Categorization of Software Quality Patterns*

Klaus Lochmann[1], Stefan Wagner[1], Andreas Goeb[2], Dominik Kirchler[2]

[1] Technische Universität München
Garching, Germany
{lochmann, wagnerst}@in.tum.de

[2] SAP Research
Darmstadt, Germany
{andreas.goeb, dominik.kirchler}@sap.com

Abstract. In software systems recurring patterns are often observed and have been collected and documented in different forms, as for example in development guidelines. These well-known patterns are utilized to support design decisions or to automatically detect flaws in software systems. For the most part, these patterns are related to software quality issues and can also be be referred to as software quality patterns. These quality patterns have to be communicated to the various roles contributing to the development of software, e.g., to software architects or developers. Hence, a comprehensive scheme to categorize the various types of patterns is needed to support effective communication. The categories should be shaped so that each category can be communicated to a single organizational role in a company. Since each pattern refers to a specific concept of the software system, a categorization based on system modeling concepts is used. The presented categorization scheme is grounded on activity-based quality models that are already used to collect different patterns related to the quality of software systems. Based on two case studies the applicability of the categorization scheme is shown. Real-world models were categorized using the scheme and the resulting distribution of entities within the different classes is discussed.

1 Introduction

In practical software engineering, a major challenge is to develop software of high quality. Professional developers use proven best practices and experiences to tackle it. In order to pass these best practices to inexperienced developers they document recurring patterns. Sources for such patterns are, for example, coding guidelines, style guides, bug patterns, and architectural patterns. Style guides for source code improve its readability by applying a consistent format. Bug patterns are used to detect and classify defects in software.

The term *quality pattern* as used by Houdek and Kempter [1] describes a structured way to document and reuse quality-related experience. The authors define a framework based on Goal/Question/Metric-Approach that contains an abstract, a problem statement in a specified context as well as a solution. Our understanding of the term *quality pattern* is more general. All product-related

* This work has partially been supported by the German Federal Ministry of Education and Research (BMBF) in the project QuaMoCo (01 IS 08023B/D).

patterns occuring in software that are related to quality are seen as *quality patterns* by us. The exact definition of the term is based on quality models and will be given in section 2.

Patterns and their relation to quality attributes of a system are represented in quality models. The first quality model that distinguishes between quality attributes and quality-influencing properties of a system is that of Dromey [2]. In this paper the approach of Deissenboeck et al. [3], which extends Dromey's ideas, is used because it sets development activities in relation to properties of the system. These properties are essentially describing quality patterns. In recent work, such quality models are used to define how to achieve quality in a software system [4,5]. All these quality models focus on software product quality by defining concepts related to the software product itself as opposed to process-related concepts.

Problem. To efficiently communicate the information contained in quality models a categorization is needed that supports their communication. Many existing quality models provide a categorization according to quality attributes. Yet, especially for communicating the patterns to different audiences and roles, this categorization is not optimal, because it does not divide the quality patterns alongside the needs of these organizational roles. Furthermore, quality models in practice tend to be large, containing hundereds of patterns. The handling and communicating of such models posed additional challenges.

Contribution. In this paper we propose an approach for classifying quality patterns according to abstraction levels of the system they are referring to. Each abstraction level is concerned with different concepts that are relevant for different development steps. Since the different development steps are performed by different roles within a company, corresponding quality patterns can be communicated to these roles.

2 Activity-Based Quality Models

Quality models have been used to describe the concept of software quality for decades [6,7]. Several authors argue that facts describing the system have to be separated from quality aspects. Dromey [2], for example, introduces a quality model that distinguishes between quality-carrying properties and quality attributes.

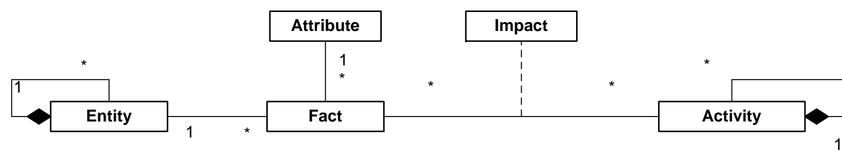

Fig. 1: Meta-Model: Activity-Based Quality Model

Existing quality models often use high-level "-ilities" for defining quality on a very coarse level, which is reported to be hard to operationalize [3]. To overcome this shortcoming, activity-based quality models (ABQM) were proposed [8]. The idea there is to break down quality in detailed facts and their influence on activities performed on the system. An example for a fact would be "unambigousness of source code", which has an influence on the activity "code reading".

An ABQM is based on a defined meta-model (see Fig. 1) whose elements are described in the following. The beforementioned facts are modeled as a composition of entities and attributes. The entities describe a hierarchically structured decomposition of the system and corresponding documents. Entities can be technical concepts of programming languages such as source code expressions but also more abstract concepts such as inheritance structure. An entity that is characterized by an *Attribute* is called *Fact*, e.g., [Source Code | UNAMBIGUOUSNESS] denotes unambiguous source code. Also the activities are organized in a hierarchy. A top-level activity maintenance has sub-activities such as code reading, modification, and testing. The impacts define a relationsship between facts and activities. They describe which fact has an influence on an activity. For denoting the previous example, the following notation is used: [Source Code | UNAMBIGOUSNESS] $\xrightarrow{+}$ [Code Reading].

ABQMs have been used for modeling maintainability [3], usability [9], and security [4]. In these case studies existing knowledge in form of guidelines, checklists, standards, etc. has been modeled as an ABQM. The occuring patterns found in these documents are modeled as facts in the ABQM, whereby the impact on activities gives a justification for the relevance of the fact.

3 Categorization of Entities

Due to the fact the entities of the ABQM are referring to very different concepts and abstraction levels of a system and the fact that real quality models are very large (e.g., 142 entities in [3]), the task of communicating the significant parts of the model to the appropriate people at the right phase during development is challenging. Since the activities are more related to the concerns of stakeholders, they are not suited as a means for categorization regarding roles in a company. A software architect, for example, has to take both maintainability and performance issues into regard.

In the following we propose an approach to structure the entities in a way that is suitable for the goals described above. The most general way of categorization is to distinguish between entities referring to the software system itself, and those referring to documents secondarily produced, like documentation (see Fig. 2a). Although ABQMs are also used to model non-system entities, like the development environment or pocess characteristics, we focus on product related entities, because they make up a strong majority of entites in our case studies. We distinguish between system documentation that describes the inner structure and working of the system, and the user documentation like operator manuals etc. Since all entities referring to the software system in some way represent a

model of it, we will use existing concepts from existing modeling techniques to structure them.

According to Schätz et al. [10] we distinguish between horizontal and vertical abstraction (see Fig. 2b). Horizontal abstraction introduces a separation of concerns with the distinction between structure, behavior, and data. Vertical abstraction introduces levels of abstraction, whereby the lower the level of abstraction, the more details are visible. An example for levels of abstraction are black-box description, white-box description, and the technical implementation of a system.

For each abstraction level research has been conducted and sophisticated models are available that we exploit here. Broy et al. [11] describe the architecture of embedded software-intensive systems on different levels of abstraction. In this work we use the concepts depicted in Fig. 2b and described as follows.

Black-box level On this level of abstraction the inner working of the system is omitted. Therefore, only the interface of the system to its environment and the user-visible behavior are described.

- *Structural Concepts* describe the structure of the system. Since, on the black-box level the internal structure is not visible, only externally visible structure is described here.
- *Behavioral Concepts* describe the behavior of the system as seen at the system boundary by its environment or rather the user.
- *Interface/Data Concepts* define the syntactic interface of the system and the data types that are used by it.

White-box level On this level of abstraction the system is described as a white box, i.e., the internal components of the system are visible. This level is also called *logical architecture*.

- *Structural Concepts* on this level are the components the system consists of, and the way in which the components are connected to each other.

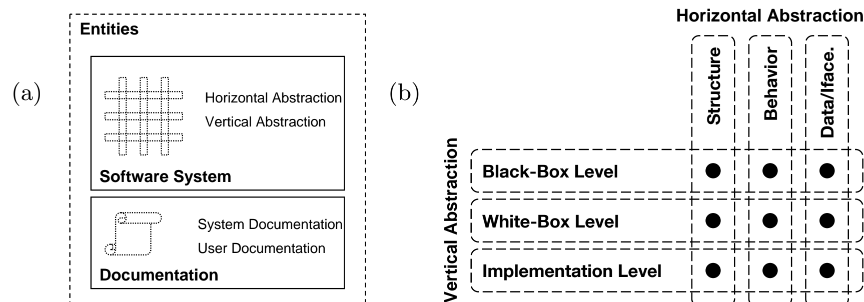

Fig. 2: Categorization of Entities

- *Behavioral Concepts* on this level define the behavior of each component.
- *Interface/Data Concepts* describe the interfaces of the components.

Implementation level On this level, the actual implementation of the system is described. It consists of both software and hardware elements.

- *Structural Concepts* describe the structure of the software, e.g., the source-code, configuration files, and hardware, e.g., CPUs, memory, and bus systems.
- *Behavioral Concepts* describe the behavior of the elements on this level. These can be timing constraints of the hardware but also behavioral concepts of the programming language as for example pointers and garbage collection.
- *Data Concepts* describe the data structures used on the source code and hardware level.

4 Case Study: TUM Maintainability Model

At Technische Universität München (TUM) an ABQM for maintainability was developed in multiple case studies [3,8]. This quality model was used for the generation of guidelines as well as for the evaluation of the results of static code analysis tools. The model consists of 152 entities, 205 facts, and 30 activities. It has to be noted, that some entities are used for structuring and are actually not referring to system concepts at all. These entities typically have no facts associated with them and were omitted in the analysis. There were 10 entities of this kind.

Example In Tab. 1 examples of the TUM maintainability model are shown. In the first row of the table the fact [Distribution | EXISTENCE] is defined. This fact just expresses whether there are distributed parts within the system or not. Because it relates to the concept that the system consists of different components, it was categorized as “white-box / structure”.

Another example is that of the third row: [Recursion | EXISTENCE]. The existence of recursive function calls is clearly related to the implementation level, and because it refers to the behavior of the code, it was categorized as “implementation / behavior”.

These two examples clearly show that the categorization is suitable for the communication of facts. The white-box facts, which would be communicated to software architects, contain information on architectural issues (in this case, distribution). The implementation facts, which would be used for coding-guidelines, would express rules for developers, in this case the use of recursive functions would be forbidden.

Discussion In Tab. 2 the results of the categorization are visualized. It can be seen that most of the entities are situated on the implementation level. However, this result is not surprising taking into account that the model was built using

Table 1: Examples of the TUM maintainability model

Entity	Attributes and Description		Category
distribution	*existence*:	"Does the system have distributed parts?"	White-box / Structure
boolean_expression	*accessibility*:	"Within a boolean expression, no assignments should be made."	Impl. / Structure
	completeness:	"Boolean expressions have to be completely bracketed."	
recursion	*existence*:	"Are there recursive function calls?"	Impl. / Behavior
runtime-checks	*existence*:	"Does the language provide runtime checks, e.g. for array bounds?"	Impl. / Behavior
design-decisions	*existence*:	"Are design-decisions documented?"	Documentation / System
	completeness:	"Is the documentation of design-decisions complete?"	

Table 2: Categorization of TUM maintainability model

(a) Software system entities

	Structure	Behavior	Data / Iface
Black-Box	1	0	0
White-Box	3	1	0
Implementation	89	11	9

(b) Documentation entities

Type	#
User Documentation	0
System Documentation	28

coding guidelines and static code analysis tools. These sources of information typically refer to the source code and not to high-level concepts of software systems. It has to be noted that also architectural issues are hardly present in this model. The entities categorized as system documentation, however, are available in a significant number.

5 Case Study: SAP Security Requirements

In this case study, the SAP Product Standard [12] for security was examined with respect to the defined categorization scheme. In a first step, all the quality requirements contained in the Product Standard were transferred into the proposed ABQM structure. In total, 205 different requirements were analyzed and modeled. To model the activity hierarchy, we used the scheme proposed by Wagner et al. [4]. The constructed entity types were categorized according to the definitions above. The total number of categorized entities was 121. The results are shown in Tab. 3. Note that more than one requirement may refer to a specific

entity and that requirements referring to the same entity may do so at different abstraction levels.

For example the entity *password* can refer to the concept that a secret phrase is used for authorization as well as to the source code variable for storing the password string and also to its value. One can easily imagine corresponding requirements: (1) "sensitive data should be protected by a password", (2) "the memory storing a password has to be overwritten after use", and (3) "a password may never be printed in clear text to log files".

In these cases the same entity was counted once for each pair of abstraction level/concept. This explains that the sum of all entries in Tab. 3 is 166 instead of 121.

Discussion As seen above, some requirements may refer to the same entity at different abstraction levels or concepts. This shows that it was not always possible during the modeling process to deduce entities from requirements in such a way, that they could be uniquely categorized afterwards.

To validate our approach, categorization of requirements and categorization of entities should yield similar distributions of results. This can be seen by comparing Tab. 3 and Tab. 4, the latter summarizing the categorization results of the original requirements.

Generally, the question arises whether it makes sense to classify the entities in comparison to classifying the single requirements. Obviously this is the case, if the number of requirements is significantly higher then the number of resulting entities (including facts and descriptions). In other cases, it may seem to be the easier way to just classify the requirements. However, having an ABQM-like structured quality model offers several additional benefits, as described in section 2. Whether these justify the extra effort of transferring the requirements into the quality model depends on the objective and the type of analysis that is intended to be conducted with the quality model.

6 Discussion & Conclusion

In this paper we presented a method for categorizing software quality patterns, which are modeled using an activity-based quality model. The categorization is based on software modeling concepts and on different abstraction levels used

Table 3: SAP model entities

(a) System entities

	Structure	Behavior	Data / Iface
Black-Box	15	23	27
White-Box	13	26	22
Implementation	20	1	1

(b) Documentation entities

Type	#
User Documentation	7
System Documentation	11

Table 4: SAP requirements

(a) System requirements

	Structure	Behavior	Data / Iface
Black-Box	16	36	32
White-Box	16	37	24
Implementation	22	1	1

(b) Documentation requirements

Type	#
User Documentation	7
System Documentation	13

herein. The patterns of the different categories could be suited for communicating the patterns to different organizational roles or to designate responsible persons for eliciting and maintaining the patterns.

The categorization scheme was evaluated in two case studies. The first case study relied on an already existing model that was constructed using coding guidelines and evaluation rules of static code analysis tools. In the second case study a real-world requirements list was modeled and then categorized.

These case studies showed that the categorization scheme is applicable to real-world models. In the second case study we can see that the number of entities in each category is quite evenly distributed with the tendency that higher abstraction levels are more present. Thus we conclude that the categorization scheme does actually define categories that are present in real documents.

By comparing the two case studies, we can see that the nature of the models is strongly reflected in the distribution of the category sizes. In the second case study a generic requirements list was modeled that contained very different requirements. This is reflected by the equal distribution of the category sizes. However, in the first case study coding guidelines and metrics of static code analysis tools were modeled, which resulted in entities that were mostly categorized as referring to implementation structure.

Moreover, the categorization of the model promoted a deeper understanding of the model itself. During the categorization the modeler had to think about the exact meaning of the entities. In the quality model that meaning was often not explicitly documented. Therefore, during the categorization it was discovered that some entities were used ambiguously in the quality model. This deficiency was then corrected by splitting the entity in two entities, in order to reflect the two different meanings. In summary, the categorization also contributed to enhance the quality of the quality model itself.

A possible problem during categorization is the decision where to place specific concepts. Since all concepts are eventually implemented as source code, there appears the general tendency to classify all as implementation level. The categorization scheme must be applied in such a way that the patterns are classified at the highest possible abstraction level. If a pattern describes user-visible functionality, it has to be classified as black-box, even though the functionality itself is implemented in source code. If a pattern describes an implementation detail of one specific programming language, it is clearly related to the imple-

mentation level, because users and even software architects are not concerned with this detail.

In our future research we will focus on how the categorization scheme can be used to improve both communication and maintenance of software quality patterns. For the communication, it is possible to communicate the entities of different abstraction levels to different organizational roles. Implementation patterns are typically relevant for developers and can be used to generate guidelines, while the white-box patterns handle architectural issues and are therefore relevant for software architects. An adequately classified quality model can be used to integrate quality knowledge into the tools that are central to these organizational roles, e.g. the programming environment for the developer. Furthermore, the categorization scheme could be used to find adequate roles to maintain existing software quality patterns. Black-box patterns are mainly concerned with high-level concepts, which are primarily elicited and handled in requirements engineering. For the maintenance of adequate white-box patterns software architects are most probably the right group, while implementation patterns have to be developed by experts of programming languages. In future research these applications of the categorization scheme have to be empirically evaluated in case studies to prove their feasibility and usefulness. In addition, this evauation may lead to a refinement of the categorization granularity, should we find evidence that a higher level of detail will raise its practical value.

References

1. Houdek, F., Kempter, H.: Quality patterns—an approach to packaging software engineering experience. In: SSR '97: Proceedings of the 1997 symposium on Software reusability, New York, NY, USA, ACM (1997) 81–88
2. Dromey, G.R.: A model for software product quality. IEEE Transactions on Software Engineering **21**(2) (1995) 146–162
3. Deissenboeck, F., Stefan Wagner, Pizka, M., Teuchert, S., Girard, J.F.: An activity-based quality model for maintainability. In: Proceedings of the 23rd International Conference on Software Maintenance (ICSM 2007), IEEE Computer Society (2007)
4. Wagner, S., Mendez Fernandez, D., Islam, S., Lochmann, K.: A security requirements approach for web systems. In: Workshop Quality Assessment in Web (QAW 2009). (2009)
5. Wagner, S., Deissenboeck, F., Winter, S.: Managing quality requirements using activity-based quality models. In: Proc. 6th International Workshop on Software quality (WoSQ'08), ACM Press (2008) 29–34
6. Boehm, W.B.: Characteristics of Software Quality. North-Holland (1978)
7. McCall, A.J., Richards, K.P., Walters, F.G.: Factors in Software Quality. NTIS (1977)
8. Broy, M., Deissenboeck, F., Pizka, M.: Demystifying maintainability. In: Proceedings of the 4th Workshop on Software Quality, ACM Press (2006)
9. Winter, S., Wagner, S., Deissenboeck, F.: A comprehensive model of usability. In: Proceedings of Engineering Interactive Systems. LNCS, Springer (2007)
10. Schätz, B., Pretschner, A., Huber, F., Philipps, J.: Model-based development of embedded systems. In: Proc. Workshop on Advances in Object-Oriented Information Systems. LNCS. Springer-Verlag (2002) 331–336

11. Broy, M., Feilkas, M., Grünbauer, J., Gruler, A., Harhurin, A., Hartmann, J., Penzenstadler, B., Schätz, B., Wild, D.: Umfassendes Architekturmodell für das Engineering eingebetteter Software-intensiver Systeme. Technical Report TUM-I0816, TU München (2008)
12. Wroblewski, M.: Compliance testing of non-functional requirements at SAP. In: Quality Engineered Software and Testing Conference (QUEST'08). (2008)