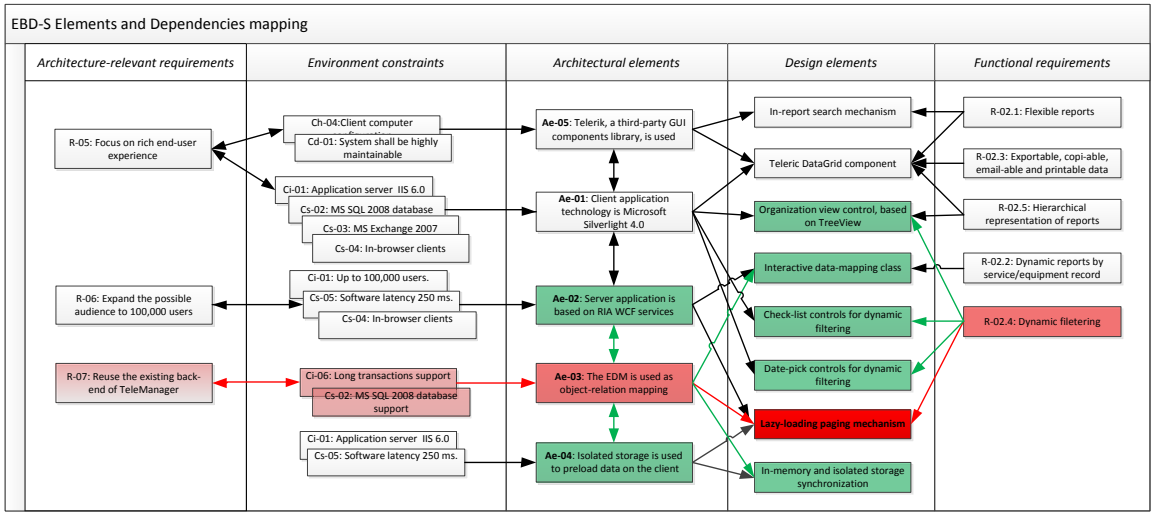

Figure 34 EBD-S change impact analysis

During the development stage, the Lazy-loading mechanism was implemented with errors, that weren't discovered by automated unit-tests. EBD-S graph model allowed to define the possible areas, related to the lazy-loading mechanism on all the levels of abstraction.

The impact chain is defined through all the abstraction levels (in red). It allows to find affected elements on the design level, related to the unstable code (in green), and retest the interfaces between the software modules (classes), associated with affected modules.

The same tool is used for change planning: each planned modification on any of the abstraction levels is analyzed on the basis of possible impact, caused by the modification.

5.8 EBD-S performance

5.8.1 Performance metrics

In order to estimate how the implementation of EBD-S affects the software development process, we recorded some process quality characteristics for two projects: before and after EBD-S implementation. To keep track of the project performance and retrieve the basis for the analysis, we use two types of quality metrics:

1. Process quality metrics, which reflect the efficiency of the development process;
2. Software quality metrics, which reflect the software code quality.

Process quality metrics are essentially related to the time, spent for specific activities:

- Estimated time for task / feature development;
- Real time spent on task / feature development;
- Lines of code per hour (LOC/h);
- Time spent on EBD-S process;
- Time for quality assurance – verification and validation;
- Time for quality assurance – correction.

The combination of these metrics, collected in consequent project iterations, allow understanding the impact of EBD-S implementation on the project performance.

Software quality metrics are based on two interrelated parameters: number of errors, discovered in the application, and overall codebase quality. Number of errors in the code (as well as number of error per 1,000 lines of code, Errors/KLOC) is a straight-forward metric; codebase quality metrics, however, require additional description.

TME codebase quality is controlled with help of Code Metrics, retrieved with help of Visual Studio Suite. Code Metrics is a tool that helps developers find and act upon complex and unmaintainable areas within the application source code. Visual Studio 2010 calculates five metrics of the source code, reviewed below.

1. **Class Coupling.** At each level, this indicates the total number of dependencies that the item has on other types. The higher this number, the more likely changes in other types will ripple through this item. A lower value at the type level can indicate candidates for possible reuse.
2. **Depth of Inheritance.** At the type level, depth of inheritance indicates the number of types that are above the type in the inheritance tree. Deep inheritance trees can indicate an over-engineering of a problem and can increase the complexity of testing and maintaining an application.
3. **Cyclomatic Complexity.** At each level, this measures the total number of individual paths through the code. This is basically calculated by counting the number of decision points and adding 1. This number is also a good indication on the number of unit tests it will take to achieve full line coverage. Lower is typically better.
4. **Lines of Code.** At each level, this is a measure of the total number of executable lines of code. This excludes white space, comments, braces and the declarations of members, types and namespaces themselves. Lower is typically better.
5. **Maintainability Index.** At the member and type level, this is an index from 0 to 100 indicating the overall maintainability of the member or type. This index is based on several other metrics, including Cyclomatic Complexity and Lines of Code. A low number (less than 80) indicates code that is complex and hard to maintain.

The key quality metrics that we use for the EBD-S performance estimation are:

- *Software process stage time over total project time ratio:* this metric show how the effort is spread in the course of the software development process, and helps to understand the overhead brought by EBD-S implementation, as well as the improvements in other stages.
- *Software process stage length, person/hours:* this metric show the recorded length of the project stages and allows for detailed comparison of the process performance.
- *Estimated development time to Real time spent ratio*: this metric reflects the effectiveness of software design activities, especially on the concept development stage of EBD-S.
- *Number of errors per 1000 lines of code (Error/KLOC)*: this metric can reveal the impact of well-thought and structured software design on the overall product quality.
- *Average time to fix one code error:* reflects the effectiveness of EBD-S change control mechanism.
- *Number of requirements errors, reported by clients*: shows the impact of EBD-S Environment Design stage of EBD-S.

5.8.2 Collection of results

One of the advantages of the centralized code control server, used in the TME development (Visual Studio Team Foundation 2010), is the automated collection of metrics.

Process quality metrics are collected according to the time, required for the code submission to the centralized code repository. Software quality metrics are recorded by the system in two ways:

1. Number of errors discovered is based on the number of work-orders, issued to the developers during the quality assurance stage;
2. Code metrics are calculated and recorded automatically on each software build.

The results, presented in this thesis, are collected for two iterations: one before the introduction of EBD-S, another after. That allows direct comparison of the results.

5.8.3 Results and analysis

TME is developed in Scrum environment, with interlaced release schedule: each even release is a beta-version, and each following release is shipped to the customers. Figure 35 reflects the timeline of four consequent Scrum iterations, united in two releases.

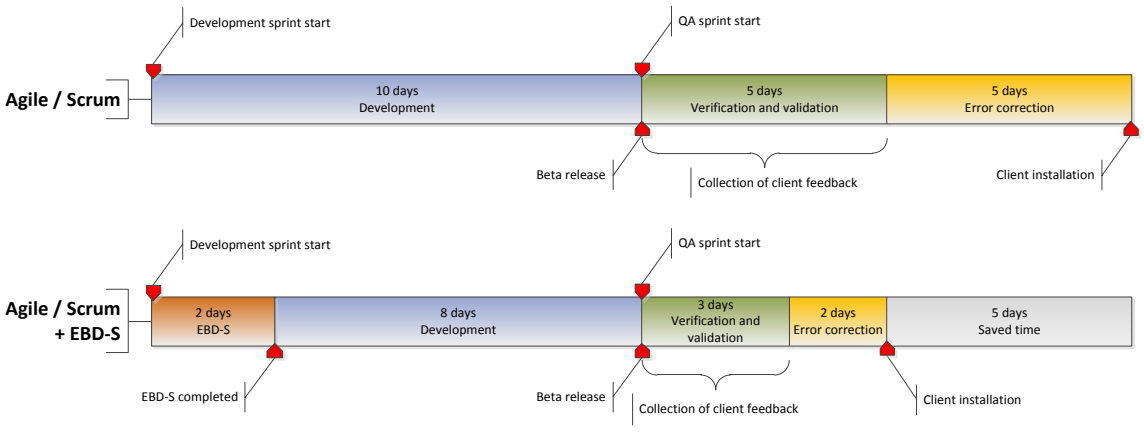

Figure 35 TME timeline

The upper timeline shows the standard Scrum process, without EBD-S application, when the lower reflects the project after implementation of EBD-S.

Both projects, Agile/Scrum and Agile/Scrum with EBD-S are based on the similar development workload, as estimated by developers. TME is developed by a team of 8 developers, which work 40 hours per week (640 person/hours per week allocated for one sprint). When requirements are selected from backlog, developers provide development time estimation in person/hours. Team lead ensures that team meets the development goals in the scrum interval.

To make a fair comparison, software development team selected two sets of software requirements to address in two consequent releases. The requirements were assessed by all team members according to implementation complexity, feasibility, availability of technology and overall implementation time, and were acknowledged as similar. The projects were named TME 2.1 for Scrum process and TME 2.3 for Scrum + EBD-S process. The version names 2.0 and 2.2 were reserved for beta versions of the software, used for initial client feedback collection.

The following process and product metrics were collected during implementation iterations, from project start to the release of beta-version (TME 2.0 and TME 2.2):

- Estimated time for project implementation
- Real time, spent for the implementation of features in the project
- Number of errors, discovered in the software at verification
- Density of errors (number of errors per 1000 lines of code)

Table 15 summarizes the cumulative TME 2.0 quality metrics, split per software project, according to the software structure.

Table 15 TME v2.0 (Scrum) quality metrics

Project	Estimated time, hours	Real time, hours	Errors	Errors/ KLOC
TME.ReportsProject	60	74.25	17	24.15
TME.Silverlight	240	288.5	71	22.18
TME.WcfService	80	82.5	12	12.66
TME.Web	150	166.5	39	18.67
PROJECT TOTAL	**530**	**611.75**	**139**	**20.02**

The difference in estimated time and real time for implementation shows that team lead shall plan a "safety cap" at 20% of the estimated project time to make sure that the project will be implemented in time.

Table 16 reflects the same set of quality metrics, collected from the EBD-S / Scrum development iteration.

Table 16 TME v2.2 (Scrum + EBD-S) quality metrics

Project	Estimated time, hours	Real time, hours	Errors	Errors/ KLOC
Etelesolv.Telemanager.Entity	8	8.25	2	15.75
Etelesolv.Telemanager.Membership	48	47.5	6	7.65
Etelesolv.Telemanager.Model	8	6.25	1	15.15
Etelesolv.Telemanager.TME	200	208.75	28	12.16
Etelesolv.Telemanager.TME.RIA	48	56.25	11	14.16
Etelesolv.Telemanager.TME.RIA.Web	40	38.5	5	11.06
Etelesolv.Telemanager.TME.Web	4	1.75	0	0
Etelesolv.Telemanager.Utility	40	36.5	4	8.2
Etelesolv.Telemanager.Utility	24	20.75	7	25.45
PROJECT TOTAL	**420**	**424.5**	**64**	**12.04**

The two main changes are evident from Table 16:

- The accuracy of time estimation is very high (implementation took 101% of the planned time versus 115% in the Scrum project)
- Number of development errors is significantly lower, as well as the error density.

Table 17 summarizes the key performance and quality metrics that were collected during two projects, labeled "Scrum" for ordinary agile Scrum process and "Scrum + EBD-S" for the agile Scrum process with EBD-S applied.

Table 17 EBD-S performance comparison

Metrics	Scrum	Scrum + EBD-S
Software process stage time, percent		
- Requirements analysis	2%*	7%
- Architecture and design	3%*	7%
- Coding	45%	52%
- Code verification and validation	25%	20%
- Error correcting	25%	14%
- Total	100%	100%
Software process stage time, person/hours		
- Requirements analysis	25	72
- Architecture and design	39	72
- Coding	576	496
- Code verification and validation	360	192
- Error correcting	360	128
- Total	1280	960
Real development time to estimated ratio	1.15	1.01
Number of errors per 1000 lines of code	20.02	12.04
Code maintainability index	84	85
Average time to fix one code error	2.3 hours	2 hours
Requirements errors, reported by clients	9	2

** Requirements analysis, architecture and design elaboration in Scrum project are included in Development stage and the durations are approximate.*

The results, reflected in Table 17, clearly demonstrate that implementation of EBD-S adds only a small time overhead (80 extra man-hours for the requirements and architecture analysis). But the advantages of this approach are significant: coding time of the comparable feature set is 14% lower than in regular Scrum process, verification and validation stage is 47% shorter, while error correction time savings are on the level of 65%. The overall project time saving, achieved with Scrum + EBD-S approach in TME project is 25% (960 hours versus 1280). Clients estimated the product, developed under EBD-S, as more relevant to their needs (2 reported missing requirements versus 9).

The implementation of EBD-S improves the coding aspects of the development. For instance, we collected the information about code submission during the coding stage of the project. This information is displayed on Figure 36.

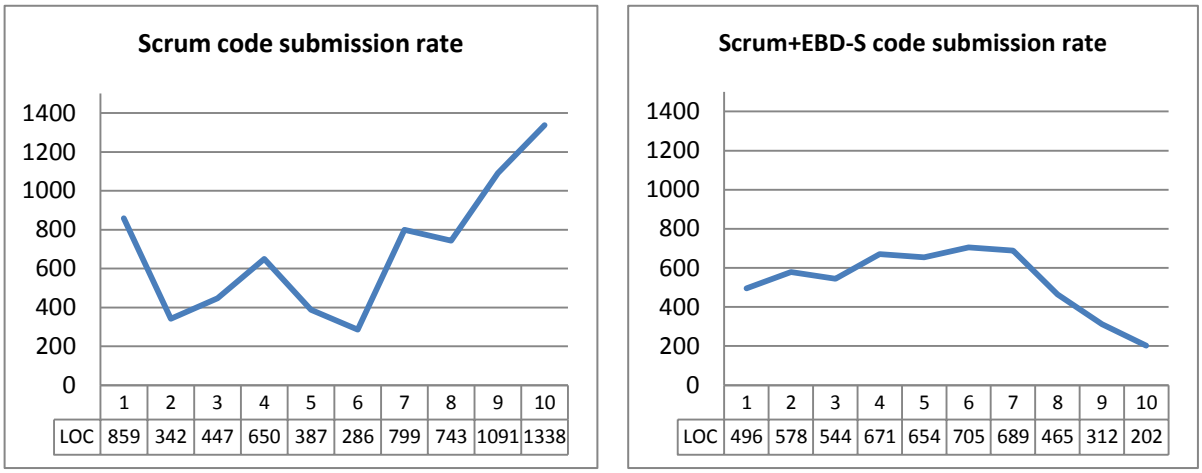

Figure 36 Code submission rates

Code submission trends, shown on Figure 36, demonstrate that in case of Scrum development, the majority of code is developed in the end of the sprint, while under Scrum/EBD-S development the code is created at the same rate during first 7 days of the sprint, and then developers refine the code and finalize the tasks without haste.

The code submission historical data, presented in Figure 37, clearly shows that the observed situation applies to Scrum projects, delivered by the same team in the past (Scrum Project #1 – #6). The rate of code submission varies a lot with the general trend of rising in the end of sprint.

The results of the next iteration of TME (v2.4), displayed in the last plot of the Figure 37, demonstrate that the effect of EBD-S application to Scrum project remains noticeable, resulting in low code submission variability and slight pace reduction in the end of sprint.

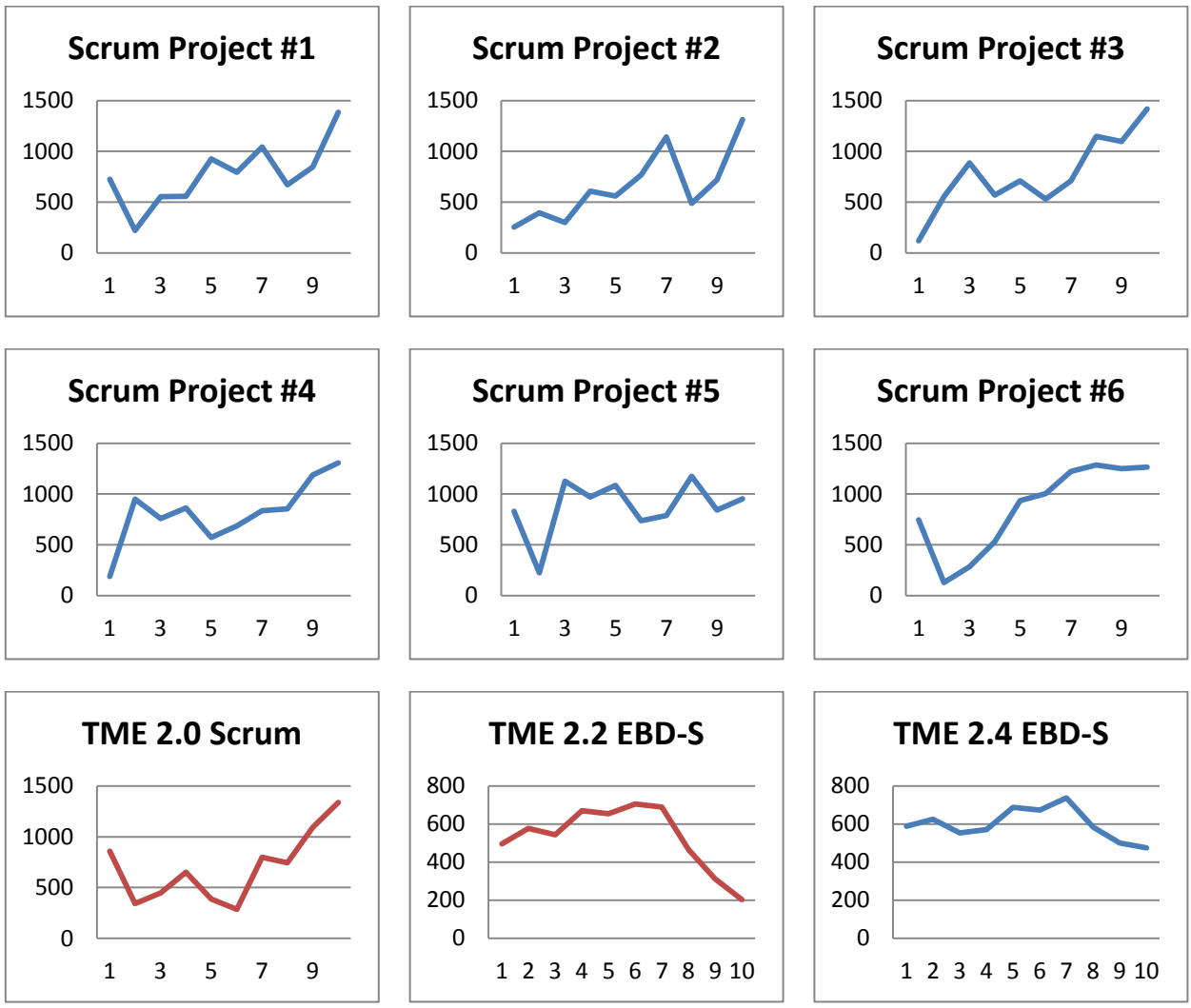

Figure 37 Code submission rate historical data

The performance of development in case of Scrum project averages at 11.35 LOC per hour, while the Scrum and EBD-S project demonstrated slightly higher, and, what is more important, more uniform performance at 12.52 LOC per hour.

In order to understand the reasons behind the improvement of coding performance and submission uniformity, code metrics of both TME 2.0 (Scrum) and TME 2.2 (Scrum/EBD-S) were analyzed.

Table 18 Code quality metrics for TME v2.0 (Scrum)

Project	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
TME.ReportsProject	24	4	4	53	704
TME.Silverlight	74	1577	7	352	3201
TME.WcfService	85	556	2	70	948
TME.Web	87	1136	4	151	2089
PROJECT TOTAL	**84**	**13039**	**2.17**	**4.21**	**6942**

Low maintainability index and relatively high cyclomatic complexity, observed in Table 18, are the results of ad-hoc development and fast architectural decisions.

Table 19 Code quality metrics for TME v2.2 (Scrum+EBD-S)

Project	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
Etelesolv.Telemanager.Entity	92	88	2	4	127
Etelesolv.Telemanager.Membership	75	239	3	41	784
Etelesolv.Telemanager.Model	94	55	2	4	66
Etelesolv.Telemanager.TME	82	1118	7	255	2302
Etelesolv.Telemanager.TME.RIA	84	375	2	41	777
Etelesolv.Telemanager.TME.RIA.Web	81	287	3	36	452
Etelesolv.Telemanager.TME.Web	91	32	3	20	45
Etelesolv.Telemanager.Utility	86	234	1	19	488
Etelesolv.Telemanager.Utility	73	67	1	9	275
PROJECT TOTAL	**85**	**10022**	**2.57**	**2.99**	**5316**

Under the EBD-S development umbrella, the code quality metrics demonstrate uniformity (see Table 19) – the maintainability index varies from 73 to 94 among the projects, averaging at 85. It represents a considerably lower variability compared to 24-87 results in TME v2.0. Higher depth of inheritance shows that object model is build better, with more classes reused. The 30% lower average class coupling demonstrates that in the TME v2.2 project the classes are better organized, and are easier to reuse.

It shows that EBD-S leads to a better-thought class design, simplifies the task of coding, and eliminates the weak spots in the project code.

5.8.4 Conclusion on EBD-S performance in TME project

Advantages of EBD-S approach, demonstrated in TME project, can be summarized as follows:

- More accurate development time estimation (less than 5% error margin instead of 20% in case of agile methods) due to:
  - Better understanding of the requirements, high-level and detailed system views, provided by Environment-Based analysis
  - Specification of architectural, functional and non-functional requirements leads to accurate problem decomposition
- 40% less errors in the code (12.04 errors per 1000 lines of code instead of 20.02) due to:
  - Clear and unambiguous requirements, leading to better understanding and planning of the functionality
  - Better structuration of the project due to requirements architecture-design coupling with Design Matrix
- 10% higher productivity (12.52 lines of code per hour instead of 11.35) due to:
  - Simpler code due to detailed functional design
  - Better project resource allocation
- Total project length (development + verification) is reduced by 25% due to:
  - CBSP-based impact analysis, which makes easier the discovery of possibly affected functions
  - Well-structured code, that requires less refactoring effort

6. Conclusions and future work

6.1 Conclusions

During the last decade, agile approaches dominate on the software development arena. They bring many advantages over the traditional approaches – faster development cycles, better interaction with clients, more frequent testing. At the same time, contemporary agile approaches have some flaws: they don't cover, or cover partially, process and project management aspects of software concept generation, architecture creation and after-implementation support. In this thesis the Environment-Based Design theory, reinforced with Design Matrix problem-solving and CBSP theory, is used to address the stated flaws of two agile approaches: Scrum and Feature-Driven Design. The proposed method is called Environment-Based Design of Software, and it provides methodological recommendations and structural foundation for the following aspects of software development (see Figure 38):

- Concept generation
- Architecture creation
- Design elaboration
- Post-implementation change control

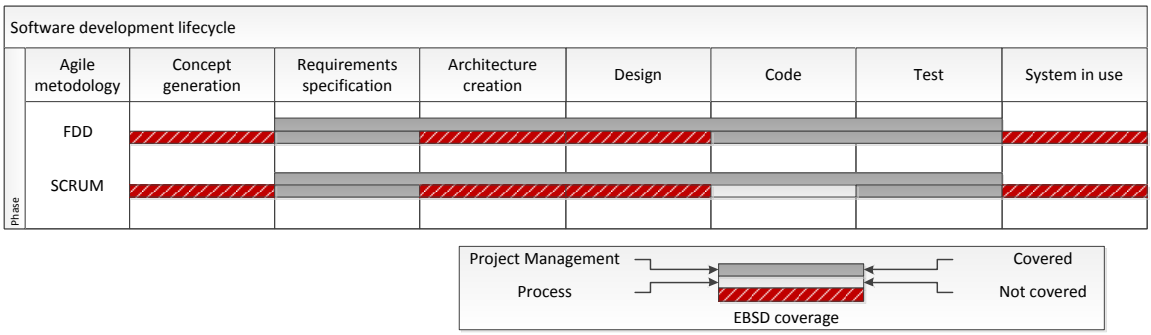

Figure 38 EBD-S application to FDD and Scrum software development methodologies

Through the application of Environment-Based Design of Software approach to the real-world software development process, described in Chapter 5, we come to the following conclusions:

- EBD-S is an effective and versatile design method, aimed to support the contemporary agile methodologies;
- EBD-S can be introduced to the software development process in steps, starting from Environment Analysis, through Design Matrix concept generation to CBSP change control;
- Each component of EBD-S brings specific advantages to the software development process, which allows controlling and monitoring the process of EBD-S introduction;
- The EBD-S application brings better product vision, more accurate development task estimation and significantly lower level of coding and requirements errors. As well it results in a higher coding performance. Thanks to all these improvements, overall project iteration length can be shortened by 25%.

6.2 Future work

The research presented in this thesis raised some problems still to be addressed. The implementation of the EBD-S to the real-world agile software development process provided a great opportunity to analyze the methodology from different points of view. We found that project managers lacked the planning techniques, relevant to EBD-S. We verified the applicability of the EBD-S to the software process, based on Scrum and Feature-Driven Development methodologies. However, one of the most promising software methodologies – Test-Driven Development – was omitted, since it was unknown to the developers in the case study environment.

The ongoing research indicates that methodological foundation of EBD-S can be reinforced with a set of advanced matrix-based techniques that would bring an extra dimension to the dependency analysis.

To summarize, in our future work we will investigate the following aspects:

- Possible extension of the EBD-S to the project management aspects of software development.
- Verify the real-world applicability of EBD-S to the Test-Driven Development, which implies the creation of automated tests before the code is written.
- Enhancement of the EBD-S concept generation technique with advanced matrix-based analysis, which takes into consideration different types of dependencies with various levels of strength.
- Enhancement of system-in-use software methods from current impact analysis to automated change plan generation, based on customer feedback.

Bibliography

Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2002). Agile software development methods: Review and analysis. *VTT Publications*(478).

Beck, K. (2001). *Manifesto for Agile Software Development*. Retrieved 03 15, 2011, from Agile Manifesto: http://agilemanifesto.org/

Brandozzi, M., & Perry, D. (2001). Transforming Goal-Oriented Requirement Specifications into Architecture Prescriptions. *Proceedings STRAW'01* (pp. 54-61). ICSE 2001.

Butler, M., Jones, C., Romanovsky, A., & Troubitsyna, E. (2006). Rigorous Development of Complex Fault-Tolerant Systems. *Springer, Lecture Notes in Computer Science, Vol. 4157*.

Chen, L., & Zeng, Y. (2009). Automatic generation of UML diagrams from product requirement requirements described by natural language. *The 2009 ASME International Design Engineering Technical Conferences (IDETC) and Computers and Information in Engineering Conference.* San Diego.

Chen, L., Ding, Z., & Li, S. (2005). A Formal Two-Phase Method for Decomposition of Complex Design Problems. *ASME Journal of Mechanical Design, Vol. 127*, 184-195.

Chen, Z., & Zeng, Y. (2006). Classification of Product Requirements Based on Product Environment. *Concurrent Engineering Research and Applications: an International Journal, Vol. 14*(No. 3), 219-230.

Chen, Z., Yao, S., Lin, J., Zeng, Y., & Eberlein, A. (2007). Formalisation of product requirements: from natural language descriptions to formal specifications. *International Journal of Manufacturing Research, Vol. 2*(No. 3), 362-387.

Chung, L., Nixon, B., Yu, E., & Mylopoulos, J. (2000). Non-Functional Requirements in Software Engineering. *Proceedings of the Second IEEE International Symposium on Requirements Engineering* (pp. 132-139). IEEE.

Cockburn, A., & Highsmith, J. (2001). Agile Software Development: The Business of Innovation. *IEEE Computer, Sept.*(9), 120-127.

Coplien, J. O. (1999). *Multi-paradigm design for C++*. Boston, MA: Addison-Wesley Longman Publishing Co.

Dorst, K., & Cross, N. (2001). Creativity in the design process: co-evolution of problem solution. *Design Studies, Vol. 22*(Nr. 5), 425-437.

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

Egyed, A., & Grunbacher, P. (2002). Automating Requirements Traceability: Beyond the Record & Replay Paradigm. *Proceedings. ASE 2002. 17th IEEE International Conference on: Automated Software Engineering* (pp. 163-171). Edinburgh: IEEE.

Fielden, G. (1975). *Engineering design.* London: British Standards Institution.

Klein, M. (1991). Supporting conflict resolution in cooperative design systems. *IEEE Transactions on Systems, Man and Cybernetics, Vol. 21*(Issue 6), 1379-1390.

Lan, C., & Ramesh, B. (2008). Agile Requirements Engineering Practices: An Empirical Study. *IEEE Software, Vol. 25*(Issue 1), 60-67.

Li, S. (2010). Extensions of the Two-Phase Method for Decomposition of Matrix-based Design Systems. *ASME Journal of Mechanical Design, Vol. 132*, 061003.

Medvidovic, N., & Taylor, R. (2002). A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering, Vol. 26*(Issue 1), 70–93.

Medvidovic, N., Egyed, A., & Grunbacher, P. (2003). Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery. *Second International SofTware Requirements to Architectures Workshop* (pp. 61-69). Portland, OR: ICSE.

Medvidovic, N., Rosenblum, D., & Taylor, R. (1999). A Language and Environment for Architecture-Based Software Development and Evolution. *Proceedings of the 1999 International Conference on Software Engineering*, (pp. 44-53). Los Angeles, CA.

Miller, G. (2001). The Characteristics of Agile Software Processes. *The 39th International Conference of Object-Oriented Languages and Systems*, (pp. 03-85). Santa Barbara, CA.

Nagel, E. (1961). *The Structure of Science: Problems in the Logic of Scientific Explanation.* Hackett Publishing Company, Inc.

Nandhakumar, J., & Avison, D. (1999). The Fiction of Methodological Development: A Field Study of Information Systems Development. *Information Technology & People, Vol. 12*(Issue 2), 176-191.

Nerur, S., Mahapatra, R., & Mangalaraj, G. (2005). Challenges of Migrating to Agile Methodologies. *Communications of the ACM - Adaptive complex enterprises, Vol. 48*(Issue 5), 73-87.

Neward, T. (2010). *Multiparadigmatic .NET, Part 2*. Retrieved 11 05, 2010, from msdn.microsoft.com: http://msdn.microsoft.com/en-us/magazine/gg232770.aspx

Nuseibeh, B. (2001). Weaving Together Requirements and Architectures. *IEEE Computer, Vol. 34*(Issue 3), 115-117.

Nuseibeh, B., & Easterbrook, S. (2000). Requirements Engineering: A Roadmap. *Proceedings on the Conference on The Future of Software Engineering* (pp. 34-46). New York: ACM.

Paetsch, F., Eberlein, A., & Maurer, F. (2003). Requirements Engineering and Agile Software Development. *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003.*, (pp. 308-313).

Pahl, G., & Beitz, W. (1988). *Engineering Design: A systematic approach.* Springer.

Perry, D., & Wolf, A. (1992). Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes, Vol. 17*(Issue 4), 40-52.

Peters, J., & Ramanna, S. (2003). Towards a Software Change Classification System: A Rough Set Approach. *Software Quality Journal, Vol. 11*(Issue 2), 121-147.

Pimmler, T., & Eppinger, S. (1994). Integration Analysis of Product Decompositions. *ASME Design Theory and Methodology Conference.* Minneapolis, MN.

Radice, R., Roth, N., O'Hara, A. J., & Ciarfella, W. (1985). A Programming Process Architecture. *IBM Systems Journal , Vol. 24*(Issue 2), 79–90.

Robertson, S., & Robertson, J. (2007). *Mastering the Requirements Process* (2nd ed.). Addison-Wesley Professional.

Shaw, M., & Garlan, D. (1996). *Software Architecture: Perspectives of an Emerging Discipline.* Prentice Hall.

Simon, H. (1996). *The Sciences of the Artificial* (3rd ed.). The MIT Press.

Steward, D. (1981). The Design Structure System: A Method for Managing the Design of Complex Systems. *IEEE Transactions on Engineering Management, Vol. 28*, 71-74.

Suh, N. (1990). *The Principles of Design.* Oxford University Press.

Tomiyama, T., Gu, P., Jin, Y., Lutters, D., Kind, C., & Kimura, E. (2009). Design methodologies: Industrial and educational applications. *CIRP Annals - Manufacturing Technology, Vol. 58*(Issue 2), 543-565.

Truex, D. (2000). Amethodical systems development: The deferred meaning of systems development methods. *Accounting, Management and Information Technologies, Vol. 10*(Issue 1), 53-79.

Ulman, D. G. (1995). Taxonomy for Classifying Engineering Decision Problems and Support Systems. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Vol. 9*, pp. 427-438.

Walls, Widmeyer, & Sawy. (1992). Building an Information System Design Theory for Vigilant EIS. *Information Systems Research, Vol. 3*(No. 1), 36-59.

Wang, M., & Zeng, Y. (2008). Asking the right questions to elicit product requirements. *22*(4), 283-293.

Wiegers, K. (2003). *Software Requirements, 2nd Edition.* Microsoft Press.

Yan, B., & Zeng, Y. (2009). The structure of design conflicts. *The 12th World Conference on Integrated Design & Process Technology.* Alabama.

Yoshikawa, H. (1981). General Design Theory and a CAD System. *Proceeding os the IFIP Working Group 5.2 Working Conference* (pp. 35-58). Amsterdam: IFIP.

Zeng, Y. (2002). Axiomatic Theory of Design Modeling. *Transactions of the SDPS: Journal of Integrated Design and Process Science, Vol. 6*(No. 3), 1-28.

Zeng, Y. (2004). Environment-Based formulation of design problem. *Transactions of the SDPS: Journal of Integrated Design and Process Science, Vol. 8*(No. 4), 45-63.

Zeng, Y., & Cheng, G. (1991). On the logic of design. *Design Studies, Vol. 12*(No. 3), 137-141.

Zeng, Y., & Gu, P. (1999). A science-based approach to product design theory Part I: Formulation and formalization of design process. *Robotics and Computer-Integrated Manufacturing, Vol. 15*(No. 4), 331-339.

Zeng, Y., & Gu, P. (1999). A science-based approach to product design theory Part II: Formulation of design requirements and products. *Robotics and Computer-Integrated Manufacturing, Vol. 15*(No. 4), 341-352.

Zeng, Y., & Gu, P. (2001). An Environment Decomposition-Based Approach to Design Concept Generation. *Interlnal Conference on Engineering Design*, (pp. 525-532).

Zeng, Y., & Jianliang, J. (1996). Computational model for design. *Proc. SPIE, 2644, 638.*