Research Article **Open Access**

Reinhold Plösch*, Johannes Bräuer, Christian Körner, and Matthias Saft

Measuring, Assessing and Improving Software Quality based on Object-Oriented Design Principles

DOI 10.1515/comp-2016-0016
Received Jul 08, 2016; accepted Oct 21, 2016

Abstract: Good object-oriented design is crucial for a successful software product. Metric-based approaches and the identification of design smells are established concepts for identifying design flaws and deriving design improvements thereof. Nevertheless, metrics are difficult to use for improvements as they provide only weak guidance and are difficult to interpret. Thus, this paper proposes a novel design quality model (DQM) based on fundamental object-oriented design principles and best practices. In course of discussing DQM, the paper provides a contribution in three directions: (1) it shows how to measure design principles automatically, (2) then the measuring result is used to assess the degree of fulfilling object-oriented design principles, (3) and finally design improvements of identified design flaws in object-oriented software are derived. Additionally, the paper provides an overview of the research area by explaining terms used to describe design-related aspects and by depicting the result of a survey on the importance of object-oriented design principles. The underlying concepts of the DQM are explained before it is applied on two open-source projects in the format of a case study. The qualitative discussion of its application shows the advantages of the automated design assessment that can be used for guiding design improvements.

Keywords: software quality, design quality, design best practices, information hiding principle, single responsibility principle.

***Corresponding Author: Reinhold Plösch:** Department of Business Informatics – Software Engineering Johannes Kepler University Linz, Linz, Austria; Email: reinhold.ploesch@jku.at
Johannes Bräuer: Department of Business Informatics – Software Engineering Johannes Kepler University Linz, Linz, Austria; Email: johannes.braeuer@jku.at
Christian Körner: Corporate Technology Siemens AG, Munich, Germany; Email: christian.koerner@siemens.com
Matthias Saft: Corporate Technology Siemens AG, Munich, Germany; Email: matthias.saft@siemens.com

1 Introduction and Research Method

Approaches for measuring and assessing object-oriented design have received much attention since Chidamber and Kemerer proposed a metrics suite that measures essential properties of object-oriented design [1]. A number of approaches have adapted and extended this set of metrics to understand the characteristics of a particular object-oriented design. Besides measuring and assessing, decisions for refactoring can be identified and design improvements can be driven. However, it has been recognized that using metrics and considering them in isolation does not provide enough insight to address improvements [2].

A second way to assess design and to guide improvements is to detect design flaws referred to as *bad smells* [3]. For example, the approaches of Marinescu et al. [2] or Moha et al. [4] show that it is useful to identify and potentially fix these bad smells.

The first approach is still metric-centric and the second can be applied for the Java programming language only. A more recent approach has been published by Samarthyam et al., who pick up the idea of measuring design principles as one part of a design assessment [5]. Nonetheless, this approach refers to skills and knowledge of experts who manually assess the compliance of design principles. In conclusion, the authors point out that the community lacks a reference model for design quality [5].

This paper addresses the gap of the missing quality model and discusses three contributions for the community. (1) We propose a design quality model (DQM) based on the established concept of design principles, where the measurement of the design principles is not based on metrics, but on the adherence to the implementation of design best practices. Each design best practice has an impact on a specific object-oriented design principle and typically a set of design best practices is necessary to measure a single design principle. (2) The second contribution is a static code analysis tool that allows automatically

identifying violations of the above mentioned design best practices directly from the source code. These measured violations represent the input for the DQM. The way the measurement result determines the design assessment is qualitatively discussed in this paper based on the comparison of the two open-source projects jEdit and TuxGuitar. (3) As a third contribution we show that our DQM is capable to identifying design flaws since the data provided by our tool can be systematically used to derive sound improvements of the object-oriented design. Deriving improvements is presented for jEdit since we stay in touch with one of the main developers of this community; a contact to TuxGuitar could not be established. Going beyond the assessment of design and suggesting design improvements is a major distinguishing feature of our DQM.

Kläs et al. argue the importance of classifying quality models to support quality managers in selecting and adapting quality models that are relevant for them [6]. For a classification, the authors differ between the underlying conceptual constructs of quality models as result of their specific application and measuring purposes. The different application purposes, for example, are the specification, measurement, assessment, improvement, management and prediction of quality. From the viewpoint of application purposes, our DQM supports specifying, measuring, assessing and improving design quality. This paper concentrates on the aspects of specifying, assessing and improving object-oriented design quality.

In order to close the gap of a missing design quality model, the research method of this work follows a three step approach reflecting the structure of the article. However, before discussing the main contributions and findings, Section 2 shows related works and clarifies basic concepts used for measuring object-oriented software design. Moreover, it provides an overview of quality models in this area. Section 3 deals with the identification of design principles which was the starting point for building the DQM. In Section 4, the concepts behind the design quality model are explained and an overview of the quality model development process is given. We applied our quality model on the two open source projects jEdit and TuxGuitar in the format of a case study described in Section 5. This gives insights into using the model and especially into improving design aspects. The latter is shown by collaboration with a jEdit developer who provided feedback on the usefulness. Lastly, Section 6 discusses threats to validity and Section 7 highlights an avenue for future work.

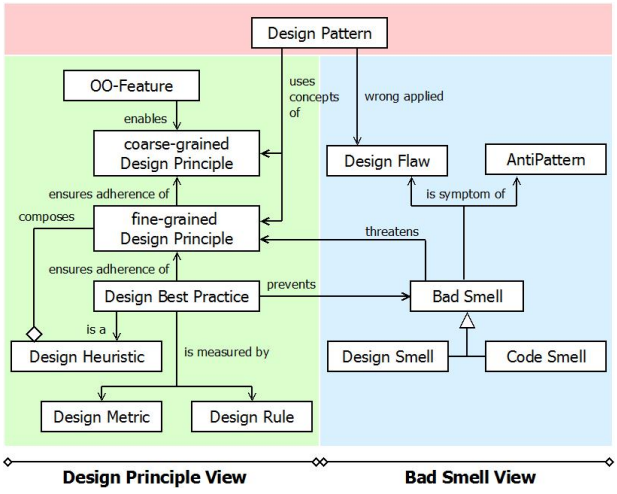

Figure 1: Ontology for Design Measurement Terms

2 Design Concepts and Related Work

Due to the importance of design principles in this work, this section first concentrates on clarifying terms for assessing object-oriented design. Then, it presents related work about design quality models.

2.1 Design Concepts for Measuring Design Quality

The research area of design assessment is multifaceted and can be considered from various viewpoints that might cause confusion when terms are not properly defined. A promising attempt to clarify the terminology is an ontology that shows the relationships between the different terms. Figure 1 depicts this ontology, which is basically derived from fundamental work in this research area we identified in a literature study and is referenced in the subsequent subsections. As highlighted in the figure, the ontology differs between the design principle view and bad smell view that are finally linked. In the following paragraphs each entity of the ontology is discussed to illustrate their direct or indirect connection to design principles.

2.1.1 Design Principle View

Initially, there is the first challenge that even the term design principle is not precisely defined and is used on different abstraction levels. According to Coad and Yourdon,

a design principle is one of the four essential traits of good object-oriented design, *i.e.*, low coupling, high cohesion, moderate complexity, and proper encapsulation [7]. As we distinguish different levels of design principles we call this type of principles *coarse-grained design principles* [5].

Coupling is a measure of the relationship between entities in contrast to cohesion that measures the extent to which all elements of an entity belong together. Thus, it is reasonable to strive for low coupling and high cohesion for sake of mainly independent software entities. While coupling and cohesion can be expressed through quantitative values [8], moderate complexity and proper encapsulation can only be captured qualitatively. The reason therefore is that there is no defined or agreed measure for expressing moderateness and properness. At this point, we do not want to clarify the measurement of complexity or encapsulation, but we want to highlight that they influence software design considerably.

Design principles such as *Single Responsibility Principle*, *Separation of Concerns Principle*, *Information Hiding*, *Open Closed Principle*, and *Don't Repeat Yourself Principle* are more specific and provide good guidance for building high-quality software design [9, 10]. Besides, they are used to organize and to arrange structural components of the (object-oriented) software design, to build up common consents about design knowledge and to support beginners in avoiding traps and pitfalls. Since these design principles are more concrete than the *coarse-grained design principles* mentioned above and to distinguish them from the others, they are considered as *fine-grained design principles* [5]. Although *fine-grained design principles* are breaking down design aspects, they are probably still too abstract to be applied in practice.

To guide a software designer and engineer with more concrete guidelines, design best practices can be used since they pertain to the use of general knowledge gained by experience. According to Riel, who refers to heuristics or rules of thumbs when talking about design best practices, these guidelines are not rules that must be strictly followed [11]. Instead, they should act as a warning sign when they are violated. Consequently, violations need to be investigated for initiating a design change if necessary. All in all, design best practices (aka heuristics) ensure the adherence of fine-grained design principles when taken into consideration and appropriately applied.

Appropriately applying a design best practice more or less depends on skills and experience of the software engineer or on a cheat sheet sticking next to the monitor. As a result, there must be means to actually check the compliance of best practices. A fundamental work, which addresses this challenge, was published by Chidamber and Kemerer who proposed a metrics suite for object-oriented design [1]. Without discussing the entire suite, measuring metrics can verify certain aspects of good-oriented design. For example, the *Depth of Inheritance Tree* (DIT) metric can provide an indicator for unpredictable behavior of methods since it becomes more difficult to predict method behavior the deeper it is located within the inheritance tree.

Although the field of object-oriented metrics is extended with various versions of metrics, there are still some design concerns that cannot be expressed by a single metric value. For instance, when a superclass calls methods of a subclass, a single value does not make sense. Instead, it is more useful to actually see where a class calls methods of its subclasses. Therefore, rules are the implementations of these design best practices that provide the functionality to show the violation in the source code.

2.1.2 Bad Smell View

While the terms from the design principles view are addressed in the discussions above, the bad smell view of the ontology is not touched so far. Starting with *AntiPattern*, it is the literary form of a commonly occurring design problem solution that has negative consequences on the software quality [12]. It uses a predefined template to describe the general form, the root causes that led to the general form, symptoms for recognizing it, the consequences of the general form, and refactoring solutions for changing the *AntiPattern* to a solution with fewer or non-negative consequences. According to Brown et al., there are three different perspectives on *AntiPatterns* that are: development, architectural, and managerial [12]. In the context of measuring and assessing object-oriented design, managerial *AntiPatterns* concerning software development and organizational issues are less important. Consequently, architectural and mostly development *AntiPatterns* are considered since they discuss technical and structural problems encountered by software engineers.

In contrast to it, a design flaw is an unstructured and non-formalized description of a design implementation that causes problems. Sometimes the term is used as synonym for *AntiPatterns* because it relates to the same type of problem (improper solution to a problem) but with a less stringent description. These improper solutions may result from applying design patterns (*e.g.*, design patterns proposed by Gamma et al. [13]) incorrectly, *i.e.*, in the wrong context, without experience in solving a particular type of problem, or from just knowing no better solution [12]. Since the term design flaw does not designate a particular

design issues, it is used when talking about design issues in a general sense.

As previously mentioned, *AntiPatterns* are recognized by symptoms that are metaphorically known as bad smells. Bad smells, or just smells, were published by Fowler et al. [3]. In this work the authors aim to provide suggestions for refactoring by investigating certain structures in the source code. In fact, they proposed a list of 22 concrete bad smells discussed in a more informal way compared to *AntiPatterns* [3]. Subsequently, researchers have added other bad smells to the original list, which follow the common understanding that a smell is not a strict rule but rather an indicator that needs to be investigated.

As shown in Figure 1, both code smell and design smell are derived from bad smell, meaning that they share the same characteristics but are more specific in a certain concern. More precisely, a code smell is a bad smell that manifests in the source code and can be observed there directly. For example, if a method contains if-statements with empty branches this simply is a correctness-related coding problem. In contrast, a design smell is inferred from the implementation of a problem solution. For instance, a subclass that does not use any methods offered by its superclass might falsely use the inheritance feature as there is no real *is-a* relationship between the subclass and its superclass. As a conclusion, a code smell can be detected on the syntactical code level whereas a design smell requires a more comprehensive semantic interpretation.

In the previous examples for explaining code and design smells, the object-oriented language features of polymorphism and inheritance are used. Actually, there are the two additional object-oriented language features encapsulation and abstraction. These four features together are the technical foundation for good design. In other words, abstraction, encapsulation, inheritance, and polymorphism are tools for software designers to satisfy software quality aspects. Without an understanding of these four techniques, bad smells are introduced and design principles as well as design best practices are violated.

2.1.3 Connecting the Principle and Smell View

At this point of explaining the ontology, there is one important connection pair missing, which shows the relationship between bad smells and fine-grained design principles as well as the opposite direction from design best practices to bad smells. To our understanding, bad smells are not symptoms of fine-grained design principles since they have a negative attitude compared to design principles, which follow a positivistic view on design. Instead, we argue that bad smells threaten fine-grained design principles – and indirectly coarse-grained design principles – when found in the source code or on a sematic level. Additionally, we think that design best practices can help to prevent bad smells as these rules of thumb address certain smell characteristics.

Next to the connection pair between bad smells and design principles, design patterns, which cannot be ignored when discussing software design, connect both sides as well. By definition, design patterns provide a general reusable solution to a commonly occurring problem [13]. Therefore, the context of the problem within the software design must be considered to avoid an inappropriate implementation. Many concepts of design patterns rest upon fine or coarse-grained design principles on the one hand. However, when they are applied in the wrong context or because they are not fully understood then the incorrect implementation can cause a design flaw on the other hand. Consequently, design patterns do not exclusively fit in one of both views since they have relations to elements of both sides why we place them right in the middle of Figure 1.

While design patterns are important when building software, we exclude them from the research area of design assessment. This decision is based on our opinion that the judgement whether design patterns are properly applied, heavily relies on the semantic context. Conformance of the application of patterns with the specification of patterns, as described in Muraki & Saeki [14], is not in our focus.

According to our understanding of these terms and their use within the domain of measuring and assessing object-oriented design, we place our DQM into the area of verifying the compliance of fine-grained design principles based on violations of design best practices (aka heuristics). In the further discourse of the model, we refer to fine-grained design principles when talking about design principles or just principles.

2.2 Related Work about Quality Models for Object-Oriented Design

Besides design quality as focused in this work, it is important to understand the conceptual approach of expressing software quality by using quality models. For several decades, these kinds of models have been a research topic resulting in a large number thereof [6]. First insights in this field were shown by Boehm et al. who described quality characteristics and their decomposition in the late 1970s [15]. Based on that understanding of modeling quality, the

need for custom quality models and the integration of tool support arose. As a result thereof, the quality models simply decomposed the concept of quality in more tangible quality attributes. The decomposition of quality attributes further enhanced by introducing the distinction between product components as quality carrying properties and externally visible quality attributes [16].

Due to pressure from practitioner and based on the quality models from the late 80s, the ISO 9126 standard was defined in 1991. While this was the first attempt to standardize software quality, concerns regarding the ambiguous decomposition principles for quality attributes were discussed [17]. Moreover, the resulting quality attributes are too abstract to be directly measurable [17]. Because of these problems, the ISO 25010 as successor of ISO 9126 has been published but addresses just minor issues why the overall critique is still valid. Regardless of the ISO standards, the resulting quality models from the late 90s did not specify how the quality attributes should be measured and how these measuring results contribute to a general quality assessment of the investigating software.

Comprehensive approaches that address these weaknesses are, *e.g.*, Squale [18] and Quamoco [19]. The research team of Squale first developed an explicit quality model describing a hierarchical decomposition of the ISO 9126 quality attributes and extended the model with formulas to aggregate and normalize measuring results. For operationalizing this model, a tool set including the measures is accompanied. The Quamoco approach – more extensively discussed in Section 4 – addresses the above mentioned weaknesses as well, but in contrast to Squale it structures quality attributes by using a product model. Besides, Quamoco provides a modeling environment that allows to integrate measuring tools. Given this quality models, approaches for measuring, assessing and improving software quality has established. Nevertheless, the more focused research area of software design quality has still open challenges.

With a focus on design quality, Bansiya and Davis made one of the first attempts to establish a quality model therefore [20]. This quality model for object oriented design (QMOOD) is structured on four levels. On the first level, the model consists of six design quality attributes that are derived from the ISO 9126 standard [21]. As mentioned by Bansiya and Davis, these attributes are not directly observable and there is no approach for their operationalization. Consequently, they introduced the layer of object-oriented design properties. *Encapsulation* and *Coupling* are examples of these design properties in QMOOD. Although design properties are a more specific way of expressing quality, there is still the problem that neither quality attributes nor properties are measurable. Thus, the third level of QMOOD specifies one (!) design metric for each design property. Lastly, the fourth layer of QMOOD represents the object-oriented design components that are objects, classes, and the relationships between these elements.

The idea of decomposing a quality or design aspect into more fine-grained properties is applied in our DQM as well. Furthermore, DQM distinguishes more clearly between the quality-carrying properties used for measurement (following the work of Dromey [16] and Wagner et al. [22]) and the impact of these properties on object oriented design principles. Although QMOOD tries to structure the assessment systematically, it still lacks a consideration of design properties from a broader view. For example, the measurement of the design property *Encapsulation* with one metric is neither sufficient to assess the compliance of this design aspect nor is it helpful for guiding improvements. Contrary, our DQM defines eight best practices for the design property *Encapsulation* that are measured using static analysis techniques.

Another attempt to address the assessment of object-oriented design is proposed by Marinescu and Ratiu [2]. According to these authors, real design issues often cannot be directly identified when single metrics are considered in isolation. Thus, they propose an approach to detect design flaws, referred to as bad smells [3], by using so called detection strategies. A detection strategy relies on measuring different aspects of object-oriented design by means of metrics and combining them to one metric-based rule. This combination of metrics allows reaching a higher abstraction level in working with metrics and expressing design flaws in a quantifiable manner.

The approach from Marinescu and Ratiu [2] can be used to indicate the problem source that causes a design flaw. Our DQM is designed with the same intent, as one of its measurement purposes is the improvement of the current design. Compared to QMOOD, the approach of Marinescu and Ratiu leads to better founded assessment results as different metrics are combined. For guiding improvements, there remains the obstacle of using metrics that are difficult to derive hints for design enhancements. While our DQM combines different design best practices to one overall result, the concentration on design best practices better guides improvement processes. Contrary to our approach, Marinescu and Ratiu do not provide a comprehensive quality model but focus on measuring a set of design and code smells.

A recently published work picks up the idea of checking the compliance of design principles [5]. The approach, which is known as MIDAS, is an expert based design as-

sessment method following a three view model: the design principles view, a project-specific constraints view, and an "ility"-based quality model view. The MIDAS approach emphasizes project-specific objectives and relies on manual measures assessed by experts.

MIDAS provides a design quality model with different views on design but does not provide support for automatic assessment or support for guiding improvements. This is a major difference to our work. In addition, the authors of MIDAS point out that a reference quality model for design would be desirable as general purpose quality models like ISO 9126 [21] are not specific enough to capture design quality. With this paper and the first presentation of the DQM, we are filling the gap of the missing reference model that is based on design principles measured by the compliance of associated design best practices.

3 Identification of Design Principles

This paper is based on *fine-grained design principles* since we are eager to use our design quality model for measuring, assessing and improving the compliance of a software product with these design principles. Unfortunately, there is no systematic work that collects these principles that are discussed in the literature and applied in practice. Thus, we first identified potential candidates in the literature, which were then used in a survey to get at least an understanding about their importance.

This survey was available from March 16th to April 20th 2015 and during this time 104 participants completed the questionnaire. Participants have been acquired by emails to senior developers or architects known to us in development organizations worldwide (with a focus on Europe). Moreover, partners were encouraged to re-distribute our call for achieving a higher distribution and we placed calls in ResearchGate and in design-oriented discussion groups on LinkedIn.

Analyzing the demographic aspects of the participants shows that more than 50% of the participants are employed in companies with more than 1.000 employees. Table 1 depicts the actual distribution.

The question regarding the current job role indicates that many software architects and developers participated. The distribution looks as follows: 12 project managers, 4 quality managers, 37 software architects, 62 software developers, 7 software testers, 2 software support engineer, 22 consultants, and 12 scientists completed the questionnaire (multiple job roles were allowed). Finally, the analysis of the engineering domains highlights a suitable distribution, with the exception of mobile systems with only one participant. All in all, 23 participants are working on web/service oriented systems, 14 on embedded systems, 17 on development tools, 25 on business information systems, 1 on mobile systems, 8 on expert and knowledge-based systems, and 16 on a system from another (unspecified) domain.

Table 1: Distribution of Participants

# of Participants	Company Size or Organization
2	< 10 employees
13	< 50 employees
12	< 250 employees
12	< 1.000 employees
54	> 1.000 employees
11	Academic organization

Table 2: Ranking of Design Principles

Design Principle	Weighted Rank
Single Responsibility Principle (SRP)	695
Separation of Concern Principle (SOC)	647
Information Hiding Principle (IHI)	611
Don't Repeat Yourself Principle (DRY)	535
Open Closed Principle (OCP)	459
Acyclic Dependency Principle (ADP)	384
Interface Segregation Principle (ISP)	378
Liskov Substitution Principle	365
Self-Documentation Principle (SDOP)	332
Favor Composition over Inheritance (FCOI)	326
Interface Separability (ISE)	326
Stable Dependencies Principle	326
Law of Demeter	326
Command Query Separation (CCS)	326
Common Closure Principle (CCP)	326

In fact, the questionnaire had just one major question – in addition to the demographical ones – which asked the participant to rank 10 out of 15 pre-selected design principles according to their importance. With 104 rankings of 15 design principles, we were than able to identify important ones by weighting their rank from 10 to 1. In other words, when a design principle was ranked in first place, ten points were added to its weighted rank; nine points were added for rank two, eight points for rank three, and so forth. Table 2 shows the final rank of all opinions reflecting the practical importance of *fine-grained design princi-*

ples. The five most important principles are explained next since they are used in the case study.

Single Responsibility Principle (SRP)

To follow the single responsibility principle, "a class should have only one reason to change [23]". The same definition is used for specifying the term responsibility since responsibility is a reason for a change [23]. In other words, when a class could be changed depending on two or more reasons, it is likely that it contains two or more responsibilities.

Separation of Concern Principle (SOC)

One of the first statements regarding the separation of concern principle was made by Edsger W. Dijkstra in his essay titled "On the role of scientific thought [24]". Although the thoughts of Dijkstra are not primarily targeted to the software engineering discipline but rather on the characteristics of intelligent thinking in general, he states that studying one aspect of a subject matter in isolation supports the consistency of the person. This cannot be achieved when various aspects are considered simultaneously. Thus, Dijkstra first coined the term "the separation of concerns" for supporting the idea of effectively ordering someone's thoughts to focus the attention on just one aspect. Mapped to the software engineering discipline, SOC stands for separating a software into distinct sections such as each section addresses a particular concern [25].

Information Hiding Principle (IHI)

Not only did Parnas introduce the concept of modularization, he also discussed the idea of the information hiding principle [26]. He argued that each model should be designed in a way that hides critical design decisions from other modules. This ensures that clients do not require intimate knowledge of the design to use the module what keeps clients less fragile against design changes of their dependent modules.

Don't Repeat Yourself Principle (DRY)

In a software development process, code duplicates can be introduced due to various situations. For example, time pressure of a release deadline can force an engineer to copy code instead of changing the design as well as mistakes in the design can also lead to duplicated segments. Having the same functionality spread across a software system, makes it difficult to maintain the software; especially, when requirements change frequently. Consequently, the don't repeat yourself principle defines that each piece of knowledge must have a single, unambiguous, and authoritative representation within a software system [27], *e.g.*, no duplicated data structures or code, no meaningless source code documentation, or no flaws in the source code documentation.

Open Closed Principle (OCP)

According to Martin [23], the open closed principle is at the heart of object-oriented design and says that "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification". In other words, the behavior of a type should be changed by extending it instead of modifying old code that already works. Symptoms indicating violations of this principle can be recognized when further changes of a type cause a cascade of changes or additional types are needed to ensure an operative solution. Thus, the design embodies a lack of awareness for changing concerns and inflexibility to adapt new requirements.

About the definition of openness and closeness, former relates to the ability to change the behavior of a type by extending it according to new requirements [23]. Closeness means that the extension of behavior does not change the source code of a type or binary code of a module [23]. Now someone could claim that it is not possible to change the behavior of a type without touching its source code. This is true when the object-oriented feature of abstraction is not or inappropriately applied. In fact, abstraction is the key to ensure the compliance of both the openness and closeness of a type or module.

Additional Design Principles

In addition to the task of ranking the principles, the survey asked about missing design principles by means of an open question. The following three design principles were selected as important ones that were missing:

- *Dependency Inversion Principle (DIP):* 13 answers referred to the DIP as one of the five SOLID principles. Many participants emphasize the power of DIP for breaking cyclic dependencies and fostering a loose coupling of software modules. Consequently,

it supports software development in reusing software components and optimizing modularity.
- *Keep It Simple and Stupid Principle (KISS):* In eight surveys the KISS principle was mentioned as missing. KISS concentrates on reducing complexity and building software that is as simple as possible, but still meets the requirements of stakeholders. By reducing complex constructs, it is possible to create a common code ownership that supports the development of comprehensive solutions.
- *You Ain't Gonna Need It Principle (YAGNI):* The third principle is YAGNI that is referenced in six answers. YAGNI has a similar goal to KISS in that it focuses on building solutions, which are not overloaded by unnecessary functionality.

4 The Design Quality Model (DQM)

This section presents our DQM. Therefore, the first part concentrates on the underlying concept and the meta model of the quality model followed by the evaluation functions that are required to express the compliance of design principles. Finally, the applied process for defining the rules and the current content of the model is summarized.

4.1 Aspect and Factor Hierarchy

DQM structures design quality using the Quamoco approach proposed by Wagner et al. [22]. Authors of this paper were part of the Quamoco development team and already developed Quamoco-based quality models for embedded systems [28], safety critical systems [29] and for software documentation quality [30]. Quamoco-based quality models rely on the basic idea of using product factors for expressing a quality property of an entity. In context of this work, an entity is a source code element like package, class, interface or method, while a (quality) property is a quality characteristic of an entity; encapsulation is an example of a property for the entity class. Design principles are a special view on the product factors called design quality aspects. The relation between product factors and design quality aspects is modelled with impacts from product factors to design quality aspects, *e.g.*, the product factor *Encapsulation @Class* has a negative impact on the design principle (modelled as a design aspect) information hiding.

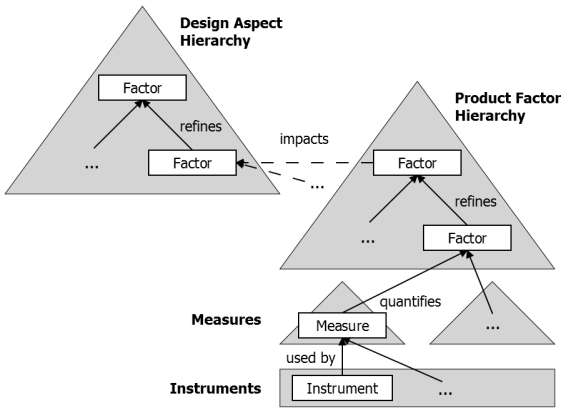

Figure 2: Quality Aspect and Product Factory Hierarchy of DQM

As shown in Figure 2, the design aspect and product factors can be refined into sub-aspects and sub-factors, respectively. Design aspects express abstract design goals on the top level, but they are broken down into design principles on the next lower level. For example, the general design aspect of abstraction comprises the three design principles: *Command Query Separation*, *Interface Separability*, and *Single Responsibility*.

Product factors are attributes of parts of the product (design) refined into fine-grained factors, *e.g.*, on the level of methods, interfaces, or source code. Compared with the design aspect hierarchy, the product factory hierarchy is broader and deeper, resulting in a larger factor tree. In addition, the leaves of the product factor tree have a special role because they can be measured. For instance, a leaf node is the design best practice *AvoidPublicInstanceVariables* that is part of the product factor *Deficient Encapsulation @Class*. In the further course of this paper, these leaf nodes, *i.e.*, design best practices, are referred to as rules that can be automatically derived from source code using static analysis.

The separation of design aspects and product factors supports bridging the gap between abstract notions of design and concrete implementations. For linking both abstractions, an impact can be defined. In fact, impacts are a key element in this model since they define which product factor affects which design aspect. Such an impact can have a positive or negative effect and the degree of the impact can be specified. Not every product factor has an impact on a design aspect because some of them are used for structuring purposes only.

Since the design quality assessment is relying on measureable product factors, the model assigns an instrument to each rule. We have developed such an measuring instrument that is called MUSE [31]. MUSE contains implemen-

tations of 67 rules like the one used above – *AvoidPublicInstanceVariables* – and it can identify violations thereof in source code written in the programming languages Java, C# and C++.

4.2 Meta-Model

The underlying meta-model of the DQM is derived from the Quamoco approach [22]. Figure 3 provides an overview of the meta-model elements in a simplified UML class notation. The central element is the factor as the abstract form of a design aspect or product factor. Both design aspect and product factor can be refined into sub-aspects or sub-factors, respectively. Nevertheless, just product factors consist of rules that are linked to a measuring instrument (MUSE). For completing the right side of the meta-model, an impact is modeled as *many-to-many* relationship from product factor to design aspect.

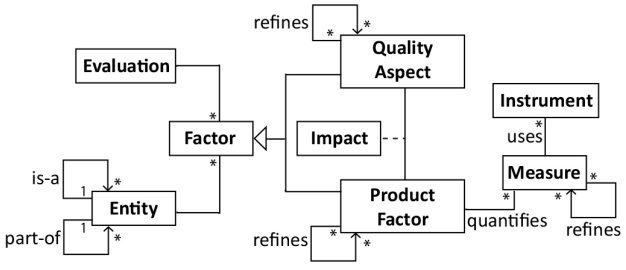

Figure 3: Meta-model of DQM in simplified UML notation

The left side of Figure 3 shows that a factor has an associated entity. This entity can be in an *is-a* or a *part-of* relationship within the hierarchy. For instance, the entity method is *part-of* the entity class and the entity method *is-a* source code. For expressing this *is-a* relationship, the name of the entity becomes important because it depicts the relationship by adding, *e.g.*, *@Class*, *@Method*, or *@Source Code* to the entity name. Next to the entity, an evaluation is assigned to a factor. This is used to evaluate and assess the factor composed of evaluation results from sub-factors or actual rules; the latter just works in case of product factors.

4.3 Content of the DQM

The DQM is a comprehensive selection of design aspects, product factors, and rules relevant for the design quality assessment. DQM was built from ground up by us and comprises 19 design aspects and 105 product factors in total. Since some factors are used for structuring purposes rather than design assessment, 14 design aspects (*i.e.*, design principles) and 66 product factors with 67 rules implemented in our tool MUSE build the operational core of the DQM. In this paper we concentrate on the five most important design principles – according to the survey in Section 3 – that are operationalized by 18 product factors and 28 rules.

In more detail, the design aspect hierarchy is composed of 19 aspects with five aspects used for structuring purposes. The remaining 14 aspects capture the design principles, as shown in Table 2 but without *Liskov Substitution Principle*, *Stable Dependencies Principle*, and *Law of Demeter*. Additionally, the design principle hierarchy contains the principle *Program to an Interface, not an Implementation*, which is neither listed in Table 2 nor mentioned as an additional one.

The product factor hierarchy contains 105 entities and 67 rules on the leaves. Due to some modeling constraints, a number of factors are used for structuring the model without containing rules. Thirty-five product factors contain rules, *i.e.*, design best practices provided by MUSE.

To illustrate the measuring of a product factor and how it influences a design aspect, the following example shows a product factor including its rules on the leaves and impacts to design principles. The product factor *Duplicate Abstraction @Type* addresses the problem that there may exist two or more types that are similar within a software design. Therefore, these types share commonalities that have not yet been properly captured in the design. The following characteristics can indicate such an issue:

- *Identical name:* The names of the types are the same.
- *Identical public interface:* The types have methods with the same signature in their public interface.
- *Identical implementation:* Logically the classes have similar implementation.

For measuring this product factor, the DQM has the following three design best practices assigned to it, which address three characteristics of the design issue:

- *AvoidSimilarAbstraction:* Entities of the same type should not represent similar structure or behavior.
- *AvoidSimilarNamesOnDifferentAbstractionLevels:* Entities of entity types on different abstraction levels (*e.g.*, namespace, class) should not have similar names.
- *AvoidSimilarNamesOnSameAbstractionLevel:* Entities of entity types on the same abstraction level should not have similar names.

These rules are provided by our tool MUSE that identifies violations thereof [31]. A high number of violations in a project is an indicator that the software contains abstraction related design flaws. Moreover, it is interesting to understand which design principles are threatened by these violations. This can be easily examined by following the impacts of the product factors on design aspects. In this particular case, the product factor *Duplicate Abstraction @Type* has a negative impact on the *Don't Repeat Yourself Principle* and on the *Separation of Concern Principle*.

In order to support the understanding of the entire quality model, we refer to Section 5 and the appendix of this article. Both sections show the 14 design principles and the impacts of their assigned product factors in tabular format. Additionally, the tables contain measuring results obtained from the evaluation functions behind the model. The way of reading the measurement is explained next.

4.4 Design Quality Evaluations

The underlying Quamoco meta-model provides support for specifying evaluation functions to calculate a quality index for a object-oriented software system. In order to better understand the evaluation capabilities of the DQM, we describe this step by step.

Product factors have one or multiple rules assigned. Consequently, the evaluation function for each product factor defines a specification for each assigned rule. This is necessary as each rule has a different value with a different semantic. In this first evaluation step, each measured value is normalized with a size entity and transposed into the range [0..1] in order to be able to aggregate the results of distinct rules later. Below is an example of a typical evaluation specification:

```
MUSE;ACIQ;#METHODS,0.0;0.5;1
```

The first part of the specification defines the tool that provides the rule, *i.e.*, MUSE. This is followed by an abbreviation of the rule name and a size entity of the project. In this example ACIQ stands for *AvoidCommandsInQueryMethods*. The size entity is used to normalize the number of findings with the number of methods. Obviously, it makes a difference whether five methods out of 500 (for a small project) or five methods out of 5,000 (for a medium-sized project) do not adhere to this design best practice. The next two elements of the specification define the slope of a linear function that returns a value between 0 and 1.

To facilitate the understanding of design assessments discussed in the next section, the outcome of the linear function must be explained. As mentioned above, the function calculates a value ranging from 0 and 1. This value depends on the number of normalized findings whereas 0 is returned when there are no findings and 1 is returned when the normalized number of findings exceeds a defined threshold. Consequently, the following reminder holds: the lower the value, the better the assessment. In the example above, a source code where 50% or more of the methods violate the best practice is considered to be very bad. For the evaluation of a product factor, the results of multiple rules – design best practices – have to be combined. The last element in our above evaluation specification defines the weight of the rule.

To distinguish between the evaluation results on rule level and on the product factor level, the values on the product factor level are expressed in the range of 0 to 100. To calculate the values on product factor level, the results of their weighted rules are summed up. Thus, they can be interpreted like rule assessments.

Next to evaluation functions on the level of product factors, design aspects (in our case the design principles) also have evaluations assigned. These evaluations depend on the impacts of product factors to design aspects and are specified as follows:

```
Deficient Encapsulation @Class;1
```

These specifications just define the product factor and its weight expressed by the last number. To aggregate the product factors to a single value for the design aspect assessment, we suggest forming relevance rankings based on available data or expert opinion. We use the *Rank-Order Centroid* method [32] to calculate the weights automatically from the relevance ranking according to the *Swing* approach [33]. This function returns a value between 0 and 10 that needs to be read differently compared with the previous evaluation results. Hence, this value can be interpreted as points gained for a good design so that a good design – with few findings – earns almost 10 points. Consequently, the reminder on this level must be flipped: the higher the points, the better the assessment.

4.5 Development of the DQM

The entire development of the DQM was carried out by four researchers (two from academia and two from Siemens corporate research). In the first step, we tried to find and to specify design best practices for each design principle in-

dependent of any product factors of a quality model. This was driven by the definitions and discussions of the design principles in the extant literature. After specifying the design best practices, each researcher independently voted on the importance of each best practice. The only design best practices that were then included in the DQM were the ones where a majority of the researchers identified a contribution to the design principle.

In a second step, we jointly built the product factor hierarchy, *i.e.*, assigned the design best practices to the product factors. This was a joint work effort by the four researchers, who were guided by some principles of how to structure the product factor hierarchy. Thus, it is not possible to define an impact from a leaf in the product factor hierarchy to a quality attribute [19, 22]. Investigating these design properties led to the identification of additional design best practices. At the end of the process we identified 85 design best practices used to measure (partially) 13 design principles.

During building this model, we followed a practice oriented approach and iteratively discussed our status of the DQM with practitioners from industry. Parallel to the model development, MUSE – the measuring instrument – has been implemented. Industry partners used MUSE in their projects and they provided feedback we incorporated in our model. With feedback from partners about the DQM and MUSE, we could also reflect on the completeness of measuring each design principle by (1) assigning a percentage of coverage by the underlying design best practices and by (2) explicitly specifying what is missing. A more formal validation of the completeness of our DQM is still pending and out of scope of this article.

Specifying the design quality evaluations was challenging and was carried out by the four researchers. For the evaluation specification of each rule (*i.e.*, design best practice) we had to define (1) the normalization value, (2) the importance of that rule, and (3) the an appropriate threshold. The normalization value (*e.g.*, number of classes, or lines of code) could be systematically derived from the specification of the rule. For the definition of the rule importance, we relied on knowledge gained from the construction phase of the DQM core and reused this knowledge to provide proper weights.

Defining the thresholds for each rule was a more complex task and was based on previous work. More specifically, we used a benchmark suite consisting of 26 Java projects that has proven to represent a comparable base; especially, for Java [34]. The entire list of projects is shown in the appendix with additional details about the release version and the number of logical lines of code. Using the benchmark suite, we derived thresholds that define the upper boundary of evaluation functions, *i.e.*, when the number of rule violations exceed this threshold the evaluation function returns the worst assessment for this particular design best practice.

Lastly, we had to specify the weights for all product factors that contribute to the evaluation of a design principle, *i.e.*, the weights of impacts as shown in Figure 2. This was carried out as a joint effort with lots of discussions; interestingly, we hardly had any different opinions on the weights.

5 Case Study

This presented case study concentrates on a qualitative discussion of evaluation results and on suggestions of design improvements after applying our DQM on jEdit and TuxGuitar. More specifically, we use our measuring tool MUSE to identify rule violations that represent the raw data of the DQM. With the number of rule violations, the assessments of design properties are calculated and the impact on design principles is derived. Finally, we select the top five design principles – according to the survey in Section 3 – to discuss different characteristics of the evaluation, to compare the design assessment, and to show improvement examples. The latter were discussed with a developer of jEdit who can justify our suggestions.

5.1 Systems of Study

This case study uses the source code of two open-source projects as shown in Table 3. They have been selected since both are desktop applications with a similar application domain. Furthermore, the size of the projects expressed by their logical lines of code is within a comparable range, which also fits to the benchmark base of the DQM and supports the understanding of the normalization step mentioned later on.

It is important to understand that there was no information on the design of the software products before applying the DQM. We just hoped that we could identify significant differences in the assessments where the differences can be discussed and justified in a systematic way. The emphasis of this case study is on the discussion of measured differences in the object-oriented design quality and on judging whether the differences in the assessments are justified. We cannot compare our results with other validated external design quality evaluations, which of course would be even more interesting.

Table 3: Projects of Study

Project	Version	LLOC	# of Classes	Application Domain
jEdit	5.3	112,474	1,277	Text Editor
TuxGuitar	1.3.0	81,953	1,816	Graphic Editor

5.2 Discussion of SRP Assessment

To read a measurement as shown in Table 4, the white rows, which depict the rule name, number of rule violations, and evaluation on rule level, need to be investigated first. The aggregation to the next higher level is shown in the light gray rows with the property name and the aggregated value computed by all assigned rules. Finally, the second top row of the table shows the assessment of the design principle impacted by the design properties.

According to this approach of reading a measuring result, the SRP assessment for both projects consists of two rules. The first rule *AvoidPartiallyUsedMethodInterfaces* determines the product factor *LargeAbstraction @Class* whereas the second rule *AvoidNonCohesiveImplementation* determines the second product factor *Non-Cohesive Structure @Class*. Lastly, the two product factors are responsible for assessing SRP of jEdit and TuxGuitar with 8.13 and 7.11, respectively. As a result, jEdit scores better for this design principle what is a valid statement as they are compared against the same benchmark base used for building the DQM.

The first characteristic of the evaluation discussed in this section is the normalization applied in the evaluation functions. It is not shown in Table 4, but the evaluation functions for both rules use the number of classes to normalize the absolute number of rule violations. Thus, the number of violations is normalized with 1,277 classes for jEdit on the one hand and with 1,816 classes for TuxGuitar on the other hand.

Without considering normalization, one would assume that a similar number of findings leads to similar evaluations. In the case of *AvoidNonCohesiveImplementation*, doubling the number of findings for jEdit leads to a similar number of findings for TuxGuitar (356 for jEdit vs. 334 for TuxGuitar). Consequently, one would expect that jEdit gets an evaluation of 0.55 which is far away from the value for TuxGuitar (which is 0.36). The reason therefore is the normalization used for *AvoidNonCohesiveImplementation*. All in all, the evaluations can use the entity sizes of packages, classes, methods, members, static fields, and logical lines of code to achieve comparability of measuring results across different projects.

Table 4: Measuring Result of SPR for jEdit and TuxGuitar

	jEdit		TuxGuitar	
Single Responsibility Principle		8.13		7.11
LargeAbstraction @Class		5.38		12.55
AvoidPartiallyUsedMethodInterface	18	0.05	57	0.12
Non-Cohesive Structure @Class		27.87		36.78
AvoidNonCohesiveImplementation	178	0.27	334	0.36

Suggestions for Improvement

For showing an improvement regarding SRP, a violation of *AvoidNonCohesiveImplemenation* is considered. This rule violations refers to the class *Mode*, which consists of two independent parts that could be separated into two abstractions. In fact, the main part of the class deals with the intended behavior of *Mode* compared to the part that could be factored out that obviously adds an additional responsibility to the class. Namely, it is used to differ between a "normal" *Mode* object and a "user" *Mode* object and consists of the methods *isUserMode* and *setUserMode* as well as the property *isUserMode*.

Instead of managing this responsibility within the *Mode* class, it makes sense to define an additional abstraction *UserMode* that is derived from *Mode* and deals with the user mode specific requirements. A client – *ModeProvider* – that is actually dealing with *Mode* objects and needs the distinction between normal and user *Mode* objects could benefit from this improvement by requesting the data type of any mode object at runtime. Moreover, the design with an additional abstraction instead of an overloaded *Mode* class is more robust against changes in regard to user mode objects.

This suggestion has been presented to the developer of jEdit. After a thorough discussion, we draw the conclusion that this suggestion is an example for good object-oriented design. However, the developer did not submit a change request on this issue because the behavior of the *Mode* object will not change in future and no additional sub-type will be expected.

5.3 Discussion of IHI Assessment

Compared to the assessment of SRP, where (currently) two rules are used to derive the compliance of the design principle, the assessment of IHI is more multifarious including eight rules. Although more rules are part of this assessment, which also shows an irregular distribution of findings, both projects arrive at a similar result with approximately 5.9 points. The reason therefore is that rules as well as properties are weighted differently as another characteristic of the design assessment.

A good example for demonstrating the different weighting is the property aggregation with the values in Table 5. In this particular case, TuxGuitar is better than jEdit at *Deficient Encapsulation @Class* with 34.77 compared to 47.22 points. Although TuxGuitar is performing worse for the product factor *Weak Encapsulation @Class* (59.47 compared to 21.92 points), this does not have a large effect as the contributing weight of this product factor in the overall calculation of the information hiding principle is just half of the weight of *Deficient Encapsulation @Class*. Consequently, the weighted sum is almost equal and responsible for the 5.9 points. In our understanding the different weights are justified, as *Deficient Encapsulation @Class* depicts massive violations of the principle while *Weak Encapsulation @Class* can be more easily accepted.

Suggestions for Improvement

One of the design flaws that violates the compliance of IHI is identified by the rule *UseInterfaceAsReturnType*. As shown in Listing 1, the class *JEditPropertyManager* implements the interface *IPropertyManager*. However, the method *getPropertyManger* in *jEdit* returns the concrete data type instead of the interface that would function at this point and would make *jEdit* more robust against changes. The reason therefore is that normally there are less changes of interface since they are well defined.

Even though the class *jEdit* plays a crucial role in the software, the developer will verify whether changing the return type from a concrete class to an interface is possible. He argues that it is likely to work since the class *JEditPropertyManager* just implements the interface and does not contain additional behavior.

Table 5: Measuring Result of IHI for jEdit and TuxGuitar

	jEdit		TuxGuitar	
Information Hiding Principle		5.91		5.90
Deficient Encapsulation @Class		47.22		34.77
AvoidProtectedInstanceVariables	115	0.36	189	0.70
AvoidPublicInstanceVariables	185	1.00	1	0.01
AvoidSettersForHeavilyUsedAtt.[1]	16	0.05	6	0.02
CheckParametersOfSetters	132	0.34	462	1.00
DontReturnCollectionsOrArrays	59	0.15	71	0.15
UseInterfaceAsReturnType	315	0.82	228	0.50
Weak Encapsulation @Class		21.92		59.47
AvoidExcessiveUserOfGetters	17	0.26	84	0.92
AvoidExcessiveUserOfSetters	13	0.20	44	0.48

[1] AvoidSettersForHeavilyUsedAttributes

Listing 1: Violation of UseInterfaceAsReturnType

```
public interface IPropertyManager{
  String getProperty(String name);
}

public class JEditPropertyManager implements
    IPropertyManager {
  ...
  @Override
  public String getProperty(String name){
    return JEdit.getProperty(name);
  }
}

jEdit.java - Line 2396
public JEditPropertyManager getPropertyManager()
    { ... }
```

Another enhancement is recommended based on the rule violation of *DontReturnCollectionsOrArrays*. Listing 2 shows this design flaw on the class *ColumnBook* that returns a vector, which is then modified by the client *ElasticTabStopBufferListener*. In fact, the client removes all elements from the collection and changes the *ColumnBlock* object without any notification. To control the external modification of the internal class property, it is recommended to avoid returning the collection children but rather provide a method, *e.g. removeAllChildren()*, that implements the functionality of emptying the collection within the class *ColumnBook*.

Listing 2: Violation of DontReturnCollectionsOrArrays

```
public class ColumnBlock implements Node {
  ...
  public Vector<Node> getChildren(){
    return this.children;
  }
}

ElasticTabStopBufferListener.java - Line 145ff

ColumnBlock innerParent = (ColumnBlock)
    innerContainingBlock.getParent();
innerParent.getChildren().removeAllElements();
```

The developer responded to this suggestion by accepting the violation of the particular design best practice. He argues that exposing the collection to the client (*ElasticTabStopBufferListener*) is an intended extension point in this particular case. However, he mentions that the team is normally concerned about returning internal data.

5.4 Discussion of DRY Assessment

In contrast to the assessment of IHI, the assessment of DRY does not weight properties differently. Not even rules on the bottom level have a weight assigned so that the number of findings determines the compliance of DRY for jEdit and TuxGuitar.

When calculating the evaluations, jEdit achieves more points compared to TuxGuitar. This results from many rule violations at the TuxGuitar project as discussed with the following examples and shown in Table 6. First, 995 public classes are undocumented in the TuxGuitar project whereas jEdit has just 37 undocumented classes. This high number of rule violations for TuxGuitar causes the worst evaluation with 100 points at the property level. In contrast, jEdit has a much lower value with 11.59 points there. The same applies for the rule that check undocumented interfaces and code duplicates. This definitely reflects our understanding of documentation quality.

The properties *Documentation Disintegrity @Methods* and *Duplicate Abstraction @Type* are composed of multiple rules, which is why their property evaluation is an aggregated value. When investigating the *Documentation Disintegrity @Methods* property, it can be identified that the 50.83 points are the result of a low number of findings (38) for the first rule on the one hand and a very high number of findings (5,876) on the other hand. The same applies for the property *Duplicate Abstraction @Type*, where two rules report a very high number of violations so that TuxGuitar gets a worse property evaluation with 66.66 points compared with 44.89 points for jEdit. This design principle is measured without specific weightings (on either aggregation level) and the normalized values more or less reflect the perceived difference in quality.

Table 6: Measuring Result of DRY for jEdit and TuxGuitar

	jEdit		TuxGuitar	
Don't Repeat Yourself Principle		7.20		2.07
Documentation Disintegrity @Class		11.59		100.0
DocumentYourPublicClasses	37	0.11	995	1.00
Documentation Disintegrity @Interf.		31.11		100.0
AvoidUndocumentedInterfaces	35	0.33	248	1.00
Duplication @SourceCode		37.97		99.70
AvoidDuplicates	299	0.38	572	0.99
Documentation Disintegrity @Method		17.50		50.83
AvoidMassiveCommentsInCode	82	0.04	38	0.01
DocumentYourPublicMethods	583	0.31	5k	1.00
Duplicate Abstraction @Type		44.89		66.66
AvoidSimilarAbstrations	32	0.50	240	1.00
AvoidSimilarNamesOnDiff.Ab.L.[1]	6	0.09	0	0.00
AvoidSimilarNamesOnSameAb.L.[2]	48	0.75	666	1.00

[1] AvoidSimilarNamesOnDifferentAbstractionLevels

[2] AvoidSimilarNamesOnSameAbstractionLevel

Suggestions for Improvement

Obviously, the most important rule for checking DRY is *AvoidDuplicates* that actually found design flaws next to many code quality issues. Latter are mostly simple copy paste code snippets that could be factored out into a single method. However, duplicates along an inheritance hierarchy and duplicates in class siblings are those issues that address design concerns. Actually, we could find examples thereof.

For instance, both *ToolBarOptionPane* and *StatusBarOptionPane* – that are siblings due to their base class *AbstractOptionPane* – contain the identical method *updateButtons*. Consequently, the design improvement has to concentrate on moving the implementation of *updateButtons* to the base class for eliminating the duplicates in the siblings.

Next to that, the rule found a design flaw where a child class implements functionality that is already available in the base class. In more detail, the class *JEditTextArea* that is derived from *TextArea* should call *handlePopupTriger* of its base class instead of duplicating the implementation.

These two design flaws caught the interest of the jEdit developer because he constructively discussed the consequences resulting from code duplicates. Next to the code quality issues resulting from copy paste snippets, he defined a change request for the second suggestion. The reason therefore is that there is no need for the redundant method and the method in the base class can be called. For the sake of completeness, the first suggestion for improvement will not be fixed as he is concerned about possible side effects.

5.5 Discussion of OCP Assessment

The previous discussion focused on understanding the impact of findings on the final assessment of the design principle. However, a closer look at the evaluation approach can raise two concerns. First, when one rule is part of a set of rules for a property evaluation, the rule can be underestimated. For instance, the rule *CheckParametersOfSetters* in Table 7 counts three times more findings for TuxGuitar than for jEdit but determines the property evaluation with just a sixth. In order to deal with this issue of under- or overweighting rules, an impact factor is assigned to a rule that determines the influence on the property evaluation. Consequently, the evaluations of the DQM are customizable depending on the requirements of a quality manager or the application domain of the project.

The second concern that can positively or negatively impact an assessment is the use of thresholds in the evaluation function. To discuss this aspect, the rule *UseAbstraction* at the *Incomplete Abstraction @Package* property in Table 7 is selected. Based on the underlying evaluation for this property, 19 findings cause a 95 point assessment of the property for jEdit and 160 findings cause a 100 point assessment for TuxGuitar. Thus, there are just five points between both assessments while TuxGuitar has many more findings. In other words, a project cannot get worse when it reaches the defined threshold. This results in a better assessment as the real state reflects.

While the effect of improper thresholds is minor when comparing multiple versions of the same project, a comparison of different projects can cause an inaccurate perception. For dealing with this issue and for deriving appropriate threshold values, we used a benchmark suite based on multiple and similar projects as shown in Section 4.5 and the appendix. Since jEdit and TuxGuitar fit within the benchmark suite due to similar characteristics, the comparison with this set is valid [34, 35]. Moreover, it is legitimate to draw the claim that one project is better than the other because both are compared against the same benchmark base.

Table 7: Measuring Result of OCP for jEdit and TuxGuitar

	jEdit		TuxGuitar	
Open Closed Principle		8.13		7.11
Deficient Encapsulation @Class		47.32		34.77
AvoidProtectedInstanceVariables	115	0.36	189	0.70
AvoidPublicInstanceVariables	185	1.00	1	0.007
AvoidSettersForHeavilyUsedAtt.[1]	16	0.05	6	0.02
CheckParametersOfSetters	132	0.34	462	1.00
DontReturnCollectionsOrArrays	59	0.15	71	0.15
UseInterfaceAsReturnType	315	0.82	228	0.50
Deficient Encapsulation @Compo.		55.41		6.18
AvoidPublicStaticVariables	43	0.55	11	0.06
Incomplete Abstraction @Package		95.00		100.0
UseAbstraction	19	0.95	160	1.00
Unexploited Hierarchy @Class		100.0		100.0
AvoidRuntimeTypeIdentification	539	1.00	74	1.00

[1] AvoidSettersForHeavilyUsedAttributes

Suggestions for Improvement

A rule that is a good indicator for identifying design flaws regarding OCP is *AvoidRuntimeTypeIdentification*. jEdit has 539 violations of this rule that need to be further investigated to find a real design problem. For doing so, it is important to focus on code fragments that show an accumulation of these violations. In fact, there is one class containing a method with six type identifications in it. This is the method *showPopupMenu* that has the general purpose of building a popup menu item depending on a selected tree node item.

When reading the source code for understanding the functionality, it is obvious that four different abstractions determine the composition of the popup menu item. Therefore, the method *showPopupMenu* contains various *if-statements* with type identifications to differ the compilation of the menu item. Assuming a new type of tree node is added to the design, the method must be extended to support the new abstraction. Consequently, this design is not open for extension without modifying existing code.

In order to comply with OCP, the functionality of building the popup menu item must get closer to the abstractions that know how to extend the item. Thus, the four abstractions – *HyperSearchFileNode*, *HyperSearchResult*, *HyperSearchFolderNode*, and *HyperSearchOperationNode* – must implement the same interface or must be derived from the same base class. Since there is already a joint interface for *HyperSearchFileNode* and *HyperSearchResult* it makes sense to use it for this improvement by introducing an additional method, *e.g.*, *buildPopupMenu*.

Given this interface extension, the four abstractions of search node must implement the functionality of building the popup menu item with content depending on their own requirements. Consequently, the logic that is implemented in the *showPopupMenu* method moves to the classes that should be responsible therefore. Additionally, the amount of code in *showPopupMenu* reduces dramatically because it just has to create a popup menu item that is handed over to the tree node abstractions by using the method *buildPopupMenu* defined in their interface. Then the popup menu needs to be returned from the *buildPopupMenu* to get displayed by *showPopupMenu*. All in all,

this significantly increases the design in order to support additional tree node abstractions and for enhancing maintainability.

Following the OCP is at the heart of good object-oriented design and urges foresight for extensions. Thus, we discussed this suggestion with the developer of jEdit from the viewpoints of refactoring the current implementation and supporting upcoming abstractions in this hierarchy tree. While he argues that it is worth to address the improvement, there is no need to support additional node elements. Further, the change would affect multiple classes what keeps him cautious in submitting a change request. Nevertheless, the developer agrees that the general design would enhance when introducing the additional abstraction level as base type of all node elements.

5.6 Discussion of SOC Assessment

Finally, the last design principle assessment focuses on SOC that is multifaceted and includes various rules. Since there are different rules involved, this discussion will summarize the evaluation characteristics mentioned above. Thus, the approach of normalizing findings with different entity sizes can be observed at different rules. For instance, the rules *AvoidLongParameterLists* and *AvoidLongMethods* are working on methods level, which is why their number of violations is divided by the number of methods. In contrast, the rule *AvoidDuplicates* is normalized based on logical lines of code because it is the only meaningfully size entity that works on the source code level.

When considering the rule *AvoidDuplicates* in more detail, it can be identified that the characteristics of the two projects are represented by the benchmark suite from which the threshold for the evaluation function is derived. The reason therefore is that TuxGuitar, which has the most findings at this rule, is neither under- nor overestimated. In other words, the 572 findings of TuxGuitar define the upper threshold and cause an assessment of 99.70 points while the 37.97 points of jEdit are relative to the 572 findings of TuxGuitar. All in all, the low number of duplicates is also one reason for the better principle assessment of jEdit over TuxGuitar based on the same benchmark base.

Despite the high number of duplicates, there are two further deviations that cause more points for jEdit than for TuxGuitar. First, the *Duplicate Abstraction @Type* property, which also impacts DRY as already discussed in the previous section, gets a better assessment for jEdit compared with TuxGuitar based on fewer findings. Second, jEdit profits from a lower property weight for *Complex Structure @Method*. In fact, the 47 long methods are influencing the design principle assessment with half of the weight compared to others since we (currently) consider this aspect as less important for assessing SOC.

Table 8: Measuring Result of SOC for jEdit and TuxGuitar

	jEdit		TuxGuitar	
Separation of Concern Principle		5.96		4.66
Abstraction @Method		3.07		7.33
AvoidLongParameterLists	7	0.03	20	0.07
Abstraction-Deficient Hier. @Pkg		25.00		50.00
AvoidRepetitionOfPkgNamesOnPath	0	0.00	0	0.00
CheckSameTermsOnDifferentPkgL.[1]	2	0.50	35	1.00
Complex Hierarchy @Package		50.00		39.21
AvoidHighNumberOfSubpackages	1	0.50	4	0.39
Complex Structure @Method		20.61		7.70
AvoidLongMethods	47	0.20	21	0.07
Degraded Hierarchy @Package		50.00		29.41
CheckDegradedDecompo.OfPkg[2]	1	0.50	3	0.29
Duplicate Abstraction @Type		44.89		66.66
AvoidSimilarAbstractions	32	0.50	240	1.00
AvoidSimilarNamesOnDiffAbs.L.[3]	6	0.09	0	0.00
AvoidSimilarNamesOnSameAbs.L.[4]	48	0.75	666	1.00
Duplication @SourceCode		37.97		99.70
AvoidDuplicates	299	0.38	572	0.99
Ill-suited Abstraction @Class		0.78		1.10
AvoidManyTinyMethods	1	0.01	2	0.01

[1] CheckSameTermsOnDifferentPackageLevels

[2] CheckDegradedDecompositionOfPackages

[3] AvoidSimilarNamesOnDifferentAbstractionLevels

[4] AvoidSimilarNamesOnSameAbstractionLevel

Suggestions for Improvement

By analyzing the *AvoidLongParameterLists* rule, we found two problem areas that both deal with imprecise abstractions. The required details for one of these problem areas are shown in Listing 3. According to this listing, the interface *FoldPainter* defines three methods and both *TriangleFoldPainter* and *ShapedFoldPainter* implement this interface. When further analyzing the classes, it can be seen that just a subset of the interface is needed, *e.g.*, *TriangleFoldPainter* does not provide an implementation for *paintFoldMiddle*.

The second design problem concerns the parameter list of *paintFoldStart* which is too specific. In other words, the implementations in the classes do not need all parameters such as *screenLine*, *physicalLine*, and *buffer*. To fix this design issue, the interface must be reduced to those methods and parameters that are needed by the sub-types.

Listing 3: Violation of AvoidLongParameterLists

```
public interface FoldPainter {
```

```
  void paintFoldStart(Gutter gutter, Graphics2D
      gfx, int screenLine, int physicalLine,
      boolean nextLineVisible, int y, int
      lineHeight, JEditBuffer buffer);

  void paintFoldEnd(Gutter gutter, Graphics2D
      gfx, int screenLine, int physicalLine,
      int y, int lineHeight, JEditBuffer
      buffer);

  void paintFoldMiddle(Gutter gutter, Graphics2D
      gfx, int screenLine, int physicalLine,
      int y, int lineHeight, JEditBuffer
      buffer);
}

public class TriangleFoldPainter implements
    FoldPainter

public abstract class ShapedFoldPainter
    implements FoldPainter
```

The design flaw and suggestion for improvement have been shown to the developer of jEdit. He responded with accepting the current implementation and not addressing the design flaw. His reason therefore is that he does not want change the interface – *FoldPainter* – as there are depending components.

6 Threats to Validity

This section presents potential threats to validity for the derived DQM and its application in the case study. Specifically, it focuses on threats to internal, external, and construct validity. Threats to internal validity concern the selection of the projects, tools, and the analysis method that may introduce confounding variables [36]. Threats to external validity refer to the possibility of generalizing the findings and construct validity threats concern the meaningfulness of measurements and the relation between theory and observation [36].

6.1 Internal Validity

For the presented case study we chose jEdit and TuxGuitar as systems for study. Despite two independent development teams with different engineering skills, we tried to control this threat by choosing projects with a similar size and application domain. Moreover, both projects have multiple versions meaning that they have been further developed and refactored over a longer period of time. Based on these project characteristics, the internal threat of validity concerning the selection of projects is addressed by having two projects that are basically comparable.

MUSE is used as measuring tool and poses threats to internal validity as well. In more detail, MUSE relies on the commercial tool *Understand* that is used to extract meta-information from the source code [31]. This information is then used by rules implementations to verify the compliance of design best practices. Although the querying and processing of the meta-information is complex, excessive tests and applications of MUSE – without DQM – in various industrial projects do not report performance or measuring problems.

Another threat to internal validity is the use of thresholds in evaluation functions. The discussion in Section 5.5 focuses on this concern and mentions that the thresholds are derived from a benchmark suite. At this point it is important to highlight that these thresholds must be used carefully; especially, when the target project addresses another application domain compared to the benchmark base.

In Section 3, a survey about the importance of design principles is presented, which contains a threat to the internal validity of its result. The reason is that the participants had to rank a list of pre-selected design principles. Consequently, the participants were biased based on our selection. Nevertheless, we accept this threat to validity as this survey aimed to get a sense of the importance of principles instead of addressing the completeness of the list. Furthermore, the pre-selected design principles were identified systematically from the research literature.

6.2 External Validity

Regarding the threats to external validity it must be noticed that the case study compares only two systems. These systems have a specific architecture and are developed by different teams which may have their specific design rules. Thus, we cannot – and do not want to – claim that our current DQM fits the design requirements for every project. For generalizing results, further validation work must consider different systems with different teams, sizes, and application domains. However, we know that our DQM supports the assessment of various projects since it provides regulators for adjusting it.

Next to the internal threat to validity of the survey, it also has an external threat reflecting the generalization of the final ranking. To control this threat, we tried to distribute the questionnaire as far as possible. The analysis of the demographic data shows that the questionnaire has been completed by participants from different software en-

gineering domains and engineering roles. This minimizes this threat and gives us the chance to generalize the result.

6.3 Construct Validity

As the main goal and theory behind the DQM is the assessment of design principles based on violations of design best practices, it is important to comment on this idea. We know that the DQM is not yet complete and contains white spots that need to be filled by future work. Nevertheless, for the first presentation of this approach and the discussion of the assessment characteristics of DQM in this paper it is considered as mature enough.

7 Conclusion and Future Work

While established approaches concentrate on metrics for assessing design quality, this work provides support for better understanding object-oriented design issues and to provide a path to improve the design. For placing this novel idea of the DQM into the right corner within the research area, the paper reflects on related and fundamental work. The result is an ontology that describes the relationship between terms used by the research community and helps to properly place the DQM.

As the discussion of the DQM in the case study of this paper shows, it is a comprehensive model and leads to traceable results as problematic design is identified by the violations of design best practices. These violations give (1) more detailed information on design flaws compared to single metrics and (2) cover design aspects (*e.g.*, encapsulation / information hiding principles) more comprehensively.

The DQM presented in this article is a step towards a comprehensive quality model as its content is derived from a systematic analysis (see Section 4.5 for details on the development process of DQM) of well-known design principles. Our survey on object-oriented design principles convinces us that following this path could lead to a more comprehensive model. The emphasis of this article – at least for the case study – was on the assessment of object-oriented design quality. We have not yet validated the completeness of our model, *i.e.*, whether our design principles are properly and sufficiently covered by the specified design best practices.

In general, quality models can serve different application purposes, such as specifying, measuring, assessing, improving, managing and predicting quality [6]. While this article focus on the benefits of the DQM for assessing and improving design, the discussion of applying DQM to two software products strengthens our confidence that it contributes to a better understanding and characterization of design principles. This goes beyond the conceptual discussions of principles and is more focused on which (measurable) design best practices capture design principles technically.

From the discussions with the jEdit developer we learnt that it is difficult to refactor a running system even though design quality would improve. One reason therefore is that developers are averse of changing source code that does not contain an obvious bug. Thus, we propose to continuously measure, assess and improve the design quality of the software product and to consider the quality management process as part of the software development process. Consequently, design flaws are addressed before the product will be deployed and further changes become difficult to implement.

Our DQM and the underlying measuring tool MUSE support the quality management process combined with the software development process since MUSE can be integrated into the build process of a system [31]. Besides, the assessments of DQM can be uploaded to a quality management environment (SonarQube [1]) for discussing upcoming decisions and for controlling enhancements. This also provides the possibility to discuss the quality of the software from various viewpoints such as from the view of, *e.g.*, the project manager, product owner, quality manager, etc.

This paper does not touch the application purposes of design quality management and prediction while future work will address these aspects specifically. Therefore, we plan to apply the DQM within an industrial setting and in cooperation with open source communities. Furthermore, we are interested in understanding the evolution of design best practices over a long time period and whether shifts in the design can be recognized and predicted. For understanding the importance of a design assessment and how well the derived improvements fit within the software development process, we are cooperating with local partners and guide their projects.

Acknowledgement: DQM is based on many constructive discussions about measuring object oriented design. We would like to thank H. Gruber and A. Mayr for their contributions to DQM.

1 http://www.sonarqube.org/

References

[1] Chidamber S.R., Kemerer C. F., A metrics suite for object oriented design, IEEE Transactions on Software Engineering, 1994, 20(6), 476–493

[2] Marinescu R., Ratiu D., Quantifying the quality of object-oriented design: The factor-strategy model, Proceedings of the 11th Working Conference on Reverse Engineering, Delft, The Netherlands, 2004, 192–201

[3] Fowler M., Beck K., Brant J., Opdyke W., Refactoring: Improving the Design of Existing Code, Addison Wesley, Reading, US, 1999

[4] Moha N., Guéhéneuc Y.G., Duchien L., Le Meur A.F., DECOR: A Method for the Specification and Detection of Code and Design Smells, IEEE Transactions on Software Engineering, 2010, 36(1), 20–36

[5] Samarthyam G., Suryanarayana G., Sharma T., Gupta S., MIDAS: A design quality assessment method for industrial software, Proceedings of the 35th International Conference on Software Engineering (ICSE 2013), San Francisco, US, 2013, 911–920

[6] Kläs M., Heidrich J., Münch J., Trendowicz A., CQML Scheme: A Classification Scheme for Comprehensive Quality Model Landscapes, Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2009), Patras, Greece, 2009, 243–250

[7] Coad P., Yourdon E., Object-Oriented Design, Prentice Hall, London, UK, 1991

[8] Henderson-Sellers B., Constantine L.L., Graham I.M., Coupling and cohesion (toward a valid metrics suite for object-oriented analysis and design), Object Oriented Systems, 1996, 3(3), 143–158

[9] Dooley J., Object-Oriented Design Principles, in Software Development and Professional Practice, Apress, 2011, 115–136

[10] Sharma T., Samarthyam G., Suryanarayana G., Applying Design Principles in Practice, Proceedings of the 8th India Software Engineering Conference (ISEC 2015), New York, US, 2015, 200–201

[11] Riel A.J., Object-Oriented Design Heuristics, 1st ed, Addison-Wesley Longman Publishing, Boston, US, 1996

[12] Brown W., Malveau R., McCormick H., Mowbray T., AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, Wiley and Sons, New York, US, 1998

[13] Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education India, 1995

[14] Muraki T., Saeki M., Metrics for Applying GOF Design Patterns in Refactoring Processes, Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE 2001), New York, US, 2001, 27–36

[15] Boehm B. W., Brown J.R., Kasper M., Lipow M., Macleod G.J., Merrit .M.J., Characteristics of software quality, North-Holland, 1978

[16] Dromey R.G., A model for software product quality, IEEE Transactions on Software Engineering, 1995, 21(2), 146–162

[17] Al-Kilidar H., Cox K., Kitchenham B., The use and usefulness of the ISO/IEC 9126 quality standard, International Symposium on Empirical Software Engineering 2005, Queensland, Australia, 2005, 126–132

[18] Mordal-Manet K., Balmas F., Denier S., Ducasse S., Wertz H., Laval J., et al., The squale model - A practice-based industrial quality model, Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM 2009), Alberta, Canada, 2009, 531–534

[19] Wagner S., Goeb A., Heinemann L., Kläs M., Lochmann K., Plösch R., et al., The Quamoco product quality modelling and assessment approach, in Proceedings of the 34th International Conference on Software Engineering (ICSE 2012), Zurich, Switzerland, 2012, 1133–1142

[20] Bansiya J., Davis C., A hierarchical model for object-oriented design quality assessment, IEEE Transactions on Software Engineering, 2002, 28(1), 4–17

[21] ISO/IEC 9126-1:2001 - Software engineering – Product quality – Part 1: Quality model, ISO/IEC, ISO/IEC 9126:2001, 2001. [Online]. Available: http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749

[22] Wagner S., Goeb A., Heinemann L., Kläs M., Lampasona C., Lochmann K., et al., Operationalised product quality models and assessment: The Quamoco approach, Information and Software Technology, 2015, 62, 101–123

[23] Martin R.C., Agile software development : principles, patterns and practices, Pearson Education, Upper Saddle River, US, 2003

[24] Dijkstra E.W., On the role of scientific thought. [Online]. Available: http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html

[25] Laplante P.A., What Every Engineer Should Know about Software Engineering, CRC Press, Boca Raton, US, 2007

[26] Parnas D.L., Software Aging, in Proceedings of the 16th International Conference on Software Engineering (ICSE 1994), Los Alamitos, US, 1994, 279–287

[27] Hunt A., Thomas D., The Pragmatic Programmer: From Journeyman to Master, Addison-Wesley Longman Publishing, Boston, US, 1999

[28] Mayr A., Plösch R., Klas M., Lampasona C., Saft M., A Comprehensive Code-Based Quality Model for Embedded Systems: Systematic Development and Validation by Industrial Projects, in Proceedings of the 23rd International Symposium on Software Reliability Engineering (ISSRE 2012), Dallas, US, 2012, 281–290

[29] Mayr A., Plösch R., Saft M., Objective Measurement of Safety in the Context of IEC 61508-3, in Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2013), Washington, US, 2013, 45–52

[30] Dautovic A., Automatic Measurement of Software Documentation Quality, PhD thesis, Deptartment for Business Informatics, Johannes Kepler University Linz, Austria, 2012

[31] Plösch R., Bräuer J., Körner C., Saft M., MUSE - Framework for Measuring Object-Oriented Design, Journal of Object Technology, 2016, 15(4), 2:1–29

[32] Barron F.H., Barrett B.E., Decision Quality Using Ranked Attribute Weights, Management Science, 1996, 42(11), 1515–1523

[33] Edwards W., Barron F.H., SMARTS and SMARTER: Improved Simple Methods for Multiattribute Utility Measurement, Organizational Behavior and Human Decision Processes, 1994, 60(3), 306–325

[34] Gruber H., Plösch R., Saft M., On the Validity of Benchmarking for Evaluating Code Quality, in Proceedings of the Joined International Conferences on Software Measurement IWSM/MetriKon/Mensura 2010, Aachen, Germany: Shaker Verlag, 2010

[35] Bräuer J., Plösch R., Saft M., Measuring Maintainability of OO-Software - Validating the IT-CISQ Quality Model, in Proceedings of the 2015 Federated Conference on Software Development and

Object Technologies (SDOT 2015), Zilina, Slovakia, 2015
[36] Wohlin C., Runeson P., Höst M., Ohlsson M.C., Regnell B., Wesslén A., Experimentation in Software Engineering, Springer, Berlin, Heidelberg, 2012

Appendix

Benchmark Suite

The following table shows the entire list of open-source projects representing the benchmark suite for our DQM. More specifically, thresholds derived from this suite are used for the evaluation functions.

Table A.1: Projects in Benchmark Suite

Name	Version	Logical Lines of Code
Ant	1.7.0	113,291
ArgoUML	0.24	185,897
Azureus	3.0.4.2	456,113
FreeMind	0.8.1	79,812
GanttProject	2.0.6	56,944
hsqldb	1.8.0.9	69,302
JasperReports	2.0.5	145,561
jDictionary	1.8	5,967
jEdit	4.2	81,754
jFreeChart	1.0.9	131,456
jose	1.4.4	110,735
junit	4.4	5,476
Lucene	2.3.1	34,939
Maven	2.0.7	36,093
OpenCMS	7.0.3	260,891
OpenJGraph	0.9.2	13,107
OurTunes	1.3.3	16,303
Pentaho	1.6.0	70,911
PMD	4.1	51,819
Risk	1.0.9.7	30,637
Spring	2.5	127,118
Tomcat	6.0.16	207,733
TuxGuitar	0.9.1	45,390
Weka	3.5.7	287,803
XDoclet	1.2.3	9,259
XWiki	1.3	80,339

Additional Principle Assessments

In addition to the measuring results shown and discussed in Section 5, the next tables provide measuring results for further design principles listed in Table 1. This should reflect the compressibility and variety of design aspects covered by the DQM.

Table A.2: Measuring Result of ADP for jEdit and TuxGuitar

	jEdit		TuxGuitar	
Acyclic Dependency Principle		0.00		0.00
Cyclic Dependency @Package		100		100
MaxStronglyConnectedComponents	1	1.00	1	1.00

Table A.3: Measuring Result of CCP for jEdit and TuxGuitar

	jEdit		TuxGuitar	
Common Closure Principle		2.08		2.97
Behavioral Disintegrity @Method		87.50		94.60
AbstratPkgShouldNotRelyOnOPkg.[1]	3	0.75	16	0.78
AvoidStronglyCoupledPackageImpl.[2]	27	1.00	163	1.00
ConcretePkgAreNotUsedFromOPkg.[3]	3	0.75	134	1.00
PackageShouldUseMoreStablePkg.[4]	15	1.00	54	1.00
Non-Cohesive Structure @Package		70.90		45.88
AvoidNonCohesivePkgImpl.[5]	11	0.709	68	0.459

[1] AbstratPackagesShouldNotRelyOnOtherPackages

[2] AvoidStronglyCoupledPackageImplementation

[3] ConcretePackagesAreNotUsedFromOtherPackages

[4] PackageShouldUseMoreStablePackages

[5] AvoidNonCohesivePackageImplementation

Table A.4: Measuring Result of CQS for jEdit and TuxGuitar

	jEdit		TuxGuitar	
Command-Query Separation		8.29		9.34
Coupled Structure @Package		17.00		6.54
AvoidCommandInQueryMethods	83	0.10	58	0.06
AvoidReturningDataFromComm.[1]	615	0.80	271	0.29
DontReturnUninv.DataFromComm.[2]	54	0.07	8	0.01

[1] AvoidReturningDataFromCommands

[2] DontReturnUninvolvedDataFromCommands

Table A.5: Measuring Result of FCOI for jEdit and TuxGuitar

	jEdit		TuxGuitar	
Favour Composition Over Inheritance		5.75		9.03
Abused Hierarchy @Class		42.41		9.63
CheckUnusedSupertypes	90	0.47	27	0.09
UseCompositionNotInheritance	55	0.28	24	0.08

Table A.6: Measuring Result of ISE for jEdit and TuxGuitar

		jEdit		TuxGuitar
Interface Separability		6.55		8.22
Coupled Structure @Class		26.30		51.33
AvoidMultipleImpl.Instantiations	70	0.21	69	0.15
CheckExistenceImpl.ClassesAsStr.[1]	18	0.16	1	0.01
DontInstantiateImpl.InClient[2]	374	0.33	1k	1.00
Cyclic Hierarchy @Class		3.13		0.0
AvoidUsingSubtypesInSupertypes	2	0.00	0	0.00
Overlooked Abstraction @Class		8.21		12.91
ProvideInterfaceForClass	66	0.10	219	0.24
UseInterfaceIfPossible	686	0.06	140	0.01
Polygonal Hierarchy @Class		100.0		6.60
AvoidDiamondInh.StructuresInter.[3]	584	1.00	30	0.06

[1] CheckExistenceImplementationClassesAsString

[2] DontInstantiateImplementationsInClient

[3] AvoidDiamondInheritanceStructuresInterfaces

Table A.7: Measuring Result of ISP for jEdit and TuxGuitar

		jEdit		TuxGuitar
Interface Segregation Principle		9.06		7.90
Large Abstraction @Class		9.39		20.92
AvoidPartiallyUsedMethodInterfaces	18	0.09	57	0.20

Table A.8: Measuring Result of PINI for jEdit and TuxGuitar

		jEdit		TuxGuitar
Program to an Interface, not an Implementation		9.17		8.70
Overlooked Abstraction @Class		8.21		12.91
ProvideInterfaceForClass	66	0.10	219	0.24
UseInterfaceIfPossible	686	0.06	140	0.01

Table A.9: Measuring Result of SDOP for jEdit and TuxGuitar

		jEdit		TuxGuitar
Self-Documentation Principle		8.38		2.39
Document Disintegrity @Class		11.59		100
DocumentYourPublicClasses	37	0.11	995	1.00
Document Disintegrity @Method		20.74		52.09
AvoidMassiveCommentsInCode	82	0.10	38	0.42
DocumentYourPublicMethods	583	0.30	5k	1.00

Table A.10: Measuring Result of YAGNI for jEdit and TuxGuitar

		jEdit		TuxGuitar
You Ain't Gonna Need It		5.89		8.13
Trivial Abstraction @Class		18.79		12.11
AvoidAbstractClassesWithOneExt.[1]	12	0.18	11	0.12
Unitilized Abstraction @Class		43.16		25.27
AvoidUnusedClasses	34	0.26	44	0.24
AvoidUnimplementedInterfaces	8	0.59	4	0.26

[1] AvoidAbstractClassesWithOneExtension