UNIVERSITÀ DEGLI STUDI DI ROMA TOR VERGATA

Facoltà di Ingegneria

Dottorato di Ricerca in Informatica e Ingegneria dell'Automazione

XX Ciclo

A Toolbox for Software Architecture Design

Davide Falessi

Tutor: Prof. Giovanni Cantone

Coordinatore: Prof. Daniel P. Bovet

"Few are those who see with their own eyes and fell with their own hearts." (A. Einstein)

Abstract

Software architecture has emerged as an important field of software engineering for managing the realm of large-system development and maintenance. The main intent of software architecture is to provide intellectual control over a sophisticated system enormous complexity. The key idea of this dissertation is that there is no silver bullet in software engineering, each method has pros and cons; the goodness of a method (tool, technique, etc.) varies based on the peculiarities of the application context.

According to a famous idiom: i) a poor craftsman blames his tool, ii) a really bad craftsman chooses the wrong tool for the job and then he blames the tool, iii) a good craftsman chooses and uses the tool well. While the software engineering community has been mainly focused on providing new methods, which usage are aimed/supposed to provide better results than older methods, there is a lack in helping the software practitioners in selecting/adapting the available tools. Hence, in this view, the contribution of the present dissertation, instead of being a new method for architectural design, which would have been easily forgotten in a bookcase, is a characterization of the existing methods. In other words, this dissertation provides a toolbox for software architecture design, from which software architects can select the best method to apply, according to the application context.

In this dissertation, the available architectural methods have been characterized by means of empirical studies. Unfortunately, the application of empirical methods on software architecture includes some troubles. A contribution of the present dissertation is a characterization of the available empirical methods by exposing their levels of challenges that researchers have to face when applying empiricism to software architecture. Such a proposed characterization should help to increase both the number and validity of software architecture empirical studies by allowing researchers to select the most suitable empirical method(s) to apply (i.e. the one with minor challenges), based on the application contexts (e.g. available software applications, architects, reviewers). However, in our view, in order to provide high levels of conclusion and internal validity, empirical methods for software

architecture should be oriented to take advantage of both quantitative and qualitative data. Moreover, based on the results from two experiments, the challenges, in conducting evidence-based software architecture investigations, might 1) highly influence the results of the empirical studies, and 2) be faced by empiricists' cleverness.

Architecting software system is a complex job and it encompasses several activities; this dissertation focuses on the following families of activities: software architecture design, resolving architectural tradeoffs, documenting design decisions, and enacting empirical studies on software architecture (as just described).

Regarding the resolution of architectural tradeoffs, based on our review of already proposed decision making techniques, we realized that no one of the available decision-making technique can be considered in general better than another; each technique has intrinsically its own level of complexity and proneness to specific problems. Since we cannot decide in advance what degree of complexity of modeling is sufficient, instead of proceeding by trial and error, we offered guidelines on which complexity to emphasize for avoiding specific problem(s). Our key idea is to expose and organize in a useful way, namely by a characterization schema, in what extent each decision-making technique is prone to specific problems. In this way, the level of proneness of specific technique to specific problems becomes a quality attribute of the decision-making technique. Furthermore, we situated in the proposed characterization schema eighteen different decision-making techniques already proposed by the literature in the domains of architecture design, COTS selection, and release planning. Such localization validates the completeness of the proposed characterization schema, and it provides a useful reference for analyzing the state of the art

Regarding software architecture design, this dissertation tried to answer to following question: "Do actual software architecture design methods meet architects needs?" To do so, we provide a characterization of the available methods by defining nine categories of software architects' needs, proposing an ordinal scale for evaluating the degree to which a given software architecture design method meets the needs, and then applying this to a set of software architecture design methods. Based on results from the present study, we argue that there

are two opposite but valid answers to the aforementioned question: a) Yes, they do. In fact, we showed that one or more software architecture design methods are able to meet each individual architect needs that we considered. b) No, they do not. In fact, we showed that there is no software architecture design method that is able to meet any tuple of seven or more needs, which means that there is still some work to do to improve software architecture design methods to actually help architects. In order to provide directions for software architecture design method improvement, we presented couples of needs, and triplets of needs that actual software architecture design methods are unable to meet. Moreover, an architect can use such characterization to choose the software architecture design method which better meets his needs.

Regarding design decision documentation, we conducted a controlled experiment for analyzing the impact of documenting design decisions rationale on effectiveness and efficiency of individual/team decision-making in presence of requirement changes. Main results show that, for both individual and team-based decision-making, effectiveness significantly improves, while efficiency remains unaltered, when decision-makers are allowed to use, rather not use, the proposed design rationale documentation technique. Being sure that documenting design-decisions rationale does help, we argued why it is not used in practice and what we can do to facilitate its usage. Older design decisions rationale documentation methods aimed at maximizing the consumer (documentation reader) benefits by forcing the producer (documentation writer) to document all the potential useful information; they eventually ran into too many inhibitors to be used in practice. In this dissertation we propose a value-based approach for documenting the reasons behind design decision, which is based on a priori understanding of who will benefit later on, from what set of information, and in which amount. Such a value-based approach for documenting the reasons behind design decision offers means to mitigate all the known inhibitors and it is based on the hypothesis that the set of required information depends on the purpose (use case) of the documentation. In order to validate such a hypothesis we ran an experiment in a controlled environment, employing fifty subjects, twenty-five decisions, and five different purposes (documentation use case) of the documentation. Each subjects practically used the documentation to enact all the five documentation use case(s) by providing an answer and a level of utility for each

category of information in the provided documentation. Both descriptive and statistical results confirm our expectancies that the level of utility, related to the same category of information in the design decision rationale documentation, significantly changes according to the purpose of the documentation. Such result is novel and implies that the consumer of the rationale documentation requires, or not, a specific category of information according the specific purpose of the documentation. Consequently, empirical results suggest that the producer can tailor the design decision rationale documentation by including only the information required for the expected purposes of the documentation. This result demonstrates the feasibility of our proposed value-based approach for documenting the reasons behind design decision.

Acknowledgment

The present dissertation represents a huge milestone from both a professional and human points of view. As a matter of fact, during these three years of PhD course I changed, hopefully in a better way, my *forma menti*s; it is by being in contact with different civilizations that I had the chance to observe and touch different ways to live. Such a PhD course wasn't easy; form a professional point of view it was quite difficult to enter in the research environment because it is characterized by strong competition and strictness, from a human point of view it was hard to be far away form my near and dear. However, now I think that such past difficulties are paying off. Despite I am the main protagonist of this successful path, it is successful at least in my opinion (which is what really matter), there are several people that I want to thank you, without which all of this would not have been possible:

- My family for providing a constant and huge support (not only economic),
- Francesca for being part of my life from more than eight years,
- Giovanni Cantone, for i) believing in my potentiality, ii) providing a constant mentoring during the entire PhD course, iii) not taking advantage of me at all.
- Professor Philippe Kruchten, for mentoring and hospitality during my six months visit to University of British Columbia, Vancouver, Canada.
- Professor H. Dieter Rombach, for suggestions and hospitality during my six moths visit to Fraunhofer Institute for Experimental Software Engineering and University of Kaiserslautern, Germany.
- Dr. Martin Becker and Dr. Dirk Muthig for the professional support during my six moths visit to Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany.
- Professor Daniel P. Bovet for his support as director of the PhD course in "Informatica ed Ingegneria dell'Automaizone".

List of publications

Publications related to this dissertation

The present dissertation addresses two main aspects of architectural design: Architectural design methods and Architectural design decision rationale documentation.

Publications related to software architecture design methods include:

- Falessi, D., Cantone, G., and Kruchten, P. 2008. *Value-Based Design Decision Rationale Documentation: Principles and Empirical Feasibility Study*. Seventh Working IEEE / IFIP Conference on Software Architecture (WICSA 2008). Vancouver, Canada. IEEE Computer Society.
- Falessi, D., Cantone, G., and Kruchten, P. 2007. *Do Architecture Design Methods Meet Architects' Needs?* Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture. Mumbai, India. IEEE Computer Society.
- Falessi, D., Kruchten, P., and Cantone, G. 2007. *Issues in Applying Empirical Software Engineering to Software Architecture*. First European Conference on Software Architecture. Aranjunez, Spain. Springer.
- Falessi, D., Kruchten, P., and Cantone, G. 2007. Resolving Tradeoffs in Architecture Design: A Characterization Schema of Decision-making Techniques, SUBMITTED TO Journal of Systems and Software.
- Falessi, D., Kruchten, P., and Giovanni, C. 2007. *A Strategy for Applying Empirical Software Engineering to Software Architecture*, SUBMITTED TO Journal of Systems and Software.
- Publications related to software architecture design decision rationale documentation:

- Falessi, D., Cantone, G., and Becker, M. 2006. *Documenting Design Decision Rationale to Improve Individual and Team Design Decision Making: An Experimental Evaluation*. International Symposium on Empirical Software Engineering. Rio De Janeiro, Brazil.
- Falessi, D., Becker, M., and Cantone, G. 2006. *Design decision rationale: experiences and steps ahead towards systematic use*, SHaring and reusing Architectural Knowledge Workshop (SHARK'2006) at the International Conference on Software Reuse, ACM SIGSOFT Software Engineering Notes, 31 (5).
- Falessi, D. and Becker, M. 2006. Documenting Design Decisions: A Framework and its Analysis in the Ambient Intelligence Domain. BelAmI-Report 005.06/E, Fraunhofer IESE.

Other publications not strictly related to this dissertation

- Falessi, D. and Cantone, G. 2006. *Exploring Feasibility of Software Defects Orthogonal Classification*. International Conference on Software and Data Technologies (Best Student Paper Award) Setubal, Portugal.
- Falessi, D., Cantone, G., and Grande, C. 2007. *A COMPARISON OF STRUCTURED ANALYSIS AND OBJECT ORIENTED ANALYSIS: An Experimental Study*. International Conference on Software and Data Technologies. Barcelona, Spain.
- Falessi, D., Pennella, G., and Cantone, G. 2005. Experiences, Strategies and Challenges in Adapting PVM to VxWorks(TM) Hard Real-Time Operating System, for Safety-Critical Software. 12th European PVM/MPI Users' Group Meeting. Sorrento, Italy. Springer.

Other activities

- Member of the Program Committee for PROFES08, the 6th International Conference on Product Focused Software Process Improvement, 23-25 June 2008, Rome Hills, Italy.
- Member of the Program Committee for ICSOFT08, the 3rd International Conference on Software and Data Technologies, 5-8 July 2008, Porto, Portugal
- Publicity Chair for WICSA08, the Working IEEE/IFIP Conference on Software Architecture, 18 - 22 February 2008, Vancouver, BC, Canada.
- Local arrangement Chair for PROFES08 the 6th International Conference on Product Focused Software Process Improvement, 23-25 June 2008, Rome Hills, Italy.
- IEEE reviewer of the ANSI/IEEE Std 1471, Recommended Practice for Architectural Description of Software-intensive Systems, which is now also ISO/IEC 42010

Table of Contents

List of figures

List of tables

1. Introduction

1.1 Research objective and approach

Architecting software system is a complex job and it encompasses several activities; this PhD dissertation focuses on the following families of activities: software architecture design, resolving architectural tradeoffs, documenting design decisions, and enacting empirical studies on software architecture. The key idea of this dissertation is that there is no silver bullet in software engineering, each method has pros and cons; the goodness of a method (tool, technique, etc.) varies based on the peculiarities of the application context.

According to a famous idiom: i) a poor craftsman blames his tool, ii) a really bad craftsman chooses the wrong tool for the job and then he blames the tool, iii) a good craftsman chooses and uses the tool well. While the software engineering community has been mainly focused on providing new methods, which usage are aimed/supposed to provide better results than older methods, there is a lack in helping the software practitioners in selecting/adapting the available tools. Hence, in this view, the objective of the present dissertation, instead of being a proposal for a new method for architectural design, which would have been easily forgotten in a bookcase, is a characterization of the existing methods. In other words, this dissertation provides a toolbox for software architecture design, from which software architects can select the best method to apply, according to the application context.

The characterizations of the architectural methods have been done by means of empirical studies; unfortunately, the application of empirical methods on software architecture includes some troubles. Another contribution of the present dissertation is a characterization of the available empirical methods by exposing their levels of challenges that researchers have to face when applying empiricism to software architecture.

1.2 Dissertation Outline

Toolboxes are usually made up of drawers, each drawer contains several instances sharing the same functional characteristic; such instances are organized according to a specific quality attribute. For example, my father's toolbox is organized in four main drawers; the first drawer contains several screwdrivers arranged based on the size. This dissertation provides a toolbox for software architecture design and its outline is like the one of a real toolbox; in fact, each chapter contains several methods to address the same activity and they are organized according to a specific quality attribute. Chapter 2 encompasses available empirical methods; such methods are organized based on their levels of challenges posed when being applied to software architecture. Chapter 3 encompasses the available software architecture design methods; such methods are organized based on their ability to support specific architects' needs. Chapter 4 encompasses available decision-making techniques for addressing architectural tradeoffs; such techniques are organized based on their levels of proneness to specific troubles. Chapter 6 encompasses the categories of information included in the documentation of design decision rationale; such categories of information are organized based on their level of utility when used for a specific purpose. Chapter 5 validates the importance of documenting design decision rationale (the instances encompassed in Chapter 6). Table 1-1 summarizes the dissertation outline and related publications.

Table 1-1: Dissertation outline.

The Toolbox for Software Architecture Design			
Functional characteristic	**Quality attribute**	**Pubblication**	**Chapter**
Empirical method	Level of challenges	* *Issues in Applying Empirical Software Engineering to Software Architecture*, First European Conference on Software Architecture. Aranjunez, Spain, 2007. ** *A Strategy for Applying Empirical Software Engineering to Software Architecture*, Journal of Systems and Software (SUBMITTED TO)	2
Software arcitecture design method	Ability to support specific architects' needs	* *Do Architecture Design Methods Meet Architects' Needs?*, Sixth Working IEEE/IFIP Conference on Software Architecture. Mumbai, India. 2007.	3
Decision-making technique for resolving architectural tradeoffs	Level of proneness to specific troubles	* *Resolving Tradeoffs in Architecture Design: A Characterization Schema of Decision-making Techniques*, Journal of Systems and Software (SUBMITTED TO)	4
Category of information in the design decision rationale documentation	Level of utility for a specific purpose	* *Documenting Design Decision Rationale to Improve Individual and Team Design Decision Making: An Experimental Evaluation*, International Symposium on Empirical Software Engineering, Rio De Janeiro, Brazil, 2006 ** *Value-Based Design Decision Rationale Documentation: Principles and Empirical Feasibility Study*, Seventh Working IEEE/IFIP Conference on Software Architecture. Vancouver, Canada, 2008. *** *Design decision rationale: experiences and steps ahead towards systematic use*, SHaring and reusing Architectural Knowledge Workshop at the International Conference on Software Reuse, 2006	5,6

1.3 Overview of Research Domain

1.3.1 Software Architecture

Software architecture has emerged [Shaw and Clements, 2006] as an important [Booch, 2007] field of software engineering for managing the realm of large-system development and maintenance [Clements et al., 2002]. The main intent of software architecture is to provide intellectual control over a sophisticated system enormous complexity [Kruchten et al., 2006b]. There are several definitions of software architecture [SEI, 2007]; one of the most used is the set of significant decisions about the organization of a software system: selection of the structural elements and their interfaces by which a system is composed, behavior as specified in collaborations among those elements, composition of these structural and behavioral elements into larger subsystem, architectural style that guides this organization. Software architecture also involves usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and tradeoffs, and aesthetic concerns [Kruchten, 2003] [Shaw and Garlan, 1996].

The word software architecture is now used everywhere, reflecting a growing concern and attention [Kruchten, 2003]. However various stakeholders have different concerns and are interested in different aspects of the architecture [Kruchten, 2003]. For this reasons, its meaning [Smolander, 2002] and representation varies for different set of stakeholder. The key idea is that a software architecture is a complex entity that cannot be described in a simple one-dimensional fashion [Clements et al., 2002]. An architectural view is the representation of a whole system from the perspective of a related set of architectural concerns; hence architectural views help architects to keep separated the different concerns of interests. The software community as a whole seems to agree that using multiple coordinated views to describe an SA provides high benefits; the literature provides predefined views (e.g. [Kruchten, 1995a]) and guidelines for their definition (e.g. [Clements et al., 2002] [Rozanski and Woods, 2005]).

There is a difference between architecture and design: the former is part of the latter. In fact, architecting is the activities of conceiving, defining, documenting, maintaining, improving, and certifying proper

implementation of an architecture [Ieee, 2000]. Thus, many design decisions are left unbound by the architecture and are happily left to the discretion and good judgment of downstream designers and implementers [Clements et al., 2002].

Software architecture is developed during the early phases of the development process; it hugely constraints or facilitates the achievement of specific functional requirements, non-functional requirements, and business goals. Hence, reviewing the software architecture represents a valid means to check the conformance of the system and to reveal early any potentially missed objective [Maranzano et al., 2005].

Each software system has a software architecture; it can be implicit (incidental) or explicit i.e. documented and specifically designed to fulfill predefined business objective or non functional requirements. In case of maintenance operation, since software architecture is key to manage large software applications, it may be the case to spend effort in documenting the underlying software architecture even if it has been to recover [Ding and Medvidovic, 2001].

1.3.2 Software architecture design

Software architecture design methods (SADM) focus on deriving a software architecture from software requirements. Difficulties in designing an architecture include [Grüenbacher et al., 2003]:

Requirements are frequently captured informally while software architecture is specified in some formal manner. Consequently, there is a semantic gap to deal with.

Non-functional requirements are difficult to specify in an architectural model.

Software is often developed in an iterative way. Certain types of requirements cannot be well defined in the early development phases [Nuseibeh, 2001a]. This implies that sometimes architects take their decisions based upon vague requirements.

Stakeholders dealing with software architecture issues view the system from different perspectives. This usually implies conflicting goals, expectations, and used terminology.

However, there are three main sources of architecture [Kruchten, 1995b]:

Reuse: architecting is a difficult task; the fact that a system was successful and it used a certain component, rule or responsibility demands for the reuse of such issues. Source of reuse can be the earlier version of the system, another system sharing key characteristics, architectural pattern [America et al., 2003]. Domain-specific Software Architecture [Li et al., 1999] and Product Line Architecture [Clements and Northrop, 2002] are different examples of two perfect marriages between software architecture and reuse.

Method: language, process model, and a systematic and conscious related technique for bridging the gap between software architecture and requirements. The present chapter deals with such an issue.

Intuition: invention of software architecture elements based on experience.

The ratio among the 3 software architecture sources is dependent on the experience of architects and the novelty of the domain.

In general, every approach aimed to solve a problem enacts an iterative process consisting in the following three phases (see Figure 1-1):

Understand the problem: Such phase consists in analyzing the problem and extract the real needs from the ambiguous and huge problem description.

Solve the problem: Provide a solution as well as possible by analyzing the abovementioned needs and the characteristics of known solutions.

Evaluate the solution: Decide if the provided solution solves the problem.

From phase 3, the process re-enters phases 1 or 2 when the provided solution is not acceptable but the problem is still considered feasible. The process ends when the provided solution is considered acceptable. There is also the case in which the problem is evaluated as unfeasible and the process continues by changing the system requirements; this case is rare and it is not included in Figure 1-1.

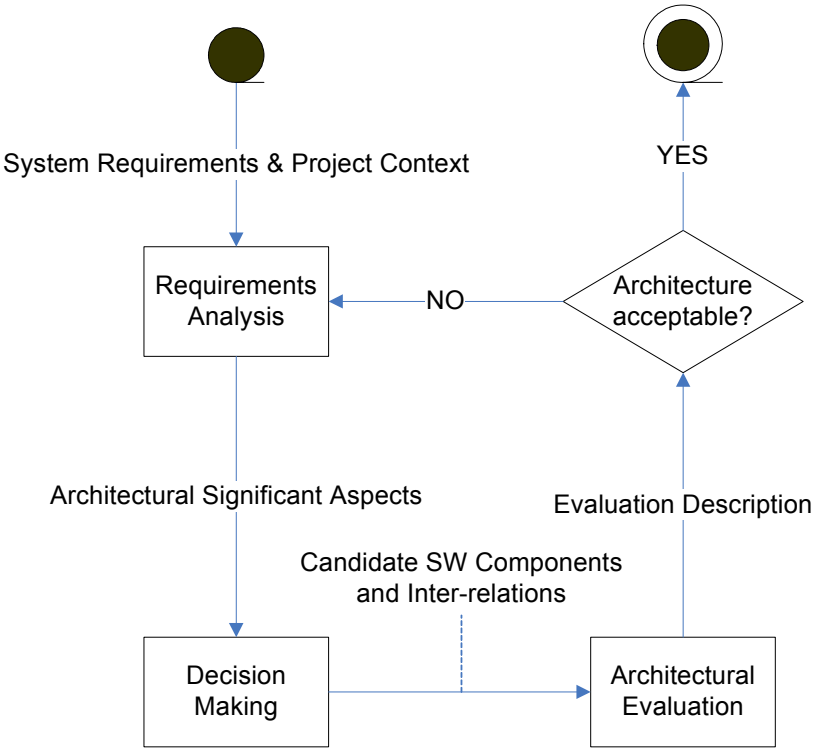

Figure 1-1: Problem solving activities and architectural design activities.

In the SADM's context, the problem consists in the system requirements, and the solution consists in the characteristics of the software components and their inter-relations. Despite the core of SADM is intrinsically the phase number 2, experience shows that such phase, in order to be enacted successfully, should be [Hofmeister et al., 2007] carefully integrated in the above-mentioned process (see Figure 1-1). Fowler [1997] and Roshandel [2004] confirm this structure by arguing the importance in SADM of the phases 1 and 3 respectively.

Despite the fact that "many of the design methods were developed independently, their descriptions use different vocabulary and appear quite different from each other, … they have a lot in common at the conceptual level." [Hofmeister et al., 2007]. Differences among SADMs

include the level of granularity of the decision to make, concepts taken into account, level of emphasis on phases, audience (large vs. small organization) and usage domain. A sound discussion regarding SADM variability and commonalities is out of the scope of the present work and can be found in [Hofmeister et al., 2007].

1.3.3 Design decision rationale documentation

During the architecture development process, many important decisions are not documented explicitly, together with their rationale, but are immediately embedded in the models the architects build [Tyree and Akerman, 2005]; consequently, some useful knowledge attached to the decision and the decision process are lost forever [Kruchten, 2004]. There is an erosion of the design and imply high costs of software architecture change [Jansen and Bosch, 2005] because decisions to make cannot reason on made decisions.

There are many definitions of design rationale. According to Jintae Lee, "design rationales include not only the reasons behind a design decision but also the justification for it, the other alternatives considered, the tradeoffs evaluated, and the argumentation that led to the decision" [Lee, 1997]. Nowadays, the use of design decision rationale documentation seems to be promising to support software architecture design and, more important, its maintenance; however, it is not applied in practice due to the existence of some inhibitors including bad timing, delayed benefit, information unpredictability, overhead, unclear benefit, lack of motivation, lack of maturity, potential inconsistencies.

Depending on the category of information documented, DDRD can be based on argumentation, history, device, process and active-document [Burge and Brown, 1998]. In the argumentation-based DDRD, design rationale is principally used to represent the arguments that characterize a design, like issues raised, alternative responses to these issues and arguments for and against each alternative. Prominent argument-based DDRD techniques are gIBIS [Conklin and Begeman, 1988], DRL [Lee, 1997] and QOC [MacLean et al., 1996]

2. A Characterization of Empirical Software Engineering Methods in Being Applied on Software Architecture

Empirical software engineering focuses on the evaluation of software engineering entities, such as methods, processes and tools, by analyzing related sets of data that are available in organizations repositories and/or result from surveys, case studies or controlled experiments. This particular approach to software engineering has contributed a valuable body of knowledge in several areas such as Software Economics and Software Quality, which in turn drove significant improvements in related tools and techniques. Unfortunately this is not (yet) the case for software architecture, where empirical studies are still scant both in number and validity of their results. In order to promote and facilitate advances in state of the art in software architecture by means of empirical studies, the aim of this chapter is to characterize the available empirical methods by exposing their levels of challenges that researchers have to face when applying empiricism to software architecture. This characterization provides a strategy to increase both the number and validity of empirical studies on software architecture by allowing the researchers to select the most suitable empirical methods to apply (i.e. the one with minor challenges), based on the application contexts. The chapter concludes by describing some practical experiences in facing those challenges.

2.1 Introduction

One of the objectives of Empirical Software Engineering (ESE) is to develop and utilize evidence to advance software engineering methods, processes, techniques, and tools. According to Basili [1996] "like physics, medicine, manufacturing, and many other disciplines, software engineering requires the same high level approach for evolving the knowledge of the discipline; the cycle of model building, experimentation and learning. We cannot rely solely on observation followed by logical thought." One of the main reasons for carrying out empirical studies is the opportunity of getting objective measures (e.g.

in the form of statistically significant results) regarding the performances of a software development technology [Wohlin et al., 2000].

Several studies in the past have stressed the need for empiricism in software engineering [Basili, 1986] and the absence of validation data in major publications [Zelkowitz and Wallace, 1998]. In the last two decades, ESE achieved considerable results in building valuable bodies of knowledge, which in turn drove important advances in different areas of software engineering. For instance, the effective application of ESE gave excellent results in the area of Software Economics [Boehm, 1981] and of Value-based Software Engineering [Biffl et al., 2005], and it improved the techniques to inspect software artifacts for defects detection [Shull et al., 2006] [Vegas and Basili, 2005] [Abdelnabi et al., 2004]. Nowadays, we can consider ESE a maturing discipline as demonstrated by a growing consensus in the software research community, by the number of research papers accepted for presentation and publication in major software conferences and journals, and by a number of ESE specialized conferences, journals, and books, e.g. [International Symposium on Empirical Software Engineering and Measurement 2007], [Basili and Briand], [Wohlin et al., 2000] and [Juristo and Moreno, 2006].

Sharing architectural knowledge seems to be one of the most promising next steps for advancing research and improving our knowledge about software architecture. This is attested by a number of workshops [Avgeriou et al., 2007], publications [Kruchten et al., 2006a], and international projects [van Vliet and Hammer, 2005]. The empirical approach seems to be a good way to achieve this knowledge sharing. Since the importance of having “good” software architecture has already been demonstrated [Booch, 2007], we should then take care about the quality of the methods for designing, analyzing, and evolving software architecture.

The present work recognizes that, despite ESE seems to be a valid means to improve software architecture maturity and knowledge sharing, the existing literature exhibits very few empirical studies on software architecture. In particular, analyzing the last three issues of the working IEEE/IFIP conference on software architecture, we found just one experimental study [Ferrari and Madhavji, 2007] out of 130 articles.

In this work we propose a characterization scheme, which should help to increase the number and validity of empirical studies on software architecture by allowing empiricists to focus on important architectural issues (i.e., current architectural challenges, see Section 2.2), and architects to select the most suitable empirical methods, given a specific context (see Section 3).

Since this work is in essence a contextualization of ESE principles to software architecture, then the content of this chapter is to consider as complementary to, a specialization of, past general ESE works, such as introductions [Wohlin et al., 2000] [Juristo and Moreno, 2006], characterizations [Kitchenham, 1996], [Zelkowitz and Wallace, 1998], and prospective works [Basili, 1996] [Sjøberg et al., 2007].

To avoid misunderstandings, we adopt here the following terminology:

Software engineering method: It is a method to develop one or more artifacts related to a software system. In this context, such artifact is something strictly related to software architecture, like architecture description languages (ADL), tools, etc.

Empirical method: It is a method in the empirical software engineering domain to gain and analyze the data (a.k.a., "the empirical data") resulting from the application of one or more "factors" (e.g., software engineering method) to a particular "object" (i.e., a software project), in order to validate one or more hypotheses about the factors performances.

The rest of the present chapter is structured as follows: Section 2.2 reasons on cultural misalignments among software architects and the empirical software engineers. Section 2.3 presents the challenges in applying empirical methods to software architecture. Section 2.4 recalls the available empirical methods and characterizes them by discussing which challenges the experimenter is likely to face, in the context of software architecture. Section 2.5 describes some of our experiences in performing empirical studies on software architecture. Section 2.6 provides final remarks and sketches future works.

2.2 Empirical software engineering and software architecture: detecting cultural misalignments

As already mentioned, it is still very small the amount of empirical studies concerning software architecture. Beside the existence of several challenges that are not easy to face (see next section), there are gaps to fill out between the software architects and the empirical software engineers.

In fact, the software architecture community is still mostly composed of industrial practitioners, who discovered the importance of software architecture and drove much of software architecture advances.

Conversely, the empirical community is still prevalently composed of academic groups together with few industrial development & research labs. Both of them usually work on the average or long term. The former looks at empirical methods as a valid means to confirm, or invalidate, software engineering hypotheses, measurement models, and hopefully to provide evidence-based laws, and eventually theories. The latter looks at specific technical issues like mixing testing techniques for defect detection.

This situation led to three main kinds of gap or cultural misalignment between the software architect practitioners and the ESE scientists:

Ontology gap: the two communities evolve in different contexts and hence they adopt different terminologies or assign different semantics to common terms.

Doctrine gap: software architects tend not to rely on empirical methods, especially experiments. On the other side, empiricists seem to incline to emphasize on results from the inferential statistics while underestimating practical concerns (and understanding gained while being around the coffee-machine).

Knowledge gap: up to now, the empirical community has not been eager to advancing software architecture knowledge. On the other side, the software architecture community is not (yet) completely aware of how to effectively apply the empirical methods.

As a result, empirical studies that architects perform tend to be invalid because they frequently neglect significant aspects of the empirical methodology. For instance, Ferrari and Madhavji in [2007] focused the attention on replicating industrial settings (e.g. objects, professional reviews, etc.) while giving low importance on avoiding the threat of the randomized controlled trial [Wikipedia]. Additionally, software architects frequently consider out of their focuses the studies that ESE scientists perform because of the excessive simplification of the context or the theoretical nature of the research questions, and the related assumptions. For instance, many ESE academic studies adopt university people (students, and sometimes professionals who works in the university) as experiment subjects performing in the role of expert software architect e.g. "We choose to use a small board of experts in the field for the study, i.e., people from academia" [Svahnberg and Wohlin, 2005]; it is unclear whether it is reasonable and in what extent to consider university software engineers or students able to represent sufficiently the role of expert software architect.

2.3 Ten challenges in applying empirical software engineering to software architecture

Challenges to face when applying ESE to software architecture include Software architecture's Goodness, Evaluation technique description, Evaluation input selection, Evaluation scenario description, Result description, Evaluation vs. resulting system, Professional review, Complexity of the experiment objects, Fuzzy boundaries, Involving experienced experiment subjects.

Next Section 4 uses these challenges to constitute the kernel of a characterization schema for selecting the most suitable empirical method(s) to apply (i.e. the one with lowest challenges), based on the application contexts (e.g. available software applications, architects, reviewers).

In the rest of the present section, each sub-section describes a specific challenge in the application of ESE to software architecture.

2.3.1 Defining software architecture "goodness"

Defining the level of goodness of software architecture is a complicated matter. According to Bass et al. [2003], "analyzing an architecture without knowing the exact criteria for goodness is like beginning a trip without a destination in mind." On the same vein, Booch writes "one architectural style might be deemed better than another for that domain because it better resolves those forces. In that sense, there's a goodness of fit—not necessarily a perfect fit, but good enough." [Booch, 2006] The concept of "good enough" may vary according to some aspects that are difficult to describe, including:

Bounded rationality: the level of goodness heavily depends on the amount of knowledge that is available at evaluation time. Software architecture is an artifact that the software development lifecycle delivers very early; this implies that software architecture decisions are often made based on unstable and quite vague system requirements. Hence, software architecture goodness depends on the existent level of risk for missing knowledge, which is difficult to describe.

Other influencing decisions: Design decisions are made based on the characteristics of the relationships that they have with other decisions, which are outside of the architect reachability range, and are called "pericrises" in [Simon, 1996a]. Additionally, the few available tools for handling software architecture design decisions are still in their infancy [Capilla et al., 2006], [Ali Babar and Gorton, 2007].

Return on investment (ROI): Usually, for the development of any system, the optimal set of decisions is the one that maximizes the ROI. In such a view, for instance, an actual architecture is more valuable than a better potential one, which would be achievable by applying some modifications to the actual one: in fact, the potential architecture would require some additional risk and delay onto the project to deliver, which typically imply financial losses. Therefore, in practice, the ROI is an important driver to define the goodness of software architecture; vice versa, it is neglected in most empirical studies.

Social factors: Social issues such as business strategy, national culture, corporate policy, size of the development team, degree of

geographic distribution, etc., all crucially influence every kind of design decisions.

The difficulties in describing the factors (bounded rationality, other influencing decisions, ROI, and social factors) that influence the development of the software architecture, constitute a barrier when trying to measure and/or control at constant level related empirical variables (i.e., Tom Demarco's quip: "you cannot control what you cannot measure," [De Marco, 1986], based on Lord Kelvin's in 1882). In other words, if there is something that you were not able to describe/identify in advance then you cannot be sure that the results of the conducted empirical study depend on the defined treatments (i.e. the analyzed architectural method) and not on something else.

2.3.2 Describing a software architecture evaluation technique

In the absence of contrary evidence, we should assume that different evaluation techniques for software architecture might lead to different results. Based on the work by Ali Babar *et al.* [2004], we propose the following set of attributes to characterize a software architecture evaluation technique and related results:

- Adopted definition of software architecture [SEI, 2007].
- Objective (e.g., risk identification, change impact analysis, trade-off analysis),
- Evaluation technique adopted (e.g., questionnaire, checklist, scenarios, metrics, simulation, mathematical modeling, experience-based reasoning).
- Level of maturity of the adopted evaluation technique.
- Quality attributes taken into account (see Section 3.4 below: "Software architecture evaluation result description").
- Stage of the software process lifecycle (e.g. initial, mid-course, post-deployment).

- Type of architectural description (e.g. formal, informal, specific ADL).
- Type of evaluation (e.g. quantitative, qualitative), non-technical issues (e.g. organizational structure).
- Required effort.
- Size of the evaluation team.
- Kind of involved stakeholders.
- Number of scenarios, if any.

This set of attributes represents just a basic frame of reference to reach an agreement for a comprehensive description of a software architecture evaluation technique.

The difficulty in describing the software architecture evaluation technique is an obstacle for a valid empirical data analysis.

2.3.3 Selecting the software architecture evaluation input

Similarly to what we already did in the previous sub-section, in order to evaluate software architecture, we assume that different types of input may lead to different results. Following Obbink *et al.* [2002], we propose the following set, as the inputs that can influence the result of a software architecture evaluation:

- Objectives (e.g., certifying conformance to some standard, assessing the quality of the architecture, identifying opportunities for improvement, improving communication between stakeholders).
- Scope (i.e. what exactly will be reviewed).
- Architectural artifacts, (e.g. software architecture document, architectural prototype, design guidelines, etc.)
- Supporting evidence (e.g., feasibility studies, exploratory prototypes, minutes, notes, whitepapers, measurements).
- Architecturally significant requirements.

- Product strategy and planning.
- Standards and constraints.
- Quality assurance policies.
- Risk assessments.

As for Section 2.3.2, this set is only a starting point to achieve a consensus for a comprehensive description of a software architecture evaluation input.

The difficulty in describing this evaluation step is a further barrier for ESE to overcome in the field of software architecture.

2.3.4 Describing the software architecture evaluation scenarios

A scenario can be defined as a brief description of a single interaction of a stakeholder with a system [Kazman et al., 2000]. "There are also some attribute model-based methods and quantitative models for software architecture evaluation but, these methods are still being validated and are considered complementary techniques to scenario-based methods."[Babar et al., 2004] In fact, different scenarios lead to different evaluations, therefore researchers should focus on two main issues when doing empirical studies of scenarios: i) to apply all the treatments on each scenario, and ii) to describe each scenario in the report, otherwise the readers would not be able to understand to what extent the empirical results are applicable to their own context. While we have quasi-standard rules for describing our experimental apparatuses (e.g., type of subjects, their incentives [Host et al., 2005]), no standard is available for describing the scenarios used for evaluating an architecture. Lacking this point, researchers describe scenarios in an *ad hoc* manner, which eventually decreases the comprehensibility of the results from their empirical studies.

The difficulty in describing completely the evaluation scenarios is a barrier for a valid empirical data analysis.

2.3.5 Describing the software architecture evaluation results

According to Bass et al. [2003], software architecture quality can be defined in terms of ease of creation, outsourceability, buy-in, conformance, manufacturability, environment impact, suitability, interoperability, security, compliance, inerrability, configurability, compatibility, correctness, accuracy, availability, fault tolerance, recoverability, safety, understandability, learnability, operability, explicitness, responsiveness, customizability, clarity, helpfulness, attractiveness, time behavior, resource utilization, analyzability, correctability, expandability, stability, testability, scalability, serviceability, adaptability, co-existence, installability, upgradability, replaceability, reusability, and many other "ilities". The contribution of such a list is twofold: (i) to provide a basic frame of reference for describing the goodness of an architecture (by including, refining, or omitting the proposed attributes), (ii) to stress that most of the software architecture quality attributes can be objectively measured, so that we can describe software architecture "goodness" in a objective way.

2.3.6 Evaluating software architecture without analyzing also the resulting system

Large complex software systems are prone to be late to market, and they often exhibit quality problems and fewer functionalities than expected [Jones, 1994]; hence, it is extremely important to uncover any software problem or risk as early as possible. Reviewing the software architecture represents a valid means to check the conformance of the system and to reveal early any potentially missed objective [Maranzano et al., 2005] because: i) software architecture is initially developed during the early phases of the development process and 2)it constraints or facilitates the achievement of specific functional requirements, non-functional requirements, and business goals. However, some of the "ilities" regarding the quality of software architecture (see Section 2.3.5 above) can be analyzed only once the system has been developed. This happens because architectural decisions constraints the other decisions (i.e. detailed design, implementation decisions), which also impact on the system functionalities. The use of a specific architectural pattern (e.g.

layers) can promote specific "ilities" but it does not ensure the achievement of a specific non-functional requirement (e.g. a certain response time) [Harrison and Avgeriou, 2007]. One of the main reason is that decisions interact to each other [Kruchten, 2004] [Eguiluz and Barbacci, 2003]; "The problem is that all the different aspects interrelate (just like they do in hardware engineering). It would be nice if top level designers could ignore the details of module algorithm design. Likewise, it would be nice if programmers did not have to worry about top level design issues when designing the internal algorithms of a module. Unfortunately, the aspects of one design layer intrude into the others." [Reeves, 1992]

2.3.7 Affording the cost of professional reviews

Reviewing software architecture is not an easy job because it requires a lot of experience in the related domain. As a consequence, the architecture review is an expensive task; according to Bass *et al.* [2003] a professional architecture review costs around 50 staff-days. Of course, such a cost is a strong barrier for enacting empirical studies with professional architects.

2.3.8 Adopting a complex system as experiment object

One of the main intent of software architecture is to provide "intellectual control over a sophisticated system enormous complexity" [Kruchten et al., 2006b]. Hence software architecture is really useful only for large software systems whose complexity would not be manageable otherwise. It would be not representative of the state of the practice, the use of software architecture artifacts for small or simple systems, like the empirical objects that are frequently adopted in academic studies with students; such study would neglect phenomena characterizing complex systems. In other words, the results concerning the use of software architecture artifacts for toy systems does not scale up because the design of large complex system involves issues that are different in the type, and not just in the amount, from the ones involved in toy system. This constitutes a barrier to the construction of valid *artificial* empirical objects; note that this does imply not only a low

generality of the results, but also that something is wrong with the results.

2.3.9 Defining the boundaries of software architecture

There is no clear agreement on a definition of software architecture [Smolander, 2002] [SEI, 2007]. Software architecture encompasses the set of decisions that have an impact on the system behavior as a whole (and not just parts of it). Hence, an element is architecturally relevant based on the locality of its impacts rather than on where or when it was developed.

The difficulty in specifying the boundaries between software architecture and the rest of the design is a barrier to the selection of empirical objects to study.

2.3.10 Affording the cost of experienced subjects

In general, software architecture decision-making requires a high level of experience. Hence the use of empirical subjects with little experience (e.g. students) is not representative of the state of the practice. The same holds for some ESE studies. For instance, regarding studies on Pair programming, results from experiments adopting professionals [Arisholm et al., 2007] are different from the ones adopting students [Williams and Upchurch, 2001]. An empirical study on software architecture requires experienced (expensive) subjects; of course, such a cost is a strong barrier for enacting empirical studies with professional architects.

2.4 Characterizing empirical methods with respect to software architecture

An objective of ESE is to advance tools and methods for software development, using the evidence-based paradigm [Kitchenham et al., 2004]. The purpose of this section is to characterize a set of well-known empirical methods by exposing the challenges that their application to software architecture places. This characterization provides a strategy to

increase both the number and validity of empirical studies on software architecture by allowing the researchers to select the most suitable empirical methods to apply (i.e. the one with minor challenges), based on the application contexts (e.g. available software applications, architects, reviewers).

In general, an empirical process consists of the following five steps:

- Conjecture and goal definition, research question specification, and hypothesis formulation.
- Study design.
- Subjects acquisition, study conduction, and data gathering,
- Data analysis.
- Results discussion.
- Different paradigms are used in ESE, including surveys, case studies, and controlled experiments, and each paradigm has its own methods. An empirical method depends on, and can be classified by, several attributes, including:

Type of projects involved in the study: they can be real projects (i.e., developed to be deployed) or artificial projects (i.e., set up specifically for the purpose of the study); and they can be on-ongoing projects (still in the developing phase) or completed projects (i.e., the system has been already deployed and being used). Although this is not the only possible classification of empirical objects and methods (see for example [Zelkowitz and Wallace, 1998] and [Kitchenham, 1996]), the proposed classification fits the aim of the present work i.e. it allows researchers to select the empirical method that is mostly adequate for the application context.

Type of data obtained from the empirical study: this is another relevant classification, orthogonal to the Type of projects one, that considers the type of the data obtained: *quantitative* data or *qualitative* data. The former is measured using a scale (i.e., nominal, ordinal, interval, or ratio); the latter consists in text, sounds or images. Quantitative data is mainly used in experiments to enact statistical test

for *confirming* hypothesis. The analysis of qualitative data is particularly useful when the research area is quite immature (i.e. no concrete hypothesis can be formulated or the variables are hard to define/quantify); qualitative data is mainly used in order to *discover* trends, patterns, and generalizations. In order to allow a triangulation activity, which in turn is a valid mean to achieve high internal validity [Seaman, 1999], it is worth to adopt both quantitative and qualitative data in an empirical study.

In the remainder of this section, subsections are organized based on the first of the aforementioned classifications, i.e. the type of adopted projects. Each sub-subsection describes in turn a specific empirical method, which matches the parent subsection's type of project. The first paragraph of each sub-subsection briefs on the method's peculiarities; the literature already described these in-depth (see for instance [Zelkowitz and Wallace, 1998], [Kitchenham, 1996] [Wohlin et al., 2000]). Based on the challenges mentioned in Section 3 above, the (second and) last paragraph of each sub-section describes how likely (in terms of levels of challenges) the current method is in being applied to the software architecture field.

2.4.1 Real ongoing project

Data concerning the application of the investigated factor (e.g. Software engineering method), are collected while an instance of a factor ("treatment") is applied to a real ongoing project. In other words, while the project is developed for business, a parallel empirical data collection activity is performed, which affects the project development process as less as possible.

Because there is a complete control over the objective of an ongoing project, this family of empirical methods presents low challenges in terms of "software architecture goodness". In other words, it is well known which features are desired from the current software architecture, even though it maybe in the form of tacit knowledge. Because the reviewers are in-house and the reviews may be part of the regular project activities, the cost to perform a software architecture review is to consider in the average. With respect to the ordinary software process, in most of the cases, only the process instrumentation and empiric data collection are the extra activities that the empirical

study requires. Since the project is real, there should be no challenge at all both in having large software applications to utilize as objects of study, and using experienced architects. Because the type of the collected empirical data could not be, and usually is not, exactly what the empirical scientist wishes, the challenge in controlling the measures is generally in the average rather than low level. This is due to the fact that meeting all the requests of the empirical researchers would result into data collection mechanisms too much intrusive in the project and/or expensive. Finally, since the project is not yet finished, there is a high level of challenge in evaluating the software architecture without analyzing the resulting system also.

2.4.1.1 Project monitoring

Project monitoring consists in achieving data while the factor (i.e. Software engineering method) is being applied to a certain project. Project monitoring is considered a passive method because data were designed for collection based on goals that are out of the focus of the empirical study but are relevant for developing the project (i.e., project management issues). Because data are actually collected, they can be used also for validating new architectural hypotheses empirically, if any related with. This method is called *field study* when two or more projects are monitored. Recall that such a project is executed for developing a real software system rather than empirical data.

Both project monitoring and field studies minimize intrusiveness and the effort spent to collect empirical data.

2.4.1.2 Case Study

The Case study method [Kitchenham et al., 1995] [Flyvbjerg, 2006] [Yin, 2002] is very similar to the Project monitoring method despite the fact that in the former the type of data collected is planned and relates to a predefined objectives or hypotheses (e.g., the hypothesis that a specific period of synchronization among the different software architecture views is better than another period).

With respect to project monitoring, a case study usually requires a little bit more of effort, provides more control on the measurements, and implies more intrusiveness on the development project.

2.4.2 Completed real project

This family of methods, also known as post-mortem analysis, is related to completed project(s), from which empirical data can be extracted after completion. Examples of data sources may be a company's database. Because the empirical data are already available, the main advantage of these methods is that their adoption requires minor effort (under the assumption that searching for data is less expensive than producing data from scratch). Moreover, since the empirical objects consist in already finished projects, there is no challenge at all in evaluating the software architecture without analyzing the resulting system also. The main disadvantage of these methods is that the available data is often incomplete and/or the database is seeded massively with unstructured data; this provides the researchers an incomplete and/or messed overview of the observed phenomenon. Because the collected data constitute the only source of knowledge for the study, there is a high level of challenges of the types: Evaluation technique description, Evaluation scenario description, Result description, and Fuzzy boundaries. Additionally, it grows the complexity of control and measurement.

2.4.2.1 Human-enacted analysis: Literature review, Legacy data, and Lessons learned

Three main methods are available for humans enacting analysis of data related to completed real projects: Literature review, Legacy data, and Lessons learned.

Literature review: researchers analyze the content of papers or other available documents to confirm hypotheses or find lack in the current state of the a art for the given field, e.g. Software architecture design methods [Falessi et al., 2007]. Techniques like Meta-ethnography [Doyle, 2003] and Meta-study [Hunter and Schmidt, 2004] can be used to synthesize data coming from different sources. The main drawback for this empirical method is that authors frequently consider their publications as means for advertising their methods, which are not compared with other ones [Wikipedia]. In fact, these kinds of papers tend to focus on advantages of the methods that they propose while neglecting their disadvantages. As proposed by a mini-tutorial on writing good software engineering research papers [Shaw, 2003], the

quality of the proposed methods should be validated; however, in our view, this validation should be complemented by a systematic description of the disadvantages in using those methods and the circumstances where competitive methods are expected to perform better than the proposed one(s). Hence, literature review may produce invalid results because the used empirical data (i.e. actual papers) might be biased.

Legacy data method concerns the analysis of quantitative data, which were collected in the past during the development of currently completed project(s). It is a kind of post mortem analysis.

Lesson learned method concerns the analysis of artifacts, which relates to experiences matured and knowledge gained while developing projects presently completed. Based on that analysis, researchers can reveal pros and cons associated with the adoption of a specific entity, e.g. a software engineering method, in the given context, i.e. characteristics of the people involved, process enacted, and product developed. This method differs from the Legacy data one since the latter utilizes just quantitative data while the former works on lessons, which are frequently qualitative. [Birk and Tauz, 1998]

2.4.2.2 Tool-enacted analysis

Tools are utilized to extract empirical data. For instance, in order to achieve metrics regarding the logical view of software architecture (e.g. module coupling, and cohesion), a tool can be used to analyze the type and amount of references into the software artifacts (e.g. the application code). There are different type of analyzers for software artifacts, including Static analyzer, Dynamic analyzers**,** and Simulators.

Because these tools require formal input, it is in the average the level of challenges to face for describing the Evaluation technique, Evaluation input, and Evaluation scenario.

2.4.3 Ad-hoc project

It characterizes this family of methods the fact that projects are done just for the purpose of the empirical study, usually a controlled experiment, and not for business objectives.

This method is mostly used to confirming/disconfirming hypotheses and theories, exploring relationships among data points that describe one variable or across multiple variables, evaluating accuracy of models, or validating measures. Case studies or controlled experiments are conducted, where the latter provide highest level of formality, rigor, and control on measure. Specific guidelines are available for their conduction [Wohlin et al., 2000] [Juristo and Moreno, 2006] and documentation [Jedlitschka et al., 2007].

The difficulty of reproducing industrial practices into an artificial setting provokes: I) relevant challenges of the types Goodness of the architecture, Evaluation technique, Evaluation input, and Fuzzy boundaries; II) medium challenges of the types Cost of reviews, Size of the system, and Subjects' experience. Finally, though the used projects are completed, they might be not representative of the industrial practice; hence, there is a medium level of challenge of the type Evaluation without analyzing the resulting system.

2.4.3.1 Artificial environment

In this method, whatever the objects used and whoever the subjects involved, the empirical study is conducted in an environment (namely, the Software lab), which the researcher adapts to the goals of the investigation in the aim of imitating, as much as possible, the target (industrial) environment. The study usually consists in a pilot experiment or controlled experiment. The artificial environment offers an optimal control over the project development. Because reproducing in lab an industrial setting for architecture design may be extremely expensive, the main disadvantage of this method is that the experiment instrumentation (computers and tools) might be unrepresentative of the industrial ones. An instance of quality indicators for artificial environments is in what extent the utilized experimental infrastructure (e.g., communication mechanisms, tools, and hardware) is up to date.

2.4.3.2 Naïf subjects

Due to cost constraints, it might be prohibitive to use professionals as subjects of empirical studies. Consequently, researchers frequently involve students to perform in the role of architects. Examples of quality metrics for these subjects are the minimum level allowed for their experience and their number. Anyway, adopting students as

experimental subjects (architects) may threaten the results, so making these not applicable to industrial realities; vice versa, the expenses for using professionals may be out of the budget.

2.4.3.3 Artificial objects

The complexity of the empirical projects is reduced up to become feasible in the given time and cost constraints. As a consequence, the empirical study neglects effects that complex applications produce and small applications do not produce (a.k.a. toy applications). This, in turn, would produce invalid results. Instances of quality metrics for artificial objects are the amounts of Function-points, Lines of code, and Use-cases of the experiment project.

2.4.4 Challenges in synthesis

Table 2-1 summarizes the contents of this Section 2.4 by describing the levels of challenge in applying an empirical method to software architecture. Because the typical controlled experiment (look at Table 2-1 for Ad-hoc project) presents the highest level of challenges, it explains why experimental methods have been the least employed in the software architecture field up to now. Conversely, empirical studies adopting a real project (see both the leftmost, and central parts in Table 2-1) seem to be the most feasible and effective type of empirical method for advancing the software architecture field.

Table 2-1. Levels of challenges in applying a specific empirical method to software architecture.

		EMPIRICAL METHODS						
		Ongoing real project		Completed real		Ad-hoc project (i.e. synthetic)		
		Project monitoring	Case Study	Human based analysis	Tool based analysis	Synthetic environment	Synthetic object	Synthetic subject
CHALLENGES	Goodness	Low	Low	Medium	Medium	High	High	High
	Evaluation technique	Medium	Medium	High	Medium	High	High	High
	Evaluation input	Medium	Medium	High	Medium	High	High	High
	Evaluation scenario	Medium	Medium	High	Low	Medium	Medium	Medium
	Evaluation result description	Medium	High	High	High	Low	Low	Low
	Final system availability	High	High	Low	Low	Medium	Medium	Medium
	Cost of review	Medium	Medium	Low	Low	High	High	High
	System complexity	Low	Low	Low	Low	Medium	High	Medium
	Fuzzy boundaries	Medium	Medium	High	High	High	High	High
	Subjects experience	Low	Low	Low	Low	Medium	Medium	High

2.5 Our experiences

This section describes some experiences we gained while handling the challenges mentioned previously (see Sections 2.3). The Section is organized in two subsections, each relating one experiment that we conducted with Master students in the Department of Informatics, Systems and Production engineering (DISP) at the University of Rome "Tor Vergata". We first describe a still unpublished experiment and focuses on the relevant effects that those challenges caused. In order to

explain what to do for preventing these effects, we reflect on another experiment where the focus is on the decisions that we made to address as much as possible those challenges.

2.5.1 Impact of the Boundary-Control-Entity architectural pattern

The Boundary-Control-Entity (BCE) is an architectural pattern that RUP provides [Kruchten, 2003], and practitioners largely use. It can be seen as a variant of the Model View Controller pattern [Gamma et al., 1995]. Based on BCE, a component, the Controller, is used for each Use case both to encapsulate the use case logic and decouple 1) the business logic and data access from data presentation, and 2) interactions with users and other actors. This structure should facilitate maintenance operations since changes to user interface should affect neither data handling nor program logic. Moreover, once defined the wanted behaviors, the entity classes should be re-implemented and reorganized without affecting user interfaces. Finally, changes in the control logics should not affect key domain abstractions.

The expectation is that the adoption of BCE, compared with letting the programmers free to choose the software structure that they prefer ("ad hoc"), should require higher up-front investments, (i.e. spending more time for the modeling phase) which should pay off by giving significant returns when such software is maintained.

In order to validate such an expectation, we performed an experiment. Twenty-one Master students in the course of Experimental Software Engineering in the DISP at the University of Rome "Tor Vergata" performed first in the role of software programmers (they received some analysis and design documents, and database design for the application), and subsequently in the roles of designers and programmers of specified enhancement-maintenance interventions.

The experiment consisted in enacting a number of iterations. In the remaining part of this chapter, we take into consideration results form the first and second iterations. During the first iteration, subjects developed individually and from scratch a web-based application for enabling family members and small groups to manage their media, like CD, DVD, books, etc. During the second iteration, subjects individually

applied an extensive maintenance intervention to their own product. In the beginning, students had been split in two groups; the ones applying the BCE pattern, the other one structuring the code as preferred. Hence the factor of the study is the Application System Structure and the treatments are BCE and Ad-hoc.

We had two main expectancies regarding the mandatory use of the pattern BCE: 1) the effort spent during the first iteration should be similar for both the treatments but give some advantage to Ad-hoc, 2) it should give a significant advantage to BCE in the average, the effort spent in enacting the maintenance intervention.

In order to measure the effort spent, we used the development time and the functional coverage of each application; eventually, we used the percentages of these coverage to compute the virtual completion time for each subject. While we collected data for all the individual phases - i.e., Requirements comprehension, Modeling (Analysis and Design), Implementation, Testing, and Delivery - in the remaining we mainly reason on time for completing the iteration as a whole.

Figure 2-1 summarizes the results from the study. Let us observe that expectancies are disconfirmed evidently. As a matter of fact, results show that, in the average, it requires a huge additional time the adoption of BCE rather than Ad-hoc as a structuring concept for developing from scratch (+ 50%), and maintaining (+ 30%) the given application. Notice that a fine-grained analysis, which the rest of the present work neglects, revealed that using BCE required more time than Ad-hoc in all the development phases.

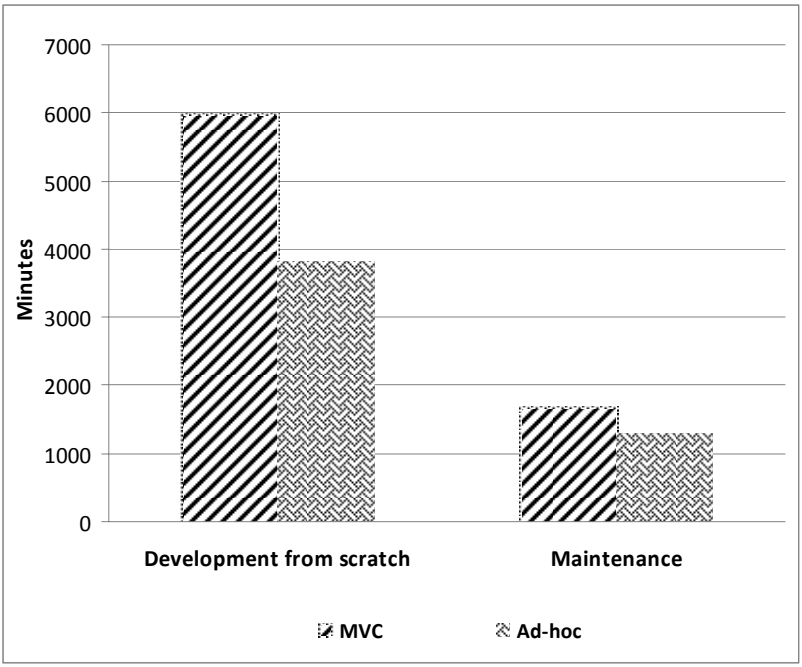

Figure 2-1: Time spent in the average, for development from the scratch and maintenance, by using BCE vs. Ad-hoc structures.

It is due to the impact of the challenges shown in Section 2.3 above, the difference between our expectations and the observed results, in our conjecture. In particular:

System complexity: In order to emulate real software, we adopted a medium size application as the empirical object to develop, which resulted in 15 KSLOC. While this is not really a minor size, it is not the only factor to take in consideration: in fact, because of the low complexity of the application as a whole, we classify it as toy software. Moreover, subjects were aware of the fact that no person would be using that application. Furthermore, they understood that we would not be able to verify the architecture model realized, due to the short time available between the end of the experiment works and the end of the course. Hence, possibly subjects found the exams pressure a motivation morally sufficient to implement just to pass our functional checking rather than to provide a business valuable application. Maybe the

adoption of an application in the same size but with business goals and real final users would have given different results.

Subject's experience: Participating subjects had already gained knowledge of BCE by attending previously the academic courses of Object-oriented Analysis and Design in the DISP. However, in our view, their experience was not representative of practitioners, and should expect different results whether using professionals as experiment subjects.

Cost of review, and **Goodness**: In our view, the development time was the right metric to estimate the "goodness" of a specific structure with respect to facilitating maintenance interventions. However, as already mentioned, we were not able yet to check the quality of the artifacts; hence, we do not know in what extent the BCE group really applied the BCE pattern. On the other side some Ad hoc subjects would have been applying BCE to some extent. Hence, there is the risk that data should move from the BCE to the Ad hoc group, and vice versa. Anyway, results show that, in the given context, in the average, it applied a "better" structuring concept the people assigned to adopt an Ad-hoc structure rather than people assigned to apply BCE (whatever the mean of that "better" might be).

The achieved result confirms the theory that architectural decisions should not be imposed *a priori,* especially when people show minor confidence with it. Hence, lessons learned are: i) design decisions (in our case, a specific structure for design and code) should be made on the fly, and based on the experience of the available developers; and, ii) decisions should aim to achieve specific business objectives (there was none in our case) by taking into account the current context peculiarities (in our case, a toy application). In conclusion, the case study presented constitutes an illustration of how difficult it is to face the aforementioned challenges, in order to validate hypotheses regarding the software architecture field, by using experiments (with naïf/junior software people, toy objects, and non-representative environments) as empirical method.

2.5.2 Impact of the Design Decision Rationale Documentation

This sub-section uses results from a further experiment (see Section 5 and [Falessi et al., 2006b]) as one example of how challenges can be effectively faced when validating hypothesis regarding the software architecture field by using experiments as empirical method. Our main expectancy was that in case of requirement changes, the correctness of the decisions made improves when the design decision rationale documentation is available, both for individual decision-making and team decision-making.

In this experiment we faced challenges effectively. The key idea was to use single decisions as the empirical objects; this is not contradictory with the current trend to consider software architecture as a set of design decisions [Kruchten, 2003] [Jansen and Bosch, 2005]. Hence our preference was for analyzing performances of software engineering methods (in our case Documenting vs. Non-documenting the design decision rationale) by using one decision at a time rather than the whole set of decisions at one time. In fact, breaking down the decision process was a cleverness that helped us to face all the main challenges. The challenges we faced by using specific cleverness include:

Fuzzy boundaries: The use of single decisions made explicit the study assumptions about the architectural boundaries (i.e., which aspects of the architecture we wanted to analyze): 1) Focusing on results from single decisions, 2) Taking into account areas like: Hardware, Communication, Software Architecture (in our case code structure) & Services Discovery, Inference, and Data Storage.

System complexity: The project was only described, not really implemented; this allowed the usage of a system sufficiently detailed and complex as experimental object. In fact, it was hard (not trivial, like it would have been with a toy application) to re-make the decisions that the subjects eventually adopted, because several opposite and inter-related objectives characterized the complex of those decisions, as it is in the reality [Eguiluz and Barbacci, 2003].

Cost of review, Evaluation technique, Evaluation input, Evaluation scenario, and **Evaluation results:** The use of single decisions facilitated the process of facing challenges concerning the evaluation procedures for cost, technique, input, scenario, and results.

Subject's experience: Also in this second experiment, we had not the chance to employ professionals, and we had to use again Master students as experimental subjects. However, we noticed that, in the average, each of those students was skilled in a specific area of IT, due to personal interests, academic vitae, and/or some industrial experiences. Hence, our consequent decision was to design the experiment for maximizing the usage of the students' expertise: as already mentioned, we assigned subjects to perform in a specific role and make decisions belonging to such a role. This seems to allow us to consider these subjects quite similar to junior professionals for level of expertise and knowledge in their preferred field.

2.6 Conclusion

In order to promote and facilitate advances in software architecture by means of empirical studies, the chapter proposed a characterization of the available empirical methods by exposing their specific levels of challenges when applied to software architecture. Such a proposed characterization should help to increase both the number and validity of software architecture empirical studies by allowing researchers to select the most suitable empirical method(s) to apply (i.e. the one with lesser challenges to face), based on the application contexts (e.g. available software applications, architects, reviewers). However, in our view, in order to provide high levels of conclusion and internal validity, empirical methods for software architecture should be oriented to take advantage of both quantitative and qualitative data.

Additionally, based on the experience gained from these experiments, such challenges might: 1) highly influence the results of the empirical studies and 2) be faced by empiricists' cleverness.

Mulla Nasruddin was a 13th century Sufi visionary who wrote the parable known as "searching the keys under the lamppost." He describes a man who lost his keys in a dark place and who then tried to recover them not were he lost them but far away, under a lamp, because this was

the only place with enough light for searching. In this context the dark areas are the aforementioned challenges. In conclusion, there are a lot of dark areas and they are the main reason of the current mismatch between ESE and software architecture. However, it is by taking these dark areas really into consideration that we can:

1) Search in the most enlightened part as possible: i.e. correctly adopt best ESE method according to the application context by taking advantage of the described strategy. For instance, we already have standard metrics to measure software architecture quality, thus we should avoid the use of other subjective dependent variables.

2) Consider studies that searched in the dark as valid, instead of invalid, if there were no light available in the surroundings (i.e. there would be no better way to search), and especially

3) Install bright lamps in key dark areas: i.e. resolve some of the aforementioned challenges.

3. A Characterization of Software Architecture Design Methods

Several Software Architecture Design Methods (SADM) have been published, reviewed, and compared. But these surveys and comparisons are mostly centered on intrinsic elements of the design method, and they do not compare them from the perspective of the actual needs of software architects. We would like to analyze the completeness of SADM from an architect's point of view. To do so, we define nine categories of software architects' needs, propose an ordinal scale for evaluating the degree to which a given SADM meets the needs, and then apply this to a small set of SADMs. The contribution of the chapter is twofold: (i) to provide a different and useful frame of reference for architects to select SADM, and (ii) to suggest SADM areas of improvements. We found two answers to our question: "do architectural design methods meet the needs of the architect?" Yes, all architect's needs are met by one or another SADM, and No, no architectural design method meets simultaneously all the needs of an architect. This approach may lead to improvements of existing SADMs.

3.1 Introduction

Software architecture (SA) plays an essential role for achieving intellectual control over a sophisticated system's enormous complexity [Kruchten et al., 2006b]. Software, architects can select among several methods for deriving a software architecture from software requirements; such methods are called "Software Architecture Design Methods" (SADM). There are several surveys describing SADMs, but all of them were done in a comparative way, highlighting similarities and differences, but without dealing with real architects needs; in fact, each SADM (which should be considered as a solution for an architect) was described based on the properties of some other SADMs (i.e., the solution space).

In this chapter we will describe SADMs by considering which architects' needs they meet (i.e., the problem space). We first identify and classify the software architects needs in nine coarse categories by

searching literature and consulting industrial architects. Then we propose an ordinal scale for each of those needs, and eventually evaluate the completeness of any SADM with respect to those categories of needs. The set of categories of needs that we use may be incomplete (in number) but is relevant to architects.

The contribution of the chapter is twofold: (i) to provide a different and useful frame of reference for architects to select SADM, and (ii) to suggest SADM areas of improvements. In other words, we try to help: i) the architects in answering the question "Which SADM can help me in addressing a set of my needs?" (rather than "Which SADM is the best for addressing a certain need?"), and ii) the SADM developers in answering the question "Which set of needs are not yet covered by an SADM?" rather than "How to cover needs not yet covered by any SADM?".

The remainder of this chapter is organized as follows. Section 3.2 introduces to fundamental concepts. Then Section 3.3 briefly sketches on the rationale of our approach, and Section 3.4 describes attributes and values of evaluation model that we propose. Sections 3.5 describes the main limitations of our study. Section 3.6 presents the result, and comments of that evaluation model. Section 3.7 describes possible future works and concludes the chapter.

3.2 Context and background

3.2.1 Survey of software architecture design methods

3.2.1.1 Approaches and techniques

From the available approaches to comparing design methods, we adopted the quasi-formal approach suggested in [Sol, 1991]. Based on Song & Osterweil [1992], such a comparison can be enacted by five different techniques:

- Describe an idealized design method and evaluate other methods against it.

- Distill a set of important features inductively from several methods and compare those methods against it.
- Formulate a priori hypotheses about a method's requirements and derive a framework from the empirical evidence in several methods.
- Define a meta-language as a communication vehicle and a frame of reference against which you can describe many methods
- Use a specific contingency approach and try to relate each method's feature to specific problem.

As briefly described in the remaining of this sub-section, the literature is limited to present SADM comparisons of using technique one, two, and four, in our best knowledge. The approach taken in the present chapter uses the third of the comparison techniques above, and extends it by taking into account industrial architects' needs.

3.2.1.2 Surveys

Hofmeister et al. [2007] compared five industrial software architecture design methods and extracted a general software architecture design approach from commonalities of those methods. Such a design approach inspired the one that we already presented (see Figure 1-1). The work by Hofmeister et al. also provided a solid foundation to this chapter.

Sharble & Cohen [1993] compared class diagrams from two different OO development methods by using complexity metrics. Wieringa [1998] compared artifacts from structured and OO specification methods. Hong et al. [1993a] compared artifacts from six OO analysis and design methods. Kim & Lerch [1992] quantified the cognitive activities of designers when using an OO design method and a functional decomposition method, respectively. Song & Osterweil [1994] proposed and used a process-modeling technique to model activities and artifacts from some methodologies (Jackson System Development, Booch's Object Oriented Design, Structured Design, and Rational Design Methodology).

Hong et al. [1993b] first analyzed the available methods by considering their activities, breaking down these activities by using a

finer granularity, and selecting the best sub-activities; then they created a "super-methodology" by combining those sub-activities; eventually, they compared their super-methodology with each of the methods they had been decomposing. Fichman & Kemerer [1992], similarly to Hong et al., compared design methods by taking the superset of activities supported by the methods they were analyzing. Eventually, they used eleven analysis activities and ten design activities. Dobrica & Niemela [2002] compared eight methods for software architecture evaluation.

3.2.2 Software architecture design methods

There is not enough room here for comprehensive description of SADMs. Of course, there are already in literature chapters and surveys that concern SADM; in order to help the reader to approach our SADM context, Table 3-1 briefly sketches some SADMs key aspects, such as references and specificities.

Table 3-1: Key aspects of SADM (as claimed by the authors of the method) and our comments

SADM	Peculiarities & *Our comments*
G&S	Goal & Scenario consists in combining goals and scenario mechanisms. Include design rationale documentation. *Still promising.*
Tropos	Tropos is suitable for agent-based, cooperative, and distributed systems; it provides support for organizzation issues; it does not provide guidelines but ad hoc process.
Rule Based	Rule Based adopts a unified language for describing both SA and requirements. It is rule-based. *Still promising.*
RUP	The Rational Unified Process well integrates SADM in the whole sw. development process, has high level of standadirzation, is driven by risk mitigation. *It may be difficult top use RUP SADM in other sw development process.*
Siemens 4V	Based on the Global Analysis phase, which aims to analize and drive iterations; the architect identifies and analyzes the factors, explores the key architectural issues or challenges, then develops design strategies for solving these issues. *Developed and used industry-wide.*
BAPO	Business Architecture Process and Organization supports the development of product families, is flexible, adaptive, and goal-oriented, provides support for short innovation cycles, is suitable for embedded systems
ASC	Architectural Separation of Concerns is based on goals, architectural decisions, architecture descriptions, verification of architectural decisions, consistency-check of architecture and implementation. *Developed and used industry-wide .*
CBSP	Component-Bus-System-Properties provides a a layer between requirements and SA. *High level of formal documentation.*
ADD	Attribute Driven Design is the only non-functional-requirements-driven method presently available. *Provides several well-assembled SADM activities. Not yet standardized.*

3.3 Rationale of our approach

In order to describe and analyze an SADM for completeness with respect to software architects needs, we started by analyzing the literature and consulting professional software architects. In fact, this step was aimed to understand the problem space of architecting software. Subsequently, we proceeded to synthesize that problem space in nine categories of needs (called here: coarse-grained needs). Such categories of needs constitute the attributes of an SADM evaluation model. This might neglect some software architect needs, however it certainly takes in consideration real needs.

The specificity of our SADM evaluation model is its total independence from the SADM state of the art. As a matter of fact, our evaluation model originates from the architects' needs rather than from SADM comparative analyses; this is what makes the present description new and original.

Our description of SADMs is based on handling them as available solutions for a software architect who is faced with the problem of designing a SA. Intuitively, it is understandable because it is direct, a solution description that is based on the presentation of the problems it is able to solve. Moreover our description of SADMs is a useful means to strongly separate the problem space (i.e., evaluation model) from the solution space (i.e., evaluation model results). In other words, the result of our model describes the current SADM state of the art.

Let us highlight that the aim of our model is not to express qualitative evaluations about actual SADMs but to characterize them from the point of view of a software architect.

Nine categories of needs compose our evaluation model, each category matching a precise type of software architect need. The categories of needs are orthogonal one to each other.

In order to enact a measurement model and improve the understandability of the measurements resulting from the application of the evaluation model to SADMs, we design nine specific category of need, and measure the fulfillment of each need category by a four points ordinal scale. In particular, whatever the current category of architects

need might be, an SADM is able to fulfill that need at one of the following levels:

- **Null** (N): the evaluated method does not include a mandatory element.
- **Minimum** (Mn): specifies a minimum quantity of an element that an SADM is requested to fulfill (else the evaluated method is not considered as an SADM) or, in case, an optional element. In the former, the Mn specification is given positively ("the method does include …"); in the latter, it might be given negatively ("the method does not include …"), i.e. Mn≡Null and the method is still a candidate SADT (for the current category of need).
- **Maximum** (Mx): specifies the best fulfillment that a SADM could reach (for the current need category).
- **Intermediate** (I): represents fulfillments other than Mn, Mx, and N.

N, Mn, I, and Mx are values of an ordinal scale; usually, N<Mn<I<Mx; sometimes, N≡Mn<I<Mx.

Note how the fulfillment levels provided by the SADM(s) for a need category can be seen as a "specific need" from the architects' prospective.

In conclusion, in order to evaluate an SADM we provide and utilize a structure of direct measurement models, whose components are the category of needs, rather than a synthesis of that structure in an indirect model; a four points ordinal scale (N, Mn, I, Mx) measures directly the fulfillment of every category of need. Limits and threats to validity are described in Section 3.6.1 for such a structure of measures.

3.4 Architects seeds

In this section, for each category of needs, we describe:

- Key characteristics and why it is relevant for software architects. To these aims, we reference the literature and synthesize on what software architects wrote us.
- Specific needs or levels, from Null up to Maximum.

3.4.1 Abstraction and refinement

I. In SADM, we can look at refinement and abstraction as kinds of complementary concepts: the former adds details (top-down direction), the latter removes details (bottom-up direction) from software architecture model(s). The Object Management Group has been promoting the usage of abstraction and refinement in SA design: the Model Driven Architecture emphasizes on such concepts [OMG, 2003].

II. Levels:

- N≡Mn:: SADM does not include guidelines either for refinement or abstraction activity.
- I:: SADM includes guidelines only for refinement or only for abstraction activity.
- Mx:: SADM includes guidelines for both refinement and abstraction activity.

3.4.2 Empirical validation

I. Software engineering is human-based; consequently, empirical investigations can strongly help in evaluating its products, processes, methods and tools. The relevance of empirical software engineering is demonstrated by the presence of specific international conferences [International Symposium on Empirical Software Engineering and Measurement 2007], journals [Basili and Briand], books [Wohlin et al., 2000], and research groups [http://www.iese.fraunhofer.de/fhg/iese/index.jsp].

II. Levels:

- **N**:: Method still in the stage of not yet verified proposal.
- **Mn**:: SADM proposed and verified on paper only.

- **I**:: SADM already used by professionals or experimentally validated in lab.
- **Mx**:: SADM both used by professionals and validated experimentally, or extensively used by professionals.

3.4.3 Risk management

I. The perspective taken in this work is that risk is inherent to any software development activity [America et al., 2003]. The inevitability of risks demands for guidelines to recognize and manage risks.

II. Levels:

- **N≡Mn**:: SADM does not take into account risk concepts.
- **I**:: SADM provides guidelines to handle different categories or levels of risk.
- **Mx**:: SADM provides guidelines to handle different categories and levels of risk.

3.4.4 Interaction management

I. Techniques to address non-functional requirements interact to each others [Eguiluz and Barbacci, 2003]; therefore, the effects of the adoption of a technique depend on other techniques: the ones that are already in use or being employed [Kruchten, 2004]. In such a context, in our view, techniques include architectural styles, patterns, and mechanisms.

II. Levels concerning Interaction Management category of need are:

- **N≡Mn**:: SADM does not provide guideline of any category for handling interactions among techniques and managing related issues.
- **I**:: SADM provides generic guidelines for handling interactions among techniques and managing related issues.
- **Mx**:: SADM provides specific guidelines for handling interactions among techniques and managing related issues.

3.4.5 Tool support

I. Complex decisions demand for tool support. The multiplicity of the relation between software architecture and requirements is many to many, i.e. an architectural decision can affect several requirements, and vice versa; the human intelligence can be in trouble while architecting, when dealing with a huge number of requirements (i.e. large software system) [Grüenbacher et al., 2003]. Nowadays, the importance of the automated support in software engineering is confirmed by the presence of specific international conferences [http://www.di.univaq.it/ase2008/], journals, books, and research groups [Ruhe].

II. Levels:

- **N≡Mn**:: no tool is available to support SADM.
- **I**:: SADM is supported only by passive tools, i.e. the tools available provide only the "visualization" feature (e.g. graphs, matrices, etc.)
- **Mx**:: SADM is supported by active tool, i.e. the tools available include passive tools and additionally provide features to realize inference (e.g. simulation, worst/wrong solution avoidance, consistency checking).

3.4.6 Concerns

I. In general, views help architects to keep separated the different concerns of interests. The software community as a whole seems to agree on that it provides high benefits using multiple coordinated views to describe an SA. Several standards already exist, each proposing a different set of basic views [Clements et al., 2002]; a popular one (4+1 views) was proposed by Kruchten [1995a].

II. In order to allow an evaluation, let us simplify the domain of the SA "concerns" by assuming the spectrum of such concerns as: "Functional", "Behavioral" and "External" (or "Physical") [1995a]. Related levels are:

- **N**:: SADM deals with no one of the concerns above.
- **Mn**:: SADM deals with one concern.

- **I**:: SADM deals with two concerns.
- **Mx**:: SADM deals with all (three) the concerns above.

3.4.7 Knowledge base

I. This category consists in the architect's need of being supported in her or his decision making by some form of knowledge that the persistent memory of a tool comes to contain, once it is adapted to the given method.

II. Levels:

- **N≡Mn**:: SADM related tool does not use any experience base for assisting architects in decision making.
- **I**:: SADM related tool use an experience base for assisting architects in decision making. Such an experience base adopts a static memory: it cannot be changed (i.e. improved) during its usage.
- **Mx**:: SADM related tool use an experience base for assisting architects in decision making. Such an experience base adopts a dynamic memory: it can be changed (i.e. improved) during its usage.

3.4.8 Requirements change management

I. The ability in reacting to requirement changes is crucial to achieve project success. Such a feature is mostly used in the beginning of a software development process, when requirements are still incomplete and/or not defined well (see Section 1.3.2).

II. Levels:

- **N≡Mn**:: Views and requirements are not traced.
- **I**:: Guidelines are provided to trace requirements from one view.
- **Mx**:: Guidelines are provided to trace requirements from many views.

3.4.9 Number of supported activities

I. The main classes of activities in a SADM are: “Requirement analysis”, “Decision making”, and “Architectural evaluation”; these provide equally important contributes for arriving to a SA. A method dealing with one of them has is a method for just that activity, not a SADM; in our view, only methods which provide guidance for enacting two or all those activities are SADM.

II. Levels:

- **N**:: the SADM provides guidelines for enacting no more than one SADM activity.
- **Mn**:: SADM provides guidelines to enact two SADM activities.
- **Mx**:: SADM provides guidelines to all the SADM activities.

3.5 Architects needs and methods not taken into consideration

Despite the management of SA evolution is one of the main challenging aspects in practice [Jansen and Bosch, 2004], we remand its analysis to a future and more focused work.

We did not consider Problem Frame [Choppy and Reggio, 2004] and Softgoal [Chung et al., 1999] as SADMs because such methods are mainly included in the CBSP and G&S, respectively.

Despite empirically validations [Falessi et al., 2006b] provided evidence that architectural design rationale documentation is useful [Tyree and Akerman, 2005], this is not yet used largely in industry [Falessi et al., 2006a]. In fact, it seems to us that G&S is the only SADM to include its own guidelines for documenting explicitly design decision rationale. Anyway, we postpone to future works the investigation of what SADM includes what design documentation rationale.

3.6 Results and comments

3.6.1 Introduction, limits and threats to validity

We did not take into account quality issues in SADM: in our view, such a topic should be analyzed empirically but this presents several challenges as described in Section X. This point is postponed to further ongoing research.

In case where an SADM should not meet a specific architect need, our conjecture is to utilize that SADM in combination with an additional method, which is aimed to meet that need. However, in order to support successfully the combination of an SADM with additional foreign methods, specific guidelines and empirical evidence should be provided.

Concerning all the SADM that this chapter is considering, we contacted their authors and encouraged them to rate their own methods by using our evaluation model. Eight of them kindly complied. Because these evaluations are consistent with our own, we will be using them in the remaining of this chapter; for the rest of the SADMs, we will be using our evaluations, based on what we know about, as reported by literature (see column References in Table 3-1).

Threats to validity of those evaluations mainly come from the models attributes that depend on the subjectivity of the evaluator, including:

- Using SADM out-of-date documentation.
- Misunderstanding the documentation.
- Evaluating the features that the documentation neglected (i.e. features that the SADM does not provide).
- Overestimating, when the SADM authors also performed in the role of evaluators.
- Misunderstanding the proposed category of needs and the related specific needs, when the SADM authors also performed in the role of evaluators

In order to gain on validity, we tried to mitigate the impact of subjectivity on evaluations by enacting several precautions, including:

- Collecting advices from several architects, who work in different contexts, and using those advises as drivers for selecting the categories of needs to put in our evaluation model.
- Avoiding ambiguity by using both a standard terminology [Ieee, 2000], and a formal and standard ordinal scale for measuring the model attributes.
- Employing one more evaluator, when the first evaluator was the author of the method, and verifying the consistency of the results.
- Concerning the last point above, let us note explicitly that it also gives our measurement model a first accredit. In fact, almost the totality of measures resulted consistent, which different subjects — the SADM authors and we — independently provided for common objects.

3.6.2 Reference for helping an architect to select the SADM mostly appropriate for the needs

Table 2 provides an overview on the SADM levels of fulfillment for architects needs. We are convinced that Table 3-2 constitutes a valuable needs-driven support for architects who are looking for a SADM. All the results described in the present section derive from Table 3-2.

Figure 3-1 shows the degree of completeness of SADMs with respect to the categories of needs. Figure 3-1 constitutes an overall picture of the state of the art in the SADM domain, rather than a further reference for architects. In fact, as already mentioned, this chapter (i) did not synthesize on the attribute of our evaluation model, and (ii) neglected quality issues. As a consequence, we are not (yet) able to provide a measurement model able to answer to questions like: “What is the best SADM for these specified needs and quality?” and: “What is the accuracy of that result?”

3.6.3 Areas of improvement in SADMs

3.6.3.1 Meeting one category of needs at a time

Figure 3-2 presents what percentage of SADMs meets a level of architects needs. Based on Figure 3-2, we can deduce that SDAM developers assign greater importance to features like “Requirements Change Management” and “Concerns”, and lesser importance to features like “Knowledge Base Support” and “Empirical Validation”.

Table 3-2: Architects Needs Model Results.

	Activities	Empirical Validation	Knowledge Base Support	Refinement and Abstraction	Requirements Change Mng.	Risk Management	Solution Interaction Mng.	Tool Support	Concerns
ADD	Mx	Mi	I	Mx	Mx	Mi	Mx	Mi	Mx
ASC	Mx	I	Mi	I	Mx	Mx	I	Mi	I
CAFCR	Mi	Mx	Mi	Mx	Mx	Mx	I	I	Mx
CBSP	Mx	I	Mx	Mx	Mx	Mi	I	Mx	Mx
G&S	Mx	I	I	Mx	Mx	I	I	Mx	I
Rule Based	Mi	Mi	Mx	Mi	Mx	Mi	Mx	Mx	Mi
RUP	Mx	Mx	Mi	Mx	I	Mx	I	Mx	Mx
Siemens 4v	Mi	Mx	Mi	I	Mx	Mx	Mx	I	Mx
Tropos	Mi	Mi	Mi	Mi	Mx	Mi	Mx	I	Mx

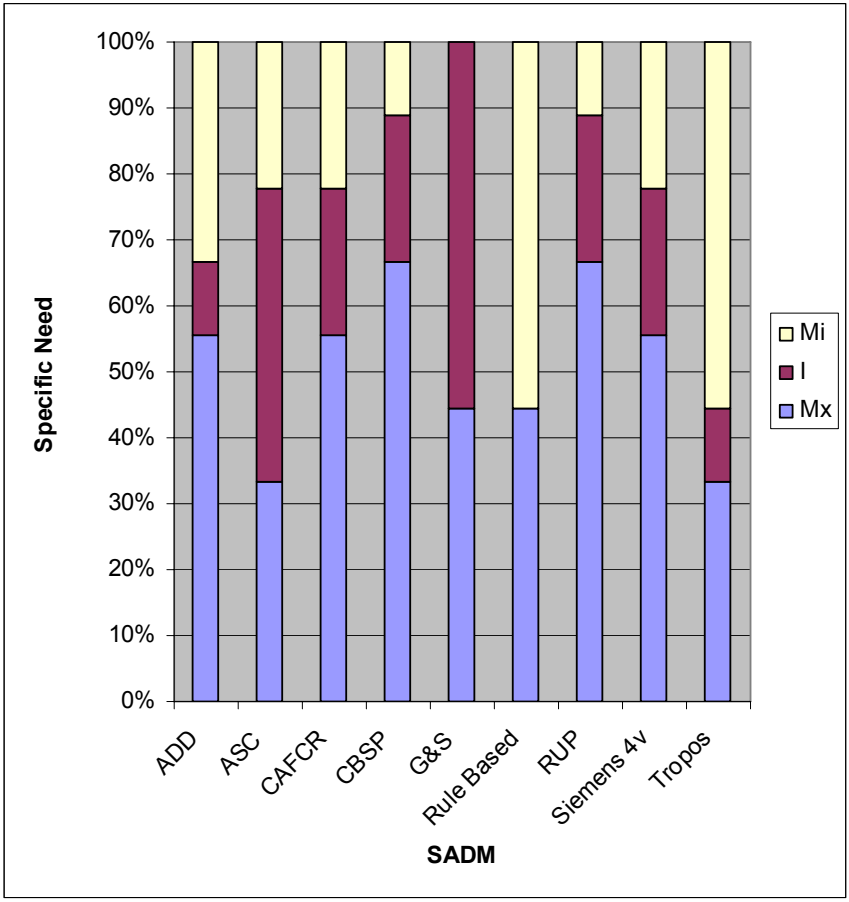

Figure 3-1. Software Architecture Design Methods with respect to specific needs fulfillment level (%).

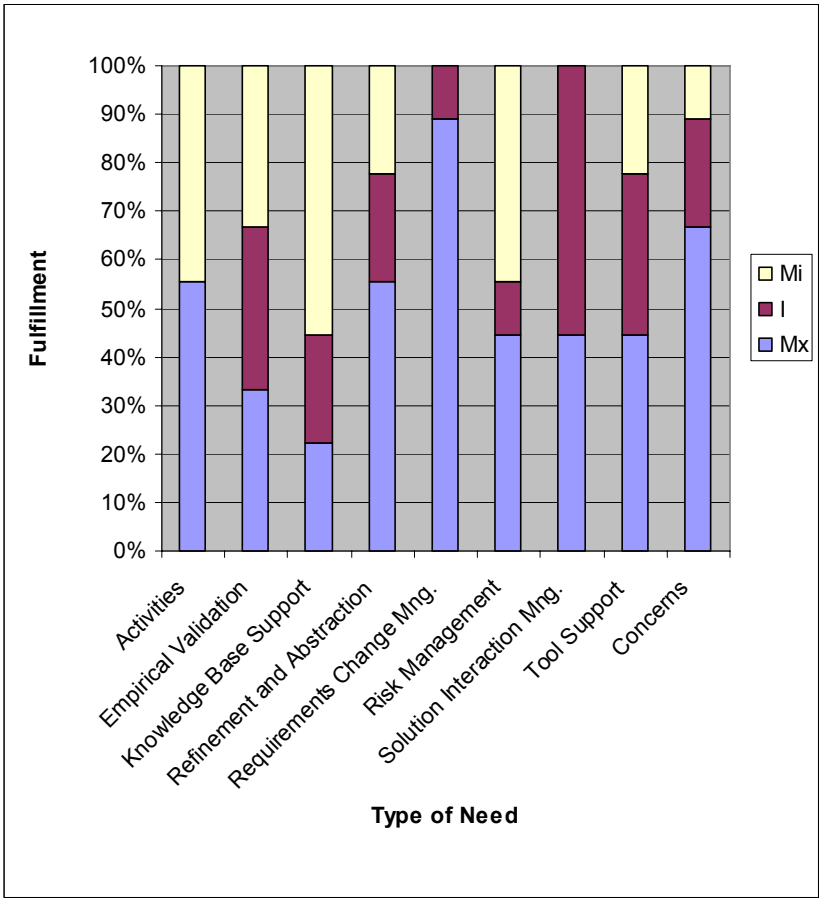

Figure 3-2: Specific needs fulfillment levels (%).

Table 3-3: Analysis of pairs of categories of needs.

	Activities	Empirical Validation	Knowledge Base Support	Refinement and Abstraction	Requirements Change Mng.	Risk Mng.	Solution Interaction Mng.	Tool Support	Concerns
Activities	6	2	1	5	5	3	1	3	4
Empirical Validation	2	3	0	2	2	3	1	1	3
Knowledge Base Support	1	0	2	1	2	0	1	2	1
Refinement and Abstraction	5	2	1	5	4	2	1	3	4
Requirements Change Mng.	5	2	2	4	8	3	4	3	5
Risk Management	3	3	0	2	3	4	1	1	3
Solution Interaction Mng.	1	1	1	1	4	1	4	1	3
Tool Support	3	1	2	3	3	1	1	4	2
Concerns	4	3	1	4	5	3	3	2	6

Table 3-4: Pair of categories of needs that no SADM is able to meet.

ID	Needs	
1	Empirical Validation	Knowledge Base Support
2	Knowledge Base Support	Risk Mng.

3.6.3.2 Meeting pairs of categories of needs

Table 3-3 shows the number of SADMs that are able to meet a couple of needs. Based on Table 3-3, we can observe, for instance: five SADMs are able to meet the needs of "Requirements Change Management" and "Activities"; vice versa, all the SADMs considered are unable to meet "Knowledge Base Support" and "Risk management". In other words, architects would be in trouble, if having both those neglected needs. Consequently, SADM developers should concentrate in providing those features. To identify potential SADM improvement areas, Table 3-4 describes pair of needs that all the SADMs unmet.

3.6.3.3 Meeting more than two categories of needs

Table 3-5 describes how many, and in what percentage, single needs, couple of needs, and so on up to nine-tuples of needs are there, that one or more SADM are able (resp. unable) to meet. Based on Table 3-5 we can observe, for instance, the considered set of SADMs are able to meet 68% of need-triples (see the item located in the third row and second column of Table 3-5).

Again, with the objective of identifying further SADM improvement areas, Table 6 describes SADM-unmet triplets of needs. Based on Table 3-6, we can observe, for example, that there is no SADM, which is able to meet the triple made by needs "Activities", "Risk Management", and "Solution Interaction Management" (see row 6 in Table 3-6).

3.7 Conclusion and future work

This chapter tried to answer to following question: "Do actual software architecture design methods meet architects needs?" To do so, we defined nine categories of software architects' needs, proposed an ordinal scale for evaluating the degree to which a given SADM meets the needs, and then applied this to a set of SADMs.

Based on results from the present study, we argue that there are two opposite but valid answers to the question placed: a) Yes, they do. In fact, we showed that one or more SADMs are able to meet each

individual architect needs that we considered (see row 1, column 2 in Table 3-5). b) No, they do not. In fact, we showed that there is no SADM which is able to meet any tuple of seven or more needs (see row 7, column 2 in Table 3-5), which means that there is still some work to do to improve SADMs to actually help architects.

In order to provide directions for SADM improvement, we presented couples of needs (see Table 3-4), and triplets of needs (see Table 3-6) that actual SADMs are unable to meet. We hope this should provide some valuable input to SADM developers.

Further investigation is needed to better understand why the coverage of our need is different across SADMs, and also to extend the list of needs categories.

Table 3-5: Analysis of architects needs available combination.

# Needs	Fulfilled		Not Fulfilled	
	#	%	#	%
One	9	100	0	0
Two	34	94	2	6
Tree	57	68	27	32
Four	47	37	79	63
Five	19	15	107	85
Six	3	4	81	96
Seven	0	0	36	100
Eight	0	0	9	100
Nine	0	0	1	100

Table 3-6: Triplets of categories of needs that no SADM is able to meet.

Needs		
Activities	Empirical Validation	Knowledge Base Support
Activities	Empirical Validation	Requirements Change Mng.
Activities	Empirical Validation	Solution Interaction Mng.
Activities	Knowledge Base Support	Risk Mng.
Activities	Knowledge Base Support	Solution Interaction Mng.
Activities	Risk Mng.	Solution Interaction Mng.
Activities	Solution Interaction Mng.	Tool Support
Empirical Validation	Knowledge Base Support	Refinement and Abstraction
Empirical Validation	Knowledge Base Support	Requirements Change Mng.
Empirical Validation	Knowledge Base Support	Risk Mng.
Empirical Validation	Knowledge Base Support	Solution Interaction Mng.
Empirical Validation	Knowledge Base Support	Tool Support
Empirical Validation	Knowledge Base Support	Concerns
Empirical Validation	Refinement and Abstraction	Solution Interaction Mng.
Empirical Validation	Requirements Change Mng.	Tool Support
Empirical Validation	Solution Interaction Mng.	Tool Support
Knowledge Base Support	Refinement and Abstraction	Risk Mng.
Knowledge Base Support	Refinement and Abstraction	Solution Interaction Mng.
Knowledge Base Support	Requirements Change Mng.	Risk Mng.
Knowledge Base Support	Risk Mng.	Solution Interaction Mng.
Knowledge Base Support	Risk Mng.	Tool Support
Knowledge Base Support	Risk Mng.	Concerns
Knowledge Base Support	Solution Interaction Mng.	Concerns
Refinement and Abstraction	Risk Mng.	Solution Interaction Mng.
Refinement and Abstraction	Solution Interaction Mng.	Tool Support
Requirements Change Mng.	Risk Mng.	Tool Support
Risk Mng.	Solution Interaction Mng.	Tool Support

3.8 Acknowledgements

We would like to acknowledge Patricia Lago for her useful suggestions and the following SADM authors for having been rated their own methods: Len Bass, Paolo Giorgini, Alexander Ran, Wendy Liu, Alexander Egyed, Henk Obbink, and Eric Yu.

4. Resolving Tradeoffs in Architecture Design: A Characterization Schema of Decision-making Techniques.

Software architecture may be defined as the set of relevant design decisions that affect the overall system functionality. Hence, architectural decisions are eventually crucial to the success of a project. Architects strive to meet as much as possible the stakeholders' needs despite the fact that these view the system from different perspectives and that they have conflicting goals. Consequently, architectural decisions (such as the selection of data structure, middleware, redundancy mechanism, interoperability mechanism, etc.) usually consist in tradeoff resolutions. The software engineering literature describes several techniques to choose among architectural alternatives, but it usually neglects to report on which circumstances which technique is more suitable. This leads to an absence of any systematic way for software engineers to choose among decision-making techniques for resolving tradeoffs in architectural design. The objective of this section is to provide a characterization schema of decision-making techniques as a guide for their selection. Moreover, in order to analyze the state of the art, provide a reference, and validate the completeness of our characterization schema, we allocated seventeen different decision-making techniques onto that schema; the literature had suggested those techniques for the activities of architecture design, COTS selection, and release planning. We conclude the section by describing open troubles and advices for avoiding mistakes while making decisions.

4.1 Introduction

4.1.1 Context and motivation

Developing software involves making a huge amount of design decisions about data structures, middleware, platform, language, and algorithms, during the whole software life cycle. "Systems engineers and managers need a reliable way of analyzing how effective their

design ideas or strategies are in meeting the requirements" [Gilb and Brodie, 2005]. Software architecture may be defined as the set of relevant design decisions that affect the system overall functionality. Hence, making good architecture decisions may be crucial to the system functionality, and eventually to the success of the project.

In the aim of architecting a software system, architects are requested to balance the many forces influencing the system development, as well as to work with all the stakeholders. The software architect "sits in between" all the stakeholders to reconcile their needs, wants, and concerns. Figure 3 shows stakeholders and their concerns, as well as the other forces influencing the activities of a software architect.

Difficulties in making architecture design decisions include [Grüenbacher et al., 2003]:

- Requirements are frequently captured informally while software architecture is specified in some formal manner. Consequently, there is a semantic gap to deal with.
- Non-functional requirements are difficult to specify in an architectural model.
- Software is often developed in an iterative way. Certain types of requirements cannot be well defined in the early development phases [Nuseibeh, 2001b]. This implies that sometimes architects take their decisions based upon vague requirements.
- Architectural decisions are hard to change after they are made because subsequent decisions are taken based on them; for example, detailed design and programming.
- Architecture decisions are key to achieving the desired system properties. Wrong architectural decisions may prevent to meet non-functional requirements, such as performance or maintainability.
- Stakeholders dealing with software architecture issues view the system from different perspectives. This usually implies conflicting goals and expectations with the developers and even difference in the terminology used.

For these reasons, the life of a software architect is a long (and sometimes painful) succession of suboptimal decisions made partly in the dark (Dixit Philippe Kruchten).

Consequently, architects need a reliable rigorous process for selecting architectural alternatives (i.e. resolving tradeoffs) and analyzing that made decisions mitigate risks, maximize profit, and minimize cost. To this aim, the literature describes several techniques to choose/analyze architectural alternatives; however it does not describe under which circumstances which technique is more suitable. In fact, the literature clearly describes the aim of, and the procedures to apply the techniques but it neglects inaccuracies, limitations, advantages, and disadvantages of such techniques. As recently stressed [Harman, 2007], this situation created the absence of any systematic way for software engineers to choose among decision-making techniques.

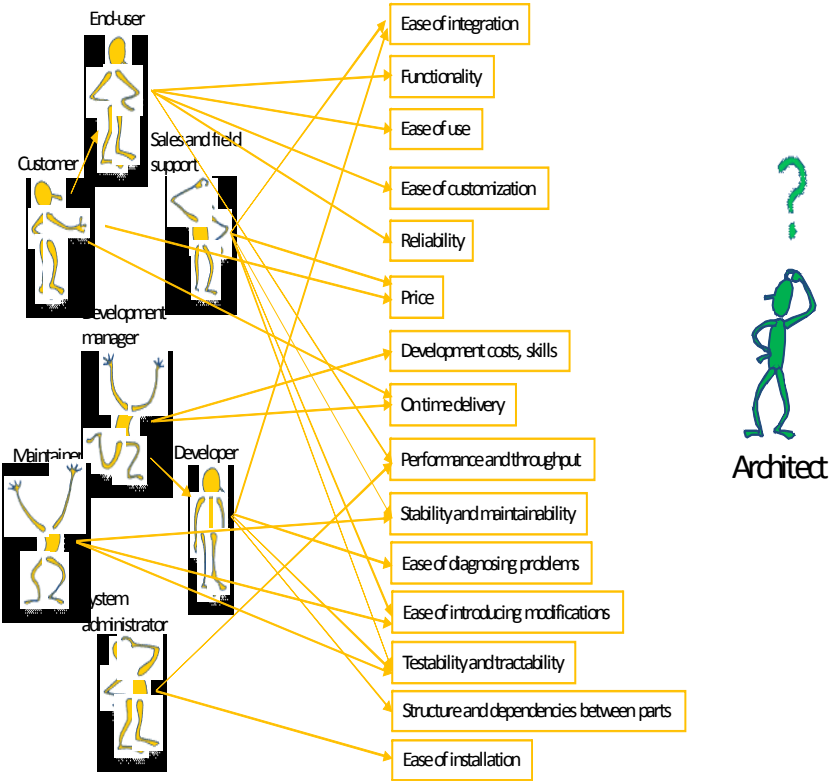

Figure 3: The architect's balancing art (Source: Rational Software 1998).

4.1.2 Contribution

In this section we provide a characterization schema for enabling the selection of decision-making techniques for choosing among architectural alternatives. Then, we situate - into the proposed characterization schema - seventeen already existent decision-making techniques belonging to different application domain (i.e. software architecture design, COTS selection and release planning); this is aimed to:

- Validating the completeness of our characterization schema,

- Acting as a reference,
- Exposing advantages and disadvantages of current decision-making techniques.

In our view, the proposed characterization schema provides a deep knowledge on decision-making techniques which would help the design phase in: (i) employing the decision-making technique, by making explicit its principles, the differences with the others techniques, and, most importantly, its limitations (i.e., which troubles may derive from using it), (ii) selecting and customizing the decision-making technique that is the most suitable to use based on the characteristics of the issue to solve, the usage context, and the troubles encountered while deciding.

4.1.3 Scope

There are three main types of decision-making techniques in software engineering:

Selecting the first available alternative: in our experience, some of the design decisions taken during the development process do not include any selection among alternatives; designers take the first alternative that seems to meet the requirements without searching for better ones. Such an approach supports the project schedule, not the optimization of the system overall design, and it is partially addressed in the *naturalistic* decision branch, where decisions are studied as the product of intuition, mental simulation, metaphor, and storytelling [Klein, 1999].

Selecting among a finite amount of alternatives: Usually, the amount of available architectural solutions is in a finite number. The branch so called *multi-attribute* decision-making techniques addresses the trouble of selecting among a finite amount of available alternatives; it requires that the value of any alternative can be expressed by a finite set of independent attributes, such as cost and performance. This is the "classic and rationale" approach of any textbook on engineering.

Selecting among an infinite amount of alternatives: Some decisions in software engineering consist in the definition of the right balance among conflicting objectives. For example, finding the right balance between cohesion and coupling is a typical trouble when

defining classes in an object-oriented application; in this case, the amount of available alternatives (i.e. classes) is infinite. It addresses the trouble of selecting among an infinite amount of available alternatives, the branch so called *multi-objective* decision-making techniques (and, more in general, the optimization research field); again, it requires that the value of any alternative can be expressed by a finite set of independent attributes.

The activity of ***analyzing*** the correctness of the chosen architectural alternative is alike to the activity of ***selecting*** among a finite amount of architectural alternatives. Consequently, with the aim to characterize techniques related to architectural tradeoffs resolutions (both made and to make), in the rest of the section we focus on multi attribute decision-making techniques (point 2 above).

4.1.4 Vocabulary

The terminology used in software engineering varies considerably from author to author. To avoid misunderstandings we adopted the following definitions:

Issue: it is what we want to address, the reason why we make the decision. An example of an issue is the selection of a middleware. Issues are described by a set of attributes.

Attribute: it is a property of the issue at hand. It is also sometimes called 'objective' or 'criteria'. Examples of attributes are "Cost", "non-Functional requirements", and "Time constraints".

Solution: It is the thing that: 1) we are looking for, 2) resolves the issue, 3) we may eventually want to choose among the available alternative solutions. It is also called 'available alternative'. Examples of solution are the architectural patterns "Pipe and filter", and "Blackboard".

Level of fulfillment: it denotes the extent in which a specific attribute is addressed by an alternative. It is also called 'satisfaction level' [Simon, 1996b]. An example of Level of fulfillment is a "Medium" level of Cost (Attribute) as provided by Blackboard (Solution).

Set of Characteristics: it is the set of main activities in which any decision-making technique can be decomposed; such activities are one each other orthogonal; their identification constitutes the core of this paper.

Characteristic: it is a specific type of variability in decision-making technique. An example of characteristic is the activity "Identification/Description" of an attribute of the issue to resolve.

Value (of a characteristic): it is a specific mechanism to realize an activity. In our schema, each value belongs exactly to one characteristic. Examples of value for identifying/describing an attribute (e.g. Performance) of the issue to resolve (Characteristic) are using "Words" (e.g., "Response time") rather than "Words and metrics" (e.g. "Response time <= 1 sec").

Figure 4 describes the relations among different terms as used in this section, by using a class diagram. Figure 6 shows further terms together with additional concepts that following sections provide.

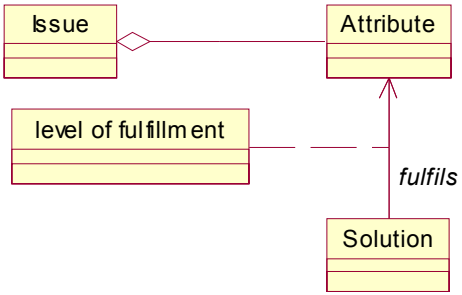

Figure 4**: Terms relationships.**

4.1.5 Structure

The remainder of this chapter is organized as follows. Section 4.2 briefly sketches related works. Then Section 4.3 describes the key idea behind this study. Section 4.4 describes troubles in making decisions and Section 4.5 describes the components of the proposed

characterization schema. Sections 4.6 provide the characterization schema. Section 4.7 describes trends in actual software engineering decision making technique. Section 4.8 and 4.9 describes open problems and advices in applying multi-criteria decision making techniques. Section 4.10 sketches possible future works and concludes the chapter.

4.2 Related work

In this section we briefly describe the main studies that helped us to shape the present work. However, to the best of our knowledge, no study already addressed our research question up to now, that is: "How to select a decision-making technique when there is to choose among design alternatives?"

Our work was inspired by the report of Moore et al. [2003], where they discuss how to change a decision-making technique (i.e. the first version of CBAM from [Kazman et al., 2001]) in order to avoid specific troubles. Our work is very similar to the Moore's one but we include a larger spectrum of troubles and variability in decision-making techniques. We do not regard the second version of CBAM better than the first but we regard it as a version less prone to troubles that are typically relevant in SEI practical application.

Search based software engineering is a branch of software engineering recently born [Harman and Jones, 2001] with the aim to study the application on software engineering of metaheuristic optimization algorithms such as genetic algorithms [Goldberg, 1989], simulated annealing [Laarhoven and Aarts, 1987] and tabu search [Glover and Laguna, 1997]. M. Harman [2007] recently described the current state and future trends of search based software engineering; such a work shares with the present one the acknowledgement of the current lack in software engineering of guidelines for selecting the most suitable decision making technique. Because architectural alternatives are in practice a finite number we believe that the resolution of architectural tradeoff can benefit from the application of multi attribute decision-making, rather than from metaheuristic optimization techniques.

Our study shares with Mohamed et al. [2005] the idea to customize the decision technique based on attributes such available effort and

project criticality. However their focus was narrower and more fine-grained; it applies only to COTS selection technique and they are dealing just with two troubles out of the many that software architects have to face.

Wanyama and Far [2005] propose a technique in the context of COTS selection, after analyzing the deficiencies in activities supported by other techniques. Differently from their approach we do not take into account the supported activities as characteristic that is important for selecting a technique. Additionally we do relay on the expectancy that a decision-making technique exists that is generally better than another one.

Blin and Tsoukiàs [2001] describe troubles in practical use of the ISO 9126 and IEEE 1061 standards for evaluating COTS. The main commonality with the present work consists in revealing technique inaccuracies and related troubles.

Karlsson et al. [1997] propose an experimental comparison of methods for prioritizing requirements; however, such methods were based on pair-wise comparison rather than multi attributes, as it is the one described in the present work.

Again, Karlsson et al. [2006] propose an empirical comparison to investigate the different performances of using the ordinal scale or the ratio scale in requirements prioritization. Such an approach belongs to the values “terms” and “ratio” respectively, in the characteristic “Type of fulfillment”, in our proposed characterization schema (Section 4.5.3). Our study shares with such an approach the view that the “techniques using a richer scale tend to be more time-consuming and complex to use […] thus, there is a trade-off […]” between the effort that more detailed model requires, and troubles caused by a coarser model. Differently from such an approach we do not focused on characterizing the differences, in specific circumstances, caused by the use of among the two values of the characterization schemas (“terms” and “ratio”) but we just expose the existence of such a difference (i.e., “ratio” are more detailed and time consumed than “terms”).

Triantaphyllou [2004] describes the differences among usages of the Analytic Hierarchy Process (AHP) method [Saaty and Saaty, 2000]. Unlike our work, the author only focused on comparing different usages

of AHP without providing guidelines to select a decision-making technique.

4.3 The proposal

4.3.1 Key idea

Each decision-making technique defines its own conceptual model of the world to which it applies, and the entities it manipulates, or creates, and their relationships; that is in some way, its own "ontology". These conceptual models, being models, abstract out some aspects and emphasize others. Neglected aspects may cause troubles during the decision-making activity. We can make a model encompassing a wider world, but the more complex the associated conceptual model, the harder the technique is to use [Karlsson et al., 2006]. Consequently, the complexity of models must be properly tuned based on the application context. Unfortunately, we do not have empirical data to define when the complexity of a model is sufficient [Berander and Andrews, 2005]; in other words, the specific circumstances under which a specific technique causes (or not) a specific trouble are not yet defined. However, software engineers have often identified gaps or deficiencies in the conceptual model of a technique that clearly lead to specific troubles in making decisions. Our key idea in this section is to expose and organize in a useful way, namely by a characterization schema, to what extent each decision-making technique is prone to specific troubles. Therefore, the level of proneness of a specific technique to specific troubles becomes a quality attribute of the decision-making technique. Since we cannot decide in advance what degree of complexity of modeling is sufficient, instead of proceeding by trial and error, we offer guidelines on which complexity to emphasize to avoid specific trouble(s).

Diagram in Figure 7 describes the suggested process for tuning a decision-making technique; the decision-making technique is chosen based on the envisioned/occurred troubles that decision-makers want to avoid.

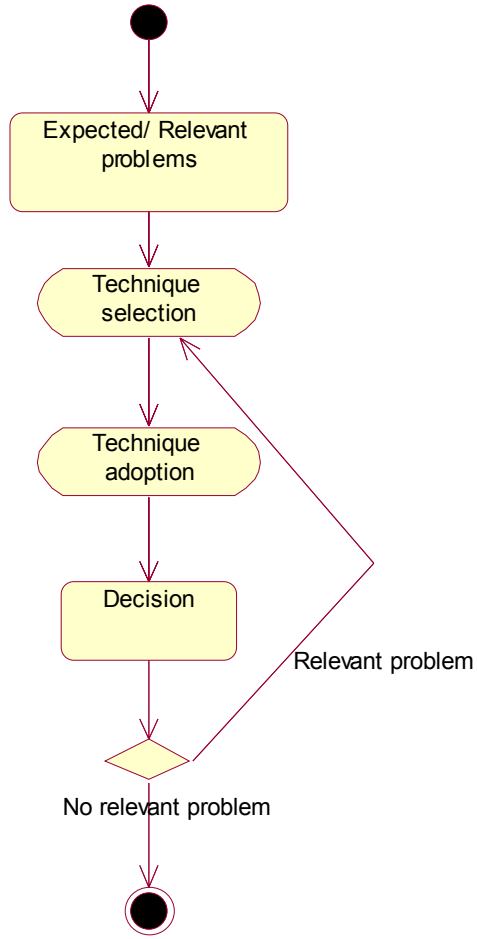

Figure 7: The suggested process for selecting a decision-making technique.

4.3.2 Approach

Our approach is to synthesize from extant knowledge in the current literature the level of proneness of specific decision-making models to specific troubles while selecting among design alternatives. Hence, in this work, we are not introducing any new decisions making technique, nor even adding knowledge to any of them; we are merely collecting and classifying existing ones. In order to provide support for selecting a decision-making technique for choosing among design alternatives, we propose a characterization schema of decision-making techniques. We defined the characterization schema by following five steps in an iterative way.

Our first step was to analyze the *similarities* in the decision-making techniques as the engineering community proposed. We discovered that decision-making techniques for selecting among design alternatives can be decomposed in five main activities: Identification of the attributes of the issues, Description of the fulfillment level, Time to express the importance of an attribute, Description of the attributes importance, and Description of the risk related to solutions.

In a second step we analyzed *variability* among techniques. Hence we realized that different activities are realized in different ways by different techniques. The five different activities of a decision-making technique (a.k.a. variation point) are called in the following **characteristics**; each the different mechanisms (a.k.a. variants) to address such activities are called in the following **value.** The spectrum of variants for a variation point is referred with the term value set.

As a third step, we collected troubles encountered while selecting among software design alternatives (see Section 4.4).

In the fourth step we realized which specific values of the characteristics are reasonably and/or empirically more prone than others to specific troubles.

In order to create a useful characterization schema, we refined the analyzed troubles and values by achieving the following peculiarities (see Figure 6):

- Each valid decision-making technique has just one value in each characteristic (this applies whether the technique is already proposed or not).
- Each value in the characteristic of the characterization schema is associated with one or more troubles encountered while applying decision-making techniques in a software context.
- All troubles taken into account are associated with one or more values of at least one characteristic.

This characterization refinement has been done in an iterative manner by defining the granularity of values via a clustering activity and by hiding troubles.

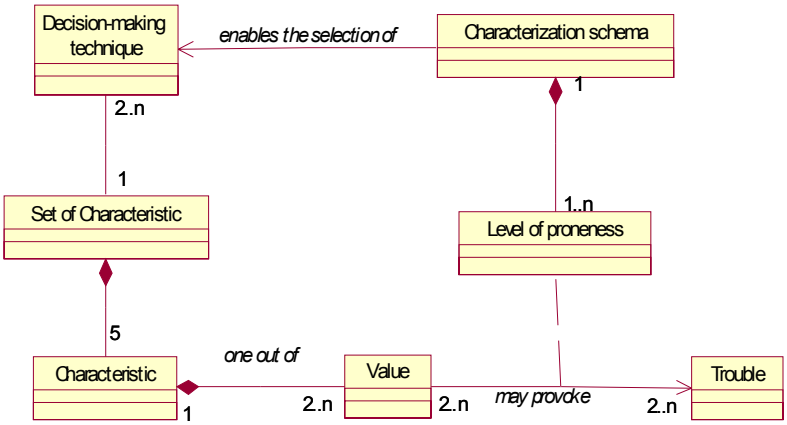

Figure 6: Peculiarities of the proposed characterization schema.

Finally, we situate - in our proposed characterization schema - seventeen different decision-making techniques found in the literature in the domains of architecture design, COTS selection, and release planning. Such situation validates the completeness of the proposed characterization schema, and it provides a useful reference for analyzing the state of the art.

Note that the trouble of selecting the best decision-making technique could lead to a paradox: the selection of a decision-making technique "needs to use the best decision-making method! This decision-making paradox which makes any attempt in solving this trouble to be of limited success" [Triantaphyllou, 2004]. However, we do not fall in such a paradox because we are not suggesting "the best" technique neither a technique to select such a best technique; we provide specific directions (i.e., which technique is more prone to specific troubles) to select the decision-technique based on specific troubles.

4.3.3 Limitations

It is the absence of validation the main limitations of this work; however, it provides a reasonable base for the content of this work, the fact that it presents already established principles from a new point of view (i.e., software engineering).

We did not take into account issues concerning:

- How to apply a value of the characterization schema (i.e., how to find the fulfillment level, how to find the amount of risk, etc.).
- Decisions interdependency,
- Decisions side effects
- Decision hierarchy,
- Stakeholders' roles involved in the decision,
- Scalability of the decision-making technique,
- Techniques to select attributes,
- How to search for alternatives,
- Social concerns [Klein, 1999].

and other details that can have relevant relation troubles (see Section 4.4). Hence, these neglected aspects are the main differences among those decision-making techniques that seem similar based on the location in the proposed characterization schema (see Table 4-5).

4.4 Troubles in decision-making

The present section describes some relevant real troubles that occur when selecting among software design alternatives. Each of the discussed trouble may eventually provoke the selection of a wrong alternative or the usage of an expensive, in case infinite, decision-making process.

In the remaining of this section, the title of each sub-section describes one of the discussed difficulty from point of view the decision maker; additionally, the title of each sub-sub-section describes possible causes for the difficulty currently discussed.

4.4.1 Attribute interpretation

Large or complex projects require the presence of some designers, each with specific skills and responsibilities. In this context, different stakeholders have different knowledge and view of the system; therefore, they could not share the meaning of terms.

4.4.1.1 Perception of a term

Because large systems provide different functionalities, there is the possibility that different stakeholders use the same name to refer to different issues of a system.

Example: Let stakeholder A be in charge of the “response time” for system service α, and stakeholder B be in charge of the “energy consumption” for system service β. By using just the term “performance” stakeholders A and B would mean the “performance” of two different system services (α and β, respectively).

4.4.1.2 Value behind a term

For a given system, stakeholders may have different perceptions of the level of an attribute.

Example: (1) Let stakeholders A and B be in charge of the “energy consumption” and the “response time” of a certain system, respectively. Stakeholder A relates the term performance to a value “10 msec” while B to a value of “2 sec”. A design solution, that duplicates “the response

time" as a side effect, would be considered as "acceptable" by the stakeholder A and as "unacceptable" by B.

4.4.2 Solution properties misunderstandings

4.4.2.1 Coarse-grain indication

The grain utilized for describing the satisfaction level is so coarse to neglect relevant levels.

Example: the costs of two alternatives would be both judged as acceptable ("+") despite the fact that one solution is two times more expensive than the other one; such a gap in the cost would not influence the decision-making process and hence it could provoke the selection of a bad solution.

4.4.2.2 Value perception

The satisfaction level is described by symbols or terms that are misunderstood among stakeholders. According to Gilb "non-numeric estimates of impact are difficult to analyze and improve upon." [Gilb and Brodie, 2005]

Example: Let stakeholders A and B agree on that it would double the energy consumption of a given system, the decision of deploying two system tasks in the same physical node. However, since they have different views and responsibilities of the system, they could characterize differently the level of fulfillment of such a decision for the attribute "Energy consumption": e.g., "bad" and "adequate", respectively.

4.4.2.3 Risk

In general, some risk is associated with estimating in what extent a solution fulfills an attribute in a specific level. According to Gilb [Gilb and Brodie, 2005], "The uncertainty estimate is at least as important as the main estimate". Obviously, the finer-grain the fulfillment level description is the higher the risk is and hence it is important to take into account risk for fine-grain fulfillment level.

Example: Assume that the decisions of adopting a certain middleware would reduce the cost by half. Let then, in practice, the real

cost double, due to the inexperience of programmers to building components on that middleware.

4.4.3 Managing stakeholders effort

4.4.3.1 Little time

Let a decision be made in a moment in the development process when there is limited amount of time to think about it. Stakeholders might have little time to interact for decision-making.

Example: Stakeholders are one day before the deadline of an important release and an algorithm for related functionality has to be chosen or changed, and then implemented. Stakeholders are not ready to spend a lot of time to carefully describe all the necessary attributes, and needs related to all the algorithms available for the given case.

4.4.3.2 Little availability

Stakeholders may have few chances of interaction due to a geographically distributed work environment or different time schedules.

Example: Let a company have half of its stakeholders in Australia and the remaining half part in Italy; the time offset of 9 hours makes any schedule difficult to establish, and stakeholders have quite limited possibilities to talk to each other.

4.4.3.3 Interest

A long decision-making process would decrease the stakeholders' perception of the decision-making process utility. Hence stakeholders may enact the decision-making technique rather superficially [Moore et al., 2003].

Example: Let stakeholders feel tired after one hour spent in applying the AHP method to derive the importance of each attribute. Consequently, they begin to see no further benefit in continuing that work, and start to enact the remaining steps to hastily decide.

4.4.4 Solution selection

After applying the decision-making technique, the suggested solutions are described by a palette that makes it difficult to select the most suitable solution.

4.4.4.1 Too much complex description

The output of the decision-making technique may be too much complex to allow the selection of the most suitable solution.

Example: The issue is described by twenty solutions and thirty attributes; it is an ordinal scale from 1 to 1000, the fulfillment level of an attribute. Hence, in order to make decision, stakeholders have to analyze 30 attributes 20 times, and discriminating 1 out of 1000 scores each time, which might result into a task quite strong to enact.

4.4.4.2 Multiple suitable solutions

The output of the decision-making model may propose too many solutions with the same higher score.

Example: Let three solutions and three attributes (e.g. cost, efficiency, and reliability) describe the issue, and an on-off value described the need (e.g. a cost less than 1k€, a response time less then 1 sec., and mean time to failure more than 8 months). It is possible that all the aforementioned solutions satisfy all the attributes, based on the output of a given decision-making technique.

4.4.5 Stakeholders disagreement

The suggested solution does not fulfill stakeholder expectations because their needs have been modeled in a wrong way.

4.4.5.1 Coarse-grain description of needs

The needs were so coarse-grain described that one or more relevant attributes were neglected.

Example: there are three attributes (e.g. cost, efficiency, and reliability) to describe the issue, and the need was described by an on/off value. The suggested most suitable solution is the only one that

satisfies all together the three attributes; however stakeholder would prefer another solution which – compared with the proposed one – shows1 ns less of efficiency and costs 1000 times less.

4.4.5.2 Linear and monotonic need

In order to cope with complexity, the need is modeled to grow linearly as the fulfillment grows [Keeney and Raiffa, 1976].

Example: Let the amounts of "energy consumption" and "reliability" be the attributes assumed for COTS selection; additionally, let energy consumption be more important than reliability. Consequently, a decision-making technique proposes a solution with 1mWh energy consumption and 10 years mean time to failure. Whether there was no practical need for a mean time to failure higher than 6 years, stakeholders would want to prefer a solution providing 1nWh energy consumption and 6 years mean time to failure. Notice that also in the latter the energy consumption is more important than reliability (but not as important as in the former).

4.5 Components of the characterization schema

In this section we present the proposed decision-making characterization schema. This is a schema for selecting decision-making technique for selecting among available software design alternatives.

We structured such a schema in five different characteristics, as described in the following five sub-sections. We also present the value set available for each characteristic; one value set per sub-sub-section for each of the following five sub-sections; values are described in an increasing order of complexity in same set.

Of course, in order to show variability among decisions making technique, the proposed characterization schema is not the only possible schema; however, it is the only one that allows the selection and customization of decision-making technique.

4.5.1 Attribute identification

When making a decision, several stakeholders desire different attributes (i.e. performance, cost) of an issue. It is extremely important to have a common and clear understanding of every attribute between stakeholders who are involved in the decision-making process.

4.5.1.1 Term

Let the attribute of an issue to solve (or optimize) be identified by using a common term like, for instance, “Performance”. The use of term requires little effort and interactions form stakeholders but it may cause meaning misunderstandings [Gilb and Brodie, 2005] [Moore et al., 2003]. Hence such a value (i.e. the use of term as attribute identification mechanism) is particularly effective in a context where all the stakeholders have the same understanding of the issue to solve.

4.5.1.2 Term related to a use-case

Let the attribute of an issue be related to the description of a use-case. For example, let the term “Performance” relate to a specific use-case. Using use-case for identifies attributes should avoid misunderstandings of attributes meaning [Moore et al., 2003],but it might require stakeholders to spend a relevant effort (in analyzing the given use case). Large and/or complex projects require designers with specific skills and responsibilities. In this context, different stakeholders have different views of the system and therefore they would mean different concepts using a term without a related UML use-case.

4.5.1.3 Term and measure together

The attribute of the issue to solve (or optimize) is described by using a term (e.g. Performance) and a measure for that term (e.g. 2 sec). While this type of attribute description requires little effort and limited interactions among stakeholders, it may avoid misunderstandings regarding perceptions behind term [Gilb and Brodie, 2005].

4.5.1.4 Term referring a use-case and measure together

The attribute of the issue is related to a scenario description and a related measure. For instance, the term "Performance" is related both to an use-case of a certain system, and a specified measurement value (e.g. 2 sec). While this type of attribute description requires stakeholders to enact a certain effort in analyzing the related use-case, it may avoid misunderstandings with regard to attributes meaning and perceptions behind term [Gilb and Brodie, 2005] [Moore et al., 2003].

4.5.2 Need modeling

This characteristic of the characterization schema regards the mechanism to describe the attributes utility (i.e. level of need, importance, desire).

4.5.2.1 No articulation

An overall number is not provided because the need was not expressed (see Section 4.5.5.1 above) [Andersson, 2000].

4.5.2.2 Rank

Attributes are sorted based on their importance; an iterative process selects the solution: the current iteration eliminates solutions (if any) that are unable to provide the maximum fulfillment level for the current most important attribute. This method is suitable when the amounts of solutions and attributes are extremely large, to be conveniently analyzed otherwise [Andersson, 2000].

4.5.2.3 Direct weight

Stakeholders weight each attribute; the weight represents the attribute utility (i.e. level of need, importance, desire). This is the most common used method, and it considers the relation between importance and fulfillment level as a linear relationship [Moore et al., 2003].

4.5.2.4 Analytic Hierarchy Process

Thomas Saaty devised AHP in the early seventies [Saaty and Saaty, 2000]. The AHF is based on the idea of decomposing a multiple criteria

decision-making problem into a criteria hierarchy. At each level in the hierarchy, it is assessed the relative importance of factors by comparing them in pairs. Finally, the solutions are compared in pairs with respect to the criteria. We can estimate the effort for applying this method by observing that if *n* is the number of objectives, the method requires *n*(n-1)/2* comparisons. AHF differs from the "Direct weight" because it offers a rigorous approach for weights elicitation. Consequently, it provides more precise results but its usage requires more effort than the method based on direct weight values.

4.5.2.5 Ranges

Different weights are associated to specific ranges of fulfillment levels for each attribute. This method provides more accurate need-description than the direct weight and AHP; in fact, the ranges allow to provide a non-linear and a non-monotone description of the need [Moore et al., 2003]. The usage of this method requires that a considerable effort be invested, which is proportionate to the amount of ranges provided to each attribute.

4.5.3 Type of fulfillment description

Different solutions address different attributes of an issue to a different extent. This characteristic of the proposed decision-making characterization schema regards the type of description for the fulfillment level provided by the available alternatives (i.e. solutions).

4.5.3.1 On-off scale

Let the level of attribute fulfillment be described by two terms. This is suitable in case an attribute of the issue to solve has just two states (a.k.a. atomic attribute), e. g. Fulfilled, not-Fulfilled. For instance, a software requirement is usually stated as "satisfied" or as "not satisfied": there is no meaning in terms like percentage or ratio of a requirement satisfaction. Attributes of this kind are frequently utilized to filter entities to include in/exclude from an evaluation set.

4.5.3.2 Ordinal scale (terms, symbols or numbers)

The levels of attribute fulfillment are roughly described by using terms (e.g. terms like “good”, “standard”, and “bad” or symbols like “++”,and “-”). It is suitable in case the solutions to select are still in a high level of abstraction; therefore, the attribute level of fulfillment is not completely defined by the solution but other future decisions will impact on the fulfillment level. For example, a high level decision, like a specific architectural pattern, can be considered as “good” or “bad” regarding a non functional characteristic, and we are not allowed yet to relate it to a specific system metric because other several lower design decisions (i.e. programming languages or algorithms) would affect such a system property [Chung et al., 1999]. This type of description may be too coarse-grain to differentiate solutions.

4.5.3.3 Ratio (and Percentage)

The level of fulfillment is described by using a ratio or a percentage. It is suitable in case the attribute to fulfill is not atomic but it is reasonable to be decomposed in sub-parts. For instance, it is important to describe the percentage of budget which a solution requires [Moore et al., 2003].

4.5.4 Risk description

The improbability of a solution to achieve a specific level of attribute fulfillment is described by the risk. The risk associated to an alternative depends on the level of knowledge that decision makers have about the alternative performances (e.g. a new middleware may provide a better response time but less reliability) or a well-stated solution peculiarity (e.g. the mean time to failure of using a LAN is well established). Different situations lead decision-makers to accept different levels of risk. For instance, risky alternatives should be avoided when dealing with safety-critical systems while they may be chosen in case of highly iterative development process.

4.5.4.1 Risk is not expressed

You can, for instance, consider two alternatives with the same performance but different levels of risk; without taking into account the risk factor the alternatives seem equivalent while in reality one is objectively better than the other one.

4.5.4.2 Risk is inferred from disagreement among decision-makers

In this case, each decision-maker estimates the level of fulfillment of a specific decision without expressing the level of confidence in such estimation. Afterwards, the risk related to a specific decision is inferred from the level of disagreement among the different decision makers.

4.5.4.3 Risk is related to the whole decision

In this case, the amount of risk is usually proportional to the reciprocal of the level of knowledge on the solution performance.

4.5.4.4 Risk is related to each level of fulfillment

In this case, the amount of risk is usually proportional to the reciprocal of the probability that a solution provides a specific level of fulfillment for a specific attribute.

4.5.5 Time for expressing the issue

This characteristic includes the "When" into the Decision-Making Characterization schema, i.e. when decision makers describe their preference about different attributes.

4.5.5.1 No articulation

The relative importance of the attribute is never expressed; hence all the attributes have the same importance. This method is suitable when stakeholders cannot or do not want to express their need (i.e. importance of attributes) because the complexity of the issue is too much or too little, respectively [Andersson, 2000]. For instance, in case of two attributes and three solutions, in order to reason about the decision to take, it is reasonable to look just at the solution characteristics instead of spending effort to characterize the stakeholders' needs (i.e. importance of attributes).

4.5.5.2 Single shot

It is enacted just one time before the solution selection, the process for expressing both the need that concerns (i.e. the importance of) the attributes of the issue to solve, and the characteristics of the solutions.

This method is suitable when the stakeholders can express their need a priori, without investigating the state of the art or analyzing the available alternative [Andersson, 2000].

4.5.5.3 Repetitive

The steps of i) searching available alternatives and ii) expressing the need in terms of attributes of the issue to solve, are enacted in an iterative way. Stakeholders start with a rough expression of their need and then they refine such an expression while the search moves on. Since stakeholders take part in the searching activity, they will accept the solutions more likely [Moore et al., 2003]. The description of need is iteratively refined and improved, hence such a method would mitigate troubles related to needs description [Andersson, 2000].

4.6 Characterization schema

Table 4-1 and Table 4-2 describe the proposed characterization schema by summarizing on relationships between value sets that the characterization schema utilizes and decision-making troubles. Each cell describes the level of proneness to a specific trouble, with respect to the other values of the same characteristic. An empty cell means that such a characteristic does not affect the specific trouble; otherwise, the more is the amount of minuses (i.e. "-") the more the related value is worst in respect of (i.e. more prone to) the specific trouble. Note that a lesser amount of minuses means less prone; this "less prone" means "better or equal than" (i.e. not worse than) rather than "better than" (as it would be expected); this is due to the fact that a trouble can be provoked by using two decision-making techniques even with different levels of proneness to such a trouble. Therefore, for a given characteristic, we have the following conditions for values A and B with an amount of minuses m(A) and m(B) for a certain trouble:

1. if m(A) < m(B) then A is "better or equal than" (i.e. not worse than) B, i.e.:
   1.1 A cannot cause that trouble if this did not occur while using B;
   1.2 A can cause the trouble if this occurred while using B;
2. if m(A) = m(B) then:
   2.1 A can cause the trouble that occurred while using B;
   2.2 A can cause the trouble that did not occurred while using B.

Hence, decision-makers should select the decision-making technique that shows one minus value (or as less as possible in accordance to compatibility issues) with respect to the troubles that are considered relevant/probable.

We stress the fact that the content of Table 4-1 and Table 4-2 relies on literature, which already provided their validation as described in Table 4-3 and Table 4-4, respectively.

In conclusion, based on the nature and content of Table 4-1 and Table 4-2 we note that it is hard to select a decision-making technique because a decision-making technique cannot be defined as better than another; this is due to two main reasons:

Two decision-making techniques can provoke the same trouble, even when they show with different levels of proneness to such a trouble (see point 1.2 above). Consequently, a more accurate model does not always pay off: this depends on the issue to address and the usage context.

No decision-making technique is more prone than other techniques to the whole set of troubles taken into account in this study. In fact, let a decision-making technique (e.g. *A*) be less prone than another one (e.g. *B*) to a given set of troubles (e.g. {i, j}); then, the former is also more prone than the latter to other given troubles (e.g. {k, m}). Consequently, for resolving architectural tradeoffs, because the quality of a technique depends on which troubles decision-makers want to avoid, there is no "best" decision-making technique to look for; however, there are techniques that are more prone than others to specific troubles.

Table 4-1: Characterization schema (Part 1).

Characteristic	Value	Troubles				
		Attribute meaning		Solution Property		
		Term Perception	Value Perception	Coarse grain	Value Perception	Risk
Attribute Identifier	Term	- -	- - -			
	Term + UseCase	-	- -			
	Term + Metric	- -	-			
	Term + UseCase + Metric	-	-			
Type of fulfillment	On-Off			- - -	- - -	-
	Terms (symbols)			- -	- -	- -
	Direct Ratio			-	-	- - -
	AHP on fulfillment level			-	-	- - -
Expression Time	No Articulation	- - -			- - -	
	A priori	- -			- -	
	Progressive	-			-	
Desire Modelling	No Articulation			- - -		
	Rank			- -		
	Direct Weight			-		
	AHP on Attribute			-		
	Ranges			-		
Risk	no risk					- - - -
	Disagreement as risk					- - -
	Risk on Decsion					- -
	Risk on fulfillment level					-

Table 4-2: Characterization schema (Part 2).

Characteristic	Value	Troubles						
		Stakeholders Effort			Solution		Stakeholders	
		Time	Availability	Interest	Complex Description	No best solution	Coarse-grain Desire	Linear Desire
Attribute Identifier	Term	-						
	Term + UseCase	- -						
	Term + Metric	- - -						
	Term + UseCase + Metric	- - - -						
Type of fulfillment	On-Off	-			-	- - -	- - -	
	Terms (symbols)	- -			- -	- -	- -	
	Direct Ratio	- - -			- - -	-	-	
	AHP on fulfillment level	- - - -			- - -	-	-	
Expression Time	No Articulation	-	-	- -				
	A priori	- -	- -	- -				
	Progressive	- - -	- - -	-				
Desire Modelling	No Articulation	-			-	- - - -		- - -
	Rank	- -			- -	- - -		- - -
	Direct Weight	- - -			- - -	- -		- - -
	AHP on Attribute	- - - - -			- - - -	-		- -
	Ranges	- - - -			- - - -	-		-
Risk	no risk	-			-	- - - -		
	Disagreement as risk	- -			- -	- - -		
	Risk on Decsion	- - -			- - -	- -		
	Risk on fulfillment level	- - - -			- - - -	-		

Table 4-3: Main references used to define Table 4-1

Characteristic	Value	Troubles				
		Attribute meaning		Solution Property		
		Term Perception	Value Perception	Coarse grain	Value Perception	Risk
Attribute Identifier	Term Term + UseCase Term + Metric Term + UseCase + Metric	(Gilb and Brodie, 2005), (Moore et al., 2003)	(Gilb and Brodie, 2005)			
Type of fulfillment	On-Off Terms (symbols) Direct Ratio AHP on fulfillment level			(Andersson, 2000), (Keeney and Raiffa, 1976)	(Gilb and Brodie, 2005)	(Gilb and Brodie, 2005), (Andersson, 2000)
Expression Time	No Articulation A priori Progressive	(Andersson, 2000), (Moore et al., 2003)			(Andersson, 2000), (Moore et al., 2003)	
Desire Modelling	No Articulation Rank Direct Weight AHP on Attribute Ranges			(Andersson, 2000), (Keeney and Raiffa, 1976)		
Risk	No risk Disagreement as risk Risk on Decsion Risk on fulfillment level					(Gilb and Brodie, 2005), (Andersson, 2000)

Table 4-4: Main references used to define Table 4-2

Characteristic	Value	Troubles						
		Stakeholders Effort			Solution		Stakeholders	
		Time	Availability	Interest	Complex Description	No best solution	Coarse-grain Desire	Linear Desire
Attribute Identifier	Term Term + UseCase Term + Metric Term + UseCase + Metric	(Anderss on, 2000)						
Type of fulfillment	On-Off Terms (symbols) Direct Ratio AHP on fulfillment level	(Anderss on, 2000)			(Andersson, 2000), (Keeney and Raiffa, 1976)	(Anderss on, 2000), (Keeney and Raiffa, 1976)	(Andersson, 2000)	
Expression Time	No Articulation A priori Progressive	(Anderss on, 2000)		(Moore et al., 2003)				
Desire Modelling	No Articulation Rank Direct Weight AHP on Attribute Ranges	(Anderss on, 2000)			(Andersson, 2000), (Keeney and Raiffa, 1976)	(Anderss on, 2000), (Keeney and Raiffa, 1976)		(Moore et al., 2003), (Anderss on, 2000), (Keeney and
Risk	No risk Disagreeme nt as risk Risk on Decsion Risk on fulfillment level	(Gilb and Brodie, 2005), (Moore et al., 2003), (Anderss on, 2000)			(Andersson, 2000), (Keeney and Raiffa, 1976)	(Anderss on, 2000), (Keeney and Raiffa, 1976)		

Software architects should pick a decision-making technique based on the type of troubles that they do not want to run into, or the ones that they just ran into and want to avoid. Since architects often cannot predict the type of troubles they could run into, then the approach needs to be adaptive. In other words, the proposed characterization schema provides a means for tuning the decision-making technique based on the application context (i.e. undesirability and probability of troubles).

For instance, during the architecture design phase, architects have little time to make their decisions because the successive phases of the development process (e.g. detailed design, programming) cannot start otherwise. On the contrary, strict time constraints do not affect architectural review, because this activity is done in parallel to, or after, the project development. Consequently, we envision that architects, differently from reviewers, prefer to adopt techniques that require little time to be enacted (described first column in Table 4-2).

Note that each described relation represents a possible future empirical study aimed at understanding under which circumstances (e.g. type of issue, context characteristics) such a value of the characterization schema provokes (or not) a specific trouble. This is the reason why the proposed characterization schema does not indicate the most suitable technique in a given context (which would also include the paradox explained in Section 4.3) but Table 4-1 and Table 4-2 certainly provide a useful framework to select a decision-making technique according to past or expected troubles.

4.7 Software engineering current practice

We found that several techniques are available for choosing among design alternatives ([Gilb and Brodie, 2005] [Moore et al., 2003] [Kazman et al., 2001] [Al-Naeem et al., 2005] [Chung et al., 1999] [Dabous and Rabhi, 2006] [Svahnberg et al., 2003] [Andrews et al., 2005]), requirements (e.g. [Azar et al., 2007] [Ruhe and Saliu, 2005]), and COTS (e.g. [Wanyama and Far, 2005] [Kontio, 1996] [Bandor, 2006] [Cavanaugh and Polen, April 2002] [Lozano-Tello and Gómez-Pérez 2002] [Ncube and Maiden, 1999]). The reason we focused on these techniques is that in all of them decision-making plays a pivotal role; although they have different objectives, stakeholders, and development process phases, they share the following characteristics:

- They involve several stakeholders with different knowledge, views and responsibilities of the system,
- They deal with competing and conflicting objectives,
- There is a level of uncertainty both in the issue description and, in the associated solutions;

- The decisions made have a high-level of interdependencies.

Table 4-5, Table 4-6, and Table 4-7 identify seventeen decision-making techniques and situate them in our proposed characterization schema. The aims of such tables are:

- **Validating the characterization schema for completeness**: each value of each characteristic has been used by at least one decision-making techniques proposed in the literature and vice versa.
- **Acting as reference**: software engineering practitioners can use it as a reference of decision-making technique state of the art.
- **Exposing limits**: we reveal some of the limits and troubles that the decision-making technique may have.
- **Discovering trends**: we deduced Table 4-8 from the contents of Table 4-5, Table 4-6, and Table 4-7. By analyzing the content of the Table 4-8, we can reason on which value, for each characteristic, is mostly common for the different software engineering application domains (i.e. Architecture design, Release planning, COTS selection).

We noticed that every method can be applied several times and hence be classified as progressive; in this section we classify a method as progressive only if it explicitly suggests an iterative usage for the same method.

The present study neglects most (around 6) of the decision-making techniques which relate to the release planning. The reason of this choice is that those tecniques do not use a multi-attribute approach but a comparative approach (e.g., AHP): they do not decompose the utility in several attributes but they judge the utility of each alternative as a whole. This approach, in contrast with the multi-attribute one, has the advantage of hiding the complex problem of finding and calibrating relations that exist between attributes; however, such a comparative approach becomes unmanageable in case of large amount of attributes.

Table 4-5: The proposed characterization schema applied to eight decision-making techniques that concern software design.

		Decision-making techniques							
	Reference	(Kazman et al., 2001)	(Moore et al., 2003)	(Gilb and Brodie, 2005)	(Al-Naeem et al., 2005)	(Chung et al., 1999)	(Dabous and Rabhi, 2006)	(Svahnberg et al., 2003)	(Andrews et al., 2005)
	Name of the decision-making technique	CBAM1	CBAM2	Impact Estimation	Quality driven	Softgoals	Dabous & Rabhi	Quality driven	A Framework for design tradeoffs
	Type	Design	Design	Design	Design	Design	Design	Design	Design
Attribute Identifier	Term				x	x	x	x	
	Term+ UseCase								
	Term+ Metric	x		x					
	Term+ UseCase + Metric		x						x
Type of fulfillment	On-Off						x		
	Terms (symbols)					x			
	Direct Ratio	x	x	x					x
	AHP on fulfillment level				x			x	
Expression Time	No Articulation					x	x		
	A priori	x		x	x			x	x
	Progressive		x						
Desire Modelling	No Articulation					x	x		
	Rank								
	Direct Weight	x		x					
	AHP on Attribute				x			x	x
	Ranges		x						
Risk	no risk				x	x	x		x
	Disagreement as risk	x	x					x	
	Risk on decision			x					
	Risk on fulfillment level								

Table 4-6: The proposed characterization schema applied to seven decision-making techniques that concern COTS selection.

		Decision-making techniques						
	Reference	(Kontio, 1996)	(Bandor, 2006)	(Cavanaugh and Polen, April 2002)		(Wanyama and Far, 2005)	(Lozano-Tello and Gómez-Pérez 2002)	(Ncube and Maiden, 1999)
	Name of the decision-making technique	OTSO	Quantitative methods	CEP	Iso 9x26	NeMo-CoSe	BAREMO	PORE
	Type	COTS Selection	COTS Selection	COTS Selection	COTS Selection	COTS Selection	COTS Selection	COTS Selection
Attribute Identifier	Term		x	x	x	x	x	
	Term + UseCase	x						
	Term + Metric							
	Term + UseCase + Metric							x
Type of fulfillment	On-Off							x
	Terms (symbols)					x	x	
	Direct Ratio		x		x			
	AHP on fulfillment level	x						
Expression Time	No Articulation							
	A priori	x	x	x	x			
	Progressive					x	x	
Desire Modelling	No Articulation							x
	Rank							
	Direct Weight		x	x	x	x		
	AHP on Attribute	x					x	
	Ranges							
Risk	no risk		x		x	x	x	x
	Disagreement as risk	x						
	Risk on decsion							
	Risk on fulfillment level			x				

Table 4-7: The proposed characterization schema applied to two decision-making techniques that concern release planning.

		Decision-making techniques	
	Reference	(Azar et al., 2007)	(Ruhe and Saliu, 2005)
	Name of the decision-making technique	Value Based	Software release
	Type	Release Planning	Release Planning
Attribute Identifier	Term	x	x
	Term + UseCase		
	Term + Metric		
	Term + UseCase + Metric		
Type of fulfillment	On-Off		
	Terms (symbols)		
	Direct Ratio	x	x
	AHP on fulfillment level		
Expression Time	No Articulation		
	A priori	x	x
	Progressive		
Desire Modelling	No Articulation		
	Rank		.
	Direct Weight	x	x
	AHP on Attribute		
	Ranges		
Risk	no risk		x
	Disagreement as risk		
	Risk on decsion	x	
	Risk on fulfillment level		

Table 4-8: Trends in software engineering decision-making

Characteristic	Value	SE Domain		
		Design	Release Planning	COTS Selection
Attribute Identifier	Term	50	71	100
	Term + UseCase	0	14	0
	Term + Metric	25	0	0
	Term + UseCase + Metric	25	14	0
Type of fulfillment	On-Off	12	14	0
	Terms (symbols)	12	29	0
	Direct Ratio	50	29	100
	AHP on fulfillment level	26	14	0
Expression Time	No Articulation	25	0	0
	A priori	63	57	100
	Progressive	13	29	0
Desire Modelling	No Articulation	25	14	0
	Rank	0	0	0
	Direct Weight	25	57	100
	AHP on Attribute	38	29	0
	Ranges	13	0	0
Risk	no risk	50	71	50
	Disagreement as risk	38	14	0
	Risk on Decsion	13	0	50
	Risk on fulfillment level	0	14	0

4.8 Open problems in multi attribute decision-making

Multi attribute decision-making is the only approach taken in consideration in this study: in fact, it is the only (plus the aforementioned "comparative") approach actually present in literature to select among a finite amount of alternatives (i.e. design solutions). However, such an approach has the following open troubles:

Needs interdependency. The family of multi attribute decision-making techniques is unable to handle the need (i.e. utility, importance) of an attribute in relation with the level of fulfillment of other attributes. For instance, it is impossible to handle the following situation (i.e. needs interdependency): if the cost (an attribute) is higher than a threshold (e.g. 1k€) then the solutions should provide performance (another attribute) lesser than another given threshold (e.g. response time less than 1 sec.)

Decision relationships. We are still far away from addressing the issues of relationships between decisions; we have only started to define them [Kruchten, 2004]. In other words, decision-making techniques focus in solving one problem per time, hence selecting one alternative per time, without taking into account the alternative selected for the problem addressed before.

The concept of optional and required. In the available decision-making techniques no distinction is made between optional attributes and required attributes. In other words, it is impossible to model a need like "a performance less than 3msec is a must, between 3 and 1 is desired, less than 1msec is forbidden". Kano introduced to such differences between needs [Kano, 1993]; concerning an attribute, he identified the kinds of need that he classified as Linear, Attractive, Must be, and Indifferent. For each attribute of the kind Must be, an optimal level is specified for mandatory fulfillment. For instance, the frequency of cathode television is an attribute to consider in the kind "Must be" because there is more dissatisfaction below a specific fulfillment threshold than satisfaction above it. In fact, a frequency less than 100Hz provoke flicker effects and hence it causes user dissatisfaction; vice

versa, the human eyes would not perceive a higher frequency, which is hence not desired. For attributes of the kind Attractive, there is no specified level to fulfill; however, an improvement of the attribute fulfillment implies an improvement of the benefits. For example, from a customer point of view, the lesser is the cost of a COTS, the better it is.

4.9 Suggestions

There is no perfect decision-making technique as there are no silver bullets in software design; however we provide the following suggestions:

Knowing the limits of the technique utilized. Whatever is inaccuracy level of the decision model used, we suggest describing examples to stakeholders where the decision model does not work properly. Those examples will help stakeholders to get confidence with the method.

Iterations for resolving the tradeoff between needed effort and provided accuracy: The level of accuracy of the method should be balanced based on the usage context: too much accuracy requires stakeholders to provide too much effort. The lesser the accuracy the higher is the possibility of select a bad alternative. Iteration should help engineers in finding the right balance between method accuracy and effort required. "Experience told us that is important to be fast" [Moore et al., 2003]; hence, it is important to understand which model property to emphasize and which one to abstract on. Iterations would also allow stakeholders in getting confident with all the obscure aspects they are involved in (e.g. terms, and decisions inter-dependencies).

Get only and all the right people involved: From one side, involving a large amount of stakeholders would require huge effort for achieving an agreement on the alternative to select because they usually have conflicting objectives. From another side, neglecting important stakeholders may imply a bad decision. [Poppendieck and Poppendieck, 2007]

Just in time decision: Delaying decisions to the last responsible moment [Poppendieck and Poppendieck, 2007] has the advantage to allow a better comprehension both of the issue and the solutions.

Interesting methods are already available [Egyed and Wile, 2006] and related benefits are well agreed [Erdogmus and Favaro, 2002].

Valorize decision deadline. Pending decisions become inevitably more and more concrete in the mind of designers. Hence, late decisions imply the additional cost to change, in case they differ from the expected ones. For example, in case the architect does not decide about the middleware to adopt than developers may start to code for the ususal middleware. Afterwards, in case the architects decide for another middleware, whatever the potential utility of it might be, such decisions imply huge re-work effort.

Do not wait for an ideal solution or a definitely clear issue description. Sometimes it is not possible to wait until the requirements or the solutions characteristics become clear. Software architecture design has to deal with unstable and vague requirements, and decisions cannot wait until the overall situation is perfectly clear; Philippe Kruchten describes the software architecture decisions as "a long and rapid succession of suboptimal design decisions taken partly in the dark" [Kruchten, 1999]. Recently, Gary Booch has been proposing the same idea by remarking that actual used architectural pattern are not perfect but they fit in a good way "for resolve the forces that impose upon a given system" [Booch, 2006].

Scenario over terms. Stakeholders may have different interpretation about the issue to solve; hence they may relate different meanings from the same word. [Moore et al., 2003].

Number over terms. Non-numeric estimates of impact are difficult to analyze and improve upon. A design idea described as 'excellent' could actually be worse than another merely described as 'good.' "A bad numeric estimate, and its definition, can still be systematically criticized and improved. In fact, a random number is a better starting estimate than flowery, descriptive terms." [Gilb and Brodie, 2005].

4.10 Conclusion

Our key idea was to organize in a useful way, namely by a characterization schema, and expose in what extent each decision-making technique is prone to specific troubles. In this way, the level of proneness of a specific technique to specific troubles becomes a quality

attribute of the decision-making technique. Hence, the characteristics of the usage context (e.g. undesirability and probability of specific troubles) affect the level of goodness of the decision-making techniques.

By reviewing already proposed decision-making techniques we realized that selecting a decision-making technique is difficult because any decision-making technique cannot be defined as better than another; this is due to the following two main reasons:

A trouble can be provoked by using two decision-making techniques even with different levels of proneness (see above point 1.2) to such a trouble. Consequently, a more accurate model does not always pay off: it depends on the issue to address and the usage context.

No decision-making technique is more (or less) prone, than other technique, to all the troubles taken into account in this study. On the contrary, a decision-making technique (e.g. *A*) that is less prone (i.e. better) than another (e.g. *B*) for a specific set of troubles (e.g. *i*, and *i+1*), also is more prone to some other troubles (e.g. *A* worse than *B* in relation to *i+2*). Consequently, for resolving architectural tradeoffs, because the quality of a technique depends on which troubles decision-makers want to avoid, there is no "best" decision-making technique to look for; however, there are techniques that are more prone than others to specific troubles.

You should pick a decision-making technique based on the type of troubles that you do not want to run into, or the ones that you just ran into and want to avoid. Since you often cannot predict the type of troubles you could run into, then the approach needs to be adaptive. In other words, the proposed characterization schema provides a means for selecting/tuning the decision-making technique based on the application context (i.e. undesirability and probability of troubles).

The provided characterization schema can be seen as the first contribution on *meta-decision-making*, which is the issue of deciding how to decide.

Furthermore, we situated - in the proposed characterization schema - seventeen different decision-making techniques, as the literature already proposed in the domains of architecture design, COTS selection, and release planning. Such localization both validates the completeness

of the proposed characterization schema, and provides a useful reference for analyzing the state of the art.

5. Assessing the Importance of Documenting Design Decision Rationale

Individual and team decision-making have crucial influence on the level of success of every software project. Even though several studies were already conducted, which concerned design decision rationale documentation approaches, a few of them focused on performances and evaluated them in laboratory. This chapter proposes a technique to document design decision rationale, and evaluates experimentally the impact such a technique has on effectiveness and efficiency of individual/team decision-making in presence of requirement changes. The study was conducted as a controlled experiment. Fifty post-graduate Master students performed in the role of experiment subjects in a controlled environment. Documented design decisions regarding the Ambient Intelligence paradigm constituted the experiment objects. Main results of the experiment show that, for both individual and team-based decision-making, effectiveness significantly improves, while efficiency remains unaltered, when decision-makers are allowed to use, rather not use, the proposed design rationale documentation technique.

5.1 Introduction

Individual and team decision-making have crucial influence on the level of success of any software project. Anyway, up to now, to the best of our knowledge, few empirical studies evaluated the utility of Design Decision Rationale Documentation (DDRD).

Several studies already have taken approaches and techniques to this end in consideration and have argued about their benefits, but only one of them [Bratthall et al., 2000] has focused on performance and has been evaluated it in laboratory.

The contribution of this chapter is twofold: (i) it briefs, for a larger audience, on the "Decision Goals and Alternatives" (DGA) DDRD technique, which was presented in form of a Technical Report [Falessi and Becker, 2006] so far, and (ii) experimentally evaluates that technique with respect to the current practice of not documenting design

rationale at all. The study was conducted as a controlled experiment with post-graduate Master students in the University of Rome "Tor Vergata".

The remainder of this chapter is structured as follows: Section 5.2 presents motivation, view, goal and hypotheses of the conducted study. Section 5.3 considers related works. Section 5.4 introduces to DGA DDRD technique. Section 5.5 addresses experiment planning and operation issues. Sections 5.6 shows experiment results and data analysis, and Section 5.7 discusses results. Section 5.8 argues about threats to validity. We conclude this chapter in Section 5.9 with some final remarks and prospective works.

5.2 Study motivation and research hypotheses

The growing interest in software engineering decision support field [Perry et al., 1994] reveals the crucial influence of decision-making on the level of success of any software project.

Because separately good decisions might be competitive, inconsistent or also in contradiction when viewed as a whole, software research still assigns great importance to individual and team decision-making, and agility in software process, as demonstrated by the relevant number of related studies [Bellotti and Bly, 1996] [Boehm and Turner, 2005] [Highsmith and Cockburn, 2001] [Kraut and Streeter, 1995] [Neale et al., 2004] [Martin, 2003]. Concerning those issues, our conjecture is that DDRD is useful for individual and team decision-making in case of changes in requirements.

Formally, according to the GQM template [Basili et al., 1994], the goal of the presented study is to analyze the DGA DDRD technique for the purpose of evaluation with respect to effectiveness and efficiency of individual-decision-making and team-decision-making in case of changes in requirements from the point of view of the researcher in the context of post-graduate Master students of software engineering. From the motivations and goal above, the following research null hypotheses (resp. alternative hypotheses) follow for the presented study. After changes in requirements, when using or respectively not using DGA documentation of taken design decisions, there is no significant

difference (H_{0--}) (resp. significant difference, H_{1--}) for individuals (H_{-I-}) (resp. teams, H_{-T-}), in the amount of time needed for (hence efficiency, H_{--Ey}) (resp., in the correctness, hence effectiveness of, H_{--Es}) decision-making. Consequently, we have four null (resp. alternative) hypotheses, which we denote by H_{0IEy}, H_{0IEs}, H_{0TEy}, and H_{0TEs}, respectively (resp. H_{1IEy}, H_{1IEs}, H_{1TEy}, H_{1TEs}).

5.3 Related work

5.3.1 Empirical evaluation of Design Decision Rationale

Two main studies empirically evaluated the importance of DDRD. Brathall et al. [2000] presented a controlled experiment to evaluate the importance of DDRD when predicting change impact on software architecture evolution. Results show that DDRD clearly improves effectiveness and efficiency. However, such improvement was only partially statistically significant and authors explicitly called for further investigations. Between the present study and the one proposed by Brathall et al. there are similarities concerning dependent variables and methods for data analysis, and dissimilarities concerning many points, including requirement-change causes, types of subjects, type of decision to manage, and type of DDRD. Karsenty [1996] proposed an empirical evaluation concerning the utility of design rationale documents in maintenance of a nine months old software project. Between the present study and the one proposed by Karsenty, there are similarities concerning the purpose, and dissimilarities concerning many points, including requirement-change causes, types of subjects, objects, treatments, dependent variables, data collection mechanisms, and methods for data analysis.

5.3.2 Cooperation

Wu, Graham, and Smith [2003] used interviews, shadowing, and communication-event-logging as data collection methods. Main results show that, concerning designers: 1) they communicate and collaborate by a wide variety of means; 2) they prefer general-purpose tools rather

than domain-specific tools; 3) they change frequently their meeting place.

Seaman and Basili [1997] studied the influence of organizational and process characteristics on the effort spent in collaboration. They used real-time observations and structured interviews. Their main result was that several organizational factors significantly affect communication effort (time spent).

Bellotti and Bly [1996] emphasized on the importance both of local mobility in collaborative software design, and face-to-face style of communication.

Kraut and Streeter [1995] utilized questionnaires and interviews for observing inter-team coordination practices. The major result was that developers cited discussion as their preferred communication means.

In the middle of 90's, Perry, Staudenmayer, and Votta [1994] conducted two experiments revealing that a large percentage of the software process cycle was devoted to organizational concerns.

Tang [1991] used videotapes to observe activities enacted by small teams in controlled environments. His major result was that the process of creating and using drawings conveys much information that is not captured within the drawing itself.

Concerning computer supported cooperative work, several studies were conducted focused on evaluating benefits on geographically distributed development [Borghoff and Schlichter, 2000]. Anyway, a very recent study [Neale et al., 2004] clearly reveals what other studies [Bellotti and Bly, 1996] [Borghoff and Schlichter, 2000] [Bratthall et al., 2000] [Herbsleb and Grinter, 1999] [Neale et al., 2004] [Ruhe, 2003] state implicitly: measuring collaboration is hard.

5.3.3 Agility

Nowadays, agile software development has become extraordinarily fashionable. Agile methodologies include tools, processes, and approaches; they aim at helping software organizations in reacting to requirement changes, using the principles exposed in the Agile Manifesto. Major agile methodologies are: Extreme Programming, Scrum, Lean Development, Crystal and Context Driven Testing

[Highsmith and Cockburn, 2001] [Martin, 2003]. However, the implementation of agile processes in traditional development organizations entails several management challenges [Boehm and Turner, 2005].

5.4 The DDRD DGA Technique

Following the definition of design rationale provided by Lee [1997] the information that we want to document are the reasons why a decision has been taken. The SEI presents its Cost Benefit Analysis Method (CBAM) method [Kazman et al., 2001] as a rational decision-making process for software architectural decisions, which is able to give stakeholders help in the elicitation of costs and benefits. The CBAM is based on attributes such as the importance of each objective for the project, the alternative decisions available and, for each alternative, to what extent that alternative fulfills those objectives. The basic principle is that each objective has its own level of importance and each alternative decision has, for each objective, its own level of fulfillment. The level of benefit related to an alternative is measured by summing the products of the level of fulfillments of each objective and the level of importance of that objective.

During the development of an Ambient Intelligence application, while trying to apply CBAM, the needs of improving collaboration among design decision makers arose, refining the grain of requirements traceability, and optimizing the usage of new technologies. Therefore, we eventually came to instantiate DDRD in a specific documentation technique, the abovementioned DGA, which is driven by the decision goals and available design alternatives.

In the DGA technique, DDRD consists in documenting the attributes of CBAM. According to DGA, whatever the software context might be, design decisions depend on basic decision goals and inter-decision relationships, as shown in Table 5-1. In our experience, the set in Table 5-1 is complete, i.e. its elements are sufficient to specify the rationale of any software design decision. In the remaining, we call them Decision Goals.

Table 5-1 Entities influencing the rationale of design decisions ("Decision Goals")

Functional requirements
Non-functional requirements (quality attributes and constraints)
Business goals
Decision relationships

DGA not only aims to document already made decisions, but also to help decision-makers in making their further decisions. In DGA, the entity Decision is refined into two sub-concepts: Decision Type (DT) and Decision Alternative (DA). DT addresses the problem the decision should solve (e.g. What is the programming environment to use?) DA represents an available option (e.g. .NET). Two main insights drove the structure of the DGA technique: the importance of a goal is a DT attribute, while the goal fulfillment is a DA attribute. As a result of this clear separation of concerns, the maintainability of DDRD increased by avoiding document erosion and improving efficiency of document maintenance. For instance, a change in technology would affect DA only (level of fulfillment), while a change in requirements would affect DT only (level of importance). In fact, it is this separation of concerns that distinguishes DGA from other DDRD techniques.

According to DGA, in order to produce documented decisions, DDRD consists of two stages: (i) understand what to document, and (ii) enact the documentation.

The activities of the first stage consist in refining the project objectives and constraints, and comprehending which decision relationships are appropriate for the project. The refinement of the decision goals in Table 5-1 in sub-goals depends on the specific usage context. In fact, we provide DGA users with much more than Table 5-1: a framework for decomposing higher-level goals that prevents lacunae, and avoids misunderstandings.

The last stage is arranged in tasks: there is an instance of such a task for each design decision to make. Decisions makers can work in parallel (hence, tasks can be enacted in parallel). A task is arranged in three sequential blocks of activities, aimed to evaluate the "score" to give to relevant attributes of the current decision, for instance the

priority of each objective in the designer view. The first block includes: (i) describing the current decision by providing information for the current DT, (ii) giving a score to each objective, to express the objective's importance for the current DT, and explaining motivations The second block includes: (i) describing each alternative of the current DT by providing more specific information, and then, (ii) for each objective, scoring to what extent the current alternative fulfills this objective, and (iii) for each relationship of the current alternative with alternatives of other DT(s), scoring to what extent the current alternative depends on each of the related DT(s), in the designer view. The last block selects in case the best alternative decision for the current DT and documents the alternative selected for the current decision. For further details about DGA we refer to [Falessi and Becker, 2006].

5.4.1 Expected effects on collaboration

Improvement of collaboration among designers can be achieved by:

- Making team members aware of newly made design decision occurrences. In fact, this helps to detect, respectively avoid, conflicts between design decisions. The description of the expected relationships between decision alternatives and other decisions (see above for DGA activity concerned with scoring relationships between alternatives of different DT(s)) should provide helpful hints to this end.
- Allowing team members to share the characteristics of design decision alternatives. In fact, this exploits the different points of views of the various members, and consequently helps decision makers to consider the different view points. The quantification to what extent a decision alternative fulfills the objectives of a decision type (see above for DGA activities concerned with scoring fulfillment of objectives) should foster this. Furthermore, DGA intends to improve the collaboration between designers and project manager by:
- Preventing designers' misjudgment of the goals' importance. Quantifying to what extent objectives are

important for DT (see above for DGA activities concerned with prioritizing objectives) should allow managers to detect and resolve misinterpretations quickly.

- Identifying the violated requirements that cannot be met based on the already made design decisions. Quantifying to what extent decision alternatives fulfill the objectives (see again objective prioritization) should help to achieve this goal. Be aware that this additionally improves communication between product managers and customers, as the impact of a request is made explicit and thus foster the argumentation.

5.4.2 Expected effects on agility

Changes in the requirements necessitate changes in the objectives. Technological changes, in their turn, extend (usually) the set of available alternatives and help designers in recognizing better alternatives (if any). As DGA quantifies to what extent a decision alternative fulfills the objectives of a decision type (see DGA activity 2.4 above), it is expected to support agility issue. In fact, in case of changing requirements or technology, designers should be able to identify affected decisions (and artifacts) quickly and to reconsider them efficiently

5.5 Experiment planning and operation

5.5.1 Experiment definition and setting

According to the study hypotheses and goal (see previous Section 5.2), we conducted a controlled experiment at the University of Rome "Tor Vergata", with fifty post-graduate local Master students performing in the role of experiment subjects. Design decisions regarding an AmI [Remagnino et al., 2005] prototype developed at Fraunhofer IESE constituted the experiment objects. The first author stayed six months in Kaiserslautern for studying the application domain and reasoning about feasibility and other characteristics of the presented study. The gained experience helped the experimenters (i.e. the experiment research team members) to carefully replicate the original context in the experimental

environment of this study, which improves the external validity of the study.

According to Section 0, the evaluation of collaboration issues implies several intrinsic threats, which affect construct validity and internal validity [Neale et al., 2004]. According to Zelkowitz and Wallace [1998], developing the study in laboratory would mitigate the impact of construct and internal threats. Additionally, both the usage of quite real objects, and the experience that experimenters matured in field, should mitigate the presence of external threats, which might derive from using a laboratory for conducting the experiment. Our consequent design decision was to conduct a controlled experiment in a synthetic environment.

The context of the current study is off-line (an academic environment) rather than in-line, based on students rather than professionals, using domain-specific and goal-specific quite real objects (as synthesized from real ones) rather than generic or toy-like objects.

In order to replicate the context of real world decision-making, the experimenters defined a synthetic software project, able to show and emphasize those aspects, which are in focus for the presented study. That project regards a hospital management system, and AmI issues (e.g. resource constraints, heterogeneous sensors, etc.) characterize it. Moreover, it is supposed that: the software system is at the start point of its second iteration (of some iterative development process, e.g. RUP [Kruchten, 2003]); in the mean time, system requirements did change, and designers, who had taken design decisions during the previous iteration, moved away and are no more available for giving explanation to current designers. Concerning the requirements change, experimenters applied change-causes, which usually affect software requirements in real projects, like: 1) Variations in the industrial strategic partnerships; 2) Changes in functional and non-functional requirements, resulting from the customer experience in using the previous version of the product; 3) Technology advances.

In order to replicate the real world context for software teams, the experimenters designed five different roles for the participants, and planned to compose teams by using those roles, one team-member per role, and five people per team. They designated each role to manage two

different design decisions: the one with DGA-documentation and the other one without DDRD. They planned to characterize each experiment subject, based on the participant's personal experience and preferences, and assigned a subject to perform in one role. Issues regarding the mapping from subjects to roles are further described in Section 5.5.3.

The experimenters specified ten different design decisions to use as experiment objects and assigned each of them a unique integer number ranging from 1 to 10 as an ID. Subjects performing in the same roles were specified to fall in different teams and manage the same couple of decisions. Vice versa, all the subjects performing in different roles were assigned to have two different decisions to manage. Concerning the decisions assigned per subject, one was without design rationale documentation, i.e., only the result of the decision was reported (not its why): let us denote such a level with "non–DGA-documented". The second decision was documented with DGA, i.e. the documentation additionally included the decision rationale.

Due to the nature of the experiment object – the AmI prototype – and planned requirement changes, decision making concerned two types of components: the Central Computation Node (CCN) and the Personal Digital Assistant (PDA) component. Driven by this fact, the experimenters partitioned experiment design decisions in two classes: five decisions concerning CCN and PDA each. Of course, both classes were arranged in two versions: the one DGA-documented and the one non-DGA-documented. In particular, based on the experiment arrangement, decisions with ID(s) from 1 up to 5 fell in the first class and those with ID(s) from 6 up to 10 fell in the second class. Additionally, decisions belonging to the same class were interrelated with each other. For instance, the decision regarding the authentication mechanism to be used in a specific system component is strictly related with the decision concerning the physical capabilities of the component.

Based on the number of participants (50) and the team size (5), ten teams were organized randomly.

The experiment material was arranged on paper supports, and it regarded each decision to make by describing the requirements of both the previous (first) iteration and the current one (second iteration) of the project.

The experiment was balanced and the assignment of treatments to subjects was randomized. A decision was assigned to the same number of roles, subjects and teams. Moreover, each decision had the same number of instances (and treatments, DGA/non-DGA documented decisions). Each team received both classes of decisions, those concerning CCN and PDA, respectively. Five out of ten teams had DGA (resp. non-DGA) -documented decisions to manage that belonged to one class. The remaining five teams had DGA-documented decisions to manage that belonged to the other class.

Two main phases characterize the experiment. Both of them aimed at evaluating efficiency, effectiveness, and perceived utility of using/not using DGA DDRD in individual/team decision making in case of requirements change.

During the first phase of the experiment, each subject received two decisions, depending on the subject's role, and managed those decisions without cooperating with other participants. Those decisions belonged to different classes. Depending on the subject's team, one out of two decisions was DGA-documented. 50% of subjects had DGA-documented decisions to make first.

During the second phase, subjects convened with their teams and all together, discussed, for acceptance, each decision made during the first experiment phase. In fact, each decision was reconsidered. Each team member contributed to improve and enhance compatibility among decisions by valorizing at team level the subjective understandings about the available alternatives.

In each phase, the experiment material guided subjects to manage decisions assigned to them in a specified order. In the initial phase, each subject handled first the decision with the minimum ID and then the remaining decision. During the second phase, team members considered and finalized a half of their decisions (ID 1-5) first. Afterwards, they passed to evaluate the remaining decisions (ID 6-10). In our case, which included relationships between decisions, the experiment setting allowed us to evaluate decision making per team by using five objects, each under two levels for the factor: DGA/non-DGA documented decisions.

Concerning the experiment validity threats, some considerations should be made at this point. In order to keep the impact of subjectivity in control, we designed a balanced experiment. Moreover, we applied both treatments to each subject. Furthermore, in order to keep in control the impact of learning effect on the experiment results, the treatments to use first were as much as to use last. Finally, because learning effects play a predominant role in decision-making, we discarded the idea of using paired design for improving validity of the experiment data.

Table 2 shows the experiment setting, i.e., the structure of treatments, objects, subjects, and teams. Each row describes a team. An item in the first column shows the corresponding list of DGA-documented decisions (default decisions were non-DGA-documented). An item in the second column indicates the ID of the corresponding team. The rest of Table 5-2 is organized in five batches, each including tree columns, which represent the ID(s) of the experiment subjects, the ID(s) of decisions assigned to, and role played by each of those subjects, ordered from left to right each. For instance, the first row in Table 5-2 shows that subject with ID 43 played the role AS ("Software Architecture & Service Discovery"), addressed decisions 1 and 6, and belonged to team A, which included subjects with ID(s) 12, 50, 2, and 10.

Three further people participated in the experiment: one playing the role of the Application System General Manager, the other ones performing in the roles of Experiment Designer and Observer, respectively. The General Manager answered questions concerning the business strategy of the software organization. The Experiment Designer provided subjects with face-to-face technical explanation related to the usage of experimental objects. The Observer enforced subjects to respect rules, in order to have collected data acceptable for filtering and analysis.

5.5.2 Training

In order to enable the subjects to attend our DDRD experiment with enough confidence, we trained all of them through three plenary training sessions for a total of eleven hours. The first session of two hours was theoretic. The experimenters explained DDRD and AmI related issues. The second session took four hours. Here the experimenters performed

in the role of decision makers and discussed five exemplary design decisions. The third session lasted five hours and was a trial of the experiment discussed in the presented work. For such a training session, all the characteristics were similar to those we would have utilized at experiment conduction time, less the application domain (a house AmI application was used for training) and related decisions. The experimenters checked that every subject was trained in every session sufficiently.

Doc.	Team	Sub.	Dec.	Role	Sub.	Dec.	Role	Sub.	Dec.	Role	Sub.	Dec.	Role	Sub.	Dec.	Role
1,2,3,4,5	A	43	1, 6	AS	12	2, 7	CO	50	3, 8	DS	2	4, 9	HW	10	5, 10	IN
1,2,3,4,5	B	38	1, 6	AS	18	2, 7	CO	39	3, 8	DS	13	4, 9	HW	35	5, 10	IN
1,2,3,4,5	C	22	1, 6	AS	23	2, 7	CO	33	3, 8	DS	32	4, 9	HW	16	5, 10	IN
1,2,3,4,5	D	34	1, 6	AS	28	2, 7	CO	26	3, 8	DS	45	4, 9	HW	21	5, 10	IN
1,2,3,4,5	E	47	1, 6	AS	29	2, 7	CO	25	3, 8	DS	37	4, 9	HW	5	5, 10	IN
6,7,8,9,10	F	42	1, 6	AS	40	2, 7	CO	19	3, 8	DS	24	4, 9	HW	36	5, 10	IN
6,7,8,9,10	G	7	1, 6	AS	41	2, 7	CO	15	3, 8	DS	4	4, 9	HW	27	5, 10	IN
6,7,8,9,10	H	30	1, 6	AS	44	2, 7	CO	14	3, 8	DS	3	4, 9	HW	6	5, 10	IN
6,7,8,9,10	I	31	1, 6	AS	46	2, 7	CO	9	3, 8	DS	11	4, 9	HW	17	5, 10	IN
6,7,8,9,10	L	48	1, 6	AS	49	2, 7	CO	8	3, 8	DS	1	4, 9	HW	20	5, 10	IN

Table 5-2. Setting objects, treatments, subjects, and teams

5.5.3 Subjects

Fifty attendees of the Experimental Software Engineering post-graduate course in their second and last year of Master Degree, participated in our work as experiment subjects, performing in the role of decision makers. While most of those subjects had already had some experiences at software companies, only few can be considered as software professionals. According to the classification scheme proposed by Höst et al. [2005] experience and incentive of subjects can be classified respectively as “Graduate student with less than 3 months recent industrial experience” (E2) and “Artificial project”(I2).

In order to approximate the structure of a work-team in the AmI domain as much as possible, we modeled five different roles for decision making, one for each of the following areas: (i) HW – Hardware, (ii) CO – Communication, (iii) AS – Software Architecture & Services Discovery, (iv) IN – Inference, and (v) DS – Data Storage. For each of these roles specific knowledge and experience were requested. Systems were viewed from a certain perspective, and

responsibilities were in place for certain types of decisions. Concerning our experiment, subjects expressed their preference for each role, according to their previous experience and level of confidence with the responsibilities of a role, well in advance of the last training session. Afterwards, subjects were mandatory assigned to those roles, which maximized the total of the expressed preferences. Hence, we split those fifty subjects into ten teams, each including five subjects, one per role.

5.5.4 Objects and materials

Concerning controlled experiments conducted in synthetic environments, threats to external validity are strong enough to play the role of the Achilles' heel due to the usage of a context, which is quite different from real ones. For this reason we invested significant effort, during experiment planning and design, in carefully synthesizing the objects we already had encountered in former AmI software projects. Additionally, in order to keep experiment decisions as close as possible to real software design decisions, and provide subjects with real requirements as well as real decisions to make, we utilized the Amigo project documentation as source of inspiration.

Figure 5-1 and Figure 5-2 show the forms that decision makers had to fill in during the first phase and the second phase of the experiment, respectively.

Subject ID:	First Phase Form			
	Initial Time	Decision Description	Final Time	Useful? (0≡NO / 1≡YES)

Figure 5-1 First experiment phase form.

Team ID:	Second Phase Form							
	Initial Time	DS	IN	CO	AS	HW	Final Time	Useful? (0≡NO / 1≡YES)

Figure 5-2. Second experiment phase form.

5.5.5 Factor and parameters

The type of design documentation was the experiment factor. As already mentioned, we used two levels for this independent variable: "DGA-documented", and "non-DGA-documented", respectively. We controlled at a constant level the remaining independent variables, like experience of subjects, experiment materials, environment, and complexity of the experiment object.

5.5.6 Dependent variables

As a general note to this section, we want to remark that we analyzed individual and team decision-making by using both quantitative and qualitative data and quantitative analysis methods. Before we proceed with the performance evaluation of DGA-documentation in respect to individual and team decision-making and reaction of designers to requirements change, let us consider efficiency and effectiveness in some details.

IEEE defines efficiency as "the degree to which a system or component performs its designated functions with minimum consumption of resources" [IEEE, 1990]. Our experiment subjects were

allowed to use as much time as they required. Moreover, there were two decisions to make per subject. Hence, we assumed the inverse of the decision time as the punctual estimator to use for the efficiency of a decision. We measured efficiency quantitatively. Subjects used a predefined structured document to record, in real-time, the "Initial time" when they started to address a decision and the "Final time" when they completed the decision (see Figure 5-1 and Figure 5-2). During the first and second phase of the experiment, the Experiment Observer checked in an unobtrusive way the correctness of data recorded by subjects. The third training section had offered us the possibility to test and improve the data collection mechanism that we finally used in the experiment. The subsequent analysis of data revealed that subjects heavily round off data when they have to record directly the amount of time they needed to make a decision. Consequently, subjects had to track their used time with two time registrations for each made decision in the presented experiment. The time spent for each decision then was computed as the difference between those time stamps.

SEI defines Effectiveness as "the degree to which a system's features and capabilities meet the user's needs". Based on such a definition, we measured effectiveness by the amount of "correct" alternatives selected in decision-making. This inevitably raises the question about the "correct" alternative for a decision: Is it the mode or the expected one? From a statistical point of view, the more an alternative is selected, the more it is correct. From an industrial point of view, the correct decision is the one that maximizes the utility. However, in our case, choosing the most useful alternative (industrial view) would depend on our subjectivity. In conclusion, we do not have clarified yet, which of the previous metrics is more appropriate for measuring the correctness of a decision in the context of the presented study. Fortunately, for our case, based on decisions that our experiment subjects made, we observed that, for every experimental object, the modal alternative coincides with the alternative that maximizes the utility, at least to our judgment. To evaluate effectiveness, we utilized qualitative and quantitative data analysis methods. In fact, for each made decision, experiment participants (individuals and teams, respectively) qualitatively described each their decision by filling in the field "Decision Description" of their form during the first phase (see Figure 5-1), and the second phase (see Figure 5-2), respectively. Since

decision makers had been trained well, they described their decisions at a uniform abstraction level, which on one side avoided the risk of having vague or deeply detailed descriptions (i.e. incomprehensible and incomparable reports), on the other side helped experimenters to synthesize on, and assign the same ID to semantically equivalent decisions.

For each decision to make, the experiment decision makers (i.e. individuals or teams) provided Subjective quantitative measures (Yes or No) of the actual (resp. expected) Utility (SU) of the DGA-documentation given (resp. non-DGA-documentation given) to them. In other words, for each subject and decision, in case of a DGA-documented decision, SU is the answer to the question "Did DGA result significantly useful to you while making this decision?" Otherwise SU is the answer to the question "In your opinion, would DGA help you to make this decision significantly?" Decision-makers recorded their SU answers by filling in a predefined field of the given form (see the last column of both Figure 5-1 and Figure 5-2). Such an additional measure of the utility allowed us to triangulate results that the experiment had given for the main utility measures: effectiveness and efficiency. According to Seaman [1999], that triangulation of empirical results helped us to observe the experiment outcome from several views, hence improved results validity.

As we utilized paper-based materials for collecting the experiment data, data entry was done manually by the experimenters and checked several times afterwards.

5.6 Experiment results and data analysis

5.6.1 Data set reduction

We removed individual decision-making data of eight subjects and team decision-making data of one team from the data set to be analyzed, because those participants had round off all their data. Keeping that data would decrease the results validity.

5.6.2 Descriptive statistics

Concerning individual decision-making in case of requirements changes, Table 5-3 shows the decision time (in minutes), the percentage of correct decisions made, and the perceived utility.

Concerning team decision-making in case of requirements changes,

Table 5-4 shows decision time, percentage of correct made decisions, and perceived utility. Note that herein a team decision results from five singular decisions, one per team member, which all members approved.

Figure 5-3 plots individual decision-making results regarding efficiency in case of requirements changes. Let us note that, due to data reduction, the experiment data are not completely balanced (decisions were made not by the same number of subjects). As a consequence, for instance, means showed in Figure 5-3 might differ from the ones obtainable from Table 5-3. Figure 5-4 shows team decision-making data results regarding efficiency. Figure 5-5 shows individual decision-making data results regarding effectiveness in case of requirements changes. Figure 5-6 shows team decision-making data results regarding effectiveness. Figure 5-7 shows individual data-results regarding the perceived utility of decisions made in case of requirements changes.

As all teams perceived DGA-documentation for each of the decisions assigned to them as useful, we do not show plots concerning perceived utility for team decision-making.

Table 5-3 Average data for individual decision-making.

	Average	With Doc.	Without Doc.
Time Required (min.)	10	11	10
Correct Decisions (%)	75	86	64
Perceived Utility (%)	77	81	74

Table 5-4 Average data regarding collaboration.

	Average	With Doc.	Without Doc.
Time Required (min.)	18	17	19
Correct Decisions (%)	39	67	11
Perceived Utility (%)	100	100	100

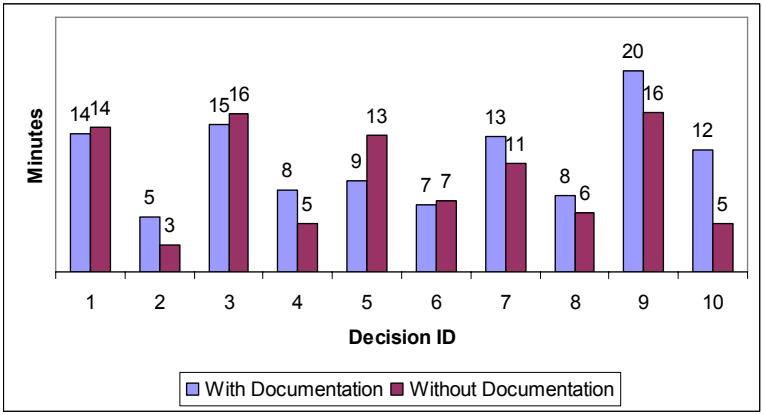

Figure 5-3. Efficiency in individual decision-making.

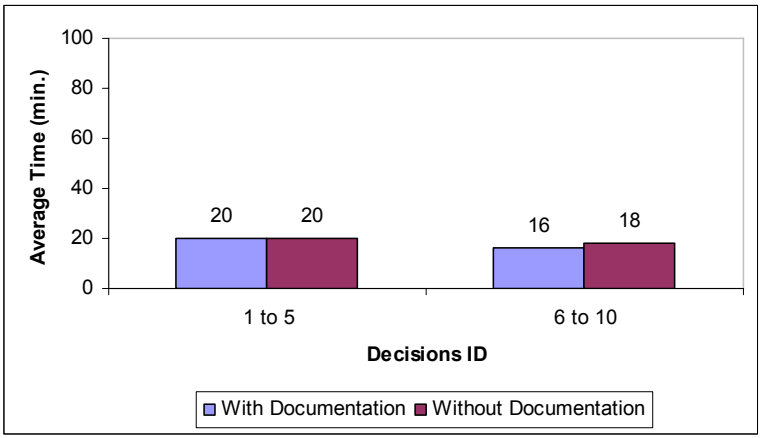

Figure 5-4. Efficiency in collaboration.

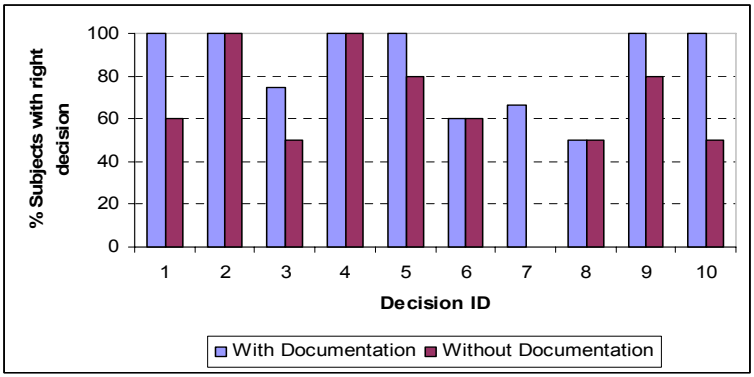

Figure 5-5 Effectiveness in individual decision-making.

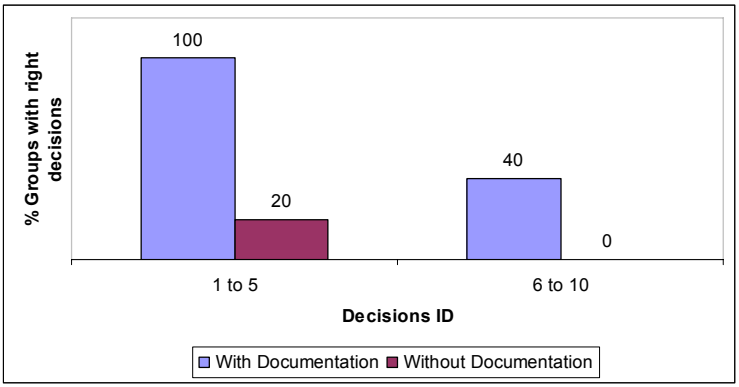

Figure 5-6 Effectiveness in collaboration.

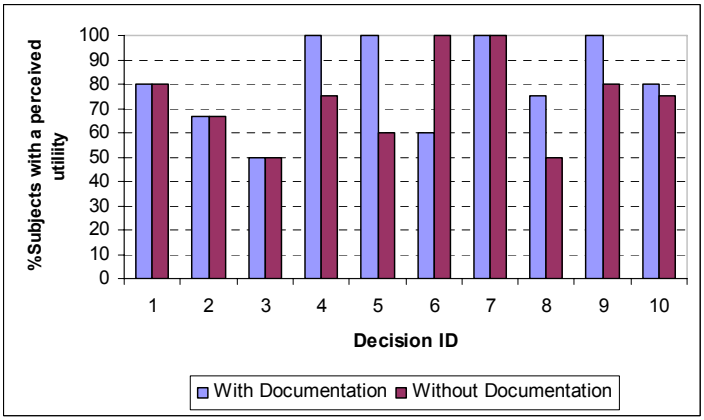

Figure 5-7 Perceived utility in individual decision-making.

5.6.3 Hypotheses testing

5.6.3.1 H_{0IEy}: DGA documentation does not affect efficiency in individual decision-making

In order to test hypothesis H_{0IEy}, we compare two samples for decision-making after requirements change. Those samples concern decision time for formerly DGA-documented decisions and non-DGA-documented decisions, respectively. For the normality tests, which we applied to

both the given data sets, the Shapiro-Wilks test provided, as P-values, 0.05532 for decision times of DGA-documented decisions, and 0.00003 for decision times of non-DGA-documented decisions. Because the latter is less than 0.01, we can reject the idea that such a distribution comes from a normal distribution with the 99% confidence level. Subsequently, for those samples of data, the Mann-Whitney test provides 0.22787 as P-value. Because such P-value is greater than 0.05, we can assert that there is not a statistically significant difference between the medians at the 95.0% confidence level. Hence, we cannot reject the null hypothesis H_{0IEy}.

5.6.3.2 H_{0IEs}: DGA documentation does not affect effectiveness in individual decision-making

In order to test hypothesis H_{0IEs}, we compare two samples for correctness of decisions made after requirements change by using DGA-documented decisions and non-DGA-documented decisions, respectively. Again, the Shapiro-Wilks test provided the lowest P-values (0.0000 in both cases) for the normality of those data sets. Since those P-values are both less than 0.01, we can reject the idea, that any of those data samples comes from a normal distribution with the 99% confidence level. Moreover, for those data samples, the Mann-Whitney test provides 0.024558 as P-value. Because such P-value is less than 0.05, we can assert, that there is a statistically significant difference between the medians at the 95.0% confidence level. Hence, we can reject the null hypothesis H_{0IEs}.

5.6.3.3 H_{0TEy}: DGA documentation does not affect efficiency in team decision-making

In order to test hypothesis H_{0TEy}, we compare two samples of decision time for team decision-making, when using formerly DGA-documented and non-DGA-documented decisions, respectively. Again, the Shapiro-Wilks test provided the lowest P-values (0.811 and 0.109, respectively) for the normality of those data samples. Because both of these P-values are greater than 0.10, we cannot reject the idea that both distributions come from normal distributions with 90% or higher confidence. The P-value given by the F-Test is 0.849. Since such P-value is greater than 0.05, we can assert, that there is not a statistically significant difference

between the samples standard deviations at the 95.0% confidence level. The P-value provided by the T-Test is 0.442307. Because such P-value is greater than 0.05, we can assert, that there is not a statistically significant difference between the means of the two samples at the 95.0% confidence level. Hence, we cannot reject the null hypothesis H_{0TEy}.

5.6.3.4 H_{0TEs}: DGA documentation does not affect effectiveness in team decision-making

In order to test hypothesis H_{0TEs}, we compare two samples regarding team decisions correctness, when using formerly DGA-documented and non-DGA-documented decisions, respectively. For both those data sets, the Shapiro-Wilks test provided the lowest P-values, 0.00024 and 0.00000, for DGA-documented and non-DGA-documented decisions, respectively. Because both of these P-values are less than 0.01, we can reject the idea that both distributions come from a normal distribution with the 99% confidence level. The Mann-Whitney test provided 0.021610 as the P-value for those data sets. Such P-value is less than 0.05, which asserts, that there is a statistically significant difference between the medians at the 95.0% confidence level. Hence, we can reject the null hypothesis H_{0TEs}.

5.7 Result discussion

5.7.1 Individual decision-making

Let us discuss now the impact of DGA documentation on the individual decision-making when requirements do change.

5.7.1.1 Efficiency

Based on results presented by Table 5-3, Figure 5-3, and Section 5.6.3.1, we can argue that the usage of DGA-documentation for design decisions does not significantly affect the decision time, when requirements do change.

5.7.1.2 Effectiveness

Based on results presented by Table 5-3, Figure 5-5, and Section 5.6.3.2, we can argue that the usage of DGA documentation significantly improves the effectiveness of decisions made when requirements do change. Figure 5-5 shows that, when DGA documentation is utilized, the effectiveness improves whatever the type of decision to make might be. We highlight the fact, that no subject was able to make the right decision with ID 7 without DGA-documentation.

5.7.1.3 Subjective utility

Based on results presented in Table 5-3, we can assert, that subjects felt comfortable with DGA documentation. In fact, they found it useful for 77% of decisions they made. Note that this result does not depend on the level of documentation utilized. However, for both levels of documentations, more than 50% of participants felt DGA as useful, as Figure 5-7shows.

5.7.2 Team decision-making

In these subsections we discuss data regarding decision made by team of five people, when decisions were DGA-documented and non-DGA-documented, respectively.

5.7.2.1 Efficiency

From

Table 5-4, Figure 5-4, and Section 5.6.3.3 we can argue, that the use of documentation does not affect the time needed in team decision-making.

5.7.2.2 Effectiveness

From

Table 5-4, Figure 5-6, and Section 5.6.3.4 we reveal, that the use of DGA-documentation affects the effectiveness of decision-making in a team of five people in a significant and positive way. Figure 5-6 shows an important result: for every decision, the relative effectiveness is higher if DGA is applied.

5.7.2.3 Subjective utility

Every team feels DGA useful in every fivefold decision.

5.8 Threats to validity

In order to help the readers qualifying the results of the presented study, we discuss the way in which we mitigated validity threats [Wohlin et al., 2000].

5.8.1 Conclusion validity

Reliability of measure is achieved by a careful selection of the data collection mechanisms, metrics, and a checked data entry. Reliability of treatments implementations are achieved by balances and randomization implemented in the experiment (see Section 5.5.1). We can argue that, in our case, the heterogeneity among subjects did not affect data validity for two main reasons:

Subjects were divided in roles based on their best attitude. Based on their background and behaviors, students seemed to have the same level of specialization.

Each subjects applied both treatments. We avoided fishing activity by: i) defining experiment data analysis details (analysis method, level of significance, etc.) according to standards [Wohlin et al., 2000] and before running the experiment; moreover ii) describing results without any omission (see Section 5.6).

5.8.2 Construct validity

We mitigated mono method bias threats by using qualitative, quantitative, objective, and subjective measures. Our work seems to be not exposed to restricted-generalizability-across-constructs threats. In fact, we cannot find disadvantages in adopting DDRD less than time documenting decisions. Based on results from our third experiment-training session: It is seventeen minutes the average development time for decision documentation (from the scratch) by using DGA. After changes in requirements, it is tree minutes the average maintenance time for decision documentation by using DGA. Evaluation-apprehension and Hawthorne-effect did not threaten our experiment, because decision

makers are subject to high pressure in the real world. We mitigated hypothesis-guessing threats by not spurring subjects on any treatment.

5.8.3 Internal validity

History threats did not show in the presented experiment, because treatments were applied one time per subject. Concerning maturation threats, subjects apparently stayed focused on their tasks. Moreover, as already mentioned, order of treatments is balanced. We mitigated instrumentation threats by refining collection forms, following suggestions that subjects gave to experimenters at training time. Characteristics and motivations of participants were enough for mitigating selection threats. In fact, subjects were at least as motivated as decision makers of real world software projects. Mortality threats did not affect the presented experiment, since no subject withdrew.

5.8.4 External validity

As we based our experiment on a synthetic environment, we cannot assure that the experiment objects represent the real world of software projects. Interaction of setting and treatment can be considered an important and unresolved (probably irresolvable) threat of this type of experiments. Nevertheless, it is important to highlight that in order to mitigate such threats the experimenters: 1) used all their experience with software projects in the real world, 2) reused data from documentation related to real software projects [Babar et al., 2005], 3) spent enough effort in designing and developing experiment objects. Additionally, we cannot assure that our experiment subjects are representative of, and can be compared for level of competence with, decision makers of real world software projects. It is important to highlight that the major part of subjects already had some experience in real world software industry. Moreover, experimenters gave roles to people in the purpose of maximizing the subjects' preferences and skill.

5.9 Conclusion and future works

This chapter presented an experimental study aimed to evaluating the effects of Design Decision Rationale Documentation (DDRD) on individual and team decision-making in case of requirements changes.

The Decision Goal Alternatives technique (DGA) was defined and used as DDRD instance.

Motivations for the presented study were: 1) Individual and team decision-making are two important issues in the development of software systems. 2) It is rational to expect that DDRD improves effectiveness and efficiency of both individual and team decision-making. 3) A previous study [Bratthall et al., 2000], which also investigated the improvement, differs from the present study in many aspects (e.g. type of decision, type of subjects and type of changes); moreover it explicitly called for further investigations.

In order to gain in validity of the experiment results, the experience of the experimenters allowed them to replicate carefully real-world AmI software projects in the adopted experimental synthetic environment. In order to facilitate the experiment replication we published materials and data concerning training sessions and experiment (see [Falessi, 2006]). However, those few, potential, welcome people, who would replicate this work are advised that the experiment planning requested a quite huge effort.

The experiment main results derive from objective data and show that, in presence of changes in requirements, individual and team decision-making perform as in the following: (1) Whatever the kind of design decision might be, the effectiveness improves when DGA-documentation is available. (2) DGA-documentation seems not to affect efficiency. Regarding the utility of DGA, supplementary results, which are based on subjective data, allowed us to confirm the main results by a triangulation activity.

Concerning future works, our plan is to investigate how DDRD can be customized, depending on the usage context (e.g. business goal, domain, design method).

6. A Value-based Approach For the Documentation of Design Decision Rationale: Principles and Empirical Feasibility Study

The capture and reuse of the knowledge acquired while making design decisions is recognized as one of the most promising step for advancing the state of the art in software architecture by preventing the high costs of architecture change, and design erosion. The explicit documentation of the rationale of design decisions is a practice generally encouraged, but rarely implemented in industry because of a variety of inhibitors. Methods proposed in the past for Design Decisions Rationale Documentation (DDRD) aimed to maximize benefits for the DDRD consumer by imposing on the producer of DDRD the burden to document all the potentially useful information. We propose here a compromise which consists in tailoring DDRD, based on its intended use or purpose; it takes advantage of a priori understanding of who will benefit later on from what set of information, and in which amount. In our view, the adoption of a tailored DDRD, consisting only of the required set of information, would mitigate the effects of DDRD inhibitors. The aim of this section of the present PhD dissertation is twofold: i) to discuss the application of Value-Based Software Engineering principles to DDRD, ii) to describe a controlled experiment to empirically analyze the feasibility of the proposed method. Results show that the level of utility related to the same category of DDRD information significantly changes depending on its purpose; such result is novel and it demonstrates the feasibility of the proposed value-based DDRD.

6.1 Introduction

The capture and reuse of the knowledge gathered while making design decisions is actually recognized to be one of the most promising steps

for advancing the software architecture state of the art by preventing its high costs of change, and the design erosion.

However, while we can find the documentation of the resulting design, it is usually not the case for the design *rationale,* i.e., the reasoning that brought the designer to a given choice. In our view, there is merit in capturing not only the design but also the rationale; however, the practice of Design Decision Rationale Documentation (DDRD) is not yet widely spread, due to some inhibitors, such as the required additional documentation effort. DDRD methods proposed in the past tend to maximize benefits for the DDRD consumer by imposing on the DDRD producer to document all the possible information. Because a tailored DDRD should focus just on the most valuable information, we suggest that the use of such a tailored DDRD would mitigate the effects of inhibitors and emphasize on the effects of motivators.

We propose a solution consisting in tailoring DDRD, based on its purpose (i.e., what we intend to do with the documentation). The objective of the present section is twofold: (i) to discuss the application of Value-Based Software Engineering principles to DDRD, (ii) to describe a controlled experiment which empirically demonstrates that DDRD can be tailored based on its purpose (i.e., different DDRD "use cases" require different information).

The section is structured as follows: Section 2 gives some background for the present work. Section 3 describes the proposed Value-Based DDRD, while Section 4 describes its empirical feasibility study. The section concludes with Section 5 including an outlook to future work.

6.2 Study motivation

6.2.1 Value-based software engineering

Nowadays, "much of current software engineering practice and research is done in a value-neutral setting, in which every requirement, use case, object, test case, and defect is equally important" [Biffl et al., 2005]. The Standish Group CHAOS reports [The-Standish-Group] figured out that value-oriented shortfalls, e.g., lack of user input, incomplete requirements, changing requirements, lack of resources, unrealistic

expectations, unclear objectives, and unrealistic time frames, are the common causes of most software project failures. Consequently, "a resulting value-based software engineering (VBSE) agenda has emerged, with the objective of integrating value considerations into the full range of existing and emerging software engineering principles and practices, and of developing an overall framework in which they compatibly reinforce each other" [Biffl et al., 2005]. In the present work we apply VBSE principles on DDRD; we propose a value-based approach to DDRD (VB DDRD), which focuses on documenting only the set of required information based on its purpose.

6.2.2 Design decision rationale documentation

"Design rationales include not only the reasons behind a design decision but also the justification for it, the other alternatives considered, the tradeoffs evaluated, and the argumentation that led to the decision" [Lee, 1997]. As described in Section 1.3.3.

Although several studies empirically demonstrated that the use of DDRD brings numerous benefits [Karsenty, 1996] [Bratthall et al., 2000] [Falessi et al., 2006b], such type of documentation is not widely adopted in practice; and we may wonder why. We focus on the following DDRD inhibitors:

Bad timing and delayed benefit. The period in which design decisions are taken is usually critical for the success of the software project. People involved in the development process, while taking design decision, are already busy in trying to do, as better as possible, the more recognized tasks and to meet the related deadline. In such circumstances, the task of enacting DDRD is considered less important than the other ones and consequently it is put at the end of the queue; and eventually it is never enacted. Our experience shows that when trying to suggest documenting design decisions rationale in the appropriate time, the people answer is likely to be "We already have a lot of problems!"

Information unpredictability. The DDRD consumer and producer are often different persons. People who are in charge to evolve a project are often not the original designers, who in the mean time moved to better, greener pastures. Hence, the DDRD producer cannot predict

which information the consumers will need in the future. As a result, the producer documents all the information that could be useful. Finally, the DDRD becomes too much expensive to write, maintain, and read.

Overhead. Several DDRD techniques already exist; however they are usually focused on maximizing the consumer benefits rather than minimizing the producer effort. Consequently, people involved in the documentation and maintenance activities are supposed to spend a huge amount of effort.

Unclear benefit. Decision-makers do not know for which purpose it is useful to read the rationale of which decision.

Lack of motivation. Caused by absence of direct benefits or no personal interest. People in charge of documenting and maintaining DDRD artifacts (the decision-makers) are not the ones that directly benefit from DDRD; hence, they are not that motivated. Lee [1997] already stressed the importance of the existing problem that DDRD producer and consumer differ. On the same idea, Grudin [1994] pointed out in the context of groupware systems that many of these fail exactly for such a mismatch. Both Lee and Grudin agreed on the relevance of information about who is playing which roles. Moreover, experts may be not interested in making their valuable knowledge explicit as they may consider it as their personal property. In other words, some experts could see no clear advantage in doing DDRD.

Lack of maturity. Only few tools are currently available to support DDRD and therefore the field is to consider as rather immature.

Potential inconsistencies. DDRD implicitly represents the results of the design. If DDRD and the design documents are not updated well, potential inconsistencies in case of decision changes might occur.

Altogether it is clear that DDRD does not come for free, and it will not pay off in all cases (e.g. DDRD is not a value for contracts that do not include maintenance). Hence, in order to exploit DDRD-related benefits in the best possible way, a thorough consideration is requested of the DDRD costs and benefits in different contexts.

6.2.3 Related works

We could not find any previous study, which applied value-based software engineering principles (or similar) to DDRD, but some works seem related. Lee [1997] already stressed that "most current design rationale systems fail to consider practical concerns such as cost effective use", and that "design a cost-effective system is one of the most urgent issue design rationale researchers face". Moreover, always in a context of DDRD systems, he stressed that "what you represent depends on what you want to do with it" and he found that different systems, aimed to support different activities, focused on different category of DDRD information. However, to the best of our knowledge, the present study is the only empirical investigation, involving subjects' opinion, aimed to evaluate the level of importance of different DDRD information for enacting different activities. Shum and Hammond [1994] pointed out that without a good Return On Investment (ROI), the system in charge to manage DDRD would not be used or finally it would be counterproductive. Heindl and Biffl [2005] reported a case study on a type of value-based requirements tracing, which aims to systematically support project managers in tailoring the traces, and is based on the following parameters: stakeholder value, requirements risk/volatility, and tracing costs. The main results of that case study are: (a) value-based requirements tracing takes around 35% of the effort required by full tracing; (b) more risky requirements need more detailed tracing. On one hand, there are many similarities between their study and ours, including the following ones:

- DDRD is strictly related with requirements tracing; in fact, DDRD can be regarded as an extension of the trace.
- Both recognize that producing an extensive documentation (for rationale and tracing, respectively) is a tedious and eventually not effective activity.
- The utilities of tracing and rationale are both strictly related to change in requirements.
- They have common main end-users (i.e., designers).

On another hand, the Heindl and Biffl's study differs from our ones on the following aspects:

- Because we are unable to predict the most useful decisions, DDRD cannot proceed in an ad hoc manner by focusing on the most valuable design objects. In fact, the importance of DDRD should depend both on the importance of the target artifact and on the amount of influence of the decision on that artifact. While the importance of the former could be estimated, there is no chance to estimate the amount of influence of the latter. In consequence of such inability to predict in what extent a decision has influence on other decisions, it follows that we still need to apply DDRD systematically rather than in an ad hoc fashion: in other words, we have to apply the same approach to every single decision.
- The estimation of the right DDRD granularity (i.e. the amount of details to capture for each decision) is still an open issue.

The relation between DDRD and its use cases was previously sketched by Kruchten, Lago and van Vliet [2005], and subsequently it was empirically investigated in two main works by Van der Ven *et al.* [2006], and Tang *et al.* [2007]. Van der Ven's *et al.* [2006] presented a use case model that arose from industrial needs, and is meant to explore how DDRD can be satisfied through the effective usage of architectural decisions by the relevant stakeholders. The use cases have been validated in practice through industrial case studies. Similarly to the present work, they clearly describe that different type of stakeholders enact specific DDRD use cases. Tang *et al.* [2007] reported on a survey trying to evaluate the importance of DDRD as perceived by industrial architects (or designers).The present dissertation shares with Tang *et al.* the concept that "practitioners recognize the importance of documenting design rationale […] however they have indicated barriers to the use and documentation of design rationale" [Tang et al., 2007]. The goal of their study was similar to our ones, as they investigated which type of DDRD information is in general more used by practitioners. We try to go a step further by investigating the level of importance, related to each category of DDRD information, for enacting specific DDRD use cases.

Tyree and Akerman [2005] proposed a framework to document design decision rationale for demystifying system architectures. In the

present study we used such a DDRD template, as an example of instance of DDRD; we are investigating the level of utility of each category of such a DDRD template to enact different DDRD UC.

6.3 A value-based approach for documenting design decision rationale

6.3.1 Rationale

Past DDRD methods aimed to maximize the DDRD consumer's return by forcing the DDRD producer to document all the potentially useful information. Such methods have been employed independently from the system requirements and business context where they should provide benefits. As a matter of fact, we still have to evaluate them as value-neutral methods (see Section 6.2.1). However, because such a value-neutral approach strengthens DDRD inhibitors (see Section **Errore. L'origine riferimento non è stata trovata.**), the question is: What approach should we follow for mitigating the impact of those inhibitors? Our conjecture is that the answer is the injection of Value-Based software engineering principles on DDRD methods, which is what we aim to investigate and describe in the remaining of this section.

Remarkably, DDRD is strongly related with making knowledge explicit and thus making the design process and the design results re-usable. Experiences with software reuse in the late nineties [Jacobson et al., 1997] showed that only strategic, pre-planned reuse that considers the features' value really pays off and allows reuse in the large. Transferring this experience to the DDRD context eventually results in the conclusion that DDRD has to be preplanned in order to be successful in the end. A clear strategy, which considers the potential benefits that can be drawn from DDRD at which costs and risks, is required. As the benefits, costs and risks differ from case to case, the strategy has to be defined anew for different development contexts. In the following section, we propose a Value-Based DDRD (VB DDRD). In particular, we propose to take crucial aspects of every software engineering practice into account: *Where* (project context), *Who* (beneficiary stakeholders), *When* (DDRD type of use), *Why* (Software or business metrics) and *How* (DDRD required information). The rationale behind

this is to take advantage of an *a priori* understanding of who will profit later on, from what set of information, in which amount, in order to cope with the additional effort that has to be spent for producing and maintaining the DDRD. This allows a more Value-Based DDRD than the other DDRD processes already proposed in the literature.

In our best understanding, there has been no studies investigating the need for different information for the different parts neither of the design nor for different type of decisions. Consequently our approach focus on tailoring the documentation based on its purpose rather than on the type of decisions (because there is no suitable categorization).

6.3.2 DDRD Use-cases

Let us consider various usage scenarios for DDRD. A DDRD Use-Case (DDRD UC) occurs when one or more actors use DDRD in order to achieve a certain advantage while enacting a particular activity in a specific project context. A DDRD UC is characterized by the following attributes:

- **ID**: the identifier (primary key) of the scenario; it allows to establish associations among different tables, as shown in the remaining.
- **Actor(s)**: an abstraction on the kind of people involved in the DDRD UC. The Producer is involved in providing and updating the DDRD. The Consumer takes advantages by the existence of DDRD.
- **Context**: System or industry characteristics where the usage scenario happens. This comprises information regarding the environment and the underlying driver.
- **Activity**: The activity that the actors enact, in which DDRD is used.
- **Advantages**: Product or process metrics that determine the benefit of the DDRD usage.

In the remaining of the present section we will call DDRD UC(s) the different purposes of the DDRD as.

Table 6-1 describes a set of five DDRD UC(s) that we will use subsequently in this section. This list is certainly incomplete but

illustrative and valid. To summarize, the use cases of DDRD we consider are:

- Identification of wrong knowledge on the solution space.
- Identification of wrong knowledge on the problem space.
- Design verification.
- Detection of conflicts.
- Impact analysis of changes.

In order to explain the meaning and practical usage of such DDRD UC(s), let us consider the usage scenario No. 5 in Table 6-1 which is a DDRD UC concerning the management of requirement changes. Nowadays, changes in the requirements or business goals are becoming more frequent than in the past. Although much effort has already been spent on software requirements engineering, we are still unable to make that discipline deterministic and fix the requirements in just one shot of the development process. Consequently, fixing software requirements is still a hard job even if customers are maturing and becoming increasingly able to define stable needs. In case of a requirements change, it is crucial for both managers and architects to understand which decisions (and which system artifacts) are still valid and which other ones have to be re-done (i.e., re-designed and/or re-implemented). Traceability among requirements, design decisions, and software artifacts allows managers to easily recognize the type of action required for each software artifact and, consequently, to re-schedule the project activities.

Table 6-1: DDRD use-case description.

DDRD UC						
ID	Actor		Context		Activity	Advantages
	Producer	Consumer	Environment	Driver		
1	Designers / Architects	Designers / Architects	Several designers with similar competence but different levels of expertise	Large system	Detection of wrong knowledge on decisions	System quality
2	Designers / Architects	Requirement Analysts	Ambiguous or conflicting or inter-related requirements	System complexity	Detection of wrong requirement understanding	Effort
3	Designers / Architects	Reviewers	Common context	Common context	Design checking (verification and evaluation)	Effort
4	Designers / Architects	Mainteners	Maintenance of a built system	System evolution	Detection of conflicts among new requirements and old decisions	Effort
5	Designers / Architects	Designers / Architects / Managers	Common context	Common context	Impact Evaluation	Effort

6.3.3 Key idea

Let us call “required information” a kind of information without which, independently from the effort of the readers, the meaning of something cannot be understood; “useful information”, a kind of information that helps to a small or large extent the readers to understand the meaning of something; “optional information”, a kind of information which helps readers, but it is not required, in understanding something.

The key idea is that all the information included in a DDRD might be useful but sometimes some information is merely optional. We expect that the amount of importance related to the information included in the DDRD depends on the DDRD UC. In other words, we expect that different DDRD UC(s) require different categories of DDRD information. In such case, the DDRD can be tailored based on the DDRD UC(s) to enact. In our view, the adoption of a tailored DDRD, consisting only of the required set of information, would mitigate the effects of DDRD inhibitors, as described in the remaining of this section.

6.3.4 Process

Figure 6-1 describes the flow of activities (rectangles) and the flow of information produced/consumed (parallelograms) in our VB DDRD process. In order to select a scenario – among those that are still left for analysis, if any – from a predefined DDRD-UC database (see Figure 6-1), the first activity in Figure 1 ("Scenario selection") is executed. Subsequently, information concerning the selected scenario is analyzed, which is:

- Context: to figure out the probability that this scenario will occur in the business and system context.
- Metrics: to realize the benefits that such a usage scenario provides.
- Required DDRD information: to estimate the cost of (amount of effort to spend on) documenting and maintaining DDR.

At this point, the adoption of DDRD in the current scenario(s) is economically evaluated for ROI. Since the benefit is achieved only in case the related scenario occurs, we weight the ROI by the occurrence probability of the specific scenario (i.e., ROI(s)=OccurrenceProbability(s) * Benefit(s) / Cost(s).

Note that this approach is orthogonal on customizing the DDRD based on the importance of the decision.

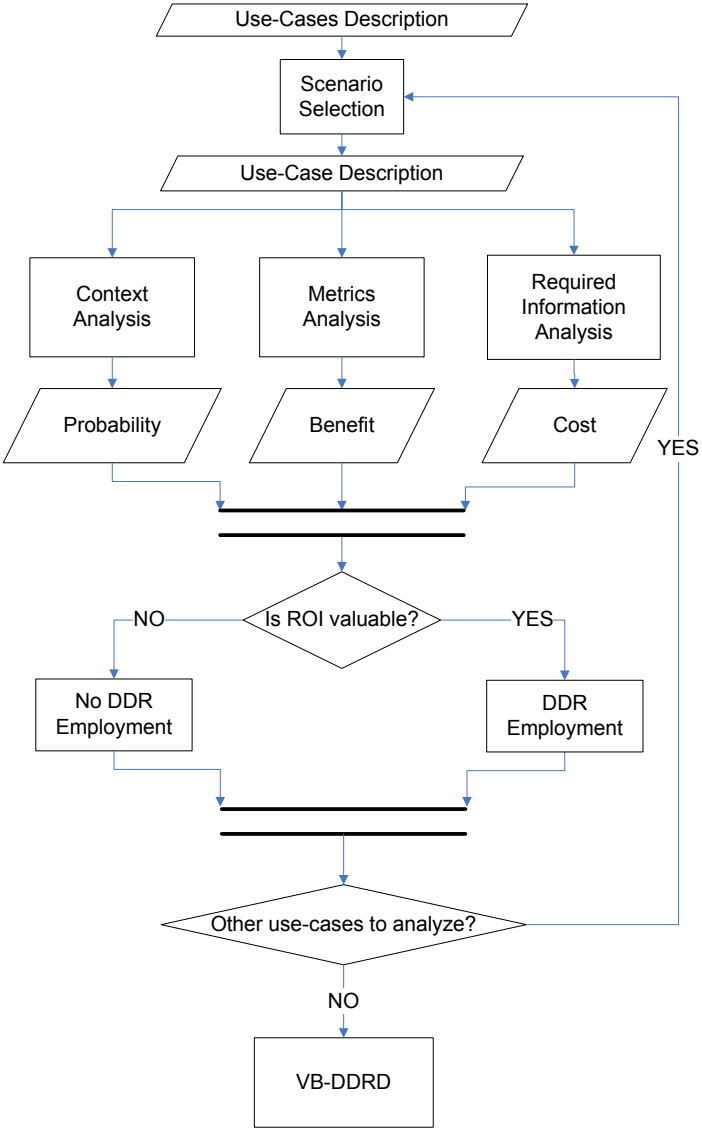

Figure 6-1: Activity and information flows for the proposed Value-Based Rational Documentation process.

6.3.5 Expected advantages

Three main elements characterize our proposed value-based rationale documentation process:

- A clear definition of the advantages, the related overall value, and the roles of stakeholders involved in a particular DDRD UC execution
- A tailored DDRD involving only the required information
- The assumption that different DDRD UC(s) require different DDRD information.

Figure 6-2 shows which of those components we expect to mitigate which DDRD inhibitors. In particular:

- **Overhead**: A tailored DDRD implies less information to document and maintain; hence, a diminished effort has the effect of mitigating the overhead.
- **Potential inconsistencies**: The tailored DDRD implies less information and hence less documentation. Less documentation implies both less required effort for DDRD maintenance and less probability of inconsistencies occurrence.
- **Bad timing & delayed benefit**: The possibility to spend less time to produce the DDRD highly increases the possibility that people, who are busy to meet their projects deadlines, find enough time to develop such DDRD.
- **Lack of motivation**: The clear definition of who will profit from who allows the existence of a role (performed by real person or virtually) in charge of controlling that the specific producers provide, and the relate consumers use, the expected DDRD.
- **Unclear benefit**: The clear definition of which advantages are achieved by enacting which scenario (DDRD UC) mitigates misunderstandings about pros and cons.
- **Information unpredictability**: despite the DDRD producer cannot perfectly estimate which information the consumer will require; in this work we provide some data to do that (see Figure 6-3).

- **Maturity**: the relationship between DDRD information and DDRD UC provides a new and promising tactics to increase maturity of DDRD.

It is rather evident that such a process to document design decision rationale will address and mitigate to some respects most of the DDRD inhibitors. However, we are still left to validate its key assumption that different DDRD use-cases require different DDRD information.

6.4 Empirical feasibility study

The experiment is one of the main methods in empirical software engineering. It is characterized by the fact that a project is performed only for the purpose of the empirical study and not for the purpose of the project itself; in particular, the type of environment and/or object (i.e., project) and/or the subjects (i.e., people) can differ from the reality. This method is mostly used to confirm extant theories, to explore relationships among data points describing one variable or across multiple variables, to evaluate accuracy models, or to validate measures. In general, an experiment, in respect to other empirical methods, provides the highest level of formality, rigor and control on measures [Zelkowitz and Wallace, 1998] [Wohlin et al., 2000].

6.4.1 Experiment process description

6.4.1.1 Research Objectives

As discussed just above, we do not imagine our VB DDRD to be a panacea, but we expect to obtain both advantages and disadvantages in adopting it. In the absence of valid formal models for our VB DDRD, we are unable to compute pros and cons of our approach. Our decision is hence to proceed empirically, in particular to conduct controlled experiments, with the goal of investigating the feasibility of our approach rather than comparing it with other ones. We preferred a feasibility study to a comparison because our method provides what other methods do not provide: a realistic means for mitigating all the known DDRD inhibitors. Additionally, it seemed that evaluating our process advantages (i.e., the amount of mitigation on DDRD inhibitors) is not an interesting direction, for the following reasons:

- **Evident advantages**: it seems obvious, for instance, that writing a subset of information (tailored DDRD) requires less effort than writing the information in full (a complete DDRD).
- **Difficult empirical validations**: it is hard, for instance, to measure how the personal interests can be mitigated by the presence of a manager who checks the DDRD development.

The goal of our study, in the sense given by Basili [1994], is *to analyze* the DDRD *for the purpose of* evaluation *with respect to* the perceived utility *from the point of view of the* researcher *in the context of* post-graduate Master students of software engineering. The aims of this empirical study is both explorative and confirmative: i) to discover the perceived levels of importance related to each category of DDRD information for enacting different DDRD use cases, ii) to confirm the hypothesis that the level of importance related to the same category of DDRD information is different for different DDRD UC(s). The latter is the key principle of our proposed VB DDRD, without which our VB DDRD would not work.

6.4.1.2 Hypotheses

In order to investigate the assumption that the levels of importance related to categories of DDRD information are different for different DDRD UC(s), we derive the following null hypotheses (respectively alternative hypotheses) for the present study. When enacting DDRD UC(s) with ID(s) *i* and *j*, there is no significant difference ($H_{0ij\text{-}}$) (resp. there is significant difference, $H_{1ij\text{-}}$) between the levels of perceived importance related to the category of DDRD information *x* ($H_{\_x}$). Notice how *i* and *j* can be any DDRD UC(s); in the present study we used the DDRD UC(s) described in Table 6-1.

We highlight that the assumption that the levels of importance related to the same category of DDRD information are different for different DDRD UC(s) is valid only if the above null hypothesis is rejected for one or more combinations of *i* and *j*. In particular, each specific combination of *i* and *j* in which the null hypothesis is rejected, identifies a specific pattern in the perceived importance level of a category of DDRD information.

6.4.1.3 Variables

The DDRD UC is the experiment factor. The experiment takes into account five DDRD UC(s) (see Table 6-1); they represent five levels of this independent variable (i.e. treatments). As dependent variables, we used the utility related to each specific category of DDRD information as perceived by subjects for enacting a specific DDRD UC. Participating subjects expressed quantitative measures of the utility of a specific category of DDRD by a 3-point ordinal scale (useless, optional, or required). We used an array of thirteen categories of DDRD as proposed by Tyree and Akerman [2]. We controlled at a constant level the remaining independent variables such as experience of the participating subjects, experiment materials, environment, and complexity of the experiment objects. We also blocked a further independent variable, the type of documentation, because we were not interested in investigating that dimension, as explained in the remaining sections.

6.4.1.4 Subjects

Fifty students at the last year of their Master Degree in something similar to computer engineering, at the University of Rome "Tor Vergata", participated in our work as experiment subjects. While most of our subjects had already had some experiences in software companies, only few of them can be considered as software professionals. Hence, in order to gain on external validity, we modeled five different roles or types of stakeholders: authentication, human interface, operative system, communication protocol, and data storage. Then, subjects expressed their preference for each role, according to their previous experience and level of confidence with the responsibilities of a role; it was done well in advance of the last training session. Afterwards, we assigned roles to subjects by trying to maximize the coverage of their expressed preferences (i.e., experience).

6.4.1.5 Design

In order to analyze the relationship, if any, between the utility of DDRD information categories and the DDRD purposes, we selected five DDRD use-cases to investigate (see Table 6-1) and five decisions for each role. We adopted the DDRD UC(s) reported in Table 6-1 among the ones available from the literature [Kruchten et al., 2005] with the aim to

minimize the validity threats, according to our context (e.g., available time, subjects experience in executing the use cases, experimenters ability in providing fine replica objects).

Each experiment decision consists in a DDRD-documented decision already made in the past by a decision-maker who virtually left, and a new set of requirements. Since a decision (and its DDRD) is compatible to all five DDRD use cases, we utilized all the twenty-five decisions with each DDRD UC.

We utilized two types of documentation (i.e. tailored or complete) because the level of perceived utility, related to a category of DDRD information (i.e. expected utility), may change based on the presence or absence of enough documentation for such category. Because we are not interested to investigate the impact of such a factor, we blocked the experiment with respect to that variable [Wohlin et al., 2000].

In order to limit the occurrences and mitigate the influence of several threats to validity, we balanced the experiment design in this way:

- Each decision has been adopted the same number of times.
- Each treatment (on each decision) has been applied the same number of times.
- Each subject applied all DDRD UC (on different decisions).
- All treatments (i.e. DDRD UC(s)) have been applied the same number of time (i.e. 10) in all the available orders (i.e. as first, second, third, fourth, and fifth).
- Each subject applied only one time each of the five DDRD UC(s); each subject encountered just one time all the five decisions belonging to his specific competence.

The experiment design is described in Table 6-2; in particular it relates a subject to a specific set of fixed values for: i) competence, ii) order of application of the received DDRD UC(s), iii) type of documentation. Three sub-tables compose Table 6-2. The first sub-table describes which subjects ID belongs to which competence; the first column describes competences while the other columns report on the

ID(s) of the assigned subjects. The second sub-table describes the order of DDRD UC execution; the first column describes the execution order, while the others describe the ID of the assigned DDRD UC(s). For example, the second column describes an order where the DDRD UC with ID 1 is the first DDRD UC to execute, while the last column indicates that it is the fifth DDRD UC to enact. The third sub-table describes the type of DDRD (complete or tailored) that is related to the decision to take; the first column describes the decision order while the others describe the type of DDRD. Subjects ID belonging to the column i in the first table also belongs to the column i in the other sub-tables. Hence, for instance, the subject with ID 49 belongs to the HI competence, executed as his or her first DDRD UC the one with ID 1, which has a tailored documentation type.

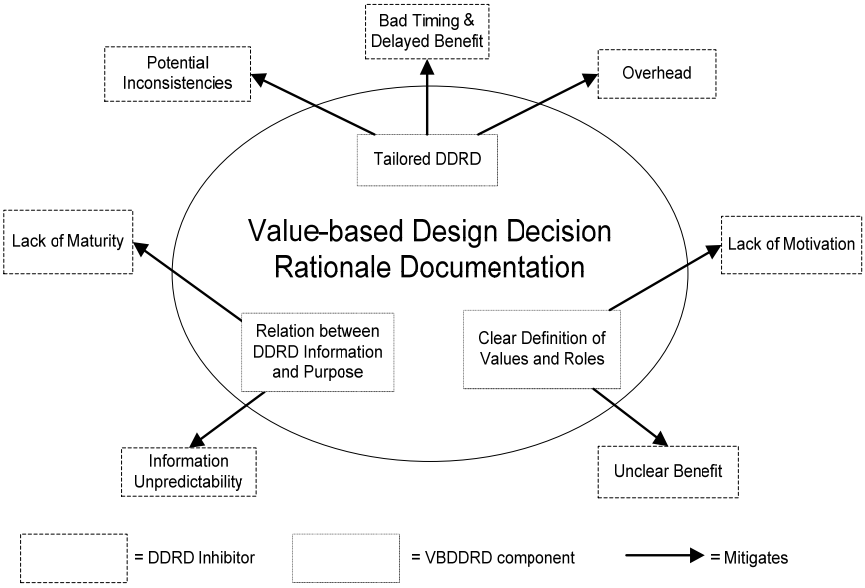

Figure 6-2: Expected effects on DDRD inhibitors

Table 6-2: Experiment Design

Competence	ID Subjects									
HI	49	20	21	23	24	28	29	30	31	32
HW	12	41	42	44	45	46	1	2	3	4
WR	9	11	6	7	8	13	22	43	27	39
AU	34	5	10	14	15	17	25	35	47	26
SO	36	48	33	37	38	40	50	51	52	53

DDRD UC Order	DDRD UC Order									
1st	1	5	4	3	2	1	5	4	3	2
2nd	2	1	5	4	3	2	1	5	4	3
3rd	3	2	1	5	4	3	2	1	5	4
4th	4	3	2	1	5	4	3	2	1	5
5th	5	4	3	2	1	5	4	3	2	1

DDRD UC Order	Type of DDRD (C=Complete, T=Tailored)									
1st	T	C	C	C	C	C	T	T	T	T
2nd	C	C	T	T	T	T	T	C	C	C
3rd	T	T	T	C	C	C	C	C	T	T
4th	C	C	C	C	T	T	T	T	T	C
5th	T	T	T	T	T	C	C	C	C	C

6.4.1.6 Experimental Material and Tasks

In order to replicate the context of the real world, we used a synthetic software project, which is quite similar to another experimental object we had already been using successfully [Falessi et al., 2006b]. The experiment project was concerned with a public transportation system characterized by ambient intelligent issues (e.g., resource constraints, heterogeneous sensors, etc.) [Remagnino et al., 2005]. It provided the possibility both to derive five decisions regarding five different competences (i.e., roles), and to provide valid experiment objects (i.e., good replica), as for a previous experiment we conducted [Falessi et al., 2006b].

The concepts related to software architecture were not covered by a single role but any role was concerned with all the decisions related to it. As a matter of fact, the adopted decisions are inter related one to each other; for examples:

The decision related to the selection of a communication protocol depends on the topology of the nodes, the specific communication

mechanism (e.g., publish-subscribe or event-driven) and architectural style (e.g., blackboard or client-server).

The selection of a data storage mechanism depends on the type of DMBS, the communication protocol, and the architectural pattern (e.g., MVC).

Other issues that drove the adopted decisions include: available budget, desired compatibility, maintainability, scalability, security, etc. Such constraints, requirements (new and old), rationale and related decisions (and their status) were described in the provided DDRD.

Concerning the requirements change, we adopted causes like: 1) Variations in the industrial strategic partnerships; 2) Changes of customer requests resulting from his or her experience in using the previous version of the product; 3) Technology advances.

Each subject received the materials containing the following items:

- Experiment rules,
- System main characteristics,
- Five different decisions (with related DDRD and new requirements),
- A description of which DDRD UC enacts on which decision, and in which order,
- The form to fill data in.

In particular, for each of the five DDRD UC(s) to enact, subjects had to exec the following steps:

- To understand the current DDRD UC to enact,
- To enter the form with the current time (initial time),
- To read the DDRD related to a specific decision,
- To enact the DDRD UC (write the requested answer),
- To write the final time on the form,
- To describe the level of perceived utility for each category of DDRD information.

Tyree and Akerman [2005] proposed their framework to document design decision rationale for demystifying past and future system architectures. Such a framework is composed of the following thirteen categories of DDRD information: Issue, Decision, Status, Assumptions, Constrains, Position, Argument, Implication, Related Decisions, Related

Requirements, Related Artifacts, Related Principles, and Notes. In the present study we used such a DDRD template and its categorization, as an exemplar DDRD instance. In fact, such DDRD template includes also the category "group" which we ended up not using, due to practical constraints.

Table 6-3 shows the form that subjects filled in during the experiment; the first two columns describe the DDRD UC order of execution. The following three columns describe the initial time, the answer to the DDRD UC (described in the same row, column 1), and the final time when the DDRD UC completed, respectively. The columns from 6 to 18 describe thirteen perceived levels of importance: a column identifies a DDRD information category; a row identifies an enacted DDRD UC. Notice how the first column in Table 6-3 changes according to the second sub-table in Table 6-2. In particular, the reported instance of experiment form was assigned to those experiment subjects with ID(s) reported in columns one and six in Table 6-2, first sub-table.

Table 6-3 Form that subjects filled in during the experiment.

Order		Answer			Utility level of DDRD information (0=Useless, 1=Optional, 2=Required)												
ID UC	ID D	Initial Time	Answer	Final Time	Issue	Decision	Status	Assumptions	Constraints	Positions	Argument	Implications	Related decisions	Related requirements	Related artifacts	Related principles	Notes
1	1																
2	2																
3	3																
4	4																
5	5																

6.4.1.7 Execution Preparation

We've gained experience over the years of our empirical studies in conducting controlled experiments similar to the one in this section, which took into consideration: type of objects (DDRD, ambient intelligence domain, paper-based), subjects (class size, experience, expectation), and context (laboratory). Such an experience helped us in: (1) designing and implementing the experiment objects, (2) settings the experiment lab, (3) motivating the students, (4) training the participating subjects. In particular, regarding the training phase: (a) we choose a time duration of five hours (three sessions), (b) we avoided to use terms which we had experienced to be source of misunderstanding, (c) we corrected the students' wrong assumptions and expectations regarding the experiment, and (d) we carefully distilled information regarding the experiment, by clearly explaining almost all the experiment related attributes less the experimenters' expectations. In order to avoid loss of balance in the experiment, due to eventual absences, we arranged for eight spare subjects.

We carefully checked for the presence of the experiment subjects at all the training sessions; as matter of fact, from a set of sixty-four students we excluded six of them, because they did not attend one or more training sessions. Hence, we had fifty-eight students regularly trained to perform as experiment subjects. In order to avoid to get the planned experiment balances lost at experiment conduction time, due to possible unbalanced absence of subjects, our decision was to select at random eight of the available students, one or two for each needed competence, and keep them as spare subjects (of course, we did not give information to students about that our design decision to discard up to eight groups of collected data).

6.4.1.8 Execution Deviations

During the conduction of the experiment, we did not observe any particular deviation from the experiment plan. Actually, three non-spare subjects were absent on the day of the experiment, and we were able to replace them with present spare subjects, without affecting the experiment balance.

6.4.1.9 Data Set Collection and Reduction

Because fifty experiment subjects enacted five DDRD UC(s), we had two hundred fifty DDRD UC(s) executed. Because for each DDRD UC execution we had a set of sixteen answers to collect (initial time, answer, the utility level of each of thirteen DDRD information categories, and final time), the overall experiment produced a total of four thousand data items.

During data transcription, in order to detect and correct mistakes, data were re-dictated and checked several times. Moreover, data were automatically submitted to some semantic analysis; for example, for each form, the "initial time" of a DDRD UC is expected both to proceed the "final time" of the same DDRD UC, and to follow the "final time" of the previous DDRD UC. Due to this kind of filtering, we were able to exclude six answers (including the related required time, correctness, and the thirteen DDRD utility levels) out of two hundred fifty answers (DDRD UC executions) from any further analysis. Subsequently, we looked for statistical outliers, and eventually we were able to find eight of them. Because, based on the experiment design, every subject enacted every DDRD UC, every DDRD UC instance was executed the same number of times, and every subject applied both treatments, our decision was to consider peculiarities (i.e. outliers) not less significant than the remaining collected data. Hence, we did not exclude statistical outliers from further analysis.

6.4.2 Experiment Result Description

6.4.2.1 Data Analysis

6.4.2.1.1 Descriptive Statistics

In general, the more a category of DDRD information is perceived as "Required", the more it is valuable for DDRD consumers.

Table 6-4 shows the mean and the variance of the subjects who perceived a specific category of DDRD information as required while enacting the DDRD UC(s). Hence, Table 6-4 helps us to understand which parts of the documentation are interesting for the readers in case they have to enact all the DDRDUC(s). In other words, DDRD producers can use Table 6-4 as a reference to choose whether to neglect

or to include any category of DDRD information, in the DDRD to be provided. This result is very similar to [Tang et al., 2007].

Table 6-4: Percentage of subjects that felt as "required" a specific category of DDRD information

DDRD Information	Mean (%)	Variance
Issue	91	25
Decision	79	110
Status	25	318
Assumptions	49	117
Constraints	54	54
Positions	72	43
Argument	67	45
Implications	21	28
Related decisions	28	104
Related requirements	74	150
Related artifacts	14	8
Related principles	21	60
Notes	5	9

Figure 6-3 shows the percentage of subjects who, when they finished executing a specific DDRD UC, marked a specific category of DDRD information as "Required". Figure 6-3 supports our understanding of the relationship, if any, between the importance of a specific category of DDRD information and specific DDRD UC(s). In other words, DDRD producers can use Figure 6-3 as a reference to choose whether to neglect or to include any category of DDRD information, in the DDRD to be provided, according to the DDRD UC(s) to enact.

In the best of our knowledge, the results presented in Figure 6-3 are completely novel; this holds not only in relation to the DDRD but also to any other type of documentation (i.e., we did not found any study investigating the amount of perceived utility related to a documentation category, for specific purposes). Finally, Figure 6-3 seems to confirm that the level of perceived utility does change both in relation to the category of DDRD information, and the DDRD UC enacted.

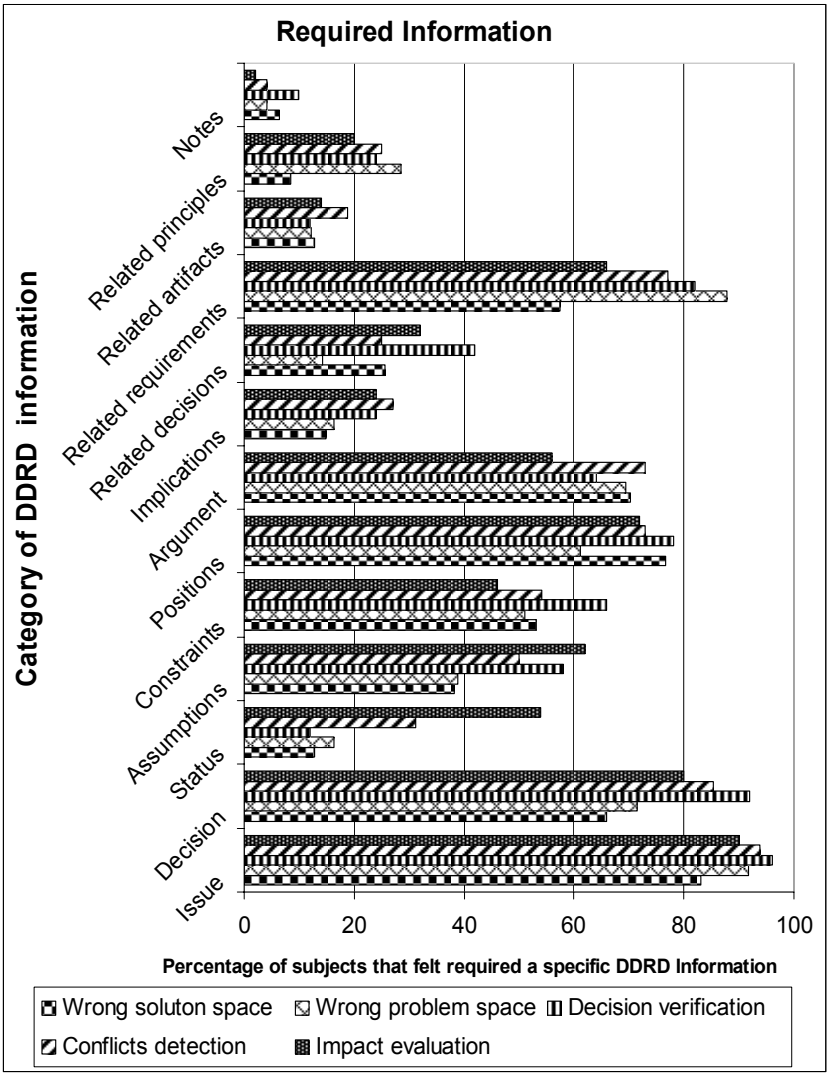

Figure 6-3: Percentage of subjects that felt as "required" a specific category of DDRD information for enacting a specific DDRD UC.

6.4.2.1.2 Hypothesis Testing

For each category of DDRD information, in order to test the null hypothesis (the level of the perceived utility insignificantly depends on

the DDRD UC), we used the Kruskal-Wall test on all the five DDRD UC(s). In case it was possible to reject (p-value < 0.05) that null hypothesis for a category of DDRD information, then we used the Mann-Whitney for testing all the possible combinations of couples of DDRD UC(s) with respect to such a category of DDRD information. Table 6-5 describes the results of the statistical test on the level of perceived utility related to a specific category of DDRD for enacting a specific combination of DDRD UC(s). Hence, Table 6-5 describes the results of applying these tests; "Yes" means a statistical significant difference (p-value < 0.05) in the level of perceived utility for enacting specific DDRD UC(s) (all the adopted DDRD UC(s) in the second column, a combination of two DDRD UC(s) in the others) regarding a specific category of DDRD information (as identified by the first column).

6.4.2.2 Result Interpretation

We briefly discuss the results related to single category of DDRD Information.

Issue. According to our expectations, results show that this category has been perceived extremely useful. Obviously, the description of the issue to be addressed by the decision is key whatever the DDRD UC might be. In particular, by observing Table 6-4 we noticed that such a category have been perceived quite always (i.e. 91% of the time) as "required". Its low variance (i.e. 25) reveals that its huge importance is shared among all the DDRD UC(s). Because its importance does not significantly change in any DDRD UC(s), Table 6-5 confirms such a result of generalizability.

Table 6-5: Statistical significance (Yes)/insignificance (No) in the difference between the level of perceived utility related to a category of DDRD for enacting a specific combination of DDRD UC(s).

DDRD Information	All 5	DDRD UC ID Combination									
	DDRD UC	1-2	1-3	1-4	1-5	2-3	2-4	2-5	3-4	3-5	4-5
Issue	No	No	No	No	No	No	No	No	No	No	No
Decision	Yes	No	Yes	Yes	No	Yes	No	No	No	No	No
Status	Yes	No	No	No	Yes	No	No	Yes	No	Yes	Yes
Assumptions	Yes	No	Yes	No	Yes	Yes	No	Yes	No	No	No
Constraints	No	No	No	No	No	No	No	No	No	No	No
Positions	No	No	No	No	No	No	No	No	No	No	No
Argument	No	No	No	No	No	No	No	No	No	No	No
Implications	No	No	No	No	No	No	No	No	No	No	No
Related decisions	Yes	No	Yes	No	No	Yes	No	No	Yes	No	No
Related requirements	Yes	Yes	Yes	Yes	No	No	No	Yes	No	Yes	No
Related artifacts	No	No	No	No	No	No	No	No	No	No	No
Related principles	No	No	No	No	No	No	No	No	No	No	No
Notes	No	No	No	No	No	No	No	No	No	No	No

Decision. Subject felt this information quite useful. In particular, for all the cases, the information was required more than the 60% of the times (see Table 6-5). However, its importance resulted relatively low when used to check wrong solution/problem space. Such a result can be explained by taking the example in which the category Related requirements describes the need for high security while the category Arguments includes low security. This mismatch is already evident without the further information provided by the category Decision.

Status. In theory, the "status" of a decision is really important in order to evaluate the impact of a change while it is not important for other considered DDRD UC(s), such as for example the identification of erroneous requirements (i.e. DDRD UC with ID 2). It confirms this theory Figure 6-3 jointly with the observations that the Status shows low importance in the average (i.e. 25%) but also presents huge variance when different DDRD UC(s) are taken in consideration.

Assumptions. By analyzing Table 6-4, we can observe that such information has been perceived as required only for the half of the times. In particular, by observing Figure 6-3, we can notice that the utility of this information is really low when used to check wrong solution/problem space. Again, it can explain this, the fact that it is more important to know Related requirements and Arguments than Assumptions for such DDRD UC(s).

Constraints. Again, such information has been perceived as required for only the half of the times. However, based on Table 6-5, the information seems not to change its level of importance according to the different purposes of the documentation.

Positions. Generally, in order to enact DDRD UC(s) of any kind, it has been perceived quite useful (i.e., 73% of times) the description of the alternatives taken in consideration while making a particular decision; it confirms this point, the absence of significant difference among any touple of DDRD UC(s).

Arguments. The information related to the reasons for making a particular decisions has being perceived in general quite useful (i.e., 73% of times) for enacting all the DDRD UC(s) as confirmed by the absence of significant difference among any different DDRD UC(s). In particular, its utility is higher than we expected for the case it was used to estimate the impact of a new decision.

Related decisions. The information regarding related decisions has been perceived quite useless (i.e,. 28% of times). We expect such information to become the most important one when standardized tools will support it by providing fast navigation through several DDRD(s) and allowing to process DDRD(s) relationships.

Related requirements. It has been perceived quite useful (i.e., 74%) the description of the project requirements that relate to a decision to re-design. However, the importance of this information significantly changes depending on the purpose of the documentation. In particular, Table 6-4 validates the following our expectation: it is not useful to know the Related requirements but just the Arguments that led to a decision, to understand if the decision maker had a good knowledge on the solution space.

Notes. For the sake of completeness, any documentation framework includes a field, like our field Notes, to allow the users to insert information, if any felt as missing and due. Based on Figure 6-3, our experiment subjects considered almost useless the category "Notes" of information (i.e., only 5% of the times). Such a result suggests that the proposed DDRD framework [Tyree and Akerman, 2005] fits well both the decisions, and the DDRD UC(s) that have been adopted in this empirical study.

By analyzing Table 6-5 horizontally, we can find that five DDRD information categories (Decision, Status, Assumption, Related decision, Related requirement) show significant statistical difference in the perceived utility level while enacting different DDRD UC(s). In other words, according to our results, the utility of such categories of DDRD information depends on the DDRD UC to enact. Hence, DDRD producers can rely on Table 6-5 as a statistical significant reference to choose whether to neglect or to include any category of DDRD information, in the DDRD to be provided, according to the DDRD UC(s) to enact. Additionally, by analyzing Table 6-5 in a vertical way, we can find that each possible combination of DDRD UC differs in the level of perceived utility for at least one category of DDRD information, despite the combination of DDRD UC with ID 2 and 4. In other words, according to our results, whether DDRD is tailored on the basis of the "expected" utility level, the DDRD UC(s) with ID 2 and 4 would have the same DDRD, while all the other DDRD UC(s) would have a different tailored DDRD (i.e. a DDRD where different categories of information are included and/or omitted).

Note that Table 6-5 describes an insignificant difference among DDRD UC(s) with ID 2 and 4; this result is completely different from defining them as identical: as a matter of fact, for instance, the columns 1-4 and 2-4 are different.

To conclude, both descriptive and statistical results confirm our expectations that the level of utility related to the same category of DDRD information significantly changes according to the DDRD UC. Therefore, the DDRD consumer requires, or not, a specific category of DDRD information according to the DDRD UC to enact. Consequently, results suggest that the DDRD producer can tailor the documentation by

including only the information required for the DDRD UC(s) that are expected to be enacted.

According to Figure 6-3 our proposed VB-DDRD would provide an effort saving of the 50% in the average (among DDRD UC(s)) by supposing that i)all categories require the same effort and, ii) a category is included in the documentation only in case its level of perceived utility is higher than 50%.

6.4.2.3 Threats to Result Validity

The aim of this subsection is firstly to help the readers to qualify the aforementioned results, and secondly, to propose future research by highlighting some of the issues associated with our study.

6.4.2.3.1 Conclusion validity

Reliability of measures: Our experience in similar experiments allowed us to refine materials and metrics in order to gain reliability; in particular regarding:

Time: we forced the subjects not to estimate the required time but to write the initial time and the final time before and after enacting each specific DDRD UC. This should have avoided wrong estimations (the elapsed time is perceived in different ways based on the mental status) or coarse grain evaluations as in [Falessi et al., 2006b].

Perceived utility: during the experiment we adopted an ordinal scale (i.e. useless, optional, and required), which has been clearly described and discussed during the training session by examples. This should have avoided misalignment in judgment.

Low statistical power: we adopted a standard threshold for rejecting hypotheses (i.e., P-Value=0.05)

Violated assumption of statistical tests: we firstly analyzed the normality of the each distribution by using four statistical tests (Computed Chi-Square, Shapiro-Wilks, Skewness, and Kurtosis). Finally we were able to reject the hypothesis that each distribution comes from a normal distribution with 99% confidence according to at least one test. Then we used the non parametric test Mann-Whitney or Kruskal-Wall, in cases where the distributions to be compared were two or more, respectively.

Fishing: all the performed analyses were planned before the execution of the experiment, hence before start to handle the result. Reasons for the performed analysis are previously described.

Random irrelevances: we balanced the design in respect to several factors; we did not perceive any disturbs during the experiment execution.

Random heterogeneity: subjects were almost homogeneous since share the university course. However since the experiment design was balanced in different aspects we would deduce that eventual differences were balanced too, and hence did not influence the results.

6.4.2.3.2 Internal validity

Internal validity is the degree to which conclusions can be drawn about the causal effect of the independent variables.

The level of internal validity should be considered high because all the subjects executed the experiment at the same time (in parallel). Moreover, the design was balanced with respect both to treatments (i.e. DDRD UC(s)) and types of documentation (complete or tailored).

6.4.2.3.3 Construct validity

Construct validity is the degree to which the independent variables and dependent variables accurately measure the concepts they purport to measure.

Mono-operation bias: this was the threat with highest priority. In fact, in order to prevent that it influences experiment results the peculiarities of the objects (decisions) used, we adopted a set of twenty five different objects. Hence, since all the subjects enacted all (five) the DDRD UC(s), the distributions analyzed (and hence the related results) include fifty data at least, and a maximum of two hundred fifty data.

Hypotheses guessing and experimenter expectancies: During the training session we carefully distilled the set of information regarding the experiment by omitting our expectations and experiment hypotheses.

Evaluation apprehension: We tried to encourage subjects to run the experiment with the highest concentration while avoiding evaluation apprehension. To these aims, we clearly described them that they would

not be evaluated for their answers (since such answers are subjective and hence not objectively judgeable).

Low motivation Subjects known that their answers would have been checked for correctness and that future interviews with us to explain the provided answers would have been possible. This fact avoided the possibility that subjects would have executed the DDRD UC without accuracy. Additionally, it improved the subjects' motivation the presence of about 10% of subjects acting as spares. We observed a high level of motivation, as it can be checked through the web[1].

Restricted generalizability across constructs: the reasons of which impact we tried to study and why is carefully described above.

Inadequate preoperational explication of constructs: we had enough time (six months) between the proposal of our approach and the experimental study (the present one).

Reliability of treatment implementation: Each subject applied both treatments and each treatment was applied using the same process.

6.4.2.3.4 External validity

External validity is the degree to which the results of the research can be generalized to the population under study and other research settings. In general, the greater the external validity, the more the results of an empirical study can be generalized with regards to actual software engineering practice.

Subject representativeness: One of the main threats to validity of the present work is the fact that we used students instead of professional

---

[1] http://ese.uniroma2.it/DDRD-Experiment-December2006.zip

as experiment subjects, for enacting the DDRD UC(s). However, since most of them have already spent internships or part-time work in industry, the experience of the subjects should be considered not null but instead very specific, to the best of our understanding. Consequently, we designed the experiment in order to maximize the students' experience by assigning them a specific role and using a project with decisions belonging to such roles. This gave us the possibility to adopt subjects with a level of experience/knowledge similar to professionals.

Object representativeness: The adopted project was just described not really implemented. This allowed us to replicate a complex system in details and with a good level of similarity. In fact, the adopted decisions were hard to (re) make because characterized by several, opposite and inter-related objectives related among decisions, as it is in the reality. We decided to develop a large amount of decisions (i.e., twenty-five, five per role) with the objective to relate the experiment results to the associated DDRD UC rather than the specific decision adopted. In other words, by using a small amount of decisions we would not be able to understand whether the result depends on the peculiarities of the specific decision.

Interaction of history and treatment: we executed the experiment in a week-day (Thursday) at a regular working hour (early afternoon).

Interaction of setting and treatment: actually there is no standard for DDRD; hence we cannot have been too different from them.

Interaction of selection and treatment: we maximize the overall experience of subjects with the aim of being representative as professional.

Interaction of history and treatment: we execute the experiment in a normal day (Thursday) in a normal time (early afternoon).

6.5 Conclusion and future work

Older Design Decisions Rationale Documentation (DDRD) methods aimed at maximizing the DDRD consumer benefits by forcing the DDRD producer to document all the potential useful information; they

eventually ran into too many inhibitors to be used in practice. In this section we propose a value-based approach for documenting the reasons behind design decision (VB DDRD), based on *a priori* understanding of who will benefit later on, from what set of information, and in which amount. Such VB DDRD offers means to mitigate all the known DDRD inhibitors and it is based on the hypothesis that the set of required DDRD information depends on the DDRD use case (DDRD UC) to enact. In order to validate such a hypothesis we ran an experiment in a controlled environment, employing fifty subjects, twenty-five decisions, five different DDRD UC(s), and 250 DDRD UC(s) executions. Each subjects practically used the documentation to enact all the five Use Case(s) by providing an answer and a level of utility for each category of DDRD. Both descriptive and statistical results confirm our expectancies that the level of utility, related to the same category of DDRD information, significantly changes according to the DDRD UC. Such result is novel and imply that the DDRD consumer requires, or not, a specific category of DDRD information according to the DDRD UC to enact. Consequently, results suggest that the DDRD producer can tailor the documentation by including only the information required for the DDRD UC(s) that are expected to be enacted (i.e. valuable). This result demonstrates the feasibility of our proposed VB DDRD.

DDRD producers can use such experiment results as a reference to choose whether to neglect or to include any category of DDRD information, in the DDRD to be provided, according to the DDRD UC(s) to enact. Note that measurements made in the experiment are repeatable in real-world projects; if you disagree with these results, you could repeat then the measurements in your own organization.

However, we expect that the use of a tailored DDRD would reduce the required time to enact a DDRD UC because the readers have to read less information. We also expect that a tailored DDRD would not affect the correctness since from one side it may omit relevant information (i.e. negative influence) from the other side it may omit useless information which would provoke readers confusion (i.e. positive influence). Rough data analyses on such aspects seem to validate our expectations. However, we defer to future works a rigorous investigation on the impact of a tailored DDRD on the correctness and required time for enacting DDRD UC(s).

7. Conclusion

The contribution of the present PhD Dissertation consists in a characterization of existing Software Architecture methods rather than in a proposal of one more method for architectural design. In particular, the work that underlines this dissertation has provided a toolbox of software architecture design methods, from which software architects can select the best method to apply, according to the application context.

By using evidence-based approach, this dissertation has been showing that the goodness of software engineering methods (tool, technique, etc.), including software architecture methods, varies based on the peculiarities of the application context.

In this dissertation, we have characterized the available architectural methods by means of empirical studies, surveys, case studies, and manly controlled experiments with Master students.

Unfortunately, the application of empirical methods on software architecture includes some troubles. A further contribution of the present dissertation is a characterization of the available empirical methods by exposing their levels of challenges that researchers have to face when applying empiricism to software architecture. Such a proposed characterization should help to increase both the number and validity of software architecture empirical studies by allowing researchers to select the most suitable empirical method(s) to apply (i.e. the one with minor challenges), based on the application contexts (e.g. available software applications, architects, reviewers). However, in our view, in order to provide high levels of conclusion and internal validity, empirical methods for software architecture should be oriented to take advantage of both quantitative and qualitative data. Moreover, based on the results from two controlled experiments, those challenges might 1) highly influence the results of the empirical studies, and 2) be faced by empiricists' cleverness, when conducting evidence-based software architecture investigations.

Architecting software system is a complex job that encompasses several activities; this dissertation focused on four families of activities: Designing software architecture, Resolving architectural tradeoffs,

Documenting design decisions, and Enacting empirical studies on software architecture (as just described).

Regarding resolution of architectural tradeoffs, based on our reviews of decision making techniques that literature already proposed, we realized that no decision-making technique can be considered either as the best or better than another one; each technique has intrinsically its own level of complexity and proneness to specific problems. Since we cannot decide in advance what degree of complexity of modeling is sufficient, instead of proceeding by trial and error, we offered guidelines on which complexity to emphasize for avoiding specific problem(s). Our key idea was to expose and organize in a useful way, namely by a characterization schema, in what extent each decision-making technique is prone to specific problems. In this way, it becomes a quality attribute of the decision-making technique the level of proneness of a specific technique to fall into specific troubles. Furthermore, we situated - into the proposed characterization schema - eighteen different decision-making techniques, which the literature already proposed in the domains of architecture design, COTS selection, and release planning. Such localization validates the completeness of the proposed characterization schema, and it provides a useful reference for analyzing the state of the art

Regarding software architecture design, the dissertation tried to answer to following question: “Do actual software architecture design methods meet architects needs?” To do so, we provide a characterization of the available methods by defining nine categories of software architects’ needs, proposing an ordinal scale for evaluating the degree to which a given software architecture design method meets the needs, and then applying this to a set of software architecture design methods. Based on results from the present study, we argue that there are two answers to this question: a) Yes, they do. In fact, we showed that one or more software architecture design methods available are able to meet each individual architect needs that we considered. b) No, they do not. In fact, we showed that there is no software architecture design method that is able to meet any tuple of seven or more needs, which means that there is still some work to do to improve software architecture design methods to actually help architects. In order to provide directions for software architecture design method

improvement, we presented couples of needs, and triplets of needs that actual software architecture design methods are unable to meet. Moreover, an architect can use such a characterization to choose the software architecture design method that better meets his needs.

Regarding design decision documentation, we conducted a controlled experiment for analyzing the impact of documenting design decisions rationale on effectiveness and efficiency of individual/team decision-making in presence of requirement changes. Main results show that, for both individual and team-based decision-making, effectiveness significantly improves, while efficiency remains unaltered, when decision-makers are allowed to use, rather not use, the proposed design rationale documentation technique. Being sure that documenting design decisions rationale can give help, we argued why it is not used in practice and what we can do to facilitate its usage. Previous design decisions rationale documentation methods aimed at maximizing the consumer (documentation reader) benefits by forcing the producer (documentation writer) to document all the potential useful information; they eventually ran into too many inhibitors to be used in practice. In this dissertation, we have been proposing a value-based approach for documenting the reasons behind design decision, which is based on a priori understanding of who will benefit later on, from what set of information, and in which amount. Such a value-based approach for documenting the reasons behind design decision offers means to mitigate all the known inhibitors; it is based on the hypothesis that the set of required information depends on the purpose (use case) of the documentation. In order to validate such a hypothesis, we ran an experiment in a controlled environment, employing fifty subjects, twenty-five decisions, and five different purposes (documentation use case) of the documentation. Each subject practically used the documentation to enact all the given (five) documentation use case(s) by providing an answer and a level of utility for each category of information in the provided documentation. Both descriptive and statistical results confirm our expectancies that the level of utility, related to the same category of information in the design decision rationale documentation, significantly changes according to the purpose of the documentation. Such result is novel and implies that the consumer of the rationale documentation requires, or not, a specific category of information according the specific purpose of the documentation.

Consequently, empirical results suggest that the producer can tailor the design decision rationale documentation by including only the information required for the expected purposes of the documentation. This result demonstrates the feasibility of our proposed value-based approach for documenting the reasons behind design decision.

8. References

Abdelnabi, Z., Cantone, G., Ciolkowski, M., and Rombach, D. 2004. Comparing Code Reading Techniques Applied to Object-Oriented Software Frameworks with Regard to Effectiveness and Defect Detection Rate. Proceedings of the 2004 International Symposium on Empirical Software Engineering. IEEE Computer Society.

Al-Naeem, T., Gorton, I., Babar, M. A., Rabhi, F., and Benatallah, B. 2005. A quality-driven systematic approach for architecting distributed software applications. Proceedings of the 27th international conference on Software engineering. St. Louis, MO, USA. ACM Press.

Ali Babar, M. and Gorton, I. 2007. A Tool for Software Architecture Knowledge Management. Workshop on SHAring and Reusing architecture knowledge . Architecture, Rationale, and Design Intent, Colocated at the 29th Int. Conference on Software Engineering (ICSE 2007). Minneapolis, USA.

America, P., Obbink, H., and Rommes, E. 2003. Multi-View Variation Modeling for Scenario Analysis. Fifth International Workshop on Product Family Engineering (PFE-5). Siena, Italy. Springer-Verlag.

Andersson, J. 2000. A survey of multiobjective optimization in engineering design. LiTH-IKP-R-1097, Linkoping, Sweden: Department of Mechanical Engineering, Linkoping University.

Andrews, A., Mancebo, E., Runeson, P., and France, R. 2005. A Framework for Design Tradeoffs Software Quality Journal 13 (4), 28.

Arisholm, E., Gallis, H., Dyba, T., and Sjoberg, D. 2007. Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise, IEEE Transaction on Software Engineering 33 (2), 65-86.

Avgeriou, P., Grisham, P. S., Kruchten, P., Lago, P., and Perry, D. E. 2007. 2nd International Workshop on SHAring and Reusing architectural Knowledge - Architecture, rationale, and Design Intent (SHARK/ADI 2007).

Azar, J., Smith, R. K., and Cordes, D. 2007. Value-Oriented Requirements Prioritization in a Small Development Organization, Software, IEEE 24 (1), 32-37.

Babar, M. A., Gorton, I. I., and Jeffery, R. 2005. Capturing and Using Software Architecture Knowledge for Architecture-Based Software Development. Proceedings of the Fifth International Conference on Quality Software. IEEE Computer Society.

Babar, M. A., Zhu, L., and Jeffery, R. 2004. A framework for classifying and comparing software architecture evaluation methods. Australian Software Engineering Conference, 2004. Proceedings. .

Bandor, M. S. 2006. Quantitative Methods for Software Selection and Evaluation. CMU/SEI-2006-TN-026.

Basili, V. 1986. Experimentation in Software Engineering, IEEE Transaction on Software Engineering 12 (7), 10.

Basili, V. and Briand, L. C. Empirical Software Engineering, An International Journal.

Basili, V., Caldiera, G., and Rombach, D. 1994. Goal/Question/Metric Paradigm, Encyclopedia of Software Engineering 1 (John Wiley & Sons), 528-532.

Basili, V. R. 1996. The role of experimentation in software engineering: past, current, and future. Proceedings of the 18th international conference on Software engineering. Berlin, Germany. IEEE Computer Society.

Bass, L., Clements, P., and Kazman, R. 2003. Software Architecture in Practice. 2nd Addison-Wesley. Reading, MA.

Bellotti, V. and Bly, S. 1996. Walking away from the desktop computer: distributed collaboration and mobility in a product design team. Proceedings of the 1996 ACM conference on Computer supported cooperative work. Boston, Massachusetts, United States. ACM Press.

Berander, P. and Andrews, A. 2005. Requirements Prioritization, in. A Aurum, and Wohlin, C., Engineering and Managing Software Requirements Verlag, Berlin, Germany.

Biffl, S., Aurum, A., Bohem, B., Erdogmus, H., and Grünbacher, P. 2005. Value-Based Software Engineering. Springer.

Birk, A. and Tauz, K. 1998. Knowledge management of software engineering lesson learned. IESE.

Blin, M.-J. and Tsouki, A. 2001. Multi-Criteria Methodology Contribution to the Software Quality Evaluation. Kluwer Academic Publishers.

Boehm, B. and Turner, R. 2005. Management Challenges to Implementing Agile Processes in Traditional Development Organizations.

Boehm, B. W. 1981. Software Engineering Economics Prentice Hall PTR Advances in computing science and technology. Prentice-Hall. Englewood Cliffs, N.J.

Booch, G. 2006. Goodness of Fit, IEEE Software 23 (6), 14-15.

Booch, G. 2007. The Irrelevance of Architecture.

Borghoff, U. M. and Schlichter, J. H. 2000. Computer-Supported Cooperative Work: Introduction to Distributed Applications. Springer-Verlag New York, Inc.

Bratthall, L., Johansson, E., and Regnell, B. 2000. Is a Design Rationale Vital when Predicting Change Impact? A Controlled Experiment on Software Architecture Evolution. International conference on product focused software process improvement. Oulu , Finland.

Burge, J. and Brown, D. 1998. Design Rationale Types and Tools.

Capilla, R., Nava, F., Sandra Perez, and Dueñas, J. C. 2006. A web-based tool for managing architectural design decisions.

Cavanaugh, C. P. and Polen, S. M. April 2002. Add Decision Analysis to Your COTS Selection Process, The Journal of Defense Software Engineering.

Choppy, C. and Reggio, G. 2004. Using UML for Problem Frame Oriented Software Development. 13th International Conference on Intelligent & Adaptive Systems and Software Engineering.

Chung, L., Gross, D., and Yu, E. 1999. Architectural Design to Meet Stakeholder Requirements, in. P Donohue, Software architecture Kluwer Academic San Antonio, Texas, USA

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. 2002. Documenting Software Architectures: Views and Beyond. Addison-Wesley. Boston.

Clements, P. and Northrop, L. 2002. Software Product Lines: Practice and Patterns. Addison-Wesley. Boston.

Conklin, J. and Begeman, L. M. 1988. gIBIS: a hypertext tool for exploratory policy discussion, ACM Trans. Inf. Syst. 6 (4), 303-331.

Dabous, F. T. and Rabhi, F. A. 2006. A framework for evaluating alternative architectures and its application to financial business processes. Australian Software Engineering Conference, 2006.

De Marco, T. 1986. Controlling Software Projects. PH PTR. New York.

Ding, L. and Medvidovic, N. 2001. Focus: A Light-Weight, Incremental Approach to Software Architecture Recovery and Evolution. Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01) - Volume 00. IEEE Computer Society.

Dobrica, L. and Niemelä, E. 2002. A survey on software architecture analysis methods, IEEE Transactions on Software Engineering 28 (7), 638-653.

Doyle, L. H. 2003. Synthesis through meta-ethnography: paradoxes, enhancements, and possibilities Qualitative Research 3 (3), 24.

Eguiluz, H. R. and Barbacci, M. R. 2003. Interactions Among Techniques Addressing Quality Attributes. CMU/SEI.

Egyed, A. and Wile, D. S. 2006. Support for managing design-time decisions, Software Engineering, IEEE Transactions on 32 (5), 299-314.

Erdogmus, H. and Favaro, J. 2002. Keep Your Options Open: Extreme Programming and the Economics of Flexibility. Addison-Wesley. Extreme Programming Perspectives. LWe al.

Falessi, D. 2006. Materials and data concerning training sessions and experiment regarding DDRD.

Falessi, D. and Becker, M. 2006. Documenting Design Decisions: A Framework and its Analysis in the Ambient Intelligence Domain. BelAmI-Report 005.06/E, Fraunhofer IESE.

Falessi, D., Becker, M., and Cantone, G. 2006a. Design decision rationale: experiences and steps ahead towards systematic use.

Falessi, D., Cantone, G., and Becker, M. 2006b. Documenting Design Decision Rationale to Improve Individual and Team Design Decision Making: An Experimental Evaluation. International Symposium on Empirical Software Engineering. Rio De Janeiro, Brazil.

Falessi, D., Cantone, G., and Kruchten, P. 2007. Do Architecture Design Methods Meet Architects' Needs? Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture. Mumbai, India. IEEE Computer Society.

Ferrari, R. and Madhavji, N. H. 2007. The Impact of Requirements Knowledge and Experience on Software Architecting: An Empirical Study. Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture. Mumbai, India. IEEE Computer Society.

Fichman, R. G. and Kemerer, C. F. 1992. Object-Oriented and Conventional Analysis and Design Methodologies.

Flyvbjerg, B. 2006. Five Misunderstandings About Case Study Research, Qualitative Inquiry 12 (2).

Fowler, M. 1997. Analysis patterns: reusable objects models. Addison-Wesley Longman Publishing Co., Inc.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc.

Gilb, T. and Brodie, L. 2005. Competitive engineering: a handbook for systems engineering, requirements engineering, and software engineering using Planguage. Elsevier Butterworth Heinemann. Oxford.

Glover, F. and Laguna, F. 1997. Tabu Search. Kluwer Academic Publishers.

Goldberg, D. E. 1989. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc.

Grudin, J. 1994. Groupware and social dynamics: eight challenges for developers.

Grüenbacher, P., Egyed, A., and Medvidovic, N. 2003. Reconciling Software Requirements and Architectures with Intermediate Models, Journal on Software and System Modeling.

Harman, M. 2007. The Current State and Future of Search Based Software Engineering. 2007 Future of Software Engineering. IEEE Computer Society.

Harman, M. and Jones, B. 2001. Search-based software engineering, Information & Software Technology 43 (14), 833-839.

Harrison, N. B. and Avgeriou, P. 2007. Leveraging Architecture Patterns to Satisfy Quality Attributes First European Conference on Software Architecture. Aranjuez, Spain. Springer.

Heindl, M. and Biffl, S. 2005. A case study on value-based requirements tracing. Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering. Lisbon, Portugal. ACM Press.

Herbsleb, J. D. and Grinter, R. E. 1999. Architectures, Coordination and Distance: Conway's Law and Beyond, IEEE Software 16 (5), 63-70.

Highsmith, J. and Cockburn, A. 2001. Agile Software Development: The Business of Innovation.

Hofmeister, C., Kruchten, P., Nord, R. L., Obbink, H., Ran, A., and America, P. 2007. A general model of software architecture design derived from five industrial approaches.

Hong, S., Goor, G. v. d., and Brinkkemper, S. 1993a. A formal approach to the comparison of object-oriented analysis and design methodologies. International Conference on System Sciences. Hawai.

Hong, S., van den Goor, G., and Brinkkemper, S. 1993b. A formal approach to the comparison of object-oriented analysis and design methodologies. System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on.

Host, M., Wohlin, C., and Thelin, T. 2005. Experimental context classification: incentives and experience of subjects. Proceedings of the 27th International Conference on Software Engineering.

http://www.di.univaq.it/ase2008/. International Conference on Automated Software Engineering.

http://www.iese.fraunhofer.de/fhg/iese/index.jsp. Fraunhofer Institute for Experimental Software Engineering.

Hunter, J. E. and Schmidt, F. L. 2004. Methods of Meta-Analysis: Correcting Error and Bias in Research Findings 2Sage Publications.

IEEE. 1990. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries : 610

Ieee. 2000. IEEE std 1471:2000--Recommended practice for architectural description of software intensive systems. IEEE. Los Alamitos, CA.

International Symposium on Empirical Software Engineering and Measurement 2007.

J. Karlsson, C. Wohlin, and B. Regnell. 1997. An Evaluation of Methods for Prioritizing Software Requirements, Journal of Information and Software Technology 39 (14-15), 939-947.

Jacobson, I., Griss, M., and Jonsson, P. 1997. Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley. Reading, MA.

Jansen, A. and Bosch, J. 2004. Evaluation of Tool Support for Architectural Evolution. Proceedings of the 19th IEEE international conference on Automated software engineering. IEEE Computer Society.

Jansen, A. and Bosch, J. 2005. Software Architecture as a Set of Architectural Design Decisions. 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 5). Pittsburgh. IEEE CS.

Jedlitschka, A., Ciolkowski, M., and Pfahl, D. 2007. Reporting Experiments in Software Engineering, Advanced Topics in Empirical Software Engineering (to appear).

Jones, C. 1994. Assessment and Control of Software Risks. P Hall.

Juristo, N. and Moreno, A. M. 2006. Basics of Software Engineering Experimentation Springer.

Kano. 1993. A special issue on Kano's methods for understanding customer-defined quality, The Center for Quality Management Journal 2 (4), 3-35.

Karlsson, L., M.Host, and Regnell, B. 2006. Evaluating the practical use of different measurement scales in requirements prioritisation. Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering. Rio de Janeiro, Brazil. ACM Press.

Karsenty, L. 1996. An empirical evaluation of design rationale documents. Proceedings of the SIGCHI conference on Human factors in computing systems: common ground. Vancouver, British Columbia, Canada. ACM Press.

Kazman, R., Carriere, S. J., and Woods, S. G. 2000. Toward a discipline of scenario-based architectural engineering, Annals of Software Engineering 9 (1-4), 5-33.

Kazman, R., Jai, A., and Klein, M. 2001. Quantifying the costs and benefits of architectural decisions. Proceedings of the 23rd International Conference on Software Engineering, 2001. (ICSE 2001). .

Keeney, R. L. and Raiffa, H. 1976. Decisions with multiple objectives : preferences and value tradeoffs. Wiley. Wiley series in probability and mathematical statistics. New York.

Kim, J. and Lerch, F. J. 1992. Towards a model of cognitive process in logical design: comparing object-oriented and traditional functional decomposition software methodologies. Proceedings of the SIGCHI conference on Human factors in computing systems. Monterey, California, United States. ACM Press.

Kitchenham, B. A. 1996. Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods.

Kitchenham, B. A., Dyba, T., and Jorgensen, M. 2004. Evidence-Based Software Engineering. Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society.

Kitchenham, B. A., Pickard, L., and Pfleeger, S. L. 1995 Case studies for method and tool evaluation, IEEE Software 12 (4).

Klein, G. 1999. Sources of Power: How People Make Decisions The MIT Press. Cambridge, Massachussets.

Kontio, J. 1996. A case study in applying a systematic method for COTS selection.

Kraut, R. E. and Streeter, L. A. 1995. Coordination in software development.

Kruchten, P. 1995a. The 4+1 View Model of Architecture, IEEE Software 12 (6), 45-50.

Kruchten, P. 1995b. Mommy, Where Do Software architectures Come from? 1st International Workshop on Architectures for Software Systems (IWASS1). Seattle, WA.

Kruchten, P. 1999. The Software Architect, and the Software Architecture Team, in. P Donohue, Software Architecture Kluwer Academic Publishers, Boston.

Kruchten, P. 2003. The Rational Unified Process: An Introduction 3rd Addison-Wesley Professional.

Kruchten, P. 2004. An ontology of architectural design decisions in software intensive systems. In 2nd Groningen Workshop on Software Variability.

Kruchten, P., Lago, P., and van Vliet, H. 2006a. Building up and Reasoning about Architectural Knowledge. 2nd International Conference on the Quality of Software Architectures. Vaesteras, Sweden. LNCS 4214.

Kruchten, P., Lago, P., van Vliet, H., and Wolf, T. 2005. Building up and Exploiting Architectural Knowledge.

Kruchten, P., Obbink, H., and Stafford, J. 2006b. The past, present and future for software architecture, IEEE Software 23 (2), 2-10.

Laarhoven, P. J. M. and Aarts, E. H. L. 1987. Simulated annealing: theory and applications. Kluwer Academic Publishers.

Lee, J. 1997. Design Rationale Systems: Understanding the Issues, IEEE Expert: Intelligent Systems and Their Applications 12 (3), 78-85.

Li, B., Zeng, G., and Lin, Z. 1999. A domain specific software architecture style for CSCD system.

Lozano-Tello, A. and Gómez-Pérez , A. 2002. BAREMO: how to choose the appropriate software component using the analytic hierarchy process. Proceedings of the 14th international conference on Software engineering and knowledge engineering. Ischia, Italy. ACM Press.

MacLean, A., M. Young, R., M. E. Bellotti, V., and P. Moran, T. 1996. Questions, options, and criteria: elements of design space analysis, in. Design rationale: concepts, techniques, and use Lawrence Erlbaum Associates, Inc.

Maranzano, J. F., Rozsypal, S. A., Zimmerman, G. H., Warnken, G. W., Wirth, P. E., and Weiss, D. M. 2005. Architecture Reviews: Practice and Experience, IEEE Software 22 (2), 34-43.

Martin, R. C. 2003. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR.

Mohamed, A., Ruhe, G., and Eberlein, A. 2005. Decision support for customization of the COTS selection process. Proceedings of the second international workshop on Models and processes for the evaluation of off-the-shelf components. St. Louis, Missouri. ACM Press.

Moore, M., Kazman, R., Klein, M., and Asundi, J. 2003. Quantifying the value of architecture design decisions: lessons from the field. Proceedings of the 25th International Conference on Software Engineering. Portland, Oregon. IEEE Computer Society.

Ncube, C. and Maiden, N. 1999. PORE: Procurement Oriented Requirements Engineering Method for the Component-Based Systems Engineering Development Paradigm. in Proceedings of the 2nd International Workshop on CBSE (in conjunction with ICSE'99). Los Angeles, USA.

Neale, D. C., Carroll, J. M., and Rosson, M. B. 2004. Evaluating computer-supported cooperative work: models and frameworks. Proceedings of the 2004 ACM conference on Computer supported cooperative work. Chicago, Illinois, USA. ACM Press.

Nuseibeh, B. 2001a. Weaving Together Requirements and Architectures, IEEE Computer 34 (3), 115–117.

Nuseibeh, B. 2001b. Weaving Together Requirements and Architectures, Computer 34 (3), 115-117.

Obbink, H., Kruchten, P., Kozaczynski, W., Hilliard, R., Ran, A., Postema, H., Lutz, D., Kazman, R., Tracz, W., and Kahane, E. 2002. Report on Software Architecture Review and Assessment (SARA), Version 1.0. At http://philippe.kruchten.com/architecture/SARAv1.pdf.

OMG. 2003. MDA Guide Version 1.0.1. http://www.omg.org/docs/omg/03-06-01.pdf.

Perry, D. E., Staudenmayer, N., and Votta, L. G. 1994. People, Organizations, and Process Improvement.

Poppendieck, M. and Poppendieck, T. D. 2007. Implementing lean software development : from concept to cash. Addison-Wesley. The Addison-Wesley signature series. London.

Reeves, J. W. 1992. What Is Software Design?, C++ 2 (2).

Remagnino, P., Foresti, G. L., and Ellis, T. 2005. Ambient intelligence : a novel paradigm. Springer. New York.

Roshandel, R., Schmerl, B., Medvidovic, N., Garlan, D., and Zhang, D. 2004. Understanding Tradeoffs among Different Architectural Modeling Approaches. Proceedings of the Fourth Working

IEEE/IFIP Conference on Software Architecture (WICSA'04) - Volume 00. IEEE Computer Society.
Rozanski, N. and Woods, E. 2005. Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives. Addison-Wesley. Boston.
Ruhe, G. Software Engineering Decision Support Laboratory.
Ruhe, G. 2003. Software Engineering Decision Support, Special Issue International Journal of Software Engineering and Knowledge Engineering 13 (5).
Ruhe, G. and Saliu, M. O. 2005. The art and science of software release planning, Software, IEEE 22 (6), 47-53.
Saaty, T. L. and Saaty, T. L. F. o. d. t. 2000. Fundamentals of decision making and priority theory with the analytic hierarchy process. RWS Publications. Pittsburgh, Pa.
Seaman, C. 1999. Qualitative Methods in Empirical Studies of Software Engineering, IEEE Transaction on Software Engineering 25, 557-572.
Seaman, C. B. and Basili, V. R. 1997. Communication and organization in software development: an empirical study.
SEI. 2007. Published Software Architecture Definitions. http://www.sei.cmu.edu/architecture/published_definitions.html .
Sharble, R. C. and Cohen, S. S. 1993. The object-oriented brewery: a comparison of two object-oriented development methods.
Shaw, M. 2003. Writing good software engineering research papers: minitutorial. Proceedings of the 25th International Conference on Software Engineering. Portland, Oregon. IEEE Computer Society.
Shaw, M. and Clements, P. 2006. The Golden Age of Software Architecture.
Shaw, M. and Garlan, D. 1996. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall. Upper Saddle River, NJ.
Shull, F., Seaman, C., and Zelkowitz, M. 2006. Victor R. Basili's Contributions to Software Quality.
Shum, B. and Hammond, N. 1994. Argumentation-Based Design Rationale: What Use at What Cost?, International Journal of Human-Computer Studies 40 (4), 603 - 652

Simon, H. 1996a. The Sciences of the Artificial 3The MIT Press. Cambridge, Mass.
Simon, H. A. 1996b. The Sciences of the Artificial Third TM Press.
Sjøberg, D. I. K., Dybå, T., and Jørgensen, M. 2007. The Future of Empirical Methods in Software Engineering Research. presented at Future of Software Engineering -- 29th International Conference on Software Engineering (ICSE 2007). Minneapolis, USA. IEEE Computer Society.
Smolander, K. 2002. Four Metaphors of Architecture in Software Organizations: Finding out The Meaning of Architecture in Practice. International Symposium on Empirical Software Engineering (ISESE 2002). Nara, Japan.
Sol, H. 1991. A Feature Analysis of Information Systems Design Methodologies Through Process Modelling. International Conference on Software Process. Los Alamitos.
Song, X. and Osterweil, L. J. 1992. Toward Objective, Systematic Design-Method Comparisons.
Song, X. and Osterweil, L. J. 1994. Experience with an Approach to Comparing Software Design Methodologies.
Svahnberg, M. and Wohlin, C. 2005. An Investigation of a Method for Identifying a Software Architecture Candidate with Respect to Quality Attributes, Empirical Software Engineering 10 (2), 149-181.
Svahnberg, M., Wohlin, C., Lunberg, L., and Mattsson, M. 2003. A Quality-Driven Decision Support Method for Identifying Software Architecture Candidates, International Journal of Software Engineering and Knowledge Engineering 13 (5), 28.
Tang, A., Babar, M. A., Gorton, I., and Han, J. 2007. A Survey of Architecture Design Rationale, Journal of Systems & Software 79 (12), 1792-1804
Tang, J. C. 1991. Findings from observational studies of collaborative work.
The-Standish-Group. 1995. CHAOS Report 1995. www.standishgroup.com.
Triantaphyllou, E. 2004. Multi-Criteria Decision Making Methods: A comparative Study Springer. Applied Optimization. PM Parlos.
Tyree, J. and Akerman, A. 2005. Architecture Decisions: Demystifying Architecture, IEEE Software 22 (2), 19-27.

Van der Ven, J. S., Jansen, A., Avgeriou, P., and Hammer, D. K. 2006. Using Architectural Decisions. 2nd International Conference on the Quality of Software Architectures. Västerås, Sweden

van Vliet, H. and Hammer, D. 2005. GRIFFIN: a GRId For inFormatIoN about architectural knowledge.

Vegas, S. and Basili, V. 2005. A Characterisation Schema for Software Testing Techniques, Empirical Software Engineering 10, 437-466.

Wanyama, T. and Far, B. H. 2005. Towards providing decision support for COTS selection. Canadian Conference on Electrical and Computer Engineering, 2005.

Wieringa, R. 1998. A survey of structured and object-oriented software specification methods and techniques.

Wikipedia. Randomized controlled trial. http://en.wikipedia.org/wiki/Randomized_controlled_trial (http://en.wikipedia.org/wiki/Randomized_controlled_trial).

Wikipedia. http://en.wikipedia.org/wiki/Publication_bias. Publication bias.

Williams, L. and Upchurch, R. L. 2001. In support of student pair-programming. Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education. Charlotte, North Carolina, United States. ACM Press.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslen, A. 2000. Experimentation in Software Engineering: an Introduction. Springer.

Wu, J., Graham, T. C., and Smith, P. W. 2003. A Study of Collaboration in Software Design. Proceedings of the 2003 International Symposium on Empirical Software Engineering. IEEE Computer Society.

Yin, R. K. 2002. Case Study Research: Design and Methods Third Edition Sage Publications Inc.

Zelkowitz, M. V. and Wallace, D. R. 1998. Experimental models for validating technology, Computer 31 (5), 23-31.