C++ Style Checker for Visual Studio Code

Bachelor Thesis

Fabian Thurnheer

&

Marco Gartmann

Department of Computer Science

OST – Eastern Switzerland University of Applied Sciences

Campus Rapperswil-Jona

Supervision:
Thomas Corbat, OST

External Co-Examiner:
Guido Zgraggen, Google, Inc.

Internal Co-Examiner:
Prof. Dr. Daniel Patrick Politze, OST

June 2021

Abstract

Programming in C++ usually relies on extensive toolchains for building, testing, and deploying applications. The fact that a given C++ source file successfully compiles does not imply that its code is of good quality and style. Moreover, it is not assured that agreed coding guidelines, which can be self-defined or well renowned, are met. Ensuring this by manual code reviews after the code passed through compilation and testing, results in a long feedback loop and negatively affects efficiency. To counteract this problem, Cevelop, a C++ IDE built by OST's Institute for Software (IFS), offers style checks that give programmers instant feedback on the code they type. While some of these checks are implemented in other IDEs and plugins as well, many of them are exclusive to Cevelop and are not available in other IDEs like Microsoft's Visual Studio Code. However, this would be desirable in the future so that students using IDEs other than Cevelop can profit from these style checks as well.

In this thesis, with the LLVM compiler project and its clang-tidy code analysis component, a feasible infrastructure was elaborated. In this infrastructure, Cevelop's style checks and new ones can be implemented in the future to make them available in Visual Studio Code. Furthermore, after an analysis of offered style checks in Cevelop, selected checks were implemented as a proof-of-concept for LLVM's clang-tidy component using the C++ programming language. In the chosen approach, all the code analysis intelligence is encapsulated in an IDE-independent language server (LLVM clangd, which includes clang-tidy). To use the implemented style checks in another IDE than Visual Studio Code, only a small plugin is needed to communicate with the language server through the Language Server Protocol (LSP). Therefore, they can also be offered in other IDEs with minimal additional effort.

This thesis laid the foundation for offering Cevelop's style checking intelligence in IDEs independent of Cevelop. Thus, users of other IDEs can be reached, and more developers can be helped to write clean C++ code. The created checks were presented to the LLVM community to be integrated into the project's code base and to make them public. Until the created style checks are integrated, a self-built executable of LLVM's clangd language server, which includes clang-tidy and the created style checks, can be used with clangd's VS Code plugin. This way, the created checks could help OST students enlisted in a C++ course to write clean C++ code and to comply with taught best practices, without being bound to Cevelop. A created developer's guide assists programmers (e.g., IFS employees) to further extend clang-tidy with style checks that would be beneficial for them or for students.

Lay Summary

Initial Situation:

Coding style guidelines are used by programmers to maintain a uniform code style and to ensure it has a good quality and meets well known best practices. Furthermore, such guidelines help to make source code more readable, and they can also improve a program's efficiency and prevent bugs. If such guidelines are not met, it is desirable that a programmer is alerted as quickly as possible, preferably as soon as he enters new code into his editor. This reduces the feedback loop and can increase efficiency significantly [1]. Development environments like Cevelop, which is developed by OST's Institute for Software (IFS), analyse the code with style checks as the programmer types and give instant feedback about violated guidelines. By doing so, Cevelop also helps OST students enrolled in a C++ course to comply with rules taught in these courses. While some of these checks are implemented in other IDEs and plugins as well, many of them are exclusive to Cevelop and are not available in other IDEs like Microsoft's Visual Studio Code. However, this would be desirable in the future so that students using other IDEs than Cevelop can profit from these style checks as well.

Approach:

To make such code analysis features available in multiple IDEs, their implementation is done in a separate component called language server. In this language server, the desired style checks can be implemented. To make style checks currently exclusive to Cevelop available in Visual Studio Code, initially, it was planned to create a language sever and to implement these checks there. However, it was evaluated that creating a new language server would take disproportionate effort. Instead, with the LLVM project, an open-source project was evaluated that already offers a language server and style checking functionality. By extending this project with Cevelop's checks, more effort can be put into the implementation of the desired code style checking functionality. After a comparison of checks existing in both projects, Cevelop style checks that were not yet available in the LLVM project were ranked by their usefulness for students, among other criteria. Selected checks were then implemented for LLVM's style checking component.

Results:

These implemented checks were contributed to the LLVM project so that its whole community can use them and profit from them. The created contribution requests are, at the time of submitting this thesis, currently awaiting approval. Until their approval, a self-built executable of LLVM's extended components can be used by OST students in Visual Studio Code to profit from the created checks. Furthermore, a developer's guide was created that lays the foundation for continuation of the work conducted in this thesis by other programmers in the future.

Table of Contents

1. Introduction

"Developing applications in C++ usually relies on an extensive toolchain including an integrated development environment (IDE), a compiler and continuous integration infrastructure for building, testing, and deploying the software." [1] The fact that a program successfully compiles does not necessarily mean that its code is of good quality and style. It is also not assured that agreed upon coding guidelines, which can be company intern or public and broadly used, are fulfilled. Both, the code's quality, and its compliance to specific guidelines need to be checked and assessed before it is deployed. To benefit a developer's efficiency, feedback on the code at hand should be given as fast as possible. Preferably, code would be checked upon being entered into the IDE and a feedback to the developer would be given instantaneously. As C++ compilation tasks are often time-consuming, reducing the feedback loop can increase efficiency drastically [1].

To achieve this, IDE plugins exist that will check and assess the entered code and give a direct feedback to developers. For instance, this feedback can be in form of wavy lines that underline problematic parts of the code at hand. Displaying diagnostic messages describing the problem and suggesting quick fixes further help the developer to achieve a good code quality and to comply with defined coding guidelines.

1.1 Initial Situation

Cevelop is a C++ IDE based on Eclipse C/C++ Development Tooling (CDT) and is developed by the Institute for Software (IFS) of the Eastern Switzerland University of Applied Sciences OST in Rapperswil-Jona. Cevelop contains plugins that implement functionality to check the style of written code and to provide feedback and quick fixes to developers. Besides utilizing well known public coding guidelines like the International Organisation for Standardization (ISO) C++ Core Guidelines, they also check the code's compliance to some rules defined by the IFS itself. The latter helps students at OST, who are for example enrolled in one of OST's C++ courses, to comply to the coding best-practices which are taught in the respective courses.

Although Cevelop's largest user group consist of students at OST, it was and might still be used by development teams in larger companies like SIX and Sonova [2]. Currently, Cevelop counts approximately 5'000 website visits per month [2].

1.2 Problem Description

While Cevelop's plugins described in Section 1.1 are helpful for students and other C++ developers, they are unfortunately exclusively available in the Cevelop IDE. Since the whole code checking functionality is encapsulated in these plugins, users of other IDEs cannot benefit from style checks and quick fixes they offer. However, it would be desirable to use these features in other C++ IDEs as well in the future [1].

1.3 Project Goals

This section describes the project's goals as they were described in the thesis's task assignment [1]. The goal of this project was to realize a proof-of-concept implementation of Cevelop's style checker functionality which is independent of Cevelop. Especially, it should be made possible for other IDEs to use this style checker functionality as well. To achieve this, a segregation of the source code analysis and the visualization of its results is needed. This segregation must be reached using the Language Server Protocol (LSP) [3], which will be described in detail later in this thesis in Section 2.1. As this thesis's

target IDE to be supported, Microsoft's Visual Studio Code was predefined. However, the usage of LSP ensures that the implemented style checking functionality can be integrated into other IDEs with minimal extra effort in the future. As part of this proof-of-concept, one of the project's main goals was to also elaborate "a feasible infrastructure to implement such functionality in the future" [1].

It should be noted that due to the project's proof-of-concept nature, it was not expected that all features of Cevelop's style checker plugins were supported at the end of this thesis. "Complete support would require in depth analysis of the C++ source code featuring an abstract syntax tree representation and a complete symbol table for the target projects." [1] Instead, to be able to test and use the project's results, the proof-of-concept was expected to implement a subset of Cevelop's source code checks. These checks should primarily be useful for students learning C++ in the context of OST's programming courses.

Besides a non-exhaustive list of existing Cevelop style checks that could have been implemented, the task assignment also proposed implementing checks for ISO's C++ Core Guidelines, since they might target a broader audience. Furthermore, it would also have been possible to define new checking features if reasonable. The checks to be implemented were to be discussed with and selected in collaboration with the thesis's supervisor.

1.4 Structure of This Report

This report describes the analysis, elaboration and implementation work done as part of this bachelor thesis as well as solutions to encountered challenges and the project's results. It is divided into the following chapters:

Chapter 2, Analysis: Reflects the research that has been done on the fundamental principles of the Language Server Protocol.

Chapter 3, Requirements: Lists elaborated functional and non-functional requirements for the chosen solution strategy.

Chapter 4, Solution Strategies: Describes evaluated solution strategies, discusses their individual strengths and weaknesses, and explains how the decision for one of them was made. Additionally, it describes how checks which were planned for implementation were elaborated.

Chapter 5, Architecture: Gives a brief overview of the architectural context of the selected solution strategy, in which the implementation work was carried out during this thesis.

Chapter 6, Checks & Fixes: Gives details about the implemented style-checking functionality. As a part thereof, encountered challenges are described and it is stated how they were overcome. Furthermore, the way the implemented checks were tested is described. In addition, this chapter introduces the elaborated workflow which served as a guideline for the implementation work of this project.

Chapter 7, Results and Conclusion: Presents and discusses this project's results. Furthermore, by-products generated by this project are pointed out. A summarization of the project's outcome forms the end of this report.

1.5 File Repositories

Two separate repositories were used to store source code and to conduct project management work and to maintain a wiki. The source code had to be stored in GitHub since the project's size exceeded the upload size limit of the GitLab instance provided by OST.

Source Code: https://github.com/mgartmann/llvm-project

Project management and wiki: https://gitlab.ost.ch/cpp-stylechecker

2. Analysis

This chapter documents research about the Language Server Protocol (LSP), which was given as a project prerequisite in the project assignment. Thereby, the fundamental principles of LSP are explained.

2.1 Language Server Protocol

A main criterion of the project assignment was the usage of the Language Server Protocol. This section describes what LSP is, how it works and what components are involved in LSP.

2.1.1 Without LSP

Traditionally, as described with Cevelop in the Introduction chapter, language feature functionalities like style checking, code completion and formatting are built into a certain IDE they are developed for. This binds their usage to the IDE and not to the programming language they support. Therefore, they are usually implemented in the same programming language as the IDE in form of a plugin. If the mentioned programming language features shall be used in other IDEs, a new plugin which reimplements this functionality must be created for each of them. Depending on the programming language for which such features should be provided, this could be time consuming and challenging. All this leads to the following three main problems [4]:

- **Plugin × IDE**: Different IDEs have their own way of presenting diagnostic messages. This prevents sharing source code of language features because no standardised interface is shared between the IDEs and the language feature functionality. This high coupling of a language's code analysis plugin to one specific IDE leads to the situation where for the same language, a separate plugin must be created for every other IDE that wants to support it. This means that if code analysis functionality for m different languages should be supported in n distinct IDEs, $m \times n$ plugins need to be created (see Figure 1). Speaking for itself, this should be avoided.
- **Implementation Language**: Cevelop's style checking functionality is developed with Java. As a result, using the same functionality inside Visual Studio Code, which is written in Typescript and runs in a Node.js environment, is not possible. A new plugin must be developed in Typescript, including the complex code analysis functionality for C++. Although developing this is possible, it is a major effort, time consuming, and an unnecessary reimplementation of functionality that already exists. Moreover, style checking, and other language feature functionalities are "usually implemented in their native programming language and that presents a challenge in integrating them with Visual Studio Code, which has a Node.js runtime." [4].
- **Performance**: Executing language features can be CPU and memory resource intensive. This could affect the performance of the IDE they are built-in, which results in an unsatisfying user experience.

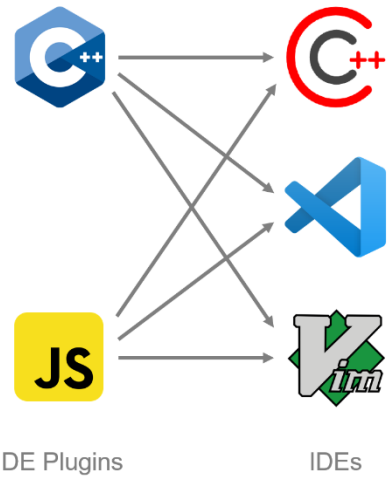


Figure 1: Plugins for Different IDEs Without Using LSP [4]

2.1.2 LSP Concept

To overcome such reimplementation, Microsoft developed the so-called Language Server Protocol. The initial idea behind it was developed during writing coding features for Visual Studio Code, which is also developed and maintained by Microsoft [5] [6]. LSP standardizes the communication between a language server (a standalone component) and a language client (usually an IDE plugin) [3]. This allows to encapsulate language feature functionalities with a universal interface into an own component. Through this, all the previously mentioned problems can be solved. As shown in Figure 2, one language server can be used with multiple IDEs to provide features like style checking, code completion or code formatting.

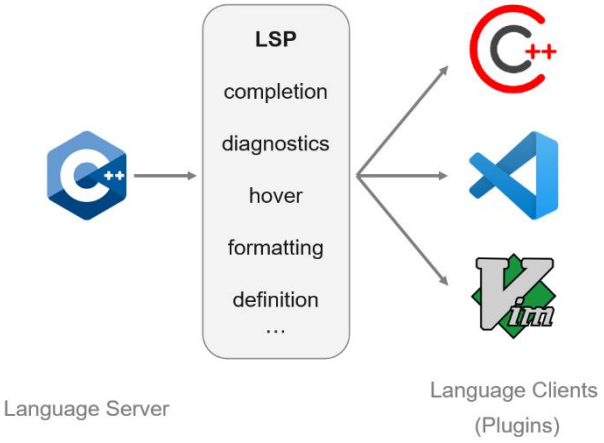


Figure 2: IDE Plugins Use a Language Server over LSP [4]

Language Server

The language server is responsible for analysing source code of a specific programming language, so it could provide different language features like style checking, code completion, formatting etc. Thanks to the language server, all the logic and intelligence needed for this is bundled in a component which can be reused in other IDEs. As described on LSP's official specification website, the language server should run in a own process and separated from the IDE [3]. This is especially useful because language analysis can be resource intensive and impact the performance of a running IDE. A language server can be implemented in an arbitrary programming language. However, it must be ensured that necessary runtime environments to run the language server are available on the target computer.

Language Client

A language client is a minimal software which connects language server functionality with a specific IDE. Besides connecting to it, it can also start the language server and it can send a shutdown request to it if its service is not needed anymore [3]. The client is commonly designed as an IDE plugin and is therefore written in the same programming language as the IDE it is developed for.

Communication

When a programmer starts his IDE with a source file of a certain language, the installed language client plugin starts a corresponding language server for the affected programming language. The communication between client and server is handled over JSON-RPC. This is a stateless protocol which uses JSON as its data format and as it is transport agnostic, it can be used to communicate over different message passing environments [7]. Due the usage of JSON-RPC and the abstraction of the code analysis features, theoretically, the client and the server do not have to run on the same computer system to communicate with each other.

A JSON-RPC object contains four members called `jsonrpc`, `method`, `params` and `id` [7]. The LSP specification describes possible remote procedure call (RPC) methods that can be called over JSON-RPC as well as valid parameters for these methods [3]. Language clients and servers utilize these method calls to trigger actions on their respective counterparts. Figure 3 shows an abstract demonstration of how a C++ language client and a corresponding language server communicate when a programmer opens his IDE with a C++ source file.

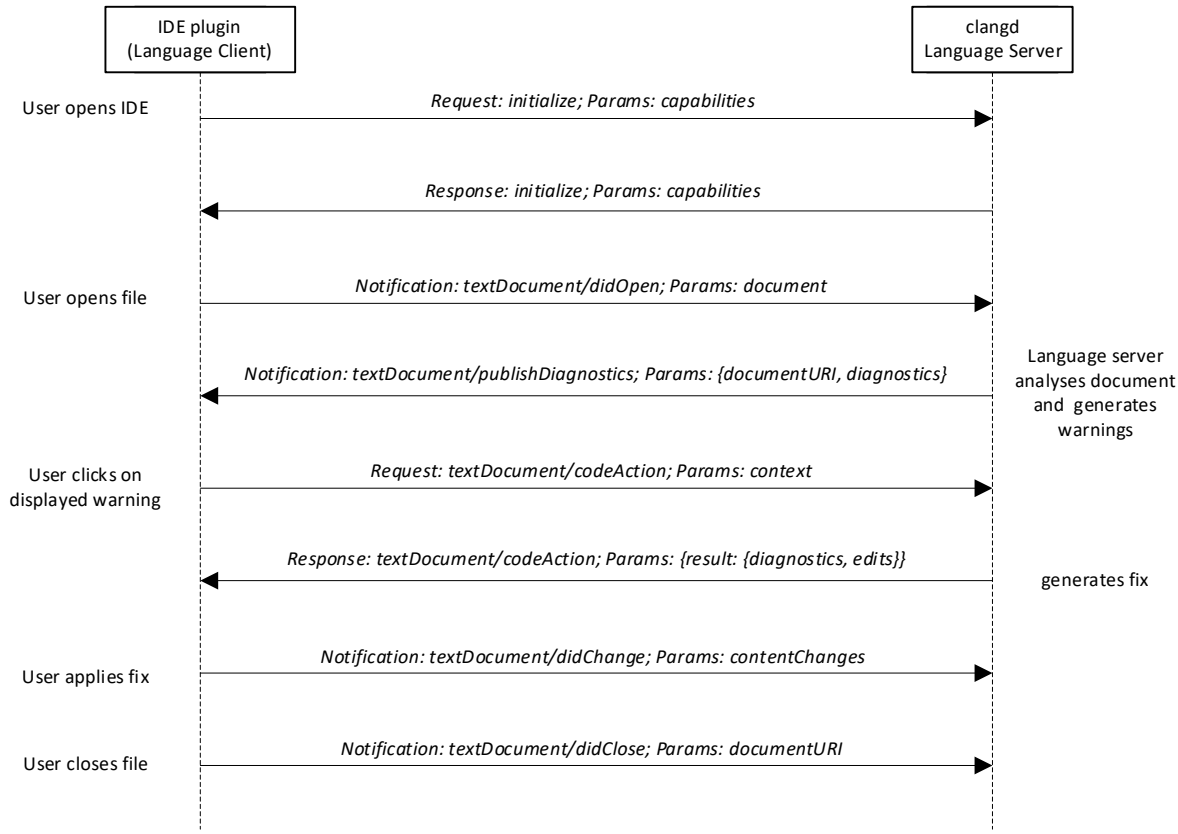


Figure 3: Communication Example Between a Language Client and a Language Server

As seen in Figure 3, the language client sends its capabilities with an `initialize` method to the server, to which the server responds with the capabilities the language server provides [3]. After that, the client sends a `didOpen` method in which the content of the C++ source file is provided as parameter. The server then analyses the given source code and detects parts of it that violate a predefined style checking rule. Subsequently, it generates diagnostic information which is sent back to the client. As the IDE receives this diagnostic information, it can display the diagnostic message in its specific style.

The programmer sees the generated diagnostic message and clicks on the affected line inside the IDE, which detects the user interaction and sends a `codeAction` method to the server. Since the server detects that there is a quick fix available for this diagnostic, a `codeAction` response is sent back to the client including the fixed code fragment. The programmer then applies the quick fix. To inform the language server about the new file content, the client sends a `didChange` method to the server, so both participants have the same content of the open file.

If the programmer decides to close the file in the IDE, the client informs the server about this event with a `didClose` method.

Figure 4 shows the `textDocument/publishDiagnostics` JSON notification which the language server sends to the client after it has analysed the given source code from Figure 3's scenario. As it can be seen, this notification contains an array of `diagnostics` objects, whose code property specifies which style check triggered this diagnostic. This information is displayed in Visual Studio Code alongside the check's `message`. Its range property defines which part of the analysed source code should be marked and to what source code range the sent diagnostic message corresponds to. Moreover, `diagnostic` objects contain a `severity` property, which distinguished the diagnostic's severity between `Error` (1), `Warning` (2), `Information` (3) and `Hint` (4) [3]. As it was found in this research about LSP, the `uri` field serves as an ID to identify the affected source code file for both the language client and the language server.

```
{
    "jsonrpc": "2.0",
    "method": "textDocument/publishDiagnostics",
    "params": {
        "diagnostics": [
            {
                "code": "cppcoreguidelines-avoid-non-const-global-variables",
                "message": "Variable 'x' is non-const and globally accessible, consider making ...",
                "range": {
                    "end": {
                        "character": 5,
                        "line": 2
                    },
                    "start": {
                        "character": 4,
                        "line": 2
                    }
                },
                "relatedInformation": [],
                "severity": 2,
                "source": "clang-tidy"
            },
             // more diagnostic objects
        ],
        "uri": "file:///c:/Users/test/Desktop/file.cpp",
        "version": 1
    }
}
```


Figure 4: `publishDiagnostics` Method Response from a C++ Language Server (clangd)

3. Requirements

In this chapter, the requirements for this project are defined. Section 3.1 lists the functional requirements whereas Section 3.2 deals with the project's non-functional requirements.

3.1 Functional

The style checking solution to be developed (including a C++ language server and a corresponding client in the form of a Visual Studio code plugin) should meet the functional requirements described in this section. All defined functional requirements refer to the processing of C++ source and header files.

3.1.1 Support of C++17

"As a style check programmer, I want to be able to analyse code written in recent versions of C++ so that the implemented style checks are also of value in code bases which use the latest C++ features."

The solution to be developed must therefore be able to analyse source code written in recent C++ versions. After consultation with the thesis's supervisor, C++17 was defined as the minimal ISO standard to support.

3.1.2 Pre-Processor Directives

"As a style check programmer, I want to analyse source code before the stage of pre-processing in order to analyse pre-processor directives like macro definitions or include directives."

The solution to be developed must therefore be able to analyse source code before the stage of pre-processing.

3.1.3 Creation of Abstract Syntax Trees

"As a style check programmer, I want to analyse compiled source code in an abstract manner in order to easily get parents, descendants etc. of a specific code part and to conduct complex queries."

The solution to be developed must therefore be able to generate an abstract representation of a given source code in the form of an abstract syntax tree (AST).

3.1.4 Semantic Analysis

"As a style check programmer, I want to analyse connections and relations between variable or function definitions and their references in order to trigger warnings for specific scenarios, e.g., depending on the location where a called function is defined."

The solution to be developed must therefore be able to conduct a semantic analysis of a given source code and to make connections between symbol declarations, definitions, and references.

3.1.5 Source Code Positions

"As a style check programmer, I want to retrieve the source code position of a specific code part in order to create fixes and diagnostic messages based on this position."

The solution to be developed must therefore provide positional information on expressions and statements of a given source code.

3.1.6 Utilizes LSP

"As a style check programmer, I want the implemented style checks to be IDE-independent so that they can be used in different IDEs without having to be re-implemented."

The solution to be developed must be usable by other IDEs besides Visual Studio Code. To achieve this, it must communicate with language clients through the Language Server Protocol, as it was defined in the thesis's task assignment (see Appendix B).

3.1.7 Allows to Implement Checks

"As a style check programmer, I want to implement new checks regularly in order to cover newly defined style guidelines."

The solution to be developed must therefore be extendable with new style checks in the future.

3.1.8 Allows to Implement Fixes

"As a style checker programmer, I want to offer quick fixes to generated diagnosis messages so that users can conveniently correct problematic code parts and are able to understand how the correct code would look like."

The solution to be implemented must provide ways to offer quick fixes alongside diagnosis messages.

3.2 Non-Functional

The project's non-functional requirements (NFRs) were determined using the FURPS+ (functionality, usability, reliability, performance, supportability, other) systematic, from which the functionality part was already addressed in Section 3.1. To achieve the result that the NFRs are both realistic and easy to agree upon, their target values were defined using agile landing zones, where appropriate. Agile landing zones reach this goal by establishing a triple of acceptable values (minimal goal, target, outstanding) instead of a single target value [8].

3.2.1 Usability

Scenario #1: Automatic Extension Activation

As soon as the user starts working on C++ files (i.e., files ending in `.cpp`), the language client plugin should be activated automatically. The user should not need to start it on his own. To benefit the general IDE performance, the extension should not be activated if only files of other types are opened.

Business goals

Provide an excellent user experience so that users appreciate the usage of the IDE's language client plugin.

Stimulus

The user who has the language client plugin installed opens a C++ file.

Response

The IDE plugin, i.e., the language client, and thus, also the language server is activated and started. Activated style checks are automatically run and their diagnostic messages are displayed automatically.

Response measure

Check if the extension is active and that the language client to language server communication is taking place as a user edits a C++ file.

3.2.2 Performance

Scenario #1: Diagnosis Does Not Affect IDE Runtime Behaviour

The communication between the language client and the language server as well as ongoing diagnosis in the language server do not affect the performance and the behaviour of the IDE in a notable manner.

Business goals

Provide an excellent user experience so that users appreciate the usage of the IDE's language client plugin.

Stimulus

User who has the IDE extension installed opens a C++ file and starts working on it.

Response

The language client (IDE plugin) sends the currently worked on file's content to the language server, which in turn analyses the received file content and sends a diagnosis back to the client. The user concurrently continues his work in the IDE.

Response measure

To measure if the IDE works unaffected of the ongoing analysis and communication, the time between a user input into the IDE and the displaying of the entered character in the IDE is determined. Table 1 lists the aspired target values in the form of agile landing zones. As defined by the Nielsen Normal Group, response times of less than 0.1 seconds are not noticed by users [9]. This should be the target. As the minimal goal, 0.2 seconds were assessed to be still acceptable. Since this requirement's target goal is already the best possible situation, no third landing zone was defined.

Minimal goal	**Target**
0.2 seconds, slight delay noticeable.	0.1 seconds, no delay noticeable.

Table 1: Agile Landing Zones for Affected Runtime Behaviour

Scenario #2: Start-up Time of IDE Extension

The IDE's language client extension must start up in a reasonable time to begin the analysis and style checking process.

Business goals

Reduce the feedback loop to increase the user's programming efficiency.

Stimulus

User who has the IDE extension installed opens a C++ file.

Response

Even if no visual feedback about its launch is shown, the IDE extension begins to communicate with the language server.

Response measure

To check if this requirement is fulfilled, the time delta between the opening of a C++ file in the IDE and the moment the language client starts communication with the language server is evaluated. The defined target values can be seen in Table 2.

Minimal goal	**Target**	**Outstanding**
$<$ 2 seconds.	$<$ 1 seconds.	$<$ 0.5 second.

Table 2: Agile Landing Zones for Start-up Time of IDE Extension

Scenario #3: Quickness of Style Checker Feedback

The user should receive feedback on his written code as immediately as possible.

Business goals

Reduce the feedback loop to increase the user's programming efficiency.

Stimulus

User who is working on a C++ file and altering its content.

Response

On modification of the file's content, the language client sends the content of this file to the language server, which analysis the received information and returns a diagnostic feedback to the client. The IDE will then make this feedback visible in the user's code.

Response measure

Critical to fulfil this requirement is the time it takes between a user input which triggers a language server reaction (i.e., a diagnostic feedback) and the moment this feedback is visualized in the IDE. Since this time may depend on a file's size, a fixed number of source lines of code (SLOC) was defined to test this requirement. The LLVM project was analysed to determine usual file sizes in a large open-source project.

For this, the SLOCs of all C++ files inside the LLVM project were determined using cloc [10]. The one C++ file which forms the 95^{th} percentile was taken as reference, meaning that 95% of all of LLVM's C++ files are smaller or have the same length.

This file (llvm-project/lld/unittests/MachOTests/MachONormalizedFileYAMLTests.cpp) counts 1'174 SLOC. However, LLVM's C++ files usually include numerous headers, which add to the file's overall size after it is pre-processed. This work has to be done by a language server (or its compiler) as well to generate the file's complete AST. Therefore, the named file was run through Clang's pre-processor, whereby all included header files were copied into the output file. Using cloc, it was determined that this resulting file contains approximately 22'000 SLOC.

A file with this number of SLOC (preferably, the file named above itself) shall be taken to verify the quickness of the developed solution. The target values are set according to Table 3.

Minimal goal	**Target**	**Outstanding**
$<$ 5 seconds.	$<$ 4 seconds.	$<$ 2 second.

Table 3: Agile Landing Zones for the Style Checker Quickness

3.2.3 Supportability

Scenario #1: Installation of the Language Client and Language Server

Users should be able to install all the required software parts in an easy way. Above all, the language server should be included in the installation process of the language client, i.e., the IDE extension.

Business goals

Get programmers to install and to use the IDE extension.

Stimulus

The user installs the IDE extension, either through the Visual Studio Code [6] extension marketplace or through an alternative installation method.

Response

Both the IDE extension (i.e., the language client), and the language server are installed. The user does not need to install the language server separately.

Response measure

As already described, the language client and the language server can be installed together. A separate installation of the language server is not needed. Since this requirement cannot be quantified any further, no agile landing zones are defined.

3.2.4 Verification of Non-Functional Requirements

During elaborating a solution strategy, its compliance to the defined non-functional requirements was assessed. This assessment's results are described in Appendix E.

4. Solution Strategies

To reach the goals described in the task assignment (see Appendix B), two possible solution strategies were elaborated, which will be covered in this chapter. First, Section 4.1 describes the possibility of extending existing code analysis tools and the idea of contributing to an open-source project. Alternatively, it would also be possible to start from scratch and to create a new language server and language client. The latter approach is discussed in Section 4.2. The conclusion in Section 4.3 summarizes each strategy's strengths and disadvantages and covers the final decision for one of the strategies. Lastly, after an applicable solution strategy was selected, Section 4.4 describes how checks were elaborated which were to be implemented with the selected solution strategy.

4.1 Extending LLVM Clang-Tidy

The reason for taking an open-source contribution into consideration is that the creation of a new language server and language client (i.e., IDE extension) involves several drawbacks. For example, a high initial effort would be needed until the first check is functional, since the language server and language client infrastructure would need to be created first. Moreover, the question arose as to how popular an IDE extension would be that (probably) covers only a fraction of existing solutions' checker functionality.

By extending an existing tool, users could benefit from its already available functionality as well as from the new extended functionality. Furthermore, needed infrastructure like a language server does not need to be self-implemented, which makes it possible to put more effort into implementing the actual style checking functionality. On the other hand, this approach also had its drawbacks. First, integrating functionality into an open-source project requires creating and managing pull-requests, which was estimated to generate extra effort. Furthermore, the benefit of the created style checks must be accepted by the project's community. There is a risk that the project's focus differs from what the Cevelop style checks focus on, and that the project's community might not approve such checks. This limits the free selection of checks to be implemented.

As a contender for the to-be-extended open-source project, LLVM's[1] clang-tidy was evaluated, which will be described in the following sections.

4.1.1 LLVM Clang-Tidy Introduction

"The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. [...] LLVM has grown to be an umbrella project consisting of a number of subprojects, many of which are being used in production by a wide variety of commercial and open-source projects as well as being widely used in academic research." [11] Clang is a C/C++/Objective-C compiler and one of the core components of the LLVM project. It serves as a frontend to the LLVM core libraries and can be used to build additional language tools on top of it [12].

In addition to Clang, the LLVM project offers extra tools (bundled under the term *clang-tools-extra*) like clangd, which is a C++ language server built on top of the Clang C++ compiler frontend [13], and clang-tidy. The latter is a C++ linter tool, which is also based on Clang, and which is responsible for the style-checking functionality of clangd's language server. Furthermore, it offers the possibility to

[1] "The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. [...] The name 'LLVM' itself is not an acronym; it is the full name of the project." [11]

propose fixes to any style problems found [14]. It can be both used as a standalone command line interface (CLI) tool as well as in combination with the clangd language server. By using it as a part of clangd, its style-checking functionality can be utilized in IDEs. Thereby, users can activate or deactivate a wide variety of existing checks and configure options for certain checks in a designated clang-tidy or clangd configuration file.

LLVM's clang-tidy was chosen as a contender for the project to contribute to because of its widespread usage. To name just a few prominent examples, clang-tidy is included automatically when choosing a C++ workload in the Microsoft Visual Studio Installer, as described on Microsoft's website: "Clang-Tidy is the default analysis tool when using the LLVM/clang-cl toolset, available in both MSBuild and CMake." [15] Furthermore, JetBrains's CLion, which is another popular C++ IDE, includes clang-tidy as a static code analysis tool by default [16]. Because of its widespread usage, LLVM and clang-tidy profit from a lively community, what was assessed to be beneficial for this thesis. What also made this open-source project interesting was the fact that a Visual Studio Code extension for clangd already existed, which includes and runs clang-tidy checks. This extension could either be used as it is or it could serve as the basis for self-implementing one if necessary.

4.1.2 Alternatives

In the process of choosing an open-source project to contribute to, two other better-known compilers and language servers were evaluated as well.

GCC by GNU is an official compiler for the GNU and Linux systems, and a main compiler for compiling other UNIX operating systems. Despite finding a mailing list article from 2017 about a proof-of-concept implementation of the LSP [17], there was no evidence found that the current production version of GCC includes an LSP implementation. Because GCC is missing a language server and does not implement the LSP (which was a requirement given in the thesis's assignment), it was not further considered as a possibility for this thesis.

Furthermore, ccls [18] was evaluated as a language server, which originates from cquery [19]. The cquery language server for C/C++ is no longer under development. Besides Clang, the cquery developers also name ccls as a valid replacement for it [19]. Despite having about 60 contributors on GitHub (as of 17.03.2021), its community seemed noticeably smaller than clangd's. Furthermore, in contrast to the LLVM project and clangd, ccls is backed by individual contributors rather than well-known tech companies, what was seen to bring the danger of API instability and an unknown future with it. As ccls itself also relies on LLVM tooling (i.e., LibClang), it was decided that it would make more sense to contribute to the LLVM project directly.

4.1.3 Contribution Possibilities

There are multiple possibilities as to how clangd and clang-tidy can be extended as part of this thesis. Most obviously, one task could be to implement an arbitrary number of Cevelop checks for clang-tidy, which are currently missing there. A comparison of existing clang-tidy checks and existing Cevelop checks (see Appendix D) revealed that 22 of Cevelop's checks were not yet implemented in clang-tidy. It should be noted that thereby, only the two Cevelop plugins GSLator and Ctylechecker were analysed. The reason for this will be explained in Section 4.4.1. Additionally, it could be useful for the clangd community if missing fixes to already available checks were created. And finally, clangd and clang-tidy developers and their communities could be asked if there were any tasks for which help is needed.

To merge contributions with the LLVM project, LLVM does not rely on GitHub pull requests but on another review tool, which is called Phabricator [20]. Based on the commit history on Phabricator and on LLVM's repository on GitHub, clang-tidy checks are regularly being added by community members.

4.1.4 Compliance with Functional Requirements

During the evaluation of clang-tidy as a possible open-source project to contribute to, it was verified that clang-tidy meets the functional requirements defined in Section 3.1. Clang, which builds the basis of clang-tidy, supports all C++17 features and already many of C++20 [21]. Furthermore, Clang allows the creation of (position related) diagnostic messages and quick fixes [22] and provides a library called `libsema`, which is responsible for semantic analysis and for building ASTs [23]. With clang-tidy, checks can be implemented which are able to analyse pre-processor directives [24]. The only functional requirement that is not directly met by clang-tidy is the utilization of LSP. However, to use clang-tidy within Visual Studio Code, LLVM's language server clangd and the corresponding Visual Studio Code plugin are used. The communication between the clangd language server and its IDE plugin happens over LSP [13]. Therefore, this requirement was also assessed to be fulfilled.

4.2 Creating a New Language Server

Apart from extending an open-source project, it would also be possible to create the language analysis infrastructure from scratch. There were two main concerns that were considered for implementing a new language server. Firstly, it had to be decided which programming language would be used to implement the language server. Secondly, the question of how the implemented language server should parse and analyse a given code file had to be answered. In the following sections, possible answers to these questions are explained and compared.

4.2.1 Implementation Language

Three programming languages were considered as an implementation language for the language server. On the one hand, C++ would have been a reasonable choice, because the language that should be analysed is C++ itself and thus, expertise would have only been needed for one programming language. Furthermore, TypeScript and C# were considered because they were already used by the project team in past projects. Table 4 shows a comparison of the mentioned languages divided into pro and contra arguments, as assessed by the project team.

Languages	**Pro**	**Contra**
C++	• The languages server could be built as a standalone executable, so no extra runtime would be needed. • Language knowledge acquired by writing checks could be used to understand the code which has to be analysed, and vice versa. • Both project members visited a C++ OST module course. Thus, a lot of lecture material could help the implementation.	• A separate compiled binary is needed for each platform on which the language server shall be run. • Project team is not very familiar with using C++ for larger projects.

Languages	**Pro**	**Contra**
TypeScript	• Node.js [25] runtime from Visual Studio Code can be used, so no extra installation is needed. • Implementation examples from Microsoft are available. • Only one language must be used in this project (VS Code plugin should be written in TypeScript). • Project team used TypeScript in past projects.	• If the language server is used with another editor than Visual Studio Code, a Node.js runtime would need to be installed additionally.
C#	• One project member used C# in a past project. • Both project members visited a C# OST module course. Thus, a lot of lecture material could help the implementation.	• Additional .NET runtime installation is needed to run the C# language server. • Implementation examples are only available from third party websites.

Table 4: Comparison of Possible Programming Languages to Implement a Language Server

Theoretically, any programming language could be used to implement the language server component, as long as the language server implements the Language Server Protocol (LSP). Since it would be cumbersome and error-prone to implement the LSP from scratch, the use of a software development kit (SDK), which implements the LSP and conveniently offers an advanced programming interface (API), was considered. Table 5 lists a benefit value analysis of the evaluated SDK options for the programming languages assessed in Table 4.

		LspCpp (C++) [26]		**Vscode-languageserver (TS) [27]**		**OmniSharp C#-LSP [28]**	
Criteria	Weight	Mark	Points	Mark	Point	Mark	Points
Project Team's Language Knowledge	15%	2	0.3	6	0.9	4	0.6
Prominence of Library Maintainer	30%	3	0.9	6	1.8	5	1.5
Quality and Amount of Implementation Examples	20%	1	0.2	5	1	4	0.8
Quality of Documentation	30%	1	0.3	4	1.2	2	0.6
Additional Components Needed[2]	5%	6	0.3	6	0.3	2	0.1
Total	**100%**	**13**	**2.0**	**27**	**5.2**	**17**	**3.6**

Table 5: Evaluation of LSP SDKs in Different Programming Languages

As seen in the *Weight* column, special attention was given to the SKD's *Library Maintainer* and its *Documentation*. An SDK without a community or organisation behind it, which constantly improves and develops it further, was seen as not reliable. Furthermore, an SDK without a proper documentation would make it tough to work with. Also, available implementation examples were considered as important because they could potentially speed up the development process. The *vscode-languageserver* SDK outperformed its competitors in this benefit value analysis by more than 1.6 points. It is maintained by Microsoft, is publicly available and provides a detailed documentation with helpful implementation examples.

Due to these analyses and a successful attempt of an implementation example, it was decided that the needed language server would be implemented with the vscode-languageserver SDK and TypeScript would be used as the implementation language. To run the application, Node.js was chosen as the runtime environment, since it is already included in Visual Studio Code, the IDE that needs to be supported.

4.2.2 Parsing and Analysing Source Code

The last decision that had to be made for this solution strategy was how the implemented language server would parse and analyse source code. Implementing functionality to parse C++ code into an abstract syntax tree (AST) would have gone beyond the scope of this thesis and the project team's knowledge. Instead, an already existing library or tool was to be used. The following sections list elaborated approaches which could be used to integrate such a library or tool into the language server. Additionally, it is assessed if they fulfil the functional requirements defined in Section 3.1.

[2] Additional components (e.g., runtimes) would be needed to run the server, provided that the user has Visual Studio Code installed.

Child Process

The LLVM project offers the C++ compiler frontend Clang, which was already mentioned in Section 4.1. This compiler can be executed as a standalone CLI tool with a specified C++ source file as its parameter. Clang then returns an AST of the specified file. As defined in Section 4.2.1, the server should be executed within a Node.js instance. From there, it should be possible to run a child process to execute Clang and to read generated ASTs from the child process's output. While trying out the described approach, the following advantages and disadvantages were found:

Advantages:

- Most of the code could be written in TypeScript.
- Clang's infrastructure is very well maintained and at the edge of C++ language standard's development. This would mean that the used language server would always be up to date with new C++ language features.

Disadvantages:

- Traversing and interpreting of the AST would have to be implemented.
- Spawning a child process for every source file analysis was assessed to be not very efficient.
- The Clang CLI tool is only able to receive complete files as a parameter. Although the language server receives changes to source files incrementally over LSP, it would have to persist each of these received changes to a file in order to pass it to Clang. This is impractical and would probably negatively affect the language server's performance.
- Also, analysing pre-processor directives can only be done by analysing the source code without sending it to the Clang executable, as they are removed in Clang's AST.

Node.js Add-On

Node.js allows importing specially compiled add-ons [29] written in C++. One approach could be to use LibTooling [30], a C++ library which is also part of the LLVM project, and which can be used to write standalone tools based on Clang. With the usage of LibTooling, an analyser add-on could be written in C++ and its functionality could be imported in the TypeScript language server. For testing this approach, a short function inside such a C++ add-on was created, which could successfully be called inside a JavaScript file. Furthermore, it was possible to read the function's return value.

Advantages:

- Traversal and interpretation of the AST could be written in C++ with strong help from LibTooling.
- As LibTooling is part of LLVM, it is well maintained and at the edge of C++ language standard's development. This would mean that the used language server would always be up to date with new C++ language features.
- Clang offers pre-processor callbacks and pre-defined AST matchers [24] [31]. Also, source code positions are available within Clang [22].

Disadvantages:

- Might be difficult to do, as it needs advanced C++ knowledge to setup such a project. Because of this, LibTooling itself was not used in this first trial.
- Barely any examples of such add-on implementations, especially ones which make use of LibTooling, were found that could help while implementing this approach.

ANTLR

Another approach would be to have a C++ parser written in the same language as the language server implementation. Therefore, the processing of the source file could be included directly into the language server code. As decided in Section 4.2.1, this parser would have to be implemented in TypeScript or JavaScript.

ANTLR [32] is a Java based powerful parser generator which can create a lexer and a parser for many languages. To do that, ANTLR needs a grammar file for the specific language and a target implementation language. On ANTLR's GitHub project, a repository with lots of predefined grammar files, including one for C++14, can be found [33]. In ANTLR's documentation, JavaScript is listed as a possible implementation language for a lexer and parser [34]. ANTLR was tested with JavaScript as target implementation language for a C++ lexer and parser. Unfortunately, only a data structure that includes some tokens generated by the provided lexer could be implemented. However, it was not achieved to generate an AST or to traverse a such.

Advantages:

- No C++ code would be needed to generate an AST and to analyse it.

Disadvantages:

- The latest C++ grammar file which could be found seems to only support C++14.
- Although the GitHub repository of ANTLR seems well maintained, this is not the case for the C++ grammar repository. Grammar files for C++17 do not exist.
- Traversing the AST and matching its nodes would have to be implemented.
- The trial was not promising. More time would have to be invested to prove if even an AST could be generated.
- No predefined functionality to extract source code positions or to analyse pre-processor directives is available.

Decision

Integrating C++ analysis functionality into an own language server was considered as not straightforward with the tested approaches. Investing more time to implementing a functional example with the generated lexer/parser from ANTLR was rated to be not worth it, since the grammar (C++14) does not fulfil the functional requirement of supporting C++17. Although Clang's AST provides source code positions and supports C++17 and above, reading out the AST output from a language server's child process is not a practicable solution, as described above. Furthermore, interpreting the AST and building a suitable traversal infrastructure would be a huge effort, compared to the generated benefit.

As a result, creating a Node.js add-on and using the power of Clang and LibTooling would be the most promising approach, although no proof-of-concept implementation which includes them inside an add-on was created yet. The only functional requirement which might not be completely fulfilled is Semantic Analysis, but it might provide enough functionality to implement style checks which do not need information spanned across multiple translation units.

4.3 Decision

After the two described solution strategies were analysed and assessed, it was decided that the first strategy, i.e., extending LLVM's clang-tidy, was more promising. The decisive factor for this was that reusing its existing infrastructure allowed starting the implementation of effective style checks earlier. If this infrastructure had to be implemented from scratch (as in the strategy described in 4.2), a great time effort would have had to be spent on this task and thus, less style checker functionality could have been implemented. Since future users mainly benefit from implemented style checks rather than from the infrastructure itself, it was assessed that by following the first strategy, the resulting product was going to be more valuable for its users. Furthermore, LLVM's infrastructure is mature and very functional. It would have made little sense to put effort into a new infrastructure that only offers a fraction of LLVM's infrastructure functionality and that would probably not be maintained further after this thesis. Additionally, with the assessed variants of the "Creating a New Language Server Approach", not all defined functional requirements could have been fulfilled.

Apart from this, other considerations led to this decision as well. For instance, by following this strategy, more users (i.e., developers) can be reached. Not only would OST students benefit from this thesis's work, but anyone using clang-tidy could activate the created checks and profit from them. Furthermore, the first strategy was seen as a great opportunity to gain experience in contributing to a large-scale open-source project and to learn from received community feedback thereby.

4.4 Evaluation of Checks

After the solution strategy of extending an existing project was selected, it was analysed and decided which checks were to be implemented in the elaborated environment. This section describes the determination of checks to be implemented as well as their prioritization.

4.4.1 Evaluation

A list of potential checks to be implemented was initially given in the thesis assignment (see Appendix B). All these proposed checks are contained within Cevelop's C++ Ctylechecker plugin. Hence, this plugin's checks were the primary implementation candidates at the beginning of the thesis. However, many of these checks are IFS best practices and do not relate to any well-known C++ coding guideline. After a first of these checks was implemented, it was evaluated that checks for rules defined in well renowned C++ guidelines would have a greater chance of being accepted by LLVM's community. Therefore, it was decided to put the focus on checks that are related to the C++ Core Guidelines [35]. Since only a few of the named plugin's checks implement a C++ Core Guideline, Cevelop's GSLator plugin was taken into consideration as well. This plugin was selected since it implements a broad selection of C++ Core Guidelines rules.

To gain an overview of the checker functionality that the named Cevelop plugins and LLVM's clang-tidy offer, a comparison of already existing checks in both projects was conducted. Summarized, from about 470 C++ Core Guideline rules (stand of 12.06.2021), the two analysed Cevelop plugins together implement 33 of them. Of these 33 checks, 17 were already covered by clang-tidy at the time of writing this report. Furthermore, six checks for non-C++-Core-Guideline rules were assessed to be not yet implemented in clang-tidy. Detailed results of the conducted comparison may be seen in Appendix D. There, it is also visible which exact GSLator and C++ Ctylechecker checks were not already implemented in clang-tidy. These checks missing in clang-tidy were considered as candidates for implementation.

4.4.2 Prioritization

To decide which of the evaluated checks should be implemented, all not-yet-implemented GSLator and Ctylechecker checks were assessed by the following two criteria:

- **Feasibility**: Determines how feasible it is to implement a given check. This factor depends on the following points:
    - What does the corresponding C++ Core Guideline demand from a check?
    - For non-C++-Core-Guideline checks, the check's function scope was estimated based on the existing Cevelop check.
    - Furthermore, it is influenced by the infrastructure that Clang offers to implement this check as follows:
        - E.g., if Clang already offers specific AST matchers needed for a check, its feasibility is rated as "simple".
        - If a new matcher function would need to be introduced or the check's enforcement was estimated to be complex, the checks feasibility was estimated to be "moderate".
        - Also, if a lot of additional functionality apart from the AST matchers needs to be implemented, feasibility was rated as "moderate" as well.
        - If new dependencies or large-scale changes would need to be introduced to clang, clangd or clang-tidy, the feasibility was rated as "hard".
- **Benefit for Students**: This criterion determines to what extent students would benefit from a check for the given C++ Core Guideline. This criterion depends on two factors:
    - First, for each potential check, it was assessed what impact disregarding the corresponding C++ Core Guideline would have. This impact was divided into three categories:
        - Readability or performance and related issues (rated as "low").
        - Program errors and error-prone code (rated as "medium").
        - And ultimately, issues leading to Undefined Behavior (rated as "high").

      The severer this impact, the higher was a check's benefit for a student estimated.
    - Secondly, the likeliness of students getting in contact with the given C++ Core Guideline was estimated and taken into account. This likeliness was estimated by the thesis team based on personal experience and was continuously discussed with the thesis's supervisor.

Depending on these criteria, the assessed checks were categorized, as it can be seen in Table 6. An exhaustive list of the assessed checks together with rationale on their categorisation can be found in Appendix D. It is important to note that before a check was implemented, its categorization was discussed with the thesis's supervisor.

	Benefit for Students			
Feasibility		Low	Medium	High
	Simple	C.44, C.37	C.45, CC, II, MACT	ES.75, C.46
	Moderate	C.83, C.84, C.85	ES.74, ES.9, C.31, SF.5, SIP	C.35
	Hard	SSI, C.20	ES.26	SF.8, MSI

Table 6: Assessment of Possible Checks to Implement

Primarily, checks which belong to the green-coloured group in Table 6 were the main targets to be implemented and were treated with the highest priority. As a secondary priority, checks in fields with a yellow filling were considered for implementation as well. Finally, checks in the red group would require disproportionate effort compared to their benefit for students. These would only have been considered for implementation if all other checks were successfully implemented.

5. Architecture

The overall goal of this chapter is to visualize the architectural context in which the defined functional requirements and the elaborated style checks were implemented. More precisely, it visualizes how different LLVM tools, that were used to realize the selected solution strategy, interrelate (Section 5.1), and gives an insight on clang-tidy's architecture (Section 5.2). Furthermore, in Section 5.3, the internal structure of clang-tidy checks is briefly explained, which might be helpful to get a better understanding of the implementation details described in Chapter 6.

5.1 Context of Used LLVM Tools

LLVM is a large project with tools for many different purposes. Figure 5 and the following sections visualize which LLVM tools were used as part of this thesis to achieve the project goals described in Section 1.3.

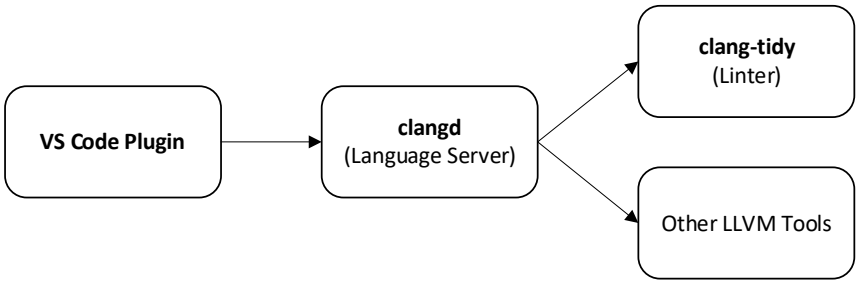


Figure 5: Overview of Used LLVM Tools

5.1.1 Clang-Tidy

Clang-tidy is known as the linter tool of the LLVM project. It already offers a high number of existing style checks, which can be extended by contributions. Clang-tidy analyses single source files against custom rules defined in so called clang-tidy checks. If such rules are violated, it creates warnings in the form of diagnostic messages. It is also possible to propose quick fixes (called `FixItHint`s in clang-tidy) to resolve detected issued.

5.1.2 Clangd

Clangd is the language server included in the LLVM project. This language server is responsible for receiving Language Server Protocol (LSP) messages and commands from a language client, to trigger the corresponding LLVM tool, and to return an answer (e.g., warnings and diagnostics) back to the language client (a more comprehensive explanation of the LSP and its messages can be found in Section 2.1). By doing this, clangd enables using a great variety of LLVM tools inside an IDE. Thanks to the LSP abstraction, to integrate tools like code completion, formatting, finding compilation errors, style checking etc. into an IDE, only a minimalistic plugin for each IDE is needed alongside the clangd language server.

5.1.3 Visual Studio Code Plugin

LLVM offers a plugin to integrate its own language server clangd into Visual Studio Code. This plugin handles the communication with the clangd language server and displays the received diagnostics inside the IDE. This plugin called "clangd" (its name overlaps with the language server's name) is available in the official Microsoft Visual Studio Code Marketplace [36].

5.1.4 Other LLVM Tools

Clangd is also able to utilize the functionality of other LLVM tools like Clang's static code analyser or clang-format, to just name two examples. While LLVM's Visual Studio Code plugin might also display warnings of such tools, they were not the main components concerned in this thesis. However, some of them were utilized in this thesis. Thanks to clang-format, which formats C++ source files according to defined rules, no consideration had to be given to formatting source code after changing it through the implemented quick fixes, for example. Thereby, a separation of concerns was achieved. To give another example of how LLVM tools are relevant for clang-tidy, clang-indexer could be utilized for checks and fixes which concern multiple translation units at once, although this was not utilized in this thesis.

5.2 Clang-Tidy Architecture Overview

Of the LLVM tools described in the previous section, clang-tidy is the most important one for this thesis because it is this tool that is going to be extended as part of the selected solution strategy. Therefore, its internal structure will be elaborated for a better understanding in this section.

Heavily abstracted, clang-tidy consists of individual checks, modules, and a core component (see Figure 6). Checks implement all the programmatic logic to define custom coding style rules and to analyse if a given source code violates this check's rule. Multiple checks are grouped and registered in modules. For instance, a module called `CppCoreGuidelinesModule` exists in which all checks related to a C++ Core Guideline rule are registered. A core component consisting of multiple files and classes forms the entry point when clang-tidy is run. It is aware of all existing modules and triggers the creation and activation of all checks that were enabled by the user.

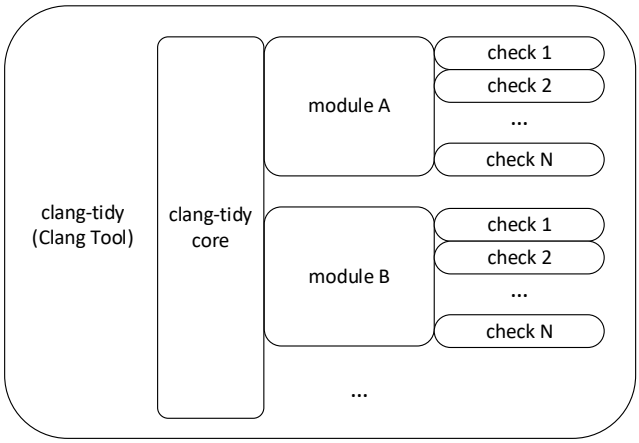


Figure 6: Relation Between Clang-Tidy Checks and Modules [37]

While the structure shown in Figure 6 was abstracted for the sake of simplicity, Figure 7 visualizes a more technically detailed overview of clang-tidy's architecture. There, it becomes visible that clang-tidy's core components are built on top of functionality offered by clang, which itself relies on LLVM's core components. The most relevant components of clang-tidy for this thesis are `ClangTidyModule`s and `ClangTidyCheck`s (marked green in Figure 3). Both are used as base classes for concrete implementations of modules (like the `CppCoreGuidelinesModule`) and its checks. To implement new checks (as it was done as part of this thesis), a new check class which derives from the `ClangTidyCheck` base class must be created. The classes contained in the `clang-tidy core` packet are, as mentioned earlier, responsible for registering and instantiating activated checks and modules.

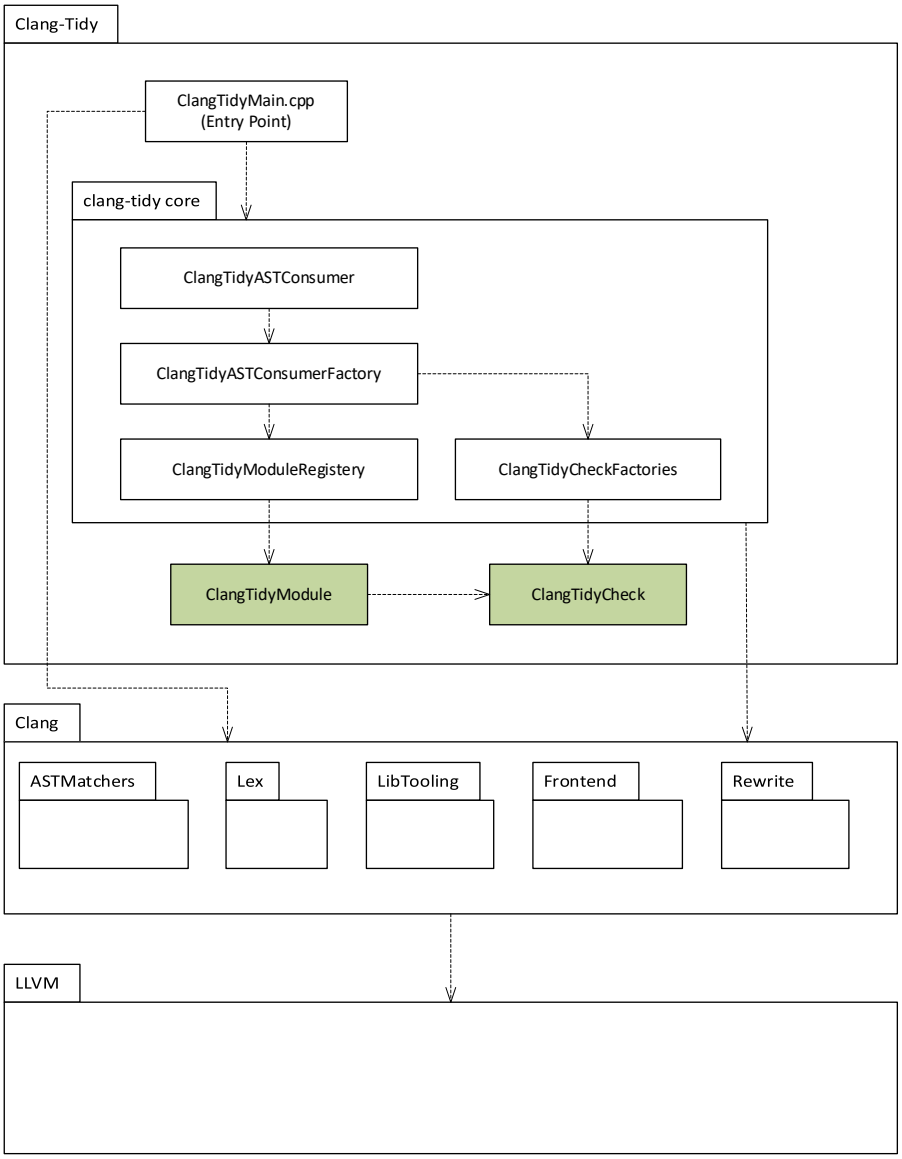


Figure 7: Abstract Architecture Layer Diagram of Clang-Tidy

5.3 Clang-Tidy Check Structure

A further zoom-in into the `ClangTidyCheck` class reveals a check's abstract internal structure, as it can be seen in Figure 8. Each clang-tidy check either has a function to register abstract syntax tree (AST) matchers, which are used to match parts of an AST, or a function to register pre-processor callbacks. The latter can be used in scenarios where parts of a source code need to be analysed which would be replaced by the pre-processor (e.g., include statements). Furthermore, each check implements a call-back function which is executed each time a registered AST matcher or pre-processor callback finds a match in a given source file. In this call-back function, the source code can be further analysed and diagnostic messages as well as quick-fix proposals can be created and returned.

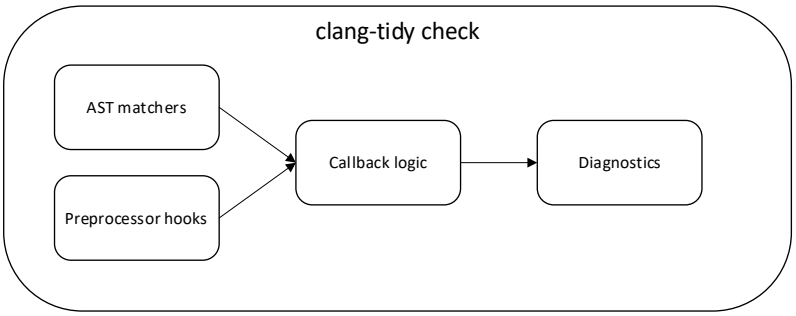


Figure 8: Internal Structure of Clang-Tidy Checks [17]

5.4 Clang-Tidy Activation Sequence

As it could be seen in Figure 7, clang-tidy consists of a variety of registries and factories which are responsible for storing knowledge about available checks and for instantiating activated checks when clang-tidy is executed for a source file. To make it understandable what these classes do when clang-tidy is executed, Figure 9 visualises the most important parts of clang-tidy's start-up. It should be noted that the leftmost component (i.e., `ClangTidy`), encapsulates logic which is part of LLVM libraries like LibTooling. To maintain a reasonable size of the diagram, this logic was abstracted and not visualized in detail.

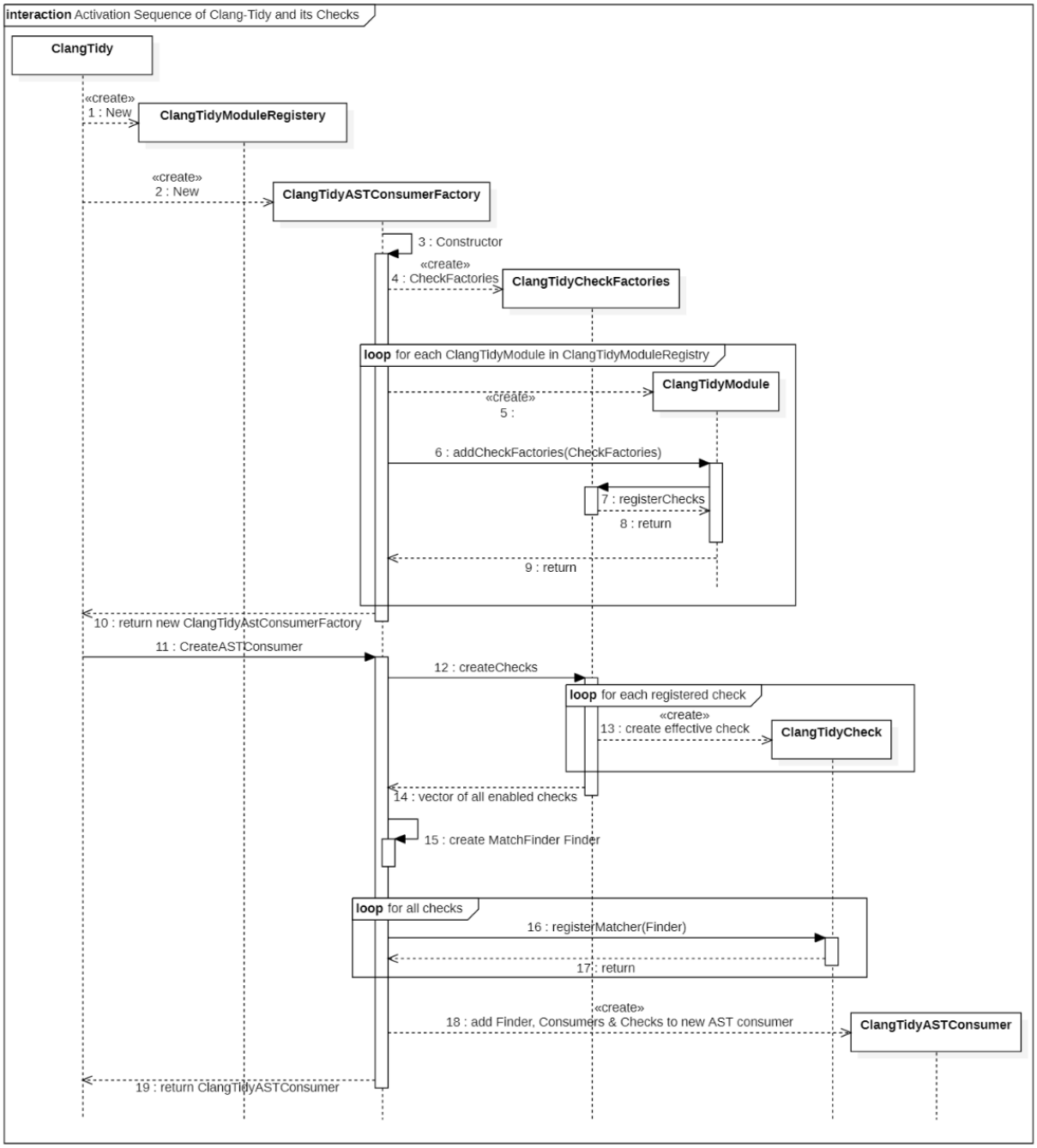


Figure 9: Sequence Diagram of Clang-Tidy Activation and Check Creation

The created `ClangTidyASTConsumer` (as seen in Figure 9) with its finder and all the activated checks will eventually process a source file. Thereby, all analysed code parts that match a check's given matcher query are registered. On a match, the corresponding check's callback logic is run to generate warnings and quick fixes.

5.5 Conclusion

The described architecture of clang-tidy provides several benefits. First, thanks to the usage of check factories, new clang-tidy checks can be implemented without having to alter clang-tidy's core component. New checks only need to be registered in their corresponding module. Because only the check's class name and its header file are needed in this registration, no adjustments are necessary if the check's code is changed in the future. Furthermore, since clang-tidy relies on libraries provided by Clang, it utilizes the same interfaces as other Clang tools. This allows clang-tidy to be integrated in Clang's language server clangd alongside other Clang tools. Lastly, clang-tidy utilizes Clang's LibTooling library, which provides functionality to run Clang tools without having to install Clang. Therefore, besides being included in clangd, clang-tidy can also be run as a standalone CLI tool.

On the other hand, this architecture also has a downside. Since style checks are a part of the clang-tidy tool itself, they are built into clang-tidy's binary, which means that a plugin architecture is not supported. Thus, it is not possible to develop checks in a separate plugin which could be integrated into the off-the-shelf clang-tidy binary. Because of this, it is also not possible to add new style checks without having to rebuild the clang-tidy binary.

Overall, the described advantages of clang-tidy's architecture overweigh its disadvantages in this thesis.

6. Checks & Fixes

The main implementation part of this thesis consisted of implementing the checks which were evaluated in Section 4.4. After the solution strategy's elaboration and the architectural context of this project were explained in earlier chapters, this chapter covers the conducted implementation work. First, it is described what Clang's AST matchers are, which form a fundamental component of clang-tidy checks (Section 6.1). Then, the defined implementation and contribution workflow is introduced (Section 6.2). Afterwards, it goes into detail on the implemented checks (Section 6.3) and fixes (Section 6.4). Since their implementation details are often very different from each other, the explanations of a coding rule's implemented check and its corresponding fixes were split into separate sections. Lastly, in Section 6.5, it will be described how the created style checks and quick fixes were tested using clang-tidy's testing infrastructure.

6.1 Clang AST Matchers

Before a file can be analysed with clang-tidy, it is compiled by Clang and an abstract syntax tree (AST) representation of the file is generated. This AST resembles the written C++ code in an abstract manner. Each of the code's declarations, statements and expressions is represented by a separate node in this AST. AST matchers form one of the most important components of creating style checks in clang-tidy. With these matchers, individual AST nodes can be matched based on their types, their attributes, their names etc.

Clang provides a large variety of predefined AST matcher functions, which can be used in combination with each other [31]. These functions form an embedded domain specific language (DSL), which can be used to create complex queries to match specific nodes of a Clang AST. An example of such an AST matcher query can be seen in Listing 1. The visualised query matches all AST nodes of type `cxxRecordDecl` (i.e., classes, structs and unions) which have a method declaration called `Foo`. In most clang-tidy checks, at least one such AST matcher query is defined to match an AST node that violates the check's style rule.

```
cxxRecordDecl ( has ( cxxMethodDecl ( hasName ( "Foo" ) ) ) )
```

Listing 1: Example of a Simple Clang AST Matcher Query

In the following sections, which describe the implementation of clang-tidy checks and fixes, such AST matcher queries will be referenced and mentioned frequently. Through the explanations in this section, these implementation details should be more understandable to readers.

6.2 Implementation and Contribution Workflow

This section explains what workflow was followed while creating a new clang-tidy check or fix during this project. After it was made sure that the check or fix intended to be implemented does not already exist, the following steps were conducted in sequence:

1. **Create branch:** On the forked LLVM project GitHub repository, a new branch for each check/fix was created. Thanks to this, the new check's branch could easily be compared to the project's untouched main branch. This simplified the process of creating a code differential for a later review on Phabricator.
2. **Create necessary files:** To create a new check, clang-tidy offers a Python convenience script called *add_new_check.py*. This script is a built-in part of the LLVM project and was not created

or altered in this thesis. Among others, the check's C++ files, a corresponding test file and documentation files are created when the script is run. In the check's C++ files, a basic class and file structure is also created by this script. Running this script formed the foundation for each implemented check.

3. **Create build files:** In the locally cloned LLVM project git repository, the project's build files had to be created. These were needed so that after implementing the check and its tests, clang-tidy and clangd could be built for manual testing. Furthermore, there is a dedicated build target to run clang-tidy's integration tests (which will be described in Section 6.5). CMake was used to create those build files from the project's predefined CMake lists. For this project, Microsoft Visual Studio was used as the build system.
4. **Exploring the AST and finding an AST matcher:** A functioning AST node matcher is the foundation of each clang-tidy check. But to be able to create such a matcher, first, it is crucial to understand the AST structure of the code that should be matched. Using clang-check, first, an AST for a problematic code snipped was generated for each check. This AST information was then used to develop the needed AST matcher query. Its correctness was validated by using clang-query and by running the developed query over the code at hand. Since clang-query can be run as a standalone tool in the shell, it was not necessary to build clang-tidy to validate the matcher query each time it was altered. Thus, the development process could be sped up rapidly.
5. **Implement the check and its tests:** After a functioning matcher was found in the previous step, the check with its matcher(s) and the corresponding tests could then be implemented in the C++ files, which were previously created by the described *add_new_check.py* Python script. The testing procedure will be explained in detail in Section 6.5.
6. **Write documentation:** Besides describing what each test does in its header file, documentation files, which are created by the *add_new_check.py* script for each clang-tidy check, needed to be filled. There, the check's goal, its justification, and an example of its behaviour on a code snippet were described. These files also form the basis of LLVM's website listing any existing clang-tidy checks [38].
7. **Review on Phabricator:** After the check functioned as intended and was formatted according to LLVM's coding guidelines, a request for feedback and approval on the newly created check had to be submitted. For this, LLVM uses a software called Phabricator. There, a differential which describes any made changes was to be uploaded. This so called "patch" was then reviewed by authorized community members. Reviews may consist of multiple review-and-enhancement cycles.
8. **Publishing of the check:** After the reviewers approved all the introduced changes, the new check could have been published. This would be done by directly pushing these changes into the LLVM project repository's main branch on GitHub. To prevent unsatisfactory code from being pushed, only selected community members have the right to perform this push. In this thesis, such a qualified person would have had to be asked to push the changes.

For a better readability, technical details as well as concrete commands were intentionally omitted in the above list. A more comprehensive workflow description, which can be used by developers continuing this project, can be found in Appendix E, Developer Guide.

6.3 Implemented Checks

This section lists the checks that were implemented as part of this bachelor thesis. It covers functional descriptions as well as implementation details and encountered challenges. In the following sections, the implemented checks are described and identified by their abbreviation (e.g., CC for Cevelop's Cin-Cout check) used in their evaluation (see Appendix D) or by their corresponding guideline number (e.g., ES.74).

6.3.1 CC - Standard Input and Output Usage Outside of Main Function

Problem Description

In C++, a simple way to write output to a screen and to obtain user input is to use the standard library's `iostream` header file. This header file defines multiple objects like `std::cin` and `std::cout`. In a source file which includes this header file, these objects can be used to generate output or to receive user input, as it can be seen in Listing 2.

The pitfall of this approach is that it negatively affects the testability of the code which uses those iostream objects. Since `std::cin` for instance is a global object, it cannot be replaced with a mocked input stream during testing [39]. Therefore, it is taught in the C++ course at OST that `std::cin`, `std::cout` etc. should only be used inside the main function. If other functions need an input or an output stream, students are encouraged to use the `std::istream` and `std::ostream` base types and to receive the stream objects as function arguments. Thus, during testing, test double stream objects can be handed to the function what enables one to inspect the input/output behaviour of the function [39]. Listing 3 visualises this recommended way of using standard stream objects.

The goal of this check is to find any uses of predefined standard stream objects (i.e., `cout`, `wcout`, `cerr`, `wcerr`, `cin`, `wcin`) and to check whether they are used inside the main function or not. If any uses are found outside the main function, a diagnosis is created for the language server in response. As a result, an IDE relying on clang-tidy checks would highlight those discouraged uses.

In answer to the feedback the check received on Phabricator, it was extended to also match C-like input and output functions like `printf`, `scanf`, `getchar` etc. These functions are defined in the C++ `cstdio` header file, respectively the `stdio.h` C header file.

```
#include <iostream>
#include <cstdio>

void some_function() {
  std::cout << "This should trigger the check.";  // NOK, output cannot be tested.
  std::printf ("This should trigger the check."); // NOK, output cannot be tested.
}

int main() {
  std::cout << "This should not trigger the check.";  // OK
  std::printf ("This should not trigger the check."); // OK
}
```

Listing 2: Improper Use of Standard IO Objects and Functions Outside `main`

```
#include <iostream>

void some_function(std::istream & in, std::ostream & out) { // OK, test doubles can be injected.
  out << "This should not trigger the check.";
  int i{0};
  in >> i;
}

int main() {
  some_function(std::cin, std::cout);
}
```

Listing 3: Encouraged Way of Using Standard IO Objects

Implementation Details

As described in Section 5.3, this check consists of two main functions. One of which is responsible for matching code parts in a source file, and a second one that examines which matcher matched and that creates a fitting diagnostic message in response to each match.

For this check, three AST matchers had to be registered to filter out all the possible occurrences of `cin`, `printf` and other unwanted objects and functions. First, all three matchers look for so called declaration reference expressions, which are represented in the matcher query by the keyword `declRefExpr`. For example, a call of a function or a reference to a variable are kinds of declaration reference expressions. Then, the matched nodes are narrowed down by their name and only such with names like `cin`, `cout` etc. are kept. Afterwards, matches are narrowed further by using more specific AST matchers. For instance, while matching global standard stream objects like `cin`, only matches whose definitions are located inside the standard namespace are kept, as it can be seen in Listing 4. Thus, less false positive matches will be criticized. For example, if a user defines an object called `cin` in his or her own namespace, it will not be criticized. This behaviour replicates how Cevelop's related style check would behave.

```
void AvoidStdIoOutsideMainCheck::registerMatchers(MatchFinder *Finder) {
  Finder->addMatcher(
      declRefExpr(to(varDecl(hasAnyName("cin", "wcin", "cout", "wcout", "cerr",
                                        "wcerr"),
                             isInStdNamespace())),
                  unless(forFunction(isMain())))
          .bind("StdStreamObject"),
      this);
  // …
}
```

Listing 4: AST Matcher for Global Standard Stream Objects

For the second use case of this check, i.e., discouraging the use of C-like functions like `printf`, two separate matchers were necessary. The reason for this is that on this check's Phabricator review, it was especially asked that not only direct function calls, but also indirect usages of the described functions should be matched. An example of such an indirect usage can be seen in Listing 5. The complexly nested abstract syntax tree of the latter use case asked for a separate matcher.

```
auto Print = &puts;
Print("This is using stdio");
```

Listing 5: Example of an Indirect Usage of C-Like Functions

Limitations

First, the check had the limitation that it matched any AST declaration reference expression node that was named `cin`, `cout`, etc. In a second version, this limitation was mitigated using Clang's AST matcher named `isInStdNamespace`. Now, only the standard library's global stream objects are matched. As described above, this behaviour was requested on this check's Phabricator review and corresponds to how the equivalent check in Cevelop would behave.

6.3.2 ES.74 – Loop Variable Declaration Outside the Initializer Part of a For-Statement

Problem Description

A loop variable in a for-statement, which is declared outside of the for-statement's initializer part, could lead to various problems. As shown in Listing 6, variable `Counter` is declared outside of a for-statement and is never used anywhere outside of it. Due to that, its scope is larger than it should be. This has also a negative effect on the readability of the given code. To address the problem in the provided example, the variable declaration of `Counter` should be moved inside the for-statement's initializer part.

```
void function() {
    auto Counter{0}; // bad, scope of the variable is larger than it should be
    for (Counter = 0; Counter < 5; Counter++) {}
}
```

Listing 6: Example for Problematic Code Violating Guideline ES.74

Possible Approaches

According to the ES.74 C++ Core Guideline, declaration reference expressions that meet all the following conditions should be considered for improvement to address this problem [40]:

- Are within a for-statement.
- Are modified.
- Their declaration is outside the for-statement.
- There are no other declaration reference expressions of the same variable declaration outside of that for-statement.

The first three of them could be implemented in a straightforward manner with provided AST matchers as shown in Listing 8.

To meet the fourth condition, no existing AST matcher could be found. The difficulty with this condition was in distinguishing if variables, which are declared outside of a for-statement but are used within it, are also used somewhere else in the source code. This would justify their declaration outside of the for-statement. To detect such variables, various approaches were tried out.

A first idea was to extend the matcher from condition three to match all declaration reference expressions which reference the matched variable declaration. During the implementation of this idea, it was found that no possibility exists to query such declaration reference expressions based on a variable declaration (e.g., with a function that could be called on the variable declaration object). Therefore, this approach could not be used to implement condition four.

Another approach could have been to start searching from a specific node and to find all descendant nodes which are declaration reference expressions, and which are not located inside the matched for-statement from condition one. A suitable starting point would have been the translation unit node. There is only one translation unit declaration in a generated AST, because it represents the root node of the tree. Therefore, every declaration reference expression would have been found. Another possible starting node could have been a compound statement that is an ancestor of the matched variable declaration. In comparison to the declaration unit, the search scope would be much smaller, because it is technically not possible that a declaration reference expression, which references the variable declaration, is outside of this compound statement.

The project team implemented the latter approach by using AST matchers. In detail, it was tried to search for the next ancestor compound statement of the variable declaration and starting from this node, to find all declaration reference expressions which are not in the for-statement and reference the matched variable declaration from condition three. During the execution it became clear that the implementation had a conceptual problem. As seen in Listing 7, the matcher inside the `varDecl`-matcher needs access to the bound variable declaration. Since this binding has not been done yet during the resolving of the inner matcher, this variable declaration could not be accessed and the whole matcher failed.

```
void DeclareLoopVariableInTheInitializerCheck::registerMatchers(
    MatchFinder *Finder) {
  Finder->addMatcher(
      declRefExpr(
          hasAncestor(forStmt().bind("ForStmt")),
          anyOf(hasAncestor(unaryOperator().bind("Operator")),
                hasAncestor(
                    binaryOperator(isAssignmentOperator()).bind("Operator"))),
          to(varDecl(
                hasAncestor(compoundStmt(hasDescendant(
                    declRefExpr(to(varDecl(equalsBoundNode("VarDecl")))))))),
                unless(hasAncestor(forStmt(equalsBoundNode("ForStmt"))))
            ).bind("VarDecl")
          )
      ), this);
}
```

Listing 7: Code Example of an Implementation Approach

Therefore, writing an own matcher, as described in Clang's documentation [31], which fits our needs was considered. The advantage of this concept, in comparison to the approach described above, is that inside such a matcher, it is possible to access the matched node [41]. Unfortunately, the implementation of this approach led to some errors and challenges. When applying the created matcher, clang-tidy always crashed. It was found that the reason for this was the usage of predefined AST matchers provided by Clang inside the self-written one. Because no solution could be found in a timely manner, the implementation of this approach was cancelled after consultation with the thesis's supervisor.

It should be mentioned that using such self-written matchers would have had a hidden downsize when working with clang-query as well. As described in the Developer Guide (Appendix E), clang-query is a useful command line tool to test AST matcher queries during the development process of a new check. Owed to the fact that clang-query does not have any knowledge of self-written matcher functions, potential matcher queries which include custom matchers could not have been tested with clang-query.

Finally, a `RecursiveASTVisitor` class was used to overcome challenges accessing the matched variable declaration node. This class inherits from a given `RecursiveASTVisitor` class as described in LLVM's documentation [42]. It uses a visitor-pattern-like approach to visit nodes inside Clang's AST. When a specific type of node is visited, a predefined function is called, which was overwritten with custom visiting logic. With this functionality, it was possible to implement condition four as desired.

Implementation Details

As already mentioned, the first three conditions could be implemented with existing AST matchers as shown in Listing 8. The matcher was split in three parts, of which each of them represents one of the conditions listed in this section's Possible Approaches section.

```
void DeclareLoopVariableInTheInitializerCheck::registerMatchers(
    MatchFinder *Finder) {
  Finder->addMatcher(
      declRefExpr(
          hasAncestor(forStmt().bind("ForStmt")),                          1)
          anyOf(hasAncestor(unaryOperator().bind("Operator")),
                hasAncestor(                                                2)
                    binaryOperator(isAssignmentOperator()).bind("Operator"))),
          to(varDecl(hasAncestor(compoundStmt().bind("Compound")),
                     unless(hasAncestor(forStmt(equalsBoundNode("ForStmt"))))) 3)
                 .bind("VarDecl"))),
      this);
}
```

Listing 8: Final AST Matcher for ES.74 Check

To cover the fourth condition, the described visitor-pattern-like approach was implemented. Because of this, a new class `OutsideForStmtVisitor`, which inherits from `RecursiveASTVisitor`, was created. As described earlier, it is possible to overwrite hook functions of the inherited class. These hook functions are executed when a specific node type is visited while traversing through the AST. In this case, `VisitDeclRefExpr` [43] was used and overwritten with custom logic, as this function is executed each time a declaration reference expression is visited. The function was implemented as shown in Listing 9.

```
private:
  bool VisitDeclRefExpr(DeclRefExpr *D) {
    if (const auto *To = dyn_cast<VarDecl>(D->getDecl())) {
      if (To == MatchedDecl &&
          !isInsideMatchedForStmt(MatchedResult, DynTypedNode::create(*D))) {
        IsOutsideMatchedForStmt = true;
        return false;
      }
    }
    return true;
  }
```

Listing 9: Hook Function VisitDeclRefExpr

The implementation of the `VisitDeclRefExpr` function checks if the currently visited declaration reference expression has a variable declaration which coincides with the one matched with the AST matcher query from Listing 8. To verify that the declaration reference expression is not inside the also already matched for-statement, a self-written function `isInsideMatchedForStmt` was created. If these two conditions are fulfilled, no more traversing is needed as it is proved that a declaration reference expression exists outside the for-statement. Since this violates the C++ Core Guideline rule, the AST traversal is stopped by returning `false` in the `VisitDeclRefExpr` function.

Possible Improvements / Known Limitations

Although the check fully implements the ES.74 C++ Core Guideline, the following enhancements were determined:

- No fix is provided. This check could be extended with a fix which automatically moves a variable declaration into the for-statement, where it is used exclusively.
- Pointer references are not covered. The check only finds variables which are modified by a unary or by a binary assignment operator.
- Variable declarations are only matched if they reside inside a compound statement. It was decided to lay the focus on local variables, since a global variable could be used in another translation unit, which makes its dependencies unclear.

Result

The first version of this check implemented the same functionality as the corresponding check from the GSLator Cevelop plugin. However, this functionality was not enough to fully cover the ES.74 C++ Core Guideline. Due to that, Phabricator reviewers demanded a complete reimplementation of the check. Thanks to the received feedback and the new implementation, the check is now even an improvement to the current one available in Cevelop's GSLator plugin.

6.3.3 C.35 – Virtuality of Base Class Destructors

Problem Description

This check addresses an issue of using inheritance and object-oriented design principles regarding the deletion of a derived class using a pointer of its base class type. This could result in Undefined Behavior if the base class's or struct's destructor is not `virtual`, as it is illustrated in Listing 10 [40]. There, it is visible that a non-virtual destructor of a base struct is no problem when a created object has the derived struct as type. Both the base struct's and the derived struct's destructors are called upon deletion of the object. However, in a polymorphic scenario, where a C++ smart pointer of type `Base` points to an object of type `Derived`, only the base struct's destructor is called. Since the base struct's destructor is not specified as `virtual`, no dynamic polymorphism happens when it is called. Hence, the destructor call is not dispatched to the destructor of the derived struct. This means that data members of struct `Derived` are not cleaned up, which could lead to an Undefined Behavior of the program.

```
// includes
struct Base {
    ~Base(){
        std::cout << "Destruction of Base. \n";
    }   // public non-virtual destructor
    // ...
};

struct Derived : Base {
    // data members
    ~Derived(){
        std::cout << "Destruction of Derived. \n";
    }
};

int main() {
    {
        std::cout << "Creating object of type Derived. \n";
        Derived foo{};
    }
    {
        std::cout << "\nCreating pointer of type Base, pointing to a Derived object. \n";
        std::unique_ptr<Base> bar = std::make_unique<Derived>();
    }
}

Will generate the following output:
Creating object of type Derived.
Destruction of Derived.
Destruction of Base.
Creating a smart pointer of type Base, pointing to a Derived object.
Destruction of Base.
```

Listing 10: Problematic Inheritance with a Non-Virtual Base Class Destructor

To circumvent this problem, the C++ Core Guideline C.35 suggests that "a base class destructor should be either public and virtual, or protected and non-virtual" [40]. Thereby, every class or struct with any virtual function is seen as a (potential) base class [40]. Specifying a base class's destructor as virtual ensures that a derived class's destructor is also called in polymorphic scenarios (as seen in Listing 10). Thus, the memory used by the derived class is cleaned up properly and no Undefined Behaviour happens.

The second part of the guideline states that destructors which are specified as protected should not be made virtual. This is reasoned by the C++ Core Guidelines on the grounds that specifying every base class destructor as virtual is unnecessary [40]. Due to the protected visibility of a base class's destructor, only derived classes can call this destructor. If an object of the base class would be created outside of this class hierarchy, they cannot be destructed, and a compiler error would be generated. Thus, a scenario in which a base class pointer points to an object of a derived class (as it can be seen in Listing 10) is not possible. Hence, it is superfluous to make the base class's destructor virtual.

Implementation Details

Prior to implementing the check, an analysis of a problematic code's AST was done to find the correct AST node that needed to be matched by this check. Through this analysis, it became visible that the uppermost nodes that needed to be matched (i.e., class and struct definitions containing a problematic destructor) were represented by a `cxxRecordDecl` node in clang's AST.

The general functionality of the planned matcher is to find all classes and structs (`cxxRecordDecls`), which have at least one member function that is virtual and whose destructor is either public and non-virtual or protected and virtual. As described earlier, a class or struct with at least one virtual function is seen as a (potential) base class by the C++ Core Guidelines [40]. Although `cxxRecordDecls` represent not only structs and classes but also unions, it was not necessary to narrow all found `cxxRecordDecls` nodes to only keep such representing structs and classes. This is because unions cannot have virtual member functions [44]. Thus, they are not matched by the resulting matcher query. The resulting matcher query can be seen in Listing 11.

To achieve the matcher expression described above (see also Listing 11), existing AST matchers, which are part of Clang's AST matcher API, were used.

Derived classes are also covered by the resulting matcher. However, they represent an edge case in this check because of the way they inherit their base class's destructor. For instance, if a derived class does not have a user-defined destructor, the implicitly compiler-generated one is publicly visible. If one of its base classes' destructors is specified as virtual (independent of its visibility), the derived class's destructor inherits this virtuality [45]. Hence, the derived class's destructor is public and virtual and is not marked by this check. Otherwise, it is marked. No special effort was needed to cover this edge case. However, it was seen as worth noting since this behaviour might not be obvious otherwise.

```
void VirtualClassDestructorCheck::registerMatchers(MatchFinder *Finder) {
  ast_matchers::internal::Matcher<CXXRecordDecl> InheritsVirtualMethod =
      hasAnyBase(hasType(cxxRecordDecl(has(cxxMethodDecl(isVirtual())))));

  Finder->addMatcher(
      cxxRecordDecl(
          anyOf(has(cxxMethodDecl(isVirtual())), InheritsVirtualMethod),
          unless(anyOf(
              has(cxxDestructorDecl(isPublic(), isVirtual())),
              has(cxxDestructorDecl(isProtected(), unless(isVirtual()))))))
          .bind("ProblematicClassOrStruct"),
      this);
}
```

Listing 11: Created AST Matcher Function for C.35

Result

For destructors that violate rule C.35, now, a diagnosis message is generated. It indicates to users that a struct or a class should either be made public and virtual or protected and non-virtual. For the user's convenience, this message also states what the destructor's current visibility and virtuality is. Additionally, if a destructor of a base class or struct is found to be private, the diagnosis message informs programmers that thus, the type will not be usable and that it should be considered making it public and virtual or protected and non-virtual.

It should be noted that for this check, quick fixes were implemented as well. Their implementation is described in Section 6.4.2.

6.3.4 ES.75: Avoid Do-Statements

Problem Description

In comparison to a while-statement, a do-statement has its condition at the end of the statement. Thus, the first iteration through a do-statement is not validated by its condition. According to the ES.75 C++ Core Guideline [40], this could lead to two major problems: readability and error-proneness. First, the condition could easily be overlooked while reading the code, because of its position at the end of the statement. Furthermore, the first iteration without checking the condition could be misleading.

Implementation Details

The ES.75 C++ Core Guideline describes the enforcement of this check with flagging all do-statements. Clang provides an AST matcher for do-statements. Because of that, the implementation was straightforward.

After uploading the check to Phabricator, a reviewer mentioned the practice of using do-statements in macro definitions. C++ programmers could potentially use do-statements to wrap multiple statements inside a macro. This prevents issues like falsely set semicolons or single line if-statements when the macro is resolved by the pre-processor [46]. Such do-statements normally have a falsy condition to not have any impact on the given code.

At the time of submitting this thesis, Clang provided no AST matcher function which could match statements that are inside a macro definition. Because of this, a custom AST matcher `isInMacro` had to be used, as seen in Listing 12 and Listing 13.

```
AST_MATCHER(DoStmt, isInMacro) { return Node.getBeginLoc().isMacroID(); }
```

Listing 12: Custom AST Matcher `isInMacro`

```
void AvoidDoWhileCheck::registerMatchers(MatchFinder *Finder) {

  Finder->addMatcher(

      doStmt(unless(allOf(

                  isInMacro(),

                  hasCondition(ignoringImpCasts(anyOf(

                      cxxBoolLiteral(equals(false)), integerLiteral(equals(0)),

                      cxxNullPtrLiteralExpr(), gnuNullExpr()))))))

          .bind("doStmt"),

      this);

}
```

Listing 13: AST Matcher for ES.75 Utilizing a Self-Written Matcher

The implementation of `isInMacro` was found in the source code of an already existing clang-tidy check [47] and could be reused.

Result

The newly created clang-tidy check now flags every occurrence of a do-statement inside a C++ source code file. If a programmer uses do-statements inside macros, for the described reason, the statements are not highlighted. Beside of this desired exception, the created matcher does fully implement the ES.75 C++ Core Guideline. Listing 14 visualizes how the check behaves.

```
#define MACRO do {} while(false) // Is not flagged
void function() {
    auto Counter{0};
    do { // Is flagged
        Counter++;
    } while (Counter < 10);
}
```

Listing 14: C++ Code Which Shows ES.75 Check's Behaviour

6.3.5 C.46: Explicit Single Argument Constructors

Problem Description

In C++, when a class object is initialized with a value of an unrelated type, a converting constructor[3] of the destination object's class can be used to initialize it. Thereby, the given value is implicitly converted into the destination class's type [48]. This behaviour might be intentional in certain scenarios, e.g., as illustrated in Listing 15. However, this conversion can also happen in scenarios where such a conversion is not intended, as it can be seen in the example in Listing 16. There, besides a `MyString` class, both a `print` function that takes a `MyString` object and one that takes a string literal exist. In this example, assume that a user of the `print` function wanted to print a one-character long string `"x"`. Accidentally, he passes the letter `x` as a character literal instead of as a string literal. Since the character type can be converted to an integer but not to a string literal, the former `print` function is called, and `'x'` is implicitly converted to an empty `MyString` object. Through this implicit conversion, surprisingly for the caller, an empty `MyString` object is printed instead of the letter `x`.

```
struct X {
  X(int);
  X(const char*);
};

void f(X arg) {
  // Here, it is intended to create an object of type X
  X a = 1;     // a = X(1)
  X b = "Foo"; // b = X("Foo");
}
```

Listing 15: Example of an Intended Type Conversion

```
class MyString {
public:
    MyString(int n){
        // allocate n bytes to the MyString object
    }
};

void print(const MyString &a) { /* ... */ }
void print(const char *a) { /* ... */ }

int main(){
    print('x'); // User intended to call: print("x");
}               // It is neither clear nor intended that a MyString object is instantiated
```

Listing 16: Example of Unintended Type Conversion

[3] Any non-explicit constructor is called a converting constructor [48]

Such unintended conversions can lead to unexpected behaviour of a program and to bugs which are, based on the thesis team's own experience, especially hard to debug for unexperienced C++ programmers.

To prevent unintended conversions, C++ Core Guideline's rule C.46 proposes to declare single-argument constructors as explicit. If a class's constructor is declared as explicit, a user is forced to explicitly cast values to this class's type, if s/he wants to convert it (e.g., by using `static_cast`). Otherwise, a compiler error is generated (which is a good thing here, since it protects programmes from unintended conversions).

Possible Approaches

In a first analysis, it was found that no current clang-tidy check implements the C++ Core Guidelines rule C.46. However, after work on its implementation had begun, it was found that Google's *google-explicit-constructor* clang-tidy check already implements parts of the given guideline rule. However, this check did not cover all parts of rule C.46's enforcement specified in the C++ Core Guidelines. Namely, the enforcement specifies that in some cases, implicit conversions through constructors might be wanted or needed. Therefore, it shall be possible to exclude such constructors and classes from generating warnings by listing them in a positive list [40].

To completely comply with the C++ Core Guidelines, one possible approach would have been to implement a new and separate check for this rule. However, since this and Google's check would have shared the same logic, this would probably have led to duplicated code (and surely to duplicated logic). Since duplicated code is a well-known code smell which should be avoided, this approach was not desirable.

Instead, a second approach in which code from Google's check was to be reused somehow was aimed for. An analysis on Google's *google-explicit-constructor* check revealed that most of the check's logic was contained in private member functions. Therefore, the possibility to include the check's header file in a new check and to reuse its functions by linking the two check's implementation files fell away. From further research, it was found that clang-tidy offers the possibility to add aliases for existing checks. These aliases may be created with a new name and in a clang-tidy module different from the existing check's module. In this rule's scenario, this meant that such an alias could be created in clang-tidy's *cppcoreguidelines* module with an appropriate name, pointing to Google's *google-explicit-constructor* check. To implement the demanded whitelisting functionality, adding a user-configurable clang-tidy option to the pointed to check was evaluated to be promising.

Implementation Details

Since the logic to match problematic constructors already existed in Google's *google-explicit-constructor* check, no initial AST analysis was necessary. Therefore also, no AST matchers had to be implemented.

Like every regular clang-tidy check, a check alias must be registered in a `ClangTidyCheckFactory` in its corresponding clang-tidy module. Normally, when creating a new check, it is automatically registered in the specified module by the *add_new_check.py* convenience script, as described in Section 6.1. In this scenario however, since only an alias was to be created for a C++ Core Guideline rule, this script could not be used and the registering of the alias in the `CppCoreGuidelinesTidyModule` had to be done manually. Thereby, the only difference in registering an alias to registering an effective check was that the pointed-to check is accessed by including its namespace. Logically, the foreign check's header file had to be included as well. Listing 17 gives an example of how this check's alias was registered. Afterwards, it was already possible to activate Google's *google-explicit-constructor* check by using its alias, *cppcoreguidelines-explicit-constructor-and-conversion*.

```
// CppCoreGuidelinesTidyModule.cpp
// ...
#include "../google/ExplicitConstructorCheck.h"
#include "AvoidGotoCheck.h"
// ...

/// A module containing checks of the C++ Core Guidelines
class CppCoreGuidelinesModule : public ClangTidyModule {
public:
  void addCheckFactories(ClangTidyCheckFactories &CheckFactories) override {
    // Registering a module-local check:
    CheckFactories.registerCheck<AvoidGotoCheck>(
        "cppcoreguidelines-avoid-goto");

    // Registering an alias to a foreign check:
    CheckFactories.registerCheck<google::ExplicitConstructorCheck>(
        "cppcoreguidelines-explicit-constructor-and-conversion");

    // ...
  }
}
```

Listing 17: Registering of an Alias to a Foreign Check

As described before, what was still needed to comply to the C++ Core Guideline rule C.46 was the possibility to ignore certain constructors from being flagged. Since the created check alias does not have any logic (and not even an own C++ file), this functionality had to be added to the pointed-to Google check. Clang-tidy options were used to allow users to specify a semicolon separated list of constructors which shall be ignored by the check. The existing check's AST matcher was extended so that upon matching problematic (non-explicit, single argument) constructors, their names are checked against this list. If it occurs in the user specified ignore list, they are not matched, and no warnings are generated.

Users of the check can define this ignore list as a semicolon separated list of class names in their clangd or clang-tidy configuration file.

Result

Thanks to the implemented alias, Google's *google-explicit-constructor* check can now also be used with its C++ Core Guidelines alias, *cppcoreguidelines-explicit-constructor-and-conversion*. This helps programmers to find the check for C++ Core Guideline C.46 where it would be expected, which is in the `CppCoreGuidelines` module of clang-tidy. Furthermore, non-explicit single argument constructors that are intentionally used for implicit conversions can now be listed in a semicolon separated ignore list. Constructors of classes contained in this list do not trigger warnings and thus, programmers have to cope with less false-positive warnings.

6.3.6 C.164: Avoid Implicit Conversion Operators

Problem Description

As already described in Section 6.3.5 for C.46's check, implicit conversions of an object from one type to another type can be intended and essential in certain cases (e.g., `double` to `int` conversion). However, they often cause surprises [40]. Apart from conversion destructors (as described in Section 6.3.5), so called conversion operators can be used as well to convert from one type to another. To prevent unwanted implicit conversions that could lead to an unexpected program behaviour and hard to spot bugs, conversion operators should be specified as `explicit` too.

Possible Approaches

During the implementation of guideline C.46's check, it was found that Google's *google-explicit-constructor* check also implements guideline C.164 nearly exactly. However, other than described in C++ Core Guideline's enforcement for C.164 [40], not all but only non-explicit conversion operators are marked by Google's check. Cross-checking this behaviour with the corresponding check in Cevelop revealed that they both function the same. Since apart from C.164's enforcement, the remaining parts of its guideline suggest preferring explicit conversion operators rather than doing without them completely, the enforcement was seen as misleading. After consultation with the thesis's supervisor, the guideline's enforcement was rewritten and the clarified one was proposed to the C++ Core Guidelines community in form of a pull-request. More details about this pull-request can be found in Section 7.3.2. The clarified enforcement now states that only non-explicit conversion operators should be flagged.

Since Google's check complies to the clarified enforcement of guideline C.164, no additional functionality had to be implemented. To make Google's check available under a name from clang-tidy's `CppCoreGuidelines` module, it was seen as practical to create an alias to Google's *google-explicit-constructor* check with a C++ Core Guidelines related name.

Implementation Details

Since Google's *google-explicit-constructor* check covers both C++ Core Guidelines C.46 and C.164 and since the topic of both are closely related, only one alias was created for both. As already described in Section 6.3.5, the created alias is called *cppcoreguidelines-explicit-constructor-and-conversion*. As C.46's section above already discusses details of how this alias was implemented, it is not described again in this section to avoid duplication. Please refer to Section 6.3.5 for further implementation details.

Additionally, functionality was added to enable users to ignore conversion operators by specifying their name in a semicolon-separated list in this check's clang-tidy options.

Limitations

On this check's Phabricator request, reviewers mentioned that templated conversion operators should also be ignorable. Listing 18 shows such a templated conversion operator and its corresponding AST. As it is visible there, the main problem of ignoring templated conversion operators is that their names in the AST are different from their names visible in the source code (`type-parameter-0-0` instead of `Ty` in Listing 18). Therefore, to ignore such conversion operators, users currently have to specify their `type-parameter-m-n` names in the check's options. These names are also displayed in Google's check's diagnostic messages and are therefore known to the user. While this works, it would be more convenient for users to use the operator's name they see in their source code. However, no practicable way was found to implement this. It was therefore decided to accept this limitation in favour of a robust and maintainable check.

```
struct Foo {
  template <typename Ty>
  operator Ty() const; // How to silence the diagnostic here?
};


// AST:
`-CXXRecordDecl 0x19bd2712470 <C:\HSR\Semester6\BA\llvm-project\..\playground\test\blank.cpp:1:1,
line:4:1> line:1:8 struct Foo definition
  |-DefinitionData pass_in_registers empty aggregate standard_layout trivially_copyable pod trivial
literal has_constexpr_non_copy_move_ctor can_const_default_init
  | |-DefaultConstructor exists trivial constexpr needs_implicit defaulted_is_constexpr
  | |-CopyConstructor simple trivial has_const_param needs_implicit implicit_has_const_param
  | |-MoveConstructor exists simple trivial needs_implicit
  | |-CopyAssignment simple trivial has_const_param needs_implicit implicit_has_const_param
  | |-MoveAssignment exists simple trivial needs_implicit
  | `-Destructor simple irrelevant trivial needs_implicit
  |-CXXRecordDecl 0x19bd2712590 <col:1, col:8> col:8 implicit struct Foo
  `-FunctionTemplateDecl 0x19bd2712878 <line:2:3, line:3:17> col:3 operator type-parameter-0-0
    |-TemplateTypeParmDecl 0x19bd2712620 <line:2:13, col:22> col:22 referenced typename depth 0
index 0 Ty
    `-CXXConversionDecl 0x19bd27127d0 <line:3:3, col:17> col:3 operator type-parameter-0-0 'Ty ()
const'
```

Listing 18: Templated Conversion Operator Alongside Its AST

Result

The implemented alias helps programmers to find Google's check for C++ Core Guideline C.164 in clang-tidy's `CppCoreGuidelines` module, where it would probably be expected. Thus, it is also clearer that this guideline is already implemented what might save a programmer from reimplementing it. Furthermore, apart from using the alias directly, Google's *google-explicit-constructor* check is also activated when a user of clang-tidy activates all *cppcoreguidelines-** checks.

6.3.7 C.45: Default Constructors that Only Initialize Data Members

Problem Description

Unexperienced developers might not primarily think about optimization and readability when defining a new class type in C++. As the thesis team knows from their own visit of OST's C++ module, creating classes is a task often done in the exercise lessons. The C++ Core Guideline C.45 [40] with its enforcement tries to support implementing a new class, especially with creating a constructor or initializing data members. According to C.45, a default constructor should do more than just initialize class member variables. If a default constructor does nothing more than that, it could be left out or be defaulted and the class member could be initialized with in-class member initialization instead. This forces the compiler to create the default constructor by itself, which the C++ Core Guideline assumes to be more efficient. This does also improve readability and prevents duplication. As seen in Listing 19, after enforcing C.45 all class data members are initialized at one place. Also, if the implicitly default constructor is used, a developer cannot initialize a class member with an in-class member initialization and a constructor list initialization at the same time which would lead to duplicated code. Furthermore, the in-class member initialization would be ignored in that case, which a developer might not be aware of [49].

Without C.45 Enforcement

```
class C {
  int x;
  int y{2};
  int z;


  // more code

  C(): x{1}, z{3} {}
};
```

With C.45 Enforcement

```
class C {
  int x{1};
  int y{2};
  int z{3};


  // more code


  C() = default; // could also be omitted
};
```

Listing 19: Cod Example to Explain C.45

When the default constructor is not defined explicitly by a developer, the enforcement could also lead to cleaner code, as no code for an unnecessary default constructor will pollute the class definition.

Possible Approaches

By interpreting the C.45 C++ Core Guideline, the following conditions should be fulfilled so that a constructor could be omitted:

- Constructor is a default constructor.
- Constructor does only initialize class members with constants.

If a constructor declaration meets these conditions, it should be flagged, and a diagnostic message should inform the programmer about the violation.

Before work on an own implementation of a check was begun, it was discovered that an existing check for the same topic exists. However, it did not implement rule C.45 exactly. The clang-tidy check *modernize-use-default-member-init* [50] does flag class member variables which are initialized by a constructor's initializer list, and which do not have an in-class initialization. Also, it provides a fix for each class member to convert the initialization from the constructor list to an in-class initialization.

Although this check does not focus on flagging a default constructor which does only initialize class members, its fix would be almost what a fix for a C.45 check would look like. The only exception is that it would be more convenient to convert all affected class members at once. While testing the found

modernize-use-default-member-init check, it was noticed that its fix does not work correctly in Visual Studio Code. For example, if the provided fix is manually applied in Visual Studio Code, the initialization value is dropped and characters like `,` or `:` from the initializer list were not deleted as they should be (as seen in Listing 19, without C.45 enforcement).

As usual for a clang-tidy check, *modernize-use-default-member-init* has a test file which contains test code to verify its functionality. After consultation with the thesis's supervisor, it was decided to find out if this test file runs successfully. If this was not the case, fixing the current existing check's fix should be considered. After analysis, no obvious error was found in *modernize-use-default-member-init*'s source code, and the executed test file worked correctly. As it was not figured out how the described problems in Visual Studio Code could be fixed, it was decided to implement a new check and a corresponding quick fix. Several advantages of implementing a new check also led to this decision. First, *use-default-member-init* does not flag a constructor declaration which violates rule C.45. Secondly, the provided fix of *modernize-use-default-member-init* is not enough for a C.45 enforcement as it does not delete or set a default constructor to `=default`. Furthermore, with *modernize-use-default-member-init*'s fix, every class member which is initialized over a default constructor's list initializer needs to be fixed manually, which is not practical. It would be more feasible if a fix on a flagged constructor declaration could convert all list initializers at once. Finally, the check is listed under the clang-tidy module *modernize*. Therefore, its functionality is not available if a developer enables the *cppcoreguidelines* style check module. And as the check does not exactly cover C.45, creating a *cppcoreguidelines* alias was not an option.

Within mind of the conditions mentioned at the beginning of this section and browsing through Clang's AST matcher reference, it was assessed that a new check to enforce rule C.45 could mostly be done with predefined AST matchers.

Implementation Details

A constructor initializes a class member variable in its initializer list or inside its body. It was decided to first lay the focus on the initializer list as this is the more widely variant and since it is also listed in the C.45 Core Guideline as its primary example.

While analysing an AST of a code example as seen in Listing 20, it was discovered that when a class member is initialized using in-class member initialization (and not by the constructor), a `CXXCtorInitializer` node is created as a child node of the given construction declaration as well. This was a challenge which had to be solved since a constructor should not be matched if only in-class member initializations are used. The first tried solution to overcome this problem was to narrow the matcher query to only match if the `CXXCtorInitializer` has a child node which is a `InitListExpr`. As a `CXXCtorInitilaizer` which was generated for an in-class member initialization does have a `CXXDefaultInitExpr` as its child instead, only the desired constructors would be matched. After further development, it was discovered that if an initialiser list member uses parenthesis instead of braces, Clang's AST handles such members differently than the ones with braces. There, the AST does not contain an `InitListExpr` for members using parentheses. Instead, a child node for the initialization value type is created (to be seen as `IntegerLiteral` in Listing 20). This prevented narrowing the matcher query to `InitListExpr` nodes, as using parenthesis is also valid to be used in an initializer list [51].

Code Example	AST Representation

`class C {`
`  int x;`
`  int y{2};`
`  int z;`
 	
`  C(): x{1}, z(3) {}`
`};`
	`-CXXConstructorDecl 0x1c56507a790 <line:9:3, col:20> col:3 ...`
	`  |-CXXCtorInitializer Field 0x1c56507a5b8 'x' 'int'`
	`
		`-InitListExpr 0x1c56507a998 <col:9, col:11> 'int'`
		`
			`-IntegerLiteral 0x1c56507a918 <col:10> 'int' 1`
			`  |-CXXCtorInitializer Field 0x1c56507a628 'y' 'int'`
			`
				`-IntegerLiteral 0x1c56507aa08 <col:16> 'int' 2`
				`  |-CXXCtorInitializer Field 0x1c56507a698 'z' 'int'`
				`
					`-CXXDefaultInitExpr 0x1c56507aa78 <col:3> 'int'`
					`  `-CompoundStmt 0x1c56507aad8 <col:19, col:20>`

Listing 20: Code/AST Example to Explain an Implementation Detail.

To finally overcome this challenge, it was decided to check if the corresponding class data member, which is initialized by the constructor in its initializer list, is already initialized with an in-class member initialization. If this is the case, the matcher query ignores the `CXXCtorInitializer` node.

While finalizing the check's matcher query, two other existing checks were found to be helpful:

- *cppcoreguidelines-prefer-member-initializer*: Finds class member initializations inside a constructor's body and provides a fix to convert them to default member initializations [52].
- *modernize-use-equals-default* (has an alias *hicpp-use-equals-default*): Replaces empty bodies of special member functions like constructors with `=default` [53].

The functionality they provide could help to implement the check and a fix for C.45. Either the ability to search if the body of a default constructor contains initializations of class members and if possible, change them to in-class member initializers. Or further, to set a constructor explicitly defaulted.

Before starting to implement a suitable fix for this check, another attempt was made to find out why the existing fixes of *modernize-default-member-init* do not work as expected. It turned out that the fixes do work, but only when executing clang-tidy as standalone tool with the parameter `--fix`. This explains why no error in the code could be found and why the test file could be executed successfully. However, inside Visual Studio Code, the check's fixes do still not work as expected.

This discovery and the two found existing checks mentioned above changed the future outcome of this check. As discovered, apart of flagging a constructor declaration which violates C.45, applying all fixes of the mentioned checks step by step successfully enforces C.45. A possible approach for the C.45 check could be to copy the source code of the mentioned checks to provide the same functionality. Since this would mean that code duplication between the checks would occur and the maintainability of the copied functionality would be worsened, this could not be a solution. Instead, the current check was submitted to Phabricator with a question asking about how the situation should be handled. Possible answers of the community might be a recommendation to activate all the three found checks or proposal to extend one of them and to integrate the other ones.

Additionally, it was tested if a *cppcoreguideline* alias could be created which combines three different checks under one check name. No satisfying result could be achieved, since only the last registered alias is adapted. After a research inside the source code of the function that registers an alias, it was found that this behaviour seems to be intentional by design.

Result

The current version of the check was contributed to Phabricator. It successfully flags default constructor declarations which have at minimum one list initializer and an empty function body (which means that they do not do more than initializing a class member with the list initializer). No fix was implemented because clang-tidy already contains checks which implement most of the needed logic. Also, finding

initializations inside the function body is covered by one of those checks. In favour of less code duplication and better maintainability, the check was not further extended, and no fix was implemented.

Since some functionality which this check could use already exists in other checks, as described above, it was asked on Phabricator how to proceed with the implementation of this check. The community's answer could be interesting for further check implementations, where needed functionality is also found in existing checks. However, until the end of this project, no answer to this question was received. Furthermore, a detailed bug report for the malfunctioning *modernize-default-member-init* fix inside Visual Studio Code was created on clangd's GitHub repository, so that this check's fixes can be used within Visual Studio Code in the future.

6.3.8 SF.5: Include Definitions

Problem Description

When designing a C++ program, a common concept is to use header files. This means that a C++ source file has a corresponding header file which includes declarations of the C++ file's implementation. Other C++ files which need to use the functionality of a C++ file can then include its header file. A main advantage of this is that the implementation behind a header declaration can be exchanged during link time without having to change other source code that depends on it. Also, since only the source files which were changed since the last build must be rebuilt, build time of a project can be reduced.

The SF.5 C++ Core Guideline describes that a C++ file must include header files which define its interface [40]. This is especially useful to detect consistency errors while working with the described concept. As the left side of Figure 10 shows, the function defined in `bar.h` does not have the same return type as the one implemented in `bar.cpp`. Because `bar.cpp` does not include `bar.h`, the files `bar.cpp` and `main.cpp` will compile successfully. However, during link time, the linker will throw an error as the implementation of `function` does not have the same signature as the one `main.cpp` includes.

As seen on the right side of Figure 10, if `bar.cpp` does include the header file like described in SF.5, the error could be detected at compile time, as the compiler will detect the same function name with two different return types.

Including No Headers with Declarations

```
// bar.h
int function();

// main.cpp
#include "bar.h"

int main() {
  int x = function();
  return 0;
}

// bar.cpp
char function() {
  return '1';
}
```

Including Headers with Declarations

```
// bar.h
int function();

// main.cpp
#include "bar.h"

int main() {
  int x = function();
  return 0;
}

// bar.cpp -> does not compile!
#include "bar.h"

char function() {
  return '1';
}
```

Figure 10: Code Examples to Explain SF.5

Possible Approaches

The check *Missing Include to Own Header* contained in Cevelop's Ctylechecker plugin tries to enforce this rule by checking if a C++ file includes a header file with the same name (`bar.cpp` should include

`bar.h`). Additionally, the check verifies that a corresponding header file exists in the user's workspace. If no such header file is present, the check will not generate a warning.

Indeed, this alone does not guarantee that a `bar.h` header file defines all implementations done by `bar.cpp`. Nevertheless, after consultation with the thesis's supervisor, it was decided that implementing a check which behaves the same way as the one from Ctylechecker would be desired [54].

To implement the functionality of Cevelop's *Missing Include to Own Header* check in clang-tidy, it had to be clarified if the criteria mentioned above could be fulfilled with clang-tidy.

First, it should be determined if an include directive inside the analysed file with a name like `<file-name>.h` exists. The usual approach to this, as described in section 6.1, would be to use AST matchers. However, in this case, this is not possible since an include directive is replaced by the pre-processor. Thus, it is not visible in an AST and cannot be matched with AST matchers. Fortunately, Clang provides a `PPCallback` class which contains a callback function called `InclusionDirective` [24]. This function is invoked when the pre-processor is processing an inclusion directive like `#include` or `#import`. As the callback has a parameter which contains the included file's filename, this filename could be used to compare it with the analysed file's name. To do such a comparison, the name of the analysed file must be extracted. Because the `PPCallback` class is handed a `SourceManager` object upon its construction, with its help, the filename can be extracted inside the `InclusionDirective` function, as shown in Listing 21.

```
void NewCheckPPCallbacks::InclusionDirective(SourceLocation HashLoc, /* ... */) {
  /* ... */
  auto filename = PPCallbacks::sourceManager.getFilename(HashLoc).str();
  /* ... */
}
```

Listing 21: Code Example of How to Extract a Filename

If it is determined that no corresponding header file is included, it would be desirable if the check contains a fix to include it. From research, it was found that Clang provides a class called `IncludeInserter` [55] which exactly provides this functionality.

Furthermore, the check needs access to the programmer's workspace path to search possible header files which could be included. LSP supports parameters like `rootUri` (deprecated) or `workspaceFolders` for its `initialize` method which contain the user's workspace path [3]. As described in Section 2.1.2, the `initialize` method is sent from the client to the server for propagating its capabilities to which the server responds with his own. But as it was seen in clangd's log output during an analysis, the server's response does not contain a `workspaceFolders` parameter, which leads to the assumption that it may not use the provided information.

After further research, it was seen in the source code of the Visual Studio Code plugin for clangd, the programmer's workspace path is set inside the server's options when starting the clangd server [56]. Nevertheless, accessing a workspace path provided by clangd represents a general problem either way since clang-tidy has to work as a standalone tool as well. Therefore, its checks must not depend on information which is only available when using clang-tidy in combination with clangd. To resolve this problem, there should be a common path source which is available in both scenarios.

Two such common sources were found. First, clangd and clang-tidy both support reading a `compile_command.json` file. This file can be generated, e.g., with CMake, and contains all C++ files of a C++ project, including their individual build commands. However, there are two downsides to this. First, the `compile_command.json` file does not contain the project's header files by default. Furthermore, as discussed with the thesis's supervisor and based on the thesis team's own experience, students enlisted

in OST's C++ courses do usually not use such a JSON-file in their course exercises. Its usage would be exaggerated for course exercise projects.

The second discovered common source are clang-tidy options. A clang-tidy check can be configured over check options which can be set in a configuration file. In an attempt, it was achieved to recursively print all filenames inside a given directory path by making use of a LLVM internal class called `llvm::sys::fs::recursive_directory_iterator` [57]. By setting a workspace path manually in the clang-tidy check options, such a path value could be used to search for header files in it. However, configuring each workspace path by hand just to support one clang-tidy check would be very unhandy. Furthermore, handling absolute or relative paths was also seen as an obstacle with this approach. Furthermore, after a discussion with the thesis's supervisor, it was determined that such a solution would most likely not be appreciated by the LLVM community.

Result and Limitations

It could not be achieved to implement a check which works like the corresponding one from Cevelop's Ctylechecker plugin. The reason for this is the described limitation of not having access to the IDEs workspace inside of clang-tidy, while also being able to run clang-tidy as a standalone tool without clangd. The found common sources for workspace paths were classified as insufficient. However, the documentation of this approach shows that limitations do exist when using an IDE-independent language server, compared to implementing style checking functionality directly in an IDE.

6.4 Implemented Fixes

Apart from the checks listed in Section 6.3, fixes to already existing and to newly created checks were also implemented. This section lists and describes these implemented fixes.

To create fixes and fix-hints, clang-tidy relies on Clang, which offers a `FixItHint` class in its `Diagnostic.h` header file. This class contains functions to create fixes and to alter the given source-file. The following functions are offered:

- `CreateInsertion`: inserts a given code string at a specific location.
- `CreateInsertionFromRange`: like `CreateInsertion`, inserts a range of code in a source file instead of a string.
- `CreateRemoval`: removes a given source code range.
- `CreateReplacement`: replaces a given source code range with a given code string.

6.4.1 Fix for I.2 – Avoid Non-Constant Global Variables Check

In clang-tidy, a check that reports violations against ISO's C++ Core Guideline I.2 *Avoid-non-const-global-variables* already existed. This section describes how this check was extended with fixes as part of this thesis.

Problem Description

As described in rule I.2 of ISO's C++ Core Guidelines, "non-const global variables hide dependencies and make the dependencies subject to unpredictable changes." [40] Therefore, declaring global variables as non-constant is discouraged. However, the existing check did not offer any fixes, which could be applied by the user to make the concerned variables constant.

These fixes were implemented. Their goal is to change matched global and non-constant variables to constant ones. Listing 22 shows a is-should comparison in form of two code extracts. There, also the AST trees corresponding to the code extracts are visible. These ASTs formed the basis for implementing the described fixes.

```
/// "Is" situation:

int x;

int *y = &x;

int &z = x;


// The associated AST looks as follows:
| ...
|-VarDecl 0x1ef4fb6b418 <C:\nonconstglobal.cpp:1:1, col:5> col:5 used x 'int'
|-VarDecl 0x1ef4fb6b518 <line:2:1, col:11> col:6 y 'int *' cinit
| `-UnaryOperator 0x1ef4fb6b5a0 <col:10, col:11> 'int *' prefix '&' cannot overflow
|   `-DeclRefExpr 0x1ef4fb6b580 <col:11> 'int' lvalue Var 0x1ef4fb6b418 'x' 'int'
`-VarDecl 0x1ef4fb6b608 <line:3:1, col:10> col:6 z 'int &' cinit
  `-DeclRefExpr 0x1ef4fb6b670 <col:10> 'int' lvalue Var 0x1ef4fb6b418 'x' 'int'

/// Should be changed to this by applying all fixes:

const int x;

const int *const y = &x;

const int &z = x;

// The associated AST then looks as follows:
| ...
|-VarDecl 0x25ccdb0f5b0 <line:8:1, col:11> col:11 used x 'const int' callinit
| `-ImplicitValueInitExpr 0x25ccdb0faa8 <<invalid sloc>> 'const int'
|-VarDecl 0x25ccdb0fb18 <line:9:1, col:23> col:18 y 'const int *const' cinit
| `-UnaryOperator 0x25ccdb0fba0 <col:22, col:23> 'const int *' prefix '&' cannot overflow
|   `-DeclRefExpr 0x25ccdb0fb80 <col:23> 'const int' lvalue Var 0x25ccdb0f5b0 'x' 'const int'
|-VarDecl 0x25ccdb0fc08 <line:10:1, col:16> col:12 z 'const int &' cinit
| `-DeclRefExpr 0x25ccdb0fc70 <col:16> 'const int' lvalue Var 0x25ccdb0f5b0 'x' 'const int'
```

Listing 22: AST Dump of Simple Non-Constant Global Variable Declarations

Functional Requirements

On the basis of the existing check, there are three general use cases for which a fix must be provided:

- Firstly, a global variable declaration must be made constant by inserting the *const* qualifier either to the left or to the right of the variable's type specifier.
- Secondly, a pointee's type of a global pointer should be made constant.
- And lastly, if a global reference variable references a non-constant variable, the type which is referenced should also be made constant.

Possible Approaches

The existing check already implements all the logic needed to match the problematic non-constant variables in the source code. To provide fixes to the detected problems, the following two approaches were evaluated.

Firstly, to provide the described fixes, it would have been sufficient to add `FixItHint`s to the diagnosis messages already emitted by the check. Briefly summarized, a `FixItHint` contains information on where in a source file to insert or to replace an arbitrary text. Currently, the described diagnosis messages only provide users with a warning about the problematic variables in their source code. By appending a `FixItHint` to them, either the missing `const` keyword could be inserted into the source code as text (using `CreateInsertion`, as described at the beginning of this section), or the non-constant variable's type could be made constant in a more abstract manner. With the latter, the `CreateReplacement` function lent itself to replace the old, non-constant type with the new, constant type. Although implementing the former solution was expected to be simpler since the required API functions were already known, the latter was to be preferred. By only instructing clang to make a variable's type constant, the correct location to insert the `const` keyword did not need to be evaluated manually, as it would have had to be done in the text-insertion approach. Furthermore, especially with handling pointer-type variables, the first solution proved to be error-prone and impractical in a quick experiment.

Another approach would have been to extend the `TransformerClangTidyCheck` class to create a clang-tidy check which uses so called `RewriteRules` to alter the source code. This class makes use of Clang's LibTooling `Transformer` class. Since there already is an existing check depending thereon, no new dependencies would have had to be introduced to clang-tidy by using this approach. However, it was assessed that this approach would not bring any benefit over the one described above. This is because in the end, `TransformerClangTidyCheck` only extracts the source code matchers and modification instructions from a given `RewriteRule` to then create a `FixItHint`. This coincides with would have been done anyways in the previously described approach.

In addition to not offering a great benefit, the `TransformerClangTidyCheck` would also require rewriting the check's existing implementation. This bears the risk of breaking existing code, and it was assumed that the community would be less keen on accepting such a change. Therefore, the approach that includes writing `FixItHints` directly was elected to be implemented.

Implementation Details

As described in the Possible Approaches part of this section, it was decided to make a variable's type constant in an abstract way instead of inserting the `const` keyword on its own. The benefit of this approach is that Clang takes care of placing the `const` keyword in the right place itself. This abstractness was achieved through the usage of an API function called `addConst()`, which is available on the `Type` property of matched variables.

After a matched variable's type was made constant, a string representation of the now constant type was used to replace the variable's original type. One challenge thereby was the fact that a type's string representation may not only contain what is visible in the source code to a programmer, but also additional type-information. An example of such a case can be seen in Listing 23, where the effective type of `bar` is `class Foo` and not only `Foo`, as it was expected. It should be noted that despite the superfluous `class` keyword, the code illustrated in Listing 23 is syntactically correct. However, this cannot be guaranteed for all possible cases of types with superfluous type-information. To ensure syntactical correctness for all possible type replacements, a dedicated function was implemented which removes any superfluous type-information. This function utilizes Clang's `PrintingPolicy` class, which was used to supress most of the superfluous type-information while generating a type's string representation. Its use for this scenario was advised in a response to a question asked by the thesis team on LLVM's Discord server. The remaining unwanted type-information (e.g., `unnamed` keywords for unnamed structs), which were not removed by the `PrintingPolicy`, was listed in a C++ vector and was then erased from the type's string representation.

```
class Foo {};
Foo bar{};
// For Clang, bar's type is "class Foo".
// Without measures, too much information would be inserted into the source code by the fix:
const class Foo bar{};
      ~~~~~
```

Listing 23: Example of a Type That Needs To Be Shortened

In addition to the implementation of the fixes, the tests belonging to this clang-tidy check were extended as well. They now also verify that the implemented fixes alter the source code correctly. Furthermore, while implementing the fixes, a not yet tested scenario of unnamed global structs was discovered. A new test-case was created for that scenario. Details on how tests for clang-tidy checks were written can be found in Section 6.5.

Result

Thanks to the implemented fixes, clang-tidy's *cppcoreguidelines-avoid-non-const-global-variables* check is now able to replace a variable's type in all cases listed in Listing 24 with their constant equivalent. For the sake of brevity of this section, Listing 24 contains only an extract of all possible cases the fixes can correct. An exhaustive list can be found in the check's test file contained in the LLVM project code base [58].

Before

```
int a{0};
int *b = &a;
int &c = a;

class Foo {};
Foo bar{};

struct D
    int x;
} e{};

struct {} f{};
```

After Fixes Are Applied

```
const int a{0};
const int *const b = &a;
const int &c = a;

class Foo {};
const Foo bar{};

const struct D {
    int x;
} e{};

const struct {} f{};
```

Listing 24: Resulting Functionality of the Implemented Fixes for C++ Core Guideline I.2

6.4.2 Fix for C.35 – Virtuality of Base Class Destructors

For ISO's C++ Core Guideline C.35 called "A base class destructor should be either public and virtual, or protected and non-virtual", a check has already been implemented as part of this thesis, as it was described in Section 6.3.3. Afterwards, to make it convenient for users to resolve the found deviations to this guideline, the check was extended with fixes. For the sake of brevity of this section, the details of guideline C.35 will not be described again. For this, please refer to Section 6.3.3.

Functional Requirements

To solve all problems with wrongly specified destructors (according to ISO's C++ Core Guideline C.35), the to be implemented quick fixes mainly need to insert or remove `virtual` keywords. What a quick fix needs to do depends on a destructor's combination of its visibility and its virtuality, as it can be seen in Table 7.

	Virtuality		
Visibility		Virtual	Non-Virtual
	Public	Complies with C.35, no actions needed.	`virtual` keyword needs to be added to the destructor declaration.
	Protected	`virtual` keyword needs to be removed from the destructor declaration.	Complies with C.35, no actions needed.
	Private	The destructor must either be made `public` and a `virtual` keyword needs to be inserted, or it needs to be made protected and the `virtual` keyword needs to be removed if present.	

Table 7: User-Defined Destructors and Actions Needed to Comply with C.35

Whereas Table 7 only describes scenarios for user-defined destructors – in other words, destructors that are explicitly declared by the user and visible in the source code – different actions need to be taken for destructors that are implicitly generated by the compiler. To fix implicit destructors with a wrong combination of virtuality and visibility, a destructor with correct specifiers needs to be inserted into the source code. The way in which the new destructor must be inserted depends on a set of conditions, which are visualized in Figure 11.

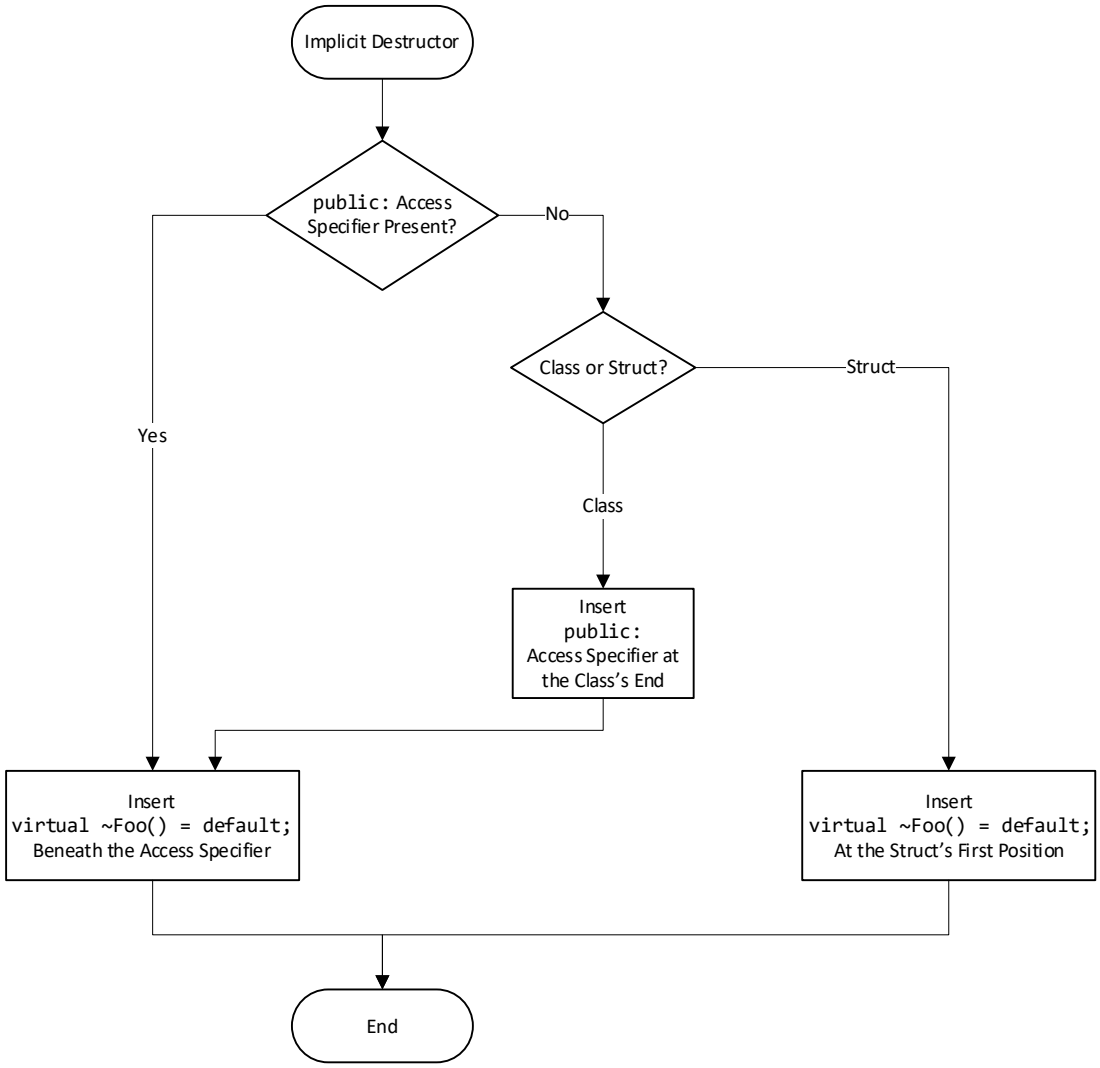


Figure 11: Flowchart of Fixing Implicit Destructors to Comply with C.35

Possible Approaches

To implement fixes for C.35, the same possible approaches existed as described in Section 6.4.1 for guideline I.2's fixes. To briefly recap them, on the one hand, it would have been possible to generate `FixItHints` which could be attached to diagnosis messages of checks. On the other hand, it would have also been possible to extend the `TransformerClangTidyCheck` class and to use `RewriteRules` to alter code. However, the latter approach would not have been compatible with the check which was already implemented for C.35 since this was created with the `registerMatchers` function and not with `RewriteRules` (see Section 6.3.3). Thus, choosing this approach would have led to a reimplementation of this check. Furthermore, `RewriteRules` themselves also create `FixItHints`, so this approach was assessed to not bring any benefit that would have justified the check's reimplementation. Therefore, the `FixItHint` approach was selected to be implemented.

Implementation Details

If user-declared destructors are already present in the source file at hand, to fix them, the `virtual` specifier simply has to be removed from the code if superfluous or added to it if needed. This was achieved by using Clang's `CreateInsertion` and `CreateRemoval` API functions. When removing the keyword, care was taken that not only the unwanted `virtual` keyword, but any following whitespace is

removed as well. Despite being challenging, this could be achieved by utilizing clang's `Lexer` class. With its lexer functionality, the start location of the token following the `virtual` keyword could be found. Everything in between these two tokens is whitespace and is deleted as well, leaving a correctly formatted source file behind.

As mentioned above, for classes and structs that have an implicit constructor, an additional line of code containing the correctly specified destructor needs to be inserted. This was more challenging to achieve, since the insertion location and what needs to be inserted depends on multiple conditions, as it can be seen in Figure 11. The code extracts in Listing 25 aim to explain this challenge visually.

```
// Before fixes are applied:
struct Foo {

    virtual void f();
};

class Bar {
    virtual void f();



};

class Baz {
    virtual void f();
public:

    int x{0};
};
```

```
// Must look like this after applying the fixes:
struct Foo {
    virtual ~Foo() = default;
    virtual void f();
};

class Bar {
    virtual void f();
public:
    virtual ~Bar() = default;
};

class Baz {
    virtual void f();
public:
    virtual ~Baz() = default;
    int x{0};
};
```

Listing 25: Before and After Comparison of Fixing Implicit Destructors

To determine if a `public:` access specifier is already present, an iterator provided by Clang (`DeclContext::specific_decl_iterator<AccessSpecDecl>`) was used to loop over all the access specifiers of a class or struct. With this information, the decision tree introduced in Figure 11 was implemented and a string containing the needed tokens was created. After calculating the correct insertion position, this string could then be inserted into the source code using Clang's `CreateInsertion FixItHint` function.

Moreover, it was achieved to offer quick fixes for private destructors. To do so, a dedicated function was created which can both make a private destructor public and virtual as well as protected and non-virtual. Briefly explained, the function takes an access specifier keyword (`public` or `protected`) and creates a string which included this access specifier followed by the original destructor code (where the `virtual` keyword is added or removed, based on the destructor's target visibility). Furthermore, the string's last part is the private access specifier, which is needed to keep everything beneath the destructor private that was previously private as well. Clang's `CreateReplacement` API function was used to create a fix which replaces the affected code parts. One challenge that had to be overcome was that the user should be able to choose whether he wants to make the destructor public or protected with a quick fix within the IDE. To achieve this and to give the quick fixes displayed in the IDE meaningful descriptions, the

offered fixes and the displayed warning message were separated. Whereas the latter is appended to a Clang diagnostic message of type `Warning`, for each of the two possible fixes, a diagnostic message of type `Note` was created. Through this, a Visual Studio Code user can hover over a flagged code part to see the generated warning message and to choose from one of the two possible quick fixes, as it can be seen in Figure 12.

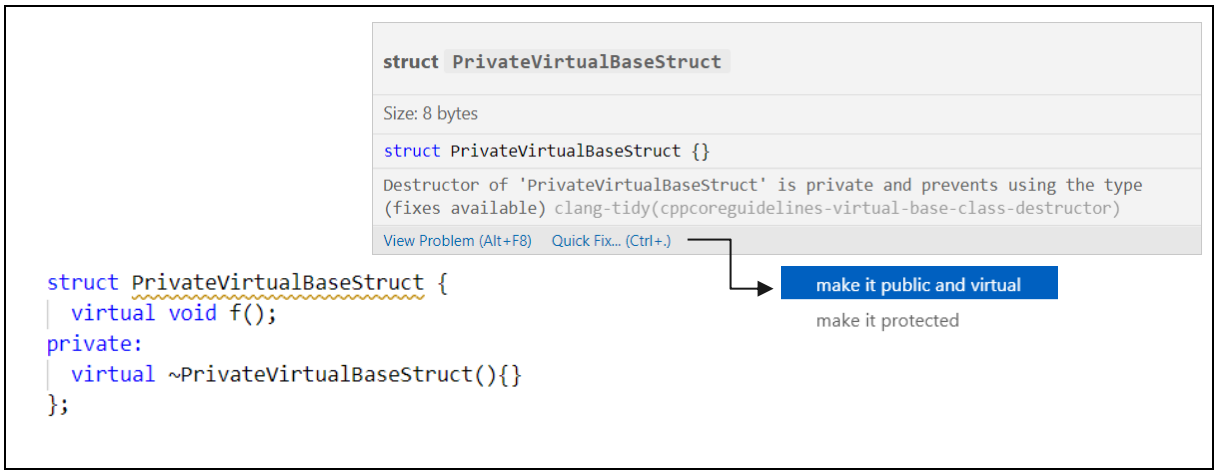


Figure 12: Selection of Multiple Fixes for C.35

Limitations

For one thing, as explained above, a user-declared destructor is inserted for classes and structs that do not have one already and which fall into the category of marked classes. This leads to the fact that a class's or struct's move constructor and its move assignment operator are no longer implicitly declared if they were implicit before [59]. This would be problematic if the code at hand used one of these constructors or assignment operators before the source code was altered by this check's fixes. In this case, the compiler would try to call the class's copy constructor or assignment operator, what would only be problematic if the class had move-only members (e.g., unique pointers) which cannot be copied. However, this case was estimated to be not very likely. Furthermore, there already exist clang-tidy checks (e.g., *cppcoreguidelines-special-member-functions* for C++ Code Guidelines rule C.21) that indicate missing move constructors and assignment operators to the user [58]. Thus, it was decided to leave it to the user to declare these move constructors and assignment operators if needed.

Result

The implemented fixes are now able to correct the virtuality of classes' and structs' user-defined destructors, that were wrongly specified before (public and non-virtual or protected and virtual). Furthermore, for classes and structs that had an implicit destructor generated by the compiler with a wrong visibility-virtuality combination, a public and virtual user-declared destructor is inserted. Thereby, a public access specifier is inserted as well if needed. Moreover, a special effort was put into assuring that the inserted code is indented correctly. Implemented user-overwritable options allowed the user to specify its desired indentation width. However, since feedback received on this fix's Phabricator request pointed out that indentation should be handled by clang-format, this functionality was ultimately removed again. A brief overview over the capabilities of the implemented fixes can be seen in Listing 26.

User-Written Code:

```
struct ProtectedVirtualStruct {
    virtual void f();
protected:
    virtual ~ProtectedVirtualStruct(){};
};

class PublicNonVirtualClass {
    virtual void f();
public:
    ~PublicNonVirtualClass(){};
};

class PublicImplicitNonVirtualClass {
  virtual void f();
};



struct PrivateVirtualBaseStruct {
  virtual void f();
private:
  int m = 0;
  virtual ~PrivateVirtualBaseStruct(){}
  int n = 0;
};
```

After Fixes Were Applied:

```
struct ProtectedVirtualStruct {
    virtual void f();
protected:
    ~ProtectedVirtualStruct(){};
};

class PublicNonVirtualClass {
    virtual void f();
public:
    virtual ~PublicNonVirtualClass(){};
};

class PublicImplicitNonVirtualClass {
    virtual void f();
public:
    virtual ~PublicImplicitNonVirtualClass() = default;
};

struct PrivateVirtualBaseStruct {
  virtual void f();
private:
  int m = 0;
protected:
  ~PrivateVirtualBaseStruct() noexcept {}
private:
  int n = 0;
};
```

Listing 26: Resulting Functionality of the Implemented Fixes for C++ Core Guideline C.35

6.5 Testing

This section describes how the created clang-tidy checks and fixes were tested. In Section 6.5.1, a description of how tests for clang-tidy were created and run is given. Furthermore, their results are discussed as well.

6.5.1 Clang-Tidy Integration Tests

LLVM's clang-extra-tools, to which clang-tidy belongs, contain an integrated test suite. One part of clang-tidy's folder in this suite is a Python script named *check_clang_tidy.py*, which will run the available integration tests. To understand how the integration tests work, it is best to take a step back and to recall what clang-tidy does with a C++ source file. Basically, clang-tidy performs linting based on activated clang-tidy checks on a C++ file. If any of the checks match, it will generate a diagnostic warning and will apply available fixes.

Therefore, to create an integration test for a clang-tidy check, a test file is needed which contains code that should trigger the check to generate a warning and code that does not. The expected clang-tidy warnings are written as C++ comments near the triggering line of code. Moreover, if fixes are expected to be applied, they are noted as comments as well. Furthermore, the first line of the file is always a comment that instructs LLVM's test runner to run the *check_clang_tidy.py* script on this file. Also, this comment activates the clang-tidy check that should be tested. The Python script then runs the activated check on the given test file and compares the expected clang-tidy warnings and fixes with the ones generated while running the test. If they match, the test is considered successful.

Something that was discovered while writing tests for checks is that for expected fixes, it does not matter where the comment is written which specifies the expected fixed code fragments. Other than with expected warnings, no exact line is specified on which the fix is expected to be applied. What the described Python script does is to gather all the expected fixes from the test file's comments and to conduct a textual search of the expected fixed code parts on the fixed source file. If code is found that coincides with what was expected to be applied by a fix, the test passes. The test also passes when such a code part was already present in the source file from the beginning and if no fixes were applied! Therefore, it is important to describe the expected fixes as specifically as possible, for example by including distinct class names in these comments. Only then can it be assured that the tests work as expected.

An example of how an integration test file for a clang-tidy check looks like can be seen in Listing 27. This test file belongs to the created *virtual-class-destructor* check, which was described in Sections 6.3.3 (check) and 6.4.2 (fixes).

```
// RUN: %check_clang_tidy %s cppcoreguidelines-virtual-class-destructor %t -- --fix-notes

struct PublicVirtualBaseStruct { // OK
  virtual void f();
  virtual ~PublicVirtualBaseStruct() {}
};

// CHECK-MESSAGES: :[[@LINE+2]]:8: warning: destructor of 'ProtectedVirtualBaseStruct' is prote...
// CHECK-MESSAGES: :[[@LINE+1]]:8: note: make it protected and non-virtual
struct ProtectedVirtualBaseStruct {
  virtual void f();

protected:
  virtual ~ProtectedVirtualBaseStruct() {}
  // CHECK-FIXES: ~ProtectedVirtualBaseStruct() {}
};

// CHECK-MESSAGES: :[[@LINE+2]]:8: warning: destructor of 'PublicNonVirtualBaseStruct' is public ...
// CHECK-MESSAGES: :[[@LINE+1]]:8: note: make it public and virtual
struct PublicNonVirtualBaseStruct {
  virtual void f();
  ~PublicNonVirtualBaseStruct() {}
  // CHECK-FIXES: virtual ~PublicNonVirtualBaseStruct() {}
};
```

Listing 27: Example Integration Test File from a Clang-Tidy Check

The LLVM Visual Studio solution contains a project which can be used to run all clang-tools-extras' integration tests, called check-clang-tools.

In Listing 28, a summary of all clang-tidy integration tests can be seen. It is visible that in total, five tests failed. However, none of the tests and checks resulting from this thesis is listed under the failing tests, which means they worked as expected. Since these tests run successfully on Phabricator's build platform and since they have no relation to the checks implemented in this thesis, no time was spent on investigating why these tests failed locally.

```
115>********************
115>Failed Tests (5):
115>  Clang Tools :: clang-apply-replacements/ClangRenameClassReplacements.cpp
115>  Clang Tools :: clang-apply-replacements/basic.cpp
115>  Clang Tools :: clang-apply-replacements/format.cpp
115>  Clang Tools :: clang-move/move-used-helper-decls.cpp
115>  Clang Tools :: clang-tidy/infrastructure/export-diagnostics.cpp
115>
115>
115>Testing Time: 173.76s
115>  Unsupported      :    9
115>  Passed           : 1231
115>  Expectedly Failed:    2
115>  Failed           :    5
115>********************
```

Listing 28: Integration Tests Results of All Clang-Tidy Checks

7. Results and Conclusion

This chapter summarizes the thesis's results and explains what has been achieved and what still can be done in the future. Section 7.1 describes the resulting product and the project's output. Furthermore, in Section 7.2, the solution's compliance to the defined functional and non-functional requirements is pointed out. The work conducted in this thesis also led to some side-effects and by-products, which are elaborated in Section 7.3. Finally, Section 7.4 concludes this report by going into this thesis's achievements, by reflecting on the evaluated solution and its limitations, and by giving an outlook.

7.1 Resulting Product

According to the contribution workflow described in Section 6.1, a new differential was created on Phabricator for each implemented check and fix (as seen in Table 8) and thereby, their inclusion into the LLVM project was requested. Each of the created Phabricator requests received feedback, which was used to improve the checks and fixes. The updated versions are now waiting for their review again. At the time of this thesis's submission, none of the created requests have been accepted yet and they are still waiting for their final approval. It is assumed that the reason for this is that clang-tidy's community members, which can be volunteers but also employees of big tech companies, might be busy with work that is more fundamental for clang-tidy. As seen in other Phabricator contributions, it is not uncommon for clang-tidy checks to take several months to be accepted. Moreover, the received feedback shows that clang-tidy's community indeed is interested in including the work conducted during this thesis in their project.

Check or Fix	**Phabricator Request**
CC	https://reviews.llvm.org/D99646
C.35	https://reviews.llvm.org/D102325
C.45	https://reviews.llvm.org/D104112
C.46 & C.164	https://reviews.llvm.org/D102779
ES.74	https://reviews.llvm.org/D100092
ES.75	https://reviews.llvm.org/D102576
I.2	https://reviews.llvm.org/D100972

Table 8: List of Created Phabricator Requests

Once all contributed work is integrated, every Visual Studio Code user will be able to use the created checks and fixes by downloading the clangd Visual Studio Code plugin. Since this plugin also includes the clangd language server (and thus, also clang-tidy), no additional components need to be installed, which is very convenient.

To make the created checks and fixes usable until they are integrated into the LLVM project, a self-built executable of clangd was created and handed in at the end of this thesis. In the settings of clangd's Visual Studio Code extension, a custom path can then be set to point to the self-built clangd executable. Afterwards, users can activate the created checks and are able to already profit from their feedback.

7.2 Fulfilment of Requirements

Through the evaluation and the selection of clang-tidy as the project, which was extended in this thesis, all functional requirements defined in 3.1 could be fulfilled, as it was already described in Section 4.1.4.

Moreover, the style checks and fixes which were planned to be implemented can also be seen as additional functional requirements of this project. As discussed in Section 4.4.2, these checks and fixes were

prioritized according to their feasibility and to what extent students would profit from them. Furthermore, it was decided that checks which cover existing C++ Core Guideline rules were to be prioritized. As it can be seen Table 9, all these prioritized checks were successfully implemented. Additionally, checks for guidelines ES.74 and C.164 (not listed in Table 9, since this check was a by-product of C.46) were implemented. Also, quick fixes could be implemented for checks I.2 (check already existed) and C.35. Furthermore, a check for ES.74 was implemented, and implementation of guideline SF.5 was begun. However, it was discovered that due to limitations of clang-tidy, it was not possible to implement the latter in this infrastructure in the intended way.

	Benefit for Students			
Feasibility		Low	Medium	High
	Simple	C.44, C.37	C.45 ✓, CC ✓, II, MACT	ES.75 ✓ C.46 ✓
	Moderate	C.83, C.84, C.85	ES.74 ✓, SF.5 (✓), ES.9, C.31, SIP	C.35 (with fix) ✓ I.2 (fix only) ✓
	Hard	SSI, C.20	ES.26	SF.8, MSI

Table 9: Visualization of Implemented Checks and Fixes

Through selecting LLVM as the open-source project, which was to be extended in this thesis, all the defined non-functional requirements (see Section 3.2) were met. LLVM's clangd language server and its corresponding Visual Studio Code plugin excelled at the conducted NFR tests (see Appendix E).

7.3 Side-Effects

Besides the results described above, this project also had some side-effects and by-products. By working with the LLVM project, we researched a lot in its documentation and worked with existing source code. Thereby, we stumbled upon some minor bugs, which we corrected and contributed. These side-effects are described in the following sections.

7.3.1 LLVM Improvements

Corrected Clangd Documentation

While trying out the clangd Visual Studio Code plugin for the first time, we had problems creating a configuration file. This occurred even though the configuration file was structured as described in clangd's documentation. After some research, it was revealed that the website of clangd's documentation was missing an important keyword. A corrected version of the documentation site was contributed to the GitHub repository of clangd's website. Also, an open issue from a user who was asking about the same problem could be closed after this contribution [60].

While creating a new clang-tidy check or using most of the available clang-tidy check modules, it could be useful to activate all possible checks in clangd's configuration file. By doing this, it transpired that the way this could be done is not the same as clangd's documentation implies. A pull request to the GitHub repository of clangd's website, which should clarify this edge case, was contributed [61].

Reported a Found Bug in a Clang-Tidy Check

As described in section 6.3.7, an existing checks' fix does not work as expected. When applying the fix in Visual Studio Code on a code example, the resulting code is not satisfying. However, when applying the fix running clang-tidy as stand-alone tool, it worked as expected. A bug report was uploaded to the clangd Visual Studio Code plugin's GitHub repository describing this issue [62].

Corrected Clang AST Matcher Reference

While implementing new clang-tidy checks and fixes, a typo in clang's AST Matcher Reference documentation was found and corrected (`cxxRcordDecl` was changed to `cxxRecordDecl`). A Phabricator differential was created to publish these fixes [63].

7.3.2 Clarification of C++ Core Guidelines

Rule C.164 of the C++ Core Guidelines stated that implicit conversion operators should be neglected and that such which are specified as explicit should be used [35]. However, the rule's enforcement part stated that all conversion operators should be flagged. In the project team's and the thesis supervisor's opinion, this enforcement did not coincide with what the rest of rule C.164 stated. Therefore, the enforcement was rewritten and clarified during this thesis. A pull-request was opened on CppCoreGuideline's GitHub repository to make this change available for the public [64].

7.4 Conclusion

In this thesis, with the LLVM compiler project and its clang-tidy style checking component, an applicable infrastructure was elaborated in which the IFS can implement their style checks in the future. Furthermore, already existing Cevelop style checks can be ported to this infrastructure, as this thesis's proof-of-concept implementations showed. Thereby, a reasonable foundation for offering Cevelop's style checks in Visual Studio Code was laid. Besides Visual Studio Code, the elaborated style checking tool clang-tidy is already integrated in other popular IDEs as well, which makes Cevelop's style checks available to an even broader group of users [65]. Since the used language server (i.e., clangd) utilizes the LSP, the created checks can also be integrated into not yet supported IDEs with appropriate effort in the future.

By comparing existing checks in clang-tidy and in Cevelop's GSLator and Ctylechecker plugins, it was assessed which of Cevelop's checks were already implemented in clang-tidy and which would need to be implemented. For seven of the assessed Cevelop style checks, their counterpart was implemented in clang-tidy, covering eight coding style rules in total. For one implemented and one existing check, accompanying quick fixes could be implemented as well. However, to be able to provide Cevelop's complete style checking functionality in other IDEs, a lot of additional Cevelop checks would have to be implemented in clang-tidy.

To fully support Cevelop's complete style checking functionality in clang-tidy, the following work would have to be done in a continuation of this project:

- Implementation of remaining assessed checks
- Consideration of further Cevelop plugins

Implementation of remaining assessed checks: as already mentioned, seven of the assessed Cevelop style checks were implemented in clang-tidy. There are 15 checks left from Cevelop's GSLator and Ctylechecker plugins which are not yet implemented in clang-tidy (as it can be seen in Appendix D). If desirable, these checks could be implemented in the future.

Consideration of further Cevelop plugins: in this thesis, only checks of Cevelop's GSLator and Ctylechecker plugins were compared to existing clang-tidy checks and considered for implementation. However, Cevelop features many other style checker plugins. If clang-tidy should represent an equivalent alternative for OST students, one would have to clarify which of these plugin's checks are already implemented in clang-tidy. Furthermore, checks not yet existing in clang-tidy would have to be implemented there.

Ultimately, it should be mentioned that during this thesis, it was found that the selected approach also has some limitations. On one hand, it was found that some of Cevelop's implementation possibilities are not given in clang-tidy. For instance, while implementing a check for C++ Core Guideline SF.5, it could not be achieved to access the user's workspace directory from clang-tidy. While style checks integrated into Cevelop have direct access to the IDE's workspace, no feasible solution for this was found using clangd and clang-tidy. Furthermore, the approach of contributing to an open-source project limit one's implementational freedom. In Cevelop, the IFS can implement and quickly deploy any style check which is of value to them. On the other hand, in clang-tidy, its community decides which checks shall be included and which are not of value for the project. The review procedure is time consuming (in this thesis, about 40% of a check's effort went into its review cycles) and bears the risk of checks being denied by the community. To overcome this problem, new checks could be offered in a self-built clang-tidy executable as an interim solution. However, to prevent having a concurrent version of clang-tidy which had to be updated and distributed manually, contributing the implemented checks to the LLVM project should be aspired.

Nevertheless, we are still convinced that extending clang-tidy was the right decision. With its Clang-based features including predefined AST matchers, creating new checks is doable even for programmers without in depth compiler knowledge. Furthermore, the LLVM infrastructure behind it is well-maintained and new C++ standards are continuously adopted. With clangd, LLVM provides a powerful LSP language server off-the-shelf. Thus, a clang-tidy contributor does not have to worry about maintaining a language server to offer style checks over LSP. Finally, the already mentioned widespread usage of clang-tidy makes a contributed style checks much more valuable, since many developers using different IDEs can profit from them.

List of Figures

List of Tables

List of Listings

Glossary

Abstract Syntax Tree
: a tree representation of the abstract syntactic structure of source code written in a programming language. [Wikipedia] 2, 9, 18, 27, 30, 33

C++ Core Guidelines
: a set of guidelines issued by ISO which aim to help C++ programmers to write simpler, more efficient, more maintainable code. Consists of ca. 470 guidelines. [ISO] 1, 2, 21, 39, 43, 44, 45, 53, 67, 98

Cloc
: CLI tool to count blank lines, comments and physical liens of source code in many programming languages. 12

CMake
: CMake is an open-source, cross-platform family of tools designed to build, test and package software. [CMake] 15, 31, 51

Lexer
: part of parsing C++ source files, splits input into tokens. 20, 59

LibTooling
: C++ library which is part of the LLVM project, and which can be used to write standalone tools based on Clang 19, 20, 27, 29, 55

Node.js
: A server-side JavaScript runtime environment 4, 17, 18, 19, 20

OST

Eastern Switzerland University of Applied Sciences, focused on its campus Rapperswil-Jona in this document. ..... ii, 1, 89, 94

Phabricator

Phabricator is a set of tools for developing software. It includes applications for code review, repository hosting, bug tracking, project management, and more. [Phacility] 15, 30, 31, 32, 33, 34, 37, 40, 49, 60, 65, 67, 84, 91, 107, 111, 112

Source Lines of Code

lines of code without comments and blank lines 12

Undefined Behavior

Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment [...], to terminating a translation or execution [...]. [ISO] 22, 38, 97, 99, 100

Abbreviations

API: Advanced Programming Interface ..... passim
AST: Abstract Syntax Tree ..... passim
CDT: C/C++ Development Tooling 1
CLI: Command Line Interface 15, 19, 85
DSL: Domain Specific Language 30
ECTS: European Credit Transfer System 82
FURPS+: Functionality, Usability, Reliability, Performance, Supportability, Other 10
IDE: Integrated Development Environment ..... passim
IFS: Institute for Software ..... passim
ISO: International Organisation for Standardization 1, 2, 53, 57
LSP: Language Server Protocol ..... passim
NFR: Non-Functional Requirements 10, 66
OST: Eastern Switzerland University of Applied Sciences ..... passim
RPC: Remote Procedure Call 6
SDK: Software Development Kit 17, 18
STL: C++ Standard Template Library 98

Bibliography

[1] T. Corbat, "Assignment for Bachelor Thesis "C++ Stylechecker for VSCode"," Rapperswil, 2021.

[2] T. Corbat, "Weekly Thesis Status Meeting," 03 May 2021.

[3] Microsoft, "Language Server Protocol," [Online]. Available: https://microsoft.github.io/language-server-protocol/. [Accessed 02 June 2021].

[4] Microsoft, adjustments by M. Gartmann, "Language Server Extension Guide," 05 May 2021. [Online]. Available: https://code.visualstudio.com/api/language-extensions/language-server-extension-guide. [Accessed 02 June 2021].

[5] Dirk Bäumer, Erich Gamma, Sean McBreen, "Eclipse Newsletter 2017 about LSP," 2017. [Online]. Available: https://www.eclipse.org/community/eclipse_newsletter/2017/may/article1.php. [Accessed 03 June 2021].

[6] Microsoft, "Visual Studio Code," [Online]. Available: https://code.visualstudio.com/. [Accessed 03 March 2021].

[7] JSON-RPC Working Group, "JSON-RPC 2.0 Specification," 04 January 2013. [Online]. Available: https://www.jsonrpc.org/specification. [Accessed 02 June 2021].

[8] O. Zimmermann, *Application Architecture: Agile Landing Zones,* Rapperswil: OST Eastern Switzerland University of Applied Sciences, 2020.

[9] J. Nielsen, 01 January 1993. [Online]. Available: https://www.nngroup.com/articles/response-times-3-important-limits/. [Accessed 11 June 2021].

[10] AlDanial, "cloc GitHub Repository," [Online]. Available: https://github.com/AlDanial/cloc. [Accessed 15 June 2021].

[11] LLVM Community, "The LLVM Compiler Infrastructure," [Online]. Available: https://llvm.org/. [Accessed 15 March 2021].

[12] The Clang Team, "Clang: a C language family frontend for LLVM," [Online]. Available: https://clang.llvm.org/. [Accessed 15 March 2021].

[13] The Clang Team, "clangd," [Online]. Available: https://clangd.llvm.org/. [Accessed 15 March 2021].

[14] The Clang Team, "clang-tidy," [Online]. Available: https://clang.llvm.org/extra/clang-tidy/. [Accessed 15 March 2021].

[15] C. Robertson, "Using Clang-Tidy in Visual Studio," 19 February 2020. [Online]. Available: https://docs.microsoft.com/en-us/cpp/code-quality/clang-tidy. [Accessed 25 March 2021].

[16] JetBrains s.r.o., "Clang-Tidy integration," 08 March 2021. [Online]. Available: https://www.jetbrains.com/help/clion/clang-tidy-checks-support.html. [Accessed 25 March 2021].

[17] D. Malcolm, “Language Server Protocol: proof-of-concept GCC implementation,” 24 July 2017. [Online]. Available: https://gcc.gnu.org/legacy-ml/gcc-patches/2017-07/msg01465.html. [Accessed 17 March 2021].

[18] MaskRay, “ccls GitHub Repository,” [Online]. Available: https://github.com/MaskRay/ccls. [Accessed 17 March 2021].

[19] jacobdefault, “cquery GitHub Repository,” 29 July 2020. [Online]. Available: https://github.com/jacobdufault/cquery#archived. [Accessed 17 March 2021].

[20] LLVM Community, “Phabricator,” [Online]. Available: https://reviews.llvm.org/. [Accessed 11 June 2021].

[21] The Clang Team, “C++ Support in Clang,” [Online]. Available: https://clang.llvm.org/cxx_status.html. [Accessed 14 June 2021].

[22] The Clang Team, “Clang Expressive Diagnostics,” [Online]. Available: https://clang.llvm.org/diagnostics.html. [Accessed 14 June 2021].

[23] The Clang Team, “Clang - Features and Goals,” [Online]. Available: https://clang.llvm.org/features.html. [Accessed 14 June 2021].

[24] The Clang Team, “clang::PPCallbacks Class Reference,” [Online]. Available: https://clang.llvm.org/doxygen/classclang_1_1PPCallbacks.html. [Accessed 12 06 2021].

[25] Node.js, “Node.js,” [Online]. Available: https://nodejs.org/en/about/. [Accessed 25 March 2021].

[26] kuafuwang, “LspCpp GitHub Repository,” [Online]. Available: https://github.com/kuafuwang/LspCpp. [Accessed 15 06 2021].

[27] Microsoft, “vscode-languageserver GitHub Repository,” [Online]. Available: https://github.com/microsoft/vscode-languageserver-node. [Accessed 15 06 2021].

[28] OmniSparp, “OmniSharp LSP GitHub Repository,” [Online]. Available: https://github.com/OmniSharp/csharp-language-server-protocol. [Accessed 15 06 2021].

[29] OpenJS Foundation, “Node.js Documentation,” [Online]. Available: https://nodejs.org/api/documentation.html. [Accessed 15 06 2021].

[30] The Clang Team, “LibTooling,” [Online]. Available: https://clang.llvm.org/docs/LibTooling.html. [Accessed 15 06 2021].

[31] The Clang Team, “Matching the Clang AST,” 2021. [Online]. Available: https://clang.llvm.org/docs/LibASTMatchers.html. [Accessed 27 05 2021].

[32] ANTLR, “ANTLR,” [Online]. Available: https://www.antlr.org/. [Accessed 17 03 2021].

[33] ANTLR, “ANTLR grammars-v4 GitHub Repository,” [Online]. Available: https://github.com/antlr/grammars-v4. [Accessed 17 03 2021].

[34] ANTLR, “ANTLR antlr4 GitHub Repository,” [Online]. Available: https://github.com/antlr/antlr4. [Accessed 05 06 2021].

[35] isocpp, "CppCoreGuidelines GitHub Repository," 13 May 2021. [Online]. Available: https://github.com/isocpp/CppCoreGuidelines. [Accessed 20 May 2021].

[36] LLVM Extension, "clangd Visual Studio Code Extension," 02 March 2021. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=llvm-vs-code-extensions.vscode-clangd. [Accessed 04 July 2021].

[37] D. Jasper, "clang-tidy: lint-like checks and beyond," in *Euro LLVM 2014*, 2014.

[38] The Clang Team, "Clang-Tidy Checks," [Online]. Available: https://clang.llvm.org/extra/clang-tidy/checks/list.html. [Accessed 20 May 2021].

[39] P. Sommerlad and T. Corbat, *Modern and Lucid C++ for Professional Programmers, Week 2 - Functions, Values and Streams,* Rapperswil, 2019, p. 28.

[40] B. Stroustrup and H. Sutter, "C++ Core Guidelines," ISO, 11 March 2021. [Online]. Available: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines. [Accessed 22 April 2021].

[41] The Clang Team, "ASTMatchersMacros.h File Reference," 2021. [Online]. Available: https://clang.llvm.org/doxygen/ASTMatchersMacros_8h.html. [Accessed 27 05 2021].

[42] The Clang Team, "How to write a RecursiveASTVisitor based ASTFrontendActions," 2021. [Online]. Available: https://clang.llvm.org/docs/RAVFrontendAction.html. [Accessed 27 05 2021].

[43] The Clang Team, "clang::tooling::RecursiveSymbolVisitor< T > Class Template Reference," 27 April 2021. [Online]. Available: https://clang.llvm.org/doxygen/classclang_1_1tooling_1_1RecursiveSymbolVisitor.html. [Accessed 27 April 2021].

[44] ISO/IEC, "Unions," in *ISO/IEC 14882:2020: Programming Languages — C++*, ISO, 2020, p. [class.union.general]/4.

[45] ISO/IEC, "Classes," in *ISO/IEC 14882:2020: Programming Languages — C++*, ISO, 2020, p. [class.dtor]/12.

[46] J. B. David Svoboda, "PRE10-C. Wrap multistatement macros in a do-while loop," 23 April 2021. [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/c/PRE10-C.+Wrap+multistatement+macros+in+a+do-while+loop. [Accessed 27 April 2021].

[47] A clang-tidy Contributor, "LLVM Github Repository," [Online]. Available: https://github.com/llvm/llvm-project/blob/main/clang-tools-extra/clang-tidy/bugprone/MultipleStatementMacroCheck.cpp#L21. [Accessed 27 April 2021].

[48] ISO/IEC, "Classes," in *ISO/IEC 14882:2020: Programming Languages — C++*, ISO, 2020, p. [class.conv.ctor].

[49] ISO/IEC, "Initialization," in *ISO/IEC 14882:2020: Programming Languages — C++*, ISO, 2020, p. [class.base.init]/10.

[50] The Clang Team, “clang-tidy - modernize-use-default-member-init,” [Online]. Available: https://clang.llvm.org/extra/clang-tidy/checks/modernize-use-default-member-init.html. [Accessed 21 06 2021].

[51] cppreference.com Community, “Non-static data members,” [Online]. Available: https://en.cppreference.com/w/cpp/language/data_members. [Accessed 14 06 2021].

[52] The Clang Team, “clang-tidy - cppcoreguidelines-prefer-member-initializer,” [Online]. Available: https://clang.llvm.org/extra/clang-tidy/checks/cppcoreguidelines-prefer-member-initializer.html. [Accessed 14 06 2021].

[53] The Clang Team, “lang-tidy - modernize-use-equals-default,” [Online]. Available: https://clang.llvm.org/extra/clang-tidy/checks/modernize-use-equals-default.html. [Accessed 14 06 2021].

[54] T. Corbat, “Weekly Thesis Status Meeting,” 31 05 2021.

[55] The Clang Team, “clang::tidy::utils::IncludeInserter Class Reference,” [Online]. Available: https://clang.llvm.org/extra/doxygen/classclang_1_1tidy_1_1utils_1_1IncludeInserter.html. [Accessed 13 06 2021].

[56] LLVM Community, “Visual Studio Code Extension for Clangd GitHub Repository,” [Online]. Available: https://github.com/clangd/vscode-clangd. [Accessed 13 06 2021].

[57] LLVM Community, “llvm::sys::fs::recursive_directory_iterator Class Reference,” [Online]. Available: https://llvm.org/doxygen/classllvm_1_1sys_1_1fs_1_1recursive__directory__iterator.html. [Accessed 13 06 2021].

[58] LLVM Community, “llvm-project GitHub Repository,” [Online]. Available: https://github.com/llvm/llvm-project. [Accessed 28 April 2021].

[59] H. Hinnant, “Everything You Ever Wanted To Know About Move Semantics,” 12 April 2014. [Online]. Available: https://accu.org/conf-docs/PDFs_2014/Howard_Hinnant_Accu_2014.pdf. [Accessed 06 May 2021].

[60] F. Thurnheer, “clangd-www GitHub Repository Pull Request 24,” 12 03 2021. [Online]. Available: https://github.com/llvm/clangd-www/pull/24. [Accessed 17 06 2021].

[61] F. Thurnheer, “clangd-www GitHub Repository Pull Request 37,” [Online]. Available: https://github.com/llvm/clangd-www/pull/37. [Accessed 17 06 2021].

[62] F. Thurnheer, “clangd GitHub Repository Bug Report 799,” [Online]. Available: https://github.com/clangd/clangd/issues/799. [Accessed 16 06 2021].

[63] M. Gartmann, “[clang] Fix Typo in AST Matcher Reference,” 20 May 2021. [Online]. Available: https://reviews.llvm.org/D102836. [Accessed 20 May 2021].

[64] M. Gartmann, “CppCoreGuidelines GitHub Repository Pull Request 1789,” [Online]. Available: https://github.com/isocpp/CppCoreGuidelines/pull/1789. [Accessed 17 June 2021].

[65] The Clang Team, "Clang-tidy IDE/Editor Integrations," [Online]. Available: https://clang.llvm.org/extra/clang-tidy/Integrations.html. [Accessed 13 June 2021].

[66] Schweizerische Eidgenossenschaft, "Verordnung des Hochschulrates über die Koordination der Lehre an den Schweizer Hochschulen," 01 January 2020. [Online]. Available: https://www.fedlex.admin.ch/eli/cc/2019/722/de#art_3. [Accessed 03 March 2021].

[67] T. Corbat, "Weekly Thesis Status Meeting," 25 May 2021.

[68] T. Rix, "Phabricator - [clang-tidy] bugprone-header-guard : a simple version of llvm-header-guard," 3 May 2019. [Online]. Available: https://reviews.llvm.org/D61508. [Accessed 24 May 2021].

[69] Cevelop, "Cevelop GitHub Repository," [Online]. Available: https://github.com/Cevelop/cevelop. [Accessed 24 May 2021].

[70] SonarSource, "SonarQube," [Online]. Available: https://www.sonarqube.org/. [Accessed 03 March 2021].

[71] GitLab, "GitLab," [Online]. Available: https://about.gitlab.com/. [Accessed 03 March 2021].

[72] Docker Inc., "Docker," [Online]. Available: https://www.docker.com/. [Accessed 05 March 2021].

[73] Google Inc., "Clang," [Online]. Available: https://chromium.googlesource.com/chromium/src/+/master/docs/clang.md. [Accessed 17 March 2021].

[74] Alibaba Tech, "GCC vs. Clang/LLVM: An In-Depth Comparison of C/C++ Compilers," 29 August 2019. [Online]. Available: https://www.alibabacloud.com/blog/gcc-vs--clangllvm-an-in-depth-comparison-of-cc%2B%2B-compilers_595309. [Accessed 17 March 2021].

[75] Wikipedia, "Use case," 9 May 2021. [Online]. Available: https://en.wikipedia.org/wiki/Use_case. [Accessed 19 May 2021].

[76] kriskbx, "gitlab-time-tracker GitHub Repository," [Online]. Available: https://github.com/kriskbx/gitlab-time-tracker. [Accessed 03 March 2021].

[77] Microsoft, "vscode-cpptools GitHub Repository," [Online]. Available: https://github.com/microsoft/vscode-cpptools. [Accessed 11 June 2021].

A. Declaration of Authorship

I hereby declare,

- that I have written this thesis without any help from others besides those explicitly mentioned in the assignment or agreed upon in writing with the supervisor,
- that I have mentioned all the sources used and that I have cited them correctly according to established academic citation rules,
- that I have not used any materials (e.g., pictures) protected by copyrights in an unauthorised way.

Student's signature:	. . . . .
Name:	Fabian Thurnheer
Location, date of submission:	Jona, 18.06.2021

Student's signature:	. . . . .
Name:	Marco Gartmann
Location, date of submission:	Jona, 18.06.2021

B. Task Assignment

Assignment for Bachelor Thesis “C++ Style Checker for VSCode” Marco Gartmann and Fabian Thurnheer

1. Supervisor and Expert

This bachelor Thesis will be developed for the Institute for Software at HSR internally. It will be supervised by Thomas Corbat (thomas.corbat@ost.ch), OST. An expert independent of OST will examine the thesis and will be present at the final presentation:

- Guido Zgraggen, Google (zgraggen@gmail.com)

2. Students

This project is conducted in the context of the module “Bachelor-Arbeit” in the department “Informatik” by

- Marco Gartmann (marco.gartmann@ost.ch)
- Fabian Thurnheer (fabian.thurnheer@ost.ch)

3. Introduction

Developing C++ applications usually relies on a extensive toolchain including an integrated development environment (IDE), a compiler and continuous integration infrastructure for building, testing and deploying the software. As high quality source code reaches far beyond a program that compiles successfully, further analysis of the source code at hand can be performed while being processed in the stages up to the deployment. Reporting the results of such an analysis to the developer should be as immediate as possible. Preferably as soon as the code is entered into the code editor. Reducing the feedback loop can increase efficiency significantly, as the whole compilation process of C++ programs is typically time-consuming.

The C++ IDE Cevelop features several plug-ins to perform style related checks that ensure some best-practices [1]. An example of such a check is verifying that a single argument constructor is declared `explicit` to avoid implicit type conversions. While the Cevelop Stylechecker plug-in currently is an exclusive feature, it would be desirable to provide such features for other C++ IDEs as well.

4. Goals of the Project

The goal of this term project is to create a proof-of-concept implementation of the Stylechecker functionality independent of the Cevelop IDE. It would desirable to have their features integrated into other source code editors and IDEs as well. To achieve this, segregation of the source file analysis and the visualization of its results is necessary.

Microsoft defined the Language Server Protocol (LSP) in order to provide various quality-of-life coding features like auto-completion and have a proper separation of the analyzing server and the integration into the code editor. The LSP is language-agnostic and extensible for custom features. The solution is required to facilitate the LSP for separating the IDE and the language server.

Target source code editor of this bachelor thesis is VisualStudio Code as it already provides a stable support infrastructure for the LSP [3]. The integration into other IDEs should not be an insurmountable obstacle afterwards. The backend server implementation is unspecified at this point. While it is likely that Clang-based tooling will be adapted for the task, the features can be implemented in a more accessible environment at the beginning.

It is infeasible that all features of Cevelop's code analysis plug-in are supported at the end of this thesis. Complete support would requires in dept analysis of the C++ source code featuring an abstract syntax tree representation and a complete symbol table for the target projects. The goal of this project is the elaboration of a feasible infrastructure to implement such functionality in the future. To be able to test and use the results of this project it would be desirable if the proof-of-concept implements source code checks that are useful for the students learning C++ in the context of OST's programming modules.

Below we have a list of possible checks that might be implemented, which are currently implemented in Cevelop's Stylechecker plug-in:

- Use of std::cin/std::cout outside the main() function
- Name format violating a preconfigured naming style guide
- Global variables
- Missing include guard in header files
- Include of <iostream> in a file that does not contain the main() function
- Initialization of member variables a constructor's body instead of the member-initializer-list
- Missing include of the own header in a source file
- Wrong include orders of own header and system headers
- Missing or unnecessary includes of standard headers
- Single argument constructors that are not marked as explicit
- Using directives in headers on namespace level

Alternatively, C++ Core Guideline checks (which are partly implemented in Cevelop's GSLator plug-in), might be a set of rules of rules that targets a broader audience [4]. It is also possible to define new checking features for the scope of this project if reasonable. Eventually, the checks to be implemented have to be discussed with and are selected in collaboration with the supervisor.

5. Documentation

This project must be documented according to the guide lines of the "Informatik" department [5]. This includes all analysis, design, implementation, project management, etc. sections. All documentation is expected to be written in English. The project plan also contains the documentation tasks. All results must be complete in the final upload to the archive server [5]. Two copies of the documentation must be handed-in:

- One in color, two-sided
- One in B/W, single-sided

6. Important Dates

22.02.2021	**Start of the bachelor thesis.**
09.06.2021	**Hand-in of the abstract to the supervisor for checking. Information about accessing the corresponding web tool will be given by the department office.** **Hand-in (by email) of the A0 poster to the supervisor.**
18.06.21 – 12:00 o'clock	**Final hand-in of the report**
TBD	**Presentation and Oral Exam**

7. Evaluation

A successful term project counts as 12 ECTS point. The estimated effort for 1 ECTS is 30 hours. (See also the module description [6]). The supervisor will be in charge for all the evaluation of the project.

Criterion	Gewicht
1. Organisation, Execution	1/6
2. Report (Abstract, Management Summary, technical and personal reports) as well as structure, visualization and language of the whole documentation	1/6
3. Content	3/6
4. Final Presentation of the results and discussion	1/6

Furthermore, the general regulations for term projects of the department “Informatik” apply.

8. References

[1] https://www.cevelop.com

[2] https://microsoft.github.io/language-server-protocol

[3] https://code.visualstudio.com/

[4] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

[5] https://teams.microsoft.com/_#/school/files/Studieninformationen?threadId=19%3A88ac24bbee8e4682a73e81839cd2103c%40thread.tacv2&ctx=channel&context=Studien-%2520und%2520Bachelorarbeiten&rootfolder=%252Fteams%252FTS-StudiengangInformatik%252FFreigegebene%2520Dokumente%252FStudieninformationen%252FStudien-%2520und%2520Bachelorarbeiten

[6] https://archiv-i.hsr.ch

C. Project Management

In general, this chapter aims to explain the used project management principles. In the first sections (up to Section C.3), the team members' responsibilities as well as the chosen project method and the required tools are outlined. Sections C.4 and C.5 describe the planned effort, the duration of the project and how the project is structured with phases and milestones. The following sections contain information on planned meetings (C.6), measures to ensure code and documentation quality (C.7) and lastly, a list of evaluated project risks (C.8).

C.1. Responsibilities

Both members of the development team are considered as full stack developers and are therefore equally responsible for all development tasks.

Member	**Responsibilities**
Fabian Thurnheer	Development, DevOps, Documentation
Marco Gartmann	Development, DevOps, Documentation

Table 10: Project Development Team Members and Their Responsibilities

C.2. Project Method

For the realization of this project, it was decided to use a variation of the Unified Process where the project duration will be divided in four phases. Namely, those are: Inception, Elaboration, Construction and Transition. The Elaboration and Construction phase will be conducted in an agile manner modelled on the Scrum framework. During the agile phases of the project, sprints of the duration of one week will be used.

This method was chosen because the project team already successfully conducted their term project with it. Furthermore, it includes advantages both of the agile and the non-agile world, which allows to structure the project with milestones and still allows to include new requirements during the duration of the project.

C.3. Tooling

To manage created source code, the version control system git is used. Furthermore, the git repository is uploaded to GitHub. Initially, GitLab was planned to be used. However, during the project it became clear that the repository was exceeded OST's GitLab upload limit. Therefore, GitHub was used from then on.

For the project management, a GitLab instance provided by the OST university is used. The needed components for the chosen project method (as described above) are realized as follows:

Work packages and milestones: work packages are created as GitLab issues and are assigned to the created milestones.

Scrum board: is mapped to a GitLab issue board. Issues are positioned on the issue board using labels.

Time tracking: worked hours are recorded on the GitLab issues. For the creation of time reports, a self-written webapp fetching GitLab's API is used.

C.4. Planned Effort

The bachelor thesis counts for 12 ECTS credits. By means of the Bologna system, each ECTS credit equals an effort of 30 hours [66]. With two project members, the total planned effort for this thesis equals 720 hours.

This thesis must be completed within the spring semester 2021. The lecture period of 15 weeks is extended by two weeks, which can be used exclusively for the bachelor thesis. With 720 hours and 17 weeks in total, the planned effort is 21 hours per week and project member.

C.5. Phases and Milestones

As mentioned before, the duration of this thesis project is divided into four phases: Inception, Elaboration, Construction and Transition. This section gives a brief overview of each phase's goal and of the planned milestones.

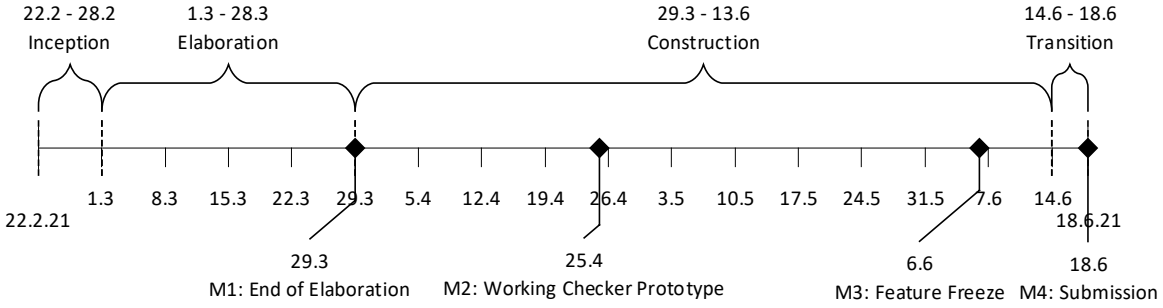


Figure 13: Overview of Phases and Milestones

C.5.1 Phases

Inception: In this phase, a first meeting with the supervisor will be held and the task definition as well as the thesis's goal will be discussed. Additionally, the development team will be starting their work on the project plan and discussing any administrative matters.

Elaboration: The aim of the Elaboration phase is to finish the project plan, which may still be altered in a later phase, to evaluate and minimize any risks, to define the requirements and to develop prototypes for the language client and the language server, if demanded by the chosen solution strategy. The latter will already be developed using an agile approach. If the language server and client do not have to be self-implemented, work on firsts checks can be begun.

Construction: During this phase, while processing the evaluated work packages in order of their importance, the majority of code will be written. Simultaneously, the produced code and the resulting product will be tested continuously. A feature freeze concludes this phase. Afterwards, implementation of new features is forbidden.

Transition: Serves the purpose to finalize work in progress and to prepare the thesis documentation for its submission.

C.5.2 Milestones

For the duration of the thesis project, milestones were defined to ensure that the project is on course. The defined milestones can be seen in Table 11.

Milestone	Date	Description
M1: End of Elaboration	28.03.2021	All risks have been identified and measures to minimize them have been defined. First requirements have been evaluated and the developed prototype proofs the validity of the chosen approach.
M2: Working Checker Prototype	25.04.2021	A first check has been implemented in the language server. This can be a simple check, e.g., for missing include guards.
M3: Feature Freeze	06.06.2021	The final function scope is fixed. Afterwards, no new work will be begun.
M4: Submission	18.06.2021	The project is finished. All MUST requirements are implemented. The software and the thesis documentation are submitted.

Table 11: Evaluated Milestones with Their Corresponding Description

C.6. Meetings

Besides any informal ad-hoc meetings between the development team members, some meetings are planned to be held regularly, as it can be seen in Table 12.

Meeting	Frequency	Purpose
Status meeting	Weekly on Mondays	Together with the thesis's supervisor, the weekly progress and any occurred problems are discussed.
Sprint meeting	After each sprint, weekly	Includes the retrospective of the finished sprint and the planning of the upcoming sprint. This contains the backlog management and thus the prioritization, weighting, and creation of issues.

Table 12: Planned Meetings

C.7. Quality Measures

Several measures were defined to ensure a good quality, both for implemented code and written documentation. This section describes these measures.

C.7.1 Use of Branches

Goal: To keep changes to the code base small and thus to reduce the occurrence and complexity of merge conflicts. Additionally, it aims to allow for code review processes on small chunks of new code.

For each issue/feature, a new feature branch is created where the necessary work is done. Code in the main branch is considered finished, working, and compliant to the coding guidelines. A feature branch is only merged into the develop branch after it passed the review process.

C.7.2 Definition of Done

Goal: To create a common understanding on when something can be categorized as done and prevent unfinished work from being merged.

For an issue/feature/user story to be mergeable, the following criteria must be met:

- No errors are shown in the IDE.
- Pipeline has been run successfully, including:
    - Built successfully, all unit tests passed, no linter errors shown.
- Tests for the newly created code have been added.
- If necessary, the software documentation has been updated.
- Newly added code has passed through the code review process.

C.7.3 Code Reviews

Goal: To ensure a high code quality and a complete and up-to-date documentation of the software documentation.

After an issue has been finished by a team member and before it can be published, the other team member conducts a review of it. The emphasis is thereby put on the following points:

- The reviewer ensures that the new feature is working as intended or that the addressed problem has been fixed.
- The reviewer verifies that there are no code smells in the code not yet identified by static code analysis and that variables and classes have meaningful names.
- The reviewer checks that the design and architecture documentation has been updated if necessary.

As it was decided to follow the approach where clang-tidy (see Section 4.1) was to be extended, the goal of each feature is to be merged into the extended project. Before they are merged, a differential showing what was changed in the project must be uploaded to a review tool called Phabricator. There, members of clang-tidy's community also conduct a review of the implemented functionality and give feedback.

C.7.4 Testing

Goal: To minimize the possibility of faults and bugs.

To achieve the mentioned goal, tests are run continuously and also before each feature is published to Phabricator.

C.7.5 Proof Reading

Goal: To eradicate spelling errors in the thesis and to ensure that it is written understandably.

On the one hand, proof reading is done continuously throughout the project by the thesis team members who review each other's texts. Additionally, it is planned that a proficient person proofreads this thesis at the end of the thesis duration.

C.8. Risk Management

This section lists the risk analysis including the risks themselves, which are defined at the begin of this project. The evaluated risks are listed in Table 13 and their severity classification can be seen in Table 14. Section C.8.1 shows how the risks were re-evaluated at the end of the Elaboration phase.

No.	Title	Description	Max. damage	Occurrence probability	Prevention	Behaviour upon occurrence
R1	GitLab failure	The GitLab platform of OST is temporary not available. Neither the Website nor the communication over the command line interface (CLI) work.	Slight delay	Low	GitLab works with git, so every developer has its own local copy of the source code. Due to that, no prevention is needed.	No special behaviour needed.
R2	Local data loss	A team member accidentally deletes his own local project data, or the hard drive of his computer is defect.	Medium delay	Very low	Every project member pushes his code on a regular basis to the GitLab instance. Documentation files are automatically synchronized to the used OneDrive store.	Rewriting of lost documentation parts or source code.
R3	Defect dependencies	Used third party libraries have several malfunctions (e.g.: Language Server Implementations or Parser).	Large delay	Low	Usage of dependencies which are broadly used, and which are well maintained.	Find a new library/ dependency as quickly as possible which can replace the current one.
R4	Implementation difficulties	The project team does not have the required know-how to fulfil the needed tasks.	Project has failed	High	Plan enough time for self-studying the core topics of the ongoing task.	After unsuccessfully trying to solve the problem ourselves, contact the supervisor for further help.
R5	Underestimated the effort	The project team cannot deliver all defined tasks in the given project time.	Slight delay	Low	Keep core functionalities small, present the project state to the supervisor in the weekly meetings.	Contact the supervisor and discuss the further approach (e.g., shrink project scope).

No.	Title	Description	Max. damage	Occurrence probability	Prevention	Behaviour upon occurrence
R6	Incorrect communication	Due to incorrect communication, project tasks are done incorrectly or not at all.	Medium delay	Low	Frequent meetings between the team members to identify misunderstandings as soon as possible.	If there occur multiple communication problems, plan meetings in fixed intervals and hold them more often.
R7	Editor dependency	The API of Visual Studio Code changes during the project time. Code must be rewritten, or the project's requirements could no longer be fulfilled.	Medium delay	Very low	The occurrence of this incident is unlikely. Due to predefined usage of LSP, it should be possible to change the editor without rewriting the core functionality of the product (e.g., the use of a language server).	Discuss the further procedure with the supervisor.
R8	Absence of a team member	A team member is absent for a longer period (e.g., disease).	Medium delay	Medium	Update the documentation frequently and hold meetings between the project members regularly.	Discuss the further procedure with the supervisor.

Table 13: Evaluated Risks at the Beginning of the Project

Impact / ***Probability***	**Slight delay**	**Medium delay**	**Large delay**	**Project has failed**
Very low		R2, R7		
Low	R1, R5	R6	R3	
Medium		R8		
High				R4

Table 14: Risk Matrix at the Beginning of the Project

C.8.1 Reassessment at the End of Elaboration

To ensure that at the end of the Elaboration phase all risks have been identified and critical risks have been eliminated, the specified risks were revaluated.

No.	Title	Reassessment	Justification
R1	GitLab failure	No	After the evaluation of LLVM, GitLab is no longer this project's code repository and is only used for administrative purposes. Because the impact classification is already at the lowest, no reassessment is necessary.
R2	Local data loss	No	Although working with GitLab changed to using GitHub as code repository, this risk is still present and unchanged.
R3	Defect dependencies	Yes	The thesis team decided to use LLVM as the infrastructure to use, which is a well-renowned project. During the implementation of the first check, no malfunction of any important component occurred. The probability that a severe defect in LLVM would emerge was assessed as *very low*.
R4	Implementation difficulties	Yes	The elaborated infrastructure was tested and a first check for LLVM's clang-tidy was successfully implemented. Therefore, the project was no longer seen as being at risk. However, implementing checks for such a large project is still complex. Therefore, the risk's impact was downgraded to *medium delay*.
R5	Underestimated the effort	No	Because a first version of a check could be successfully developed, and further checks are incrementally evaluated, it was decided not to change this risk's already low classification.
R6	Incorrect communication	Yes	As expected, the project team is well-established and communicates daily with each other during the working week. No communication problem occurred until this reassessment, so the probability was set to *very low*.
R7	Editor dependency	No	No reassessment is needed. This risk's probability is still *very low* and the usage of LSP is set, which mitigates this risk.
R8	Absence of a team member	Yes	Although the COIVD-19 pandemic is still present, due to the measures of the federal government and constantly working on the thesis from home, this risk's probability was be downgraded to *low*.

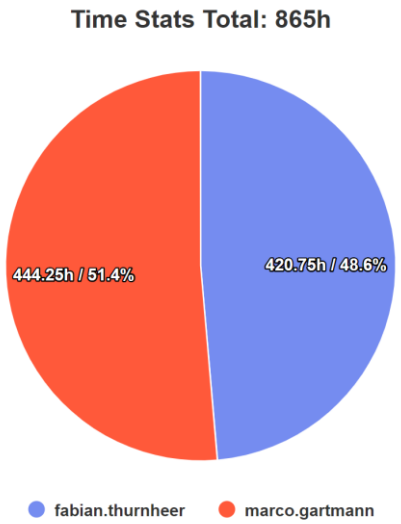
Table 15: Reassessment of the Defined Risks at the End of Elaboration

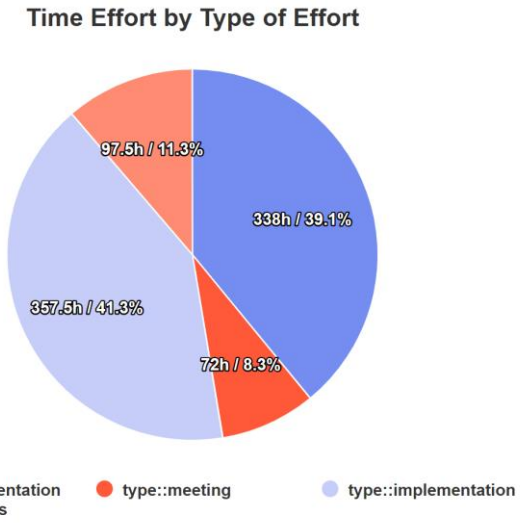
Impact / *Probability*	Slight delay	Medium delay	Large delay	Project has failed
Very low		R2, R6, R7	R3	
Low	R1, R5	R8		
Medium		R4		
High				

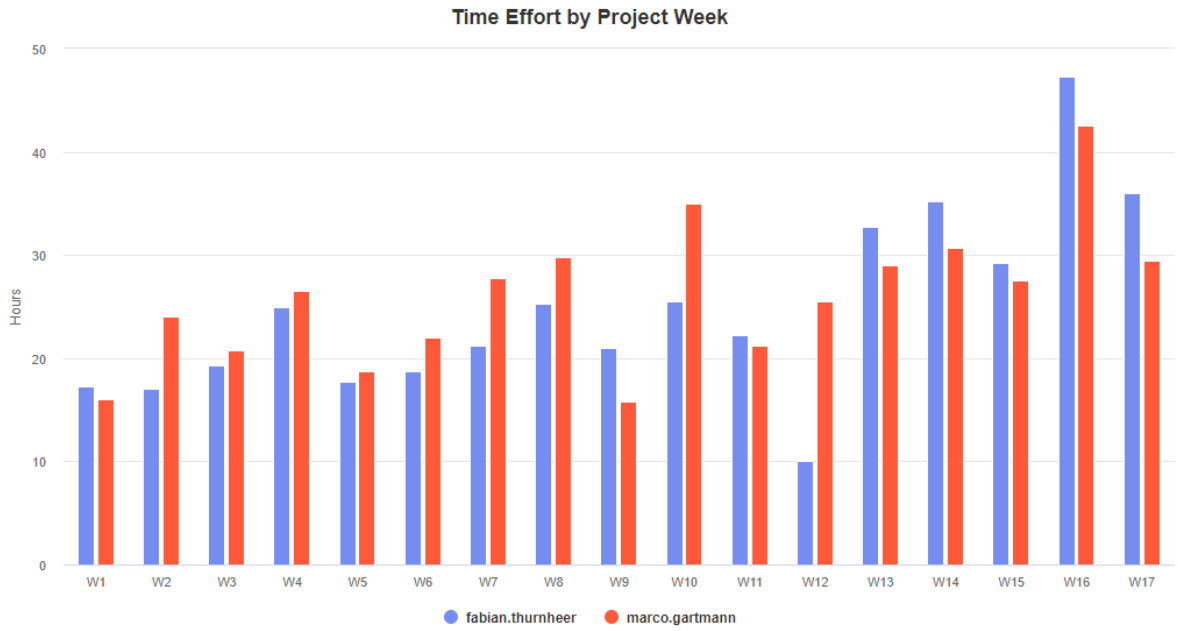
Table 16: Risk Matrix at the End of Elaboration

Conclusion: Because no risk was evaluated to be project-critical, the project's success is not at risk anymore.

C.9. Time Statistics







D. Comparison of Clang-Tidy and Cevelop Checks

As part of elaborating different design approaches for this thesis, a comparison of existing clang-tidy checks and existing Cevelop checks was conducted. The existing Cevelop checks originate from the two Cevelop features "C++ Ctylechecker" and "GSLator". The results from these comparisons are visible in Table 17 to Table 18.

C++ Core Guideline	**C++ Style Checker**	**Corresponding Clang-Tidy Check**	**Fix available?**	**Categorisation Comments**
CC*	Cin Cout Problem	-	-	• Feasibility: provided AST matchers can be used, not much extra logic needed → simple • Likeliness: this rule is probably new to most OST students visiting the C++ module of OST, iostreams are regularly used for testates and exercises → high • Error impact: only influences testability of a program → low
-	Dynamic Style Problem (naming conventions)	readability-identifier-naming	Yes	(already implemented)

* This check does not cover a C++ Core Guideline, so the project team defined its own alias for identification purposes.

C++ Core Guideline	**C++ Style Checker**	**Corresponding Clang-Tidy Check**	**Fix available?**	**Categorisation Comments**
-	Dynamic Style Problem for Files	-	-	Creation of the Dynamic Style Problems checks in Cevelop was covered by a whole bachelor thesis on its own. [67] It would not be feasible to implement this as a part of this thesis [67]. Therefore, this check is not considered for implementation as part of this thesis.
I.2	Global Non Const Variable Problem	cppcoreguidelines-avoid-non-const-global-variables	No	Since no fix was provided, it was assessed as well: • Feasibility: variable type needs to be extracted and cleanly printed, high effort is estimated → moderate • Likeliness: const can easily be forgotten by novice programmers, who might not be aware of the impact this causes → high • Error impact: error prone since non-const global variables are subject to unpredictable changes → medium

C++ Core Guideline	**C++ Style Checker**	**Corresponding Clang-Tidy Check**	**Fix available?**	**Categorisation Comments**
SF.8	Include Guard Missing	-	-	• Feasibility: clang-tidy utility Utils/HeaderGuard exists which does most of the work. However, it would have to be analysed how it can be utilized. A new dependency would be introduced to an evaluated clang-tidy module → hard • Likeliness: concept is new to novice C++ programmers. Although taught in OST's C++ module, it might be forgotten → high • Error impact: error prone, program might not compile if One Definition Rule is violated because of missing include guards → medium Worth noting: a similar check was already pushed to Phabricator but has not been merged yet [68].
II*	Iostream Include Problem	-	-	Is strongly related to the "Cin Cout Problem", the same categorisation thoughts apply as above.
C.49	Member Initializer Not Used in Constructor	cppcoreguidelines-prefer-member-initializer	Yes	(already implemented)

* This check does not cover a C++ Core Guideline, so the project team defined its own alias for identification purposes.

C++ Core Guideline	**C++ Style Checker**	**Corresponding Clang-Tidy Check**	**Fix available?**	**Categorisation Comments**
SF.5	Missing Include to Own Header	-	-	• Feasibility: pre-processor call-backs would be needed; existing checks could be helpful for implementing this. However, no known AST matchers can be used → moderate • Likeliness: is an important concept taught in OST's C++ module, students are regularly confronted with this issue → high • Error impact: error prone; possible inconsistencies between function return types in header files and its implementations can only be noticed during link-time if own header file is not included, and not earlier, e.g., in an IDE or during compile-time → medium

C++ Core Guideline	**C++ Style Checker**	**Corresponding Clang-Tidy Check**	**Fix available?**	**Categorisation Comments**
MSI*	Missing Standard Include	-	-	• Feasibility: based on the existing Cevelop check, it would need a great effort to implement this [69](cevelop/CevelopProject/bundles/com.cevelop.ctylechecker/src/com/cevelop/ctylechecker/checker/includes/MissingStandardIncludeChecker.java) → hard • Likeliness: it is seen as likely that novice C++ programmers might forget includes and do not know what includes are needed → high • Error impact: error prone, the compilation will fail because no declaration is provided → medium

* This check does not cover a C++ Core Guideline, so the project team defined its own alias for identification purposes.

C++ Core Guideline	**C++ Style Checker**	**Corresponding Clang-Tidy Check**	**Fix available?**	**Categorisation Comments**
MACT*	Multiple Asserts in Cute Test	-	-	• Feasibility: Existing AST matchers could use to implement this check. Also, a list of possible assert function names is needed → simple • Likeliness: Writing CUTE tests during OST's C++ module is common practice. So, a student will likely forget this OST own rule → high • Error impact: It improves readability of CUTE test code → low
C.46	Non-'explicit' Single Argument Constructor	google-explicit-constructor	Yes	(already implemented) Note: in Table 18, a similar Cevelop check is listed with no corresponding clang-tidy check. This is because the google-explicit-constructor check does not cover the corresponding guideline completely. Since this check is not part of Cevelop's GSLator plugin. Thus, it does not demand implementing rule C.46 exactly and the google-explicit-constructor is seen as sufficient.
-	Redundant Access Specifier Problem	readability-redundant-access-specifiers	Yes	(already implemented)

* This check does not cover a C++ Core Guideline, so the project team defined its own alias for identification purposes.

C++ Core Guideline	**C++ Style Checker**	**Corresponding Clang-Tidy Check**	**Fix available?**	**Categorisation Comments**
SIP*	Self-Include Position	-	-	• Feasibility: possible with pre-processor call-backs offered by clang, but very different to checks using AST matchers → moderate • Likeliness: if no attention is paid to this issue, it is likely to include the source file's header file in the wrong position → high • Error impact: worse readability of source files. Furthermore, if not placed in first position, it is harder to distinguish if the header file is self-contained. No errors if done wrong → low

* This check does not cover a C++ Core Guideline, so the project team defined its own alias for identification purposes.

C++ Core Guideline	**C++ Style Checker**	**Corresponding Clang-Tidy Check**	**Fix available?**	**Categorisation Comments**
SSI*	Superfluous Standard Include	-	-	• Feasibility: would require to extended analysis of all used names and to keep a list of names with their needed includes, AST matchers are only slightly helpful for this → hard • Likeliness: can happen to novice and more experienced programmers. However, the contrary (forgetting needed includes) is estimated as much more likely → medium • Error impact: introduces unnecessary includes, leading to larger translation units, does not lead to errors → low
-	System Include Before Own Include	llvm-include-order	Yes	(already implemented)
SF.7	Using in Header Problem	google-global-names-in-headers	No	(already implemented)

Table 17: Cevelop C++ Style Checkers and Their LLVM Counterparts

* This check does not cover a C++ Core Guideline, so the project team defined its own alias for identification purposes.

C++ Core Guideline	**GSLator Plugin**	**Corresponding Clang-Tidy Check**	**Fix available?**	**Categorisation Comments**
ES.9	AvoidALLCAPSnamesChecker	-	-	• Feasibility: own AST matchers needed for caps check, every statement needs to be checked, could be very noisy → moderate • Likeliness: macros are barely used in OST's C++ module → low • Error impact: no Undefined Behavior but hard to find bugs → high
ES.20	AlwaysInitializeAnObjectChecker	cppcoreguidelines-pro-type-member-init cppcoreguidelines-init-variables	Yes	(already implemented)
ES.26	DontUseVariableForTwoUnrelatedPurposesChecker	-	-	• Feasibility: dependencies between every variable declaration and declaration reference expression needs to be analysed if they are related, "unrelated purposes" is hard to interpret and to implement → hard • Likeliness: likely to run into for beginners → high • Error impact: leads to bad readability, could lead to security issues when using buffers multiple times → medium
ES.46	AvoidLossyArithmeticConversionsChecker	cppcoreguidelines-narrowing-conversions bugprone-narrowing-conversion (alias)	No	(already implemented)
ES.49	IfMustUseNamedCastChecker	cppcoreguidelines-pro-type-cstyle-cast google-readability-casting	Yes	(already implemented)
ES.50	DontCastAwayConstChecker	cppcoreguidelines-pro-type-const-cast	Yes	(already implemented)

C++ Core Guideline	**GSLator Plugin**	**Corresponding Clang-Tidy Check**	**Fix available?**	**Categorisation Comments**
ES.74	DeclareLoopVariableInTheInitializerChecker	-	-	• Feasibility: loop through all declRefExpr is needed to check if its variable is declared outside the loop, some effort needed → moderate • Likeliness: for-statements are often used by beginners instead of STL algorithms, likely to run into this problem. Use of for-statements is discouraged in OST's C++ module → high • Error impact: bad readability, prevents optimizations → low
ES.75	AvoidDoStatementsChecker	-	-	• Feasibility: AST matcher for do statements exists, not much effort → simple • Likeliness: is a basic language feature and possibly known to novice programmers → high • Error impact: according to C++ Core Guidelines, this leads to bugs, and bad readability → medium
ES.76	AvoidGotoChecker	cppcoreguidelines-avoid-goto hicpp-avoid-goto (alias)	No	(already implemented)
C.20	RedundantOperationsChecker	-	-	Together with the thesis's supervisor, it was assessed that this check would require a unproportionally high effort for only low benefit for students. • Feasibility: hard • Likeliness: low • Error impact: low

C++ Core Guideline	**GSLator Plugin**	**Corresponding Clang-Tidy Check**	**Fix available?**	**Categorisation Comments**
C.21	MissingSpecialMemberFunctionsChecker	cppcoreguidelines-special-member-functions hicpp-special-member-function (alias)	No	(already implemented)
C.31	NoDestructorChecker DestructorHasNoBodyChecker DestructorWithMissingDeleteStatement Checker	-	-	• Feasibility: simple enforcement would be doable but would likely require a complex AST matcher → moderate • Likeliness: since usage of *gsl::owner* and explicit memory management is not taught in OST's first C++ module, students of this module would not profit much → low • Error impact: memory leaks → medium
C.35	BaseClassDestructorChecker	-	-	• Feasibility: AST matcher is estimated as simple, however, implementing a fix could be complex → moderate • Likeliness: something that novice C++ might not be aware of. However, only affects variables which have a dynamic type (new, make_unique), which is not the majority of use-cases for novice C++ students → medium • Error impact: could lead to Undefined Behavior [40] → high

C++ Core Guideline	**GSLator Plugin**	**Corresponding Clang-Tidy Check**	**Fix available?**	**Categorisation Comments**
C.37	DestructorShouldBeNoExceptChecker	Not implemented, but worth mentioning: bugprone-exception-escape	-	• Feasibility: AST matcher could be realised as *cxxDestructorDecl(unless(isNoThrow()))*. Only matches if throwing class is used → simple • Likeliness: program would only not terminate if destructor thrown exceptions are "catched". Throwing destructors are probably rare in beginner's code. Furthermore, use of destructors is not taught in-depth in OST's C++ module [67] → low • Error impact: could lead to memory leaks and in the worst-case to Undefined Behavior → high Because of the high error impact, this rule's benefit for students would result in a medium rating. However, since in-depth destructors are not part of OST's C++ module, the target audience would not profit much and would probably be irritated by warnings of this check. Thus, it was decided to override this rating to low.

C++ Core Guideline	**GSLator Plugin**	**Corresponding Clang-Tidy Check**	**Fix available?**	**Categorisation Comments**
C.44	NoexceptDefaultCtorChecker	-	-	• Feasibility: AST matcher is expected to be simple (*isNoThrow()*)→ simple • Likeliness: this topic is not taught in OST's C++ module [67] and is thus not relevant for the target audience → low • Error impact: hardens error-handling and reasoning about move operations, but does not result in errors itself → low
C.45	InClassInitializeChecker	-	-	• Feasibility: should be possible with existing AST matchers → simple • Likeliness: for beginners, it may be reasonable to initialize member variable with default values in the initializer list. It probably is not obvious why this should not be done → high • Error impact: only has an impact on performance and readability → low
C.46	DeclareSingleCtorExplicitChecker	-	-	• Feasibility: enforcement does not demand much → simple • Likeliness: unintended conversions are not obvious and can happen easily if a programmer is not aware of this → high • Error impact: bug prone if done wrong, leads surprising conversions which are not easy to detect → medium

C++ Core Guideline	**GSLator Plugin**	**Corresponding Clang-Tidy Check**	**Fix available?**	**Categorisation Comments**
C.47	InitializeMemVarsInRightOrderChecker	clang(-Wreorder-ctor)	Yes	(already implemented in clang)
C.48	PreferInClassInitializerToCtorInitChecker	cppcoreguidelines-pro-type-member-init	Yes	(already implemented)
C.49	NoAssignmentsInCtorChecker	cppcoreguidelines-prefer-member-initializer	Yes	(already implemented)
C.60	CopyAssignmentNonVirtualChecker CopyAssignmentParameterByConstRefChecker CopyAssignmentReturnByNonConstRefChecker	cppcoreguidelines-c-copy-assignment-signature	No	(already implemented)
C.63	MoveAssignmentNonVirtualChecker MoveASsignmentReturnByNonConstRefChecker	cppcoreguidelines-c-copy-assignment-signature	No	(already implemented)
C.66	MoveOperationsShouldBeNoExceptChecker	performance-noexcept-move-constructor hicpp-noexcept-move (alias)	Yes	(already implemented)
C.83	ValueLikeTypesShouldHaveSwapChecker	-	-	• Feasibility: no direct AST matcher for swap functions available, new AST matcher would have to be created → moderate • Likeliness: again, this topic is not part of OST's C++ module [67]. The target audience (module visitors) would more likely be irritated by this check's warnings than they would profit from it [67] → low • Error impact: only affects performance and efficiency, no errors result from doing this wrong → low
C.84	MakeSwapNoExceptChecker	-	-	Covers the same topic as C.83, the same categorisation thoughts apply as above.

C++ Core Guideline	**GSLator Plugin**	**Corresponding Clang-Tidy Check**	**Fix available?**	**Categorisation Comments**
C.85	NamespaceLevelSwapFunctionChecker	-	-	Covers the same topic as C.83, the same categorisation thoughts apply as above.
C.164	AvoidConversionOperatorsChecker	google-explicit-constructor	Yes	(already implemented)

Table 18: Cevelop GSLator Plugin and Their LLVM Counterparts

E. System Tests

This chapter documents the conducted tests of the non-functional requirements defined in Section 3.2 and their results. These tests were conducted after the solution strategy of extending clang-tidy was selected. Since this thesis's work had little to no impact on the LLVM project's non-functional properties, it was decided that it would not make sense to conduct a second test on its NFRs at the project's end.

E.1. Test of Non-Functional Requirements

Responsible: Marco Gartmann

Execution time: 23.03.2021

Visual Studio Code Environment:

- Code: 1.56.2 (054a9295330880ed74ceaedda236253b4f39a335)
- OS: win32(10.0.19042)
- CPUs: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz(8 x 1992)
- Memory(System): 15.85 GB(8.16GB free)
- Memory(Process): 191.70 MB working set(166.75MB private, 0.90MB shared)

E.1.1 Usability

Scenario	Description	Test Execution	Expected Result	Effective Result
#1	Automatic extension activation	Opened Visual Studio Code, ran the VS Code "Developer: Show Running Extensions" command, verified that clangd is not active, opened a C++ file and verified that clangd is now listed in the activated extensions.	Clangd VS Code extension is started as soon as a user opens a C++ file.	Passed. Is started after a C++ file is opened. Is ensured with the extension's `"activationEvent": "onLanguage:cpp"` setting.

Table 19: Test of Usability NFRs

E.1.2 Performance

Scenario	Description	Test Execution	Expected Result	Effective Result
#1	Diagnosis does not affect IDE runtime behaviour	Opened Visual Studio Code, opened the following C++ file: llvm-project\clang-tools-extra\ clang-tidy\google\AvoidCStyleCastsCheck.cpp (190 lines of code). Started adding new code fragments by hand.	Time between user input and displaying of the entered character in the IDE: Minimal: 0.2s, delay noticeable Target: 0.1s, no delay	Passed (outstanding). No delay on user input was noticeable.
#2	Start-up time of IDE extension	Opened Visual Studio Code, opened a C++ file, ran the VS Code "Developer: Show Running Extensions" command, inspected clangd's displayed activation time. This was done ten times and the runs average was calculated.	Time between opening a C++ file and the moment the clangd extensions starts communicating with its language server: Minimal: < 2 seconds Target: < 1 second Outstanding: < 0.5 seconds	Passed (target). Average activation time was 713ms.
#3	Quickness of Style Checker Feedback	With all clang-tidy checks activated, Visual Studio Code was run, the following C++ file was opened (having the defined 22'000 SLOC with its headers included, see 3.2.2), which is part of clang-tidy itself: `C:\HSR\Semester6\BA\llvm-project\lld\unittests\MachOTests\MachONormalizedFileYAMLTests.cpp` This triggers several clang-tidy checks. In VS Code's Output window of clangd, the time between the "didOpen" LSP message to the language server and the first received diagnostic was assessed. To see these times, in the VS Code settings of the clangd extension, "-log=verbose" was added to the "Clangd: Arguments" setting.	Time between a user input that triggers a diagnostic and the moment this diagnostic is received in VS Code: Minimal: < 5 seconds Target: < 4 seconds Outstanding: < 2 second	Passed (target). Time of "didOpen" LSP message: 14:02:32.264 Time of first received diagnostic: 14:02:35.321 Delta: 3.06s It should be noted that in this time, Clang also loaded lld's complete compile commands database to resolve all includes and symbols. Analysis of C++ files like those created for OST's C++ course exercises usually take less than 100ms.

Table 20: Test of Performance NFRs

E.1.3 Supportability

Scenario	Description	Test Execution	Expected Result	Effective Result
#1	Installation of the language server and language client	On a computer which did not have the clangd extension installed, opened Visual Studio Code, installed the clangd extension from the VS Code marketplace and assessed that communication between client and server happens (visible in clangd output log in VS Code).	When a user installs the extension (language client), its corresponding language server does not need to be installed separately.	Passed. While installing the Visual Studio Code extension "clangd", the corresponding language server "clangd" is also installed.

Table 21: Test of Supportability NFRs

F. Developer Guide

This guide should help future developers working on clang-tidy checks or create new ones. This includes insight on how to setup a working environment with LLVM on Windows with Visual Studio. Furthermore, this chapter also contains a contribution workflow which describes how clang-tidy checks and fixes can be implemented and contributed to the LLVM project. Moreover, some pitfalls, which the thesis team encountered during creating the workflow, are documented as well.

F.1. LLVM General

F.1.1 Documentation / Community

If any questions during the development process should occur, help may be found in the following sources.

Name	Link
Clang-tidy documentation	https://clang.llvm.org/extra/clang-tidy/
Clangd documentation	https://clangd.llvm.org
LLVM's Discord Server	https://discord.com/invite/xS7Z362
Forum	https://llvm.discourse.group/

F.1.2 GitHub / Phabricator

LLVM's complete source code is hosted on GitHub. Special about the project is how it handles contributions. LLVM uses a tool called Phabricator to review and discuss contributions. When a contribution is accepted by the community, it is manually merged into the GitHub repository.

Name	Link
LLVM GitHub Repository	https://github.com/llvm/llvm-project
LLVM Phabricator	https://reviews.llvm.org/
LLVM Phabricator - How to Use	https://llvm.org/docs/Phabricator.html

F.2. Setup Development Environment for Clang-Tidy

F.2.1 GitHub Fork

Since the LLVM project is hosted on GitHub, its recommended to fork the repository with an own GitHub account. It should be mentioned that LLVM is a very actively maintained project so the forked repository will be deprecated soon. Regularly fetching the newest changes from the upstream repository helps to prevent merge conflicts in Phabricator's pre-merge tests when local changes are contributed to Phabricator.

F.2.2 LLVM for Visual Studio

To work on clang-tidy, which is a part of clang-extra-tools which is again a part of LLVM, it is necessary to setup a certain development environment. The recommended way to do that is described on the official documentation page:

- https://clang.llvm.org/get_started.html

The thesis team worked on Windows 10 with Visual Studio (without Ninja). During this setup, it was found that a command must be changed to get the right development environment for clang-tidy. At

the time of writing this thesis, the documentation recommended generating build files without clang-tools-extra, which is needed to get build files for clang-tidy. Also, the Visual Studio version was outdated. The correct command to generate the needed build files is as follows:

- `cmake -DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra" -G "Visual Studio 16 2019" -Thost=x64 ..\llvm`

As described in the official documentation, the above listed command will generate a Visual Studio solution file inside your `build` folder (which is created during the mentioned setup above) called `LLVM.sln`. This solution is structured in several sub-projects which can be used to build executables for a variety of LLVM tools, including a test suite that executes written clang-tidy tests.

➔ Beware: The first build of an executable like clang-tidy could take between one to two hours, depending on the computer's system resources!

➔ The build folder could take up to 20 GB of space!

Name	**LLVM.sln Project**
Clang-tidy build project	Clang executables\clang-tidy
Clangd build project	Clang executables\clangd
Clang-extra-tools test suite	Clang extra tools' tests\check-clang-tools

F.2.3 Visual Studio Code

Within mind of not writing clang-tidy checks only to use them with clang-tidy as standalone tool, its recommended installing LLVM's language server clangd. This can be done by downloading Visual Studio Code and installing the clangd plugin from the Visual Studio Code Marketplace. The installation can be used to test if a clang-tidy check already exists or if a developed check works as it should.

To include a clangd language server that was self-built, change the `Clangd: Path` setting inside the clangd plugin to the build folder of your LLVM project, e.g., `C:\llvm-project\build\Release\bin\clangd.exe`. Thus, every time a clangd executable is built with Visual Studio, it is immediately active inside Visual Studio Code.

Tools which are used by clangd such as clang-tidy can be configured how they should perform with clangd. Clangd's documentation website describes how different tools can be configured. To activate all available checks and remove some noisy ones, the following configuration can be used. By doing so, this can help to verify if a check, which is to be implemented, not already exists.

```
# %LOCALAPPDATA%\clangd\config.yaml
CompileFlags:
  Add: [-xc++, -Wall] # treat all files as C++, enable more warnings

Diagnostics:
  ClangTidy:
    Add: '*' # enable all possible clang-tidy checks
    Remove: [altera-unroll-loops, llvm*, fuchsia*]
```

Name	**Link**
Visual Studio Code	https://code.visualstudio.com/

Name	Link
Clangd Extension	https://marketplace.visualstudio.com/items?itemName=llvm-vs-code-extensions.vscode-clangd
Clangd Documentation	https://clangd.llvm.org/

F.2.4 Installation of LLVM Tools

During the development of new checks, using LLVM tools could be helpful, e.g., to generate an AST of a given source file. Download the latest `LLVM-X.X.X-win64.exe` executable which includes a lot of pre-built binaries. After running the mentioned executable, a folder `C:\Program Files\LLVM\bin` is created. The folder should be available inside the `Path` environment variable so that the pre-built binaries are available from the command line.

Name	Link
LLVM Pre-Built Binaries	https://github.com/llvm/llvm-project/releases/

F.3. Contribution Workflow for Clang-Tidy

This is a step-by-step manual to create a clang-tidy check. Prerequisite for this is that the steps specified in section Setup Development Environment for Clang-Tidy were followed. This is a finer structured workflow than the original documentation from https://clang.llvm.org/extra/clang-tidy/Contributing.html. It is also enriched with experience from the thesis team and includes steps which could speed up the development process.

1) Make sure the check does not already exist.

a) Go through the official documentation of clang-tidy checks and make sure the check you want to implement does not already exist.

i) https://clang.llvm.org/extra/clang-tidy/checks/list.html

b) To verify this, create a C++ test file in Visual Studio Code (clangd extension enabled) with code that should trigger the check you want to implement.

c) Enable all possible checks in clangd's configuration file (the provided configuration file from above is sufficient) and verify that your code example is not flagged by an existing check.

2) Add a new git branch.

a) Add a new git branch to be able to generate an exact diff between LLVM's main branch and the new check.

i) *git checkout -b branch-name*

b) To push this branch to the forked GitHub repo, change its upstream.

i) *git push --set-upstream origin branch-name*

3) Run a script to generate all needed files for creating a check.

a) First, define in which clang-tidy folder/module the check belongs. Decide between one of the given folders from: *llvm-project\clang-tools-extra\clang-tidy*

b) Define a proper name for the check. This name should be carefully selected, since changing it afterwards can be troublesome (see Section F.4 Pitfalls).

c) Execute the script *add_new_check.py* inside *llvm-project\clang-tools-extra\clang-tidy*:

i) *.\add_new_check.py <module> <check-name>*

ii) *E.g.: .\add_new_check.py cppcoreguidelines declare-loop-variable-in-the-initializer*

d) Be aware of the pitfalls which occur by running this script (see Section F.4 Pitfalls)!

4) Build Visual Studio Solution.

a) To integrate the new created files in the current Visual Studio LLVM solution, regenerate the solution file. Execute the following command inside *\llvm-project\build*:

i) *cmake -DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra" -G "Visual Studio 16 2019" -Thost=x64 ..\llvm*

b) After regenerating the solution, the implementation files could be found inside the Visual Studio Solution under the *Object Libraries* folder.

i) *Object Libraries/obj.clangTidy<module>Module*

5) Find the right nodes within clang's AST.

a) To understand which node of the AST should be matched by a check, it is useful to display Clang's AST generated from a source file with code examples that shall be flagged by the check in the future. There are different ways to display an AST. The most proven variant was using clang-check, another LLVM tool:

i) *clang-check -ast-dump C:\users\test\desktop\test.cpp --*

ii) More information about clang-check: https://clang.llvm.org/docs/ClangCheck.html

b) Introduction to Clang's AST: https://clang.llvm.org/docs/IntroductionToTheClangAST.html

6) Find a matcher query with clang-query.

a) Clang-query, another LLVM tool, makes it possible to execute AST matcher-queries on a given C++ source file. The provided sources include a complete list of such matchers and how they can be used. This tool is useful to experiment with AST matcher queries to match the node which should be flagged by the new check. This approach's advantage is that it is not required to build clang-tidy to test matcher queries, what saves build-time.

b) Sources:

i) AST matcher reference: https://clang.llvm.org/docs/LibASTMatchersReference.html

ii) How to use: https://clang.llvm.org/docs/LibASTMatchers.html

c) Usage:

i) Open clang-query command line: *clang-query.exe C:\users\test\desktop\test.cpp --*

ii) Run AST matchers on the given file, e.g.: *match forStmt(unless(has(declStmt())))*

7) Implement the check.

a) Once a satisfactory matcher query was found in the previous step, the check can now be implemented inside the check's previously generated .ccp/.h files. They are stored inside the folder you have chosen in step 3) and visible in Visual Studio as described in step 4).

b) If an already existing check from Cevelop should be re-implemented, its source code may serve as an inspiration:

i) https://github.com/Cevelop/cevelop/tree/develop/CevelopProject/bundles/

c) Test the check's implementation by regularly building the clangd or clang-tidy executable. After this is done as described in F.2.2, the executable can be used as described below.

i) Clangd: Use your Visual Studio Code installation from F.2.3. Open a test file in the editor to verify if a code snippet, which should trigger the check, is flagged.

ii) Clang-tidy: Run the following command to test your check:

(1) *llvm-project\build\Release\bin\clang-tidy.exe test.cpp --checks=<module>-<check-name>*

(2) With * every check is enabled, so the new create one should also be visible.

(3) Source: https://clang.llvm.org/extra/clang-tidy/

8) Write tests for the created check.

a) Write a test file with code snippets that should trigger the created check and others that should not. The *add_new_check.py* script generated the needed test file already. Clang-tidy's test files can be found in the following directory:

i) *llvm-project\clang-tools-extra\test\clang-tidy\checkers*

b) Section 6.5 Testing from the thesis's main part describes how checks can be tested.

9) Write documentation.

a) Add a short description about the check to clang-tools-extra's release notes:

i) *\llvm-project\clang-tools-extra\docs\ReleaseNotes.rst*

b) Add the same description as written in the release notes into the header file of the check. The header file can be found in the same folder as the check itself.

c) Add a description about what the check does to the check's documentation file. If reasonable, add code examples as well. This file also forms the basis of LLVM's website https://clang.llvm.org/extra/clang-tidy/checks/list.html, listing any existing clang-tidy check. The checks documentation file can be found in the following location:

i) *llvm-project\clang-tools-extra\docs\clang-tidy\checks*

d) Format C++ statements like while, for, do etc. with double ticks as ``for`` inside documentation texts.

e) An online RST viewer (e.g., http://rst.ninjs.org) might be helpful to check the appearance of written .rst files and to detect any formatting errors.

10) Merge fresh main branch.

a) LLVM is a frequently updated project. Because of this, its main branch is updated often. To prevent merge conflicts while contributing a new check to Phabricator, it is recommended to fetch the current version of LLVM's main branch and to merge the updated main branch into the check's specific branch before contributing. If this is not done, Phabricator's pre-merge tests might fail in their "setup" phase. The following documentation describes how to sync the local LLVM fork:

i) https://docs.github.com/en/github/collaborating-with-pull-requests/working-with-forks/syncing-a-fork

11) Format your written code.

a) Before committing to Phabricator, make sure every file is formatted correctly.

b) In Visual Studio, the shortcuts *Ctrl + k* and *Ctrl + d* can be used inside the LLVM solution to automatically format source files according to LLVM style guidelines.

c) To format an entire git commit, clang-format could be used inside the local LLVM repository:

i) *git clang-format HEAD~1*

ii) Increase the number after *HEAD~* by 1 and execute the command again.

iii) Do that for the number of commits which were created to implement the check.

12) Run the test suite.

a) It is important to run the test suite after formatting the code, because this could have an impact on the test results.

b) Open the Visual Studio LLVM.sln solution and build the test project: *Clang extra tools' tests\check-clang-tools*

13) Create a patch file.

a) Create a diff/patch file between the main branch and the check's branch. This file is needed to commit to LLVM's Phabricator.
   i) Inside the *llvm-project* folder, run this command: *git diff main -U9999999 > patch.txt*

14) Commit to LLVM over Phabricator.

a) Source: https://llvm.org/docs/Phabricator.html
   i) During this thesis, requests were created with *Requesting a review via the web interface.*
b) Helpful YouTube video: https://youtu.be/C5Y977rLqpw?t=707
c) Go to https://reviews.llvm.org/ and register a new user account.
d) Open https://reviews.llvm.org/differential/diff/create/ to upload a diff.
   i) Raw Diff: Use *Raw Diff*, this proofed to be the most reliable option during the thesis. Enter the content of the previous generated *patch.txt* file.
   ii) Repository: *rG LLVM Github Monorepo*
   iii) Visible To: *Public*
   iv) Click: *Create Diff*
e) On the next side, review the submitted code and choose *Create new Revision.*
f) On the next side, fill out the given fields and submit.
   i) Title: *[clang-tidy] <module>-<check-name>: a new check*
   ii) Summary: A short description of what the check does.
   iii) Reviewers:
      (1) One possible reviewer for clang-tidy is its code owner, listed in the following file: *llvm-project\clang-tools-extra\CODE_OWNERS.TXT*
      (2) It was found that often, the active reviewers were different from the ones mentioned in the CODE_OWNERS.TXT file. To find currently active maintainers, search through https://reviews.llvm.org/source/llvm-github/history/main/clang-tools-extra/clang-tidy/ and note which users are the ones with recent review comments.
   iv) Repository: *rG LLVM Github Monorepo*
   v) Visible To: *Public*
   vi) Editable By: *All Users*
   vii) Tags: *clang-tools-extra*
   viii) Subscribers: *cfe-commits (Mailing List)*

15) Observe the contribution.

a) After fulfilling step 14), Phabricator redirects to the created differential's page. Note that LLVM's term for "pull-request" is "differential". On this page, the selected reviewers comment their feedback and improvement suggestions. Furthermore, from this page, the uploaded patch can be updated with a newer version after feedback was incorporated.
b) For an overview of all created differentials and their status, visit: https://reviews.llvm.org/differential/
c) Also, if a LLVM reviewer comments on a differential, an email is sent to the email address of the used Phabricator account.

F.4. Pitfalls

- **add_new_check.py**: When generating all needed source files for a clang-tidy check with the given *llvm-project\clang-tools-extra\clang-tidy\add_new_check.py* script, it was found that some

unwanted changes were made. Inside the *llvm-project\clang-tools-extra\docs\clang-tidy\checks\list.rst* file, nearly all lines are deleted. What should happen is that only one new line for the documentation file of the new check is created, leaving all existing lines untouched. To overcome this problem, the changes made in *list.rst* should be reverted and a new line should be created manually. The already existing lines help doing this correctly.

- **rename_check.py**: clang-tidy offers a Python script to rename a check. During this thesis, upon using it, it was found that it does not work as expected:
  - Under *llvm-project\clang-tools-extra\docs*, the check must be renamed by hand inside *ReleaseNotes.rst.* The script does not do this automatically.
  - In the folder *llvm-project\clang-tools-extra\docs\clang-tidy\checks*, the checks file name must be renamed by hand. Also, inside the check's file, the old name has to be replaced with the new one. Inside the file *list.rst*, the checks name must be adjusted with the new name. Again, this is not done by the script.
  - The test file of the check in folder *llvm-project\clang-tools-extra\test\clang-tidy\checkers* also needs manual modifications. Its first line and every *CHECK-MESSAGE* comment must be adjusted so that they include the check's new name. Also, the file name itself must be changed.
  - The check's header file and the check's registration need to be added to the *llvm-project\clang-tools-extra\clang-tidy\<ModuleName>\<ModuleName>TidyModule.cpp* file.
  - Under *llvm-project\clang-tools-extra\clang-tidy\ModuleName\CMakeLists.txt*, the name of the check's .cpp file must be added. The already existing lines should serve as a good template for this.

G. Clangd Configuration File for Students

This clangd configuration file enables all checks which already exist inside clang-tidy from the comparison made in Appendix D, Comparison of Clang-Tidy and Cevelop Checks. Beside from that, also the ones created or altered in this thesis are listed. Please note: They currently work only with the self-built clangd server, which was submitted with this thesis, as they are not merged into the official clang-tidy code base yet.

The file can be used as documented on: https://clangd.llvm.org/config

- **Global:** `%LOCALAPPDATA%\clangd\config.yaml`
- **Project scope:** `.clangd`

```
# %LOCALAPPDATA%\clangd\config.yaml
#
# The first 8 lines are checks which already existed in clang-tidy, covering Cevelop checks
# The last 6 lines are checks which were created or altered in this thesis
# cppcoreguidelines-avoid-non-const-global-variables -> is redunant because a fix was implemented
# cppcoreguidelines-explicit-constructor-and-conversion -> is an alias for google-explicit-
constructor
#
# To enable a fix for cppcoreguidelines-avoid-init-default-constructors,
# activate additionally: modernize-use-default-member-init

CompileFlags:
  Add: [-xc++, -Wall] # treat all files as C++, enable more warnings

Diagnostics:
  ClangTidy:
    Add: [readability-identifier-naming, cppcoreguidelines-avoid-non-const-global-variables,
          cppcoreguidelines-prefer-member-initializer, google-explicit-constructor,
          readability-redundant-access-specifiers, llvm-include-order,
          google-global-names-in-headers, cppcoreguidelines-pro-type-member-init,
          cppcoreguidelines-init-variables, cppcoreguidelines-narrowing-conversions,
          cppcoreguidelines-pro-type-cstyle-cast, cppcoreguidelines-avoid-goto,
          cppcoreguidelines-special-member-functions, cppcoreguidelines-c-copy-assignment-signature,
          performance-noexcept-move-constructor,

          misc-avoid-std-io-outside-main, cppcoreguidelines-virtual-class-destructor,
          cppcoreguidelines-avoid-init-default-constructors,
          cppcoreguidelines-explicit-constructor-and-conversion,
          cppcoreguidelines-declare-loop-variable-in-the-initializer,
          cppcoreguidelines-avoid-do-while,
          cppcoreguidelines-avoid-non-const-global-variables]
```