Rationale Management

Stefan Puchner

May 23, 2006

Contents

1 Overview 3

2 Motivation 3

3 Foundations 4

4 History 5

5 Issue Modeling Representation 6

6 Managing Rationales 10

7 Conclusion 10

8 References 11

1 Overview

The purpose of this document is to give an overview of the role of Rationale Management within Software Engineering processes. The first section, Motivation, is a short introduction into how rationale management could be useful. Following, in the section Foundations, some background on rationale management is given. The section History deals with some well known approaches for rationale management. Later on, a possible representation for rationale models is proposed, followed by information about capturing and managing rationales. In the end, a small conclusion that brings together advantages and disadvantages of rationale management is given.

2 Motivation

To clarify the use for rational models, a short example originating from the cooking world might help.

Mary asks John, her husband, why he always cuts off both ends of the meat loaf before putting it in the oven. John responds that he is following his mother's recipe and that he had always see her cut the ends of the loaf. He never really questioned the practice and thought it was part of the recipe. Mary, intrigued by this answer, calls her mother in law to find out more about this meat loaf recipe.

Ann, John's mother, provides more details on the meat loaf cutting, but no culinary justification: she says that she has always trimmed about an inch off each end of the loaf as her mother did, assuming it had something to do with improving the taste.

Mary continues her investigation and calls John's maternal grandmother, Zoe. At first, Zoe is very surprised: she does not cut the ends of the meat loaf and she cannot imagine how such practice could possibly improve the taste. After much discussion, Zoe eventually remembers that, when Ann was a little girl, she used to cook on a much narrower stove which could not accommodate standard sized meat loaves. To work around this problem, she used to cut off about an inch from each end of the loaf. She stopped this practice once she got a wider stove.

(Out off [Bruegge & Dutoit])

In the world of software development it would be much harder for Mary to gain such reasoning information. Due to size and complexity of modern software systems, the person that might have the right information is much harder to identify. It is difficult to understand the design of a bigger system and its implementation just from source code and graphical representations of the system models. A lot of knowledge that is created during the development of a software system is not included in these sources and gets lost over time. This is either due to personnel changes or due to the limited memory capacity of human beings. For example, other possible design solutions that where discussed but not implemented are not documented and get lost over time. Rationale Management means storing all this knowledge that was created during the software development process in an additional model, called a rationale model.

Especially developers new to a software system encounter a difficult task if they need to get familiar with an existing software system. They may not be able to understand why the system was designed the way it is. And, even worse, they may not foresee the effect of changes they might make. For instance, developers might design an output format of a system's data in a way that is more complicated than necessary, to ensure compatibility with legacy systems. A new developer inspecting the code months later might not understand the reasons for this complicated format, because there is no information about the legacy system in the source code or the available models. That is just like Mary not understanding why John cuts off the ends of the ham. This missing understanding might encourage the developer to change the output format to a more intuitive and simpler one, without recognizing the additional knowledge and reasons that led to the original format. Developers that are already familiar with a system might experience similar difficulties. They are usually not able to be familiar with all details of a bigger software system and they might also forget their knowledge over development time. Similarly, John's grandmother Zoe had problems with remembering the reasons for cutting the ends of the ham.

In contrast to that, a developer who has access to an appropriate rationale model could make much more educated decisions. If encountering a point in the system at which she doesn't understand the design, she would be able to comprehend the reasoning that led to the current design by looking it up in the rationale model. There she would find information about *why* the system was designed in that way. But as the reader might already have realized, gathering and externalizing all data of this kind – including problem statements, resolutions and the corresponding arguments – means a huge additional effort in the development process.

3 Foundations

The general meaning of rationale is justification for decisions. Rationale management in the context of software engineering means to externalize knowledge about the software project. Since knowledge is generated by reflecting on issues, externalizing knowledge means to capture information about decision making processes. That means while other system models represent information about *how* a system looks like, the rationale model tells *why* a system looks like it does.

Software developers engage in many decision making processes throughout the day. Most of them are rather short and unconscious, for instance, whether a piece of code should be implemented in an extra method or directly in the caller method. Such decision making processes are nearly impossible to externalize. There are several reasons why this information is hard to capture: on the one hand, because the developer is not aware of the fact that she is producing knowledge she should externalize. And on the other hand, because the quantity of such decisions is so huge that documenting the decision finding would take significantly more time than the actual coding work of the developer. But after all, such decisions are usually not of interest when creating a rationale model, since most simple issues that any experienced developer could solve herself are not of interest. Including such decisions in a rational model would flood the rational model with trivial information that nobody would be interested in later on.

The more important decision making processes take usually much more time and are therefore worth the effort it takes to externalize them. Furthermore more than one person may be involved in these decisions. So they are more conscious and explicitly identified as decision making processes, which is a requirement for storing them in the rationale model. Decisions of this kind may be major design decisions like whether to use a database system or a file system for persistent data storage. The reasoning about such major decisions may be of interest for developers even later on.

In [Fisher et al.] an approach of keeping negotiations rational and avoiding more or less accidental outcomes of negotiations is described. The entity types of a rational negotiation are identified as

- the actual problem the negotiation is about,
- interests the different negotiators try to meet with the decisions,
- proposals that are possible solutions to the problem, and
- arguments that enforce or weaken proposals.

This model of multilateral negotiation can be matched to most kinds of decision making processes, multilateral as well as unilateral. In software development, multilateral decision making processes are generally meetings in which developers discuss different design proposals. Unilateral ones – if only one developer is reflecting over a certain issue – have these different entities as well. In the further discussion, following types will be used for categorizing the different thoughts of decision making processes in software development:

- *issue* as generalization of problem
- *criteria* as generalization of interests
- *proposals* as possible solution to the problem
- arguments to measure how good proposals match the criteria
- *resolution* instead of decision

Every rational decision making process can be described by this model. Splitting a decision making process up into these entities has various advantages. The one addressed by [Fisher et al.] is that the negotiation process is fairer, faster and overall of more benefit for all stakeholders. Humans naturally tend to develop proposals that match their – possibly unconscious – interests too early, and fix them as their position that must be defended at all costs. A more flexible way of negotiation is to split positions into interests (criteria) and proposals. Based on the finding and fixing of the existing interests, proposals that fit much better to these interests than the initial positions can be developed. Such proposals can then be accepted more easily by the different stakeholders. Additionally, this approach helps to impersonalize the proposals and therefore improves the climate between the negotiators.

Another advantage is that more educated decisions are possible, even in unilateral decisions. If the decision maker categorizes his thoughts with the entity types described above, she will be able to have a more neutral view, cleared from her current mood or feelings. The outcome would be a more rational decision.

Finally, there is the advantage that if a decision making process should be externalized into a rationale model, this can be done best with a useful typing. Without different types a lookup in the rational model would require a text search algorithm, which might not lead to the desired information algorithmically. When using types, the thoughts can be linked and structured in a way that reflects the path of reasoning. That way a developer may find out on which resolutions the current issue depends on, and see which criteria are fulfilled by which proposals.

4 History

Starting with the Issue-Based Information System [Kunz & Rittel], researchers proposed several approaches to build rationale models. These approaches include not only the representation of rationales, but the process of capturing them as well. In this section a short introduction to these approaches is given.

Issue-Based Information System (IBIS) was created to address so called wicked problems, meaning problems that can not be solved algorithmically but rather through discussion. IBIS describes issues, positions, and arguments as different entities for rationales. It does not feature resolution and criterion types as discussed above. Criteria are implicitly stated together with proposals in position nodes (Figure 1).

The Questions, Options, and Criteria (QOC, [MacLean et al.]) model is based on IBIS. For the representation, it splits positions up into options and criteria (Figure 2). Therefore, it is more aligned to the model with the five types stated before. QOC and IBIS differ not only in the representation of rationales, but in the process of rationale capturing as well. IBIS proposes a capturing process while the decision making process is ongoing and therefore offers a historical

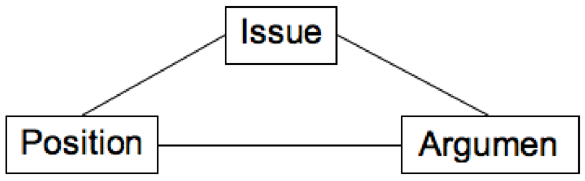

Figure 1:

record of the decision making process. QOC proposes to capture rationales after decisions have been made and externalize the knowledge about the different solutions that were under discussion. This will usually lead to less detail than the former approach.

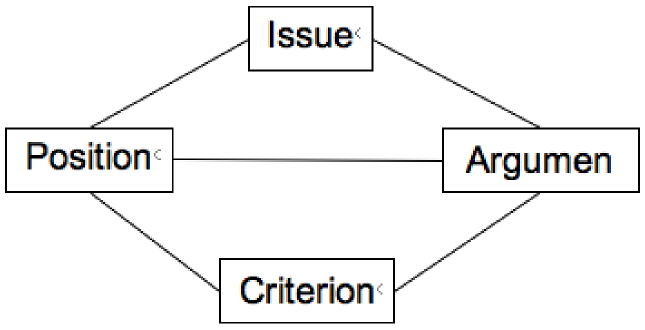

Figure 2:

Another rational model description that is based on IBIS is Decision Representation Language (DRL, [Lee]). It significantly increases the complexity compared to IBIS by featuring seven different types, namely decision problem, alternative, goal, claim, achieve link, procedure, and question.

The NFR Framework [Chung et al.] does not extend IBIS, but proposes a model to describe dependencies between nonfunctional requirements and design decisions, and stores information about different proposals. Like models of the other approaches, an NFR model can be represented as graph. Nonfunctional requirements are called goals and may be decomposed to sub-goals. An NFR graph may also feature so called operationalizing goals. These represent concrete system features in contrast to other goals which are nonfunctional requirements.

5 Issue Modeling Representation

In this section the Issue Modeling approach, as introduced in [Bruegge & Dutoit], is described in more detail than the approaches in the section history (Figure 3). It is based on the IBIS representation model but does not asses the way of capturing rationales. It uses the five entity types that were already described above, namely:

Issue: Problem with no single correct resolution. It contains a subject, a description as question and a status (open or closed). It may be decomposed into sub-issues and may be raised by proposals.

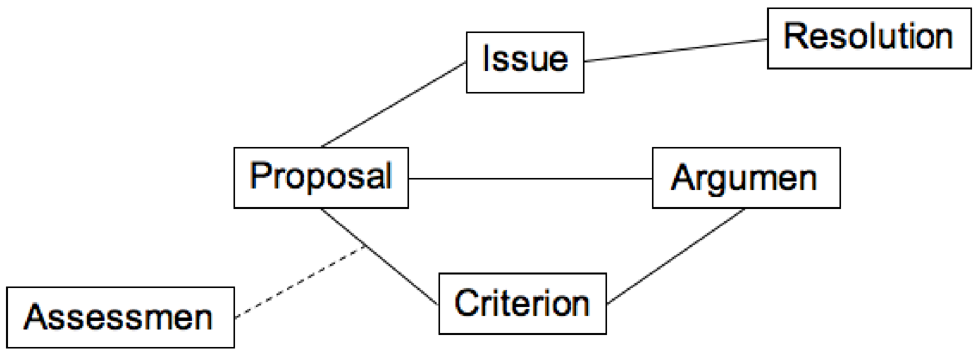

Figure 3:

Proposal: Possible solution to an issue. It contains a subject and a description, but no information about value, advantage, disadvantage. It may not be a good answer to an issue, may address more than one issue, or it may overlap with other issues.

Criterion: A quality a proposal should have. It consists of a subject (phrased positively), a description and is connected to proposals through Assessment association (including value and weight).

Argument: Opinion of a person towards a proposal, criterion, or assessment. It contains a subject, a description and is connected to the entity under discussion through 'is opposed by' or 'is supported by' association.

Resolution: Selected alternative to close an issue. It consists of a subject, a description, and a status (active or obsolete). It may be based on several proposals.

The assessment node in Figure 3 is an association class. In general a criterion may not only be addressed or be not addressed by a proposal. Rather, it is reasonable to define how well the criterion was addressed, for instance on a scale from one to ten. The assessment class holds quantitative information about the relationship between a proposal and a criterion.

The following example will give a better understanding for the issue based representation model (Figure 4). The issue node 'storage' represents the issue: Which storage technology should be used in our system for persistent storage? The two proposal nodes suggest the usage of a database system or the file system for persistent storage. While the proposal 'database' meets the criterion 'flexibility' it fails to meet 'simplicity'; and vice versa for the 'files' proposal. The argument 'extensibility-first' expresses opposition to the 'files' proposal since the resolution should be easily extensible. It is supported by the 'flexibility' criterion. If the 'database' proposal was chosen, a new sub-issue would be opened: Which database system should be used? The resolution node 'storage in database' indicates that the 'database' proposal was selected.

Externalizing knowledge and typing thoughts is the most time intensive task that comes with the creation of a rationale model. This section describes in which areas of software development rationales could be captured, which detail levels of rationale capture can be identified, and which activities are suitable for capturing rationales. Rationales can be captured during diverse phases of software development:

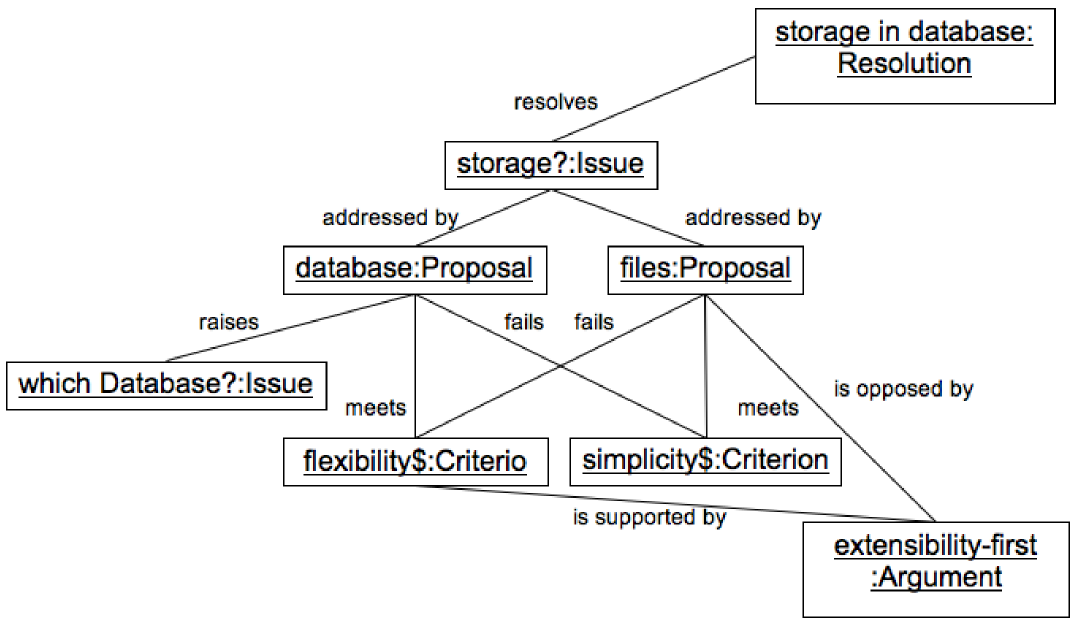

Figure 4:

Requirements elicitation and analysis: During requirements elicitation and analysis, far-reaching nonfunctional and functional requirements are defined. These influence the whole system design. If requirements change during the development process, it would be useful to developers to know about which design decisions where made based on these requirements, so the developer would know where to consider a redesign on the system. Furthermore, the rationales externalized during the requirements elicitation and analysis phase could be useful for defining user acceptance tests. That is because the crucial parts of a system concerning a specific requirement could be identified more easily, if the coherence between the requirement and the design decisions are visible.

System design: Rationales captured during system design can be helpful later on, if parts of the system get changed. In that case a developer could find out which other design decision depend on the one he changed and knows where redesign considerations are necessary.

Project management: Risk management is an important field of project management that can benefit from rationale models. For example, if a project manager externalizes a decision making process about choosing technology A or technology B, she would write down the proposals for each technology and express the risk considerations as arguments and criteria. The project manager may decide for the cheaper technology A that has a high risk of failure, since it is quite new on the market. If it later turns out that technology A is unexpectedly not suitable for the required purpose, the project manager could review the other proposals pertaining to this risk management decision again, and may decide for technology B. Here again, the rationale model shows the other decisions that were influenced by this one.

Integration and testing: During integration and testing phases, a developer could not only find out in which area of the model a conflict occurred, but may also find out which design decisions are responsible for it. This may, on the one hand, help to find more conflicts of this kind, and on the other hand, enable a resolution to the conflict with minimal impact on the rest of the system.

Externalizing rationale knowledge can be done in different degrees of detail and effort. We call the following degrees levels of rationale capture:

No explicit rationale capture: For this level, no additional efforts have to be made to create a rationale model. The rationales just reside where they are anyway; that is in E-mails, memos, on napkins, or in the memories of developers. Of course, such a rationale model is of limited use since developers might forget rationales, napkins are not useful for archiving knowledge, and even electronic texts like E-mails can only be searched by text strings but not by dependencies.

Rationale reconstruction: In this approach rationales get captured after the decision has been made, in a way of documentation. A rationale model captured on this level might not contain other proposals besides the one finally decided on, and does not feature argumentation.

Rationale capture: When applying this level, all rationale information gets externalized. Over time, a huge and interconnected model – for instance represented by a graph – is created. This level of capture requires huge investments, since the capturing process is very time intensive.

Rationale integration: Rationale integration is similar to rationale capture, but the generated rationale model does not just reside beside other models. Rather, the changes on the rationale model induce changes on other models. The rationale model becomes the central model which all other models depend on.

Finally, there are different rationale activities that can be captured. Following are four categories that cover most of the multilateral rationale activities.

Meetings: This is perhaps the most important category, since major design decisions and most difficult issues are often discussed in meetings. The capturing process includes taking the minutes, decomposing the minutes into the different types of the rationale model, and entering this information in the electronic rational model.

Electronic communication: Since electronic communication is increasingly used, the amount of rationale activities based on electronic communication increases as well. To cover this kind of communication in creation of rationale models, users could be encouraged to use a special groupware client that features functionality for capturing rationales.

Changes If changes in system design or requirements occur, the rational model should be updated. That means, for instance, marking the current resolution as out of date and adding the new one, together with new arguments and criteria, if applicable.

Reconstructing rationale This way of rationale model construction is applied when using rational reconstructing as a level of rationale capture. First, decisions are made and later on these decisions get justified in a documentation step.

6 Managing Rationales

If rationale models are implemented in professional software development, it is important that the rationale models be built up meticulously and comprehensively. But a developer usually sees no direct connection between the benefit a rationale model might have for her, and the additional time and effort she spends in creating it. Therefore, developers are not likely to use a rationale model voluntarily. So it remains the responsibility of management to assure that developers really do work with and on rationale models.

One part of this management responsibility is to ensure that the rationale model is easy to access and has high usability for the developers. That way the barrier for using the rationale model can be lowered. Another important factor in managing rationales is the assignment of responsibilities by the manager. For capturing a meeting, you will usually need a minute taker who captures all information that was said. Later, a rationale editor would extract this information from the minutes, type the different parts, and enter it in the rationale model. To ensure the integrity and the completeness of the rational model, a reviewer needs to inspect it regularly and request and add additional information from the developers if it is missing in the rationale model. To ensure that this reviewer is accepted by the other developers and not seen as a kind of controlling big brother, the reviewer must not be manager in this project. Furthermore, the reviewer should be available to the other developers for questions concerning the rationale model. That way, both sides can help each other, resulting in a better atmosphere at work than if just on side depends on the other unidirectionally.

7 Conclusion

Rationale management can help to improve software development dramatically. Firstly, there is the huge advantage of enabling easier and safer changes on existing systems, due to the availability of all knowledge necessary for the changes. Secondly, rationale management helps to make the success of a software project more independent of single developers. Since the knowledge of developers is not solely stored in their own head but accessible to all, a single developer can more easily be exchanged. Finally, rationale management may lead to more rational decisions, because it forces developers to decompose their thoughts and encourages an objective view on issues. It needs to be evaluated for each project if the huge benefits of rationale management balance the great effort that comes with it, and on what level of detail a rationale model should be created. Further research should evaluate how capturing and maintaining rationales can be done efficiently and what kind of rationale model tools provide the highest usability for a given situation.

8 References

Bruegge & Dutoit B. Bruegge & A. Dutoit. Object-Oriented Software Engineering (Second edition, International edition). Pearson Prentice Hall, 2004.

Fisher et al. R. Fisher, W. Ury, & B. Patton. Getting to Yes: Negotiating Agreement Without Giving In (Second edition). Penguin Books. 1991.

Kunz & Rittel W. Kunz & H. Rittel, Issues as elements of information systems (Working Paper No. 131). Institut fuer Grundlagen der Plannung, Universitaet Stuttgart, 1970.

MacLean et al. A. MacLean, R.M. Young, V. Bellotti, & T. Moran, "Questions, Options, and Criteria: Elements of Design Space Analysis," Human-Computer Interaction, 6 (3&4): 201-50, 1996.

Lee J. Lee, "A qualitative decision management system," In P.H Winston & S. Shellard (Eds.), ARTIFICIAL Intelligence at MIT: Expanding Frontiers. Cambridge, MA, MIT Press. Vol 1, 104-33, 1990.