QuOnt: an ontology for the reuse of quality criteria

Remco C. de Boer and Hans van Vliet
Department of Computer Science
VU University Amsterdam
the Netherlands
{remco,hans}@cs.vu.nl

Abstract

Software product audits are knowledge-intensive tasks in which architectural knowledge plays a pivotal role. In the input stage of a software product audit, quality criteria are selected to which the software product should conform. These quality criteria resemble architectural tactics and can be viewed as a definition of the Soll-architecture of the product. Like tactics, the same quality criteria can be applied to different software products. However, there are currently no models that support the codification of quality criteria as reusable assets. In this work, we present an ontology that supports the reuse of quality criteria in the input stage of software product audits.

1. Introduction

On occasion, organizations may experience the need to verify the quality of a software product. Such a need may arise, for example, prior to acquisition or in the case of contracted-out development. A way to assess the quality of a software product is to let an independent party perform an audit.

Software product audits are knowledge-intensive tasks in which architectural knowledge plays a pivotal role. For example, architectural design decisions and their rationale provide insight into the trade-offs that were considered, the forces that influenced the decisions, and the constraints that were in place.

Knowledge work, according to Mackenzie Owen, incorporates "the gathering, processing, creating, sharing and disseminating of knowledge" and consists of three distinct stages: input, throughput, and output [13]. In each of these stages, knowledge is employed: in the input stage, relevant existing knowledge and data are gathered; in the throughput stage, knowledge and data are analyzed and processed; and in the output stage, the results of the previous stage are recorded and disseminated.

A typical software product audit consists of the following activities:

- Input stage: gather quality attributes, quality criteria, and product artifacts.
- Throughput stage: compare the product artifacts with the desired level of quality, expressed in terms of quality attributes and quality criteria.
- Output stage: lay down any findings regarding deviations from the desired level of quality in a report.

We can distinguish three stakeholders in a software product audit: the customer (i.e., the party who requested the audit), the supplier (i.e., the party who developed the software product), and the auditor (i.e., the party who independently assesses whether the supplier's software product conforms to the customer's needs).

In the input stage, the product artifacts are obtained directly from the supplier. The important quality attributes are usually determined in a workshop with the customer. The result from such a workshop could for example be that security, maintainability, and usability are the three most important quality attributes to a customer, and that of those three security has the highest and usability the lowest priority. From the quality attributes and their priorities, quality criteria are derived that the product should satisfy. Unlike quality attributes, which are still a fairly abstract representation of 'quality', quality criteria represent concrete measures that may or may not be found in the software product. For example, in a system where security is important proper user authentication would probably be a quality criterion; without such authentication, the necessary level of security is unlikely to be reached.

Although the derivation of quality criteria from quality attributes again involves deliberation with the customer, it requires a fair amount of technical background knowledge. Hence, auditors generally have the lead in deciding upon which quality criteria to use.

SHARK'09, May 16, 2009, Vancouver, Canada
978-1-4244-3726-9/09/$25.00 © 2009 IEEE

Software product audits should not be regarded as isolated projects. Rather, individual audits affect each other even if they target unrelated software products. For instance, lessons learned in one project might be applicable to another. Moreover, the applicability of certain quality criteria is not limited to a single project alone. Similar projects might use similar quality criteria, and some general quality criteria might even be applicable to virtually all software products. For example, in high-security systems some form of user authentication will always be needed.

In short, many quality criteria are reusable assets. However, there are currently no models known from literature that support the structured codification of quality criteria for reuse. Consequently, such reuse tends to occur on an ad-hoc basis, and may involve rereading past audit reports to identify previously used criteria applicable to the situation at hand. Some audit organizations may take this one step further and collect reusable criteria in a central location. As part of our research into reuse of quality criteria, we collaborated with DNV-CIBIT. This organization experienced difficulties in reusing applicable quality criteria from past projects, despite attempts to create a central repository. Since this repository essentially consisted of a list of quality criteria linked to quality attributes, important additional knowledge including dependencies between criteria could not be captured.

In this paper, we draw on architectural knowledge management theory to propose QuOnt, an ontology that supports codification of quality criteria with reuse as primary goal. To support reuse, we must pay particular attention to the relationships between quality criteria and quality attributes, as well as the interrelationships between quality criteria themselves. To this end, in Section 2 we first briefly describe how quality criteria can be viewed as decisions that determine the 'soll' architecture of a software product. In Section 3 we briefly discuss related work. In Section 4 we then draw analogies between Kruchten's ontology of architectural design decisions and the repository structure that originated in DNV-CIBIT from their experience with quality criteria. Based on overlap and distinctions between the two structures, we derive our proposed ontology for quality criteria. We evaluate this ontology in Section 5 by working out a scenario that shows how QuOnt supports reuse and therewith solves the shortcomings of ad hoc reuse of quality criteria.

2. Quality Criteria: Deciding the SOLL-Architecture

Architectural knowledge plays a role in software product audits on at least two levels. On the one hand, there is the architectural knowledge that relates to the current state of the software product. This knowledge originates from the product's supplier, and in its codified form can be found in product artifacts such as code and documentation. On the other hand, there is the architectural knowledge that relates to the desired state of the software product. This knowledge originates from the customer, is subsequently enhanced by the auditor, and takes the form of what we call 'quality criteria'.

Quality criteria are in a way similar to what Bass et al. call 'tactics' [2]. Both relate architectural choices to their effect on quality attributes. A major distinction, however, is that tactics are usually employed from an optimistic perspective: as potential improvements in a forward engineering sense. Therefore, the focus is on tactics as *contributors* to the system's quality (e.g., improve or attain security through user authentication). Quality criteria, on the other hand, have an inherently more pessimistic nature: their focus is not only on how the system's desired quality level could be reached, but also on design choices that are *inhibitors* to the desired level of quality.

For example, 'authenticate users' could indeed be a quality criterion for a system in which security is an important quality attribute. In that case, the quality criterion resembles (or maybe even equals) the idea of a tactic, in that it represents a contribution to attainment of the desired level of security. On the other hand, when security is not important but, let's say, usability is the prime target, the applicability of 'authenticate users' as a tactic disappears. As a quality criterion, however, it is still an important consideration, since a system in which users are required to authenticate is less user-friendly than a system without authentication mechanisms. Hence, an auditor might still want to look for authentication mechanisms in the system, motivated by the fact that authentication actually lowers the system's quality (i.e., usability) and therefore should not occur. Obviously, in a realistic situation the desired level of quality will never be expressed in terms of *either* security *or* usability, but one quality attribute may be preferred over the other. In any case, in a software product audit it is important to know of any positive *and* negative effects of a design choice in order to determine whether that choice is allowed or even required to attain the desired level of quality.

The resemblance between quality criteria and tactics is indicative of the dual nature of quality criteria: they are a reflection of quality requirements pertaining to the software product, but at the same time show characteristics of design decisions. This directly relates the concept of quality criteria to other studies that discuss the characteristics of architectural knowledge, many of which employ a decision-oriented view on architecture and design [3]. Indeed, 'decisions' can be seen as an umbrella concept that unifies different views on architectural knowledge [9].

In earlier work, we discussed how architectural design

decisions are related through what essentially is a decision loop [4]. This loop reflects how architectural design decisions introduce new design issues for which new decisions need to be taken. One of the implications of this loop is that there is no clear-cut distinction between architecturally significant requirements and architectural design decisions. In fact, we have argued that requirements and decisions are similar statements, only viewed from different angles [5]. This similarity shows again in the role of quality criteria in a software product audit. Thus viewed, the quality criteria gathered in the input stage of the audit process can be seen as a set of architectural design decisions that outline the desired state, or *Soll*-architecture, of the software product.

3. Related Work

Several researchers have proposed ontologies or reasoning frameworks for software architecture and software quality. Unfortunately, none is directly applicable to the selection of quality criteria in the input stage of a software product audit.

SEI's ArchE tool [6], for instance, serves a goal that is similar to ours. It targets, albeit from a forward engineering point of view, guidance of the architect in selecting the right tactics given certain quality requirements. However, it relies on quantitative reasoning models, which are available for some quality attributes (e.g., performance or modifiability) but not for others [1]. This limits its applicability to only a subset of the quality attributes that one may need to consider in a software product audit.

Erfanian and Shams Aliee proposed an ontological approach to architecture evaluation [7]. Their approach differs from ours in that it targets the throughput stage of a quality assessment. In their method, the actual architecture of the software system is formally codified and compared with a set of tactics and their effects on quality attributes. Our work, on the other hand, targets selection of quality criteria in the input stage of a quality assessment, and therefore does not rely on the formal codification of the actual architecture.

Another quality related ontology originated in the context of service engineering. SECSE's Quality of Service ontology (QoSont) bridges a potential semantic gap between service suppliers and service consumers by providing thorough definitions of Quality of Service properties and their metrics [14]. Although this ontology provides a formal approach to quality attribute specification, it does not address the way in which the desired level of quality can or cannot be attained. This black box perspective on quality contrasts with the white box perspective of our work.

Kruchten, finally, has proposed an ontology for architectural design decisions [11]. This ontology has been driven mainly by the need to document architectural design decisions, either on-the-fly or after-the-fact, to support evolution and maintenance of complex software-intensive systems. Unlike the ontology we propose, it has not been specifically designed to support reuse of preexisting knowledge. Nevertheless, it exhibits many features that can be translated to the world of quality criteria, given our perspective on quality criteria as architectural design decisions for a *Soll*-architecture.

4. Crises and criteria

Kruchten argues that there are three major classes of decisions (or 'crises') [11]: ontocrises (existence decisions, with their counterpart anticrises or non-existence decisions), diacrises (property decisions), and pericrises (executive decisions). Likewise, we can distinguish three analogous major classes of criteria, each of which has to be assessed using a different approach:

- *Ontocriteria* (existence criteria) represent concrete elements or artifacts that must appear in the software product. Anticriteria, their counterpart, represent elements that must not appear in the product. Both ontocriteria and anticriteria can be traced to a single product artifact. An example of an ontocriterion could be 'there must be a custom API for communication with back-end systems'. An example anticriterion could be 'there must be no code duplication'.

  This class of criteria are the easiest to assess, since finding a single instance of some element in the software product suffices to prove (for ontocriteria) or disprove (for anticriteria) that the product satisfies the criterion. However, most ontocriteria need to be complemented with diacriteria, another class of criteria, since the mere presence of a certain element in the product is not enough to warrant the desired level of quality. In the case of the above example, the existence of the API must be followed by its use. In actual software product audits, those complementary criteria may be implied by the ontocriterion instead of being fully specified.

- *Diacriteria* (property criteria) represent properties ('overarching traits') that hold for the whole system and cannot be traced to a single product artifact. An example diacriterion could be 'The rationale of design decisions must be documented', or – as a complement to the example ontocriterion above – 'All communication with back-end systems must use the custom-made API'. Diacriteria can be quite difficult to assess, since they cannot be traced to a single artifact and therefore require a full and thorough examination of the complete software product. A feasible alternative is to assess diacriteria using spot checks.

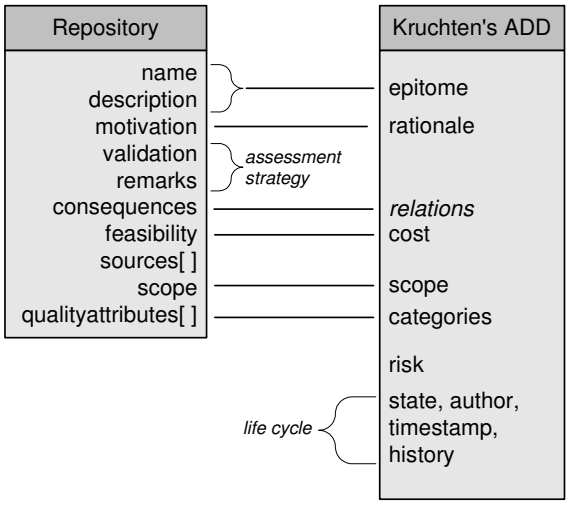

Figure 1.

- *Pericriteria* (executive criteria) are criteria that are not directly related to the software product's properties or quality attributes, but to criteria surrounding the audit process itself. An example pericriterion could be 'All stakeholders agree on the evaluation plan'. Whenever pericriteria are not satisfied, the quality of the audit process itself is in jeopardy.

In addition to different classes of decisions, Kruchten specifies several attributes that have an almost one-to-one correspondence with quality criteria. Fig. 1 shows a comparison between attributes from DNV-CIBIT's quality criteria repository and Kruchten's architectural design decisions.

From Fig. 1 we see that the decision attributes epitome, rationale, cost, and scope have a direct counterpart in the quality criteria repository, with one notably different interpretation: for quality criteria, the 'cost' involved in selecting a criterion relates to the feasibility of assessing the product's conformance with that criterion. There are also some differences between the two models, which are a direct result of the different use goals of the repository (reuse) and ontology (documentation):

- Attributes related to a decision's life cycle (state, author, timestamp, history) are not present in the repository;
- Attributes related to a quality criterion's assessment strategy are not present in the decision ontology;
- Since the decision ontology inherently models product-specific decisions, there are no 'sources' to keep track of;
- The risk related to (non-compliance with) a quality criterion would be the *outcome* of an audit process and would not be part of the input, hence there is no 'risk' attribute in the repository.

Finally, there are several comparable attributes that have slightly different representations in the two models:

- *Categories* in Kruchten's ontology are an open-ended list that may include concerns and quality attributes. In our ontology, we draw on the notion of concerns (which include quality attributes) as first class entities. This notion is central to the decision loop that is part of our core model of architectural knowledge [4], and enables us to a) further define relations between quality attributes to form a quality model such as the ISO 9126 quality model [10], and b) to make inferences (including trade-off analyses) from the relations between criteria and quality attributes (and between quality attributes themselves).
- *Relations* are much more extensively defined by Kruchten than in the repository, where they are just a textual description of 'consequences'. Kruchten recognizes 10 types of decision-to-decision relationships that can be transferred directly to criterion-to-criterion relationships. Two additional relationships ('traces from/to' and 'does not comply with') are the type of relationships auditors would use during the throughput stage of an audit to relate quality criteria to architectural design decisions and other artifacts in the software product (e.g. method X in class Y *does not comply with* criterion 'no code duplication').

Moreover, from discussions with DNV-CIBIT's auditors we know that, apart from direct relationships between criteria, one other important piece of knowledge should be captured. In essence, the effects from quality criteria on quality attributes must be reified, so that comparisons between the effects can be made. This makes it possible to capture such statements as '1024 bit encryption has a higher positive effect on security than 512 bit encryption'.

Figure 2 depicts the essential concepts and relationships of the ontology we propose based on our analysis. It consists of the following elements:

QualityCriterion is the main element in the ontology. It has the following attributes, as discussed above: name, description, motivation, validation, remarks, feasibility, and scope. Four types of criteria can be distinguished: ontocriteria, anticriteria, diacriteria, and pericriteria. The *isRelatedTo* relationships capture the way in which a quality criterion can be related to other quality criteria, following the ten types of relationships defined by Kruchten [11].

QualityAttribute represents a quality characteristic that can be further specialized in subcharacteristics. For example, in ISO 9126 'efficiency' is further divided into

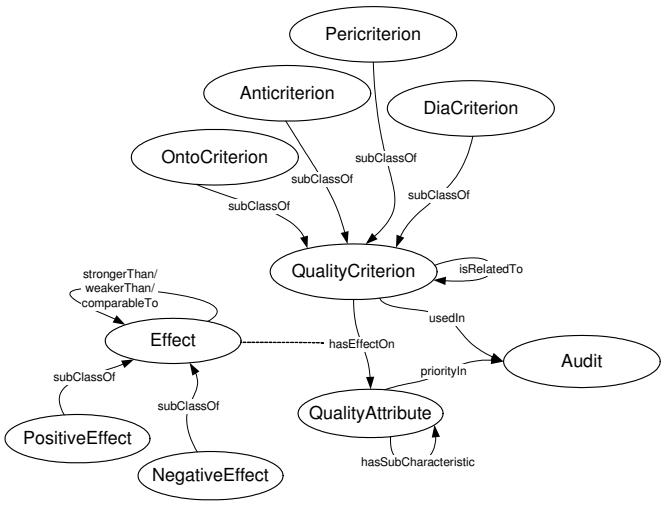

Figure 2.

'time behaviour', 'resource utilisation', and 'efficiency compliance' [10].

Effect is a reified relation from criterion to quality attribute, with two attributes of itself: type (positive or negative) and 0 to n reciprocal relations to other 'effect' relationships, which indicates the relative strength of each of the 'effect' relations.

Audit represents a software product audit in which particular quality criteria have been used to assess a prioritized set of quality attributes. The *usedIn* relation captures the relation between criteria and audits. The *priorityIn* relation captures the relation between quality attributes and audits.

When the ontology from Fig. 2 is used to codify quality criteria as reusable assets, support for the selection of those criteria in the input stage of a software product audit becomes possible.

5. QuOnt in Action: Selecting Quality Criteria for Reuse

In order to support reuse of quality criteria, the ontology from Fig. 2 must be instantiated first. To illustrate the value of our approach, we discuss two scenarios in which we instantiate QuOnt. For reasons of confidentiality, we are not able to disclose quality criteria used by DNV-CIBIT. We therefore instantiate the ontology with several tactics described by Bass et al. [2, Ch.5], since – as we have seen in Section 2 – tactics may serve as quality criteria, provided their positive *and* negative effects on quality attributes are taken into account.

Based on the ten types of relations described by Kruchten, we can specify several constraints that the ontology instantiation must satisfy. Let x and y be quality criteria, let $\text{relation}_{x,y}$ denote a relation from criterion x to criterion y, and let select_x denote the selection of criterion x. Then the following ontological constraints on the relationships between x and y hold (cf. [11, 8]):

C1 $\forall x, y : \text{enables}_{x,y} \Rightarrow \neg\text{constrains}_{x,y}$

C2 $\forall x, y : \text{constrains}_{x,y} \Rightarrow (\neg\text{select}_x \Rightarrow \neg\text{select}_y)$

C3 $\forall x, y : \text{constrains}_{x,y} \wedge \text{constrains}_{y,x} \Rightarrow \text{isBoundTo}_{x,y}$

C4 $\forall x, y : \text{isBoundTo}_{x,y} \Rightarrow \text{isBoundTo}_{y,x}$

C5 $\forall x, y : \text{forbids}_{x,y} \Rightarrow (\text{select}_x \Rightarrow \neg\text{select}_y) \vee \text{overrides}_{y,x}$

C6 $\forall x, y : \text{subsumes}_{x,y} \Rightarrow (\text{select}_x \Rightarrow \text{select}_y)$

C7 $\forall x, y : \text{conflicts}_{x,y} \Rightarrow \text{forbids}_{x,y} \wedge \text{forbids}_{y,x}$

C8 $\forall x, y : \text{overrides}_{x,y} \Rightarrow \text{forbids}_{y,x} \wedge \text{select}_x$

C9 $\forall x, y : \text{alternative}_{x,y} \Rightarrow \neg(\text{select}_x \wedge \text{select}_y)$

C10 $\forall x, y, z : \text{alternative}_{x,y} \wedge \text{alternative}_{y,z} \Rightarrow \text{alternative}_{x,z}$

C11 $\forall x, y : \text{comprises}_{x,y_{1,2,\ldots,n}} \Rightarrow (\neg\text{select}_x \Rightarrow \neg\text{select}_{y_1} \wedge \neg\text{select}_{y_2} \wedge \ldots \wedge \neg\text{select}_{y_n})$

C12 $\forall x, y : \text{depends}_{x,y} \Rightarrow \text{constrains}_{y,x} \vee \text{comprises}_{y,x} \vee \text{overrides}_{x,y}$

These constraints on the one hand confine the facts that can be specified in a QuOnt-based knowledge base, and on the other hand may act as production rules that support the selection of quality criteria. Fig. 3 graphically depicts a small knowledge base comprised of an instance of our ontology. This instance is valid under the constraints outlined above, and consists of the following elements:

OntoCriterion OC1 Use passwords

OntoCriterion OC2 Single sign on

DiaCriterion DC1 Secure access

Effect E1 OntoCrit1 hasPositiveEffectOn Security

Effect E2 OntoCrit2 hasPositiveEffectOn Security

Effect E3 OntoCrit1 hasNegativeEffectOn Usability

Effect E4 OntoCrit2 hasNegativeEffectOn Usability

Effect E5 OntoCrit1 hasNegativeEffectOn Maintainability

Effect E6 OntoCrit2 hasNegativeEffectOn Maintainability

Relation R1 E1 comparableTo E2

Relation R2 E3 strongerThan E4
(consequently E4 weakerThan E3)

Relation R3 E5 weakerThan E6
(consequently E6 strongerThan E5)

Relation R4 OntoCrit1 alternativeTo OntoCrit2

Relation R5 DiaCrit1 constrains OntoCrit1, OntoCrit2

Given a list of prioritized quality attributes, the knowledge base can be used to support the auditor's decisions which quality criteria could be included. In short, all criteria with a known effect on one of the high-priority quality attributes (including the attribute's sub-characteristics), should be selected and serve as an assessment baseline in the throughput stage of the audit.

For instance, suppose that our customer indicates that the following quality attributes are the most important ones, in the order presented:

1. Security
2. Usability
3. Maintainability

A valid question would be: which of the codified quality criteria from our knowledge base should we select, given this list of prioritized quality attributes? Through a stepwise inference, we can arrive at the answer to this question. In the following list of steps, the abbreviations refer to elements from the knowledge base (OC, DC, E, R) or to constraints on the relationships (C).

Step 1 Both OC1 and OC2 have a positive effect on security (E1,E2), and a negative effect on usability (E3,E4).

Step 2 Security has a higher priority than usability, hence both OC1 and OC2 are candidates to be selected as criteria for this audit. However, since they are alternatives (R4) only one of them may be selected (C9), so we need to make a choice between the two.

Step 3 Although both criteria have a comparable positive effect on security (R1), the negative effect on usability from OC1 is stronger than that from OC2 (R2). Hence, OC2 should be preferred over OC1. We therefore select 'single sign on' as a quality criterion in this audit.

Step 4 Since DC1 constrains OC1 and OC2, none of the ontocriteria can be selected without the selection of DC1 (C2). Since OC2 has been selected, DC1 must be selected too. Hence, we select 'Authenticate Users' as a quality criterion as well.

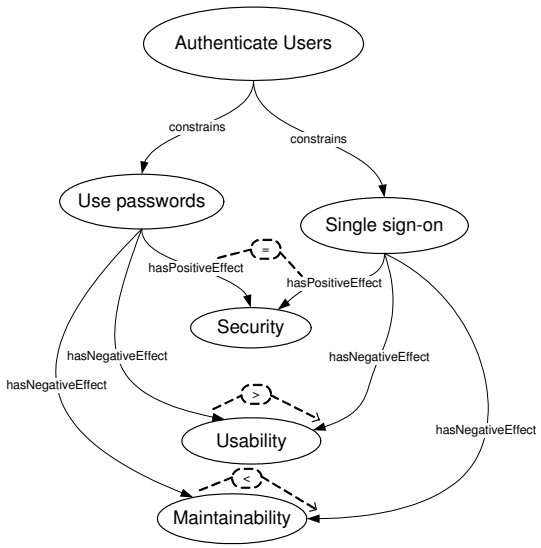

Figure 3.

In summary, the four inference steps lead to the conclusion that, given the provided list of prioritized quality attributes, the software product should have a mechanism for user authentication and, more specifically, that this authentication should take place using passwords. Obviously, should the quality attributes have been prioritized differently, the outcome of the inference would have been different too. If, for example, maintainability would have been deemed more important than usability, the third step would have led to the conclusion that OC1 should be selected instead of OC2, since R3 (the weaker negative effect of OC1 on maintainability) would have been of greater significance than R2 (the stronger negative effect of OC1 on usability).

Let's investigate a slightly more complicated scenario. Again, we show a graphical depiction of a knowledge base that is valid under the relational constraints (Fig. 4). To reduce the complexity of the figure, we have represented the effects of a criterion on a quality attribute with an annotation instead of a reified relation; for this example we assume that the criteria's effects on quality attributes are of comparable strength.

Figure 4 shows a total of 11 criteria: 5 diacriteria (prevent ripple effect, use standard API, authenticate users, target Java platform, and target dotNet platform) and 6 ontocriteria. Three ontocriteria (information hiding, use intermediary, and maintain interface) have a positive effect on maintainability. The three other ontocriteria (use JAAS, use COM+ security call, and develop in-house authentication module) have a positive effect on security. Additionally, the ontocriterion 'use intermediary' has a negative effect on performance due to the overhead it introduces.

The elements from Fig. 4 can be specified in a QuOnt-

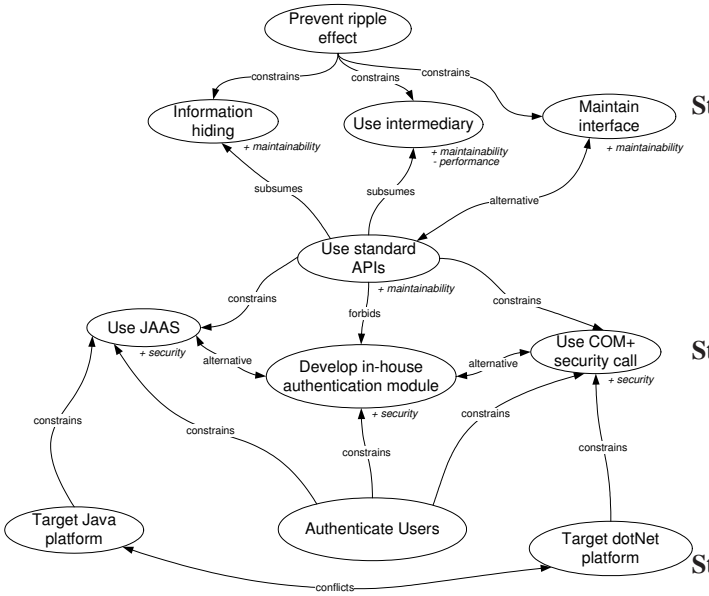

Figure 4.

based knowledge base, much the same as the elements from Fig. 3 above. For reasons of brevity, however, we will omit this specification and proceed directly with a discussion of inference steps.

Suppose that our customer again indicates that the following quality attributes are the most important ones:

1. Security
2. Usability
3. Maintainability

Additionally, suppose the criterion 'Target Java Platform' must be selected, for example because this has been a preexisting customer requirement. The question, again, is: which of the codified quality criteria from our knowledge base should we select, given this list of prioritized quality attributes? Inference of the answer to this question goes along the following lines.

Step 1 Since security has top priority, 'Use JAAS', 'Use COM+ security call', and 'Develop in-house authentication module' are all candidate criteria to be selected.

Step 2 Since the three candidate criteria are alternatives, only one of them can be selected (C9). Moreover, since 'target Java platform' has been preselected and conflicts with 'target dotNet platform', selection of the dotNet platform is ruled out (C7, C5).

Step 3 Two of the security-related criteria remain: JAAS and in-house development. Both, we assumed, have comparable effects on security and consequently for the time being there is a tie between the two.

Step 4 None of the criteria in our knowledge base have a known effect on usability, but there are three ontocriteria with a positive effect on maintainability. The three criteria are not alternatives, hence all three of them could in principle be selected. Performance is not an issue in this audit, so the negative effect of 'use intermediary' on performance does not inhibit its selection.

Step 5 There is a higher-level diacriterion 'Use standard API' that subsumes 'information hiding' and 'user intermediary'. The use of standard APIs has an added positive effect on maintainability, and is an alternative to 'maintain interface' which presumes control over a particular interface.

Step 6 The use of a standard API is to be preferred, since its combined positive effect on maintainability is stronger than the effect of its alternative 'maintain interface'.

Step 7 The use of a standard API forbids the development of an in-house authentication module. Hence, unless we override this constraint (C5, C8), the criterion 'use JAAS' has to be selected.

In summary, the inference steps lead to the conclusion that, given the provided list of prioritized quality attributes, the software product should have mechanisms to authenticate users and to prevent a ripple effect. More specifically, the use of standard APIs should prevent a ripple effect from occurring and (therefore) JAAS should be used to implement authentication. Here again, should the quality attributes have been prioritized differently the outcome of the inference would have been different too. In particular, the use of JAAS was mandated by the preexisting requirement that the software product should target the Java platform.

From the latter scenario it is obvious that our ontology cannot support a fully automated procedure for quality criteria selection. Certain decisions, such as whether or not the criterion that a standard API be used may be overridden for the authentication of users (see step 7), need to be taken by the auditor – preferably in conjunction with the customer.

Nevertheless, the type of inference shown in the two scenarios can be implemented in a semi-automated, interactive decision support system. Such a system would iteratively consider the important quality attributes, starting at the one with the highest priority, and identify quality criteria that are candidates to be selected. Whenever a criterion can unequivocally be selected, i.e. when there are no ties and its selection does not violate the relational constraints, the system can automatically decide upon its selection into the set of quality criteria to be used in the audit. We have seen this happen in the first scenario we discussed. In the case of ties

between criteria, further inference based on a lower priority quality attribute may resolve the situation as we have seen in the second scenario. If the tie cannot be resolved automatically, or when selection of a candidate criterion would result in inconsistencies under the relational constraints, the system would need to request input from the user to resolve the resulting deadlock. This happened in the final inference step of the second scenario.

A prototype decision support system has been implemented and presented to DNV-CIBIT's auditors [12]. Although the prototype only supported partial inference (especially consistency checking of the selected quality criteria), it showed the potential of the approach we propose in this work. It also brought to light some points of attention, most notably the need for proper visualization of the knowledge base's contents.

6. Conclusion

In this paper we have presented an ontology that supports the reuse of quality criteria in software product audits. We derived this ontology from the perspective that quality criteria can be viewed as decisions upon a *Soll*-architecture of a software product. We have demonstrated analogies between a theory of architectural design decisions and quality criteria. Based on those analogies, we have proposed our ontology, the value of which we demonstrated by walking through two scenarios of quality criteria selection.

Although our ontology cannot support a fully automated procedure for quality criteria selection, the type of inference discussed in the two scenarios can be implemented in a semi-automated decision support system. An early prototype implementation of the ideas presented in this work showed the potential of our approach.

We have presented our work from the perspective of product audits and quality criteria. Given the decision characteristics of quality criteria, however, we have no reason to believe that our results are limited to this perspective alone. On the contrary, we conjecture that scenarios similar to the ones described in this paper can also be used in a forward engineering setting. In that sense, our work shows the added value of maintaining explicit and reified relations to quality attributes, for quality criteria and design decisions alike.

Acknowledgments

This research has been partially sponsored by the Dutch Joint Academic and Commercial Quality Research & Development (Jacquard) program on Software Engineering Research via contract 638.001.406 GRIFFIN: a GRId For inFormatIoN about architectural knowledge.

We thank our Master student Juan Carlos Loza Torres for his effort in the construction of a prototype knowledge base for quality criteria selection, which contributed to the ideas in this work. In addition, we thank all auditors of DNV-CIBIT with whom we have exchanged thoughts, in particular Mark Hissink Muller, Matthijs Maat, Frank Niessink, and Viktor Clerc.

References

[1] F. Bachmann, L. Bass, and M. Klein. Deriving Architectural Tactics: A Step Toward Methodical Architectural Design. Technical report, Software Engineering Institute, March 2003.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley Pearson Education, Boston, second edition, 2003.

[3] R. C. de Boer and R. Farenhorst. In Search of 'Architectural Knowledge'. In *3rd Workshop on SHAring and Reusing architectural Knowledge (SHARK)*, Leipzig, Germany, 2008.

[4] R. C. de Boer, R. Farenhorst, P. Lago, H. van Vliet, V. Clerc, and A. Jansen. Architectural Knowledge: Getting to the Core. In *Third International Conference on Quality of Software-Architectures (QoSA)*, volume 4880 of *LNCS*, pages 197–214. Springer, 2007.

[5] R. C. de Boer and H. van Vliet. On the Similarity between Requirements and Architecture. *Journal of Systems and Software*, In Press.

[6] A. Diaz-Pace, H. Kim, L. Bass, P. Bianco, and F. Bachmann. Integrating Quality-attribute Reasoning Frameworks in the ArchE Design Assistant. In *4th International Conference on the Quality of Software Architecture (QoSA)*, University of Karlsruhe (TH), Germany, 2008.

[7] A. Erfanian and F. Shams Aliee. An Ontology-Driven Software Architecture Evaluation Method. In *3rd Workshop on SHAring and Reusing architectural Knowledge (SHARK)*, Leipzig, Germany, 2008.

[8] R. Farenhorst and R. C. de Boer. Core Concepts of an Ontology of Architectural Design Decisions. Technical Report IR-IMSE-002, Vrije Universiteit, September 2006.

[9] R. Farenhorst and R. C. de Boer. Knowledge Management in Software Architecture: State of the Art. In M. Ali Babar, T. Dingsøyr, P. Lago, and H. van Vliet, editors, *Software Architecture Knowledge Management: Theory and Practice*. Springer, Under submission.

[10] ISO/IEC. Software engineering - Product quality - Part 1: Quality model. Technical Report ISO/IEC 9126-1, 2001.

[11] P. Kruchten. An Ontology of Architectural Design Decisions in Software-Intensive Systems. In *2nd Groningen Workshop on Software Variability Management*, Groningen, NL, 2004.

[12] J. C. Loza Torres. *Reusability of Quality Criteria in Software Audits*. Master's thesis, Vrije Universiteit, 2008.

[13] J. M. Owen. Tacit Knowledge in Action: Basic Notions of Knowledge Sharing in Computer Supported Work Environments. In *European CSWC Workshop on 'Managing Tacit Knowledge'*, Bonn, 2001.

[14] P. Sawyer and N. Maiden. How to Use Web Services in Your Requirements Process. *IEEE Software*, 26(1):76–78, 2009.