Preprint - original version can be found here: https://dl.acm.org/citation.cfm?id=3143454

Improving Object-Oriented Design Quality: A Portfolio– and Measurement-Based Approach

Johannes Bräuer
Johannes Kepler University Linz
Linz, Austria
johannes.braeuer@jku.at

Reinhold Plösch
Johannes Kepler University Linz
Linz, Austria
reinhold.ploesch@jku.at

Matthias Saft
Corporate Technology Siemens AG
Munich, Germany
matthias.saft@siemens.com

Christian Körner
Corporate Technology Siemens AG
Munich, Germany
christian.koerner@siemens.com

ABSTRACT

Current software development trends have shortened release cycles and forced developers to implement short-term solutions that cannot cope with increasing product complexity. This phenomenon of introducing hasty design choices or applying bad design practices becomes something known as technical debt, in particular design debt. To pay off this debt, the literature offers approaches for identifying these design flaws; however, few methods for properly prioritizing investment efforts are available. In this paper, we propose an approach that supports the decision-making process regarding design improvements. It identifies violations of design best practices that are then arranged within a two-dimensional portfolio matrix. This matrix combines the importance of practices of design quality with actual achievement relative to a benchmark suite. To show the application of the approach in a quality-improvement process, we performed a feasibility study on three open-source projects and a benchmark suite containing 50 projects. This study clearly shows that the importance of the design best practices greatly impacts the improvement decisions and must be aligned with the strategic quality goals of the product.

CCS CONCEPTS

• **Software and its engineering** → **Software design tradeoffs**; *Software verification and validation*;

KEYWORDS

software quality, design quality, technical debt, design debt

ACM Reference format:
Johannes Bräuer, Reinhold Plösch, Matthias Saft, and Christian Körner. 2017. Improving Object-Oriented Design Quality: A Portfolio– and Measurement-Based Approach. In *Proceedings of Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement, Gothenburg, Sweden, October 2017 (IWSM-Mensura),* 11 pages.
DOI: 10.475/123_4

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
IWSM-Mensura, Gothenburg, Sweden
© 2017 Copyright held by the owner/author(s). 123-4567-24-567/08/06…$15.00
DOI: 10.475/123_4

1 INTRODUCTION

Software systems must continuously evolve to cope with constantly changing requirements and environments [26]. To handle these challenges, many software projects choose design and development approaches that support short deployment cycles but lack the ability to manage increasing product complexity. Without consideration of the flaws that have been introduced due to time pressure, overall development and maintenance costs can increase in the long run. In 1992, Cunningham already pointed to this phenomena introducing the term technical debt [8].

Design debt, as a particular type of technical debt referring to flaws in software design, expresses the additional maintenance effort required in the future that results from hasty and inappropriate design choices in the past or from applying bad programming practices. Generally, these issues are symptoms of higher-level design problems, known as design flaws [19], design smells [10], anti-patterns [4] or violated design principles [21]. Thus, identifying these issues is the first step towards paying back the debt.

To identify design issues, this work follows the approach of finding violations of object-oriented design best practices directly in the source code. In the remainder of this article, we use the term design best practices while always referring to object-oriented design best practices. Typically, design best practices allow the reuse of expert knowledge and can therefore be applied to guide software designers and developers. To find the violations of design best practices, we developed a tool called MUSE because there is no tool available that would provide the flexbility to specifiy and check such best practices directly in code [25]. MUSE contains a set of design best practices (design rules) for the programming languages Java, C# and C++. Most design rules cover all three programming languages, since they are related to object-oriented concepts rather than language features.

Based on a MUSE measurement, which is a compiled list of design best practice violations referencing exact locations in the source code, developers can start to investigate the problem spots. Nevertheless, they are left in the dark regarding where to actually invest effort to pay off particular instances of the design debt. Consequently, an important next step is to provide guidance for decisions about addressing appropriate design improvements and suggestions for refactoring. As explained by Brown et al. [3], this decision depends on various factors, such as the value of the debt,

the interest rate one currently pays on the debt and potential impact the debt has on future development. Typically, debt will be paid for the purpose of making a lucrative investment in the software.

In order to guide developers towards the most lucrative investments, this paper proposes a portfolio matrix approach that incorporates two dimensions for quantifying and prioritizing design improvements. First, it considers the importance of the violated design best practices derived from a survey [5]. Second, it expresses the current quality level of each practice relative to a benchmark suite. These two dimensions are then mapped in order to recommend which debt items should be paid and which can be ignored in the first place. To show the benefits of this portfolio matrix, this paper describes the approach and then provides a realistic example using three open-source projects and a benchmark base that amount to 4.55 million logical lines of code (LLOC). This investigation is limited to the programming language Java, meaning that all projects are written in this language and the set of design rules in MUSE is reduced to Java rules. A full validation is not provided but will be addressed in future research.

The remainder of this article is structured as follows. The next section dives into related work in regard to object-oriented design assessment and the prioritization of refactorings. Afterwards, our approach for guiding design improvements is presented by discussing the concepts of measuring design best practices and assessing a project against a benchmark suite. In Section 4, the feasibility study shows the way to proceed from measuring over assessing to deriving appropriate design improvements in open-source projects. Finally, the paper draws conclusions and provides avenues for future work.

2 RELATED WORK

For this work, we rely on contributions from two research areas. First, this section shows approaches for measuring and assessing object-oriented software design. Second, it reflects ideas for prioritizing and guiding design improvements.

To measure object-oriented design quality, the literature distinguishes several different approaches. Metric-based approaches such as QMOOD [1] and the metric suite provided by Chidamber and Kemerer [6] are typical in that they try to measure design quality by means of, for example, cohesion, coupling and inheritance-related metrics. The effort made to measure these design aspects is typically low. Empirical studies indicate that some of these metrics (e.g., inheritance depth, number of methods, coupling between objects) are useful for measuring design quality in the sense that there is a correlation between the metrics and external quality (usually bugs) [2, 31]. Nevertheless, metric-based approaches fail to support developers, since it is difficult to grasp the semantic relation between a metric and the source code that has to be changed in order to improve the metric.

Another set of approaches tries to identify so-called design smells or code smells [9, 15, 19, 23]. The work on design smells is influenced by the design heuristics of Riel [27] as well as the research on refactoring and anti-patterns by Fowler et al. [10] and Brown et al. [4], respectively. These design smells, for instance, data clumps or feature envy [10], semantically express a design flaw and make it tangible for software developers. From our point of view, this is an improvement over metric-based approaches. However, typical measurement approaches (e.g., [19, 32]) for design smells are also metric-based, with the difference that they do not rely on a single metric but on a combination thereof. Thus, there still remains the problem of properly finding the causes in the source code.

Ganesh et al. [11] collect and categorise design smells using a pattern language. They focus on semantically grasping the design problem without providing any hints of how to measure design smells [12]. In addition, this group published a design assessment approach called MIDAS [28], which refers to the skills and knowledge of experts who manually assess adherence to design principles. By using this more abstract view of design quality, a better understanding of issues and problem sources can be achieved. To standardise these manual investigations, a model for design quality should be used that is currently not available [28]. Our idea of design best practices (see Plösch et al. [25] and the discussion in the next section) follows the approach of Ganesh et al. [11], providing 68 measures for capturing object-oriented design quality issues.

The second area of related work covers the prioritization of design improvements. In the recent past, this topic has been addressed by work around measuring the technical debt of software systems. Technical debt is an abstraction of the term design debt or code debt. The debt metaphor is used to communicate the software quality issues contained in a product [22]. Accordingly, it tries to express the required effort to clean up source code or software design in an attempt to limit risks to future development and maintenance. This is why technical debt approaches can be seen as a prioritization technique for quality issues that can guide targeted improvements.

As mentioned above, design smells are used to expose symptoms at the design level. For identifying these symptoms, detection strategies have been proposed by Marinescu [19]. This idea has been elaborated by considering the influence of design smells on design flaws [20]. To determine the individual influence of a design smell, seven senior software designers were interviewed. The derived influence levels are then used to direct improvement decisions towards those design smells with the highest impact on design quality.

A similar idea was adopted by another research group [34]. They proposed an approach that relies not on an external view of design smells but rather calculates the cost and impact of design debt. The calculation of the cost dimension is based on standard metrics [34]. To determine the impact of design debt on quality, the two aspects of correctness and maintainability are taken into consideration and rest upon the likelihood of defects and changes [34, 35]. Finally, the identified design smells are ranked according to the cost and impact dimensions and visualised in a cost-benefit matrix.

SQALE (Software Quality Assessment based on Lifecycle Expectations) is another model for calculating technical debt [18]. It defines specifications for calculating so-called remediation indices, that is the estimated remediation effort required to fix a violation of a rule. The remediation indices can be aggregated and translated to "financial debt", a term used by SQALE that corresponds to remediation costs. SQALE calculates the required effort to fix all violations for each rule. The resulting technical debt is given in person days.

3 APPROACH TO GUIDE DESIGN IMPROVEMENTS

Modern software development aims to develop high-quality products within limited time and budget constraints; this work narrows software quality down to design quality for object-oriented software. Thus, investing the proper amount of effort to refactor the most important design flaws is required to stay within the defined constraints and to pay off design debt. Notwithstanding this, it is necessary to identify those design flaws that have the highest impact on quality. The current quality state of the project must be known, because it is typically not possible to fix all identified design flaws. Given this information regarding those findings with the highest impact, we can then systematically derive which improvements are the most efficient.

For this purpose, we introduce the concept of a portfolio matrix (see Section 3.3), which incorporates two dimensions to quantify and prioritize design improvements. The first dimension is the importance of design best practices, on the vertical axis while, the second dimension represents quantiles determined from a benchmark approach, on the horizontal axis. The latter puts design best practices relative to their values in a benchmark base and can therefore reflect the quality state of a project. To explain the approach in more detail, the two dimensions are described next, before the portfolio matrix is explained at the end.

3.1 Design Best Practices

As mentioned in Section 2, the community is missing a reference model for design quality [28]. In the long run, our work tries to close this gap, using a different approach than that of related work in the past. In contrast to measure metrics or design smells to express the design of software, we are interested in understanding the compliance of source code with (fine-grained) design principles such as *information hiding*, the *single responsibility principle* or *don't repeat yourself* [24]. Therefore, it is important to facilitate the understanding of design deficiencies at a level more abstract than design smells but not as coarse as that suggested by Ganesh et al. [12], who cluster design problems based on abstraction, encapsulation or modularisation.

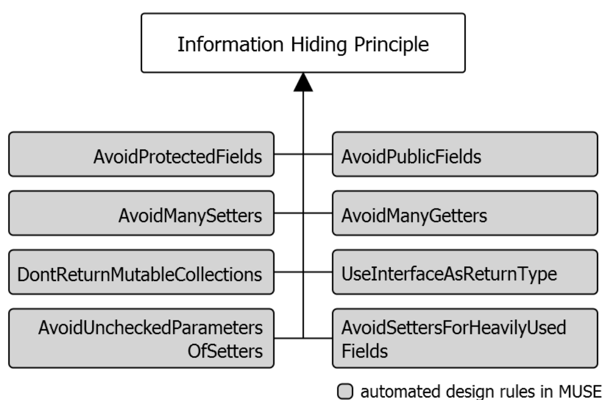

Figure 1: Design principle and related design best practices

A list of 32 design principles was systematically identified, with important ones determined based on a survey [24]. Guided by this set of principles, a top-down approach was applied to identify and specify design best practices related to the design principles. An example of a design principle and its related design best practices is illustrated in Figure 1. In this example, the information hiding principle is affected (threatened) by, for instance, classes that provide public fields, methods that return mutable collections or methods that could declare an interface instead of a concrete class as the return type. While the example is just an abstract view of measuring design principles by means of design best practices, a more elaborate discussion is provided in Plösch et al. [24], which explains the underlying meta-model and its applicability.

As another major difference to Ganesh et al., we did not stop at the specification level but also implemented the static code analysis tool MUSE, which analyses Java, C# or C++ source code and finds violations of these design best practices directly from the source code [25]. As already mentioned above, this article concentrates on the Java part of MUSE, comprising 49 design best practices. The resulting MUSE measurement represents a list of violations that can be uploaded to SonarQube[1]. In SonarQube, the findings are linked to the source files, where developers can start investigating identified issues.

MUSE can be applied to manage design improvements for a software product. However, software developers and designers should focus on the most important and critical violations to efficiently invest their effort. In order to understand the importance of design best practices, we conducted another online survey [5]. In total, 214 people participated in this survey, resulting in an average of 138 opinions for each practice. The importance scale the participants used to categorize the 49 design best practices ranged from very low to very important. Table 1 depicts all 49 design best practices, sorted by their perceived importance for design quality but without an assessment in regard to a particular design principle; latter will be addressed by future work.

Additionally, Table 1 shows an importance range for each design best practices, derived from the average and standard deviation of all opinions for each best practice. The only rule that does not provide a range is *AvoidDuplicates*, because there is common agreement on its very high importance. For all the other design best practices, the range provides some degree of freedom when using the rules, for instance, to assess software design. This flexibility is needed since many design decisions are context-depended, as survey participants mentioned. Altogether, the range defines an appropriate space to adjust the importance of design best practices for a particular project, but it is not recommended to go beyond the upper or lower boundaries.

3.2 Benchmarking

Obtaining measurement values by executing, for instance, our measuring tool MUSE, is not sufficient to make improvement decisions. For this aim, it is necessary to know whether the number of violations is critical for a project or if it is instead within an acceptable range. To identify this criticality regarding the number of violations, we rely on benchmarking.

[1] https://www.sonarqube.org/

Table 1: Design best practices ordered by importance

Id	Rule Name	Importance	Range
R1	AvoidDuplicates	vh	vh
R2	AvoidUsingSubtypesInSupertypes	vh	h - vh
R3	AvoidPackageCycles	vh	h - vh
R4	AvoidCommandsInQueryMethods	vh	h - vh
R5	AvoidPublicFields	vh	h - vh
R6	DocumentInterfaces	h	m - vh
R7	AvoidLongParameterLists	h	h - vh
R8	UseInterfacesIfPossible	h	m - vh
R9	AvoidStronglyCoupledPackages	h	m - vh
R10	AvoidNonCohesiveImplementations	h	m - vh
R11	AvoidUnusedClasses	h	m - vh
R12	DontReturnUninvolvedDataFromCommands	h	m - vh
R13	AvoidNonCohesivePackages	h	m - vh
R14	DocumentPublicMethods	h	m - vh
R15	UseCompositionNotInheritance	h	m - vh
R16	DocumentPublicClasses	h	m - vh
R17	AvoidPublicStaticFields	h	m - vh
R18	AvoidDiamondInheritanceStructuresInterfaces	h	m - vh
R19	AvoidLongMethods	h	m - vh
R20	AvoidSimilarNamesForDifferentDesignElements	h	m - vh
R21	AvoidUnusedAbstractions	h	m - vh
R22	CheckUnsuitableFunctionalityOfClass	h	m - vh
R23	AvoidSimilarAbstractions	h	m - vh
R24	DocumentPackages	h	l - vh
R25	UseInterfacesAsReturnType	h	l - vh
R26	AvoidUncheckedParametersOfSetters	h	h - vh
R27	AvoidSimilarNamesForSameDesignElements	m	m - h
R28	CheckObjectInstantiationsByName	m	l - h
R29	AvoidRepetitionOfPackageNamesOnAPath	m	l - h
R30	ProvideInterfaceForClass	m	l - h
R31	AvoidRuntimeTypeIdentification	m	l - h
R32	AvoidDirectObjectInstantiations	m	l - h
R33	CheckUnusedSupertypes	m	l - h
R34	AbstractPackagesShouldNotDependOnOtherPkg	m	l - h
R35	DontReturnMutableCollectionsOrArrays	m	l - h
R36	AvoidMassiveCommentsInCode	m	l - h
R37	AvoidReturningDataFromCommands	m	l - h
R38	UseAbstractions	m	vl - h
R39	CheckUsageOfNonFullyQualifiedPackageNames	l	l - m
R40	AvoidManySetter	l	vl - m
R41	AvoidHighNumberOfSubpackages	l	vl - m
R42	AvoidConcretePackage	l	vl - m
R43	AvoidSettersForHeavilyUsedFields	l	vl - m
R44	AvoidAbstractClassesWithOneExtension	l	vl - m
R45	DontInstantiateImplementationsInClients	l	vl - m
R46	AvoidManyGetters	l	vl - m
R47	AvoidProtectedFields	l	vl - m
R48	CheckDegradedPackageStructure	l	vl - m
R49	AvoidManyTinyMethods	l	vl - m

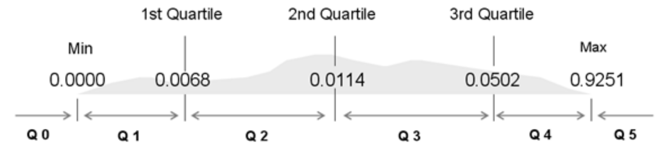

Figure 2: Quartile-based value distribution for one design best practice

Benchmarking is a well-established concept in many business areas, where (similar) objects such as products, services, processes or organizations are checked against each other for a specific purpose [7]. The purpose can be, for example, the evaluation of a relative object value or derivation of suggestions for improvements.

Based on the work of Simon et al. [30] on software quality benchmarking, we developed an automatic assessment approach for software quality that relies on the benchmarking concept [13, 22]. For assessing the design quality of software projects, we built up a benchmark database that contains the quality data of all reference projects. In this context, it is adequate to store the number of rule violations for each design best practice and for each reference project. In addition, we store values of various size metrics for each project as needed for normalization; examples of size metrics are lines of code, number of methods or number of classes.

Normalization of a measurement result makes raw results comparable with other projects. As an example, in a project with several thousand functions (e.g., 2,000), 20 undocumented functions might be a negligible deficit. But in a project with only 50 functions, having 20 undocumented function might be wholly unacceptable. In this way, the normalization operation eliminates the influence of the size of the project on the measurement results. After having applied normalization strategies for each measure, the normalized data are comparable with data from other projects (e.g., 1% undocumented functions in project A versus 40% in project B).

Based on the benchmark database, which can be used for several assessments, a benchmark suite needs to be generated in a second step. A benchmark suite is tailored to a specific application of benchmarking, comprising only those projects that are selected as reference projects. For instance, a benchmark suite might be limited to embedded systems projects written in the C programming language. To create the benchmark suite, we calculate a value distribution for each measure on the basis of their normalized values. As a result, these value distributions reflect commonly used statistical figures such as quartiles, quintiles, deciles or percentiles.

A quartile-based distribution, as demonstrated in Figure 2, divides the values into four areas Q1 to Q4, where Q1 and Q4 are delimitated by the minimum and maximum value of the best and worst project in the benchmark suite. If an investigated project is better than the benchmark minimum, it falls in the additional area Q0. Correspondingly, this applies to the maximum value indicated by Q5. All in all, the numbers of rules assigned to particular quartile express the current quality state of a project relative to the benchmark suite.

3.3 Improvement Portfolio Matrix

The idea of using portfolio matrices for strategic product decisions is quite old, going back to the two-by-two matrix proposed by the *Boston Consulting Group (BCG)* and *McKinsey*. The idea of using portfolio matrices need not be restricted to product decisions but can also be used for decisions related to software, as shown by Zazworka et al. [34].

Subdividing a two-dimensional matrix to four areas is a common approach used in strategic product placement, as already mentioned. The *Boston Consulting Group matrix*, for instance, is a known tool to place a product in one of four quarters depending on its relative market share and potential market growth [14]. Accordingly, managers can make better investment decisions based on the quarter the product is in. McKinsey, in cooperation with General Electrics, proposed the *GE/McKinsey matrix*, which provides a more elaborate view on a two-dimensional matrix. The scales on the two dimensions are more selective in order to allow better differentiated planning. Additionally, the *GE/McKinsey matrix* divides the matrix not into four quarters but rather to action fields with a size depending on the application context. The advantage of this portfolio approach is that it allows defining and executing standard improvement decisions for each action field.

To define our design improvement portfolio, we followed the ideas behind the *GE/McKinsey matrix* for two reasons. First, our dimensions provide a broader spectrum of distinguishing characteristics, that is ranging from very low to very high on the vertical axis and from Q0 to Q5 on the horizontal axis. Hence, our matrix should provide all possible options. Second, dividing the field in four equivalent quarters would limit the variety of the dimensions. In other words, we would lose the details needed to provide more targeted design improvement recommendations.

Based on our investigations regarding the importance and derived quartile of each practice, we suggest the following general investment strategies.

Restricted investment area: A design best practice in the bottom left corner should deserve little attention, as in this case the project team largely follows the design best practices and performs better than 50% of the projects in the benchmark suite. Moreover, addressing violations of this design best practice does not improve overall design quality to a large extent. Consequently, no or little effort should be invested.

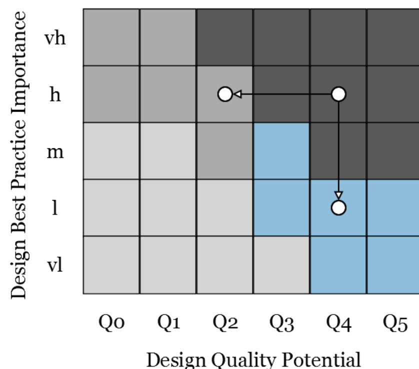

Figure 3: Design improvement matrix

Selective investment area I: A design best practice in the top left corner deserves some attention. The project team is already concerned about the practice, so additional effort is hardly required. We recommended investing effort in a selective way.

Selective investment area II: A design best practice in the bottom right corner deserves some attention, possible more than the design best practices in the selective investment area I. Although the affected design best practices are not that important for the overall design, the project performance is weak, that is similar to the worst projects. We recommend investing effort selectively.

Immediate investment area: A design best practice in the top right corner deserves the most attention. In fact, following such a design best practice is considered to be important for the overall design quality, as the project team has not adhered to important design best practices (at least compared to reference benchmark base). Besides, the project is worse than 50% of the projects in the benchmark suite. As a result, effort must be invested into addressing these violations.

As depicted in Figure 3, Q0 and Q1 have the same proportion of the investment area. We decided not to differentiate between these quartiles because Q0 contains just those design best practices for which the project is performing better than its best counterpart in the benchmark suite. While it might be interesting for a project team to see where it is performing very well, we would not provide a differentiated improvement recommendation compared to Q1. Equally, the same applies for the opposite side with Q4 and Q5.

While the above high-level recommendations provide first hints for driving targeted improvements, for the most critical area, represented by the top right corner, more specific guidance can be given. This specification considers Q5 to be equal to Q4. Design best practices that are in Q4/vh are those design flaws that must be addressed first. A high-quality product cannot ignore design flaws in this field. Concentrating further improvements on these design best practices achieves significant quality enhancements. The next wave of improvement must be invested in Q3/vh and Q4/h, since they are still critical and not followed well. Finally, the remaining time and budget should flow into design best practices in Q2/vh, Q3/h and Q4/m. This concludes that these design best practices move from the recommendation space of immediate to selective improvements.

By applying the improvement strategy discussed above, the project team commits to the default importance settings. Thus, a project team would first focus on design best practices that are considered to be (very) important according to the survey. However, many project teams have an 'it depends'-view of best practices, so they need a way to adjust the default settings depending on context. This flexibility is provided by the importance range for each design best practice. Accordingly, the team can vertically move design best practices within the matrix by up- or downgrading them. Changing the importance of a design best practice should not be a comfortable, ad-hoc decision that moves design best practices, for example, from the immediate investment area to the selective investment area. Instead, it must be considered from a strategic viewpoint and always must be aligned with business and product goals.

Given this option to adjust the importance within defined boundaries, our portfolio matrix provides another lever for handling design best practices, especially, border cases in Q2/h, Q3/m and Q4/l that are close to the immediate investment area. Consequently, a team should reconsider the practices in these fields to identify candidates that are worth following. By upgrading an interesting practice, the team commits to invest improvement effort since it moves into the top right corner.

4 FEASIBILITY STUDY

In this feasibility study, three projects are assessed against a benchmark suite containing 50 reference projects to present (1) the effects of different levels of importance for design best practices on overall quality evaluation and (2) to demonstrate our proposed improvement matrix approach. The entire set of reference projects is shown in the Appendix. The selection process for projects in the benchmark base focused on open-source Java projects with a high number of downloads and therefore seem to be used by a larger community. Additionally, we only selected projects that are in development, meaning that their last release date was within the last two years. This ensures that they use a more current Java version and programming techniques. Another selection criterion was project size, ranging from small (10 KLLOC) over medium (~100 KLLOC) to large (>500 KLLOC). In total, the logical lines of code of all projects amount to 4.55 MLLOC.

After downloading the source code, external libraries needed by the reference projects were collected. This step was required because MUSE guarantees a higher measurement accuracy when all referenced libraries are available that provide additional details on type declarations and their usage. Typically, just the source code of the reference projects is stored on the public repository platform Github, while build technologies like Maven or Gradle are applied to determine the needed dependencies. Consequently, libraries are not stored along with the source code but rather acquired dynamically. In order to collect all required libraries for a project, we used our tool LibLoader, which automatically downloads them.

Afterwards, MUSE was executed on all 50 reference projects. This measurement returned the number of rule violations and a normalization value for each design best practice and for each project. While normalization is often carried out on the basis of LLOC, our experiments showed that this simplistic approach distorts the results. Choosing a normalization factor that better fits with design best practice leads to more realistic values. We therefore defined individual kinds of normalization for each design best practice. After calculating the normalized findings by dividing the number of rule violations by the normalization value, these findings were stored in the benchmark suite, representing the raw data for calculating the quartile-based distribution discussed in Section 3.2.

Before deriving the quartiles, an outlier detection approach was conducted in order to remove measuring results that distort the normal distribution. Specifically, the outlier labeling technique proposed by Hoaglin and Iglewicz [16] was applied. We followed the recommendation of using g'=2.2 for $50 < n < 100$ [16]. As a result, we identified abnormal design rule violations on both extrema. Finally, the cleaned-up data were used to derive the quartile boundaries for each design best practice.

Table 2: Study projects

	Version	Release Date	LLOC
Elasticsearch	5.3.0	2017-03-28	294,948
Vaadin	7.7.8	2017-03-27	144,254
Apache ZooKeeper	3.5.3-rc0	2017-03-28	34,112

The three investigated projects in this study were the open-source projects Apache ZooKeeper, Elasticsearch and Vaadin, reviewed in Table 2. They were selected mainly based on their project size, application domain and development state. While ZooKeeper is medium and Vaadin is large compared to the benchmark suite, Elasticsearch would be the third-largest project in the data base. Moreover, all three projects are widely used and developed by a large community; for example, Elasticsearch has 804 contributors on Github.

After selecting the projects for study, the quality investigation could proceed. Therefore, we followed the steps proposed by Klï£¡s et. al. [17], who discuss the usage scenarios of quality models. The first step focused on measuring design quality with MUSE. Based on measurement results, the design quality of each of the three projects was assessed using the derived quartile distributions. Lastly, we selected one project to discuss the applicability of the improvement portfolio matrix.

4.1 Design Measurement

The MUSE measurements for all three projects are shown in Table 3. There, the 49 design best practices are referenced by their identification numbers defined in Table 1. The first column of each project depicts the absolute number of violations. In other words, MUSE returned 141 violations of the design best practice *AvoidDuplicates* for ZooKepper, as shown in the first row, which contains the identification number R1. In addition to the absolute number of violations, the total number of entities checked by each design rule is provided in the second column.

Based on the total number of entities, the numbers of violations are normalized. The importance of normalization is briefly mentioned in Section 3.2. We also recognized that MUSE needed to be enhanced in this regard because the prior version used size metrics for normalization that were too coarse-grained. Formerly, MUSE returned, for instance, the number of methods in the project to normalize the design best practice *AvoidCommandsInQueryMethods* (R4) and *AvoidLongParameterLists* (R7). However, this approach was unfair because R4 works just on query methods that represent a considerably lower number than to the total number of methods for a project. Hence, the MUSE rules were enhanced by returning the exact number of entities in the project that are verified. The relative number of violations for each rule was calculated by dividing the absolute number by the number of entities. This relative value was then checked against the benchmark suite, returning a quartile for each rule and each project.

4.2 Design Assessment

In the upper part of Table 4, a summary of the derived quartile distribution for all three projects is shown. According to this table,

Table 3: Design measurement results

	ZooKeeper		Elasticsearch		Vaadin	
Id	findings	entities	findings	entities	findings	entities
R1	141	3,411	1,324	29,494	301	14,425
R2	4	33	85	464	8	222
R3	39	34,112	4,290	294,948	170	144,254
R4	7	623	116	4,886	77	3,285
R5	41	982	83	8,783	539	3,951
R6	81	278	499	1,783	247	1,636
R7	11	2,762	240	31,135	6	14,078
R8	146	1,637	9,853	39,026	905	9,344
R9	3	21	83	349	12	171
R10	54	276	528	2,751	290	1,125
R11	19	276	210	2,751	157	1,125
R12	8	424	116	3,161	15	2,630
R13	4	21	26	349	16	171
R14	673	1,204	7,721	13,565	1,683	5,773
R15	3	73	36	432	66	431
R16	106	313	1,628	2,950	364	1,181
R17	0	44	40	101	21	106
R18	4	78	124	1,344	254	734
R19	44	2,762	169	31,135	157	14,078
R20	4	567	18	5,857	115	2,583
R21	10	74	86	753	115	588
R22	4	239	69	2,500	35	927
R23	0	232	42	2,784	0	1,137
R24	0	0	0	0	0	0
R25	38	395	3690	12,509	315	2,703
R26	28	32	255	370	439	853
R27	8	567	175	5,857	147	2,583
R28	5	2,073	4	18,417	4	4,365
R29	0	21	1	349	2	171
R30	59	313	431	2,950	75	1,181
R31	109	34,112	1052	294,948	729	144,254
R32	31	276	524	2,751	87	1,125
R33	5	199	294	2,655	69	865
R34	2	21	24	349	13	171
R35	24	127	547	1,300	74	354
R36	7	2,762	134	31,135	148	14,078
R37	66	424	1,684	3,161	206	2,630
R38	10	21	211	349	91	171
R39	0	21	21	349	7	171
R40	1	36	20	340	46	257
R41	0	21	13	349	4	171
R42	4	21	172	349	47	171
R43	5	316	71	2,724	59	2,243
R44	2	19	46	357	10	130
R45	141	553	3962	7,997	395	1,238
R46	5	36	56	340	55	257
R47	125	982	220	8,783	190	3,951
R48	2	21	16	349	11	171
R49	10	276	162	2,751	30	1,125
Total	2,093	-	41,171	-	8,806	-

Table 4: Design assessment results

Quartile	ZooKeeper	Elasticsearch	Vaadin
Q-0	6	1	2
Q-1	6	3	11
Q-2	15	9	13
Q-3	13	17	8
Q-4	9	18	13
Q-5	0	1	2
Q-Index	448 (53.65%)	325 (38.92%)	421 (50.42%)
Q-I. soft	315	229	302
Q-I. strict	567	441	527

ZooKeeper outperforms the entire benchmark suite in six cases as indicated by Q0. In other words, ZooKeeper has fewer or the same number of violations for six design best practices (design rules) compared to the best project in the benchmark suite. For further six design best practices, this project is in Q1 and within the top quarter. The majority of design best practices fall in Q2 and Q3, with 15 and 13 design best practices, respectively. Q4 has nine design rules, while no design rule returned more findings than the worst project in the benchmark suite. This was not the case for Elasticsearch; because it has too many package cycle so that the rule *AvoidPackageCycles* (R3) fell in Q5.

From a general point of view, Elasticsearch has more design debt than the other two projects because 36 of 49 design rules fell in quartiles higher or equal to Q3. In contrast, ZooKeeper has 22 and Vaadin has 23 rules in quartiles Q3, Q4 or Q5. To get a more discrete assessment regarding the measured design quality, a quality index was calculated for each project.

To calculate this quality index, the design best practices were weighted according to their importance in Table 1 and multiplied by their achieved quantile. For example, *AvoidDuplicates* (R1) has an importance of very high, representing 5 points; vice versa, very low importance counts for 1 point. In ZooKeeper, this design best practice achieves Q3, so that the importance is multiplied by value 2; Q1 represents a multiplier of 4, Q2 a multiplier of 3, Q3 a multiplier of 2 and Q4 a multiplier of 1. As a result, *AvoidDuplicates* provides 10 points to the quality index of ZooKeeper. Lastly, adding up all design rules returns a quality index of 448 points for the first project. This value is then put in relation to the maximum points to determine the relative quality index shown in brackets.

Given this purely quantitative quality assessment, a comparison of the three projects is valid because they are all measured against the same comparison base. Thus, ZooKeeper and Vaadin achieve almost the same quality index, while Elasticsearch is performing worse than the others. In fact, there remains a gap of 96 points (11.5%) between Elasticsearch and Vaadin, the second-best project. Generally, this is a fair comparison because all three projects are relatively assessed against the same benchmark suite and their quality indices are determined in the same way.

As mentioned in Section 3.1, each design best practice provides an importance range. Developers and design analysts can take

advantage of this range to up- or downgrade design rules depending on their perception. When upgrading all design rules to the highest possible importance value, a project would commit to a very strict assessment. This strict assessment raises the quality gate to a high level, forcing the developers to very strictly follow design best practices. A very soft assessment would downgrade all design rules, resulting in few design best practices at a high importance level.

To show the difference between the two extrema a project might theoretically acquire, we calculated strict and soft quality indices for all three projects. As depicted in Table 4, there is actually a wide space between the two assessments. ZooKeeper has a soft quality index (Q-I. soft) of 315 points and a strict one (Q-I. strict) of 567 points. Focusing on the 567 points, it is obvious that the index increased to this value since intensifying the quality gate raises the importance level of 21 design rules to very high. Consequently, there are now 26 rules that all contribute 5 points to the quality index.

With a set of 26 design rules in the very high area, developers are left in the dark about where to start investing improvement effort. In other words, changing the design and fixing issues by randomly selecting rule violations may cause over-engineered parts on the one hand, while design flaws are ignored on the other hand. Hence, it is required to guide developers in selecting the design rules that are worth investing resources. For this purpose, our improvement matrix provides an appropriate tool, since it maps the rules in the very high area to the achieved quality state expressed by the measured quartile.

4.3 Design Improvement

This part discusses the application of the portfolio matrix for Apache ZooKeeper. Although the previous section motivates the utilization of the matrix to investigate the 26 very important design best practices of a strict assessment, this discussion starts with the default setting shown in Table 1.

To determine the improvement matrix for ZooKeeper, the quantile assessment is combined with the importance scale. Subsequently, the distribution of design best practices can be seen in Figure 4. It is evident from the result that 15 design best practices are in the immediate investment area, 10 are in the restricted investment area, and 24 are in the selective investment area. This already helps reduce the entire set of 49 design best practices to a subset that is worth for improvements. In other words, the rest can be ignored in the first place.

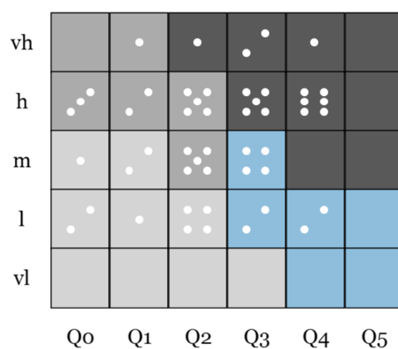

Figure 4: Default improvement matrix for ZooKeeper

As discussed in Section 3.3, the most critical field is Q4/vh. In the case of ZooKeeper, only *AvoidPublicFields* (R5) is in this field. The reason is that the ratio of 41 violations in 982 classes is too high compared to the benchmark suite. In more detail, the relative value of violations for *AvoidPublicFields* (R5) is 0.042, while Q3 requires a value of 0.029. For ZooKeeper, this means there can be a maximum of 28 violations to achieve Q3. Consequently, the team must eliminate 13 design rule violations or 31% of the 41.

The next two fields that warrant further improvement are Q3/vh and Q4/h. According to Figure 4 and as described in Table 5, eight design best practices fall in these two fields. Since we do not differentiate between Q3/vh and Q4/h, further improvement could come from any of the eight rules. To help select a practice, Table 5 contains the limits to the next lower quartile. Given this information, it is possible to derive the absolute number of violations that must be addressed. For instance, when removing nine code duplicates, *AvoidDuplicates* would move from Q3 to Q2.

From a pragmatic point of view, the next improvement decision could rest upon the lowest number of violations that need to be fixed. Based on Table 5, *AvoidNonCohesivePackages* is the practice that should be followed, because the cohesiveness of just two packages must be enhanced. However, it is likely that changing the package structure of these packages consumes more resources than documenting nine public classes, as indicated by *DocumentPublicClasses* (R16). Nevertheless, this is just an estimate; we do not know the effort to fix individual violations of design best practice. Further work will concentrate on investigating this aspect.

The six remaining practices in the immediate investment area are *AvoidPackageCyles* in Q2/vh and *AvoidStronglyCoupledPackages*, *UseInterfacesIfPossible*, *AvoidUnusedAbstractions*, *DocumentInterfaces*, and *DocumentPublicMethods* in Q3/h, representing three problem areas. First, two of the six rules indicate that ZooKeeper developers have issues in regard to their package structure, since they build package cycles and the packages are too strongly coupled. Second, the way of dealing with abstractions causes some violations. Hence, instead of using the concrete class for type definitions or return type, an interface – implemented by the class – could be applied. Moreover, ZooKeeper defines abstract classes or interfaces that are never used. If they are required for providing extension points, they should be excluded from the MUSE analysis. Third, design deficits are found in the missing documentation of interfaces and public methods.

Following our approach and investing at least in the immediate investment area with the goal of changing the quality level for each affected design best practice by one quality level (i.e., from Q4 to Q3, from Q3 to Q2 and from Q2 to Q1), resources are needed to fix 1,335 violations of the 15 distinct design best practices. Compared to the overall number of violations (2,093) for ZooKeeper, our approach identifies the most relevant problems in the source code. Applying technical debt calculation (see, e.g., the SQALE approach [18].) would even allow us to monetarize and thereby even better plan the needed resources.

The chosen level of importance obviously has major impact on the results of evaluation, as already shown in Section 4.2. When

Table 5: Details about design rule violations in Q3/vh and Q4/h

	absolute No. of Violations	No. of Entities	relative No. of Violations	Quartile Limits		No. of Fixes
Q3/vh				to Q2/vh		
(R1) AvoidDuplicates	141	3411	0.041	0.039	132	9 (6%)
(R2) AvoidUsingSubtypesInSupertypes	4	33	0.041	0.058	1	3 (75%)
Q4/h				to Q3/h		
(R16) DocumentPublicClasses	106	313	0.339	0.310	97	9 (8%)
(R19) AvoidLongMethods	44	2762	0.016	0.014	38	6 (14%)
(R26) AvoidUncheckedParametersOfSetters	28	32	0.875	0.732	23	5 (18%)
(R7) AvoidLongParameterLists	11	2762	0.004	0.003	7	4 (36%)
(R13) AvoidNonCohesivePackages	4	21	0.190	0.132	2	2 (50%)
(R20) AvoidSimilarNamesForDifferentDesignE.	4	567	0.007	0.000	0	4 (100%)

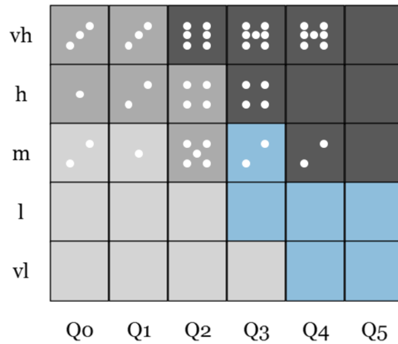

Figure 5: Strict improvement matrix for ZooKeeper

upgrading all design rules to the highest possible importance value, a project would commit to a very strict assessment with considerable implications for the improvement program. For ZooKeeper, this would result in the improvement matrix depicted in Figure 5, which indicates more improvement effort compared to Figure 4. Investing at least in the immediate investment area with the goal of changing the quality level for each affected design best practice by one quality level would require resources to fix 1,632 violations of 26 distinct design best practices. Compared to the overall number of violations (2,093) for this project, our approach focuses on the most relevant problems in the source code. Nevertheless, this is 22% more than the standard median importance discussed before.

The feasibility study clearly shows that the importance level has large impact on improvement decisions and should therefore not be taken lightly. It should always be aligned with the strategic quality goals of the product or project.

5 LIMITATIONS

In any experimental study, there are factors that can influence the findings and represent threats to validity. Threats to external validity concern the ability to generalise the results [33]. In this regard, restricting the programming language of the projects to Java represents an external threat to validity. While our general idea of using a portfolio matrix to guide design improvements is not affected, the importance of individual design best practices differ among object-oriented programming languages. Thus, Table 1 cannot be applied to drive improvements for C++ or C# projects since the ranges are slightly different [5].

Threats to internal validity concern any confounding variables that could have influenced the results of our study [33]. In this work, our measuring tool might represent a confounding variable. MUSE offers the possibility to adjust threshold values or to exclude source code parts from being analyzed. This is necessary to perfectly integrate MUSE in a continuous quality improvement process depending on project requirements or strategic projects goals, which define the adjustment of the MUSE levers in practice. However, for this experiment we decided to leave the default settings for the analysis of the three investigation objects and the reference projects in the benchmark base. This decision ensures equal measurement conditions for all projects.

Another threat that results from the measurement is the likelihood of wrongly identified violations, which should be classified as false-positives. When designing software, it sometimes occurs that a team intentionally violates a design best practice in order to simplify, for instance, maintenance tasks of a designated component. Without excluding parts or customizing our design rules, a measurement can contain violations even though the design decision may have been deliberately chosen. To address this threat, the 53 project teams must individually classify wrong violations. Since this task would be very time consuming and because all projects are exposed to the same likelihood of containing false-positives, we decided to accept this threat.

In addition, the selection of the reference project for the benchmark base leads to another internal threat. To control this issue, we selected Java projects of different sizes and with a last official release date after January 2015. Based on these criteria, we created a benchmark base of 50 projects that amount to 4.55 MLLOC. We assume that this is a fair basis for comparison because this comprehensive analysis reflects common understanding and application of the design best practices. Besides, we also conducted outlier detection to eliminate measurements that are not within a normal distribution and may thereby distort the results.

6 CONCLUSION AND FUTURE WORK

From a general point of view, this paper contributes to the research area of improving software quality, in particular object-oriented design quality. Design flaws that evolved due to time pressure and quick fixes rather than long-term solutions increase the complexity of further product development. Known as design debt, this metaphor expresses the effort required to clean up the design in an attempt to limit development and maintenance risks. Studies of design debt have shown that the detection of design flaws, which are valuable to developers, has been well addressed [29]. Nevertheless, making the most efficient improvement decisions for paying off design debt with limited resources is still an open area of research.

Our improvement portfolio matrix can be seen as a tool that provides exactly that support for making proper decisions about design improvement. It therefore combines the importance of design best practice with benchmark values and can derive an overview of design flaws for a particular project. This already helps developers in focusing on the most important issues first. In addition, the portfolio matrix allows flexibility by allowing up- or downgrading practices depending on the project context. This should not be a comfortable, ad-hoc decision made to bypass painful changes but rather must be considered from a strategic point of view, always aligned with business and product goals.

From a practice-oriented view, the portfolio matrix represents the missing stone after design measurement. While developers may be not as interested in an overview of design flaws, they are concerned about refactoring the matching problems in the source code. This process can now be perfectly implemented by first identifying design rule violations with MUSE, then publishing the findings on SonarQube, and finally mapping the measurement to the portfolio matrix. As a result, it is possible to begin an investigation from the portfolio matrix and the dig down to the source code.

As briefly mentioned in Section 4.3, future work will concentrate on understanding the effort required to refactor the different kinds of design best practice violations. This should reveal a clearer picture of the required investment effort and is an additional parameter to drive decisions. Besides, an estimation of effort in terms of person days would be a lever to monetarize design debt for decision-makers.

A validation of the approach in cooperation with one of the three studied projects is another avenue for future work. In fact, we are interested in which importance values the collaborating team would select for the project when using the framework we proposed in this work. Based on the resulting portfolio matrix, a statement and general perception of the recommended improvements would underline or reject the suitability of our approach. Moreover, our approach now compares a project against a benchmark suite and derives a quality assessment. However, it is not obvious whether this relative quality assessment reflects the *real* quality of the product. Therefore, the external views of independent experts should be taken into consideration.

Finally, aggregating design best practices to a higher-level design problem is on our research agenda. Indeed, in previous work we proposed a design quality model for measuring and assessing design principles such as *information hiding*, the *single responsibility principle* or *don't repeat yourself* [24]. A current research study further investigates this direction so that we can better understand the relation between design best practices and design principles. Based on these results, we plan to use the portfolio matrix to provide improvements not on the fine-grained level of practices but rather on the coarse-grained level of design principles. In other words, we would want to show a project team that the product performs well in, for example, *information hiding*, while having deficits in regards to the single responsibility principle.

A APPENDIX

A.1 Benchmark Base

Table 6 lists all 50 open-source projects that were analyzed to build the benchmark base.

Table 6: Benchmark suite

Project	Version	Release Date	LLOC
Vuze	5740	2016-11-01	624,049
Weka	3.9.1	2016-12-01	319,356
Hibernate	5.2.5	2016-11-24	284,550
JasperReports	6.4.0	2016-10-03	275,346
Cassandra	3.9	2016-09-29	270,420
Infoglue	3.5.1.0	2016-11-01	236,478
SVNKit	1.8.14	2016-10-10	211,579
HSQLDB	2.3.4	2016-05-16	171,018
H2	1.4.193	2016-11-01	137,220
Findbugs	3.0.1	2015-06-06	113,139
JEdit	5.3.1	2015-10-01	112,385
RSyntaxTextArea	2.6.1	2017-02-06	111,838
Ant	1.9.7	2016-04-01	108,146
JGit	4.6.0	2016-09-21	103,278
Apache jMeter	3.1	2016-11-20	102,252
BlueJ	3.1.6	2016-02-01	99,314
Lucene	6.3.0	2016-11-08	96,514
SweetHome3D	5.4	2017-01-01	94,361
TuxGuitar	1.4	2016-12-01	93,793
Pixelitor	4.0.2	2016-05-03	84,149
Apache Mahout	0.12.3	2016-09-07	74,699
Hudson	3.3.2	2016-02-15	74,054
GanttProject	2.8.1	2016-08-05	59,354
Jersey	2.25	2016-12-08	57,660
Apache Tika-Core	1.13	2016-05-09	53,030
Log4j	2.8	2017-02-01	46,332
FitNesse	20161106	2016-11-01	39,572
Restlet	2.3.9	2016-12-19	36,837
Checkstyle	7.3	2016-11-03	34,704
jHotDraw	8	2016-06-01	32,710
spring-core	4.3.4	2016-11-07	32,521
testng	6.10	2016-11-29	31,310
joda-time	2.9.6	2016-12-19	28,718
HttpClient	4.5.2	2016-02-26	28,717

Project	Version	Release Date	LLOC
Apache Comm. Collection	4.1	2015-11-01	27,820
Apache Comm. Lang	3.5	2016-10-01	26,578
Zxing	3.3.30	2016-09-16	24,257
Xstream	1.4.9	2016-03-16	22,161
PMD	5.5.4	2017-01-01	21,932
jackson-core	2.8.5	2016-11-14	21,471
myBatis	3.4.2	2017-01-01	20,524
M2Eclipse	1.8	2016-06-01	18,682
Apache Shiro	1.3.2	2016-09-09	18,575
jBehave	4.1	2016-12-01	16,249
Mockito	2.7.1	2017-02-01	14,127
JUnit	5.0.0M3	2016-11-01	12,043
Cobertura	2.2.1	2015-02-26	10,157
Gson	2.8.0	2016-10-27	9,624
slf4j	1.7.22	2016-12-13	7,811
Retrofit	2.1.0	2016-06-15	3,713

REFERENCES

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

[2] Lionel C. Briand, Jürgen Wüst, John W. Daly, and Victor D. Porter. 2000. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software* 51, 3 (2000), 245–273. DOI: https://doi.org/10.1016/S0164-1212(99)00102-8

[3] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, and Nico Zazworka. 2010. Managing Technical Debt in Software-reliant Systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER '10)*. ACM, New York, NY, USA, 47–52. DOI: https://doi.org/10.1145/1882362.1882373

[4] William Brown, Raphael Malveau, Hays McCormick, and Thomas Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* Wiley and Sons, New York, NY, USA.

[5] Johannes Bräuer, Reinhold Plösch, Matthias Saft, and Christian Körner. 2017. A Survey on the Importance of Object-Oriented Design Best Practices. In *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications, SEAA '17.* Vienna, Austria, accepted for publication.

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

[7] José P. Correia and Joost Visser. 2008. Benchmarking Technical Quality of Software Products. In *15th Working Conference on Reverse Engineering, 2008. WCRE '08.* 297–300. DOI: https://doi.org/10.1109/WCRE.2008.16

[8] Ward Cunningham. 1992. The WyCash Portfolio Management System. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum) (OOPSLA '92)*. ACM, New York, NY, USA, 29–30. DOI: https://doi.org/10.1145/157709.157715

[9] Eduardo Figueiredo, Claudio Sant'Anna, Alessandro Garcia, and Carlos Lucena. 2012. Applying and evaluating concern-sensitive design heuristics. *Journal of Systems and Software* 85, 2 (2012), 227–243. DOI: https://doi.org/10.1016/j.jss.2011.09.060

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

[11] Samarthyam Ganesh, Tushar Sharma, and Girish Suryanarayana. 2013. Towards a Principle-based Classification of Structural Design Smells. *Journal of Object Technology* 12, 2 (2013), 1–29.

[12] Suryanarayana Girish, Samarthyam Ganesh, and Sharma Tushar. 2014. *Refactoring for Software Design Smells - Managing Technical Debt* (1 ed.). Morgan Kaufmann. http://www.designsmells.com/resources.php

[13] Harald Gruber, Reinhold Plösch, and Matthias Saft. 2010. On the Validity of Benchmarking for Evaluating Code Quality. In *Proceedings of the Joined International Conferences on Software Measurement IWSM/MetriKon/Mensura 2010.* Shaker Verlag, Aachen.

[14] Donald C. Hambrick, Ian C. MacMillan, and Diana L. Day. 1982. Strategic Attributes and Performance in the BCG Matrix-A PIMS-Based Analysis of Industrial Product Businesses1. *Academy of Management Journal* 25, 3 (1982), 510–531. DOI: https://doi.org/10.2307/256077

[15] Salima Hassaine, Foutse Khomh, Yann-Gaël Guéhenéuc, and Sylvie Hamel. 2010. IDS: An Immune-Inspired Approach for the Detection of Software Design Smells. In *2010 Seventh International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 343–348. DOI: https://doi.org/10.1109/QUATIC.2010.61

[16] David C. Hoaglin and Boris Iglewicz. 1987. Fine-Tuning Some Resistant Rules for Outlier Labeling. *J. Amer. Statist. Assoc.* 82, 400 (1987), 1147–1149. DOI: https://doi.org/10.2307/2289392

[17] Michael Kläs, Jens Heidrich, Jürgen Münch, and Adam Trendowicz. 2009. CQML Scheme: A Classification Scheme for Comprehensive Quality Model Landscapes. In *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications, SEAA '09.* Patras, Greece, 243–250. DOI: https://doi.org/10.1109/SEAA.2009.88

[18] Jean L. Letouzey and Thierry Coq. 2010. The SQALE Analysis Model: An Analysis Model Compliant with the Representation Condition for Assessing the Quality of Software Source Code. In *2010 Second International Conference on Advances in System Testing and Validation Lifecycle.* Nice, France, 43–48. DOI: https://doi.org/10.1109/VALID.2010.31

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

[20] Radu Marinescu. 2012. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development* 56, 5 (2012), 9:1–13. DOI: https://doi.org/10.1147/JRD.2012.2204512

[21] Robert C. Martin. 2003. *Agile software development: Principles, Patterns and Practices.* Pearson Education, Upper Saddle River, NJ, USA.

[22] Alois Mayr, Reinhold Plösch, and Christian Körner. 2014. A Benchmarking-Based Model for Technical Debt Calculation. In *Proceedings of the 14th International Conference on Quality Software (QSIC 2014).* IEEE, Dallas, TX, USA, 305–314. DOI: https://doi.org/10.1109/QSIC.2014.35

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

[24] Reinhold Plösch, Johannes Bräuer, Christian Körner, and Matthias Saft. 2016. Measuring, Assessing and Improving Software Quality based on Object-Oriented Design Principles. *Open Computer Science* 6, 1 (2016). DOI: https://doi.org/10.1515/comp-2016-0016

[25] Reinhold Plösch, Johannes Bräuer, Christian Körner, and Matthias Saft. 2016. MUSE - Framework for Measuring Object-Oriented Design. *Journal of Object Technology* 15, 4 (2016), 2:1–29.

[26] Roger S. Pressman and Bruce R. Maxim. 2014. *Software Engineering: A Practitioner's Approach* (8th ed.). Mcgraw-Hill Education - Europe, New York, NY.

[27] Arthur J. Riel. 1996. *Object-Oriented Design Heuristics* (1st ed.). Addison-Wesley Longman Publishing, Boston, MA, USA.

[28] Ganesh Samarthyam, Girish Suryanarayana, Toshi Sharma, and Swastik Gupta. 2013. MIDAS: A design quality assessment method for industrial software. In *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013).* San Francisco, CA, USA, 911–920. DOI: https://doi.org/10.1109/ICSE.2013.6606640

[29] Jan Schumacher, Nico Zazworka, Forrest Shull, Carolyn Seaman, and Michele Shaw. 2010. Building Empirical Support for Automated Code Smell Detection. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10).* ACM, New York, NY, USA, 8:1–8:10. DOI: https://doi.org/10.1145/1852786.1852797

[30] Frank Simon, Olaf Seng, and Thomas Mohaupt. 2006. *Code-Quality-Management: technische Qualität industrieller Softwaresysteme transparent und vergleichbar gemacht.* dpunkt-Verlag.

[31] Ramanath Subramanyam and Mayuram S. Krishnan. 2003. Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering* 29, 4 (2003), 297–310. DOI: https://doi.org/10.1109/TSE.2003.1191795

[32] Adrian Trifu and Radu Marinescu. 2005. Diagnosing design problems in object oriented systems. In *12th Working Conference on Reverse Engineering.* IEEE, 10 pp. DOI: https://doi.org/10.1109/WCRE.2005.15

[33] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering.* Springer Berlin Heidelberg, Berlin, Heidelberg.

[34] Nico Zazworka, Carolyn Seaman, and Forrest Shull. 2011. Prioritizing Design Debt Investment Opportunities. In *Proceedings of the 2Nd Workshop on Managing Technical Debt (MTD '11).* ACM, New York, NY, USA, 39–42. DOI: https://doi.org/10.1145/1985362.1985372

[35] Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn Seaman. 2011. Investigating the Impact of Design Debt on Software Quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt (MTD '11).* ACM, New York, NY, USA, 17–23. DOI: https://doi.org/10.1145/1985362.1985366