Evaluating Alternatives for Architecture-Oriented Refactoring

Sven Wohlfarth, Matthias Riebisch
Department of Software Systems and Process Informatics
Technical University Ilmenau, 98684 Ilmenau, Germany
{ sven.wohlfarth | matthias.riebisch }@tu-ilmenau.de

Abstract

Refactoring of software systems represents an fundamental way of improving their quality properties. Large-scale refactoring has to be performed at an architectural level to execute such changes for larger systems. Architecture-oriented refactoring requires decisions with multiple, partly contradicting objectives and uncertain consequences. To minimize risks and effort, the decisions about optimal refactoring alternatives have to be performed in a systematic way. In this paper decision theory is adapted to architecture-oriented refactoring. Methods for the evaluation of refactoring alternatives are shown which are applicable even to decisions with multiple and partly uncertain consequences. Furthermore, the complex decision process is structured in a rational way. In an example the effects of an increased quality requirement to architectural evolution are demonstrated.

***Keywords**: Refactoring, Reengineering, Software Architecture, Software Quality, Decision Theory*

1. Introduction

During their use, software systems have to be changed permanently. Frequently, changes are implemented in an incomplete or inconsistent way, leading to a loss of architectural quality, i.e. in terms of maintainability and understandability. This effect is called Architectural Decay. To enable further changes, the architectural quality has to be restored. In other cases, changed non-functional requirements demand in a revision of the architecture. In both cases, a refactoring of this system is necessary, e.g. the architecture has to be changed without changing the behavior of the system.

There are some refactoring steps described in literature, mostly operating at the source code level [1]. For larger-scale changes, the architecture level is more suitable for performing the refactoring because it enables a reduction of the architecture's complexity, e.g. by introducing reference architectures, styles and patterns.

Architecture-oriented refactoring is an approach to restructure the software architecture of a system and to improve its internal software quality. The principal software quality goals for refactoring include understandability, flexibility, dependability and time behavior. The key aspect of refactoring is the preservation of the behavior, e.g. the functional behavior of the software will be unchanged.

A refactoring at the architecture level demands for larger changes than at the source code level. Therefore it is a much more complex, difficult and risky task. One of the critical points is the choice of an optimal refactoring solution from the set of refactoring alternatives [1]. Normally there is more then one suitable alternative. But which of them offers the strongest quality improvement and the lowest effort or risk? The refactoring alternatives have to be evaluated and compared. Furthermore, the objectives of a refactoring are frequently ambiguous, vague or even contradictory. It is difficult for the developer to make a rational choice under consideration of more than two or three factors and objectives.

This paper describes an evaluation methodology for refactoring alternatives based on rational decisions. The well-established decision theory methods will be adapted to refactoring decisions at the architectural level. They are applied to create a system of objectives, to generate refactoring alternatives and to support the choice of a refactoring alternative, which matches the preferences of the decision maker in the best way. Later, the selected refactoring alternative will be implemented and the achievement of the quality objectives will be checked.

This paper is organized as follows: in section 2 the State of the Art of decision theory is introduced in brief. Section 3 introduces the methodology and demonstrates it by a case example. Section 4 draws the conclusions.

2. State of the Art of Decision Theory

Decision theory comprises the prescriptive and the descriptive decision analysis [2]. The intention of descriptive decision analysis is to describe the human decision process on a cognitive level. The focus of the paper is on prescriptive decision analysis, because it helps the decision maker to handle complex decision problems in a rational way.

The prescriptive decision analysis decomposes and evaluates the decision problem [3]. A complex decision problem can be decomposed into fundamental components: the objectives and preferences of the decision maker, the alternatives, their impacts and their consequences.

The critical point in this decision problem is the evaluation of the alternatives [3]. The alternatives are several options or strategies. The decision maker has to choose one alternative, which matches the objectives and preferences on a high level. During the evaluation process, the decision maker has to consider many factors, like the consequences of the alternatives, impact factors and his preferences. Furthermore, some factors can only be determined uncertainly. Decision analysis methods help to evaluate the various alternatives in a rational way. The methods help to quantify the value or the estimated utility of the alternatives and match them with the desired preferences of the decision maker.

The decomposition of the decision problem and the structured evaluation are important factors for rational decision-making. Other rationality requirements will be presented in the following sections.

Decision theory has proven its practical applicability in various contexts, e.g. financial or investment decision problems [4]. The alternatives of financial decision problems are several investment objects e.g. machines, patent licenses. The consequences are even uncertain, because they depend on unpredictable influence factors, e.g. the demand of the customers.

3. Establishing a Decision Process for Architecture-Oriented Refactoring

Architecture-oriented refactoring usually has to solve different competing requirements. The consequences of a decision are mostly uncertain, e.g. because the effort for deriving them is too high for practical cases. These characteristics of refactoring decisions encourage us to apply decision analysis here.

3.1. Requirements for Rational Decisions

To make a rational decision, the decision process has to fulfill two criteria: The procedure of the decision has to be rational and the basis for the decision has to be consistent [3, 5, 6].

The following aspects are relevant for a rational decision process:

- *Information*: It is important, that the decision maker has considered and processed the relevant information. The investigation effort should correspond to the importance of the decision problem. It is not necessary to analyze the whole software system if only a small part has to be improved.
- *Expectation*: The decision maker has to focus on objective expectations about the future.
- *Objectives and preferences*: The objectives and preferences have to be described clearly. Furthermore, they have to be weighted according to the importance of the quality problems.

The second criterion of a rational decision-making is a consistent decision basis. This is determined by the following factors:

- *Future orientation*: The choice between alternatives should be determined by their expected consequences. Former events and factors are irrelevant.
- *Transitivity*: If the decision maker prefers the alternative *a* before *b* and *b* before *c* - then he has to prefer the alternative *a* before *c*.
- *Invariance*: The preferences of the decision maker should be invariant to each other. The factor costs of the refactoring correlates with the refactoring effort for example.

3.2. Creating a System of Objectives

The first phase of the refactoring process is the determination of the objectives. The objectives are the required, but unfulfilled software quality characteristics. The software quality is defined as the totality of characteristics of an entity that bears on its ability to satisfy stated and implied needs [6]. The software quality characterizes the quality of the software product and the engineering process. The product quality is the quality of the software product itself, like understandability or flexibility. The quality of the refactoring process is represented by quality characteristics like efficiency or transparency.

The determination of the quality objectives is intertwined with the analysis and evaluation of the software system and particularly with the analysis of the software architecture. Architecture analysis is a structured way to identify the problematical parts of the architecture (e.g. in components, in sub-systems and in their relations). These parts are the reason for already unfulfilled quality objectives. There is a broad field of architecture analysis methods. A well-accepted method is the Architecture Tradeoff Analysis Method (ATAM) [7]. Besides the determination of the quality objectives, the architecture analysis is the basis for the generation of refactoring alternatives to restructure the problematical parts.

The set of quality objectives needs to be structured. The intention is to resolve contradicting objectives. They are removed by refining and classifying them [5]. The objectives are classified into fundamental and means objectives. Fundamental objectives represent the preferences of the decision maker directly. Means objectives are necessary to fulfill fundamental objectives. They represent the preferences of the decision maker in an indirect way.

For refactoring activities we have to focus on fundamental objectives first. If a fundamental objective is too complex, it can be refined using a tree structure [5]. A complex quality objective at the root or at a node is refined by more detailed leaves. Understandability for example will be refined to sub-objectives loose coupling and testability. The detailed fundamental objectives (leaves) have to be supplemented with metrics (for measuring) and concrete thresholds. The procedure of building a system of objectives can be top-down or bottom-up.

RSS Reader Example

We have chosen an RSS reader [9] as practical example for illustrating the method and the decision process. This reader is part of the functionality of a cell phone. It accesses and shows news as RSS feeds from the web. For the next product version the reader is to be improved concerning its usability. Typically for a cell phone, only 11 keys are available for typing. Especially the direct input of URLs of RSS feeds lacks of usability. The product manager defined this quality requirement that does not directly affect the functional behavior of the cell phone.

To implement this quality requirement, a revision of the system and software architecture is necessary - an architectural refactoring. In the design process mostly the software architect acts as the decision maker.

As solution, the required usability of the RSS reader shall be reached by offering an additional RSS feed manager running on a personal computer. The reader on the cell phone then only imports the feed lists. This additional RSS feed manager as well as the import functions are not available. The reader has to be changed accordingly. This change demands for flexibility, similar to many functional enhancements during the evolution of a cell phone product. Therefore the architect decides for the introduction of a switching mechanism as general solution for enable a switching of behavior. In this case it will be applied for switching between the internal and the external RSS feed manager. The interface to the external RSS feed manager will use the same facilities as the synchronize functions of calendar and address book of the cell phone.

There is a contradicting objective influenced by the refactoring – safety & security. This contradiction is resolved by refining the fundamental objectives to sub-objectives and by classifying them into fundamental and means objectives (figure 1). In our case the decision for increasing the flexibility interferes with the quality objective for safety & security – especially due to the additional interface to an external platform. As resolution, the import interface (a means objective) is specified rigorously to enable the application of security restrictions.

The aimed switching to an external RSS feed manager (means objective 1) requires flexibility for switching behavior as means objective 2. To achieve this objective on an architectural level, a loose coupling of the components is necessary (means objective 3).

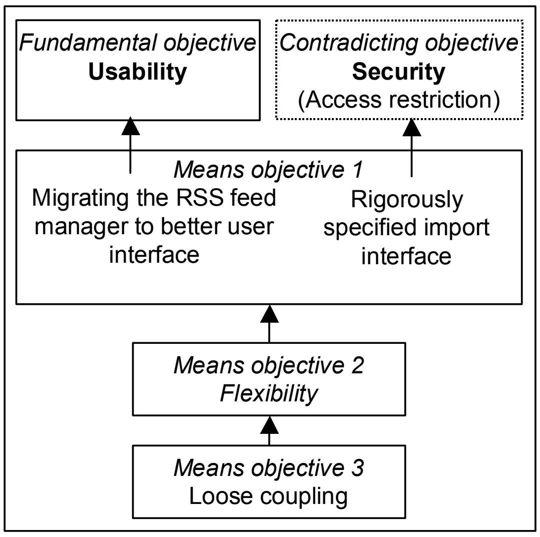

Figure 1. A sample system of objectives

Even if the changed quality requirement demands only for an architectural refactoring without changing the overall system behavior. However, components and parts within the architecture have to be functionally changed to fulfill the non-functional, quality objective usability.

3.3. Generating Refactoring Alternatives

The second step is the generation of refactoring alternatives. A refactoring alternative is an option or a strategy to restructure a critical part of the software system.

A refactoring alternative can be a single refactoring step or it can be a combination of several refactoring steps. A refactoring step is a useful combination of several refactoring activities, which belong together (see picture 2). There are many documented refactoring activities available. The refactoring catalog [10] represents an example. Refactoring activities can be adapted easily. However, not all activities are suitable for architecture-oriented refactoring. They are designed for minor changes at the level of software code.

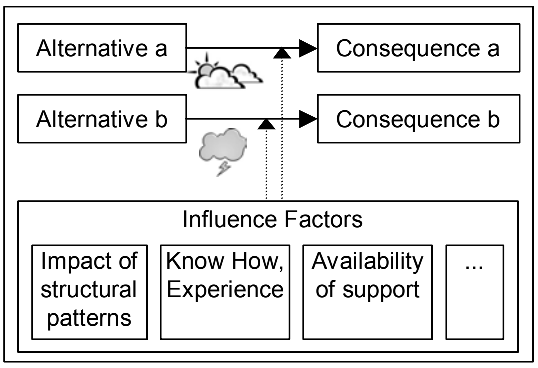

Figure 2. The components of a refactoring step

Sources for the generation of the refactoring alternatives are patterns and styles, personal experiences or former solutions. The patterns include design and architectural patterns [11, 12]. The patterns are common used solutions for design or architectural problems [13]. They describe a solution structure for the problematical parts of the software architecture. For several design patterns, sequences of refactoring activities that help to transform an existing system into one implementing the pattern have been described in [14]. Architectural styles are more abstract. They describe fundamental responsibilities and relations of the components of a software system.

Consequences of refactoring alternatives are determined by influence factors, e.g. a positive or negative impact from design patterns on other parts of the solution, effort or risks. If impacts are uncertain, probabilities will be used instead. The consequences of the refactoring alternatives depend on the influence factors and the attributes of the refactoring steps, like the effort or the estimated risk.

If the decision maker has to deal with uncertainty, the impact should be modeled to ease a later evaluation. The following types of impact models are useful [4, 5]:

- *Decision matrix*: The refactoring alternatives will be combined with the influence factors (e.g. states, events) and their occurrence probabilities in a matrix. The points of intersection are the consequences. A decision matrix is useful, if the consequences are a functional relation between the influence factors and the attributes of the refactoring. An example is the functional relation between refactoring efforts and costs.
- *Decision tree*: Such a tree combines the alternatives (modeled as squares), the corresponding events (circles), the occurrence probabilities and other factors (like costs, effort). The different refactoring alternatives will be modeled as branches. There is a root as starting point. The consequences are the result of each branch. A decision tree is useful to describe refactoring strategies.
- *Decision diagram*: A decision diagram is less detailed than a decision tree. Instead of modeling each refactoring step, a decision diagram contains sets of refactoring steps (squares), which belong together. The influence factors will be aggregated and modeled as circles. The several consequences are not modeled in detail. A decision diagram contains the impact on the objectives directly (modeled as a hexagon). The refactoring alternatives and the states or events can be combined flexibly via borders. Cycles are not allowed. A decision diagram is comparable to network plans, like PERT or Event-Driven Process Chains [15]. A decision diagram is useful, if the decision problem is too complex due to many refactoring steps and influence factors.

Alternatives for the RSS Reader

In the example, alternative *a* consists in the application of the Strategy design pattern. It enables a interchange between two algorithms by encapsulating them into classes with a common abstract superclass [13]. It can here be applied to enable a switching between internal and external RSS feed handler. One of the required refactoring steps is the encapsulation of the corresponding classes with an identical interface, here the method *subscribeToFeed()*. This refactoring step has specific attributes and consequences. They are determined by influence factors, like the positive or negative effects of already implemented design patterns.

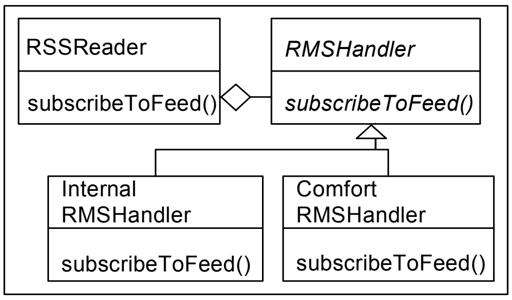

Figure 3. RSS reader with Strategy pattern

Alternative *b* consists in an inline extension of some classes by an import interface. This refactoring alternative was developed from the experience of the architect.

The architect has modeled the refactoring steps with the required effort (hours) in form of a decision tree (see picture 3). The result of each step is a state with either a better loose or a continued tight coupling of the *FeedList*. The corresponding occurrence probabilities are given in brackets. The consequences for the refactoring effort and the occurrence probabilities are aggregated at the end of each branch. The impact for the flexibility of *RMSHandler* is estimated with grades between 1 and 5 (1 high – 5 low).

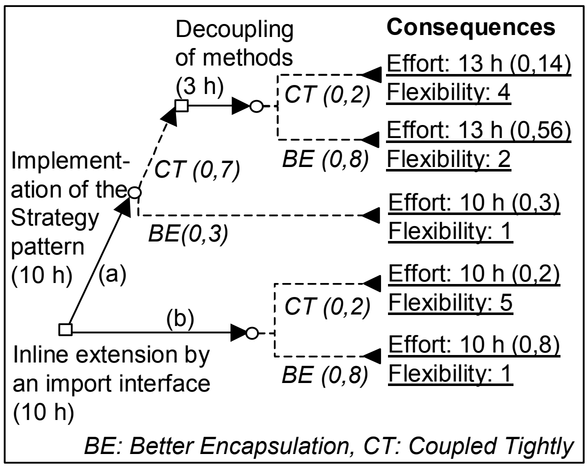

Figure 4. A sample decision tree

The decision tree starts with the square on the left side of the figure 4. The first step of alternative *a* is the implementation of the Strategy pattern with a development time of 10 h. The result of this step could be a better encapsulation of the *RMSHandler* with a probability of 30 %. The flexibility would be of grade 1. With a probability of 70 % a loose coupling of the *RMSHandler* could not be reached. An additional decoupling of methods is necessary, because the source code of the *RMSHandler* is tangled with the source code of other features. After the decoupling activity, which has taken 3 h of development time, the encapsulation of the *RMSHandler* could be improved (probability 80 %) or a loose coupling could not be reached (probability 20 %). In the better case, the flexibility will be of grade 2 with an total effort of 13 h. The total probability of this branch is 56 %. In the other case the flexibility will be of grade 4 with a total effort of 13 h. This case can occur with a total probability of 14 %.

The first step of alternative *b* is an inline extension by an import interface. This step takes 10 h of development time. This step could enhance the encapsulation of the *RMSHandler* (probability 80 %) or a loose coupling of the *RMSHandler* could not be reached (probability 20 %). In the better case, the flexibility will be of grade 1. In the other case it will be of grade 5.

3.4. Evaluating the Refactoring Alternatives

In the evaluation, the consequences of the refactoring alternatives are compared to the preferences of the decision maker. It has to be analyzed which of the alternatives matches the preferences in the best way. There are different ways of evaluation if there are more than one consequences, or if consequences cannot be determined certainly [5, 16].

If the decision maker has to consider only one consequence of the alternatives, and this consequence can be determined certainly, a value function *v* is used for evaluating. The value function *v* assigns numbers between 0 and 1 to each alternative. If the decision maker prefers *a* before *b*, then a higher number is assigned to a (formula 1).

$$v(a) \geq v(b) \Leftrightarrow a \succ b, \qquad a,b \in A \qquad \textbf{(1)}$$

The value function *v* for alternatives with more than one consequence is based on an additive model. An alternative *a* is characterized by a vector of consequences $a = (a_1,\dots, a_m)$. The value of the consequences is normalized in the interval [0,1]: The highest value is normalized with 1, the lowest with 0. The value of an alternative is calculated by the value function *v* (formula 2). Each of the consequences is weighted by a factor *w* according to its relevance.

$$v(a) = \sum_{r=1}^{m} w_r v_r(a_r) \qquad \textbf{(2)}$$

If there is exactly one consequence with an uncertain value, a utility function is used to determine the expected value (*EV*). Such a utility function applies a probability *p* of a situation $s = (s_1,\dots, s_n)$ as the weight of the corresponding consequence *a*.

$$EV(a) = \sum_{i=1}^{n} p_i u(a_i) \qquad \textbf{(3)}$$

A utility function with more than one uncertain consequences is similar to the value function *v* of formula 2. Each consequence is weighted by a factor *k* according to its relevance and by the probability *p* of a situation s = (s_1,…, s_n).

$$EV(a) = \sum_{i=1}^{n} p_i \cdot \left[\sum_{r=1}^{m} k_r u_r(a_{ir}) \right] \qquad \textbf{(4)}$$

In general, the determination of the probabilities is difficult. Another way to evaluate the alternatives is based on fuzzy logic [17]. Probabilities are modeled as linguistic variables, with positive, neutral or negative values. The consequence of an alternative is evaluated by fuzzy sets. However, fuzzy logic is imprecise and has only a low relevance for the decision theory. Therefore, it is not discussed here in detail.

The preferences of the decision maker have to be expressed quantitatively to enable an evaluation. For their determination there exist different methods, e.g. the Direct Rating Method [16]. By this method, the highest and the lowest values of the consequences are normalized to 1 and 0. The values in-between are inserted to the interval and an interpolation is performed.

Evaluation of the RSS Reader Alternatives

Two sample utility functions for the consequences of the alternatives *a* and *b* (see figure 4) are shown in figure 5. The minimum flexibility grade (5) and the maximum effort (13 h) is of the lowest preference of the decision maker, thus assigned to an utility of 0. The utility of 1 is assigned to the highest flexibility grade and the lowest effort.

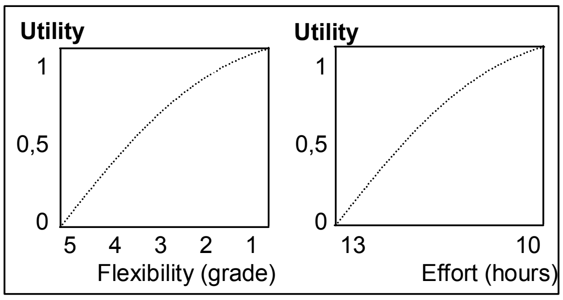

Figure 5. Utility functions for two consequences

The alternatives *a* and *b* can now be evaluated using the utility functions shown in figure 5. In our case, the decision maker has weighted the impact on the flexibility (a_1 and b_1) with 0.6 and the effort (a_2 and b_2) with 0.4. The expected utility derived using formula 4:

$$\begin{aligned} EV(a) &= EV(a_1) + EV(a_2) = 0.48 + 0.12 = 0.60 \\ EV(b) &= EV(b_1) + EV(b_2) = 0.48 + 0.40 = 0.88 \end{aligned} \qquad \textbf{(5)}$$

As a result of the evaluation, the expected utility of alternative *b* is higher than the one of alternative *a*. However, this result is based on estimations and is therefore a suggestion for the decision maker.

3.5. Implementation and Final Analysis

The selected refactoring alternative has now to be implemented. In a planning step, critical or especially complex refactoring activities are analyzed in detail. Unit tests and reviews have to prove, that the external, viewable behavior remains unchanged. For the RSS reader example, only the system behavior for the RSS feed subscription remains unchanged, described by some use cases.

A final analysis has to check if the changes to the software architecture have lead to an improved software quality. In the final analysis the quality objectives are compared to the actual quality state.

The implementation phase of the architecture-oriented refactoring process is well supported by several refactoring tools, like the refactoring functions of the open source IDE Eclipse [18].

4. Conclusion and Summary

This paper wants to facilitate refactoring, an activity to improve the software quality without changing the functional behavior of a software system [1], especially on the level of the software systems architecture. During refactoring activities decisions between refactoring alternatives have to be made, which are driven by several, partly contradicting objectives, and which consequences are partly uncertain. The characteristics of these decisions encourages the application of decision theory [5].

In this paper, the methods of decision theory are adapted to architectural refactoring. By this adaptation the structure of the architecture-oriented refactoring process could be improved. This provides the basis for a rational procedure for architecture-oriented refactoring decisions. It helps to structure the objectives and to resolve conflicting objectives as well as risks. Refactoring alternatives are developed and evaluated according to their consequences. The paper presents methods for evaluating refactoring alternatives by decision theory methods, even if the set of consequences is difficult to interpret and the values are uncertain. The application of the method is demonstrated using examples from a case study.

As an experience we can conclude, that architecture-oriented refactoring represents a very helpful technology for improving software quality. The presented method helps to reduce the large effort for evaluating alternatives, caused by a large amount of influence factors and by incomplete information about interdependencies. Furthermore it helps to reduce uncertainty, that would otherwise lead to a reduced value of the results.

However, the suitability of the support of the architecture-oriented refactoring decisions is limited by the correctness of the modeled refactoring scenario. Such a scenario has to include all relevant events and environmental influences with realistic probabilities

From an economic point of view the method helps to reduce risks, to simplify the development process and to increase its efficiency. Furthermore, the method supports planning and risk management by providing decision support and by optimization.

Finally it can be stated, that the support for architecture-oriented refactoring decisions can help to avoid additional expenses and unwanted side effects. The efficiency of the refactoring process and the communication with the stakeholders can be improved.

5. References

[1] M. Fowler, K. Beck and E. Gamma, *Refactoring: Improving the design of existing code*, Addison-Wesley, Boston, 2005.

[2] P.R. Kleindorfer, H.C. Kunreuther and P.J.H. Schoemaker, *Decision Science: An integrative perspective*, Cambridge University Press, 2003.

[3] T.L. Saaty, *Decision Making for Leaders*: *the Analytic Hierarchy Process for Decisions in a Complex World,* RWS Publications, Pittsburgh, 2001.

[4] M. Florenzano, P. Gourdel and V. Marakulin, “Implementing Financial Equilibrium of Incomplete Markets – Bounded Portfolios and the Limiting Case”, In *Applied Decision Analysis*, edited by F.J. Girón and M.L. Martínez, 181-191, Kluwer, Boston, 1998.

[5] R. Clemen and T. Reilly, *Making Hard Decisions with Decision Tools*, Pacific Grove, Brooks Cole, 2004.

[6] E. Forman and M.A. Selly, *Decision By Objectives*, World Scientific Publishing Company, 2002.

[6] ISO Standards 8402, “Quality management“, 1995.

[7] R. Kazman, M. Klein and P. Clements, “ATAM: Method for Architecture Evolution” *Technical Report CMU/SEI-2000-TR-004* (2000), http://www.sei.cmu.edu/pub/documents/00.reports/pdf/00tr004.pdf.

[9] J2ME RSSReader: http://sourceforge.net/projects/j2merssraeder, (accessed Jan. 10, 2006).

[10] M. Fowler, “Alpha list of refactorings”, http://www.refactoring.com/catalog/index.html (accessed Nov. 01, 2005).

[11] J. Bosch, *Design & Use of Software Architectures: Adopting and evolving a product line approach*, Addison-Wesley, Harlow, 2000.

[12] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *Pattern-oriented software architecture - A system of patterns*, Wiley, Chichester, 2001.

[13] E. Gamma, R. Helm, R. Johnson and J. Vlissides: *Design Patterns – Elements of reusable object oriented Software*, Addison-Wesley, Munich, 2004.

[14] J. Kerievsky: *Refactoring to Patterns*, Addison-Wesley, Boston, 2005.

[15] G. Hoblik, *Dynamic organization methods for networked process- and infrastructure planning*, Austrian Art & Culture, Vienna, 2000.

[16] D. Bouyssou, T. Marchant, M. Pirlot, P. Perny, A. Tsoukias and P. Vincke, *Evaluation and Decision Models. A Critical Perspective*, Kluwer, Boston, 2000.

[17] J.J. Buckley, Fuzzy probabilities and fuzzy sets for web planning, Springer, Berlin, 2004.

[18] D. Gallardoa, “Refactoring for everyone”, ftp://www6.software.ibm.com/software/developer/library/os-ecref.pdf (accessed Nov. 01, 2005).