# What to Fix? Distinguishing between design and non-design rules in automated tools

Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord
Carnegie Mellon University Software Engineering Institute
Pittsburgh, PA
Email: {nernst,sbellomo,ozkaya,rn}@sei.cmu.edu

***Abstract*—Design problems, frequently the result of optimizing for delivery speed, are a critical part of long-term software costs. Automatically detecting such design issues is a high priority for software practitioners. Software quality tools promise automatic detection of common software quality rule violations. However, since these tools bundle a number of rules, including rules for code quality, it is hard for users to understand which rules identify design issues in particular. Research has focused on comparing these tools on open source projects, but these comparisons have not looked at whether the rules were relevant to design. We conducted an empirical study using a structured categorization approach, and manually classified 466 software quality rules from three industry tools—CAST, SonarQube, and NDepend. We found that most of these rules were easily labeled as either non-design (55%) or design (19%). The remainder (26%) resulted in disagreements among the labelers. Our results are a first step in formalizing a definition of a design rule, to support automatic detection.**

***Index Terms*—Software quality, software design, software cost**

## I. Introduction

Static analysis tools evaluate software quality using rules that cover languages, quality characteristics, and paradigms [1]. Software quality rules have accumulated as practitioners gradually recognize code 'smells' and poor practices. The first tool to automate rule-checking was the C language tool `lint` in 1979 [2]. Static analysis tools have traditionally focused on what we might call the *code-level*, rather than design. For instance, Johnson's initial description of `lint` mentions type rules, portability restrictions, and "a number of wasteful, or error prone, constructions which nevertheless are, strictly speaking, legal" [2].

Increasingly, quality rules are targeting design problems, such as paradigm violations or architecture pattern violations (e.g., the work of Aniche et al. on MVC frameworks [3]). This is because design problems are often more significant than coding errors for long-term software maintenance costs. This view is supported by the results of recent survey and interview [4] and issue tracker analysis [5] studies, which found that syntax and coding problems are rarely important sources of these long-term costs; instead, poor design choices lead to accumulating costs and technical debt.

Static analysis tools often generate many false positives, leading developers to ignore the results [6], [1], [7]. One potential improvement to this problem is to separate *design rules* from other rules. We examine the software quality rules of three typical tools to understand the extent to which their quality rules are design-related. This raises the question of what we mean by *design*, a thorny question in software engineering research [8, p.14]. We use an extensional definition [9, ch 1.1] of design by creating a design rule classification rubric, using rater agreement on classification labels as our metric. Design is clearly more than what the (imperfect) rubric suggests. Most importantly, our rubric is limited to the rules we used as input, and each rater's understanding of design. In this sense, the rubric interprets design as a collection of automatable conformance rules.

Our contributions and findings in this work include:

- A classification rubric for evaluating design rules.
- Tools do have rules that check for design quality. 19% of the rules we examined were design-related.
- Rules included examples of complex design concepts, such as design pattern conformance and run-time quality improvements. 68% of the rules that were labeled as design rules were examples of such rules.

## II. Background

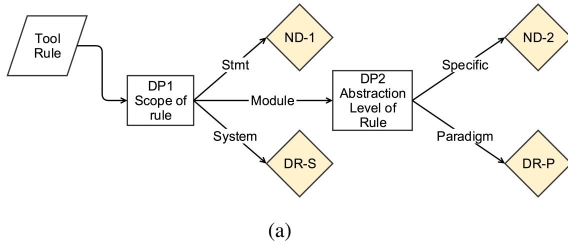
Tools that incorporate design analysis ideally provide reliable, automated, repeatable results to address these goals:

- Find poor architectural decisions and shortcuts and identify refactoring opportunities [4], [10].
- Understand when payoff is economically justified [11].
- Find increased numbers of {defects bugs churn} above baseline (hotspots) [12].
- Understand the trends and rate of change in key indicators (e.g. lines of code, test coverage, or rule violations) [13].
- Provide traceability across architectural tiers, frameworks, and languages [3].

### A. Tool and Rule Selection

We chose rule sets from three commercial software quality management tools that have stated they have capabilities to detect design: NDepend[1], SonarQube[2], and CAST[3], and that we have access to (for reasons of licensing and installability). While the tools provide broader capabilities for quality management, we narrowly focused on their quality measures

[1]https://blog.ndepend.com/technical-debt-avoid-ndepend/
[2]https://blog.sonarsource.com/evaluate-your-technical-debt-with-sonar/
[3]http://www.castsoftware.com/research-labs/technical-debt




(a)

Fig. 1. Final version of Design Rule Classification Rubric

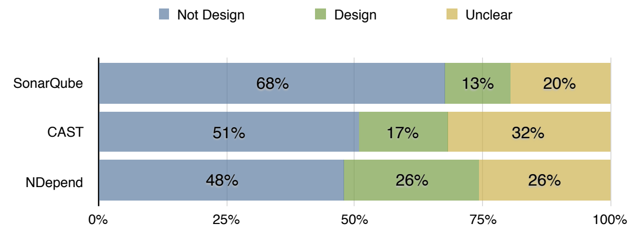



Fig. 2. Categories of rater agreement, normalized across tools.

and rules for static analysis of code. We focused on Java and .Net rules and rules that the tool documentation stated applied generically to all code under analysis. Our analysis of the rules in no way implies endorsement or critique of the tool itself.

### B. *Rubric Creation and Refinement*

We created our initial design rules classification rubric based on a taxonomy of quality analysis, empirical data collected in our previous studies [4], [5], and example rules extracted from the three tools. The rubric is also motivated by established architecture principles, such as assessing the scope of an issue as local, non-local and architectural [8].

We first created a simple definition of a design rule rubric, then iterated on it with examples from interviews and survey responses to 'test' how well it handled these cases. The input for the rubric is a single rule from one of the example rule sets. The labeler (person classifying the rule) considers the rule, then applies the decision criteria to the rule. In our classification guidelines we specified that labelers should look at each rule on its own, without considering long-term accumulated impact of multiple violations of the rule. For example, numerous 'dead stores' may indicate a bigger problem than a single instance would. We then refined the rubric and conducted a final round of classification, rotating assignments so two new labelers approached the dataset. The results reported below apply to this final round.

## III. Classification Rubric

Fig. 1 shows our final version of the rubric. The first decision point is rule scope (DP1), with three potential branches as statement, module, and system. To explain how it works, we give examples of each branch of the tree below.

**Statement-level:** At the statement-level filter, scope is limited to a single code statement and rules are typically syntactical. We categorized as ND-1 (non-design rule) if the rule scope is limited to single code statement (e.g., internal to method (switch, case, if/else, expression). Empty methods, dead stores, also fit here. *For loop stop conditions should be invariant* is an example rule labeled ND-1 as the violation is likely to be found within a method, checking for how local variables are set before for loops, a basic coding construct.

**Module-level:** Module-level filter includes groups of statements (that might be bundled into a file or package) or a language construct (method, class) that can be executed independently, reused, tested, and maintained or is a composition of other modules. Module-level rules can include syntactical as well as design aspects, leading to a DP2 decision point.

DR-P label aims to identify rules that encapsulate known design paradigm principles. These include object-oriented, functional, imperative programming, etc.; architectural styles, such as concurrent, model-view-control, pipe-filter, etc.; use of design patterns and paradigms, such as exception handling, singletons and factories, etc. *Action Classes should only call Business Classes* is a rule labeled as DR-P since it enforces an aspect of the MVC pattern.

We label module-level syntax checking rules, similar to statement-level, as non-design, ND-2. These rules typically cannot translate easily into another language or paradigm, encapsulate keywords or reserved words or concepts that can't be translated to generalizable concept (e.g., *violating Spring naming conventions* is a rule with no obvious commonalities in other frameworks).

**System-level:** System-level filter includes problems detected across system boundaries (e.g., between languages and/or architectural layers). This also includes rules where system-level metric thresholds are reported (e.g., complexity, dependency propagation). *Avoid having multiple artifacts deleting data on the same SQL table* is a rule labeled DR-S. This is a design rule because not multiple system elements are involved, and their architectural responsibilities are critical, in this case enforcing the data model.

Study artifacts include the categorization guidance (with rubric), design rules spreadsheet, and labeling results. [4]

## IV. Results

### A. *Applying the Rubric to Software Quality Rule Sets*

Of the 466 rules we analyzed, 55% were easily labeled as **non-design** and 19% were labeled as clearly **design**. The remaining had rater disagreements, either because they had characteristics making it hard to classify or where our rubric did not provide sufficient guidance (Fig. 2). Table I lists examples of design, non-design, and hard-to-classify rules.

Table II summarizes the inter-rater agreement (using Cohen's Kappa) after applying this rubric on the data set. For simplicity of reporting, we collapse both design (DR-S and DR-P) and non-design (ND-1 and ND-2) labels together. The

[4]https://goo.gl/u82G2B

TABLE I
EXAMPLES OF DESIGN, NON-DESIGN, AND HARD TO CLASSIFY RULES

| *Design Rules* |
| --- |
| Action Classes should only call Business Classes |
| Avoid high number of class methods invoked in response to a message |
| Avoid Classes with a High Lack of Cohesion |
| *Non-Design Rules* |
| Try-catch blocks should not be nested |
| All script files should be in a specific directory |
| *Hard to Classify* |
| Avoid hiding attributes |
| Avoid defining singleton or factory when using Spring |
| Avoid declaring an exception and not throwing it |
| Lines of code covered by tests |

TABLE II
RATER AGREEMENT. DR=DESIGN RULE. ND=NON-DESIGN. GRAY CELLS SHOW AREAS OF RATER DISAGREEMENT.

| | DR | ND |
| --- | --- | --- |
| DR | 39 | 33 |
| ND | 5 | 71 |

(a) NDepend. Cohen $\kappa = 0.48. N = 148$

| | DR | ND |
| --- | --- | --- |
| DR | 17 | 14 |
| ND | 12 | 90 |

(b) SonarQube. Cohen $\kappa = 0.44. N = 133$

| | DR | ND |
| --- | --- | --- |
| DR | 32 | 24 |
| ND | 35 | 94 |

(c) CAST. Cohen $\kappa = 0.28, N = 185$

low Cohen's $\kappa$ values were due to a high level of disagreement over the hard-to-classify rules.

### B. Validation Feedback

We validated our rubric with three senior architecture analysts, not connected with our team. Each person commented on the rubric, and labeled a random sample (n=74) of the hard to classify rules from the three tools. Our rationale for validating only the hard to classify rules was to gather input on the effectiveness of our subsequent classification rubric improvements, but this may have contributed to low agreement numbers and does not validate our two other categories (their validity relied on internal rater agreement). We compared each label to a reconciled set of labels that the authors created (where reconciled means we discussed each disagreement to derive a consensus label). We calculated Cohen's Kappa and confusion matrices, like that in Table II).

We asked the participants for their impressions of the rubric.
**Scope**. Validators 1 and 3 expressed difficulty in how to interpret 'scope' for design relevance. Consider the rule: *Constructors of abstract classes should be declared as protected or private.* Although the method visibility aspect of this rule appeared to have design implications, Validator 3 struggled with the scope being statement or system.
**DR-P/DR-S overlap**. The validators commented on the overlap between the decision branches leading to DR-P and DR-S. For example, a violation that occurs at one location makes DR-P applicable, however, system boundary implications make DR-S also applicable. Validator 1 said the decision was further complicated by trouble deciphering what it meant for problems to cross system boundaries, "Is the problem located at multiple points in the system? Does it affect multiple points?"
**Metric threshold rules**. Metric threshold is about rules such as *Module complexity over x limit*. Validator 1 said that scope frequently led him down the ND-2 module path because heuristics are frequently applied at the method or class level. However, metric thresholds show up as an example in DR-S, too.

## V. DISCUSSION AND RELATED WORK

Research on the fitness of quality analysis tools for design analysis has taken the following approach: run multiple tools on the same data sets, compare the results with each other [14], or with some other design ground truth [15]. Such an approach has limitations due to potential feature limitations of tools and focus on what cannot be done. Our goal in analyzing the rule sets is to assess the characteristics of automatable rules that check for design problems.
*Are syntactic rules checking for design conformance?* Our rubric led us to classify rules checking purely code-level implementation as non-design. However, the goal of some syntactic rules is to enforce design conformance. Examples of such rules are "avoid declaring an exception and not throwing it" or "classes should not be empty" indicating dead code in some cases.
*Are metric threshold rules indicative of design problems, and thus design rules?* While a number of software metrics violating a certain threshold are available as rules (e.g., "cyclomatic complexity <7", "depth-of-inheritance <5"), there is evidence that such heuristics have a wide range of false positives and disregard context. These thresholds make sense only when combined and correlated with other system observations. These rules are helpful in creating other complex rules, yet are not useful for design assessment solely by themselves.
*Are reporting rules design rules?* Reporting rules measure source lines of code, class size, method size, line length, number of parameters used, and the like. We concluded that they are non-design rules, but need to be treated as contextual parameters for improved analysis of systems, particularly over time.

Software quality tool vendors are grappling with the same challenges. Recently SonarQube simplified its quality model to create a better encapsulation of code issues, maintainability issues and run-time aspects that are most critical such as vulnerabilities[5]. CISQ recently surveyed developers to understand how they perceive the time to fix violations that such rules tag to better assess technical debt[6] in a similar attempt to categorize such rules.

There is a renewed interest in software analytics, and many corporations are adopting them. Sadowski et al. [6] describe how Google integrates static analysis and technical debt identification into Google's development practices and

[5]https://goo.gl/rdGsgS
[6]http://it-cisq.org/technical-debt-remediation-survey/

environments, using a tool called Tricorder. Tricorder addresses the challenges by supporting domain-specific analysers but mandating low (<15%) effective false positive rates, i.e., automatically disabling plugins with feedback developers label annoying or uninformative.

We identified and address the following threats to validity:

**Manual inspection:** To minimize biased manual classification, multiple researchers labeled the rules, and we revised the classification guidance accordingly. Experts external to the research team classified a random sampling of the issues. Those classifying the rules are experts in software design, but they are not all experts in the tools or languages studied.

**External validity:** Our results are based on the rules we used from the three tools, and will not be applicable to other tools unless they share similar rules (which they frequently do). We continue to work on generic definitions of design rules based on customer and open source projects.

**Internal validity:** We checked inter-rater reliability by having two of us assess rules, and rotating raters between rounds. We resolved disagreements and reflected the outcomes in the rubric. We only focus on a subset of rules for Java and C# for three tools. However, they focus on basic language constructs that are transferable.

**Construct Validity:** Our rubric relies on our understanding of design to characterize software quality rules, as well as the literature and tool's internal documentation.

## VI. Conclusion

Software quality tools mix design rules and code quality rules. Separating these is important since design problems often result in longer-term costs. This study labeled the software quality rules of three tools as *design* or *non-design*, using an iteratively defined classification rubric validated with experts.

Our study suggests that progress in automated design analysis can be achieved by addressing the following:

1) **Defining design rule scope**. Our classification results revealed design rules go beyond statement level quality checks. Tools that reported scope of impact (and not just time to fix), would aid in classification.
2) **Validating properties of design rules**. Based on the design rules we identified, we extracted initial properties of design rules. Existing know-how on design tactics and expert observations can help validate and improve the properties of automatable design rules.
3) **Improving context sensitivity**. The hard to label rules in our data are an important outcome of our study. Often, the reason they were hard to label was due to context-specificity. As reported by Microsoft researchers [16], it is only when your analytics efforts work closely with the information needs of the stakeholders that there is real impact.

## VII. Acknowledgments

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

[Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. DM-0004376

## References

[1] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the ACM/IEEE International Conference on Software Engineering*, San Francisco, CA, USA, 2013, pp. 672–681.

[2] S. Johnson, "Lint, a C program checker," Bell Labs, Tech. Rep. 65, 1979.

[3] M. Aniche, G. Bavota, C. Treude, A. van Deursen, and M. A. Gerosa, "A validated set of smells in model-view-controller architecture," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, 2016.

[4] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2015, pp. 50–60.

[5] S. Bellomo, R. L. Nord, I. Ozkaya, and M. Popeck, "Got technical debt?: surfacing elusive technical debt in issue trackers," in *Proceedings of the International Working Conference on Mining Software Repositories*, 2016, pp. 327–338.

[6] C. Sadowski, J. van Gogh, C. Jaspan, E. Soederberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Proceedings of the ACM/IEEE International Conference on Software Engineering*, 2015.

[7] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010.

[8] R. Kazman and H. Cervantes, *Designing Software Architectures: A Practical Approach*. Addison-Wesley, 2016.

[9] C. Menzel, "Possible worlds," in *The Stanford Encyclopedia of Philosophy*, winter 2016 ed., E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2016.

[10] E. Bouwers, J. P. Correia, A. van Deursen, and J. Visser, "Quantifying the Analyzability of Software Architectures," in *Proceedings of the IEEE/IFIP Working Conference on Software Architecture*, Boulder, CO, Jun. 2011, pp. 83–92.

[11] K. Sullivan, P. Chalasani, and S. Jha, "Software Design as an Investment Activity: A Real Options Perspective," in *Real Options and Business Strategy: Applications to Decision Making*, Dec. 1999, pp. 215–262.

[12] L. Xiao, Y. Cai, and R. Kazman, "Titan: a toolset that connects software architecture with quality analysis," in *International Symposium on Foundations of Software Engineering*, 2014, pp. 763–766.

[13] K. Power, "Understanding the impact of technical debt on the capacity and velocity of teams and organizations: viewing team and organization capacity as a portfolio of real options," in *International Workshop on Managing Technical Debt (MTD)*. IEEE, 2013, pp. 28–31.

[14] D. Falessi and A. Voegele, "Validating and prioritizing quality rules for managing techncial debt: An industrial case study," in *International Workshop on Managing Technical Debt*, 2015.

[15] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, "Comparing software architecture recovery techniques using accurate dependencies," in *Proceedings of the ACM/IEEE International Conference on Software Engineering*, 2015, pp. 69–78.

[16] D. Zhang, S. Han, Y. Dang, J.-G. Lou, H. Zhang, and T. Xie, "Software analytics in practice," *IEEE Software*, vol. 30, no. 5, pp. 30–37, 2013.