**Design Decision Rationale:**

**Experiences and Steps Ahead Towards Systematic Use**

Davide Falessi  
Univ. of Roma "Tor Vergata", DISP  
Rome - Italy  
+39 0672597942  
falessi@ing.uniroma2.it

Martin Becker  
Fraunhofer IESE  
Kaiserslautern - Germany  
+49 63168002246  
martin.becker@iese.fraunhofer.de

Giovanni Cantone  
Univ. of Roma "Tor Vergata", DISP  
Rome - Italy  
+39 0672597392  
cantone@uniroma2.it

---

**ABSTRACT**

Design decisions crucially influence the success of every software project. While the resulting design is typically documented quite well, the situation is usually different for the underlying rationale and decision-making process. Despite being recognized as a helpful approach in general, the explicit documentation of Design Decision Rationale (DDR) is not yet largely utilized due to some inhibitors (e.g., additional documentation effort). Experience with other qualities, e.g., software reusability, evidently shows that an improvement of these qualities only pays off on a large scale and therefore has to be pursued in a strategic, pre-planned, and carefully focused way. In this paper we argue that this also has to be considered for documenting DDR.

The contributions of this paper are: (i) an experience report dealing with documenting DDR, (ii) identification of needs towards more goal-oriented and thus systematic DDR documentation approaches, and (iii) a value-based process for systematic DDR usage. The latter considers crucial aspects of every software engineering practice, e.g., where (project context), who (beneficiary stakeholders), when (DDR type of use), why (sw/business metrics), and how (DDR required information).

The paper is structured as follows: Section 2 sets the scene regarding the relevance of DDR and its documentation (DDDR) as well as approaches followed to this end. To pave the way towards a more systematic usage of DRR, we identify motivators and inhibitors for it in section 4 and draw some conclusions from them. Section 5 proposes a more systematic approach for DDR. The paper closes with a conclusion and an outlook to future work.

---

2. **Background**

In the following, we set the scene regarding the relevance of DDR and DDDR as well as approaches followed to this end.

2.1 **Software Architecture**

The Rational Unified Process® (RUP®) defines software architecture as “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” [12][17].

2.2 **Architectural Analysis and Design**

The purpose of analysis and design is (i) to evolve a robust architecture for the system-to-be, (ii) to transform the system requirements into a design, (iii) to adapt the design to match the implementation environment, designing it for performance, and (iv) to produce a design model that serves as an abstraction of the source code [23].
There are differences between architectural analysis, architectural design, and implementation design with regard to focus and emphasis. Architectural analysis is centered on the notion of architecture-to-be, and addresses the question: “What are the conceptual infrastructure and behaviors we are going to allow for this software application?” [24] Architectural design focuses on system structure (logical view), design mechanisms (patterns), design classes and subsystems, reuse opportunity, and design model organization, and addresses the question: “How are we going to implement this software application?”, while implementation design addresses the question: “How are we going to code this software application?” [30]. Anyway, as different notions are associated with design decisions, it is appropriate to clarify the definitions applied in this paper. A design decision is “a selection of an option among zero or more known and unknown options concerning the implementation, structure, interaction and usability of a software application” [30]. An architectural decision is a type of design decision, which can be defined as “activities related to deployment issues of a software application” [30].

2.3 Architectural Decision

Decision making is critical for the success of any software project. Architectural decision making is a key issue in achieving non-functional requirements.

Independent of the design process followed, there are three main sources of architecture [16]:

- **Reuse.** Architecting is a difficult task. Hence, experience from past systems, e.g., components used, rules, or responsibilities should be reused, if possible. The source of reuse can be earlier versions of the system, another system sharing key characteristics, or architectural patterns [1].
- **Method.** A systematic and conscious approach for bridging the gap between requirements and software architecture.
- **Intuition.** An architectural decision is made without conscious reasoning. Obviously such an approach is error prone. As a consequence, validation and verification techniques have to be applied extensively.

Architecture decision-making is for several reasons a difficult task. These include [11]:

- 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. This implies that architectural decisions have to be made based upon sometimes vague requirements.
- Certain types of requirements cannot be understood in the early development phases [22].
- Stakeholders dealing with software architecture issues view the system from different perspectives. This usually implies conflicting goals, expectations, and terminology used.

2.4 Design Rationale

Commonly, during the architecture development process, decisions are not documented explicitly but are reflected by the models the architects build [28]; consequently, some useful knowledge attached to the decision and its process is lost [18].

There are many definitions of design rationale. According to 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” [19]. Some questions suggest themselves from the definition above: Is it worth documenting the DDR, i.e., developing and maintaining DDR documentation? What information has to be documented? What are the characteristics and the focus of such an additional documentation? What are pros and cons for process changes?

Burge and Brown [7] placed the information to be documented into one of five categories, depending on the focus of the design rationale: argumentation, history, device, process, and active document. Some argument-focused DDR documentation techniques has already been developed to date, e.g., gIBIS [8], DRL [20], and QOC [21].

Three important studies empirically investigated the benefits of using DDR. Karsenty [14] proposed an empirical evaluation concerning the utility of design rationale documents in the maintenance of a nine-month old software project. Brathall et al. [6] presented a controlled experiment to evaluate the importance of DDR when predicting change impact on software architecture evolution. Their results show that DDR clearly improves effectiveness and efficiency. Falessi et al. [10] presented a controlled experiment showing again that effectiveness significantly improves for both individual and team-based decision-making, while efficiency remains unaltered, when decision-makers (fifty in the experiment) are allowed to use, rather not use, the DGA DDR framework (see Section 3.2), in the presence of requirement changes.

2.5 Value-Based Software Engineering

Nowadays, a large amount of software engineering methods, tools, and techniques are applied in a value-neutral setting. In other words, all requirements, use cases, objects, test cases, and defects, etc. are handled similarly, without considering their respective value. The Standish Group CHAOS reports [25] 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” [3].

2.6 Cost Benefit Analysis Method

The SEI presents its Cost Benefit Analysis Method (CBAM) [15] 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, the extend to which it 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 adding the products of the level of fulfillments of each objective and the level of importance of that objective.
3. Experience in Dealing with DDR

In this section, we present experiences that we gained with documenting DDR and lessons learned from them.

3.1 Context

Ambient Intelligence (AmI) is a vision of intelligent environments that react in a sensitive and adaptive way to the presence of humans and objects in order to provide various services to people [26]. AmI emphasizes higher user-friendliness and efficient support for human interactions and user empowerment. Such systems typically are highly distributed, heterogeneous, and hierarchical. New combinations of required qualities, e.g., flexibility and efficiency, interoperability, and safety call for a thorough consideration of design decisions and their impact. This raises specific issues, especially with respect to design and decision making. Moreover, compared to the development of conventional systems, in most cases additional stakeholders with specific knowledge and views are involved in analysis and construction activities, as AmI systems typically consist of many different nodes.

Especially during analysis and design, the stakeholders have to negotiate and trade off their objectives against each other. During implementation, assembling becomes crucial and tricky, as the pieces of the whole system have to be interconnected with each other. Whereas the technological compatibility problem often can be easily managed in the analysis phase and properly followed during the developing phase, it often turns out to be quite difficult to manage quality attributes of systems, like performance and resource constraints, in some small subset of the whole system. Usually, the related decisions show widespread impacts throughout the whole systems and therefore have to be considered by all involved architects and designers. All of them have to negotiate their objectives that interfere with the issues of others. The non-functional AmI characteristics, e.g., mobile, embedded, result in tight resource constraints to be met. Functional requirements, e.g., to assist the everyday life tasks, are feature-intensive. The combination of tight resource constraints and demanding functionality calls for the most advanced technology. Usually, the more advanced a technology is, the less it is standardized, compatible, and tested. As a consequence, in such a context, requirements do change substantially, and some of them can be recognized as inflexible requirements only when a part of the system has been developed. Hence, incremental-iterative process models seem to be typical for current AmI system development.

During the development of an AmI prototype in the BelAmI project [4], we experienced the need for documenting and using design rationale for the reasons mentioned above. Additionally, the following issues motivated us to deal with DDR:

- **Objectives.** What is important for the decision maker, and to what extent? Answering that question helped us to check, whether the objectives are compatible with the expectations of the involved stakeholders who did not participate directly in the decision making process.
- **Origin.** Where do objectives originate from? Which decision maker claims an objective and for what component? Answering this question helped us to check on a finer level of granularity the consistency of objectives from different points of view, that is, objectives of specific nodes of the system as viewed by the decision makers and the other affected stakeholders (for that part of the system), respectively.
- **Tradeoffs.** Which DDR objectives are difficult to achieve in combination? Answering this question helped us to demonstrate potential conflicts in requirements and design alternatives.

These issues drove the development of our DGA DDR Framework, which we will introduce in the following section.

3.2 The Decision Goal and Alternatives DDR Framework

Following the definition of design rationale provided by Lee [19] (see Section 2.4), we document the reasons why a decision has been taken. While trying to apply CBAM (see Section 2.6) as is, the need arose for improving collaboration among design decision makers by refining the grain of requirements traceability and optimizing the use of new technologies. Therefore, we devised a specific DDR documentation technique, which is driven by the decision goals and design alternatives available: the Decision Goal and Alternatives (DGA) DDR technique.

In our DGA technique, the rationale behind a design decision documents the attributes of CBAM. According to DGA, whatever the software context might be, design decisions depend on basic decision goals and inter-design-relationships, as shown in Table 1.

<table>
<thead>
<tr>
<th>Decision Goals</th>
<th>Decision Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional requirements</td>
<td>Non-functional requirements (quality attributes and constraints)</td>
</tr>
<tr>
<td>Business goals</td>
<td>Decision relationships</td>
</tr>
</tbody>
</table>

In our experience, the abovementioned list is complete, i.e., its elements are sufficient for specifying the rationale of any software design decision. In the remainder of this paper, we call them Decision Goals.

DGA aims not only to document decisions already made, but also to support decision makers in making their decisions. In DGA, the entity Decision is refined into two sub-concepts: Decision Type (DT) and Decision Alternative (DA). DT addresses the problem to be solved, e.g., what programming environment to use? DA represents an available option with respect to a DT, e.g., “.NET”. Two insights drove the structure of DGA: (i) the importance of a goal is a DT attribute, (ii) the goal fulfillment is a DA attribute. As a result of this clear separation of concerns, the maintainability of the DDR documentation was increased by avoiding document erosion and improving the efficiency of document maintenance. For instance, a change in technology affects DA only (level of fulfillment), while a change in requirements affects DT only (level of importance). In fact, this separation of concerns distinguishes DGA from other DDR documentation techniques.

According to DGA, DDR documentation consists of two stages, which aim to: (i) understand what to document and (ii) enact the documentation, respectively.

The activities of the first stage consist of refining the project objectives and constraints as well as comprehending which decision relationships are appropriate for the project. The refinement of the decision goals in Table 1 into sub-goals depends on the specific usage contexts. In fact, DGA provides users with
much more than Table 1: a framework for decomposing higher-level goals, preventing lacks and avoiding misunderstanding. A detailed discussion of DGA DDR is not within the scope of this paper; it can be found in a technical paper [9].

The last stage is arranged in phases, one phase for each design decision. The ones that were already made, if any, and the ones that still have to be made. Phases can be enacted in parallel. Each phase is arranged in three sequential blocks of activities, aimed to evaluate the “score” to give to relevant attributes, for instance the priority of each objective, and then of each decision, 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 and explaining motivations; such a score expresses the importance that the objective has for the current DT. The second block includes: (i) describing each alternative of the current DT by providing requested specific information, and then, (ii) for each objective, scoring to what extent the current alternative fulfills this objective, and finally, (iii) for each relationship of the current alternative with alternatives of other DT(s), score to what extent the current alternative depends on each of the related DT, in the designer view. The last block selects the best alternative decision for the current DT and documents the alternative selected for the current decision. Further details about DGA can be found in [9].

3.3 Results
In this paragraph, we describe the results of the employment of the DGA technique in our AmI context. The score of objectives importance ranges from 1 to 5. Table 2 lists the overall importance of each possible objective and offers an interesting overview for decision-makers. For instance, by analyzing the objective with the highest importance (which is “Cost / Effort Short” with the average score of 5), we can deduce that huge importance is placed on meet the cost and effort constraint.

Table 2: Objectives ranked based on their average importance.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Objectives</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cost / Effort Short</td>
<td>5.00</td>
</tr>
<tr>
<td>2</td>
<td>Changeability</td>
<td>4.11</td>
</tr>
<tr>
<td>3</td>
<td>Cost / Effort Long</td>
<td>4.00</td>
</tr>
<tr>
<td>4</td>
<td>Maintainability</td>
<td>2.92</td>
</tr>
<tr>
<td>5</td>
<td>Distributed</td>
<td>2.92</td>
</tr>
<tr>
<td>6</td>
<td>Mobile</td>
<td>2.83</td>
</tr>
<tr>
<td>7</td>
<td>Stability</td>
<td>2.61</td>
</tr>
<tr>
<td>8</td>
<td>Understandability</td>
<td>2.50</td>
</tr>
<tr>
<td>9</td>
<td>Transparency</td>
<td>2.50</td>
</tr>
<tr>
<td>10</td>
<td>Learnability</td>
<td>2.50</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>47</td>
<td>Bandwidth</td>
<td>1.00</td>
</tr>
<tr>
<td>48</td>
<td>Accuracy</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 3 shows the result regarding the importance interrelated with every possible objective. Such an importance is computed by averaging over all the documented decision types that relate to the component, where each decision has the main impact. Analyzing Table 3, it is easy to understand that objectives like heterogeneity, mobility, and distribution are of high importance for the middleware component with respect to other objectives like analyzability and replaceability, which have the main importance for the ICup component.

As already mentioned, due to space constraints, we refer the interested reader to a technical report [9] for results concerning tradeoffs.

4. MOTIVATORS AND INHIBITORS FOR DDR
It is well recognized [27] that DDR bears some inevitable advantages. However, the fact that DDR is often omitted raises the question for the reasons behind this. To reveal the latter, we identify motivators and inhibitors for DDR and draw some conclusions from them afterwards.

As motivators for DDR we consider:
- **Improved communication.** Writing things down helps to improve the communication efficiency among designers and between designers and other stakeholders interested in the design, e.g., product managers, by revealing ambiguities or misinterpretations.
- **Improved design quality.** By documenting DDR, the design process (sequence of decisions, facts considered, etc.) is documented, at least partially. Hence, not only information on the What but also on the How is captured and thus becomes available. At first, this improves the understandability of the design result, as it becomes easier to understand why things are as they are. For instance, DDR can reveal that an obviously advantageous design alternative simply has not been chosen, as the designers were not aware of it during the design or misjudged its impact. Furthermore, the maintainability and changeability of the design is improved, as possible improvements and their impact might already have been considered during the design.
**Improved reusability.** By documenting design alternatives and underlying assumptions, as well as constraints, possible changes to adapt the solution to a new context become easier. Consequently, the solution is more reusable, and one can profit from this in related projects. Besides the design result itself, the design process becomes reusable as well. This is especially of interest if the solution itself cannot be reused in a specific context, but parts of the design process can be. DDRD enables the identification of best practices within the design processes.

**Explicit domain knowledge.** Valuable domain knowledge (what to consider, decision choices, pros and cons, constraints, heuristics) is made explicit by DDRD and thus easier to grasp by other employees.

It has to be noted that not all of the aforementioned benefits can be realized in every software development project. For instance, if a product is not maintained after delivery, the respective company will not draw benefits from an enhanced maintenance due to a sound DDR; or if only one designer is involved in the design, improved communication is not an issue.

**As inhibitors for DDRD we consider:**

- **Additional effort.** As various types of information are, in principle, of interest with respect of design decisions, it takes a considerable amount of time to capture this information in an adequate form (e.g., 15 min / decision in our previous empirical investigations). On the one hand, this increases the overall amount of time spent on design, and on the other hand, this also slows down the design progress and therefore could be experienced as an impediment.

- **Unclear benefit.** Usually, people in charge of documenting and maintaining DDR artifacts (decision-makers) are not the ones that directly benefit from DDRD. In consequence, they are not that motivated. Without a clear interrelation between DDR costs and benefits, the DDRD will not be done adequately.

- **Missing competence.** Techniques and methods of how to do DDRD in an appropriate way are unknown. Furthermore, the task is not supported well by suitable tools.

- **Potential inconsistencies.** DDRD implicitly represents the results of the design. If DDR and the design results are documented and not well linked, potential inconsistencies in case of decision changes may occur.

- **Personal interests.** Experts are often not that interested in making their valuable knowledge explicit – this holds especially for the domain knowledge – as they consider it as their personal property.

From the aforementioned motivators and inhibitors we conclude that despite the various advantages, DDRD is not for free, and it will not pay off in all cases. Hence, a thorough consideration of the costs and benefits related to DDRD is required in different contexts in order to exploit DDR related benefits the best possible way.

Up to now, DDRD methods have been employed independently from the system requirements and business context where they should provide benefits. Consequently, we still have to judge them as value-neutral methods (see Section 2.5). Because such a peculiarity strengthens DDR inhibitors (see Section 4), the question is: What approach should we follow for mitigating the impact of those inhibitors? One of the conjectures we take in consideration and plan to investigate is that injecting value-based software engineering principles into DDR methods should enact a systematic DDR employment and, consequently, mitigate the effects of the inhibitors.

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 [13] [29] have shown that only strategic, pre-planned reuse that considers the features’ value really pays off and allows reuse in the large. This is also one of the key rationales behind Fraunhofer’s PuLSE™ method [2].

Transferring this experience to the DDRD context 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 and costs/risks differ from case to case, the strategy has to be defined anew for different development contexts. In the following section, we propose a more systematic approach to DDRD.

### 5. TOWARDS SYSTEMATIC DDR USE

#### 5.1 Rationale

In order to promote a systematic and strategic DDR use, we propose a method to take crucial aspects of every software engineering practice into account: Where (project context), Who (beneficiary stakeholders), When (DDR type of use), Why (sw/business metrics) and How (DDR required information). The rationale behind this is to determine a priori, who will profit from what information in which amount later on, in order to cope with the additional effort that has to be spend on DDRD. This allows a more value-based documentation of DDR.

#### 5.2 Usage Scenarios and Required DDR Information

We consider a DDR use case (DDRUC) as a scenario where one or more actors utilize DDR in order to achieve a certain advantage while enacting a particular activity in a specific project context. Such a DDRUC can be represented by the following information:

- **Actor:** Actors participating in the DDRUC. In other words, stakeholders who take advantage of the DDR (i.e. payee).

- **Context:** System or industry characteristics where the use case happens. This comprises information regarding the environment and the underlying driver.

- **Activity:** The activity enacted by the payee, in which DDR is used.

- **Advantages:** Product or process metrics that determine the benefit of the DDR usage.

In order to enable systematic use of DDR, we identified a set of thirteen DDRUC listed in Table 4. The list is supposed to be incomplete and thus subject to change, but it is already illustrative and valid to the best of our knowledge.

We used a framework recently proposed in the literature [28] and briefly described in Table 5 as our DDR framework reference. Table 6 shows a traceability matrix, where the information required by a certain DDRUC is traced to the entire set of DDR information (see Table 5). By analyzing the contents of Table 6, we can derive that the amount of effort needed for using DDR heavily depends on the specificities of the DDRUC.
Moreover, Table 6 shows that on average, 60% of DDR information is not used (percentage of empty box).

5.3 Describing a DDR Use Case

In order to explain the meaning and practical usage of Table 4 and Table 6, let us consider use case No. 5 in Table 4, which is a DDRUC that concerns the management of requirement changes. Nowadays changes in the requirements or business goals are becoming more frequent. 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, the business world will not stop for them. In case of a requirements change, it is crucial that managers and architects are able to understand, which decision (and relative artifacts of the system) are still valid and which others have to be re-done (i.e. re-design and/or re-implementation). Traceability among requirements, design decisions, and software artifacts allows managers to easily realize the type of action required for each software artifacts and, consequently, to re-schedule the project activities.
5.4 Process

The flowchart in Figure 1 describes, using MS Visio™ format, the flow of activities (rectangles) and the flow of information produced/consumed (parallelograms) in our value-based process that aims to systematic DDR use. In Figure 1, the first activity (“Scenario selection”) is activated for selecting a scenario that still remains to be analyzed from a predefined DDRUC Database (see Section 5.2). Subsequently, managers analyze information concerning the selected scenario(s), which is:

- **Context:** to figure out the probability that this scenario will occur in the business and system context, \( p(s) \).
- **Metrics:** to realize the benefits that such a use case provides, \( b(s) \).
- **Required DDR information:** to estimate the cost of (amount of effort to spend on) documenting and maintaining DDR, \( c(s) \).

At this point, the adoption of DDR in the current scenario is economically evaluated for return on investment. Since the benefit will be achieved only in case the related scenario occurs, we compute the “utility” as the standard recommended “benefit divided by cost” [5] and multiplying it by the probability that the selected scenario has to occur (i.e. \( u(s) = \frac{p(s) \cdot b(s)}{c(s)} \)). When all the known DDRUC(s) have been analyzed, the managers utilize the list of valuable scenarios to:

- Persuade the stakeholder in charge of the DDR documentation of the importance of such an activity.
- Document and maintain DDR information only of valuable scenarios (one or more).
- Give to payees the information about the DDR activity specified for each of those valuable DDRUC(s), and persuade them to enact them.

Let us underline that the process activities mentioned above are adequate to human capabilities, i.e., they can be performed directly by people, without computer support. Still, an automated tool to support managers in the analysis activities (probability, benefits, and cost estimation) is feasible, and would certainly be helpful. Additionally, such a tool could be designed for reuse of DDR information, hence providing decision making with higher value supports. In fact, when we apply DDR to different DDRUC(s), the information concerning a certain DDRUC can be reused (maybe partially) when considering subsequent DDRUC(s). Of course, an automated tool could help in figuring out the DDR information that maximizes the total utility of those scenarios. The value-based process, that we proposed and sketched above, completely neglects such a reuse perspective, due to difficulties found in dealing with it in the absence of a computer-driven process.

6. CONCLUSION

The paper presented: (i) the Decision, Goal, and Alternatives (DGA) DDR framework, (ii) experience in dealing with DGA, (iii) motivators and inhibitors of using DDR, and (iv) an approach for systematic DDR employment. Such an approach follows value-based software engineering principles and uses a set of scenarios characterized by specific payees, business context and product characteristics, type of DDR activity, benefits (which we evaluated by using a specific quantitative metric), and required DDR information. We provided a set of thirteen scenarios, which is supposed to be incomplete, illustrative, but actually seeded with valid items.

7. FUTURE WORK

In future, we plan to address the following five DDR research issues:

- Investigate the relationships between “Usefulness” and the “Which”, “When”, and “How” of design decision documentation, respectively. In fact, although Karsenty [14] demonstrated the usefulness of documenting design rationale for design reuse and Shum et al. [27] discussed pros and cons of documenting design rationale, we are still missing knowledge concerning those relations.
• Develop a decision-making support tool; besides helping managers who intend to adopt our value-based process for systematic DDR use, such a tool should support us in conducting empirical-evidence-based validation of our proposals.
• Figure out incompatibility issues and which design rationale information is already available, so that we do not need to document them again when using a certain software architecture design method.
• Investigate the relationship between design rationale information and the “type”, “granularity” and “hierarchies” of a decision.
• Support the proposed value-based approach with a metamodel.

8. ACKNOWLEDGEMENTS

The work presented in this paper was partially carried out in the UoRMTV-UoKI-UoMD Intl. Research Cooperation projects, funded by the University of Rome “Tor Vergata”, Italy, and the BelAmI (Bilateral German-Hungarian Research Collaboration on Ambient Intelligence Systems) project, funded by the German Federal Ministry of Education and Research (BMBF), Fraunhofer-Gesellschaft, and the Ministry for Science, Education, Research, and Culture (MWWFK) of Rhineland-Palatinate.

9. REFERENCES

[26] Research Center Ambient Intelligence, http://www.eit.uni-kl.de/ami/