

**Ruprecht-Karls-Universität Heidelberg**  
**Institut für Informatik**  
**Lehrstuhl für Parallele und Verteilte Systeme**

**Masterarbeit**  
Design and Implementation of a tool for  
automatic,non-trivial code guideline  
checking

Name: Enrico Kaack  
Matrikelnummer: 3534472  
Betreuer: Name des Betreuers  
Datum der Abgabe: dd.mm.yyyy

Ich versichere, dass ich diese Master-Arbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Heidelberg, dd.mm.yyyy

---

## Zusammenfassung

Abstract auf Deutsch

## Abstract

This is the abstract.



# Contents

<b>List of Figures</b>	<b>1</b>
<b>List of Tables</b>	<b>2</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Motivation . . . . .	3
1.2 Goals . . . . .	3
1.3 Structure . . . . .	3
<b>2 Background and Related Work</b>	<b>4</b>
2.1 Code Quality . . . . .	4
2.2 Clean Code . . . . .	5
2.2.1 General Rules . . . . .	6
2.2.2 Naming . . . . .	6
2.2.3 Functions . . . . .	7
2.2.4 Comments . . . . .	9
2.2.5 Data Structures and Objects . . . . .	10
2.2.6 Classes . . . . .	11
2.2.7 Exception Handling . . . . .	11
2.2.8 Code Smells . . . . .	12
2.2.9 Additional Guidelines . . . . .	12
2.3 Quantitative Metrics for Code Quality . . . . .	13
2.3.1 Cyclomatic Complexitiy . . . . .	13
2.3.2 Halstead complexity measures . . . . .	14
2.3.3 Software Maintainability Index . . . . .	15
2.4 Tools for Code Quality Analysis . . . . .	16
2.4.1 PyLint . . . . .	17
2.4.2 PMD Source Code Analyzer Project . . . . .	17
<b>3 Approach</b>	<b>19</b>

## *Contents*

---

<b>4</b>	<b>Quantitative Evaluation</b>	<b>20</b>
<b>5</b>	<b>Conclusion</b>	<b>21</b>
	<b>Bibliography</b>	<b>22</b>

# List of Figures

# List of Tables



# 1 Introduction

This chapter contains an overview of the topic as well as the goals and contributions of your work. Description of the domain Description of the problem Summary of the approach Outline of own contributions and results

## 1.1 Motivation

## 1.2 Goals

## 1.3 Structure

Here you describe the structure of the thesis. For example:

In Kapitel 2 werden grundlegende Methoden für diese Arbeit vorgestellt.

## 2 Background and Related Work

General description of the relevant methods/techniques

What has been done so far to address the problem (closer related work)

Possible weaknesses of the existing approaches

### 2.1 Code Quality

Code Quality describes the quality of source code with regard to understandability and readability. Developers can understand well-written code easily. High-quality code has an impact on onboarding new developers, writing new code and maintaining the existing code.

The onboarding of new developers is an investment of time and money in the developer. The faster a new developer can understand an existing code base, the faster the developer can start writing productive code and providing value.

Maintaining existing code and adding features is part of most software today (TODO source). Agile development is a methodology used in software development that reflects this requirement. Source Code is improved and changed with runnable software versions at the end of each iteration. From a business standpoint, the always-changing code is modeled by subscription-based contracts that include new features and bugfixes. The easiness to change source code is business-critical, and a high-quality code can affect this requirement in the following kinds [3]:

1. Well-written code makes it easy to determine the location and the way source code has to be changed.
2. A developer can implement changes more efficient in good code.
3. Easy to understand code can prevent unexpected side-effects and bugs when applying a change.
4. Changes can be validated easier.

The International Organization for Standardization provides the standard ISO/IEC 25000:2014 for “Systems and software Quality Requirements and Evaluation (SQuaRE)”[10].

Code Quality is measured by the following code characteristics:

1. Reliability
2. Performance efficiency
3. Security
4. Maintainability

Besides the mentioned maintainability characteristics, Code Quality also depends on reliability (like multi-threading and resource allocation handling), performance efficiency for efficient code execution, and security (like vulnerabilities to frequent attacks like SQL-injection).

## 2.2 Clean Code

Clean Code is a concept for high-quality code, coined by the book Clean Code by Robert C. Martin [7]. The root cause for unclean code is chaotic code. Developers produce chaotic code in a conflict between deadline pressure based on the visible output (the functionality of the software) and extra effort to make code more intuitive. The latter is not directly visible as productive output, although an accumulation of chaotic code reduces the productivity over time [7]. A bigger legacy system with chaotic code will slow down later modifications or additions of code. By following the Clean Code guidelines and best-practices, this productivity loss can be minimized.

The Clean Code techniques focus mainly on maintainability by providing intuitive code. This has a positive effect on the security and reliability aspect as well, since developers can find edge cases in non-logical behaviour more easily in intuitive code. Some of the following Clean Code principles may decrease performance efficiency. Still, in many software projects, developer performance is a more valuable resource than actual runtime performance (TODO source).

The following sections explain the clean code guidelines following the book by Robert C. Martin [7]. Since these rules are based on experience of the author, they are controversial. The critique will be explained in section TODO.

### 2.2.1 General Rules

Developers should follow the general rules consistently for developing new features or fixing bugs. They are the essential building blocks for the understandability and reliability of the code.

Follow standard conventions is the first rule. Programming languages have conventions on formatting, naming, etc (e.g. python<sup>1</sup>). If developers follow these conventions, the code feels more familiar for other developers using the same conventions. It is also common practice for big open-source projects to have their own contributing guidelines with coding conventions. The Visual Studio Code GitHub repository contains a "How to contribute" documentation and a section on coding guidelines<sup>2</sup>. Especially in such large open-source projects, many developers are working asynchronously on code. Therefore, having conventions and enforcing the compliance of the guidelines by rejecting pull requests prevents the code from becoming a mosaic of different coding styles.

The rule keep it simple is a summary of most rules in clean code. Simplicity in code makes it simpler and faster to understand for developers. A simpler software architecture enables a developer to modify code and checking all dependencies for potential side effects. Unnecessary complexity increases the time to understand and modify code and introduces bugs by having complex, non-obvious behaviors.

Everytime a developer touches the code, he should also improve the quality of the code or at least not worsen it. By doing a small fix and postponing the clean code principles, the code will become more chaotic until it is refactored. As a result, every change should keep at least keep the code level quality if not improving it.

### 2.2.2 Naming

It is important for understandable code to have good namings for variables, functions, types and classes. "Good" naming is an opinionated topic; The author describes the key components to good naming as follows: Names should be descriptive for the object. Abbreviations or mathematical annotations like *a1*, *a2* are not descriptive and do not provide information about the meaning. Implementations of mathematical expressions intentionally use the same terms as the expression itself. Since this increases the understandability for developers familiar with the mathematical expression, this can be seen as a valid exception. Descriptive names should include a verb or verb expression for

---

<sup>1</sup><https://www.python.org/dev/peps/pep-0008/>

<sup>2</sup><https://github.com/Microsoft/vscode/wiki/Coding-Guidelines>

functions, since functions express an action. Conversely, class names should contain a noun to emphasize the object character of a class.

If the descriptive names are pronounceable too, it is easier to read and it is easier to talk about the code with other developers. Therefore, it is useful to make name longer but descriptive and pronounceable, especially since the autocomplete feature of IDEs will free the developer from typing the long name. Additionally, searching for long names works better than for short names, since long names are more likely to be unambiguous compared to shorter names. Short variables in a small scope (e.g. variable *i* in a loop) are not problematic, but using *i* in a large scope could be ambiguous and troublesome for searching.

An old naming convention is the Hungarian naming convention. In Hungarian naming, the type is encoded as a prefix of a variable name. Nowadays, this is seen as mostly redundant, since IDEs can infer and show the type automatically. The automatic type inference also prevents confusion for the reader if the type in the notation and the actual type are inconsistent. The same logic applies to a prefix for member variables and methods, since a IDE can automatically highlight those tokens.

### 2.2.3 Functions

Functions should be small in length. Exceeding 20 lines should not be necessary in most cases. If a function is small, it can be read more easily and without scrolling. Inside a function, if-, else- and while-statements should contain a function call for the condition and one function call for the body. By following the naming schemas in section 2.2.2, the function calls document the meaning of the condition in an intuitive way without the need for additional comments. Consequently, function calls replaces nested structures in a more readable way.

A general guideline for functions is to fulfill one purpose. This is a rather impractical view, since functions may have to call several functions subsequently to perform the required computation. Therefore, Robert C. Martin specifies that to one abstraction level per function. A low abstraction level handles data access and manipulation, e.g. string manipulation. On a middle abstraction layer, these low-level operations are orchestrated. On the next higher level, the mid-level functions are arranged etc. Switch-Statements violate the one purpose rule and is prone to be duplicated in several other code locations. Since they are a sometimes necessary construct, placing switch-statements into an Abstract Factory and creating different Subclasses for the different behaviours, the switch-statement can be replaced by polymorphism.

The number of function arguments should be three or lower. With many function arguments, it becomes harder to use the function. Additionally, testing becomes harder with more arguments; especially if all argument combinations should be tested. Functions with none or one argument simplifies testing alot. Since testing reduces the number of bugs, testable code is crucial. More arguments should be bundles in a config object or class, so many arguments are passed as one. Furthermore, using config objects enables grouping of arguments for the same context. A special focus lies on "output arguments"; these get passed as reference and are mutated in the function. Output arguments are common in low-level languages like C, but they require additional attention by the reader. Functions are expected to use the input values and return the output as a return value. Consequently, if a developer fails to notice an output argument, he may does not expect it to be mutated by the function and could face a logical error. Using function arguments as immutable is the most intuitive way. Although it is sometime necessary to mutate the input arguments in a function for performance reasons (TODO source).

A particular bad practice in the function body are side-effects. Side-effects happens, if a function modifies a variable outside its scope without explicitly mentioning this in the name. This leads to dependencies between the functions that are not obvious. A developer who uses the function checks the signature (meaning the name, input arguments and return type) and expects the funciton to do what the name suggests. If the function has a side effect, it is an unintended behaviour and will lead to mistakes that are time-consuming to debug. Especially side-effects that initialize other objects result in a time-dependency that is hard to identify and could be overseen at all.

Following the one purpose per function guideline, a function should either perform an action or retrieve information. Especially actions that return a value are unintuitive, since it is not clear if the return value encodes the success state or indicate something else. Furthermore, returning error codes violates this rule and errors should be indicated by raising an exception, if the language supports it. This allows a better seperation between aplication logic and error handling code. The code for catching and handling an exception may be seperated into an additional function, to provide a clean structure for higher level functions. In this case, error handling counts the one purpose of this function. A general, important principle for software engineering applies directly to functions: Don't repeat yourself. Repeated Code is dangerous and chaotic, since it requires changes in multiple locations if it has to be modified. This makes duplicated code very prone to copy and paste errors and small mistakes that may not be obvious

during development but will lead to a fatal crash at runtime. Many patterns and tools have been developed to mitigate duplicated code (TODO: some references to duplicated code detection etc).

### 2.2.4 Comments

Comments are part of every programming language and can play an important role in code quality. Good comments clarify the meaning of the code and help to understand the code. However, comments can become outdated and wrong or provide useless information. In a perfect world, the programming language and the programmer would be expressive enough that commenting is not required for clarity. Following the clean code guidelines for naming and functions makes many descriptive comments obsolete, since the function description is encoded in the function name.

Comments will not help to turn chaotic code into clean code. If a developer explains a line of code by a comment, it is often more helpful to call a function with a descriptive name to replace the comment. Since comments are not part of the program logic, if the explaining comment is not updated with the code, the comment gives misinformation to the reader like described before.

In brief, before writing a comment, the developer should think about expressing the same in code. Robert C. Martin gives some exceptions for good comments [7]:

**Legal notes:** Legal notes like copyright or license information and author mentions may be necessary. Although they should be short in link to an external licensing document in full extent.

**Explaining comments:** Some explaining comments can be helpful and are not easy to encode in normal code. For instance an explanation of special encodings or file formats are better to understand.

**Intention:** Explaining an intention in a comment that is not obvious by the source code is also a valid use for a comment and can help to understand code that otherwise would not make any sense.

**Warning for consequences:** Some parts of the code can have special consequences like not being thread safe or using many system resources. A warning can help a developer from using a function and having trouble.

**Emphasize:** A comment to emphasize a seemingly unimportant part of the code prevents breaking modifications of the code.

### 2.2.5 Data Structures and Objects

Data Structures and Objects can be easily mixed together, but their purpose is converse. Objects hide data behind abstractions and has functions that work with these data. Conversely, data structures expose the data and do not provide functionality. Based on this definition, the Law of Demeter (TODO source) is defined: Objects should know as little as possible about the implementation of objects it manipulates. As a result, code becomes decoupled, since objects do not depend on specific data structures of other objects. In object-oriented programming, a method *m* of an object *O* may only call methods of the following components:

- the object *O* itself
- the parameters of the method
- objects that are created within *m*'s scope
- instance variables of *O*

The method should not call methods of objects, that are returned from the allowed objects. If a method calls specific functions of these objects, it assumes this behaviour. As a result, the two objects are closely coupled. Such bad code practice is called train wreck, since subsequent method calls look like multiple train carriages. A split in multiple variables and method calls improves the Train Wreck. Nevertheless, method calls on objects assume knowledge of the internal implementation of the objects and would violate the Law of Demeter. However, if the methods are accessor functions, it would not violate the law, since the method calls are just an access to the data structure. Instead of calling a method of an object and calling a method on the returning object, the first called object could provide a function that takes care of calling the other object. This reduces the coupling between the three objects and would preserve the Law of Demeter in all objects.

Data structures are often used as Data Transfer Object (DTO) to communicate with other processes or services. These objects do not contain any functions, they only have accessible member variable. A specialisation of DTOs are Active Records, that contain additional methods for data storage, since they are used to represent a data source like a database. For both types, it is bad practice to insert business logic into these objects, since they should be treated as data structures. This creates a hybrid between an object and a data structure that should be avoided.



### 2.2.6 Classes

A Class should be small like functions. Instead of counting lines, the size of a class is the number of responsibilities. The name of the class should describe its responsibility, therefore, a single responsibility per class is intended. This is called Single-Responsibility-Principle and it restricts a class to one responsibility. A class should only have one reason for modification, consequently a class has a single responsibility (TODO source). The Single-Responsibility-Principle lead to a system of many small classes with one, clearly defined responsibility.

In addition, classes should have a high cohesion. Cohesion is high, if a method manipulates many instance variables. If all methods of a class manipulate all instance variables, the class is maximal cohesive. With a high cohesion, the methods and the class can be seen as a logical unit. A low cohesion indicates to split the class into multiple classes, since the methods are not a logical unit. As a result, classes gets smaller and will more likely comply to the Single-Responsibility-Principle. (TODO add metric for cohesion)

Dependencies of classes are critical, since they may be modified. If the class depends on specific details of the implementation, it has to be changed, if the dependency changes. Therefore, classes should depend on abstract classes that describe a concept, not the implementation details. Since the concept is unlikely to change if the implementation details change, the dependency is isolated.

### 2.2.7 Exception Handling

Section 2.2.3 briefly mentioned a clean code guideline for exception handling. Exceptions should be handled by raising and catching exceptions instead of error codes or flags. Error code handling introduces code, that is not coherent with main logic. Error codes have to be checked immediately and error handling has to be implemented directly. As a result, the two concerns main logic and error logic are mixed. By raising and catching errors, the error handling can be completely extracted into a separate function and can be removed from the main logic. Furthermore, the try-catch block enforces a transactional behaviour. At the end of either the try or the catch block, the application should be in a consistent state. Error codes hide this explicit transactional behaviour. Notwithstanding, Go as a more recent programming language<sup>3</sup> returns an explicit error types instead of raising an exception. This was designed to “encourage you to explic-

---

<sup>3</sup>designed in 2007 as a response to problems with C++, Java and Python[1]

itly check for errors where they occur”, instead of “throwing exceptions and sometimes catching them”[5]. Exceptions should contain enough context to find the cause and the source of the error. Exception classes accomplish the grouping of cause source of the error.

In case of error handling with a third-party library, the library called should be wrapped and multiple, possible exception types should be grouped into a unified library exception type. The wrapping reduces the coupling to the third-party dependency. If the library changes (e.g. changing the signature and adding another exception type), only one location has to be adapted to comply to the changed library interface. Additionally, changing to another library is simple and tests can mock the third-party library. For exception handling, wrapping unifies the exception from the library that is clearly distinct from own exceptions.

Another form of exception handling is returning null. Instead of checking for an error code, the caller method checks for a null return value. As easy it is to miss an error code, as easy it is to miss a null checking. Consequently, the program will terminate with an unhandled `NullPointerException` at runtime. Tony Hoare introduced the null reference in 1965 and later called it his “billion-dollar mistake”[6], since it led to many bugs and security vulnerabilities throughout the decades. Languages like Kotlin are designed with Null Safety enforced. Kotlin distinguishes between nullable and non-nullable references, with a compiler enforcing null checking[2]. If a language does not enforce null safety with a compiler, a special case like an empty collection or an optional type can be returned. Passing null to a function as a parameter is also dangerous, since a function would explicitly assert for non-null parameter values. Since this is not a clean way, null parameters should not be used.

### 2.2.8 Code Smells

Aforementioned guidelines can be rephrased as code smells. Code smells are a characteristic of code that possibly indicate a problem with the clean code rules and therefore maintainability of the system.

### 2.2.9 Additional Guidelines

Robert C. Martin describes more guidelines for system design, multi-threading, testing and third-party code[7]. These topics are not part of this chapter, since the focus of this work are the aforementioned guidelines and principles.

## 2.3 Quantitative Metrics for Code Quality

Quantitative metrics express code quality as a quantitative unit. This has multiple advantages:

- A metric sums up the quality of a project in a single unit.
- A quantitative approach tracks the changes in code quality over time. Therefore, it is obvious if code quality improves or not. In case the quality undercuts a threshold, special measures like mandatory refactoring can be undertaken.
- A developers performance can be evaluated based on the code quality. Since the maintainability and reliability of the software depends on the code quality, this is a good incentive to enforce high quality work.
- A certain level of code quality can be required by a contract. As a result, a customer can expect less bugs and a smaller maintenance effort.

Metrics for code quality are a subset of more general software metrics. The following sections describe software metrics that express code quality.

### 2.3.1 Cyclomatic Complexity

Cyclomatic Complexity is a metric for the complexity of a section of code. It was introduced by Thomas McCabe, Sr. Is is measured by counting the number of linearly independent execution paths. (TODO source)

To compute the Cyclomatic Complexity, the control-flow graph is required. The control-flow graph represents possible execution paths. A node represents a basic block, a code sequence without branching. A directed edge represent jumps in the control flow; e.g. an if-statement without an else-case has two edges. One edge to the basic block in the if-branch and one edge to the basic block after the if-statement. Beside cyclomatic complexity, the control-flow graph is used for compiler optimisation techniques like dead code elimination and loop optimization (<https://ag-kastens.cs.uni-paderborn.de/lehre/material/compil/folien/c101-217a.pdf>).

The cyclomatic complexity on a control-flow graph is defined as:

$$M = E - N + 2P$$

E is the number of edges, N the number of nodes and P the number of connected components. A connected graph is a subgraph, in which all nodes are connected to each other by a path. P represents the number of subprograms (like multiple functions or classes will). Following this definition, the cyclomatic complexity can be calculated.. See figure TODO for a visualisation.

MacCabe recommends to limit the cyclomatic complexity to 10. NIST confirms this recommendation in TODO. A lower cyclomatic complexity improves testability, since the complexity represents the number of execution paths that need to be tested. Therefore,  $M$  is the upper bound for number of test cases for full branch coverage. Furthermore, studies suggest a positive correlation between cyclomatic complexity and defects in functions. TODO source Safety standards like ISO 26262 (for electronics in automobiles) or IEC 62304 (for medical devices) mandates a low cyclomatic complexity TODO source.

### 2.3.2 Halstead complexity measures

Maurice Halstead introduced the Halstead complexity measures in 1977. It is a collection of multiple countable measure and relations.

The Halstead metrics operates on a sequence of tokens that are classified to be an operator or an operand.

The classification enables to count the following properties:

- $\eta_1$  as the number of distinct operators
- $\eta_2$  as the number of distinct operands
- $N_1$  is the sum of all operators
- $N_2$  is the sum of all operands

The total program vocabulary can be computed as:

$$\eta = \eta_1 + \eta_2$$

The program length is not calculated by the lines of code, but as a sum of all operators and operands:

$$N = N_1 + N_2$$

The voume of a program in terms of program length and program vocabulary is defined as:

$$V = N * \log_2 \eta$$

An important metric is the difficulty of understanding a program, defined as:

$$D = \frac{\eta_1}{2} * \frac{N_2}{\eta_2}$$

The major contribution to difficulty is the number of distinct operators  $\eta_1$ . The combination of difficulty and volumes is the effort for understanding or changing code:

$$E = D * V$$

The effort translates into real coding time following the relation

$$T = \frac{E}{18} s.$$

The number of bugs correlates with the effort following

$$B = \frac{E^{\frac{2}{3}}}{3000}$$

(TODO general critique etc. on this metric since it seems to be arbitrary)

### 2.3.3 Software Maintainability Index

The software maintainability index was developed by Dan Coleman and Paul Oman in 1994. 16 HP engineers evaluated 16 software systems and scored it in a range from 0 to 100, with 100 representing best maintainability. Following a regression analysis, they identified the following equation to match the maintainability of the evaluated systems (TODO source from Using metrics to evaluate software system maintainability):

$$MI = 171 - 5.2 * \ln \bar{V} - 0.23 * \bar{M} - 16.2 * \ln \overline{LOC} + 50 * \sin \sqrt{2.4 * C}$$

where  $\bar{V}$  is the average Halstead Volume,  $\bar{M}$  the average cyclic complexity,  $\overline{LOC}$  the lines of code and  $C$  as fraction of comments.

The software maintainability index was defined many years ago with a limited sample size of developers and projects. Additionally, programming languages have changed significantly over time. A study by Sjøberg et. al suggest no correlation between the software maintainability index and actual maintainability effort in a controlled environment[11]. Although the study only analysis four software projects and lacks generalisation, they find a strong anti-correlation with different maintainability metrics.

Only the code size seems to correlate with the actual maintainability. The former seems to be consistent with a systematic review on software maintainability predictions and metrics by Riaz et. al[9].

Kritik an Clean Code

??Teaching clean code, the paper from one german university

## 2.4 Tools for Code Quality Analysis

It is good practice to use static code analysis tools to improve code quality. A static code analysis examines a program by analysing an abstract syntax tree, the control or data flow, a pointer analysis or an abstract (approximated) execution. In comparison to dynamic analysis, the code is not executed. The result of a static analysis can be based on approximations, so there might be false positive results. An expert has to examine the result and correct results if necessary[8].

Besides the use of static code analysis for compiler and optimisations, it is also helpful for analysing the code quality. Different categories of code quality related principles can be analysed:

**Code Guidelines:** A static code analysis can ensure compliance to structural, naming and formatting code guidelines.

**Standard Compliance:** A requirement for a software may be compliance to an industry standard like IEC 61508 or ISO 26262 (TODO source). Static code analysis can ensure or at least assist with the compliance.

**Code Smells:** Some code smells follow a known pattern and a static code analysis can detect those smells. Since code smells can be an indication for chaotic code, it is best if these smells are detected and removed early on.

**Bug Detection:** Although bug detection with static analysis can not detect all bugs, every detected bug before a software release is important. Examples for detected bugs are unhandled expressions or concurrency issues [4].

**Security Vulnerability Detection:** Static code analysis tools can detect some security vulnerabilities like SQL Injection Flaws, buffer overflows and missing input validation with high confidence. Since these are some common, easy to exploit vulnerabilities, a analysis for security vulnerabilities can increase the security of the overall software [12].

The following sections provide an overview of a few static code analysis tools with focus on code quality and maintainability. I selected them based on popularity and if the projects are still maintained.

### 2.4.1 PyLint

PyLint is a code analysis tool for the Python programming language. It is open-source and licensed under the GNU General Public License v2.0 and available for all common platforms. PyLint can be executed as a standalone program or can be integrated into common IDEs like Eclipse. A continuous integration pipeline may include PyLint as well to ensure an analysis on every build.

With PyLint, the developer can make sure the code complies to the PEP8 (TODO) style guide for python coding. This includes name formatting, line length and more. It does not calculate a metric for the code but instead warns about violated principles. Additionally, PyLint can detect common errors like missing import statements that may cause the program to crash at start or later at runtime. To support refactoring, PyLint can detect duplicated code and will suggest to refactor the code.

PyLint can be configured to ignore some checks and to disable specific rules. To expand the rule set, a developer can write a "checker", an algorithm to check for a specific rule. The algorithm can analyse the raw file content, a token stream of the source code or an AST representation. The checker can raise a rule violation by providing the location information and the problem type to the PyLint framework and it will be included in the PyLint analysis report.

### 2.4.2 PMD Source Code Analyzer Project

PMD is an "extensible cross-language static code analyzer" for Java, JavaScript and more. As an open-source project, it is licensed under a BSD-style license and is available for MacOS, Linux and Windows based systems. It integrates into build systems like Maven and Gradle as well as into common IDEs like Eclipse and IntelliJ. PMD can run as part of a continuous integration pipeline and is included in the automated code review tool Codacy (see TODO).

Depending on the target language, PMD supports different rules. For Java, PMD has rule in the following categories:

**Best practices:** Best practices include rules like one declaration per line or using a logger instead of a *System.out.print()* in production systems.

**Coding Style and Design** Several rules to improve readability of the code like naming conventions, ordering of declarations and design problems like a violation of the law of Demeter, Cyclomatic complexity calculation and detection of god classes.

**Multithreading:** Rules to mainly prevent the use of not thread safe objects or methods. Due to the nature of multi-threading and unpredictable scheduling of threads, problems like unpredictable values of variables or dead locks may occur. Since they may not occur in every execution, they are hard to spot and to debug. Consequently, warnings of using non thread safe code may save hours of debugging.

**Performance:** These rules flag known operations that have hidden performance implications. For example, string concatenation with the plus operator in Java causes the Java Virtual Machine to create an internal String buffer. This can slow down the program execution if numerous String concatenations are performed.

**Security:** Security wise, PMD only checks for hard coded values for cryptographic operations like keys and initialization vectors.

**Error Prone Code:** PMD checks for several known code structures that will or may cause a bug at execution. Some rules are part of the clean code principles described in 2.2 and some rules are specific to the target language.

Integration in continuous integration, manual review

Tools review for all tools that check different stuff

Single Responsible principle, Open-Closed principle



# 3 Approach

Approach(es) without the quantitative results (or only selected results to motivate the choices/the problems)

- What has possibly failed (short)

- What has worked and why

- What could be possible method extensions/improvements

## 4 Quantitative Evaluation

Research questions Description of the evaluation environment/setup Results/answers per research question Optional comparison to prior work Discussion and analysis of strengths/problems Note: see

## 5 Conclusion

An overall short summary of results What was successful, what not (and hypotheses why) Hints for further future work/extension

# Bibliography

- [1] Go at Google: Language Design in the Service of Software Engineering.
- [2] Null Safety - Kotlin Programming Language.
- [3] Robert Baggen, José Pedro Correia, Katrin Schill, and Joost Visser. Standardized code quality benchmarking for improving software maintainability. 20(2):287–307.
- [4] Aurelien Delaitre, Bertrand Stivalet, Elizabeth Fong, and Vadim Okun. Evaluating Bug Finders – Test and Measurement of Static Code Analyzers. In *2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS)*, pages 14–20. IEEE.
- [5] Andrew Gerrand. Error handling and Go.
- [6] Tony Hoare. Null References: The Billion Dollar Mistake.
- [7] Robert C Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education.
- [8] Herbert Prahofer, Florian Angerer, Rudolf Ramler, and Friedrich Grillenberger. Static Code Analysis of IEC 61131-3 Programs: Comprehensive Tool Support and Experiences from Large-Scale Industrial Application. 13(1):37–47.
- [9] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 367–377. IEEE.
- [10] ISO Central Secretary. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE.
- [11] Dag I K Sjøberg, Dag Sjøberg, Bente Anda, and Audris Mockus. Questioning Software Maintenance Metrics: A Comparative Case Study. page 4.
- [12] Dave Wichers and Eitan Worcel. Source Code Analysis Tools | OWASP.