Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

MUSE
A Framework for Measuring Object-Oriented Design Quality

Reinhold Plösch[a] Johannes Bräuer[a] Christian Körner[b]
Matthias Saft[b]

a. Johannes Kepler University Linz, Department of Software Engineering, Linz Austria
http://www.se.jku.at

b. Siemens AG, Corporate Technology, Munich, Germany
http://www.siemens.com

Abstract Good object-oriented design facilitates the maintainability of a software product. While metric-based approaches and the idea of identifying design smells have been established, there still remains the gap of verifying the compliance of design best practices in source code. Thus, there is no comprehensive set of metrics or design best practices that does not only support design measurement and evaluation but can also guide the improvement process. This paper proposes a novel approach based on measuring design best practices that closes the gap between the identification of design flaws and the support for improvements. An expert group six researchers captured a set of 67 design best practices that are implemented by the framework MUSE (Muse Understand Script Engine). For a first validation of MUSE in this paper, its measuring result is compared with QMOOD, which is an established metric-based approach for measuring the quality of object-oriented design. The qualitative assessment based on data from six versions of the Java tool jEdit shows that MUSE is better suited to guide improvements than QMOOD, e.g., for the design property encapsulation QMOOD indicates no substantial changes in the design quality while the data provided by MUSE highlights that the encapsulation property of jEdit became worse over time. These first promising results of the application of MUSE have to be further validated and future work will concentrate on measuring object-oriented design principles.

Keywords design best practices, object-oriented design quality, design quality assurance, design measurement

1 Introduction

Software design decisions always lead to inevitable tradeoffs that are necessary to handle product requirements. Although the nature of object-oriented design already avoids the introduction of obvious pitfalls, some design decisions are still error prone. Consequently, developing a product without controlling and assessing the software and design quality possibly results in a product that implements the functional requirements but fails to fulfill quality aspects such as maintainability, extensibility, or reliability.

For dealing with the challenges of assessing design quality, fundamental approaches, e.g., [CK94] and [BD02], proposed metric-based suites for checking object-oriented design aspects. Typically, such suite consists of a set of metrics that expresses object-oriented characteristics of the source code by a single value. However, real design issues often cannot be directly identified when using single metrics and considering them in isolation [Mar04]. In other words, metrics can raise an alert that the software quality fails at some points, but software engineers and architects lack specific clues how to improve the design. Thus, if measurement is not only about the status of a piece of software with respect to design quality but should also provide better support for improvements, more advanced approaches are necessary.

In previous work, we have already made good experience and gained knowledge in assessing source code quality by measuring quality aspects based on violations of best coding practices as shown in, e.g., [PGH$^+$07], [WGH$^+$12] and [MPS14]. We want to map this idea to the domain of object-oriented design by identifying violations of object-oriented design best practices directly from the source code. The application of this approach presumes tool support for the definition of design best practices and their automatic identification. Due to the lack of a comprehensive tool that is designed for this purpose and provides the flexibility for future extensions, we have developed an analysis framework that fits our needs.

In this paper, we introduce the framework MUSE (MUSE Understand Scripting Engine). MUSE is a Perl library aimed to measure object-oriented design by identifying violations of design best practices based on meta-information from source code. These violations can be visualized in SonarQube[1] dashboards for supporting software developers and software architects in detecting design problems in the source code. Besides, MUSE provides an explanation and self-explaining name for each violation in order to best guide software engineers in correcting the design problems and implementing improvements.

The purpose of this work is to compare the measuring results provided by MUSE to a well established and documented metric-based assessing approach for object-oriented design quality. Specifically, the quality model QMOOD (Quality Model for Object-Oriented Design) [BD02] is selected because it claims to provide a practical quality assessment approach for a number of design properties as shown in Section 5. Moreover, QMOOD is relying on a metric suite, which is hierarchically structured and the starting point for estimating the design quality. In order to compare the result of MUSE with QMOOD, this paper focuses on a qualitative discussion about the measuring results of several releases of the open-source Java project jEdit. Finally, the comparison reveals the benefits and drawbacks of using MUSE combined with its underlying concept of finding design best practice violations. QMOOD and MUSE have completely different approaches to measurement; while QMOOD relies on metrics, MUSE focuses on the violation of design best practices. Nevertheless, both approaches

[1]https://sonarqube.org

claim to provide insigths into object-oriented design properties like abstraction or encapsulation. In this paper we want to show that a best practice based approach like MUSE is more suitable than a metric-based approach like QMOOD - at least when we have systematic and sustainable quality as well as design improvement in mind.

The rest of the paper is organized as follows: First, Section 2 shows related work in this research area and discusses it within the context of this article. Section 3 specifies requirements for our toolset and Section 4 gives an overview of the architecture and of selected implementation aspects of MUSE. Section 5 provides insights into the QMOOD model and describes how QMOOD specifications are adopted to measure Java projects. Section 6 highlights the results of the validation. Hence, it shows the detailed result from applying QMOOD and MUSE on six versions of jEdit. The discussion underlines the usefulness of the data provided by MUSE on a qualitative level and makes some quantitative comparisons of the measurement result from QMOOD and MUSE. Finally, Section 7 discusses various threats to validity before Section 8 presents the conclusions drawn from comparing MUSE with QMOOD including thoughts and ideas for future work.

2 Related Work

Primarily, related work in this research area comprises approaches and tools that are used to assess software quality in general. Nevertheless, the discussion in this section concentrates on those tools and approaches that are applied for assessing and improving object-oriented design directly from the source code in order to better align the related work with our MUSE approach.

2.1 Design Measuring Approaches

When investigating the literature of measuring object-oriented design quality, there are multiple approaches that focus on finding design flaws usually referred to as bad smells [FBBO99]. One of these approaches for detecting bad smells is based on so called detection strategies [Mar04]. Such a detection strategy relies on measuring different object-oriented design metrics and combining them to one metric-based rule. This allows reaching a higher abstraction level in working with metrics and expressing design flaws in a quantifiable manner.

As shown by the idea of detection strategies and also considered by the MUSE framework, it is necessary to investigate multiple problem sources to identify one design flaw. In knowing the problem sources that are causing a particular design issue, it is easier to localize the fragments and to refactor the design at this point.

Another and more recently published work in specifying and detecting bad smells is presented by Moha et al. [MGDLM10]. In this work, the authors introduce DÉCOR a method that illustrates all necessary steps to define a detection technique. Specifically, step three of DÉCOR is the translation of a textual bad smell definition into algorithms that can be applied for detecting it. To validate the method, the authors instantiated DÉCOR and tested the detection techniques on four bad smells.

The idea behind DÉCOR is similar to our goal of identifying violations of design best practice in an automatic manner. However, the generated detection algorithm just supports the investigation of Java source code. The reason is that the algorithm generator works with templates that are excerpts of Java code and used for replacing the concrete source code [MGMD08]. Based on the limitation of the programming language and the actual purpose of finding bad smells, we could not build on the DÉCOR framework.

2.2 Design Measuring Tools

While formal definitions of measuring approaches are important for demonstrating their concept and ideas, they cannot be applied in a development environment; especially, without tool support for quality analysis (QA). In order to support software engineers, there are multiple tools available that can be divided into two categories: QA tools integrated in IDEs and standalone QA tools.

An example for an IDE integrated QA tool is inCode [MGV10]. InCode aims to detect design smells based on the detection strategies mentioned before. Thus, it is obvious that it has been developed by the authors who proposed the concept of detection strategies. With the intent to transform quality assessment from a one-time activity to a continuous and development life-cycle integrated task, inCode is designed as Eclipse plugin [GVM15].

As major benefit, the developers of inCode point out that the tool better guides refactoring tasks. The reason is that design flaws are immediately reported to the software engineer who is working on the source code and is familiar with the context of the design problem. In fact, this is an advantage but results in the compromise that inCode only supports Java. Since one of our requirements for MUSE demands support for multiple programming languages, this tradeoff could not be accepted.

Next to inCode, there are additional Eclipse plugins like Checkstyle[2]. This plugin can be triggered by the build process of the project and detects code-style violations. Due to its purpose of checking coding standards, Checkstyle is not applicable for measuring object-oriented design properties as intended by MUSE.

In contrast to IDE integrations, there exist (open-source) quality assessment tools such as PMD[3] and Findbugs[4]. Both are static code analyzers that work on a rule base and identify programming flaws like empty catch blocks and unused variables. Compared to Findbugs, which can analyze Java source code only, PMD supports multiple programming languages except of C++ and C#. Consequently, this was one reason why these tools did not meet our requirements too. Furthermore, PMD and Findbugs also concentrate on code quality with just a few rules for object-oriented design.

2.3 Problem Statement

As shown by the related work above, the research community in the field of object-oriented software design assessment has no tool available that (1) supports the analysis of multiple programming languages, (2) can be extended by self-developed design rules, (3) and can be integrated into a source code analysis environment such as ConQAT[5] or SonarQube.

[2]http://checkstyle.sourceforge.net/
[3]https://pmd.github.io/
[4]http://findbugs.sourceforge.net/
[5]https://www.conqat.org/

3 Requirements for MUSE

In order to address the problem statement mentioned above, this section describes the requirements that had been specified beforehand and taken into consideration while MUSE has been developed. For the specification of these requirements it was important to stress the framework character of MUSE to support customization and extensability.

3.1 Capture essential object-oriented Design Properties

As the most important functional requirement, MUSE should help to check object-oriented design best practices automatically. Relevant best practices were identified by a literature review in the fields of object-oriented design smells (e.g., [FBBO99] and [Rie96]) and object-oriented design principles (e.g., [Lis87] and [Doo11]). The selection and specification of design best practices was carried out based on the results found from studying the literature by up to six researchers. Each decision, i.e., which best practice to include in MUSE, was at least discussed by a team of four researchers. As a result, MUSE currently provides 67 design best practices. We did not - at any time in the development process - look at metric-based approaches like QMOOD.

Tables 1 - 4 show a subset of the 67 rules selected for the programming language Java and for the design properties *abstraction*, *coupling*, *encapsulation* and *inheritance*. Consequently, 21 design best practices are listed here while the rest is shown in the appendix. Intentionally, we concentrate on these 21 design best practices as they naturally fit in with the selection of design properties covered by the QMOOD approach, which is then used for the validation of our MUSE approach (see Section 5).

Table 1 – Abstraction-related design best practices

Design Best Practice	**Definition**
AvoidPartiallyUsed-MethodInterfaces	A class should be used as a whole and not only partially, i.e., for each client class we find out the set of used methods. The more client-classes use the entirety of the provided methods of a class, the better.
AvoidSimilarAbstractions	Different types should not represent a similar structure or behavior. Two classes have a similar structure if the attributes with same type and a similar name (word stem) overlap by a particular percentage. Two classes have a similar behavior if methods with the same return and parameter types as well as similar name (word stem) overlap by a particular percentage.
AvoidSimilarNamesOn-DifferentAbstractionLevels	Types, such as classes, interfaces, or packages, on different abstraction levels should not have similar names.
AvoidSimilarNamesOn-SameAbstractionLevel	Types, such as classes, interfaces, or packages, on the same abstraction level should not have similar names.
ProvideInterfaceForClass	A public class has to implement an interface. Classes that provide only access to static members are excluded by this rule.
UseInterfaceIfPossible	Use the interface of a class for variable declarations, parameter definitions, or return types instead of the public class when the interface provides all operations that are needed. If there exists more than one interface, it has to be checked whether one of the available interfaces provides the required operations.

Table 2 – Coupling-related design best practices

Design Best Practice	Definition
AvoidMultipleImplementationInstantiations	The instantiation of a class by other classes should be as restricted as much as possible - with exception of factory classes.
AvoidStronglyCoupledPackageImplementation	A package should not heavily rely on other packages. Therefore, the dependencies between classes, which can be either an implementation, use, call of a class or an interface, are aggregated to package level for expressing the relatedness to other packages.
CheckExistenceImplementationClassesAsString	A class should not be instantiated by its class name stored as string value.
DontInstantiateImplementationsInClient	A class should not be instantiated by a client that is not within the same package.
PackageShouldUseMoreStablePackages	A package should use more stable packages, i.e., the stability of a package defines the minimum stability of its dependent packages.

Table 3 – Encapsulation-related design best practices

Design Best Practice	Definition
AvoidExcessiveUseOfGetters	The ratio between getter methods and the total number of non-const attributes should not exceed a certain threshold.
AvoidExcessiveUseOfSetters	The ratio between setter methods and the total number of non-const attributes should not exceed a certain threshold.
AvoidProtectedInstanceVariables	A class should avoid having protected attributes.
AvoidPublicInstanceVariables	A class should avoid having public attributes.
AvoidPublicStaticVariables	Global variables, i.e., public static attributes of a class, should be avoided.
AvoidSettersForHeavilyUsedAttributes	There should not exist setter methods for a private class attribute that is heavily used. An attribute is heavily used if it is read or written in more than five methods including accessor methods.
CheckParametersOfSetters	Class attributes should only be set by method parameters that are checked before. This can be verified by checking whether setting the attribute by a parameter of a (set)-method is always (or at least often) guarded by checks.
UseInterfaceAsReturnType	If the return type of a method is not a base data type, it should be the interface of the class. If a class that is used as return type inherits from an abstract class, this abstract class should at least be used as return type.

3.2 Support for multiple Programming Languages

Another important requirement is that MUSE must support the object-oriented programming languages Java, C++, and C#[6]. This requires that MUSE has to be

[6]The definitions given in Tables 1 - 4 are valid for Java since the validation shown in this work relies on the open-source Java project jEdit. We also provide implementations for C++ and C# with slightly adopted definitions depending on language particularities (see Appendix).

Table 4 – Inheritance-related design best practices

Design Best Practice	Definition
CheckUnusedSupertypes	If clients (not subclasses!) of a class use only the public interface (methods) of the current subtype and do not use any methods of a supertype, then there is not a true is-a relationship between the class (subtype) and its supertype.
UseCompositionNot-Inheritance	A class should use composition instead of inheritance when the class accesses only public members (methods or attributes) from the particular superclass. Interfaces and abstract classes are excluded by this rule.

implemented as stand-alone framework and not as plugin or extension for an Integrated Development Environment (IDE) even if the IDE would have been programming language independent.

3.3 Extensibility

MUSE has to be designed as framework to support adding new design best practices when necessary. This feature is required because new design best practices can arise according to design discussions, and it allows quality experts to add new measurable design aspects according to their needs. The implementation of this plugin mechanism must be usable without touching core elements of the framework. Within the framework, design best practices are depicted by so called rules, i.e., a rule is the implementation of a single design best practice.

3.4 Integration into Dashboards

The measurement output of MUSE, basically a list of rule violations, has to be represented in a generic format. Hence, it can be displayed by different dashboards of the source code analysis environment SonarQube. SonarQube is chosen as target environment since it is open-source and has some additional features for quality related tasks. Next to the centralized reporting mechanism provided by this source code analysis environment, it can be easily integrated in the build process of a software product. This feature is crucial for the application of MUSE because it drives the idea of a continuous design quality assessment and supports software engineers in dealing with improvements of design flaws.

3.5 Configurability

MUSE must support configurability on different levels. First, rules may require settings that cannot be hard coded within the scripts. Thus, it is necessary to offer an option that allows configuring rules. Second, projects typically have different project characteristics that require adjusting MUSE according to these needs. In other words, the framework must be configurable on project level as well.

4 MUSE Architecture and Implementation

This section shows the architecture and implementation details that address the requirements mentioned above.

4.1 Architecture

Figure 1 provides a high level overview of MUSE from a usage perspective. Starting at bottom left, it shows that source files are the input of the commercial tool *Understand*[7]. After setting up an *Understand* project and specifying the source files of a project, including its dependencies to external libraries, *Understand* extracts meta-information from the source code files and stores this information in a database. The meta-information comprises, e.g., type definitions, class relations, or method definitions. For using *Understand*, a graphical user interface as well as a command line interface is provided. Latter facilitates the integration of *Understand* in an automated build environment when a project team plans to use MUSE as part of their software development cycle. After successfully extracting the required information, MUSE uses the Perl API of *Understand* to query the information that is required to execute the implemented design best practices. Besides, size entities of the project can be calculated which are used for normalizing measuring results. Finally, MUSE generates two output files that represent the rule violations and size characteristics of the project.

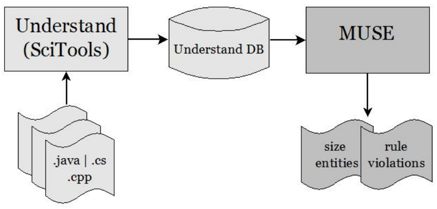


Figure 1 – High level overview of MUSE execution

MUSE represents the novel part of the tool chain that is used to measure object-oriented design. Figure 2 provides a deeper insight and shows the layered architecture. Although Figure 2 gives the impression that the elements of the gray rectangles represent directories. By logically separating the Perl scripts into directories, the framework becomes more structured and easier to understand.

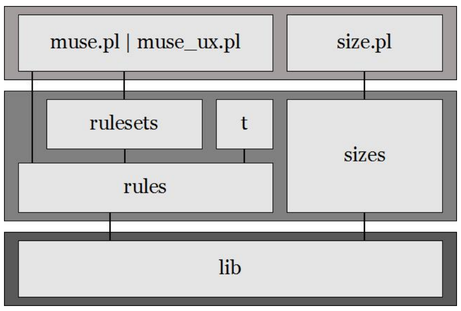


Figure 2 – The layered architecture of MUSE

At the bottom end of the architecture of MUSE there is the library directory *lib*. This directory contains Perl scripts that implement basic utility functions used by layers above. For instance, these functions handle the communication with the *Understand* database, write the result files and provide general sub routines used by many rules, e.g., a sub routine for retrieving all public classes of a project. Actually, this *lib* directory comprises 20 scripts.

[7]https://scitools.com/

On the next higher layer, the core functionality of MUSE is implemented. This layer contains the rules for measuring object-oriented design as well as rule sets. Rule sets allow the flexible and simple combination of individual rules to themes, e.g., all rules related to the theme abstraction. Each rule is implemented as a self-standing Perl module and represents a single file in the *rules* directory. Since coding practices of Perl define that a module name must match with the file name, each file is named after the rule name. This facilitates the handling of rules; especially, when searching a particular one. All in all, the current version of MUSE consists of 67 rules for checking object-oriented design best practices.

In parallel to the *rules* directory, the *t* directory contains at least one unit test for each rule. These unit tests verify the correct implementation of the rules and ensure that violations are detected. Moreover, they are used to avoid side effects caused by changes to rule modules. Next to these unit tests, the *t* directory contains additional basic unit tests to verify the correctness of the library modules (contained in the *lib* directory) as well as the communication to the *Understand* database.

An additional directory with Perl scripts is the *sizes* directory. The *sizes* directory contains scripts used for calculating size entities of projects written in Java, C++ and C#. The last element in the core layer is the *rulesets* directory. In this directory proven combinations of rules can be specified as a rule set. This facilitates the application of MUSE.

Lastly, the topmost layer of the entire layer stack contains the scripts *muse.pl* and *sizes.pl*. The main purpose of these scripts is the management of the rule execution including the handling of configuration and result files. Similar to the *muse.pl* script, the *size.pl* script manages the execution of the scripts for calculating the size entities of projects. Therefore, it uses the scripts in the *sizes* directory as depicted in Figure 2. All scripts can be called by the command line and they support different command line options for a customized execution.

4.2 Implementation

Under the hood, MUSE encompasses implementations that are necessary to address the requirements defined in Section 3. Some of them are discussed next in order to better illustrate the functioning of the framework.

Anatomy of a rule implementation: As previously mentioned, *muse.pl* triggers the execution of a rule check, e.g., the rule *AvoidPublicInstanceVariables*. In other words, the script loads the particular rule module and calls its main function with handing over the database connection and the configuration file. Afterwards, the rule module is working autonomously and first checks whether it supports the programming language of the project stored in the *Understand* database. In case the programming language is supported, it continues with running the rule check. Therefore, the *Understand* Perl API becomes important because it is required to query the meta-information that is needed to conduct the rule check. For instance, the code snippet shown in Listing 1 - taken from the rule *AvoidPublicInstanceVariables* - shows the query to gather all public members that do not have a static or final visibility.

Depending on the intention of the rule and the design aspect that need to be measured, more advanced and cascaded queries are required. This may result in complex rule structures which cover sophisticated design knowledge. Nevertheless, each rule implementation comes to a conclusion whether the rule is violated or not. If a violation has occurred, a finding will be written to the result file. After checking all entities for their rule compliance, the rule implementation returns back to the *muse.pl* script, which continues to trigger the next rule check.

In case two or more rules report a violation for the same element, e.g., class, interface, or method, all identified problems are reported. Currently, there is no aggregation function available to emphasize that this element might need special attention because of multiple rule violations.

Parallel execution of rules: Since rules are self-standing Perl modules they can be executed in parallel supported by MUSE. This functionality is available in Unix environments only because Windows follows a different process management model that prohibits the instantiation of additional child processes. This is the reason why *muse.pl* comes with two versions; one that works on both Unix and Windows and one that can be used on Unix exclusively. The number of processes that are used for parallel computing is defined by a command line option of *muse_ux.pl*, but limited to the total number of processors that are available on the CPU. This feature decreases the execution time of MUSE.

```
sub checkInstanceVariables {
        my $db = shift;
        my $checkedEntities = 'Public Variable Member ~Static ~Final';

        foreach my $ent ($db->ents($checkedEntities)) {
                print $ent->name()."\n";
                // rule logic
        }
}
```

Listing 1 – Structure of a typical rule implementation

Bundling of rules: Next to the parallel execution of modules, one module can measure two or more rules. This occurs when two rules are very similar in their core logic, but one is just more specific than the other or there is just a slight difference between them. For example, the rules *AvoidPublicInstanceVariables* and *AvoidProtectedInstanceVariables* share the same core logic but differ in their query definition. In order to reduce execution time and to avoid multiple executions of the same implementation, which mostly includes complex algorithms, similar rules are measured at the same time, but only specified ones are reported to the result file.

Dynamic loading: The execution of MUSE is context and project dependent since software design analysts may be interested in different design issues. Besides, it is unlikely that an analyst wants to get informed about all rule violations. Consequently, MUSE offers the functionality to select desired rules and to load those libraries - Perl modules - that are required therefor. The feature of dynamically loading the necessary libraries keeps MUSE lightweight and reduces the amount of allocated memory.

Muse configuration file: A muse configuration file can be defined for a project on which MUSE will be executed. This file stores the entire configuration that is project-specific and required by MUSE.

Test utilities: As briefly mentioned in the previous section, the MUSE core includes a test set. Each implemented rule is accompanied by a unit test and some more unit tests are available for the base libraries. Besides ensuring the functional correctness of the implemented rules, the tests guarantee that the external dependencies to the *Understand* API remain valid.

Restricting the analysis: Running MUSE on a huge project can be tedious and might produce too many findings in which design analysts are not interested. Consequently, MUSE provides the functionality to narrow down the execution to package and component level. This allows to targetedly conduct the object-oriented design measurement and to gather the portion of information that is most valuable for analyzing the design.

5 Comparing MUSE with metric-based Approaches

There exist a number of metric-based approaches for measuring the object-oriented design of software. According to Olague et al. [OEGQ07] and also according to our understanding the three most important metrics suites are the CK metrics [CK94], MOOD [BeAM96] and QMOOD [BD02]. Importance is defined here by the number of references to the respective papers as well as their usage in studies and experiments. Olague et al. [OEGQ07] compare the above mentioned metric suites and come to the conclusion that the CK metric suite and QMOOD are the best metric-based approaches to be used for the measurement of object-oriented design quality.

As MUSE does not only try to give measures for object-oriented design in general but also for object-oriented design properties such as abstraction or encapsulation, we choose QMOOD for our study because it provides aggregation and evaluation functions for a number of object-oriented design properties. MUSE provides rules for the design properties *abstraction*, *coupling*, *encapsulation*, and *inheritance* next to rules listed in the appendix.

5.1 System for Study

For comparing the MUSE approach with QMOOD, six major releases of the open source project jEdit have been selected as shown in Table 5. This selection is based on the criteria that it is a mid-sized project with approximately 112 KLLOC and it is implemented in Java. Additionally, we choose jEdit as it is an open source project with a long tradition, with a large number of installations and with an assumed interest in good (design) quality as the size and therefore functionality of jEdit considerably evolved over time, e.g., from ~65 KLOC in release 4.1 to ~112 KLLOC in the current release 5.3. Furthermore, the domain of (text) editors is well suited for exploring object-oriented design aspects since editors still are valuable sources for discussing object-oriented design.

Table 5 – Overview of jEdit releases

Release	**# of Files**	**# of Classes**	**# of Logical Lines of Code**	**Release Date**
4.1	300	567	65,861	2003-05-24
4.2	355	701	81,754	2004-12-01
4.3	495	958	101,940	2009-12-23
4.5	528	1,186	104,410	2012-01-31
5.0	550	1,244	109,942	2012-11-22
5.3	553	1,263	112,474	2016-03-21

5.2 Measuring object-oriented Design with QMOOD

After selecting the project for study, we began to measure the object-oriented design of each release with both the QMOOD and MUSE approach. However, before explaining the way of measuring object-oriented design based on the concept of QMOOD, an overview of the hierarchal quality model is given. Afterwards, the technical implementation of metrics is explained.

5.2.1 Overview of the QMOOD Quality Model

The quality model of QMOOD is structured into four levels as shown in Figure 3. On the first level, the model consists of six design quality attributes that are derived from the ISO 9126 standard [iso01] and shown in Table 6. As mentioned by Bansiya and Davis [BD02], these attributes are not directly observable and there is no approach for their operationalization. Consequently, the authors introduced an additional layer of object-oriented design properties. In total, eleven design properties are defined that express object-oriented design characteristics such as the relationships of objects; the visibility of attributes, methods and classes; or the complexity and inheritance tree of classes. These eleven properties are depicted in the formulas of Table 6 and each is measured by one metric.

We concentrate on the design properties *abstraction*, *coupling*, *encapsulation*, and *inheritance* (see Table 1 - 4), as MUSE provides a substantially different view on how to measure these design properties. For the QMOOD properties *design size*, *hierarchies*, and *messaging* the definition of these properties is narrow and hardly allows a different selection of measures as those proposed by QMOOD. Furthermore, for the QMOOD design properties *cohesion*, *complexity*, *composition*, and *polymorphism* we agree with the measures suggested by QMOOD and have no further suggestions here. Nevertheless, MUSE itself covers additional design properties like *design documentation* and *cyclic references* that are not covered at all by QMOOD (see appendix). As we choose to validate MUSE with established design quality models, we can only directly compare a subset of MUSE with QMOOD. In future work we will try to validate MUSE in its entirety, but this needs other scientific methods, as according to our analysis, no other comparable design quality model known from the literature can be used for a direct and reasonable comparison.

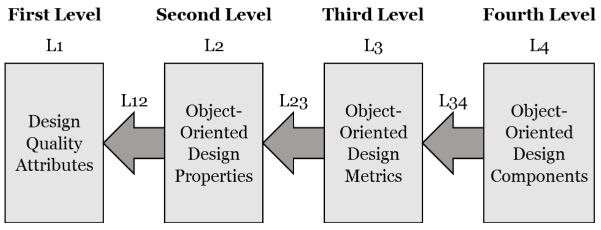

First Level
L1
Design Quality Attributes
Second Level
L2
Object-Oriented Design Properties
Third Level
L3
Object-Oriented Design Metrics
Fourth Level
L4
Object-Oriented Design Components
L12
L23
L34

Figure 3 – Four levels of the QMOOD quality model

For linking the first and the second level of the QMOOD quality model, computation formulas for quality attributes are defined that weight and combine design quality properties to one single quality attribute as shown in Table 6. We provide this overview table for sake of completeness; nevertheless, it does not play any role for the validation, as the validation concentrate on the level of object-oriented design properties for comparing QMOOD and MUSE.

Table 6 – QMOOD quality attributes

Quality Attributes	**Calculation**
Effectiveness	0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism
Extendibility	0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism
Flexibility	0.25 * Encapsulation - 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism
Functionality	0.12 * Cohesion + 0.22 * Polymorphism + 0.22 * Messaging + 0.22 * Design Size + 0.22 * Hierarchies
Reusability	-0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * Design Size
Understandability	-0.33 * Abstraction + 0.33 * Encapsulation - 0.33 * Coupling + 0.33 * Cohesion - 0.33 * Polymorphism - 0.33 * Complexity - 0.33 * Design Size

At this point it is important to mention that the quality model of QMOOD was designed for the programming language C++. This means that language specifics such as multiple inheritance or the virtual visibility of methods are influencing the metric definitions. For the sake of completeness, the fourth and last layer of QMOOD represents the object-oriented design components that are objects, classes and the relationships between these elements. Those measures that are relevant for us in this validation, i.e., for the design properties abstraction, coupling, encapsulation, and inheritance, are described in Table 7. All other QMOOD metrics are intentionally not described or discussed in order to keep the focus on the selected design properties and their design metrics.

Table 7 – QMOOD design properties and metric defintions

Design Property (Design Metric)	**Definition**
Abstraction (ANA)	A measure to show the degree of generalization-specialization within a design. **ANA** (Average Number of Ancestors) represents the average of super classes from which a class inherits behavior. It considers both direct and indirect inheritance.
Cohesion (CAM)	This property shows the togetherness of methods and attributes in a class.
Complexity (NOM)	This property deals with the understandability and comprehensibility of the internal and external structure of classes.
Composition (MOA)	A measure to express the “part-of", “has", or “consists-of" relationships between classes of the design.
Coupling (DCC)	This property shows the closeness of objects to each other. An object is close to another object, when it offers functionality that is required by the other object to function correctly. **DCC** (Direct Class Coupling) represents the average of classes that are referenced by one class. A class is referenced by another class when it is used as declaration for a class member declaration or method parameter. In this case, Java standard libraries are taken into consideration too. When the number of class declarations or method parameter declarations is computed, an interface is considered as class.

Design Size (DSC)	The size of the design expressed by the number of classes.
Encapsulation (DAM)	The enclosing of data and behavior within a single construct expressed by the visibility characteristics of variables. **DAM** (Data Access Metric) is the ratio of all variables with private, protected and package visibility to the total number of variables.
Hierarchies (NOH)	The number of non-inherited classes that inherit behavior to sub classes. In other words, the number of classes that act as root within an inheritance tree.
Inheritance (MFA)	A measure to express the "is-a" relationship between classes of the design. **MFA** (Measure of Functional Abstraction) is the ratio of the number of methods inherited by a class to the total number of methods available (inherited or defined) to that class.
Messaging (CIS)	A measure to shows the number of services provided by a class. A service is defined as public method that can be accessed from external.
Polymorphism (NOP)	The ability to substitute an object with another object at run-time.

5.2.2 Technical Implementation of QMOOD Metrics

In [GRI$^+$14] the authors selected QMOOD for the validation of their research questions and also faced the challenge of finding an appropriate way for measuring these metrics. To calculate the metric values, they used the tool *Understand* and implemented the metrics by querying the Phyton API, which is available next to the Perl API. Since we are very familiar with *Understand* due to its essential role in the MUSE framework, we wrote our own Perl scripts that implement the eleven metrics. Despite a solid knowledge in developing the scripts, there is still the problem that the QMOOD quality model was originally designed for C++ and does not consider Java-specific requirements [GJ14]. Besides, the definitions of some metrics are vague and require an own interpretation [OC06]. Consequently, we used the work of O'Keeffe and Cinnéide [OC06] as basis and defined assumptions for measuring Java source code. Specifically, the language feature of interfaces was determining some assumptions because it is not considered in QMOOD. As a result, Table 7 describes the assumptions we made. For those metrics needed for the validation, their name is highlighted in bold and a more detailed assumption is provided.

5.3 Measuring object-oriented Design with MUSE

In order to compare the MUSE approach with the metric-based approach of QMOOD, it is necessary to specify the design best practices that violate one of the four design properties. Therefore, we selected rules from the rule pool of MUSE based on a previously conducted and QMOOD independent grouping of them. As already mentioned, the design best practices were deductively derived from the literature of design principles in software engineering and the six researches assigned them to design properties (as well as design principles which are on a lower aggregation level). Table 1 - 4 describe the design best practices derived for this validation. For the sake of completeness, only rules for Java source code are chosen. As already mentioned, the identification of relevant design best practices was done via identification of best practices from literature and the selection and specification of the rules was carried out by at least four researchers.

6 Results and Discussion

This section presents and discusses the results from calculating the QMOOD metrics and executing the MUSE framework for the four selected design properties. The discussion part of this section compares QMOOD with MUSE on each particular design property. Therefore, it must be considered that both approaches focus on operationalizing the same design property but with a different perception; on the one hand with a single metric, on the other hand with a elaborated view based on multiple design best practices.

Table 8 shows the measuring result of the four QMOOD metrics used in this validation study. Since this metrics are mapped to design properties according to the QMOOD model, the validation concentrates on level 2 - the design property level - of Figure 3. Level 1 could not be used for comparision because MUSE does not provide an aggregation function that would combine multiple design properties to quality attributes as QMOOD does.

Besides, the other design properties are not presented in Table 8 to keep the focus on abstraction, coupling, encapsulation and inheritance. According to the last column in Table 8 there are trends indicating the change of design properties over time. While there is a mild negative trend for abstraction and inheritance, the QMOOD measures show a slight increase for coupling and encapsulation. Nevertheless, this is just an aggregated quantitative value that gives no guidance for improvement at all, i.e., which specific actions to approach.

Table 8 – Result of QMOOD metric execution

Design Properties	**4.1**	**4.2**	**4.3**	**4.5**	**5.0**	**5.3**	**% Change 4.1 - 5.3**
Abstraction (ANA)	2.43	2.34	2.28	2.27	2.28	2.29	-6%
Coupling (DCC)	5.24	5.40	5.47	5.44	5.46	5.55	+6%
Encapsulation (DAM)	0.82	0.80	0.82	0.84	0.84	0.84	+2%
Inheritance (MFA)	0.77	0.76	0.76	0.76	0.76	0.75	-2%

6.1 Abstraction

By calculating the average number of ancestors, QMOOD tries to estimate the degree of generalization and specialization applied within the software design. However, by just using ANA for expressing this design aspect, other abstraction related issues are ignored. Table 9 lists the results provided by MUSE for the design property abstraction. Here we can, e.g., see that the number of classes with similar names on the same abstraction level - see *AvoidSimilarNamesOnSameAbstractionLevels* - has risen from 18 to 54 and drop to 24 rule violations for the current version. This can indicate a design problem because classes with the same word stem may implement similar behavior and the design fails in applying appropriate generalizations.

In the context of the prior mentioned rule *AvoidSimilarNamesOnSameAbstraction-Level* MUSE identifies the problem, but cannot propose a solution for fixing it. The reason is that the software engineer or architect has to verify whether the concept of abstraction is broken or just the naming of the class is misleading. Nevertheless, MUSE points at least to the questionable classes or interfaces. In contrast, there are other rules for which MUSE can offer a recommendation for improvements. An example is the rule *AvoidPartiallyUsedMethodInterfaces* that checks into how many parts a class could be split up by identifying sets of public methods that are jointly

Table 9 – Violations of abstraction realted MUSE rules

	4.1	**4.2**	**4.3**	**4.5**	**5.0**	**5.3**	**% Change 4.1 - 5.3**
AvoidPartiallyUsed-MethodInterfaces	78	87	109	20	20	18	-77%
	1.18	1.06	1.07	0.19	0.18	0.16	-86%
AvoidSimilarAbstraction	2	14	12	10	24	32	+1500%
	0.03	0.17	0.12	0.10	0.22	0.28	+837%
AvoidSimilarNamesOn-DifferentAbstractionL.	8	8	12	6	6	6	-25%
	0.12	0.10	0.12	0.06	0.05	0.05	-56%
AvoidSimilarNamesOn-SameAbstractionLevel	18	54	54	20	22	24	+33%
	0.27	0.66	0.53	0.19	0.20	0.21	-22%
ProvideInterfaceForClass	34	50	62	64	67	66	+94%
	0.52	0.61	0.61	0.61	0.61	0.59	+14%
UseInterfaceIfPossible	446	445	569	655	665	686	+54%
	6.77	5.44	5.58	6.27	6.05	6.10	-10%
Total	586	658	818	775	804	832	+42%
	8.90	8.05	8.02	7.42	7.31	7.40	-17%

used by other client classes. These sets of methods are then presented to the design analyst who can split up the overloaded class according to the suggestions.

From a quantitative perspective, Table 9 shows a decrease in quality for the two design best practices *AvoidSimilarAbstraction* and *ProvideInterfaceForClass* on the one hand. On the other hand, the definition of methods for classes measured by *AvoidPartiallyUsedMethodInterfaces* was considerably enhanced. In total and without specific weighting of the design best practices, we see an overall increase in quality (with respect to abstraction) of 17%. Thus, the positive trend indicated by QMOOD can be confirmed by the MUSE rules but with much more detailed information for improvements.

6.2 Coupling

The QMOOD model measures the design property of coupling by counting references to other classes. A class is referenced by another class when it is used as declaration for a member or method parameter. As shown in Table 8, this metric slightly increased from 5.24 to 5.55 over the six releases. In fact, we specified no MUSE rule that exactly implements the same concept of DCC, however, the rules *AvoidMultipleImplementationInstantiations* and *DontInstantiateImplementationsInClient* do have a similar intent. Thus, *AvoidMultipleImpelementationInstantiations* is based on the idea that classes should not instantiate implementation classes (except for factories). Therefore, the rule checks that classes are created from other classes as restricted as possible and the rule fires a violation when the number of instantiations exceeds a defined threshold. Compared to DCC, the rule *DontInstantiateImplementationsInClient* does only count instantiations of those classes that are not defined within the same package. This concept is based on the idea that it is accepted to couple classes from the same package, but it is an issue when class coupling crosses package boundaries. In order to fix this design issue, the MUSE rule recommends to specify an interface for classes which are defined in another package.

As briefly highlighted in the previous discussion, MUSE rules also consider the use of the package construct, which is an important technique for structuring software

Table 10 – Violations of coupling realted MUSE rules

	4.1	**4.2**	**4.3**	**4.5**	**5.0**	**5.3**	**% Change 4.1 - 5.3**
AvoidMultipleImplemen-	42	56	69	69	70	70	+67%
tationInstantiations	0.64	0.68	0.68	0.66	0.64	0.62	-2%
AvoidStronglyCoupled-	13	15	21	23	25	27	+108%
PackageImplementations	0.20	0.18	0.21	0.22	0.23	0.24	+22%
CheckExistenceImple-	51	19	18	18	18	18	-65%
mentationClassesAsString	0.77	0.23	0.18	0.17	0.16	0.16	-79%
DontInstantiateImple-	242	283	337	347	365	374	+55%
mentationsInClient	3.67	3.46	3.31	3.32	3.32	3.33	-10%
PackageShouldUseMore-	7	8	13	13	15	15	+114%
StablePackages	0.11	0.10	0.13	0.12	0.14	0.13	+25%
Total	433	577	764	758	812	827	+91%
	6.57	7.06	7.49	7.26	7.39	7.35	+12%

design. More specifically, the rules *AvoidStronglyCoupledPackageImplementation* and *PackageShouldUseMoreStablePackages* check the appropriate application of packages and fire a violation when packages are relying on each other or one package refers to another package that is less stable.

The fifth MUSE rule for measuring coupling is *CheckExistenceImplementation-ClassesAsString*. This rule may indicate the most problematic way of coupling classes because it checks for each string value whether it matches to a fully qualified type name, which could be used for dynamic instantiations of this type. While the dynamic instantiation takes place at run-time and not at compile-time, this issue can cause unexpected side effects. As shown in Table 10, the number of violations regarding this bad design practice decreased from 51 to 18 what makes the current version of jEdit less vulnerable to unverified instantiation of classes.

All in all, this comparison of applying QMOOD and MUSE for measuring coupling shows that MUSE follows a more comprehensive approach and tries to identify additional aspects that threaten this design property. Especially, the consideration of packages is important when measuring design. The reason therefore is that classes within the same package mostly share communalities resulting in a strong coupling. However, by just providing a metric value as done by QMOOD approach, the engineer or architect remains clueless whether the value expresses class coupling within packages (that is more accepted) or class coupling across package boundaries. MUSE provides multiple indicators that may show the problem of this design flaw.

From a quantitative perspective, a deterioration of package stability can be identified as shown by the increase of rule violations for *AvoidStronglyCoupledPackageImplementation* and *PackageShouldUseMoreStablePackages*. In contrast, a decrease of rule violations can be seen for *AvoidMultipleImplementationInstantiations*, *DontInstantiateImplementationsInClient*, and *CheckExistenceImplementationClassesAsString*. Although the relative decrease of rule violations for *CheckExistenceImplementation-ClassesAsString* is high with 79%, this does not signficantly influence the total quality assessment of coupling due to the low absolute number of violations. In other words, the quality decrease of 12% results from equal weights for all five design best practices. Interestingly, QMOOD also shows a decrease in the quality of coupling what matches the MUSE assessment.

6.3 Encapsulation

According to Table 8, the value of the DAM metric to measure encapsulation has slightly changed over the six releases although the size of the project grew by 47 KLLOC. Thus, this discussion tries to identify whether MUSE can reveal a difference between the six releases or can show that design practices of engineers did not change.

Table 11 gives an overview of the MUSE findings in this regard. Obviously, The development team did a good job in checking parameters of setters and in using interfaces as the rule violations for *AvoidSettersForHeavilyUsedAttributes*, and *UseInterfaceAsReturnType* decreased over time. However, there is quite an increase of public and protected instance variables (*AvoidPublicInstanceVariables*, *AvoidProtectedInstanceVariables*) as well as of public static variables (*AvoidPublicStaticVariables*). Merely taking this into account, these findings already justify the overall normalized decrease in quality of 12%. Since public, protected and static instance variables can be seen as the most contradicted aspect of encapsulation, the relative increase of 12% can be viewed as a considerable decrease in quality - an aspect that is not reflected by QMOOD, where the overall quality remains unchanged.

Table 11 – Violations of encapsulation realted MUSE rules

	4.1	**4.2**	**4.3**	**4.5**	**5.0**	**5.3**	**% Change 4.1 - 5.3**
AvoidExcessiveUseOf-Getters	9	7	9	12	16	17	+89%
	0.14	0.09	0.09	0.11	0.15	0.15	+11%
AvoidExcessiveUseOf-Setters	4	4	10	9	12	13	+225%
	0.06	0.05	0.10	0.09	0.11	0.12	+90%
AvoidPublicStatic-Variables	15	49	47	43	43	43	+187%
	0.23	0.60	0.46	0.41	0.39	0.38	+68%
AvoidProtectedInstance-Variables	32	52	109	114	116	115	+259%
	0.49	0.64	1.07	1.09	1.06	1.02	+110%
AvoidPublicInstance-Variables	90	159	191	171	175	176	+96%
	1.37	1.94	1.87	1.64	1.59	1.56	+15%
AvoidSettersForHeavily-UsedAttributes	38	34	34	16	16	16	-58%
	0.58	0.42	0.33	0.15	0.15	0.14	-75%
CheckParametersOfSetters	57	56	74	95	131	132	+132%
	0.87	0.68	0.73	0.91	1.19	1.17	+36%
UseInterfaceAsReturn-Type	188	216	290	298	303	315	+68%
	2.85	2.64	2.84	2.85	2.76	2.80	-2%
Total	433	577	764	758	812	827	+91%
	6.57	7.06	7.49	7.26	7.39	7.35	+12%

Comparing QMOOD with MUSE in a quantitativ manner is difficult because only QMOOD specifies a desirable target value for DAM. Since DAM ranges from 0 to 1, a value close to 1 is an indicator for good encapsulation [BD02]. In contrast, the number of findings that is calculated by MUSE stands freely in the room when no context for interpretation is provided. Thus, it is unclear if 433 rule violations for version 4.1 of jEdit are an indicator for bad design quality or other similar projects do even worse in encapsulation.

When discussing the number of violations per classes, it can be pointed out that an average class in version 4.2 had more violations compared to the five other versions. The same effect can be seen by the interpretation of the DAM metric since it decreased in version 4.2 resulting in a bad design according to [BD02]. Nevertheless, the normalization by using KLLOC cannot indicate any quality switch between versions, but rather shows that the number of violations increased by approximately 0.5 violations per KLLOC.

6.4 Inheritance

At several points in this paper, it has already been mentioned that metric values indicate a design problem, but let the engineer or architect in doubt about the original issue. For illustrating the benefit of locating the source of a design flaw, the following discussion uses the MFA metric and compares it with the MUSE approach for measuring inheritance related design aspects. The idea behind the MFA metric is to express the is-a relationship between classes of the design. Therefore, it calculates the ratio of the number of methods inherited by a class to the total number of methods available (inherited or defined) to that class. Finally, all ratios are accumulated to project level and to a single number, e.g., 0.77 for jEdit 4.1 as shown in Table 8.

Table 12 – Violations of inheritance realted MUSE rules

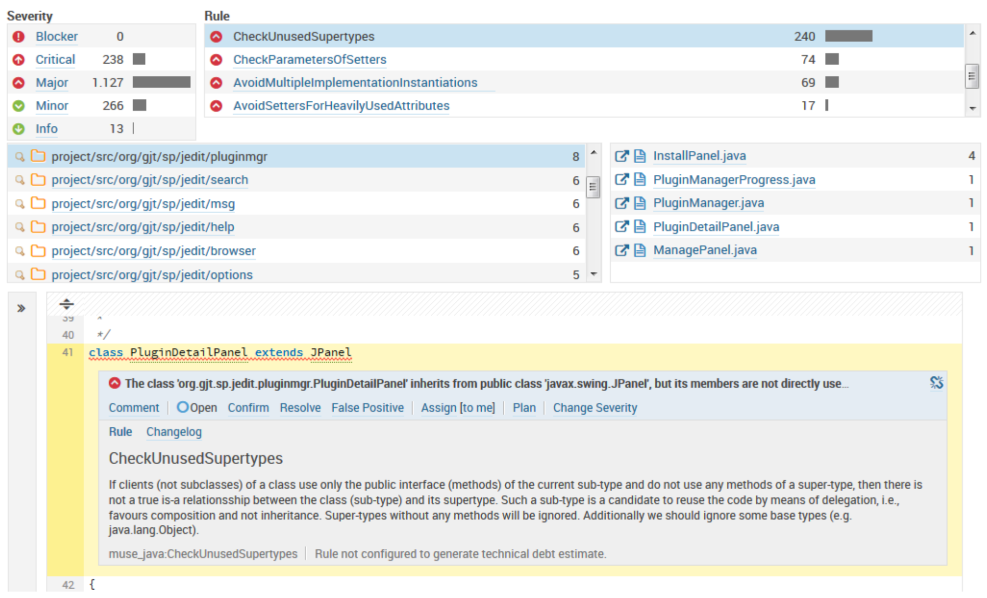
	4.1	**4.2**	**4.3**	**4.5**	**5.0**	**5.3**	**% Change 4.1 - 5.3**
CheckUnusedSupertypes	34	42	55	60	61	64	+88%
	0.52	0.51	0.54	0.57	0.55	0.57	+10%
UseCompositionNot-Inheritance	38	42	53	56	57	55	+45%
	0.58	0.51	0.52	0.54	0.52	0.49	-15%
Total	72	84	108	116	118	119	+65%
	1.09	1.03	1.06	1.11	1.07	1.06	-3%

In contrast to QMOOD, MUSE knows the classes that fail the inheritance concept. For instance and shown in Table 12, jEdit version 4.3 contains 240 situations where an instantiation of a class uses only the public interface (methods) of the current subtype and do not use any methods of a supertype. This is an indicator that there is not a true is-a relationship between the class (subtype) and its base class (supertype). Such a subtype is a candidate to reuse the code by means of delegation. However, for refactoring this class and excluding it from the inheritance tree, the class must be identified. MUSE supports this task because the results can be uploaded to a SonarQube instance. Next to some useful features for managing code quality, our MUSE plugin for SonarQube provides the functionality to drill down into the source code and to locate the violations of, e.g., the rule *CheckUnusedSupertypes*.

A screenshot of this drill down mechanism is shown in Figure 4. There it can be seen that also a recommendation for the improvement and a description of the rule are displayed. All in all, this perfectly supports refactoring because the software engineer can understand the design flaw and start fixing the rule violation. This feature is of course not only available for the discussed rule *CheckUnusedSupertypes* but for all MUSE rules.

7 Threats to Validity

This section discusses potential threats to validity in context of the QMOOD and MUSE comparison. Specifically, it focuses on threats to internal, external, and construct validity [WRH$^+$12].



Severity	
Blocker	0
Critical	238
Major	1.127
Minor	266
Info	13

Rule	
CheckUnusedSupertypes	240
CheckParametersOfSetters	74
AvoidMultipleImplementationInstantiations	69
AvoidSettersForHeavilyUsedAttributes	17

project/src/org/gjt/sp/jedit/pluginmgr	8
project/src/org/gjt/sp/jedit/search	6
project/src/org/gjt/sp/jedit/msg	6
project/src/org/gjt/sp/jedit/help	6
project/src/org/gjt/sp/jedit/browser	6
project/src/org/gjt/sp/jedit/options	5

InstallPanel.java	4
PluginManagerProgress.java	1
PluginManager.java	1
PluginDetailPanel.java	1
ManagePanel.java	1

```
 */
class PluginDetailPanel extends JPanel
```

The class 'org.gjt.sp.jedit.pluginmgr.PluginDetailPanel' inherits from public class 'javax.swing.JPanel', but its members are not directly use...

Comment | Open Confirm Resolve False Positive | Assign [to me] | Plan | Change Severity
Rule Changelog

CheckUnusedSupertypes

If clients (not subclasses) of a class use only the public interface (methods) of the current sub-type and do not use any methods of a super-type, then there is not a true is-a relationsship between the class (sub-type) and its supertype. Such a sub-type is a candidate to reuse the code by means of delegation, i.e., favours composition and not inheritance. Super-types without any methods will be ignored. Additionally we should ignore some base types (e.g. java.lang.Object).

muse_java:CheckUnusedSupertypes | Rule not configured to generate technical debt estimate.

```
{
```

Figure 4 – MUSE plugin for SonarQube

7.1 Internal Validity

Threats to internal validity concern our selection of the project, tools, and the analysis method that may introduce confounding variables. Thus, the choice of the system of study poses a threat. In this paper we chose jEdit with six selected releases as object for investigation. Despite a significant increase of LLOC from release 4.1 to 5.3, no design improvements could be observed that would have influenced the QMOOD metrics as shown in Table 8. Except of DSC and NOH, which naturally rise with additional LLOC, the other metrics do not show a switch in design decisions. Consequently, the comparison of QMOOD with MUSE would have benefited from investigating releases with more significant design improvements. The selection of the QMOOD quality model and the tool *Understand* is discussed as threat to construct validity although they represent a threat to internal validity too.

7.2 External Validity

Threats to external validity affect the possibility of generalizing the results. Regarding this aspect it must be noticed that the validation is comparing only six versions of one project. This project has its specific architecture and is developed by a team which may have its specific design rules. Thus, we cannot - and do not want to - claim that this proposed approach of measuring design best practices fits for every project. In order to generalize results, further work must consider different systems, with different teams, sizes, and application domains. However, we know that MUSE supports the assessment of various projects since it can be customized and extended based on its framework character.

7.3 Construct Validity

Construct validity refers to the meaningfulness of measurements and in the context of this paper the measuring of the quality properties. The choice of the QMOOD model rather than another assessment approach represents a threat to this validity concern. The reason is that it has not been evaluated whether the metrics of QMOOD are appropriate for the measurement of the quality properties. Instead, they are used in this paper as they are defined without questioning their intent. Nevertheless, [OEGQ07] consider QMOOD to be a good choice for measuring object-oriented design as pointed out earlier.

Another threat is that the QMOOD quality model was originally designed for the C++ programming language; however, we used a project written in Java. Thus we had to interpret the defined metric definitions. As mentioned by [GRI$^+$14] and [OC06], the QMOOD model lacks unambiguous metric descriptions so that a threat stems from an inaccurate interpretation of the metrics.

Lastly, the choice of using the tool *Understand* for both the implementation of QMOOD metrics and the extraction of meta-information for MUSE poses an additional threat. Thus, the classification of objects such as classes, methods or attributes is complex and the query to get these objects may not reflect the set intended by the QMOOD metric definition. This may result in minor deviations from the original QMOOD metrics. In regard to the extraction of meta-information for MUSE, the rule implementations have to address language specific features independently. This is a tradeoff we have to accept for supporting multiple languages.

8 Conclusion and Future Work

There exist a number of metric-based approaches for measuring the object-oriented design quality of software as discussed in this paper. One major contribution of this paper is the availability of a framework for measuring object-oriented design based on best practices together with a simple yet powerful and comprehensible evaluation model. This approach leaves the established path of measuring design quality by means of metrics, but concentrates on good design best practices:

- *MUSE tooling.* MUSE currently implements 67 design best practices and is therefore a quite comprehensive suite for analyzing Java, C++, and C# projects. This is a contribution by itself, as we are not aware of any other object-oriented design measuring tool that provides such a variety of measurable rules. Most of the more comprehensive approaches either rely on metric metrics like the CK-metrics suite, the QMOOD model or similar approaches.

- *Design properties and rules.* The MUSE model does not only provide a set of configurable rules, but also associates the rules to design properties like abstraction, encapsulation, etc. Having this explicit relation at hand is a good starting point for identifying blind spots for measurement, but also for reasoning about the importance of individual rules. This identification of white spots is future work to be done.

Furthermore, we could show in our validation that using our design best practice based MUSE approach has major advantages compared to metric based approaches:

- The qualitative discussion of the individual results shows the advantage of rule-based approaches over metric-based approaches in general - at least when the purpose of measurement also has improvements or refactoring in mind. The analysis given in this paper shows that improvement actions can be better planned and executed, as we do exactly know which pieces of code have to be improved.
- We can also conclude that the quantitative results given by MUSE are comparable to the results given by QMOOD. For the design property encapsulation we could even show (see Section 6.3) that the quantitative results provided by QMOOD neglect a major increase of bad practices that corrupt the encapsulation design property. This is the case, although we know, that our current simplistic evaluation method could be exchanged by a more comprehensive evaluation model. In the context of code quality we already developed an elaborate evaluation model [WGH$^+$15] that will be applied to all rules provided by MUSE in the future.

Apart from future work discussed above, we will comprehensively pilot MUSE for a number of software projects in industry. First yet unsystematic feedback from industry indicates that MUSE is of value for the development organizations, as it helps to uncover design flaws directly from the source code.

As a second major task for the future we will investigate whether well-known object-oriented design principles (e.g., Open-Closed Principle, Single Responsibility Principle) can be measured by using our MUSE tool. We are currently working on an underlying design quality model based on design principles. The quality model in work follows the Quamoco approach [WGH$^+$15] and assigns design best practices to design principles. In order to understand the usefulness of each best practices as well as the coverage of design principles by their assigned best practices, we plan to validate this in a systematic and scientific manner.

References

[BD02] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002. `doi:10.1109/32.979986`.

[BeAM96] Fernando Brito e Abreu and Walcélio Melo. Evaluating the impact of object-oriented design on software quality. In *Proceedings of the 3rd International Software Metrics Symposium*, pages 90–99, 1996. `doi:10.1109/METRIC.1996.492446`.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994. `doi:10.1109/32.295895`.

[Doo11] John Dooley. Object-Oriented Design Principles. In *Software Development and Professional Practice*, pages 115–136. Apress, 2011. `doi:10.1007/978-1-4302-3802-7_10`.

[FBBO99] Martin Fowler, Kent Beck, John Brant, and William Opdyke. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, Reading, MA, 1999.

[GJ14] Puneet K. Goyal and Gamini Joshi. QMOOD metric sets to assess quality of Java program. In *International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT 2014)*, pages 520–533, 2014. doi:10.1109/ICICICT.2014.6781337.

[GRI$^+$14] Isaac Griffith, Derek Reimanis, Clemente Izurieta, Zadia Codabux, Ankita Deo, and Barry Williams. The Correspondence Between Software Quality Models and Technical Debt Estimation Approaches. In *Sixth International Workshop on Managing Technical Debt (MTD 2014)*, pages 19–26, 2014. doi:10.1109/MTD.2014.13.

[GVM15] George Ganea, Ioana Verebi, and Radu Marinescu. Continuous quality assessment with inCode. *Science of Computer Programming*, 2015. doi:10.1016/j.scico.2015.02.007.

[iso01] ISO/IEC 9126-1:2001 - Software engineering – Product quality – Part 1: Quality model. ISO/IEC 9126:2001, ISO/IEC, 2001. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749.

[Lis87] Barbara Liskov. Keynote Address - Data Abstraction and Hierarchy. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*, OOPSLA '87, pages 17–34, New York, NY, USA, 1987. doi:10.1145/62138.62141.

[Mar04] Radu Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 350–359. IEEE, 2004. doi:10.1109/ICSM.2004.1357820.

[MGDLM10] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010. doi:10.1109/TSE.2009.50.

[MGMD08] Naouel Moha, Yann-Gaël Guéhéneuc, Anne-Francoise Le Meur, and Laurence Duchien. A Domain Analysis to Specify Design Defects and Generate Detection Algorithms. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering*, number 4961 in Lecture Notes in Computer Science, pages 276–291. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-78743-3_20.

[MGV10] Radu Marinescu, George Ganea, and Ioana Verebi. InCode: Continuous Quality Assessment and Improvement. In *14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*, pages 274–275, 2010. doi:10.1109/CSMR.2010.44.

[MPS14] Alois Mayr, Reinhold Plösch, and Matthias Saft. Objective Safety Compliance Checks for Source Code. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*, ICSE Companion 2014, pages 115–124, New York, NY, USA, 2014. doi:10.1145/2591062.2591178.

[OC06] Mark O'Keeffe and Mel Ó. Cinnéide. Search-based software maintenance. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*, pages 249–260, 2006. doi:10.1109/CSMR.2006.49.

[OEGQ07] Hector M. Olague, Letha H. Etzkorn, Sampson Gholston, and Stephen Quattlebaum. Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes. *IEEE Transactions on Software Engineering*, 33(6):402–419, 2007. doi:10.1109/TSE.2007.1015.

[PGH$^+$07] Reinhold Plösch, Harald Gruber, Anja Hentschel, Christian Koerner, Gustav Pomberger, Stefan Schiffer, Matthias Saft, and Stephan Storck. The EMISQ Method - Expert Based Evaluation of Internal Software Quality. In *Proceedings of the 31st Annual IEEE Software Engineering Workshop (SEW 2007)*, pages 99–108, Columbia, USA, 2007. doi:10.1109/SEW.2007.71.

[Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996.

[WGH$^+$12] Stefan Wagner, Andreas Goeb, Lars Heinemann, Michael Kläs, Klaus Lochmann, Reinhold Plösch, Andreas Seidl, Jonathan Streit, and Adam Trendowicz. The Quamoco product quality modelling and assessment approach. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, pages 1133–1142, Zurich, 2012. doi:10.1109/ICSE.2012.6227106.

[WGH$^+$15] Stefan Wagner, Andreas Goeb, Lars Heinemann, Michael Kläs, Constanza Lampasona, Klaus Lochmann, Alois Mayr, Reinhold Plösch, Andreas Seidl, Jonathan Streit, and Adam Trendowicz. Operationalised product quality models and assessment: The Quamoco approach. *Information and Software Technology*, 62:101–123, 2015. doi:10.1016/j.infsof.2015.02.009.

[WRH$^+$12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-29044-2.

Appendix

Remaining Design Best Practices

Table 13 lists 30 additional rules that are not used in this work but can be applied for measuring the compliance of design best practices in Java, C#, and C++. Since these rules are valid for three different programming languages, their descriptions are defined in a generic manner. Nevertheless, this is not possible for all descriptions why some terms must be exchanged to match with the naming convention and language features of a particular programming language. More specifically, the term *interface* referes to a pure virtual abstract class in sense of C++ and the term *package* refers to namespace in sense of C++ or C#.

Table 13 – Design best practices for Java, C#, and C++

Name	**Description**
AbstractPackagesShouldNot-RelyOnOtherPackages	A package containing a high number of abstractions should not rely on other packages.
AlwaysUseInterfaces	Use interfaces for variable declarations, parameter definitions, or return types instead of public classes. System library classes are excluded if no interface is available.
AvoidAbstractClassesWith-OneExtension	An abstract class must have more than one subclass.
AvoidCommandsInQuery-Methods	A public method identified as query method should not change the state of the object and can only call query methods of the same class. A public method is identified as query method when its name starts with a defined prefix such as get, has, or is.
AvoidDiamondInheritance-StructuresInterfaces	An inheritance structure cannot result in a diamond structure in which a class implements an interface by two direct or indirect ways.
AvoidDuplicates	There should not be duplicates in the source code.
AvoidHighNumberOf-Subpackages	A package should not contain many direct sub-packages.
AvoidLongMethods	Long public methods should be avoided.
AvoidLongParameterLists	A method should not have long parameter lists.
AvoidManyTinyMethods	Short public methods - ignoring getter and setter methods - should be avoided.
AvoidMassiveCommentsIn-Code	A method should not have too many comment lines in code. The method documentation (API-documentation) as well as blank lines are not considered.
AvoidNonCohesive-Implementation	A class should not have methods sets that are not related to each other. Related means that they use/change the same set of instance variables or are connected by method calls.
AvoidNonCohesivePackage-Implementation	A package should be as cohesive as possible. Therefore, the number of parts, in which a package can be divided with respect to the cohesion of its classes, expresses the cohesiveness. A value greater than 1 indicates that the implementation of a package is not cohesive.
AvoidRepetitionOfPackage-NamesOnAPath	A package path of a package hierarchy should not contain repetitions, i.e., the relative name (non-fully qualified name) of a package should not be used twice.

AvoidReturningDataFrom-Commands	A method identified as command method should not return any kind of data regardless whether the data is related to the internal state of the object or not.
AvoidRuntimeType-Identification	Type checks of objects, i.e., use of instanceof operator in Java or the typed operator as well as the dynamic_cast operator in C++, should be avoided.
AvoidUndocumentedInterfaces	An interface must have an API-documentation for the interface declaration and each method signature.
AvoidUnimplementedInterfaces	An interface must be implemented by at least one class or used as super-type for another interface. An abstract class must be extended by at least one class.
AvoidUnusedClasses	A class must be used, i.e., it must be either instantiated, must have at least one subtype, or must provide public static members or methods used by a client.
AvoidUsingSubtypesIn-Supertypes	A class must not know anything about its direct and indirect subtypes.
CheckDegradedDecomposition-OfPackages	The decomposition of the package structure in a system should be balanced.
CheckSameTermsOnDifferent-PackageLevels	A relative package name (non- fully qualified name) should be used only on the same package hierarchy level across the system.
ConcretePackagesAreNotUsed-FromOtherPackages	A concrete package containing a high number of concrete classes needs to be used from other packages.
DocumentYourPackages	A package must have an API-documentation that can be either plain comments for C#,C++ or Doxygen comments for Java.
DocumentYourPublicClasses	A public class and public struct (i.e., classes and structs as well as typedefs in C++ header files) must have an API-documentation, i.e., comments above the declaration or the definition of the particular entity.
DocumentYourPublicMethods	A public method within a public class must have an API-documentation, i.e., comments above the declaration or the definition of the particular entity.
DontReturnUninvolvedData-FromCommands	A command method that changes the state of the object or class cannot return data that is not related to the change.
PackageCycle	This measure calculates all minimal cycles between packages whereas each cycle is evaluated by the number of packages involved.
UseAbstraction	A package should provide a sufficient number of abstract classes and interfaces - expressed by the ratio between abstract and concrete types.
UseInterfaceIfAvailable	Use the interface of a class for variable declarations, parameter definitions, or return types instead of the public class. If there exists more than one interface, one of the available interfaces should be taken.

Table 14 shows an additional rule that can be applied for Java.

Table 14 – Design best practices for Java

Name	**Description**
DontReturnCollectionsOrArrays	If the return type of a method is an array or an instance of a collection type, then the returned value should be immutable or cloned before.

Table 15 shows an additional rule that can be applied for C#.

Table 15 – Design best practices for C#

Name	**Description**
AvoidExcessiveUseOfProperties	The ratio between properties and the total number of non-const attributes should not exceed a certain threshold.

Table 16 lists 14 rules that can be applied for C++ only due to additional or different language features. Examples for such language features are templates, macros, or the const declaration.

Table 16 – Design best practices for C++

Name	**Description**
AvoidDiamondInheritance-StructuresImplementation	An inheritance structure cannot result in a diamond structure in which two base classes directly or indirectly extend a common base class.
AvoidExcessiveUseOfLarge-Macros	A large macro should be used as limited as possible.
AvoidFriends	There should not exist friend classes or friend methods that have access to the internal state of an object. However, friend functions to implemented operators are always allowed.
AvoidGlobalVariables	Global variables, i.e., non-const variables declared or defined in a C/C++ header file, should be avoided.
AvoidInlineForPublicMethods	Declaring a public method as inline should be restricted to class-internal usage. This includes C functions that are declared in a header file.
AvoidLargeMacros	A functional macro should not exceed a defined length.
AvoidMultipleImplementation-Inheritance	A class must not inherit from more than one base class except pure abstract classes.
CheckIdenticalTemplate-Instantiations	Templates should not be repeatedly instantiated with identical parameters.
DontExposePointersAsReturn-Values	A method should not return the address of an internal class attribute. Thus, it is not allowed to directly return an attribute that is a pointer, or the address of an attribute. It is only allowed to return the address of a single class object; except of class types that are collection classes (e.g. map, list or vector).

MakeAbstractClassesPure	A public abstract class with little implementation should be proposed to be made pure abstract, i.e., to have only pure virtual functions.
UseConstForMethodsWhen-Possible	A method should be marked as const when it does not change any class attribute.
UseConstForQueryMethods	A method indicated as query method should express this behavior by using const. A query method does not modify or set members of the class (directly or indirectly via other function calls).
UseMutableWithConstMethods	A const method may not modify class attributes by using const_cast or similar workarounds. However, it is allowed to use mutable attributes.
UseVirtualMethods	A certain number of public methods should be set to virtual since they are non-virtual by default; otherwise the class is too closed for extension.

About the authors

Reinhold Plösch is associate professor for Software Engineering at the Department of Business Informatics - Software Engineering at the Johannes Kepler University Linz. He is interested in source code quality - ranging from basic code quality to quality of embedded and safety critical systems. He is also interested in automatically measuring the object-oriented design quality based on design principles.

Johannes Bräuer is PhD student and employed at the Department of Business Informatics - Software Engineering at the Johannes Kepler Universitiy Linz. His PhD thesis concentrates on measuring and assessing software design based on fundamental design principles. Therefore, cooperations with communities of open-source projects and industrial partners are addressed for getting answers to research questions in this area.

Christian Körner is *Senior Key Engineer* at Siemens Corporate Technology in Munich. Professional interests are in the area of technical and management methods for *Development Efficiency*. Projects focus in the recent years was on developing and applying artefact based assessment methods for development organisations and automatic evaluation of software (design) quality. Projects range from small project interventions to large research collaborations with international partners.

Matthias Saft is working at Siemens Corporate Technology on software development related topics. His focus is code and design quality, its measurement, visualization and improvement. A corresponding architectural foundation is obligatory, and likewise considered. Additionally, he is interested in large scale lean and agile development methodologies, and their application in an industrial context.

Acknowledgments MUSE has grown successively and is based on many constructive discussions about measuring object oriented design, the development team is not limited to the authors of this paper. Initial work on the MUSE library was carried out by A. Dautovic, H. Gruber, and A. Mayr. J. Bräuer is currently working on some additional features and integrating rules developed by graduate students.