A Survey on the Importance of Object-oriented Design Best Practices

Johannes Bräuer and Reinhold Plösch
Department of Business Informatics – Software Engineering
Johannes Kepler University Linz
Linz, Austria
[johannes.braeuer, reinhold.ploesch]@jku.at

Matthias Saft and Christian Körner
Corporate Technology
Siemens AG
Munich, Germany
[matthias.saft, christian.koerner]@siemens.com

***Abstract*—To measure object-oriented design quality, metric-based approaches have been established. These have then been enhanced by identifying design smells in code. While these approaches are useful for identifying hot spots that should be refactored, they are still too vague to sufficiently guide software developers to implement improvements. This is why our work focuses on measuring the compliance of source code with object-oriented design best practices. These design best practices were systematically derived from the literature and can be mapped to design principles, which can help reveal fundamental object-oriented design issues in a software product. Despite the successful applications of this approach in industrial and open source projects, there is little accepted knowledge about the importance of various design best practices. Consequently, this paper shows the result of an online survey aimed at identifying the importance of 49 design best practices on design quality. In total, 214 people participated in the survey, resulting in an average of 138 opinions for each practice. Based on these opinions, five very important, 21 important, 12 moderately important and 11 unimportant design best practices could be derived. This information about importance helps managing design improvements in a focused way.**

Keywords—design best practices; design principles; design quality; software quality.

I. Introduction

Software products with good object-oriented design are flexible, reusable and maintainable [1]. In the past, software metrics were used to express the compliance of source code with object-oriented design aspects [2], [3]. Nevertheless, it has been recognised that metrics are vague for dealing with the complexity of design and for driving concrete improvements [4]. Based on this cognition, the idea of identifying code or design smells in source code has been established and is used to find hot spots that threaten the flexibility, reusability and maintainability of software products [5].

Despite good progress in localising design flaws based on the identification of design smells, these design smells are still too fine-grained to conclude a design assessment. Hence, a more recent approach has been published by Samarthyam et al. [6], which picks up the idea of measuring design principles as part of a design assessment. This approach is called MIDAS and refers to the skills and knowledge of experts who manually assess the adherence of design principles. By using this more abstract view of design quality, a better understanding of issues and problem sources can be achieved. However, to standardise these manual investigations, a model for design quality should be used that is currently not available [6].

In order to bridge this gap of the missing object-oriented design quality model, this work follows the approach of identifying violations of object-oriented design best practices directly in source code. In the remainder of this article, we use shorter-term design best practices, while always referring to object-oriented design best practices. Generally, 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 [7]. MUSE contains a set of 67 design best practices – design rules – for the programming languages Java, C# and C++. Most of the rules cover all three programming languages since they are related to object-oriented concepts rather than language features.

MUSE has passed various accuracy checks and has been applied in industrial as well as open-source projects [7]. Users are interested in the measurement results, which are compiled as a list of violations of design best practices with an exact location for each violation. Nevertheless, up to now, we could not recommend which violations should be addressed first based on their importance. Consequently, we decided to conduct a survey to gather data that allow a more differentiated view of the importance of our design best practices. This set of 67 rules has been reduced to 49, since future work will focus on Java, which is covered by this selection of these 49 rules.

The remainder of this article is structured as follows. In the next section, related work is discussed. Then, our novel approach for design quality improvement is briefly explained before showing the research design in Section IV. An overview of the survey is given in Section V. Section VI analyses the results and derives the importance of the 49 design best practices. Finally, the paper draws a conclusion from the results and provides an avenue for future work.

II. Related Work

To measure object-oriented design quality, the literature distinguishes different approaches. Metric-based approaches such as QMOOD [3] and the metric suite provided by

Chidamber and Kemerer [2] try to measure design quality by means of, e.g., cohesion, coupling and inheritance-related metrics. The effort to measure these design aspects is typically low. Empirical studies indicate that some of the 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) [8], [9]. Additionally, it is possible to aggregate these metrics into an overall quality index as proposed by QMOOD and to compare the results of different projects or project versions. Nevertheless, metric-based approaches fail to support developers since the semantic relation between a metric value and source code is difficult to grasp.

Riel coins the term object-oriented design heuristics [10]. In his book, he takes a constructive approach and formulates design heuristics in a way that a software developer can use them as templates when designing or implementing software [10]. Riel associates his design heuristics with more general language-oriented design topics such as inheritance relationships, association relationships and the relationships between classes and objects. The design heuristics provided by Riel are described with a pattern language, but give no hints on the impact of these heuristics on general quality attributes or on their importance.

Another set of approaches tries to identify so-called design smells or code smells [5], [11], [12]. The work on design smells is influenced by the design heuristics of Riel [10] as well as the research on refactoring and anti-patterns by Fowler et al. [13] and Brown et al. [14], respectively. These design smells, for instance, *data clumps* or *feature envy* [13], semantically express a design problem and therefore try to make it tangible for software developers. From our point of view, this is an improvement over metric-based approaches. However, typical measuring approaches (e.g., [5], [15]) for design smells are also metric-based with the difference of not relying on a single metric but on a combination thereof.

DÉCOR is another smell-based approach that is not based on metrics but rather on a method and language that allows to specify design smells [16]. The underlying approach provides all the necessary steps to define a so-called detection technique. Specifically, step three of DÉCOR is the translation of a textual bad smell definition into algorithms that can be applied to detect it. To validate the method, the authors instantiate DÉCOR and test the detection techniques on four bad smells [16].

Ganesh et al. [17] collect and categorise design smells by using a pattern language; in other words, they focus on semantically grasping the design problem without providing any hints on how to measure design smells [18]. This work contains a large number of structural design smells at the micro-architecture level with an emphasis on object-oriented design principles. In addition to collecting and describing the design smells, Ganesh et al. relate them to the object-oriented design principles abstraction, encapsulation, modularity and hierarchy.

Focusing on design aspects rather than the mere source code level provides a better lever for enhancing software quality. While some scientific work investigates the impact of code and design smells on the quality attribute maintainability [19], little work systematically investigates and validates the relation between design smells and (object-oriented) design quality in general. Furthermore, there are no systematic investigations that try to answer questions related to the importance of design best practices or design smells apart from focused work in the realm of design metrics. In fact, Lozano et al. [20] point out that there is little evidence of the negative impact of design smells.

III. Novel Approach for Design Quality Improvement

As mentioned above, the community is missing a reference model for design quality [6]. In the long run, our work focuses on this gap but using a different approach compared with related work in the past. In contrast to using 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*. 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. [18], who cluster design problems based on *abstraction*, *encapsulation* or *modularisation*.

A list of 32 design principles has been systematically identified and important ones have been derived based on a survey [21]. 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 shown in Figure 1. Based on this example, the *information hiding principle* is affected (threatened) by, e.g., 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 shown example is just an abstract view of measuring design principles by means of design best practices, a more elaborated discussion is provided in Plösch et al. [21], which explains the underlying meta-model and its applicability.

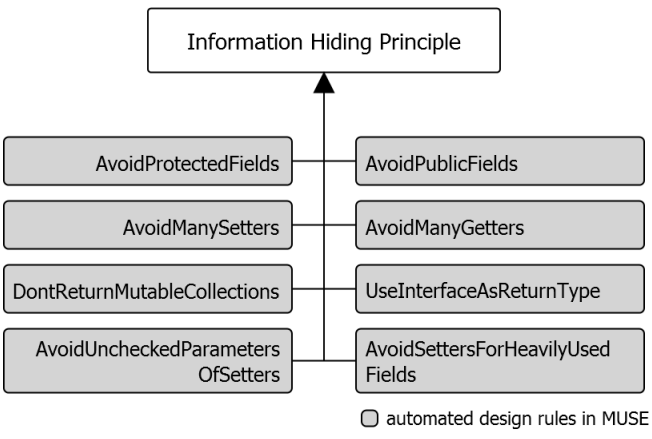

Fig. 1. Design principle and its design best practices

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 that analyses Java, C# or C++ source code and finds the violations of these design best practices [7]. As already mentioned in the Introduction, this article concentrates on Java measures. 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 the developer can start investigating the identified issues.

IV. Research Design – Survey Research

This section concentrates on the research direction by highlighting the research goal of this work and explaining the design of the conducted survey.

A. Research Question

MUSE can be applied to manage design improvements for a software product. However, there remains the limitation that users cannot be guided to the most important and critical violations. In order to provide a default setting for the importance of the design best practices in MUSE and to support managers in prioritising improvement actions, we derived the following research question at the beginning of the investigation: *RQ: How important are various design best practices for achieving good object-oriented design?*

To answer this question, a broad view of the topic is required since object-oriented design is context-dependent and gathering many opinions can compensate for the impact of the context. In other words, conducting, e.g., a focus group discussion with a relatively small set of participants may run into the problem of non-generalisable findings. The same problems arise for case studies or similar research methods. Therefore, we choose a survey-based research approach to answer the research question.

B. Survey Design

The study objects in this survey are the 49 design best practices suitable for Java. Each of these practices has a unique name and a description that clarifies the intention and provides additional details to understand the design best practice. In order to guarantee that the name and description are well understood, a pre-test with eight experienced Master's students was conducted. In this pre-test, the students had to interpret the name and description of each design best practice by writing a code snippet that shows a violation of this design best practice. Problems that occurred during this pre-test were collected and discussed. As a result, improvements to the name and descriptions could be made. Finally, a cross check ensured that various notations and terms were consistently used across all descriptions.

After this pre-test, the questionnaire was designed based on a guideline [22] and the recommendations given in [23]. The questionnaire consists of four question blocks and the end page:

Personal Information: The first question block asked the participants about the affiliation, the job position and for how many years they have been working in the current job position.

Background and Experience: The second question block gathered information on programming skills and expertise in object-oriented software design. Therefore, one question asked about the business domain in which the participants are working and another about the level of experience in Java, C#, and C++. To collect data about the design expertise, the participants had to self-assess their design expertise and how they learned object-oriented design.

Assessment Blocks: Each assessment block allows assessing the importance of seven randomly chosen design best practices with a five-point scale from 1 (very unimportant) to 5 (very important). The answers were mandatory, meaning that it was not possible to skip a design best practice. After assessing the first block, the questionnaire continued with the next block of seven randomly chosen design best practices and so forth.

After each individual assessment block, we explicitly highlighted the current progress and showed appreciation for the already invested effort. In addition, we motivated the participants to continue. If the participant had no more time, it was possible to directly go to the open question block close to the end of the survey. By providing this kind of freedom and some distraction from the monotonous task of assessing the importance, we tried to keep the motivation high to get as many assessments as possible.

Open Question Block: In this block, the questionnaire invited the participants to note object-oriented design aspects not covered by the questionnaire. Additionally, feedback could be posted here.

End of Survey: Finally, the last page notified about the successful submission of the questionnaire. Additionally, the participants were invited to subscribe for a summary of the survey results.

C. Participant Recruitment

We followed two approaches for recruiting participants. First, we sent personalised emails to a list of people who we know have software engineering skills in research or practice or who are leaders in software developing organisations. This survey invitation, including the survey link and access token, contained the request to distribute the participation details among colleagues and employees. Second, we recruited people from social networking and business platforms such as LinkedIn, XING and ResearchGate. For this purpose, a generic post was published in designated discussion groups and forums on these platforms. These posts explained the purpose of the survey and the benefits a participant can get in return. Additionally, the survey link and an access token were provided.

V. Survey Overview

In this section, an overview of the survey sample is provided by discussing the drop-out rate, showing demographics and presenting the software engineering and oo-programming skills of the participants.

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

A. General Overview

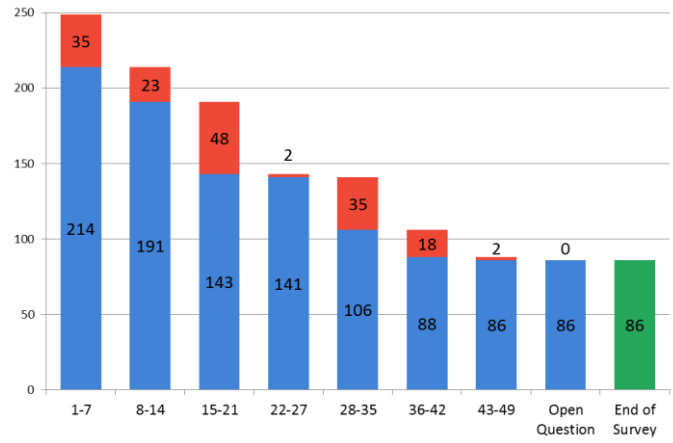

Fig. 2. Drop-outs among question blocks

The survey was available from 26th October until 21st November 2016. In total, 294 individuals interacted with the survey, meaning that they followed the invitation link and hit the start button on the welcome page. Then, there was a drop-out of 28 people on the personal information page and 17 people on the background & experience page. A large proportion (35) left during the assessment of the first seven design best practices, resulting in 214 successfully finished questionnaires.

To our surprise, 52.9% of the 214 participants were recruited by our posts on the social networking platforms. Although the posts were placed in topic-related discussion groups and forums, the number of people who were willing to contribute in this survey is high, at least compared with previous surveys we conducted.

As mentioned above, the design of the survey provided some degree of freedom for the participants and they could decide to end the survey after each assessment block. The analysis of the drop-out rate regarding this degree of freedom is shown in Figure 2. According to the graph and as mentioned before, 214 participants expressed their opinion for at least seven design best practices; afterwards, 23 quit the survey. The drop-out rates between the assessment of the second and third as well as the fourth and fifth blocks are conspicuous but explainable. At these points, the questionnaire presented an image and a short text that showed appreciation for the already invested effort. In addition, the participants had the chance to skip the rest of the survey at these stages. Nevertheless, 86 participants finished the entire survey, meaning that they continued until the end of the survey.

B. Personal Information

The first question in the personal information block asked about the current affiliation of the participants. Figure 3 shows the distribution of the participants among academic organisations, self-employment and companies differentiated by the number of employees.

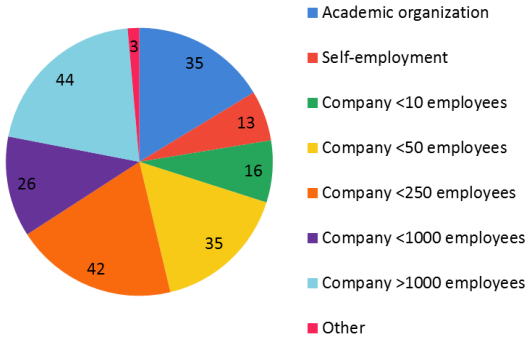

Fig. 3. Distribution of affiliations

According to this diagram, there is a well-balanced distribution between all affiliations, ranging from 13 self-employed participants to 44 participants working at a company with more than 1,000 employees. However, there is strong participation from industry with 176 people with a practical point of view rather than a theoretical one. The three people who selected the option *other* are currently not working or employed at a governmental organisation with more than 25,000 clerks.

The distribution of the participants among the job roles shows that many software engineers and architects expressed their opinion. In more detail, 93 software engineers and 45 software architects – together they represent 64% of the total quantity – successfully completed the survey. Figure 4 depicts the number of participants for each job position. The 12 participants that selected the option *other* noted the following roles: trainer, teaching, technical director, managing director (technical), CTO, director of institute, founder, tech lead, software craftsman and systems engineer.

C. Background and Experience

The second question block focused on background information and the expertise of the participants in programming languages. The analysis of the domains in which the participants are working highlights two disciplines leading the field. The top business domain is the development of web/service-oriented systems (66) followed by business

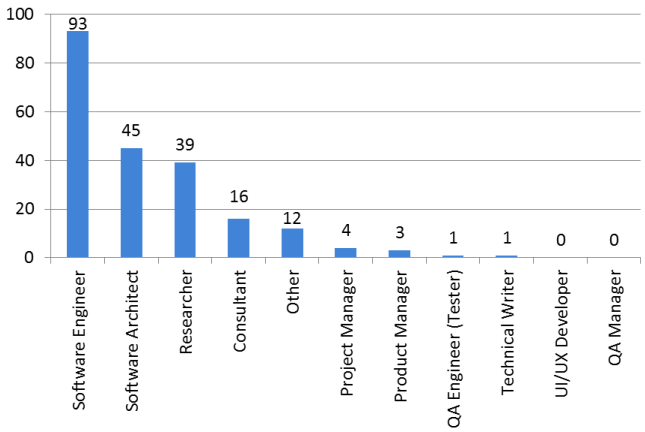

Fig. 4. Distribution of job roles

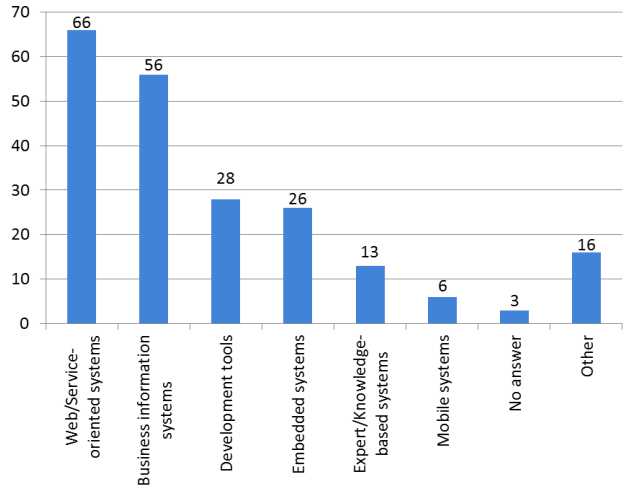

Fig. 5. Distribution of business domains

information systems (56). Figure 5 shows the distribution among the other domains. For the sake of completeness, the 16 participants who selected the option *other* are developing user interfaces, desktop applications, CAD/CAM systems, human–computer interaction, language implementation, information systems, virtual machines, compilers, GUI systems, finance, CAD systems, healthcare, nuclear reactors and manufacturing applications.

In order to filter the answers, we asked the participants to assess their self-appraisal regarding their expertise in a particular object-oriented programming language. The data analysis shows that many participants have top and good experience in Java (125), while C# (80) and C/C++ (84) are less dominant. Nevertheless, there is still a representative group of C# and C/C++ engineers with top and good experience therein. For each participant, we consider the highest experience level regardless of the programming language. As shown there, one participant had no experience in any of the three languages but good experience in object-oriented PHP. Figure 6 depicts the total number of participants with top, good, moderate, some or no experience itemised by programming language.

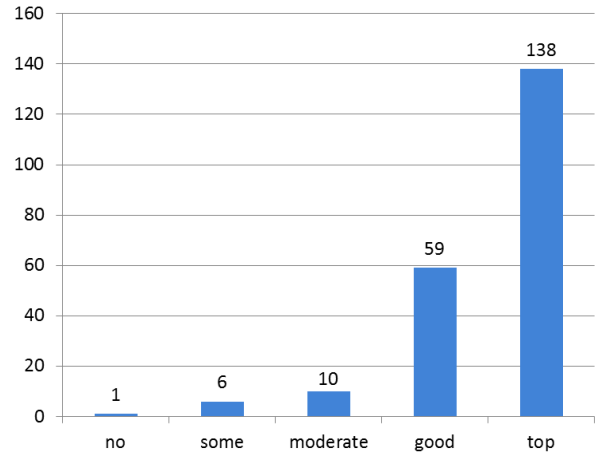

Fig. 6. Accumulated programming skills in Java, C# and C++

In addition to the programming languages, a question asked about the self-appraisal regarding object-oriented programming skills. Interestingly, a strong majority assessed this aspect with good or top expertise. This insight strengthened the representation of our database and addressed the threat to internal validity as we can show that the participants are familiar with object-oriented concepts and can assess design best practices at least to some degree.

VI. Importance of Design Best Practices

A. Individual Result

Considering the opinions of all 214 participants, we calculated an average assessment for all 49 design best practices regarding their importance. This calculation uses an equally increasing weight from very unimportant to very important, meaning that the assessment of very unimportant is weighted with 1 and very important with 5. As a high-level result, we can show that five rules are considered as very important, 21 as important, 12 as moderately important and 11 as unimportant. The exact details are depicted in Table I.

Additionally, this table shows the standard deviation and skew of answers for each design best practice. The standard deviation ranges from 0.79 to 1.24. A low standard deviation means that the participants agree in their opinion, while a high value stands for disagreement. The skew – as an indicator of the asymmetry of the answers about the mean – shows the tendency towards either very important or very unimportant. For this discussion, the skew is multiplied by -1 in order to better express its meaning. Thus, a positive value expresses an important design best practice in contrast to a negative value that represents a tendency towards unimportance.

The average of all opinions for one design best practice and the standard deviation were used to calculate an importance range for (almost) each design best practice. The only rule that does not provide a range is *AvoidDuplicates* because there is a general agreement on its very high importance. For all the other design best practices, the range is shown in the last column of Table I and this provides some degree of freedom when using the rules (e.g. assessing software design). Hints for up- or downgrading various design best practices are discussed below.

B. Analysis of Border Cases

From a broader perspective, 33 rules are classified as high and moderate. Thus, it should be discussed whether candidates are close crossing a border. Figure 7 shows the distribution of rule assessments within the high range. According to this figure, the three rules *DocumentInterfaces*, *AvoidLongParameterLists* and *UseInterfacesIfPossible*, which are marked with an h↑ importance in Table I, have an average assessment of 3.96 and are very close to the upper boundary. Consequently, these three rules are good candidates for a rise to very high.

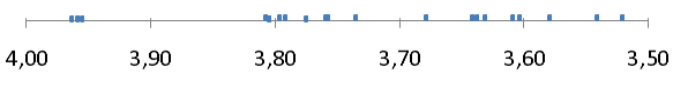

Fig. 7. Accumulated programming skills in Java, C# and C++

TABLE II. DESIGN BEST PRACTICES ORDERED BY IMPORTANCE

	Average	Default Importance	Standard Deviation	Skew * (-1)	Range
AvoidDuplicates	4.42	vh	0.79	1.74	vh
AvoidUsingSubtypesInSuper.	4.27	vh	0.94	1.11	h-vh
AvoidPackageCycles	4.27	vh	0.94	1.43	h-vh
AvoidCommandsInQueryM.	4.23	vh	0.93	0.94	h-vh
AvoidPublicFields	4.09	vh	1.09	1.22	h-vh
DocumentInterfaces	3.96	h↑	1.07	0.95	m-vh
AvoidLongParameterLists	3.96	h↑	0.85	0.82	h-vh
UseInterfacesIfPossible	3.96	h↑	1.08	0.99	m-vh
AvoidStronglyCoupledPack.	3.81	h	0.92	0.41	m-vh
AvoidNonCohesiveImpleme.	3.80	h	1.05	0.78	m-vh
AvoidUnusedClasses	3.80	h	1.06	0.76	m-vh
DontReturnUninvolvedData	3.80	h	0.92	0.72	m-vh
AvoidNonCohesivePackages	3.79	h	0.93	0.51	m-vh
DocumentPublicMethods	3.77	h	1.11	0.89	m-vh
UseCompositionNotInherita.	3.76	h	1.08	0.60	m-vh
DocumentPublicClasses	3.76	h	1.12	0.68	m-vh
AvoidPublicStaticFields	3.74	h	1.15	0.63	m-vh
AvoidDiamondInheritanceSt.	3.68	h	0.95	0.45	m-vh
AvoidLongMethods	3.64	h	1.12	0.44	m-vh
AvoidSimilarNamesForDiff.	3.64	h	1.04	0.35	m-vh
AvoidUnusedAbstractions	3.63	h	1.19	0.61	m-vh
CheckUnsuitableFunctionality	3.61	h	0.95	0.54	m-vh
AvoidSimilarAbstractions	3.60	h	0.94	0.49	m-vh
DocumentPackages	3.58	h↓	1.19	0.55	l-vh
UseInterfacesAsReturnType	3.54	h↓	1.20	0.40	l-vh
AvoidUncheckedParameter.	3.52	h↓	1.02	0.45	h-vh
AvoidSimilarNamesForSame.	3.49	m↑	1.03	0.39	m-h
CheckObjectInstantiatiosByN.	3.45	m↑	1.07	0.26	l-h
AvoidRepetitionOfPackage.	3.44	m↑	1.05	0.48	l-h
ProvideInterfaceForClass	3.34	m	1.12	0.33	l-h
AvoidRuntimeTypeIdentifica.	3.32	m	1.07	0.49	l-h
AvoidDirectObjectInstantiati.	3.32	m	1.07	0.41	l-h
CheckUnusedSupertypes	3.32	m	0.91	0.28	l-h
AbstractPackagesShouldNot.	3.31	m	1.03	0.33	l-h
DontReturnMutableCollectio.	3.31	m	1.08	0.23	l-h
AvoidMassiveCommentsInC.	3.24	m	1.20	0.19	l-h
AvoidReturningDataFromCo.	3.14	m	1.05	0.21	l-h
UseAbstractions	3.02	m↓	1.24	0.12	vl-h
CheckUsageOfNonFullyQual.	2.97	l	0.96	0.09	l-m
AvoidManySetter	2.96	l	1.05	-0.24	vl-m
AvoidHighNumberOfSubpkg.	2.92	l	0.96	0.07	vl-m
AvoidConcretePackage	2.92	l	0.90	0.07	vl-m
AvoidSettersForHeavilyUsed.	2.86	l	1.16	-0.14	vl-m
AvoidAbstractClassesWithO.	2.85	l	1.07	-0.17	vl-m
DontInstantiateImplementatio.	2.85	l	1.08	-0.20	vl-m
AvoidManyGetters	2.68	l	1.10	-0.43	vl-m
AvoidProtectedFields	2.67	l	1.11	-0.39	vl-m
CheckDegradedPackageStru.	2.62	l	0.94	-0.12	vl-m
AvoidManyTinyMethods	2.60	l	1.02	-0.50	vl-m

vh – very high, h – high, m – moderate, l – low, vl – very low

After this group of three rules, there is a gap to the next design best practices when reading Figure 7 from left to right. These design best practices are close to the centre of the assessment scale and should adhere to the high importance level. Being close to the centre means that their assessment deviates by +/-0.15 points. In other words, rules below 3.60 are candidates for a downgrade to moderate. According to Figure 7 and indicated by h↓ in Table I, the three rules *DocumentPackages*, *UseInterfacesAsReturnType* and *AvoidUncheckedParametersOfSetters* could have a moderate importance level.

Similar to the discussion on design best practices within the high range, three candidates within the moderate range could also rise to high. More specifically, the rules *AvoidSimilarNamesForSameDesignElements*, *CheckObjectInstantiations* and *AvoidRepetitionOfPackageNamesOnAPath* have an average weight above 3.40 as depicted on the scale in Figure 8 and indicated with m↑ in Table I. On the opposite side, only *UseAbstractions* represents a candidate for the low importance level. The remaining moderate design best practices are close to the centre when checking the +/-0.15-point deviation from 3.25.

C. Application for Design Improvement

Given these findings, it is now possible to guide software developers or designers when analysing MUSE measurements. Hence, those violations of design best practices considered to have very high importance on design quality should be addressed before investing effort into violations of design best practices with lower importance. Consequently, this supports decision making and ensures efficient design quality improvement from an empirical point of view.

In addition, the data from the survey help configure the importance of design best practices by providing a reasonable range within which to act. Depending on the context, a development team might decide not to use the standard value given in the column *Default Importance* in Table I. From the survey data, we can then help developers by providing a reasonable range that should be used; see the column *Range* in Table I. For example, if the project team tries to enforce proper documentation, then an importance level of *high* for the design best practice *DocumentPublicClasses* might not be demanding enough and could therefore be raised to the importance of *very high*. On the other hand, it is not advisable to choose *low* or *medium* for this rule according to our survey data regardless of the perceived importance of the development team.

An additional value of the importance level for all design best practices is the adjustability of design quality assessment approaches such as calculating design debt or a quality index. Therefore, our proposed benchmarking-based model for technical debt can be applied to derive design debt based on the MUSE rules [24]. To enhance this model, the importance level of each rule can now better clarify the non-remediation costs of

Fig. 8. Accumulated programming skills in Java, C# and C++

each rule, which is an important input factor for the calculation. Finally, the calculated design debt is useful for developers, but it seems especially useful for managers to understand the cost and revenue benefits of refactoring parts, which might not be seen as a waste of effort but rather an investment in future development.

VII. Threats to Validity

Threats to internal validity are influences that can affect the independent variable with respect to causality [25]. Thus, one issue is understanding the presented design best practices. To ensure a solid description and name for each rule, a pre-test was conducted. As a consequence of this pre-test, misleading descriptions were refined and names were adapted. Hence, it can be argued that the design best practices are well understood based on the refinements.

Another confounding variable is the perception of the participants while answering the questions. In other words, the participants' opinion about the practical implementation or relevance of design best practices could have influenced the estimation. In order to focus attention on only considering the general importance of a design best practice, a simple check-box question highlighted the purpose of the survey; the participant could not continue without marking the *"I understood"* check box.

Threats to external validity are conditions that limit the ability to generalise the results of the investigation [25]. We explicitly choose an open survey research method to foster the generalisability of the results. Facts that supported the decision were the low complexity of the questions and the low processing time, which are critical success factors [26]. Consequently, the survey could be distributed among many people, resulting in a good response rate. The analysis of demographic aspects shows that the result is derived from a diverse but experienced sample. This includes various business domains and job roles. All in all, the strong participation of practitioners could be achieved, which strengthens the ability to generalise the findings.

This discussion should consider the implications for generalising the result to other oo-programming languages. More specifically, we currently know a set of 67 design best practices (see Plösch et al. [7]) containing rules for Java, C++ and C#. Consequently, the general findings can be transferred to other programming languages but with special care; for C++, additional design features such as multiple inheritance or macros are available. For C#, there are not as many differences compared with Java.

VIII. Conclusion and Future Work

While previous research on enhancing object-oriented software design has focused on carefully identifying (smelly) source code spots characterised by design metrics, there is little evidence on the importance of individual design best practices. In this study, a survey was conducted in order to investigate this aspect. Based on the opinions of 214 participants from different software engineering fields, we derived the importance of 49 measures; five design best practices were considered to be of very high importance.

In fact, avoiding duplicates (code clones), the use of subtypes in supertypes, package cycles, commands in query methods and public fields were the design concerns considered to be very important. In other words, following these design rules in practice can enhance and foster the flexibility, reusability and maintainability of a software product. Although it is possible to oversee the compliance of these five aspects by using a checklist or cheat sheet, monitoring the evolution of the software design becomes challenging when additionally considering at least the 21 important rules identified from the survey.

MUSE was developed to automatically identify violations of design best practices in source code [7]. The provided measurement data can be used to discuss upcoming design decisions and control enhancements. These tasks are now supported by the contribution of this survey because software engineers and architects can concentrate on very important (or important) design best practices first, knowing that the professional knowledge of a large number of survey participants leads to the categorisation of these design best practices to be of high or very high importance.

Having a set of 49 measurable design best practices for Java is a valuable support for practitioners in software engineering. Nevertheless, the completeness of our design best practice collection is not known. Consequently, further work should focus on finding missing design best practices.

Strengthened by the comments from the survey, we will continue to investigate aspects for measuring and assessing design principles such as the *information hiding principle*, *open-close principle* or *single responsibility principle*. Software engineers and architects know design principles and are concerned about them. However, we are not aware of any method or tool that allows measuring design principles in source code. Therefore, we defined a design quality model as briefly addressed in Section III. First insights into this approach are given in Bräuer [27] and Plösch et al. [21], but a comprehensive validation is missing.

For this purpose, we want to understand the coverage of design principles due to design best practices as depicted in Figure I. The importance data gained from this survey are a first input. However, for a deeper understanding, we are currently setting up a number of focus groups to identify the completeness of our design best practices with regard to the design principles. Moreover, this investigation should reveal white spots in measuring design principles and object-oriented design in general.

References

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

[2] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[3] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, 2002.

[4] R. Marinescu, "Measurement and quality in object-oriented design," in *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05*, 2005, pp. 701–704.

[5] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," presented at the Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on, 2004, pp. 350–359.

[6] G. Samarthyam, G. Suryanarayana, T. Sharma, and S. Gupta, "MIDAS: A design quality assessment method for industrial software," in *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*, 2013, pp. 911–920.

[7] R. Plösch, J. Bräuer, C. Körner, and M. Saft, "MUSE - Framework for Measuring Object-Oriented Design," *J. Object Technol.*, vol. 15, no. 4, pp. 2:1-29, 2016.

[8] L. C. Briand, J. Wüst, J. W. Daly, and D. Victor Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *J. Syst. Softw.*, vol. 51, no. 3, pp. 245–273, 2000.

[9] R. Subramanyam and M. S. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects," *IEEE Trans. Softw. Eng.*, vol. 29, no. 4, pp. 297–310, 2003.

[10] A. J. Riel, *Object-Oriented Design Heuristics*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

[11] S. Hassaine, F. Khomh, Y.-G. Guéhenéuc, and S. Hamel, "IDS: An Immune-Inspired Approach for the Detection of Software Design Smells," presented at the Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the, 2010, pp. 343–348.

[12] E. Figueiredo, C. Sant'Anna, A. Garcia, and C. Lucena, "Applying and evaluating concern-sensitive design heuristics," *J. Syst. Softw.*, vol. 85, no. 2, pp. 227–243, 2012.

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

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

[15] A. Trifu and R. Marinescu, "Diagnosing design problems in object oriented systems," presented at the Reverse Engineering, 12th Working Conference on, 2005, p. 10

[16] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, 2010.

[17] S. Ganesh, T. Sharma, and G. Suryanarayana, "Towards a Principle-based Classification of Structural Design Smells," *Journal of Object Technology*, vol. 12, no. 2, pp. 1–29, 2013.

[18] S. Girish, S. Ganesh, and S. Tushar, *Refactoring for Software Design Smells - Managing Technical Debt*, 1st ed. Morgan Kaufmann, 2014.

[19] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 306–315.

[20] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Assessing the impact of bad smells using historical information," presented at the Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting, 2007, pp. 31–34.

[21] R. Plösch, J. Bräuer, C. Körner, and M. Saft, "Measuring, Assessing and Improving Software Quality based on Object-Oriented Design Principles," *Open Comput. Sci.*, vol. 6, no. 1, 2016.

[22] B. A. Kitchenham and S. L. Pfleeger, "Personal Opinion Surveys," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds. Springer London, 2008, pp. 63–92.

[23] F. Keusch, "How to Increase Response Rates in List-Based Web Survey Samples," *Soc. Sci. Comput. Rev.*, p. 894439311409709, 2011.

[24] A. Mayr, R. Plosch, and C. Korner, "A Benchmarking-Based Model for Technical Debt Calculation," in *Proceedings of the 14th International Conference on Quality Software (QSIC 2014)*, Dallas, Texas, 2014, pp. 305–314.

[25] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[26] E. Smith, R. Loftin, E. Murphy-Hill, C. Bird, and T. Zimmermann, "Improving developer participation rates in surveys," in *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, 2013, pp. 89–92.

[27] J. Braeuer, "Measuring Object-Oriented Design Principles," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 882–885.