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

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 provididng 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 ressource 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[1]). 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[2]. 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

[1] `https://www.python.org/dev/peps/pep-0008/`
[2] `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 pronouncable 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 pronouncable, 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 unambigious 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 ambigious and troubleful 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 neccessary 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 fullfill one purpose. This is a rather unpractical 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 neccessary 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 programm logic, if the explaining comment is not updated with the code, the comment gives missinformation 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 ressources. 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 there purpose is converse. Objects hide data behind abstractions and has funcitons 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 implementaion of objects it manipulates. As a result, code becomes decoupled, since objects do not depend on specific data strucutres 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 this 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 implemenation 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 preserver 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 varaiable. A sepcialisation 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 this 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 responsiility. 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)

Depdendencies of classes are critical, since they may be modified. If the class depends on specific details of the implemenation, 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 unlikly 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 exeception 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 imideatly 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 completly extracted into a sperate 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 tranactional behaviour. Notwithstanding, Go as a more recent programming language[3] returns an explicit error types instead of raising an exception. This was designed to “encourage you to explic-

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

itly check for errors where thy 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 thrid-party libary, 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 programm 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 lead 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 paramter is also dangerous, since a function would explicitly assert for non-null paramter values. Since this is not a clean way, null paramters 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 Complextiy

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/compii/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 numder 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:

- η_1 as the number of distinct operators
- η_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 η_1. The combination of difficulty and volumes is the effort for udnerstanding 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 effor 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):

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

where $\overline{V}$ is the average Halstead Volume, $\overline{M}$ the average cyclic complexity, 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 foormer 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 requiremnt 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 vulnerabilites, 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 plattforms. PyLint can be executed as a standalone program or can be integrated into common IDEs like Eclipse. A continous 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 provididng 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 svailable 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 Intelij. PMD can run as part of a continous 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 decleration 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 declerations and design problems like a violation of the law of Demeter, Cyclomatic complexity calculation and detection of god classes.

Multihtreading: 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.

A user can expand the PMD ruleset in two ways: A XPath expression can be spezified and will be validated against the AST by the PMD framework. For more control, it is possible to write a Java-based plugin that implements a custom AST traverser. The latter allows for more sofisticated rules and checks.

2.4.3 Codacy

Codacy is a software to "automate your code quality" (TODO source) for more than 30 programming languages. It is available as a cloud-based subscription service with a free tier for open-source projects and a self-host option for enterprise customers. Codacy runs as a cloud software and a user can connect a GitHub, GitLab or Bitbucket repository that will be scanned automatically on every push or trigger a scan of local files with a command-line program. Additionally, a badge can be added to the readme page to show off the analysis results.

Codacy can be seen as a plattform that runs multiple different "analysis engines". These analysis engines combine multiple tools depending on the langauge. For Python,

Codacy uses PyLint, Bandit (a scanner for security issues) and a metric plugin. Due to the licensing of those analysis engines, the engines are open-sourced, wheras the Codacy platform is propriatary.

Depending on the language, Codacy supports scanning for common security and maintainability issues. The later is faced with scanning for Code Standardization, Test Coverage and Code Smells to reduce technical debt.

Codacy allow customization by disableing certain rules and changing rule paramters like the pattern to fit the rules to the projects preference.

2.4.4 Sonarqube

Sonarqube is a analysis tool to maintain high code quality and security. It supports 15 programming languages in the open-source version licensed under the GNU Lesser General Public License, Version 3.0. In the Developer Edition, Sonarqube supports 22 languages and 27 languages in the enterprise edition. Sonarqube is a self-hostable server application with an SonarScanner client module that can be integrated in build pipelineslike Gradle and Maven as well as a command-line tool for other build pipelines. With the Developer Edition, branch and pull request analysis are possible and a pull request will be annotated with the analysis results.

For Python, Sonarqube offers more than 170 rules. These rules are in the following categories:

Code Smells: Code Smells like duplicated String literals are flagged with four different levels of severity (from higher to lower severity): Blocker, Critical, Major, Minor. Explanations and examples are accessible for the developer to understand and fix the issue. The analysis is more in-depth than AST-based solutions, since it additionally uses control and data-flow analysis.

Bug: Multiple rules cover bugs that would definetively result in a runtime exception and program termination. As an example, calling a function with the wrong number of arguments will be flagged with the highest severity, since it will raise a TypeError during runtime. Although programmers may notice issues like this during coding and testing, the issue can remain hidden if it only triggers in a special execution path of the system.

Security Hotspot: The Security Hotspot analysis unfolds pieces of code that may be a real vulnerability and requires a human review. The scanning has a hotspot

detection for seven out of the OWASP Top Ten Web Application security risks (TODO src). A flagged hotspot contain a detailed explanation of the reason for being flagged and a guide on how to review this hotspot. The recommendation to double check a piece of code can help to prevent a security vulnerabiliy from being deployed to production.

Security Vulnerabilities: A security vulnerabilites analysis reveal code that is at risk of being exploited and has to be fixed immediately. Especially misconfiguration of cryptographic libraries can be revealed easily and future damage prevented.

Additionally, Sonarqube offers several metrics like test coverage and an custom derivate of Cyclomatic Complexity named Cognitive Complexity. The authors sees several shortcommings in the original Cyclomatic Complexity model:

- Parts of code with the same Cyclomatic Complexity do not neccessarily represent equal difficulty for maintenance.
- Cyclomatic Complexity does not include modern language features like try/catch and lambdas. Therefore, the score can not be accurate.
- Cyclomatic Complexity lacks useability on a class or module level. Since the minimum complexity of a method is one and a high aggregated Cyclomatic Complexity for a class is measured for small classes with complex methods or large classes with low complexity per method (e.g. a domain class with just getter/setter).

Cognitive Complexity adresses these points, especially the incorporation of modern langauge features and meaningfullness on class and module level.

The calculation is based on three rules:

Ignore Shorthand Structures : Method calls condense multiple statements into one easy to understand statement. Therefore, method calls as shorthand are ignored for the score. Similiarly, shorthand strcutures like the null-coalescing operator reduces the Cognitive Complexity compraed to an extensive null check and is therefore ignored for score calculation.

Break in the linear control flow : A break in the expected linear control flow by loop structures and conditionals adds to Cognitive Complexity as to Cyclomatic Complexity. Additionally, Catches, Switches, sequences of logical operators, recursion and jumps also adds to Cognitive Complexity, since these strcutures break the linear control flow.

Nested flow-breaking strcutures : Flow-breaking structures that are nested additionally increase the Cognitive Complexity, since it is harder to understand than non-nested structures.

The method-level Cognitive Complexity score represents a relative complexity difference between methods that would have the same Cyclomatic Complexity. Furthermore, the aggregated Cognitive Complexity on a class or module level now differentiate between classes with many, simple methods and classes with a few, but complex methods. As a result, domain classes (containing mainly getters/setters) have a lower score compared to classes with complex code behaviour. (TODO source whitepaper multiple times)

Developers can extend Sonarqube similiarly to PMD. They can write a Java plugin implementing a custom AST parser. The Java plugin is compiled into a .jar file and placed into the plugin dir of the Sonarqube installation. For simpler rules, XPath exressions are possible through the web interface and allow a quick extension. For instance, if developers observe bad code style during a pull request review, they can quickly write a rule to enforce this rule in all subsequent pull requests automatically.

Integration in continous 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

This section is divided into the approach for the Clean Code Analysis Plattform and the Clean Code classification.

4 Clean Code Analysis Plattform

The goal of the design and implementation of the Clean Code Analysis Plattform (CCAP) is a tool for software developer to imnprove the code quality of existing and new code. The tool accepts an directory containing source code files as input and analyses the input for snippets of improveable code quality. If the analysis classifies a code snippet as problematic, it should help the developer to improve the snippet by providing information about the problem. Ultimately, this should train the developer to spot problematic code by its own and to write clean code by default, so the number of alerts should decrease. At the same time, the overall software quality of a project increases immediately at rewriting a marked snippet and in the long term at training the developer to write code with higher quality.

In order to use the tool effectively, the design and implementation should cover the following requirements:

Useability : The CCAP should be an easy-to-use tool. Developers shall be able to install and run the tool. The extra effort of using this tool should be small and the developer should use the tool in his day-to-day workflow without additional friction. The developers can interpret the issue and localise the problematic code spot immediately.

Expandability : The extension of the detected code problems should be easy. A clear defined interface for extensions is required so an extension developer would not need specific knowledge about the internal architecture of the tool. The expandability allow a desired workflow of a developer finding problematic code in a e.g. peer-review, formalizing it into an extension and sharing this extension with the team. With each iteration, the code quality of all team members would increase.

Integration : The tool should be easy to integrate into different systems. This includes lokal workflows like git pre-commits or build systems and remote continous integration/delivery/deployment pipelines.

A more specific requirement is Python as an input language and the expansion langauge. After JavaScript, Python is the second most popular programming language 2019 according to the Github statistics (`https://octoverse.github.com/#top-languages`). Besides the general popularity, Python is heavily used in the scientific community for machine learning and in universities for teaching programming. These groups are part of the potential target audience and students in specific can benefit from automated reporting of low quality source code.

4.1 Architecture

The CCAP architecture is divided into a static part and two extension possibilities: An extension with analysis plugins adds more rules that are validated by the system. Adding an output plugin allows to specify the output format to fit custom workflow needs. The static part consists of four components: A core component to act as a orchestration unit, a component for handling the source code input and a component for handling the analysis plugins as well as output plugins.The design follows the requirements and goals for the plattform. (TODO: High level architecture schematic diagram)

The core component contains the main function and handles the argument validation and parsing. Furthermore, it orchestrate other components by initializing and executing those. This process is divided into the argument parsing, initialisation and execution phase: In the argument parsing phase, the command line arguments are parsed and validated. It valdiates the existance of the required input directory argument and the optional plugin path configuration for the analysis plugins and the output plugins. Additionally, the logging level and the output format can be defined. The latter determines, which output plugin will be used, although the existance of the specified plugin is not validated in this phase. A parsing or validation error will cause a program termination without further processing. The initialisation phase instantiate all components and the analysis plugin handler will scan the specified directories for plugins and keeps an index of all found plugins in memory. The output plugin manager scans for an output plugin, that satisfies the specified argument. If no plugin matches the output argument, the programm will terminate with a one exit code and a failure message to indicate the problem. In the execution phase, the input handling component scans the input directory for files ending with *.py* and parses the source code into an Abstract Syntax Tree (AST) per file. In the next step, the core passes the parsed data to the analysis plugin handler. The latter will execute all plugins for all files and collects the results. After-

wards, the core component calls the output plugin component to output the results. If no exceptions occur during the execution phase, the program will be terminated with a zero exit code indicating the succesfull run.

The input component scans the given input directory for all Python source code files and parse the source code into an AST. For scanning the input directory, an algorithm will walk recursively over all folders and files. The detection of Python source code files is based on the file ending *.py*. The algorithm will return a list of file paths and the corresponding file content. Next, the AST parser is called and will add a parsed AST object to the list besides the file path and content. This list will be passed to the analysis plugins by the analysis plugin handler. An alternative approach would be to not read and parse the code in the input component, but instead let the plugins read and parse the file content if needed. With many files to scan, the latter approach would have a lower memory footprint since the file content and the AST will not be held in memory. Consequently, every analysis plugin has to perform an expensive read operation from disk and the performance scales with the number of files and the number of analysis plugins. If the input component reads all files, the information are held in the main memory and the performance only scales with the number of files. Since the files are text-based files, the number of files needed to excess the main memory is expected to be high enough to fit most projects. (TODO sample calculation?).

All analysis plugins are managed by the analysis plugin handler. This component finds all plugins, executes the plugin and collects the reported results. During the initialisation phase of the core component, the analysis plugin handler will scan the plugin directory for all Python files. It imports all Python files and scans those for classes, that inherit from an abstract *AbstractAnalysisPlugin* class. The abstract class defines all methods that need to be implemented in the concrete plugin subclass. All found classes are instanziated by the analysis plugin handler. During the core execution phase, the core receives a list with the file name, file content and the parsed ast. All plugins are called on a specific entry point method that is defined in *AbstractAnalysisPlugin* with the afforementioned list. The plugin will return an instance of *AnalysisReport* with the plugins metadata and a collection of problems. The report is collected for every plugin into an *FullReport*. Additionally, the report contains information about the overal plugin execution time and run arguments. After all plugins have been executed for all files, the analysis plugin handler returns the *FullReport* to the core component.

The last component in the execution chain is the output plugin handler that will pass the *FullReport* to the specified output plugin. It implements the same algorithm as

the analysis plugin handler to find all plugins that inherit from *AbstractOutputPlugin*. Instead of keeping track of all plugins, only the plugin that correspond to the output format argument is instantiated. The output plugin has an entry point as defined in *AbstractOutputPlugin* that is called with all the collected results.

4.1.1 Analysis Plugins

Analysis plugins provide the easy extendability of the plattform by developers. All users of the tool can expand the set of problems it can detect by implementing a plugin Python and placing it into the plugin directory of the tool. In order to be compatible with the core compoennts, a plugin has to be a class that inherits from the *AbstractAnalysisPlugin* class. Firstly, the abstract class indroduces a class member variable *metadata* of the class *PluginMetaData*. A concrete plugin class sets this class member in the constructor to provide a plugin name, author and optional website. This meta data is used in the output to show more information about the plugin that reported a problem. Secondly, *AbstractAnalysisPlugin* specifies a *do_ analysis* method that serves as a entrypoint that the plattform will call. It accepts a list of *ParsedSourceFile* objects, that contains the file path, the file content and the corresponding AST for all input files. With this information, the plugin can implement any logic to detect problems. For example, the plugin can traverse the AST to look for specific node types, it can use a tokenizer on the files content and analyse the token stream or it can run sophisticated machine learning algorithms on the source code. The plugin can even import third-party libraries, although they have to be installed by the user on the system. After detecting all problems, the plugin returns an *AnalysisReport*. The report contains the plugins metadata and a list of found problems. These problems are instances of a problem class inheriting from *AbstractAnalysisProblem*. The abstract class expects a file path and line number as constructor arguments and requires the plugin developer to override the problem name and description. (TODO Refer to the sample in Return None plugin). The problem name and description will be shown in the final output and should follow the following guidelines:

- Have a suitable name that allows experienced developer to quickly recognize the problem.
- Explain what code construct is problematic.
- Give reasons why this code is seen as problematic.

- Show guidance and examples on how to fix the problem and improve the code.

Although it is possible to use one plugin for multiple, different problem types, having one plugin for one problem type helps to reuse and share the plugin. Additionally, it can be disabled easily by removing the plugin from the plugin folder.

Steps to create an analysis plugin

In order to expand the CCAP with an analysis plugin, the following steps are required for a developer:

1. Create a *.py* file with a class inheriting from *AbstractAnalysisPlugin*.
2. Instanziate *PluginMetaData* and assign it to the metadata member.
3. Define a problem class inheriting from *AbstractAnalysisProblem* and set the problem name and description following the guidelines above.
4. Implement the *do_ analysis* method with a *ParsedSourceFile* parameter. Return an *AnalysisReport* instance with all found problems.
5. Place the *.py* file into the analysis plugin directory of the tool.

Whereas these are the minimum required steps to implement the plugin, the developer is free to add additional methods, classes or import libraries as necessary. Furthermore, it is adviceable to implement several tests to ensure the plugins correctness. CCAP uses pytest [1] for testing, implementing a test is as easy as writing a function with a "test_" prefix and running the *pytest* command. TODO: further test instruction after test implementation

4.1.2 Return None Plugin

A rather simple analysis plugin demonstrate the capabilities of the CCAP: The Return None Plugin scans the source code for functions that return the None value. Returning None may result in runtime exceptions, if the function caller does not expect and handle a potential None return. Although the None can be returned directly or as a value of the returning variable, this plugin only focus on the explicit *return None* statement to showcase the options for the developer to analyse the source code.

[1] `https://docs.pytest.org/en/stable/`

Detecting *return None* is possible in multiple ways. The following shows the possibilities for developer to implement such a detector:

Regular Expression : In the *do_ analysis* method, a developer has access to the source code as a string. Therefore, it is straitforward to use a regular expression to detect a *return None* statement.

```
import re
matches = re.finditer(r"return␣None", source_file.content, re
```

Since the regex library only returns the start and end index of matches, these have to be converted to line numbers. Afterwards a *ReturnNullProblem* instance can be created for every match and added to the *AnalysisReport* for this plugin.

This approach uses regular expressions to match patterns in a string without utilizing the structureness of the code. It is a simple but powerful way and most developers are familiar with regular expression. On the flip-side, source code may have various ways syntactic ways to express a semantic. Consequently, the regular expressions have to be designed carefully to cover all variations. For instance, it is possible to encounter a doubled whitespace, that would break the afforementioned regular expression. Additionally, regular expressions do not operate on the structure of the code, therefore they can not detect high level patterns on the code structure.

Tokenization : The process of dividing a character stream into a sequence of tokens is called tokenization (also know as lexical analysis). With the Python tokenizer, a token can contain multiple characters and has a token type like name, operator or intendation. A token sequence provides more information about the code structure that can be used to detect problematic patterns. A token sequence for a *return None* statemant would be the following:

```
[...(type: NAME, value: return), (type: NAME, value: None)...]
```

A simple algorithm would scan the sequence for two subsequent name tokens with the values "return" and "None". The abstraction level of a token sequence is higher than of a character sequence. Since the whitespace between "return" and "None" provide no semantic meaning, it is removed on this abstraction level. Whereas the regular expression based approach would have to deal with problems as the doubled whitespace, the token-based approach profits from the higher abstraction level and can access meaningful tokens directly.

Abstract Syntax Tree : After tokenization, a parser takes the token stream and prases it into a hierachical datastructure like an Abstract Syntax Tree. With an AST the code is represented in an structural way and it is possible to traverse the tree following the structure of the code. The AST consists of nodes with different type and children, for example a node representing an if statement. It is possible to access the condition and the body of the if statement as descendant nodes in the AST. Analysing an AST allows an higher abstraction level then analysing the token string, since the code strucutre is represented. Therefore, more abstract rules that analyse the code structure are possible.

To detect a *return None* statement in the abstract syntax tree, an algorithm would traverse all AST nodes looking for a return-typed node. If found, the value descendant contains the statement part after the return keyword. A *None* value would be represented as a node of type constant or name constant (depending on the parser version). The value descendant of the constant node may then be checked for equality to *None*. All AST nodes contain the corresponding line number, that are necessary for creating a problem message:

```
for node in ast.walk(a.ast):
    if isinstance(node, ast.Return):
        return_value = node.value
        if isinstance(return_value, ast.Constant) or isinstance
            if return_value.value is None:
                problems.append(ReturnNullProblem(a.file_path,
```

Analysing the AST is best for problems in the code structure, since the AST represents the code structure in a well-defined, traversable datastructure. Although simpler patterns like the *return None* could be detected using regular expressions, a detection on AST level allows to detect the semantic meaning instead of the syntactic representation of a problem. For more complex problems, it is inevitable to use the AST most of the time.

This implementation covers only the simple case, in which *return None* is written explicitly. Of course, several modifications are possible, that would not be detected by this plugin and would required more sophisticated algorithms. Evaluating, if machine learning models trained on this simple rule can also detect such modifications is part of the second part in research question 3 (TODO: check, jump link).

Since detection alone is not a great help for the user, a usefull description of the problem is neccessary. As described in section (TODO: background, clean code, return none), there are some alternatives to returning none. If the none happens due to an error, it is better to raise an exception and provoke an explicit error handling, which prevent runtime errors. If the normal return type is a collection, an empty list is more appropriate, since the function caller will most likely write logic for an unknown amount of list items. But if the nornal return type is a single object, it is not obvious what to return. With PEP484 (TODO source), an optional type was introduced for type hints in Python 3. Using a static type checker for Python like mypy (TODO source) outputs a warning, if a developer forget to check an optional for not being none.

4.1.3 Condition with Comparison

The second plugin detects a direct comparison in an conditional statement. For better readability and understandability, it is suggested to use a function call with an appropriate name that evaluates to true or false. This allows a natural reading of the conditional statement without deciphering the meaning of the boolean logic.

An if-statement consists of an condition and body part. If the condition evaluates to true, the body is executed. For the condition, any logical expression will be evaluated. A simple algorithm would take the AST, search for if nodes and check, if the condition part is a compare node. But negation and logic AND/OR would not been detected by the simple approach. Consequently, a recursive algorithm has to follow logic operators and checks the expressions for comparison nodes.

Following this considerations, an advanced algorithm would scan for if nodes in the ast. The conditional part as a boolean expression is evaluated in a recursive function that returns true if a direct comparison is done anywhere in boolean expression. The conditional part can be a boolean operation (like AND/OR) or a unary operator (like NOT). In these cases, the recursive function will be called again with the respective expressions. If the conditional part is neither a boolean operation nor a unary operator, a true is returned if the conditional part is not a method call. The last case would be the base case in the recursion. Listing 4.1 shows the Python implementation of the recursion.

Listing 4.1: Recursive function to analyse an if statement for direct comparisons. Since a condition should contain a method call, the function returns False if this is not the case.

```
def _check_if_direct_comparison(self, node):
    if isinstance(node, ast.BoolOp):
        violated = False
        #check all expressions of the boolean operator
        for value in node.values:
            if self._check_if_direct_comparison(value):
                violated = True
        return violated
    elif isinstance(node, ast.UnaryOp):
        return self._check_if_direct_comparison(node.operand)

    return not isinstance(node, ast.Call)
```

4.1.4 Output Plugins

With output plugins, CCAP adds additional flexibility towards the output format. Depending on environmental requirements, the output can be addapted with custom logic by defining an output plugin. For instance, running the tool localy could write the results to the standard output in an human-readable way or create a formatted html file to be displayed in the browser. Running inside automated workflow, a machine-readable json output may be preferred.

Output plugins follow similiar concepts as analysis plugins. A plugin class inherits from *AbstractOutputPlugin*. The abstract class introduces the metadata member of the *PluginMetaData* class to encapsulate plugin name and author information. Additionally, a second member variable *output_format* has to be defined with a short abbreviation for the output format. This field is used by the the output plugin handler to select the output plugin in accordance to the run arguments. Consequently, it should be unique, otherwise, the first found plugin with a matching *output_format* field is selected by the output plugin handler. Therefore, using a custom prefix string is recommended.

As entrypoint method, the method *handle_report* has to be implemented. The method provides an instance of *FullReport* as its argument. The *report* field holds a collection of *AnalysisReport* for every analysis plugin that has been executed. With

metadata and problem information, the output plugin has access to all reuqired information to produce a desired output.

4.1.5 Steps to create an output plugin

To expand the output capabilities of the CCAP, the following steps are, similiar to analysis plugins, required:

1. Create a *.py* file with a class inheriting from *AbstractOutputPlugin*.
2. Instanziate *PluginMetaData* and assign it to the metadata member.
3. Set the *output_format* field with a unique abbreviation for this output format. Since it should be unique, a custom prefix prevents a name collision with preexisting plugins.
4. Implement the *handle_report* method with a *ParsedSourceFile* parameter.
5. Place the *.py* file into the output plugin directory of the tool.

Vorteil: läuft lokal, time to feedback ist geringer als bei cloud-basierten ansätzen wie codacy. (läuft bspw als git pre-commit hook oder direkt als VSCode plugin)

5 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

6 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 Sjoberg, 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.