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 [1]:

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)"[3].

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 [2]. 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 [2]. 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 [2]. 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 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)

2.3 Quantitative Metrics for Code Quality

Kritik an Clean Code

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

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] Robert Baggen, José Pedro Correia, Katrin Schill, and Joost Visser. Standardized code quality benchmarking for improving software maintainability. 20(2):287–307. Publisher: Springer.

[2] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

[3] ISO Central Secretary. Systems and software engineering — systems and software quality requirements and evaluation (SQuaRE) — guide to SQuaRE.