A Question-based Method for Deriving Software Architectures[⋆]

Marco Müller, Benjamin Kersten, and Michael Goedicke

paluno - The Ruhr Institute for Software Technology, University of Duisburg–Essen
Gerlingstraße 16, 45127 Essen, Germany
{marco.mueller,benjamin.kersten,michael.goedicke}@paluno.uni-due.de

Abstract. Although several approaches exist for deriving architectures from requirements and environmental constraints, most solutions rely on experienced architects for proposing and choosing feasible architectural solutions. It is critical to develop architecture systematically and without strong dependencies on experienced architects, because the architecture has a deep impact on the quality of a system. This paper presents a question-based approach for efficiently finding architecture candidates using annotated pattern and style catalogues. Following this approach allows for a systematic development of architecture, that provides documented common experience.

1 Motivation

The development of software architecture is a challenge, even when requirements, environmental constraints, and the domain of a software system are clear. This is especially true, when a development team includes no experienced architect. For building upon common knowledge and best practices, the usage of catalogues containing architectural patterns and styles (e.g. [1, 2]) has shown to be valuable. These catalogues are subject to architectural design methods, that aim to be a guide for deriving architectures from requirements (e.g. [3–6]). In these methods, the catalogues are used as a reference to find solutions for an architectural problem by choosing applicable patterns and styles (called *solution candidates* in this document) from the catalogue.

Trying to find an applicable solution candidate is, however, not enough. Several questions should be answered during that selection: E.g. What are the criteria for a candidate to be applicable? Are there constraints (e.g. business or technical)? Is the candidate a good choice to meet the quality requirements? When quality requirements are contradictory, is the candidate a good trade-off? These questions target the selection of candidates from a catalogue. Most existing approaches are imprecise or don't provide any aid for choosing good candidates from catalogues [7].

⋆ The work reported herein was funded by the German Research Foundation under the grant no. GO 774/5-1.

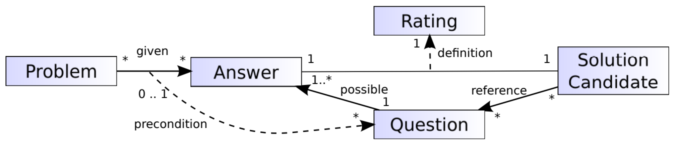

Fig. 1. Elements of the question-based method and their references

Sequentially evaluating the candidates of a catalogue for given requirements and constraints in a given domain (called *problem* in the remainder of this document) is not efficient. In this paper, we propose a process providing aid in this selection, by relating a specific problem to common experience. We extend the solution candidates in catalogues with rated questions. By answering these questions regarding a specific problem, a small set of candidates can quickly be identified as promising for that problem. The presented approach aims at helping to derive architectures that also meet quality requirements. To achieve this goal, the presented approach does not aim to elicit a single solution. Instead, it suggests solution candidates that are promising to meet the quality requirements. Before actually choosing a candidate, the proposed process provides for iteratively evaluating the most-promising candidates.

The remainder of this paper is structured as follows: Section 2 explains the extension of meta data for solution candidate catalogues. Section 3 describes the process using this meta data for efficiently selecting solutions. The provided tool support is introduced in section 4. Section 5 describes the current state of evaluation of the approach. Section 6 presents related work, before we conclude and present future work in section 7.

2 Annotating Architecture Patterns with Questions

In order to efficiently find solution candidates for problems, we annotate each candidate in a catalogue with questions and rated answers. The questions' goal is to find out how the solution candidates relate to a specific problem. Thus the question may target functional requirements, quality requirements, environmental constraints, and domain constraints.

The meta data for solution candidates is structured as shown in figure 1. Questions define a set of possible answers. These questions are referenced by the candidates. Thus different candidates can share the same questions. For each solution, a different rating may be defined for an answer. The rating describes whether a solution contributes positively $(0, 1.0]$ or negatively $[-1.0, 0]$ to a problem. The rating may also be *excludes*, which means that the solution candidate contradicts the problem, and is thus excluded. If a question is not answered (yet), its rating is 0 by default. In addition, specific answers can be preconditions for other questions to be asked. The elements are formally defined as follows:

– p is a problem definition,
– Q is a set of questions,

- a_q^p is an answer given to question q regarding problem p,
- $A_q, q \in Q$ is a set of possible answers for a question,
- A^p the set of given answers a_q^p for all questions $q \in Q$,
- S is a set of solution candidates,
- $r_s(a) \in [-1.0, +1.0] \cup \{\text{excludes}\}$ is the rating for the answer $a \in A_q$ of question $q \in Q$ for the solution candidate $s \in S$.

A solution candidate is a pattern description as found in pattern catalogues. For our purpose we define a solution candidate to be $s = \langle Q_s, \text{pre}_s \rangle$, with $Q_s \subseteq Q$ questions referenced by the solution candidate and $\text{pre}_s : Q_s \to 2^{A_{q_s}}, q_s \in Q_s$ being a function stating the answers that are preconditions for a question reference. The set of answers given for the questions referenced by s is A_s. A question reference q_s is called *enabled* for the problem p if: (1) it has not been answered for the problem in focus and, (2) it has either no precondition or (3) any precondition is a given answer:

$$\text{enabled}_{p,s}(q_s) := \nexists a_q^p \wedge (\text{pre}_s(q_s) = \emptyset \vee \exists a(a \in \text{pre}_s(q_s) \wedge a \in A^p))$$

Otherwise the reference is called disabled. A question is called enabled if it has any enabled question reference: $\text{enabled}_p(q) = \exists s(s \in S \wedge \text{enabled}_{p,s}(q))$. Analogously, a question is called disabled otherwise. At last, the rating r_s for a solution candidate s regarding all given answers A^p is the average of the single ratings or *excludes* if any rating is *excludes*:

$$r_s(A^p) := \begin{cases} \text{excludes} & \text{, if } \exists a(a \in A^p \wedge r_s(a) = \text{excludes}) \\ \frac{\sum\limits_{a \in A^p} r_s(a)}{|A_s|} & \text{, else} \end{cases}$$

3 A Process for Identifying Architectural Candidates from Requirements and Context

The elements described in section 2 can be used for efficiently identifying promising solution candidates for problems. In this section, we introduce the process for systematically deriving architecture candidates in an iterative top-down approach. This process is schematically depicted in figure 2. In each iteration promising candidates are found for the architecture. During the first iteration, the considered problem is the overall system, while in later iterations, the candidates are refined. Each iteration builds upon the results of the preceding iterations. Thus a tree of architecture candidates is spanned. The candidates are evaluated after each iteration. Due to this evaluation, branches of candidates that cannot fulfill the quality requirements are excluded early. The result of the process is a set of promising architecture candidates. The process is supposed to be executed by a person knowing the requirements, the context, and the domain of the system. This person is the reference for the process to the concrete problem.

In the following, the process is described in detail. Step (1) is to *get promising candidates* for the problem. The promising candidates is an ordered set $(S_{A^p}, \geq)$.

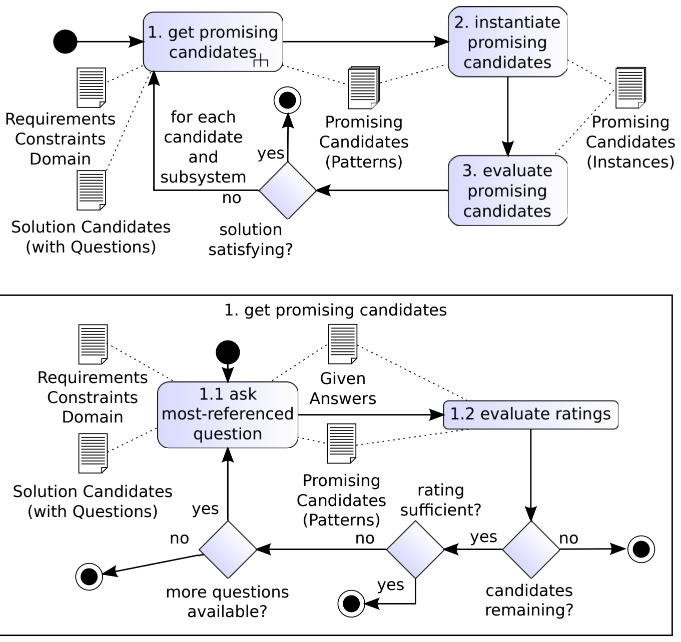
1. get promising candidates
2. instantiate promising candidates
3. evaluate promising candidates
Requirements Constraints Domain
Solution Candidates (with Questions)
for each candidate and subsystem
yes
no
solution satisfying?
Promising Candidates (Patterns)
Promising Candidates (Instances)
1. get promising candidates
Requirements Constraints Domain
1.1 ask most-referenced question
Given Answers
1.2 evaluate ratings
Solution Candidates (with Questions)
Promising Candidates (Patterns)
rating sufficient?
yes
no
yes
no
yes
no
more questions available?
yes
candidates remaining?

Fig. 2. An overview of the process

S_{A^p} is the set of candidates which have not been excluded by one of the given answers A^p:

$$S_{A^p} = \{s | s \in S \wedge \forall a (a \in A^p \wedge r_s(a) \neq \text{excludes})\}.$$

The candidates are ordered by the ratings of the given answers:

$$s \geq s' \Leftrightarrow r_s(A^p) \geq r_{s'}(A^p).$$

To find these candidates, the first substep *ask most-referenced question* (1.1) is executed. In this substep, one of the questions that are referenced by the most remaining candidates are asked to the person knowing the problem p. The number of enabled references to a question is identified by

$$ref(q) = \sum_{s \in S} |\{q | q \in Q \wedge \text{enabled}_{p,s}(q)\}|.$$

Using this, the ordered set of enabled questions can be defined as $(\text{enabled}_p(q), \geq)$, with $q \geq q' \Leftrightarrow |ref(q)| \geq |ref(q')|$. In the next substep (1.2), the ratings are evaluated. Due to the definition of $\text{enabled}_{p,s}(q)$, the recently answered question is removed from the set of enabled questions. Analogously, solution candidates with $r_s(a^p_q) = \text{excludes}$ are removed from the set of promising candidates. Their question references are thus disabled.

At this point, no more candidates might be available. In this case, the catalogue of solution candidates does not provide an applicable solution candidate. If more questions are enabled, substep (1.1) is repeated with these questions. Otherwise step (1) is finished. The result is a set of patterns that represent promising candidates. The candidates with the highest rating are the most promising. Step (1) can be stopped after each evaluation, because the process does not require all questions to be answered. The rating is more specific with more answered questions.

The process aims at proposing candidates based on common knowledge. Thus the resulting candidates still need to be evaluated to confirm their feasibility and applicability. In the next step *instantiate & evaluate promising candidates* (2), the patterns resulting from step (1) are instantiated to model an architecture that fulfills the functional requirements and the constraints. The instantiated architectures are then evaluated e.g. using tradeoff analysis techniques. The process does not constrain the choice of methods or tools for evaluating architectures. When the evaluation shows that a candidate will most likely not meet the quality requirements, or is excluded due to a tradeoff analysis, it is removed from the list of candidates for the given process interation. If the instantiated architecture is detailed enough, the process can be stopped. Otherwise, for each candidate, each subsystem is taken as problem p for a next iteration. The set of questions and candidates are reset and the process starts a next iteration. Eventually, the iterations result in an architecture on the desired level of detail, or the process shows that no solution candidates are applicable. In the latter case, more patterns are necessary, or the requirements have to be adapted.

4 Tool Support

To support the approach, a tool was developed to store the description of solution candidates, including questions and ratings. The tool allows for easily modifying and querying the catalogue of solution candidates. We implemented this catalogue using a Semantic Media Wiki (SMW), a semantic extension for Media Wiki. The wiki allows for easily creating and modifying informal descriptions of candidates, that define relationships to each other for an easy understanding of the candidates and their environment (e.g. related patterns). The semantic extensions allow for defining typed relationships between pages, as well as properties for pages. Using these properties, relationships and pages are enriched with semantic information. The structure of the semantic properties is used as a meta model. Semantic wikis allow for defining complex queries over the modeled data.

In the semantic wiki used as solution candidate catalogue, the meta model was designed to define the structure necessary for the presented approach, as shown in figure 1. The elements in this figure represent data types (boxes) for wiki pages and their relationship types (arrows). The rating for an answer regarding a specific solution was realized using complex property types (the record type in SMW).

Furthermore, to support the structured description of solution candidates, we used semantic forms. They provide forms to guide editors of the wiki to reuse questions that are already defined, and to provide further semantic information, e.g. related patterns.

5 Example

The process was evaluated in internal software development projects. In this section, we show an excerpt of the process execution for designing a chat application, as it was used by students in a seminar. In this evaluation, a pattern catalogue with 10 patterns referencing 25 questions was used. The ratings were defined by experience. The functional and quality requirements for the application were given, as well as technical constraints: the framework to use was predefined due to the environment the software should be run.

Candidate \ Question	(1)	(2)	(3)	...	Average
Client/Server	1	0.5	0.9		0.19
Simple Peer-to-Peer	1	0	-0.1		0.22
Standalone	excludes	x	x		0.03
Shared State	-0.5	x	-0.8		0.07
Batch-Sequential	0.2	0	x		0.02
Pipes & Filters	0.2	0	x	...	0.0
Publish-Subscribe	0.2	0.4	-0.1		0.14
Event-Based	0.1	0.2	-0.3		0.13
Blackboard	x	0.6	-0.8		0.04
Layered	x	x	0.3		0.11

(1) Is the system necessarily distributed? → Yes
(2) Are there more clients expected than can be handled by a single node? → No
(3) Does the system conduct sensible / confidential data? → Yes

Table 1. Rated answers for the first iteration of the chat application

In the first iteration, in step (1) of the process, the most-referenced questions were asked. An excerpt of the questions and given answers in this iteration is shown in table 1. The tables shows only the ratings for the given answers. As a result of step (1), the candidates *Client/Server* and *Simple Peer-to-Peer* were identified to be the most-promising candidates. The other candidates were excluded or had a significantly lower rating. In step (2), both candidates were instantiated. I.e. the styles were used as alternative designs for the application on the top-most abstraction level. In step (3), the instantiated architectures were evaluated by a performance-test, as the performance was one of the most important requirements. Both candidates passed the evaluation. Thus in the next iterations, each candidate was considered further when the architecture was refined. After five iterations, one solution (client/server-based) was considered detailed enough and satisfying.

6 Related Work

Hofmeister et al. [8] compared five industrial software architecture design methods to extract a general design model from them. In their model, our approach can be used as the the Architectural Synthesis activity.

Related work is first of all found in different architecture derivation methods. These methods vary in their abstraction level and the development phases that are considered. For instance, Rational Unified Process (RUP)[9] is dealing with any software development process phase from early requirements to production and evolution. When it comes to the selection of patterns as solutions for a problem, RUP allows to integrate different architecture methods. It thus does not provide any guidance for this task.

Methods that are more specific in this point emphasize the design phase and describe how to select certain architecture styles and patterns for a given problem. For example, Attribute Driven Design (ADD) [3] is a method approximating this task. It defines a process to derive an architecture from requirements using patterns. However, ADD does not describe how to elicit a matching pattern from a large pattern catalogue, except for sequentially evaluating each element in the catalogue (cf. [10]). There are more architecture methods that are imprecise at this point such as Object Oriented Modeling and Design [4], Siemens 4 Views [5] or Architectural Separation of Concerns [6]. Therefore, our approach bridges the gap of pattern elicitation found in related work. Our approach is not designed to complement these methods, but it can be integrated to enrich them.

Zdun uses questions to select architecture patterns in [11]. In this approach, questions are directed to key characteristics of a group of patterns (e.g. "*How to realize asynchronous result handling?*"). The answers are patterns, which are related to criteria supporting the decision process. We believe that our approach is more suitable for less experienced teams, because of the are related to the system's requirements and context. This is, however, subject to validation.

Bode and Riebisch [7] relate solutions to quality requirements. Their work focuses on rating the impact of patterns (called solution instruments in this context) on quality goals. They first refine quality goals with subgoals. Then they group patterns to solution principles, e.g. modularization. A experience-based rating is then defined between solution principles and subgoals. In contrast to our work, Bode and Riebisch develop context-independent ratings. We are confident, that the specific requirements, the environmental constraints, and the domain influence these ratings.

7 Conclusion and Future Work

In this paper we presented a question-based approach for systematically proposing architecture candidates for software systems and subsystems. To select appropriate architecture patterns and styles we annotated each of these candidates within a catalogue with semantic meta data: questions, answers, and ratings. A developer who is familiar with the requiremens answers questions within a defined process to exclude irrelevant candidates and to guide the selection by rating the candidate regarding the given answers. To evaluate our approach, we evaluated the process in several internal software development processes.

As future work, we plan to further evaluate the process. For a deeper reflection, a larger base of solution candidates is necessary, as well as refined ratings for answers. We thus plan to extend the evaluation to larger projects with students and industrial partners. Furthermore, we are developing an interactive application on top of the wiki as already provided tool, to guide users through the process in a user-friendly way. We also plan to publish the semantic wiki as catalogue for general usage and for using the common experience of practitioners and researchers for extending the catalogue with more patterns and styles, and for refining the ratings. We also plan to find methods for refining the ratings by evaluating data collected from projects that use the presented process.

References

1. Taylor, R.N., Medvidovic, N., Dashofy, E.: Software Architecture: Foundations, Theory, and Practice. 1. auflage edn. John Wiley & Sons (2 2009)
2. Gamma, E., Helm, R., Johnson, R.E.: Design Patterns. Elements of Reusable Object-Oriented Software. 1st ed., reprint. edn. Addison-Wesley Longman, Amsterdam (10 1994)
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice (SEI Series in Software Engineering). 2. a. 2003. edn. Addison-Wesley Longman, Amsterdam (4 2003)
4. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorenson, W.: Object-Oriented Modeling and Design. United states ed edn. Prentice-Hall (12 1991)
5. Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture: A Practical Guide for Software Designers (Addison-Wesley Object Technology). Addison-Wesley Longman, Amsterdam (11 1999)
6. Jazayeri, M., Ran, A., van der Linden, F.: Software Architecture for Product Families. Addison-Wesley Longman, Amsterdam (5 2000)
7. Bode, S., Riebisch, M.: Impact Evaluation for Quality-Oriented Architectural Decisions regarding Evolvability. In Babar, M., Gorton, I., eds.: Software Architecture. Volume 6285 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2010) 182–197
8. Hofmeister, C., Kruchten, P., Nord, R.L., Obbink, J.H., Ran, A., America, P.: Generalizing a model of software architecture design from five industrial approaches. In: WICSA. (2005) 77–88
9. Kruchten, P.: The Rational Unified Process: An Introduction (Addison-Wesley Object Technology). 2 sub edn. Addison-Wesley Longman, Amsterdam (4 2000)
10. Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R., Wood, B.: Attribute-driven design (add), version 2.0. Technical report, Software Engineering Institute (2007)
11. Zdun, U.: Systematic Pattern Selection using Pattern Language Grammars and Design Space Analysis. Software: Practice and Experience **37**(9) (2007) 983–1016