Eötvös Loránd University

Faculty of Informatics

Dept. of Software Technology and Methodology

Architectural examination of student assignments with static code analysis

Supervisor:
Cserép Máté
Assistant Lecturer

Author:
Fekete Dóra
Computer Science MSc

Budapest, 2024

Contents

1 Introduction **3**

2 Background and related work **5**

2.1 The evolution of static code analysis 7

2.2 In education 8

2.3 Environment 10

2.3.1 Clang/LLVM 10

2.3.2 Clang-tidy 11

2.3.3 Creating custom checkers 14

2.3.4 AST Matcher library 16

2.3.5 Clang-static-analyzer 19

2.4 Literature review 21

3 Research and methodology **24**

3.1 Motivation 24

3.2 Architecture 25

3.3 Checkers 26

3.3.1 QWidget base 28

3.3.2 Persistence file 29

3.3.3 Representation leak 30

3.3.4 Illegal layer access 31

3.3.5 Structured namespace 32

3.3.6 Public members 33

3.4 Analysis of previous assignments 34

3.5 Statistics 37

3.5.1 QWidget base 38

3.5.2 Persistence file 39

3.5.3 Representation leak 40

3.5.4 Illegal layer access 40
3.5.5 Structured namespace 41
3.5.6 Public members 41
3.5.7 Summary 42

4 TMS integration **44**

5 Conclusion and future work **48**
5.1 Future work 49

Bibliography **50**

List of Figures **53**

List of Codes **54**

Chapter 1

Introduction

Evaluating and verifying student assignments in university classes is a great challenge to lecturers not only at Eötvös Loránd University but also worldwide as other studies and researches state. Molnar, Motogna, and Vlad [1] described it as "evaluating the quality of student code is a tedious and time consuming activity". Usually quite a large amount of students enroll into programming classes and with the limited time and resources teachers have there is the risk that they overlook certain errors in the students' works.

TMS (Task Management System), ELTE's internally developed – but open-source – tool is a web application to where students can upload their assignments. If configured that way, it is already able to perform certain checks, e.g. running tests with preset test cases or executing general static analysis automatically upon submitting.

Automating the process of verification with reliable tools takes off some parts of the burden of lecturers. It is not possible to eliminate the need of manual evaluation completely, because assignments don't always have requirements that cover everything, giving students relative freedom in implementation. For example, it may be an expectation in certain classes to create an application that uses the model-view architecture but to achieve this, there is not only one exact correct solution.

Another purpose of using automated tools that are also available to the students is that they can check their own work easily at any time before the deadline. The feedback from the tool gives them the possibility to enhance the quality of their assignments through the course of the class.

While students usually aim to upload their solution for an assignment only once,

especially for simpler tasks, if they see that the result of the analysis executed on TMS resulted in errors, they are likely to correct them. Getting feedback to their work this way is convenient for the students as they don't need to set up static analysis tools on their machine. Ayewah et al. [2] concluded the following on using FindBugs, a Java static analyzer tool: "The main lesson we learned from this experience is that developers will pay attention to, and fix, FindBug warnings if they appear seamlessly within the workflow." Johnson et al. [3] also emphasized the importance of these verification steps being built-in.

The purpose of this thesis is dual. On one hand, the course specific checkers integrated into TMS support both the students and the lecturers. The output of the analysis during the development helps the students by pointing out the problematic parts of their solutions, and during evaluation it is expected that the teachers will come across fewer errors, making evaluation easier and quicker for them.

On the other hand, numerous assignments from previous semesters are analyzed to prove the usefulness of the checkers and to identify common errors that students make in their work.

This work concentrates on student assignments written in C++ and verifying them structurally with custom static analysis using Clang's tools e.g. clang-tidy.

Chapter 2

Background and related work

There are multiple ways how the quality of a computer program can be measured. They can be grouped into two separate categories, depending on whether the analyzed program is executed.

When the running of the software is required, it is called dynamic program analysis. Unit, integration, system, and acceptance tests are all different levels of dynamic testing. These can be broken into different types like black or white box, automated or manual etc.

The name of the other category is static program analysis. This is the kind of analysis that is performed without the software being executed [4]. Code reviews can also be considered as part of this group, but that is a completely manual activity. The use of automated tools is a great help towards understanding the program and code, however, it cannot replace code review completely [5].

It is important to be aware of what automatic static analyzers can and cannot do and why they are considered useful. First of all, they cannot detect all problems [6]. What they can check is whether the program code corresponds to a predefined list of rules. If the result of an analysis shows zero problems that only means, that the possible error cases a person thought about earlier do not appear in the code. However, it can also be that in this situation we have a false negative, meaning that there is an unidentified issue. On the contrary, false positive warnings state that there is an issue where in fact there is none. Both cases can be dangerous. False negatives give a fake sense of security, while too many false positives can discourage developers from using the tools. In general, it can be said that the more precisely these rules, checks are defined, the less probable the occurrence of false negative or

false positive incidents.

By definition, "Rice's theorem states that all non-trivial semantic properties are undecidable" [7]. It is a generalization of the halting problem which "is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever" [8]. These mean that even with the most thorough static analysis checks it cannot be stated that the software will not ever run into errors during execution, but it is possible to give a good approximation [4].

The types of issues static analysis can identify are various. It can detect simple and obvious possible coding errors like syntax problems but can also mathematically prove properties. The latter is called the formal method. An example for that is the Hoare logic, which is a term used in contract-based programming. For programs, strict preconditions are postconditions are defined to which they must correspond. Other methods are abstract interpretation, data-flow analysis, model checking and symbolic execution.

The current, third generation of static analysis tools are able to detect around 15% of software weaknesses [9]. Lebanidze calls it a plateau, a limitation of usefulness of the out of the box static analysis tools. With customization, it is however possible to reach a somewhat higher percentage.

There is a difference between bugs and flaws [9]. The first one is more related to the programming languages themselves, while flaws are indicating problems in the implementation, like a misunderstood requirement or a fault in logic. Static analyzers are mostly capable of identifying only bugs.

The analysis can be made on different levels, from a single statement to the whole source code. Local analysis checks only one function by itself at a time, module-level examines a whole translation unit or class and the relationships inside the module. The top level is the global analysis where the entire program is checked [6].

In the industry static code analysis is the most important in the domains that are the most safety-critical. Such areas are medical, nuclear, aviation, automotive and machines [4]. High quality is essential for these kinds of software, because literally lives depend on them functioning correctly. Among other techniques, static analysis attempts to ensure that.

2.1 The evolution of static code analysis

Currently the static analysis tools that are used on the market belong to the third generation [9]. It's interesting to see how it evolved over time as it gives more understanding about what they are capable of now.

The first static analysis tools were simple home-made scripts that were only capable of lexical analysis. The first ones were used on C programs, `lint` is considered the first widely used static analysis tool. Mainly `grep` was used to search for certain keywords in the source code. This approach resulted in a huge amount of false positive issues, making the tools inconvenient to use.

The second generation tools were also available to the public as open source software like RATS, ITS4 and Flawfinder. Instead of lexical analysis, the source code was tokenized and the checks were applied to the tokens. This way the analyzer was able to differentiate for example between comments and function names, eliminating numerous false positives with it. Besides identifying frequently used dangerous functions, more types of software security weaknesses are covered, for example the TOC-TOU (Time Of Check, Time Of Use) problems. Checks for other languages, e.g. PHP, Python became available, too.

At the beginning of the 2000s computer hardware began to significantly be better, enabling more complex approaches. The need for them also grew because in the industry it was realized that the earlier stage a vulnerability is found, the cheaper it is to fix it. Also, many problems were not even solvable at a later stage. The different tool creators took different ways of implementing static analyzers, but there are certain aspects that are characteristic to all of them.

The largest improvement in the third generation is the use of the *abstract syntax tree* (AST). First, the AST is generated from the source code (for this, the code needs to be able to compile), and then the checks are executed on that. Besides the still used `grep` findings, control and data flow can also be construed.

The modeling concept also helps understanding the program by updating the changes to it during the process. Symbolic execution extends this by also keeping track of conditional states. While this approach is supposed to achieve a higher percent of accuracy, simulating the running of the program code has more drawbacks than advantages [10]. For example, an issue with byte code is that compilers usually do optimization, thus removing unnecessary statements. Obviously, if these do not

exist in the byte code, the analyzer is not able to report them either. Source code analyzis doesn't have such limitations [11].

A lot of convenient options were introduced in the third generation, such as suppressing and filtering certain warnings, the handling of false positive cases. More and more languages and frameworks are covered, e.g. Java. Another improvement is that the tools are now more flexible and extendable, making it possible to create custom checks.

2.2 In education

Multiple researches were conducted with the purpose to easing the workload of lecturers in computer programming classes [1, 11, 12]. Usually, these classes have a high number of enrolled students. In each year more and more people apply to programming studies at universities locally and globally. As Figure 2.1 shows, this trend can also be seen at ELTE. The statistics [13] covers the past 10 years, filtered for only the Faculty of Informatics. In Hungary education usually starts in the first, autumn semester of the year. The chart corresponds with this, as it can be seen that in every second semester the number of students is slightly fewer. While in 2014 around 28,000 students were admitted into the Computer Science programme, in 2023 this number was more than 38,000, showing a 36% increase in enrollment. In parallel with this, the need for more lecturers is also increasing.

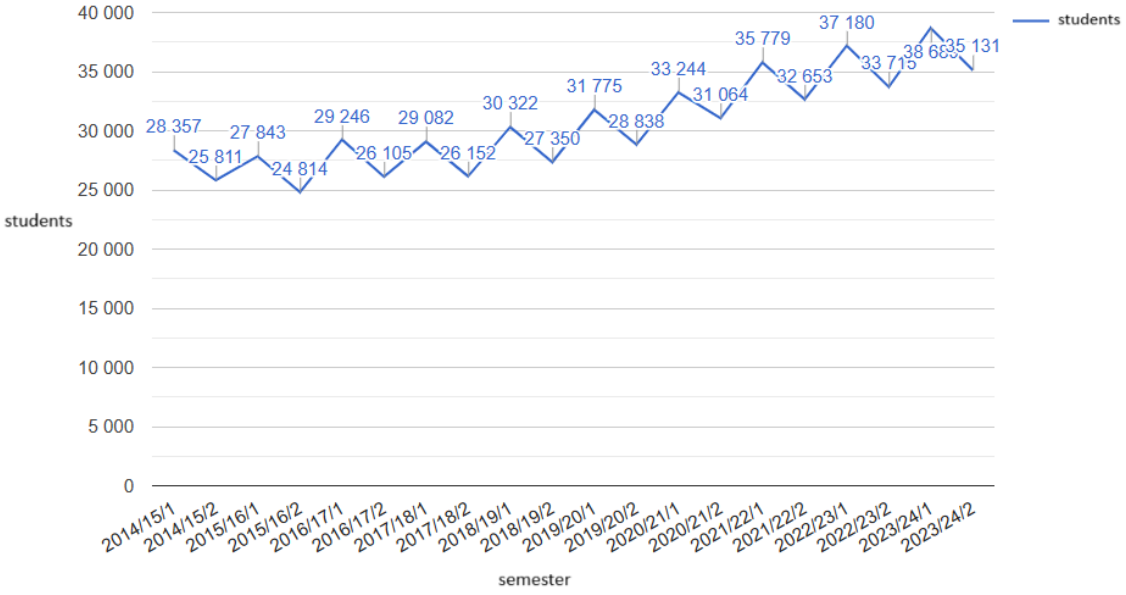

Figure 2.1: Number of students at ELTE, Faculty of Informatics

However, finding and training new teachers is difficult and time consuming, and the graduated students are more likely to start working in the industry than to stay at the university. Among the reasons it is a factor that jobs on the market offer more competitive salaries while their projects are also interesting to the graduates. This eventuates that the number of lecturers isn't growing proportionally as much as the students.

Because of these, the teachers of the courses face more challenges, e.g. the continuously increasing group numbers. It's not uncommon to have hundreds of assignments to be graded. Thus, automation of the grading process, or even partly having it supported by various tools is always welcome as it accelerates the evaluation.

TMS's basic feature is allowing students to upload files or compressed directories to the web application, which is then accessible to the lecturers. The simplest method of evaluation for the teachers is to download the handed in solutions, and then manually see through them. This takes a lot of time. For more complicated assignments, the work consists of multiple files.

Also, it's often a requirement in assignments to have a solution that compiles. To validate this, the student's work needs to be built. If done manually, this requires the teachers to have an environment that is also able to build with the same setup. However, having all these built-in in TMS enables lecturers to omit this step, as it can already be done via the web application.

This way the first step of the grading, namely seeing if the program compiles at all, is automated. The result of the compilation is available in the application. Other than building, test cases can also be defined to be run on the assignments. The output of the test result also supports the lecturers in the grading process.

Another feature, that is also already available in TMS is the usage of various out of the box static analyzer tools, e.g. CodeChecker, for multiple coding languages. Chapter 4 explains how it can be extended and used with even custom rules.

Besides teachers, the result of the automated tools in TMS is also available to students. Before the deadline it enables them to improve their solutions and eliminate the detected incidents, solely by seeing the compiler, test or static analysis errors and warnings. The students have an installed development environment for the implementation, but this way they don't need to install further tools. The testing process and the static analysis are executed only via TMS, in a stable setup. By reacting to the warnings and fixing them locally by the students on their machines,

it is very likely that the issues will be resolved. In worse case, other issues can be unearthed. Usually, the number of uploads is not limited in TMS for assignments, so the students are able to safely do their changes and verify its correctness by using the platform.

Although nowadays C# and Java are the popular choices when defining the tools to be used for a class, C++ is also available in some courses. Despite being less emphatic at the university, C++ is still an important and widely used programming language in the industry. As beginner programmers, students first get to know console applications while learning the basic concepts of programming. But for example, with the Qt framework, it's also possible to create applications with graphical interfaces. Because of these reasons C++ is a steady choice for static analysis.

2.3 Environment

2.3.1 Clang/LLVM

When building C++ applications, programmers can choose from various compilers. Different IDEs have their own built-in tools for compilation, configured with their custom setups. However, if one doesn't want to depend on the development environment, programs can also be built by using the Terminal.

The most used compiler infrastructures are GCC and Clang. As part of the LLVM project, which provides tools for C, C++ and Objective C, Clang is available along with lots of extra [14]. Among these tools there are clang-include-fixer, clang-rename, clangd, clang-format and for static analysis, clang-tidy and clang-static-analyzer. The tools are designed to support the developers working with C++ code, resulting in higher quality code. While some of these tools are aiming at making programmers life easier during code creation, others come handy in later stages.

The convenience tool, clangd, is a language server, which can be embedded into editors, offering code completion, and navigating through the source (e.g. go-to-definition). It can also highlight possible compile errors.

Another tool is clang-format, which is not limited to the C++ language. It can handle a wide variety of languages and numerous standards, resulting in a unified look of the source code. A custom configuration is also possible, allowing the individual projects to have their own formatting rule sets for their own requirements.

Clang-include-fixer automates adding `#include` directives to the source code, clang-rename is useful for refactoring, making it simple to change the name of an e.g. variable or method throughout the whole source. As it can be seen, these tools support the implementation effectively. Besides these, finding bugs and other problems is an area that is also important in development. Finding an issue as early as possible can be crucial in certain situations. Having automated tools that can detect them raises attention to deviances early on. To help this, clang-tidy and clang-static-analyzer are used widely.

2.3.2 Clang-tidy

Clang-tidy is a 2 in 1 tool, as it is not only able to detect certain issues, but can also fix them. It is based on Clang and can be described as a linter tool [15]. It can detect style violations (like clang-format), typical programming errors and certain bugs as well. It is also able to run clang-static-analyzer checks. On of the tool's limitation is that it can only check one translation unit at a time. There is a Python script (`run-clang-tidy.py`) that enables running multiple clang-tidy checks in parallel on a project, which is useful performance-wise and making the tool more usable, however, one of the reasons why it can be run parallel is that there is no connection between the translation units in the aspect of static analysis. For example, if a function is not used in a file, clang-tidy cannot detect if it is really unused, because it cannot carry the information along the analysis.

The tool consists of so-called checkers which are grouped into different categories. A check is an individual small building piece that examines one type or possible error on the translation unit. There are 25 groups currently, some are related to certain standards, like the Google or Darwin coding conventions, some others are scanning for errors in various technologies, e.g. boost, OpenMP API. They cover more general areas, too, like readability and performance. There is a separate category for modernization that helps porting to modern (C++11 and later) standards. Besides these, there is a miscellaneous group for everything that doesn't fit into any other categories.

While clang-tidy is a standalone tool, it can also be embedded into different kinds of IDEs, the most popular ones (e.g. CLion, Visual Studio Code, Vim, CodeChecker) already include it. The integration level can vary, while some editors offer on-the-fly

inspection, meaning that as the code is written, the analysis is done right away, others require running the tool separately. Another difference is the extension of configuration. In some of the IDEs the list of checks can be defined by the user instead of the default settings, some even enable using a custom clang-tidy binary.

However, using the tool standalone gives the maximum level of freedom of customization. A typical invocation of clang tidy is shown in Code 2.1 where in the program arguments the checkers that are supposed to run are specified along with the to be analyzed file. `-*` is used to disable all checks, then either a fully specified checker name is given in the format of `group-name-of-the-check` or the `group-*` is used as wildcard to include all. If a checker is listed with a `-` prefix, if means that it should be omitted. Thus test.cpp is run against the use-using checker of the modernize category and all of the readability checks except namespace-comment.

```
clang-tidy -checks=-*,modernize-use-using,readability-*,-
    readability-namespace-comment test.cpp
```

Code 2.1: Typical usage of clang-tidy (specify checkers)

As mentioned above, clang-tidy is capable of fixing certain issues. If the checker provides a suggestion on how to correct the detected problem, it is possible to apply that correction automatically with the program option `-fix` as seen in Code 2.2. Not all kinds of errors can be fixed in a simple way, for those probably no such suggestion is available.

```
clang-tidy -checks=* --fix test.cpp
```

Code 2.2: Typical usage of clang-tidy (apply fixes)

The output of the analysis is very similar to compiler warnings; thus it is easy to understand them. Code 2.3 shows a snippet of the result of a test code. The warning describes the issue (the usage of `typedef` should be avoided in modern C++) and it also offers a proper solution. If clang-tidy is run with the `-fix` argument, the source code is changed to the suggestion (`using Type = int`).

When running clang-tidy on a file or a project, the tool needs to know how and with what parameters the code should be compiled, even though the executable is not created nor used. Either compilation options (Code 2.4, they are listed after the `--`) or the path to the compilation database (Code 2.5, argument `-p`) needs to be provided. A combination of both is also possible. Clang and CMake can generate a compilation database for C++ projects. The result is a file named

`compile_commands.json` which contains all the translations units of the project source code and for each of them specifies "how to replay single compilations independently of the build system" [16].

```
...\clang-tools-extra\test\clang-tidy\checkers\modernize\use-using.
   cpp:3:1: warning: use 'using' instead of 'typedef' [modernize-
   use-using]
    3 | typedef int Type;
      | ^~~~~~~~~~~~~~~~
      | using Type = int
```

Code 2.3: Output snippet of the modernize-use-using checker

This is necessary for clang-tidy as it is an abstract syntax tree (AST) based tool and as such it has to know exactly how to parse the translation units to be able to create the AST. In the JSON formatted file there is a list of command objects, one for each translation unit, describing the working directory of the compilation, the name of the file and the arguments/command of the compilation. Arguments is a list of strings, while command is the concatenated version of arguments, the first one is preferred.

```
clang-tidy -checks=* test.cpp -- -DEXAMPLEFLAG=1
```

Code 2.4: Typical usage of clang-tidy (passing compilation options)

```
clang-tidy -checks=* -p build test.cpp
```

Code 2.5: Typical usage of clang-tidy (compilation database)

As header files are generally not considered as translation units, they are not listed in the generated `compile_commands.json` file. However, they can also contain errors that can be found by clang-tidy, thus when the program option `-header-filter=.*` is passed (see Code 2.6), the included header files are also analyzed by the checkers. Instead of `.*` any other regular expression can be given to specify which headers should be part of the analysis.

```
clang-tidy -checks=* -header-filter=.* test.cpp
```

Code 2.6: Typical usage of clang-tidy (headers)

Certain checkers can be configured by the `-config` command argument. To specify these parameters, a YAML or JSON formatted text is passed to the option, containing a CheckOptions map object. For example, the identifier naming checker in

the readability module can examine different casings, thus it is necessary to provide the type of convention to the checker as different projects comply to different standards.

In some cases, it might be necessary to suppress certain or all checks for a line or a bigger part of code, for example when facing a false positive warning. Clang-tidy offers a solution for this as well with the `NOLINT`, `NOLINTNEXTLINE` and `NOLINTBEGIN ... NOLINTEND` comments [15]. These keywords need to be used in code comments at the place of the code part in question. Code 2.7 show a few examples of how to use them.

```
// Suppress all the diagnostics for the line
Foo(int param); // NOLINT

// Silence all checks from the ‘google‘ module for all lines
   between the BEGIN and END
// NOLINTBEGIN(google*)
Foo(bool param);
// NOLINTEND(google*)
```

Code 2.7: Suppressing clang-tidy warnings

2.3.3 Creating custom checkers

Clang-tidy is designed to be modular and to be easily extended. The source comes with a Python tool (`add_new_check.py`) which takes care of registering the new checker in the specified module and adds all the necessary boilerplate code [10]. The invocation of the script is seen in Code 2.8. The first parameter is the module where the checker will be registered, the second is the name of the checker in dash-case. As result, a new class with the name of the checker is created in the corresponding module.

```
add_new_check.py misc structured-namespace
```

Code 2.8: Creating a new clang-tidy checker

It is also registered into the module, and the source is added to the `CMakeLists.txt` file to be included in the build process. Besides those, to be filled documentation notes are created at the appropriate places. Lastly, a test file is also created, that is used for unit testing the checker.

The newly created class is derived from `ClangTidyCheck` that contains two abstract methods, `registerMatchers(ast_matchers::MatchFinder *)` and `check(const ast_matchers::MatchFinder::MatchResult &)`. These methods are already overridden in the generated file, providing an example of how the implementation could look like. This example provides a checker that is very unlikely to be used anywhere in the industry. In `registerMatchers(...)` it binds to all function names in the translation unit and then in `check(...)` inspects whether the matched function name starts with the "awesome" prefix and reports a warning if it does not. It also offers a so-called quick fix in these cases. If the fix is applied, the function name is prefixed with this string.

This gives the developers a good starting point in implementing their own checkers. The `FIXME` comments are pointing the attention of the programmer directly to the code parts where it should be modified. However, it is also needed to understand the structure. `ClangTidyCheck` is the base class of all clang-tidy checkers, it is derived from the `MatchCallback` class. If the checker is enabled, for each translation unit a separate instance of the class is created, meaning that it is not possible (as of today) to store information among the translation units with this tool.

The `registerMatchers(...)` method is called first, filtering the abstract syntax tree for the desired defined parts. The parameter object `ast_matchers::MatchFinder` collects all the necessary matches in the AST via its `addMatcher(...)` method. After all of them are registered, the AST of the translation unit is traversed (pre-order), identifying all nodes or sub-trees that satisfy the predicates. For this the Clang ASTMatcher library is used, explained in detail in Section 2.3.4. Multiple matchers can be added.

Then for each match the `check(...)` method is called. Here it is possible to perform further filtering if needed. To report an incident, the `diag(SourceLocation Loc, StringRef Description, DiagnosticIDs::Level Level)` method is used. The first parameter is the location, a position in the source file, near where the error message is displayed in the output. The easiest way to get this is from the result node object, as most commonly it is sufficient to point to the beginning of the problematic code part. The second parameter is an arbitrary text, defined by the developer of the checker. Although it can be anything, it is advised to provide "useful hints" [11] instead of just stating that there is a problem at this line. The third parameter is the level of the diagnostic, ranging from Ignored to Fatal, by

default set to Warning.

As the `diag(...)` method returns a `DiagnosticBuilder` object, further elements can also be piped into it by the `«` operator. If it is clear how to fix the reported issue, `FixItHints` can be provided, which basically imitate how the source file would be edited manually. It has methods like `CreateReplacement(...)`, `CreateInsertion(...)` and `CreateRemoval(...)`, pointing to start and end locations, providing the new content. As mentioned in the by default generated example ("awesome" prefix for functions), a hint like that will take the original name of the function and replace it with the new name extended by the prefix.

In the generated unit test file both positive and negative cases should be present to be able to thoroughly test the checker before using it in an actual project. To verify the correctness of the test, a tool named llvm-lit (LLVM integrated tester) is provided in the llvm-project. In the beginning of the test file, comments starting with the `RUN` keyword contain all necessary information that is provided to the tool to run the test. Additional options, like specifying the C++ standard version is possible by passing the desired option here. If the checker is configurable, the config options can also be provided like that.

The tool llvm-lit [17] first runs clang-tidy on the test file, then it attempts to apply the fix it hints. The test file usually contains `CHECK-MESSAGES` and `CHECK-FIXES` comments. The messages are the exact warning messages that are expected in the result of the clang-tidy analysis. These are compared to the output result of the analysis and after applying the hints, it is compared to the source file that the content of the `CHECK-FIXES` comments appear there. If the result equals to the `CHECK-*` comments, the test is passed, otherwise the summary of the failures are reported.

2.3.4 AST Matcher library

After parsing and tokenizing the source code of the translation unit, the AST of it is built. "An abstract syntax tree (AST) is a data structure used in computer science to represent the structure of a program or code snippet" [18]. It is a tree that shows the structural connections, a formal description of the source code. The leaves of the tree are usually constant values or variables, while the nodes can be various statements or expressions.

If a source file has syntax or other errors, the AST cannot be created. As clang-

tidy heavily relies on the AST, this is why a compilable translation unit is the precondition for using the tool. Code 2.9 shows a very simple C++ program that creates a variable and calls a function, passing this integral variable as parameter. This code can be built, there are no compilation errors.

```
1  void fun(int & num)
2  {
3    ++num;
4  }
5
6  int main()
7  {
8    int i = 3;
9    fun(i);
10 }
```

Code 2.9: Example C++ file

The corresponding AST is shown in Code 2.10. The root node of the tree is a `TranslationUnitDecl`. It has two `FunctionDecl` child nodes, the first is `fun`, the second is `main`. Below the `fun` function there are two child nodes, a `ParamVarDecl`, as it has an `int &` parameter, and a `CompoundStmt`. Even though the statement has only one child, it is named compound. Inside the statement there is the unary increment operator with the variable that is passed to the function by reference. The second function declaration is the `main` function itself. It contains two children inside its `CompoundStmt`. The first is the creation of variable `i` and setting its value to 3 and the second is invoking `fun`, indicated by a `CallExpr`.

"LibASTMatchers provides a domain specific language to create predicates on Clang's AST" [19]. As seen on the example, most of the nodes have the same postfix, `Decl`, `Stmt` and `Expr`. These are the basic building parts of the DSL, `Expr` is considered as derived from `Stmt`. When traversing the AST, the execution starts from the root node (`TranslationUnitDecl`) and then recursively goes over all child nodes.

Clang provides a tool for displaying the AST of the translation units. Clang-check can be used with the `-ast-dump` program option to see the content of the whole file in the abstract syntax tree. Usually there are only parts of the code that are interesting for a problem, for that the tree can be filtered. The filtering is done by the AST matchers. For this purpose, another tool, clang-query can be used. It is a console application, that takes a file name as its program argument. In the console,

the `match` keyword is used, followed by an expression formed in the language of the AST matcher. For example, `match functionDecl()` in the previous example returns two matches, one for each function declaration. Not only the nodes themselves are returned, but the whole subtree of the match. If `match parmVarDecl()` is queried, only one match is returned, the parameter of function `fun`. As it is a leaf node, in this case the subtree consists of only one node. The matchers can be combined in various ways, for example to return only the `main` function, one could query `match functionDecl(hasName("main"))`.

```
1  TranslationUnitDecl 0x192e8cef110 <<invalid sloc>> <invalid sloc>
2  ... (omitted TypedefDecl parts for built-in types)
3  |-FunctionDecl 0x192e8e902a8 <...\ast_example.cpp:1:1, line:4:1>
       line:1:6 used fun 'void (int &)'
4  | |-ParmVarDecl 0x192e8e901e0 <col:10, col:16> col:16 used num 'int
       &'
5  | `-CompoundStmt 0x192e8e903d8 <line:2:1, line:4:1>
6  |   `-UnaryOperator 0x192e8e903c0 <line:3:2, col:4> 'int' lvalue
       prefix '++'
7  |     `-DeclRefExpr 0x192e8e903a0 <col:4> 'int' lvalue ParmVar 0
       x192e8e901e0 'num' 'int &'
8  `-FunctionDecl 0x192e8e90440 <line:6:1, line:10:1> line:6:5 main '
       int ()'
9    `-CompoundStmt 0x192e8e906c8 <line:7:1, line:10:1>
10     |-DeclStmt 0x192e8e905c0 <line:8:2, col:11>
11     | `-VarDecl 0x192e8e90530 <col:2, col:10> col:6 used i 'int'
           cinit
12     |   `-IntegerLiteral 0x192e8e90598 <col:10> 'int' 3
13     `-CallExpr 0x192e8e906a0 <line:9:2, col:7> 'void'
14       |-ImplicitCastExpr 0x192e8e90688 <col:2> 'void (*)(int &)' <
             FunctionToPointerDecay>
15       | `-DeclRefExpr 0x192e8e90640 <col:2> 'void (int &)' lvalue
             Function 0x192e8e902a8 'fun' 'void (int &)'
16       `-DeclRefExpr 0x192e8e90620 <col:6> 'int' lvalue Var 0
             x192e8e90530 'i' 'int'
```

Code 2.10: AST of the example C++ file

There are three different categories of matchers [20]. Node matchers are filtering for specific type, e.g. `functionDecl()`, `varDecl()`. The narrowing matchers are used to check for attributes, e.g. `isConst()`, `hasName("str")`. There are some

special matchers in this category, like `unless()`, with which the expression can be negated, or `anyOf()`, meaning that it is sufficient to satisfy at least one condition among its parameters for matching. The third category is the traversal matchers. These kind of matchers focus on the relationship between the nodes, e.g. `hasLHS()`, `isDerivedFrom()`. With these, dependencies between certain types can be revealed. This category also has some special matchers, for example `forEach()`, which is useful when multiple nodes need to be filtered and/or matched.

To be able to reference the matches, they need to be binded to a name. This is reached with the `bind()` function. For example, Code 2.11 only matches the `main` function in the example code and can be reached by the name `non_void_func`.

```
1 functionDecl(unless(returns(voidType()))).bind(''non_void_func'')
```

Code 2.11: Bind matcher

This name binding is useful when implementing the custom checkers for clang-tidy. In Section 2.3.3 the structure of the checkers was described. This AST matcher syntax can directly be used in the C++ code, the filters are added to the MatchFinder object in `registerMatchers(...)`. In the `check(...)` method the matched nodes can be invoked by the statement seen in Code 2.12.

```
1 const auto * MatchedNode = Result.Nodes.getNodeAs<FunctionDecl>("
    non_void_func");
```

Code 2.12: Retrieve binded node

2.3.5 Clang-static-analyzer

The tool clang-static-analyzer [21] uses symbolic execution to find bugs in the code. This method examines all paths in the program, building a graph of reachable program states. It uses symbols instead of actual values, making it more general to detect errors. The simplest problems that it can detect are for example the graph reachability problem, where a branch of the code is never reached/executed. Code 2.13 shows an example of such faulty code part. First, when printing the value of `num` onto the console, it hasn't been intialized, so memory garbage will be displayed, second of all, the else branch will never be executed, thus `fun2()` is not called. The static analysis tool can find both of these kinds of issues.

As the analysis is being executed, a graph is being built, at each decision point, new child nodes are added to it. The tool can carry information towards the nodes, like the states of variables, in the case of the example, whether `num` had been initialized. With the saved states, it can also handle dependencies between multiple translation units.

```
1  void fun()
2  {
3    int num;
4    if (true)
5    {
6      std::cout << num;
7    }
8    else
9    {
10     num = 2;
11     fun2(num);
12   }
13 }
```

Code 2.13: Example C++ file

This tool is based on event handling. Via the symbolic execution, different events are triggered, and by subscribing to them, various callbacks can be called. In these callbacks either the program state is modified or extended, or if the state reveals an issue, it can be reported. Similarly to clang-tidy, clang-static-analyzer checks also need to be registered to the checkers, however, this tool does not provide a utility tool for this, it needs to be done manually by the developer [22]. The checker classes are derived from the `Checker` template class, which defines what kinds of events the checkers attempts to process. Such an event can be e.g. `check::PreCall`, which is triggered before a method is executed in the source code. In its callback function, the developer can get and modify the current context of the checker, prevent further analysis on the investigated branch (by calling `generateSink()`), and if necessary, emit a `BugReport` object, where the problem is well described.

2.4 Literature review

The first step of the research was to look up other related previous works in the topic. Among the found papers not all of them were relevant here. Some of the sources simply described an assignment handling tool, similar to TMS, while others focused on the struggles students have during learning computer programming. The relevant research papers all agreed on the usefulness of static analysis in the evaluation of student assignments.

"Using static analysis tools to assist student project evaluation" [1] welcomes automation as it can shorten the time spent on grading. As it is all the time available for students, too, the result of the analysis can be used to improve the quality and detect issues. Having them identified before the grading also reduces the cost of the development, in this case, time. Identifying typical common issues helps in general, regardless of the focus of the assignment.

"Static analysis of source code written by novice programmers" [12] focuses on the improvement of code quality, revealing the typical programming errors and checking coding standard rules. As for the students writing their first coding assignments can be quite challenging, automation in the assessment applications is getting more and more popular. The study also compares the usage of Clang Static Analyzer and CppCheck by various viewpoints. The tools do not necessarily report the same errors, even though probably there is a common cross section. It revealed also that the most common errors students make are forgetting the initialization of variables and leaving unused ones in the submitted code. It is also concluded that "for students, locating bugs is more difficult than fixing them", meaning that if a tool is pointing the issues out to them, they are likely to be able to fix them by themselves.

"A study of the difficulties of novice programmers" [23] states that the most challenging for programming students is to learn and apply programming structures, and not understanding them, as teachers assume.

"A review of static analysis approaches for programming exercises" [11] concentrates on the quality of the feedback of the analyzer tools and what types of issues they can detect. Three big groups are identified: the code is fine, but the concept behind it is faulty; the code is not fine in a special context; and the code is not fine in any context. It is a common scenario that the code itself is syntactically correct, but it contains misunderstood concepts, such as not following coding conventions.

This is affecting not only students, but it is also a phenomenon that can be observed in production code environment. It can also happen, that in a certain context, using some structures are not allowed. The example describes the following situation: if the assignment is to implement a linked list data structure from scratch, the usage of `java.util.LinkedList` is not allowed (the study uses Java examples). The same scenario in C++ would be using `<list>`. The feedback that students get from the automated tools should be clear and they should not only identify the problems, but also try to provide a suggestion on how to fix it.

Besides education related sources, some industrial research papers were also reviewed. These also focus on the code quality and its improvement. Obviously, the university studies prepare the people for the professional work environment, so it is necessary to learn to apply good practices early on. During education, code review by peers is very rare, only required in the Software technology course. At other times, the solutions are only checked by tutors. However, in work environment, colleagues look at each other's code, which can also be a time-consuming activity as well as grading for teachers. Also, in this situation the inclusion of automation is popular.

Nowadays many different programming languages are used, and for them many different static analyzer tools exist, even multiple for a single language. A few examples are clang-tidy, clang-static-analyzer and CppCheck for C++, Rosslynator for C#, PHPStan for php and FindBugs for Java. However, the goal is the same for all of them: improving quality. "Using Static Analysis to Find Bugs" [2] analyzes the tool FindBugs and its capabilities. This study also states that "users analyzing older, more stable code bases are less likely to change code in response to a warning than users analyzing recently written code." Using the static analyzer tools from early in the development process increases the responsiveness to them and the elimination of the found issues.

Can Static Analysis replace Code Reviews? [5] is an article comparing code review to static analysis and whether the latter can substitute the manual code reviews. The conclusion is that it cannot, the best is to use both methods in parallel. Because of the analyzer tools' limitations, as it cannot understand the requirements and is not able to find every incident, the full automation is impossible.

"Why don't software developers use static analysis tools to find bugs?" [3] strengthens many of the already identified points. Besides the code quality and the meaningful error messages, it identifies the usage of the tool also as an impor-

tant factor. If the tool is built into the procedures of developer tasks and it is not interrupting the workflow too much, the programmers are more likely to use them. But false positive warnings can easily deter professionals from using these tools.

As this research focuses on assignments written in C++, the literature in this area was also reviewed. "Analysis of Include Dependencies in C++ Source Code." [24], "Clang matchers for verified usage of the C++ Standard Template Library" [10] and "Static analysis toolset with Clang" [25] are all studies that used Clang as the technology for the static analysis. They extended clang-tidy with specific checkers to resolve the researched C++ language specific issues.

Summarizing the resources, static analysis is a helpful tool in grading student assignments. As they are available prior to the evaluation, the students can also make use of it by improving their work before the final submission. When uploading the assignments, the instant feedback increases the chance of reviewing and resolving the reported issues. If the warnings are meaningful or even include suggestions about how to resolve the problems, they are more likely to be fixed. For the special assignments context specific analysis is recommended.

Chapter 3

Research and methodology

The question of this research is what kinds of architectural errors static analyzer tools can find and how can this support the evaluation of student assignments. It fits best into the correlational strategy; the used methods are testing and analysis. The test in this situation is not conducted with involvement of students, but rather executing tests on their assignments from previous semesters. The gathered data is later analyzed for frequency of typical errors identified by these tests.

3.1 Motivation

In the reviewed literature, the static analysis tools that were used for grading student assignments are out of the box tools that are not customized for the assignments themselves. Although Striewe and Goedicke [11] raise the idea of context specific checkers, the study remains general. The base concept of this research can be articulated as the following: "Since exercise specific feedback can only be given if exercise specific checks are created, this is a core criterion" [11]. Providing both the students and the teachers clear warning messages aims to support better results for the students, thus less time-consuming evaluation for the lecturers.

Both the analyzer tool and the analyzed code are written in C++. Despite losing from its popularity in the recent years, C++ is still a widely used programming language that can be used to teach various programming concepts. Coupled with different frameworks, e.g. Qt, it can also be the base of GUI applications. The course GUI programming with Qt was chosen as the subject of this study, where this technology is used. The main requirement of the assignments is to create a GUI

application following the model-view architecture. Examining whether the structure is correctly applied can be done by defining a set of rules the code should correspond to. Each rule is a separate checker.

Before implementing checkers, the tools of Clang were reviewed to decide on the more appropriate one. Between clang-tidy and clang-static-analyzer, the first one seems more usable to examine structural concepts, as it is focused on the AST of the translation units. The clang-static-analyzer is a better choice for verifying predicates during execution as it is based on event triggers and callbacks. Thus, the chosen tool for the research is clang-tidy. For each file, a set of rules (checkers) are executed in the student assignment projects, possibly revealing structural errors. The tool comes with a limitation; as it cannot analyze the dependency between the source files, it cannot be checked with it whether a layer is missing from the programs.

The expectation is for the research that some number of warnings will appear in the result, but not too many. Besides that, the integration of the checkers in the TMS application will support the students and the teachers in the future during the GUI programming with Qt course.

3.2 Architecture

There are numerous design patterns in software engineering [26]. During development the programmers usually face problems that can be resolved in a very similar way. Instead of reinventing the wheel repeatedly, these patterns offer a convenient solution, reducing development time and at the same time improving quality.

There are three pattern categories defined by the Gang of Four: creational, structural, and behavioral. Creational patterns are used for object instantiation. The most widely known is the singleton pattern, that ensures that a class has only one instance and that this instance can be accessed. Structural patterns define relationships, compositions between different objects, like the Adapter that converts one interface to another one (useful when using third party libraries). Behavioral patterns are used for communication between objects. At ELTE, one of the earliest patterns that student learn is the Iterator, which is also used in multiple courses at the university. This pattern allows accessing elements of an aggregated type without knowing the underlying structure.

Besides these categories, architectural patterns also exist. One of them is the model-view-controller [27] pattern, mainly used in web applications, however, it can be applied to most of the programs that have a graphical user interface. The first part of it is the model, which is the internal representation of the used data, the second is the view, that is displayed to the user, and they can interact with it. The third part connects these two layers together.

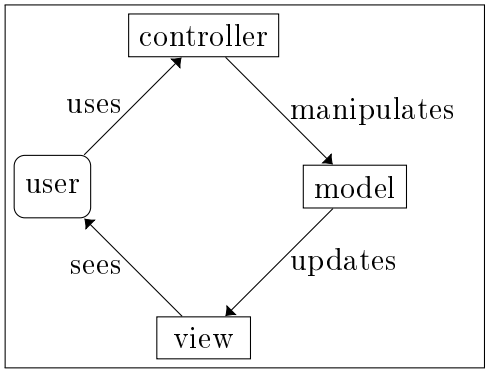

Figure 3.1: Model-view-controller architecture

Figure 3.1 shows the dependency between the different layers of the application. The model is the central component, if handles the data, the logic and the rules. When it changes, it updates the view. The view is displaying the data in its own format that can be different from the model's. The user of the application sees the view, usually it is the GUI. When the user clicks something or enter values on the graphical interface, the new information, command is forwarded from the view to the controller. In this layer, the input is converted to a command to the model, so that it can be updated. It is important that the view does not manipulate the model directly.

In the student assignments usually another layer is present, the persistence. This is responsible for saving the data onto the disk/database. It should be the only layer that accesses the storage, and it should be used by only the model. The correct architecture is that the view contains the model, and the model contains the persistence.

3.3 Checkers

The custom clang-tidy checkers extend the llvm-project code base [28]. The implemented additions are based on release 17 of the project. At the time this was the

latest stable version. As the repository is open-source and also actively improved, extended, using the main development branch could result in an unstable version of the software, because of its in-progress status.

To avoid interference with the work of other developers, the project was forked on GitHub, to be able to freely make modifications in the code base. As a first step, the development environment was set up. Because of limited resources, the installation happened on Windows platform. However, the code itself is cross-platform, so it does not influence the later usage of the extended tool.

The various checkers were created as described in Section 2.3.3. In total six checkers were implemented in the misc category. For each one, unit tests were also written. The Qt-specific code parts were mocked by creating empty classes with the appropriate names. This practice is also seen e.g. in the tests of the boost checkers.

"Solutions are often created based on code templates or at least prescribed method signatures, so assumptions about existing names for methods and perhaps variables can be used as an entry point for static analysis" [11]. With that thought in mind all the checkers depend on the naming of the classes created by the students in their assignments. For the model component it was mostly clear, the students used names like GameModel or similar. For other components the identification was not that simple, for example the persistence component goes from DataAccess to Persister. For the view component, it was considered what to detect and what not, as some students named it simply widget. However, QWidget is a very commonly used item in Qt GUI applications, so identifying everything as the view layer that have widget postfix would have led to far too many false positive cases. So, in this case it was decided to not include them as such, allowing some possible false negative cases.

As identified by the literature review (Section 2.4), it is a best practice to avoid false positive warnings as much as possible. Simultaneously it can result in not finding all the problems a checker could detect, but for the usability of the tool it has been considered by every checker, how strict it should be. If a checker result shows too many false positive warnings consistently, it may discourage teachers and students from using the tools or taking the result of the analysis into consideration to make further changes in the code.

The next sub-sections describe the implemented checkers in detail. Even though clang-tidy provides the possibility to offer quick fixes, none of the custom checkers

contain such solutions, as the structural problems are usually not that simple to resolve. The repository is available at GitHub[1].

In most cases (except the structured namespace checker) the matchers are well defined enough so that the `check()` method does not need any further processing on the matched node, it is sufficient to just report it.

3.3.1 QWidget base

`QwidgetBaseCheck` checks whether the persistence and model components do not derive from or use QWidgets, and in case of the view component, checks whether it has QWidget as base class. In a correct architecture setup, the persistence and model layers belong strictly to the backend, so they are not supposed to use any GUI elements, or be of such type. On the contrary, the view layer is responsible for the display, so it must be derived from QWidget.

For both the persistence and the model components, 4 matchers are added:

1. Binds all C++ record declarations that are defining a class, match the regular expression of the model/persistence component and derive from `QWidget` (not necessarily directly), except when the class name starts with `Ui_` (classes with this prefix are generated by the uic tool).

2. Binds all fields in C++ record declarations that are defining a class, match the regular expression of the model/persistence component, except when the class name starts with `Ui_`, and the field is or derives from `QWidget`.

3. Binds all C++ method declarations that are defining a method of classes that match the model/persistence component's regular expression, except when the class name starts with `Ui_`, and anywhere inside their AST there is a variable (including function parameters) that is or is derived from `QWidget`, including references and pointers.

4. Binds all C++ method declarations that are defining a method of classes that match the model/persistence component's regular expression, except when the class name starts with `Ui_`, and their return type is or is derived from `QWidget`, including references and pointers.

For the view component, only one matcher is registered. It binds all C++ record declarations that are defining a class, match the regular expression of the view

[1]https://github.com/feketedora/llvm-project/tree/masterthesis-17, the extract of the modifications is attached to the thesis.

component, except if it is derived from `QWidget`, or when the name of the class or any of its bases starts with `Ui_`.

```
class Model : public QWidget {}; // 1st model/persistence matcher
class Model {
  QWidget MyWidget; // 2nd model/persistence matcher
};
class Derived : public QWidget {};
class MyPersistence {
  void fun1 (Derived & D, QPushButton * B) { // 3rd model/
     persistence matcher
    QWidget Var;
  };
class Model {
  QWidget * getWidget(); // 4th model/persistence matcher
};
class View : public QObject {}; // view matcher
```

Code 3.1: Examples where the QWidget base checker binds matches

All matched nodes are reported to the diagnostics with the corresponding explanation. Code 3.1 shows an example for all kinds of matchers the checker operates with. At first the checker only checked the model component, but it was then extended to the persistence layer as well. The view layer check was also included. The `Ui_` prefix filtering was added at a later point of the implementation, when running the checker on the student assignments revealed a few false positive cases because of this.

3.3.2 Persistence file

`PersistenceFileCheck` checks whether there is file handling outside of the persistence component. In the correct structure only the persistence layer should access the disk and save to or load from files. Besides the common C++ streams, the types are extended by `QFile` from Qt.

There are 2 matchers in this checker, the first one filters the fields of C++ classes that do not match the persistence component's regular expression, or when the class name starts with `Ui_`, and the fields are of or derived from any of the following types: `std::fstream`, `std::ifstream`, `std::ofstream`, `QFile`, including references and pointers.

The second matcher filters the variables (including function parameters, references and pointers) of C++ methods in classes that do not match the persistence component's regular expression and they are not of any of the file stream classes, but the variables are of any of these types.

```
class Model {
  std::ofstream S; // 1st matcher
  void write(QFile * F); // 2nd matcher
};
```

Code 3.2: Examples where the persistence file checker binds matches

All matched nodes are reported to the diagnostics with the corresponding explanation. Code 3.2 shows examples of matched nodes. The first version of the checker didn't exclude the file stream classes from the C++ method matcher, resulting in a lot of false positive matches, because the implementation of the streams also contains objects of these classes. This was fixed in the final state.

3.3.3 Representation leak

`RepresentationLeakCheck` checks whether a class leaks model, view or persistent component representation via either a public field or function return type. When such a component is exposed, it is possible that other components take advantage of this, and directly use them instead of via the correct structural layers.

For each component, 2 matchers are registered. The first one matches the C++ method declarations that are public, are not the `operator=` functions, and the return type is either a pointer or a non-const reference to an object of classes that match the model, view or persistence components' regular expression. The `operator=` function is filtered out as it returns a reference to the object of the type, resulting in false positive matches in case of the component classes. Returning a copy of a const reference to the object is allowed, because even it is not fortunate to access the data of these components this way, the manipulation is still avoided.

The second matcher filters the public fields of C++ classes, except when the class name starts with `Ui_`, where the type of the field matches any of the components' regular expression, including references and pointers.

All matched nodes are reported to the diagnostics with the corresponding explanation. Code 3.3 displays examples of matches the checker produces.

```
class MyClass {
public:
  Model & refFunc(); // 1st matcher
  Model publicModel; // 2nd matcher
};
```

Code 3.3: Examples where the representation leak checker binds matches

3.3.4 Illegal layer access

`IllegalLayerAccessCheck` checks the correct containment of the components: view contains model, model contains persistence. All the other containments are considered illegal, either because the view component omits the model layer to access the persistence component directly, or the inclusion is reversed, resulting in a structural logic error. Figure 3.2 illustrates the described situation, the orange arrow represents omitting the intermediate component, the red arrows indicate logical issues.

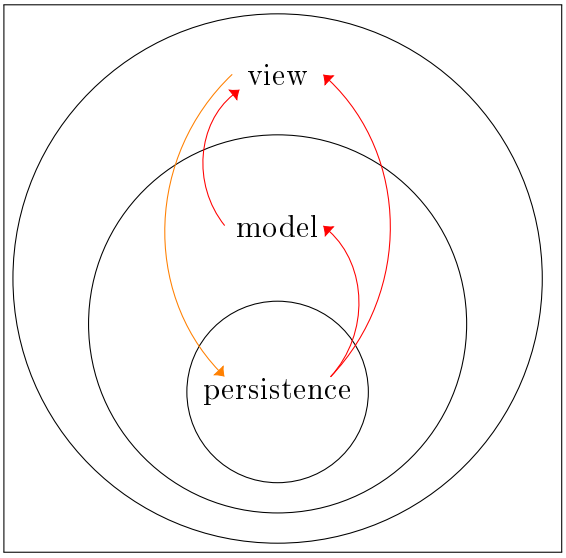

Figure 3.2: Model-view-persistence illegal layer accesses

Each illegal access type consists of 2 components, where from one (container) to the other (contained) the access is not allowed. For each of these types 2 matchers are registered. In both cases the container and the contained class also match derived classes, too. The first binds fields in classes, where the classes are of the container type, and the fields are of the contained type, including references and pointers.

The second matcher binds variables in C++ class methods (including function parameters), where the class is of the container type and the variables are of the contained type, including references and pointers.

```
class View {
  Persistence * persistence; // 1st matcher
};
class Persistence {
  void fun (Model * M); // 2nd matcher
};
```

Code 3.4: Examples where the illegal layer access checker binds matches

All matched nodes are reported to the diagnostics with the corresponding explanation. Code 3.4 shows 1-1 example for both the field and variable cases where the matchers find nodes. In the first version of the implementation the focus was only on the skipped layer, but the result of the analysis of the student assignments revealed that some students had issues with the containment of the components, so the checker was extended with the reversed relationships.

3.3.5 Structured namespace

`StructuredNamespaceCheck` checks whether the components (model, view, persistence) belong to the correct corresponding namespace structure, if any namespaces exist. For each component, 1 matcher is registered. It binds C++ record declarations of class definitions where the name of the class matches the components' regular expressions, except if name starts with `Ui_`.

```
namespace view {
  class Model {};
}
```

Code 3.5: Examples where the structured namespace checker binds matches

The different variants (e.g. for the persistence layer also database is acceptable) of the component names are checked individually, in every case the matched nodes are processed further. For each of them the full qualified name contains all the namespaces of the class. Splitting them by the `::` separators enables comparing all the namespace levels to the current variant. If a namespace contains the variant, it

is reported to the diagnostics with the corresponding explanation. Code 3.5 shows an example of an incorrectly used namespace structure.

As the usage of namespaces is not a requirement in the examined course, this checker is not expected to create any results, not even false positive cases.

3.3.6 Public members

`PublicMembersCheck` is not connected to the model-view-persistence architecture, it aims at a more general bad programming practice, namely the exposing of field members. As a rule of thumb, all class members should be either protected or private, and if accessing them from other classes is necessary, getter and/or setter methods should be implemented to provide this data.

This checker checks public fields in C++ records that are defined with the class keyword. It binds the public fields of C++ record declarations, if the record is introduced by the class keyword, except when name of the class starts with `Ui_` or it matches the data regular expression.

```
class MyClass {
public:
  int Var; // matcher
};
```

Code 3.6: Examples where the public members checker binds matches

All matched nodes are reported to the diagnostics with the corresponding explanation. Code 3.6 displays an example where the checker finds a match. There is a common cross section between the result of this checker and representation leak checker, but while this one is not restricted to the component names, the other one is also able to detect return types as well.

The first implementation considered the public members of every record type, but as this resulted in an enormous number of matches, it has been made stricter and only considering the records that use the `class` keyword, allowing public members in records defined by the `struct` keyword.

3.4 Analysis of previous assignments

As part of the research, previous assignments from the GUI programming with Qt class were analyzed. To be able to create statistics, a Python script[2] was developed that handled the mass processing of the data.

With a few exceptions all student assignment directories contained a `.pro` file, the QMake project file. Besides that, the headers, source files and in some case .ui files were included. The latter kind of file contains the representation of a widget tree, in XML format [29]. At compile time compilable C++ code is generated from them. For example, if the source has a `gamewidget.ui` file, in the corresponding `GameWidget.h` file the include of `ui_gamewidget.h` is needed. During compilation, CMake runs the `uic` tool that creates the header from the `.ui` XML file.

Some directories consisted of more than one project. Usually in these cases the second one was a test project, if the student decided to create it separately. In other cases, the test directory was part of the only project. During the analysis the test projects were omitted.

As described in Section 2.3.2, clang-tidy requires a `compile_commands.json` file. It can be created with different tools. The chosen procedure in this case was the following: Qt provides a Python script (`run_pro2cmake.py`) that generates a `CMakeLists.txt` file from a `.pro` file. Then the configuration of cmake has to include the `CMAKE_EXPORT_COMPILE_COMMANDS` option, and using Ninja as the generator, while also specifying the correct CMake prefix path. After these settings, the project was successfully built and the `compile_commands.json` file was generated in the build directory. With one exception using C++17 as the C++ standard was sufficient, one student however utilized C++20 features, thus their assignment needed the newer version.

As the assignments were originated from earlier years, they mostly use Qt5 as the framework. However, this created a few challenges as the `run_pro2cmake.py` tool is only available in Qt6, and the output was also using Qt6 features. Fortunately, some simple textual changes in the `CMakeLists.txt` files fixed the compatibility issues so the projects could be built with Qt5. As there are several differences between the two versions of Qt, it was necessary to do the compilation with the same major version as the assignments. For example, the use of deprecated features only appeared as

[2]The script (`analyze_all.py`) is attached to the thesis.

warnings in Qt5, but in Qt6, these became obsolete, causing compilation errors. Some student works were created with Qt6, in this case this modification was not necessary.

Building the projects in general is not necessary for clang-tidy as it operates on the source code, not on the executable, but in this situation, it was needed. Without building, the C++ files were not generated from the `.ui` files and this led into false positive compiler errors as seen in Code 3.7.

```
error: 'ui_gamewidget.h' file not found [clang-diagnostic-error]
```

Code 3.7: False positive compiler error without build

The next step was running clang-tidy on the solutions. The tool used for it is a Python script (`run-clang-tidy.py`) provided by the llvm-project repository, allowing clang-tidy to run in parallel on multiple translation units, accelerating the analysis process. Among its arguments it is possible to define which checkers should be executed. All the custom checkers are listed besides `clang-diagnostic-error`. The diagnostic errors provide extra information, to see why the compilation of the project failed. In most cases despite containing build errors, clang-tidy was still able to process the checkers on the code.

The final step was processing the clang-tidy outputs. For each assignment, all student works were collected into one result file. The errors and warnings were listed for each student, and one file summarized all the findings.

Figure 3.3 shows the algorithm of the data processing. As the student assignments did not change, it was sufficient to only prepare them once, and if the `compile_commands.json` file has already existed, clang-tidy could be run at once.

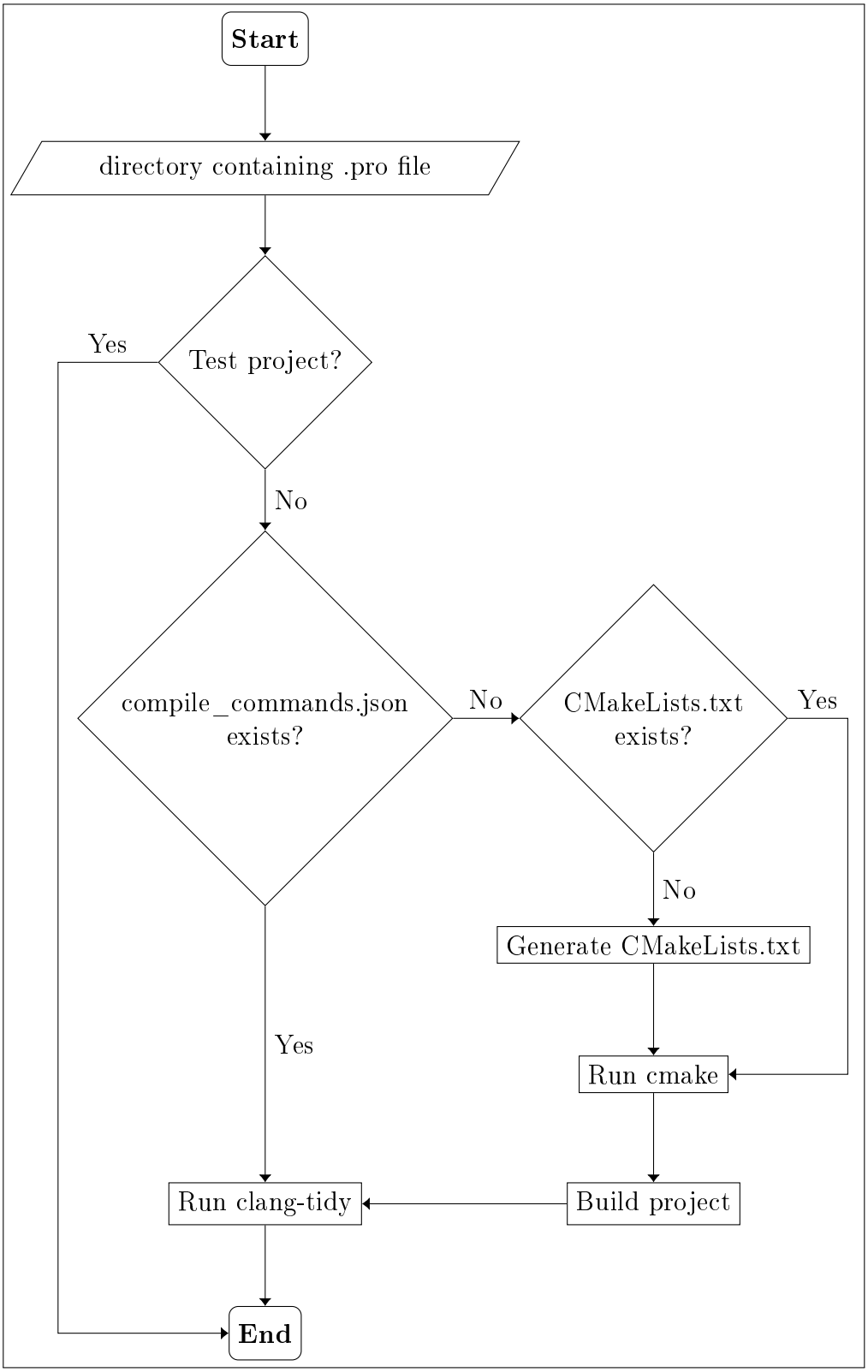

Figure 3.3: Processing of a student assignment

3.5 Statistics

```
Total assignments: 290
Total analyzed assignments: 282
Total analyzed assignments containing errors: 9

clang-diagnostic-error: 19
misc-qwidget-base: 33
misc-persistence-file: 22
misc-representation-leak: 26
misc-illegal-layer-access: 16
misc-structured-namespace: 0
misc-public-members: 374
```

Figure 3.4: Statistics summary

The data used to generate statistics from originate from the GUI programming with Qt course, from 2 autumn semesters (2021 and 2022). Both courses contain assignments and exam works. The first one has 2 assignments, 1 exam and 1 retake exam, the second one has 1 assignment, 2 exams and 1 retake exam. The exam is similar in structure to the other assignment, it will be referenced also as assignment for clarity. Each assignment directory contains around 40-50 student works, except the retake exams, which contain around 10 works. In one semester, it is expected to have mostly the same students' works 3-4 times, one in each different assignment. In the first assignment of the first semester the existence of the persistence component was not a requirement.

Figure 3.4 shows the cumulated numerical results[3] of the static analysis using the custom checkers. The total number of assignments that were available for the study is 290, from which 8 could not be analyzed. In 1 case the problem was the lack of a build file (neither `.pro` nor `CMakeLists.txt` files were present). In the 7 other assignments, the directories were empty.

In 9 of the assignments, compilation errors were detected. In a few cases these were caused by using non-standard (`bits/stdc++.h`) or third-party headers, in others undeclared identifiers, incomplete types (because of missing includes) or even syntax errors were found. Most of these assignments however produced also output from the custom checkers.

[3]The whole result data is attached to the thesis.

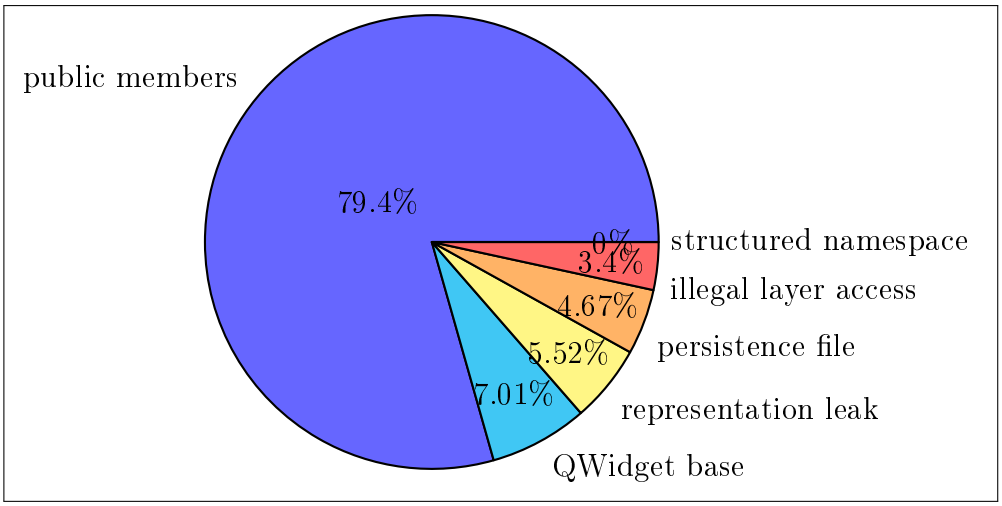

Figure 3.5: Proportion of errors per checker

The rest of the assignments were successfully built, and their clang-tidy outputs can be considered completely valid. While some assignments did not result in any warnings, others produced multiple types, occasionally even 3 different ones. There was no clear pattern of the occurrence of the separate types or warnings. Sometimes the student made the same type of error multiple times, in other case, it happened only once, and not necessarily in the first homework.

Four checkers produced around 5-5% of the warnings, as it can be seen on Figure 3.5, however, these cases show actual structural errors the students made in their works. The result of the other two checkers is less relevant.

3.5.1 QWidget base

The QWidget base checker identified 33 cases in a total of 9 assignments. The most common warning types were that the students derived their model or persistence class from `QWidget`, or they used elements like `QPushButton` or `QLineWidget` in the model component, which should belong to the view component. Some examples are shown in Code 3.8 In no cases was it detected that the view component wouldn't be derived from `QWidget`.

In 1 case the student used `QFileDialog` in the persistence component. They called the `saveGame()` method directly from the view component, omitting the model component, even though the code had the correct interface for it. In this situation the correct structure would be to have the `QFileDialog` in the view component, upon accepting the dialog, forwarding the file name to the model component

and through it to the persistence component, which then finally does the saving onto the disk.

```
...\raktarmodel.h:8:7: warning: model class methods must not return
    QWidget or its descendants [misc-qwidget-base]
    8 | class RaktarModel : public QWidget
      |       ^

...\model.cpp:59:32: warning: model class must not use QWidget or
   its descendants [misc-qwidget-base]
   59 | void Model::build(QPushButton *btn)
      |                                ^
```

Code 3.8: QWidget base checker warnings

3.5.2 Persistence file

```
...\persistance.cpp:14:11: warning: function handles files outside
   persistence class [misc-persistence-file]
   14 |     QFile file(fileName);
      |           ^

...\beadando\menugui.cpp:47:15: warning: function handles files
   outside persistence class [misc-persistence-file]
   47 |         QFile file(actualpath);
      |               ^
```

Code 3.9: Persistence file checker warnings

The persistence file checker raised 22 warnings in 11 student assignments, 2 in average for each of them. None of the exams contained any errors from this category, only two of the homework assignments. Most of the found issues in this case were false positives, because the students either used an uncommon naming (e.g. `SaveAndLoad`) or they had a typo in the name of the persistence class, most commonly they named it "persistance". The misspellings should be considered as errors, but for a different reason. In the industry it can lead to information loss or severe confusion, even program errors, if a typo goes deep in the code base, or the same structure is referenced by several different names. The students should learn early on to avoid misspellings and to be more aware of it. In a few cases of the result of the persistence file checker the warnings were valid, however it was difficult to tell,

which component does their class belonged to, assuming the view from the naming and content. This can be seen in the second example of Code 3.9.

3.5.3 Representation leak

The representation leak checker found 26 cases in 18 assignments. Most of them were also found by the public member checker, but this one is more specific and checks return values of methods, too. The most common error the students made was having a public model object pointer in the view component, in some other cases, having a public persistence object pointer in the model component. In fewer cases, the component was leaked through the return type of a public class method. Code 3.10 shows two examples from the detected warnings, the first one is where the view component leaks the model by a public field, the second one leaks the persistence from the model via the return type of the `get_persistence()` method.

```
...\view.h:21:12: warning: field leaks representation [misc-
   representation-leak]
   21 |     model* _model;
      |            ^

...\bead2\chameleons\test\chameleonsModel.h:34:28: warning: method
   leaks representation via return type [misc-representation-leak]
   34 |     ChameleonsPersistence* get_persistance() const;
      |                            ^
```

Code 3.10: Representation leak checker warnings

3.5.4 Illegal layer access

The illegal layer access checker detected 16 warnings in 10 assignments. One student made the same type of error in 3 of their assignments. Except 2 situations, the detected error was including the persistence component directly into the view.

```
...\bead2\ConnectFour\app\view.h:27:18: warning: view class must
   not have persistence or its descendant fields [misc-illegal-
   layer-access]
   27 |     Persistence *persistence;
      |                  ^
```

```
...\Test\robotpigsdatapersister.h:9:21: warning: persistence class
   must not have model or its descendant fields [misc-illegal-layer
   -access]
    9 |     RobotPigsModel* _model;
      |                     ^
```

Code 3.11: Illegal layer access checker warnings

The exceptional cases were more concerning, as in these cases the persistence component contained the model. Code 3.11 displays 1-1 warning for each identified error type.

3.5.5 Structured namespace

As expected, the structured namespace checker detected 0 issues, as the usage of namespaces was not expected from the students.

3.5.6 Public members

On the contrary, the public members checker produced hundreds of warnings, even with the restriction of only detecting records that are defined with the class keyword. Although it is considered as a bad practice, the lecturers probably did not define it as a hard requirement. This was the most common detected warning type, it appeared in 61 assignments, which is 22% of the total assignments. Usually if a student used public members in classes, they used more than once, either around 3-4, or even 10-15 times, 6 in average. The greatest number of public members in a single assignment was 22. An example of the warnings is seen in Code 3.12, where the student made the `gameLost` field accessible to other classes as well.

```
...\Hirsz\searchmodel.h:29:5: warning: avoid using public members,
   use private members with getter and setter methods instead [misc
   -public-members]
   29 |     bool gameLost;
      |     ^
```

Code 3.12: Public members checker warnings

3.5.7 Summary

Table 3.1 shows an overview of all checkers and their results per student assignments. The four relevant checkers produced warnings in an average of 12 assignments. Compared to the total number this does not seem like a high value, however, they mostly appear accumulated, and the homeworks usually contain more errors than the exams. The exams are usually smaller tasks, thus less prone to errors, while the home works, where the students can spend days or even weeks on the solution, are more complex, and if some of the concepts are unclear, they will more likely appear in these assignments. It was also seen at multiple occasions that the student made the same error in the exam that they made previously in the assignment. Had the problem been identified during the homework, they probably wouldn't have kept the wrong structure.

The representation leak checker detected issues in the most assignments (apart from the public members). This shows that the concept of encapsulation and hiding data is an area that is challenging for some students. The Qt specific checker also revealed conceptual difficulties.

The grades of the assignments were not examined, as they could produce false output. It cannot be identified from the available data whether a student got the bad grade because of the quality of their work or because they failed to upload it until the deadline.

	2021/2022/1			
	assignment 1	assignment 2	exam	retake exam
misc-qwidget-base	0 in 0	5 in 2	3 in 1	0 in 0
misc-persistence-file	0 in 0	7 in 3	0 in 0	0 in 0
misc-representation-leak	0 in 0	6 in 2	5 in 3	0 in 0
misc-illegal-layer-access	0 in 0	7 in 4	1 in 1	1 in 1
misc-structured-namespace	0 in 0	0 in 0	0 in 0	0 in 0
misc-public-members	7 in 2	81 in 16	55 in 8	11 in 2

	2022/2023/1			
	assignment	exam 1	exam 2	retake exam
misc-qwidget-base	15 in 3	0 in 0	10 in 3	0 in 0
misc-persistence-file	15 in 8	0 in 0	0 in 0	0 in 0
misc-representation-leak	9 in 7	0 in 0	6 in 6	0 in 0
misc-illegal-layer-access	7 in 4	0 in 0	0 in 0	0 in 0
misc-structured-namespace	0 in 0	0 in 0	0 in 0	0 in 0
misc-public-members	92 in 12	34 in 7	84 in 12	10 in 2

Table 3.1: Summary of the static analysis result (#issues in #assignments)

Chapter 4

TMS integration

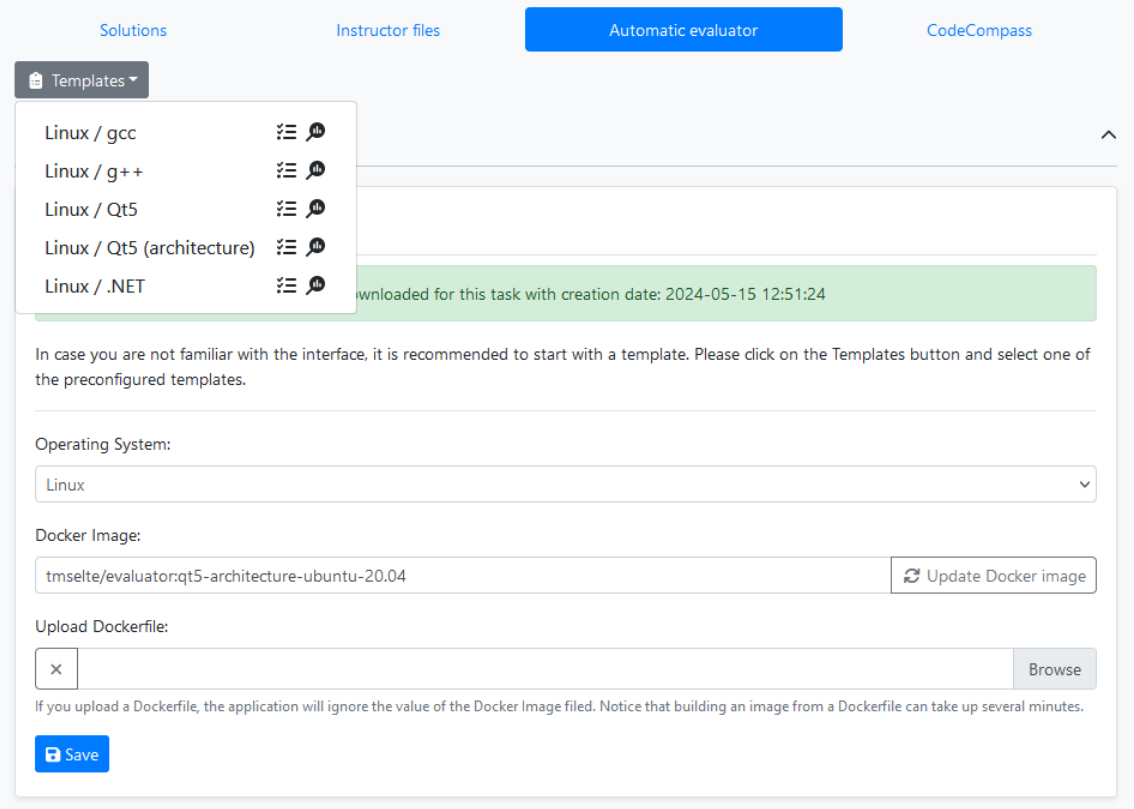

Figure 4.1: Automatic evaluator templates in TMS

As the result of the work of Kaszab and Cserép [30, 31] it is already possible to execute different kinds of evaluators and static analyzer tools on the student assignments in TMS. Figure 4.1 shows the list of templates[1] that can be used in the application for Linux operating system. Linux / Qt5 (architecture) is the one that

[1]The merge request of the template configuration is available at https://gitlab.com/tmselte/backend-core/-/merge_requests/159

contains the custom clang-tidy checkers. A separate Docker image[2] belongs to all these templates, where the environment for the evaluation is set up.

Docker [32] is a Platform-as-a-Service system, allowing a way to set up infrastructures to be able to run applications easily. The Docker image Linux / Qt5 (architecture) is based on the image of Linux / Qt5, but instead of the pre-installed clang and clang-tidy, these programs are built from the repository that contains the custom checkers. The llvm-project provides scripts to create and install the executables of the project from any branch. In this case, even the fork is customizable, as the checkers are not part of the official llvm-project repository. As these checkers are too special, it was decided to keep them separately.

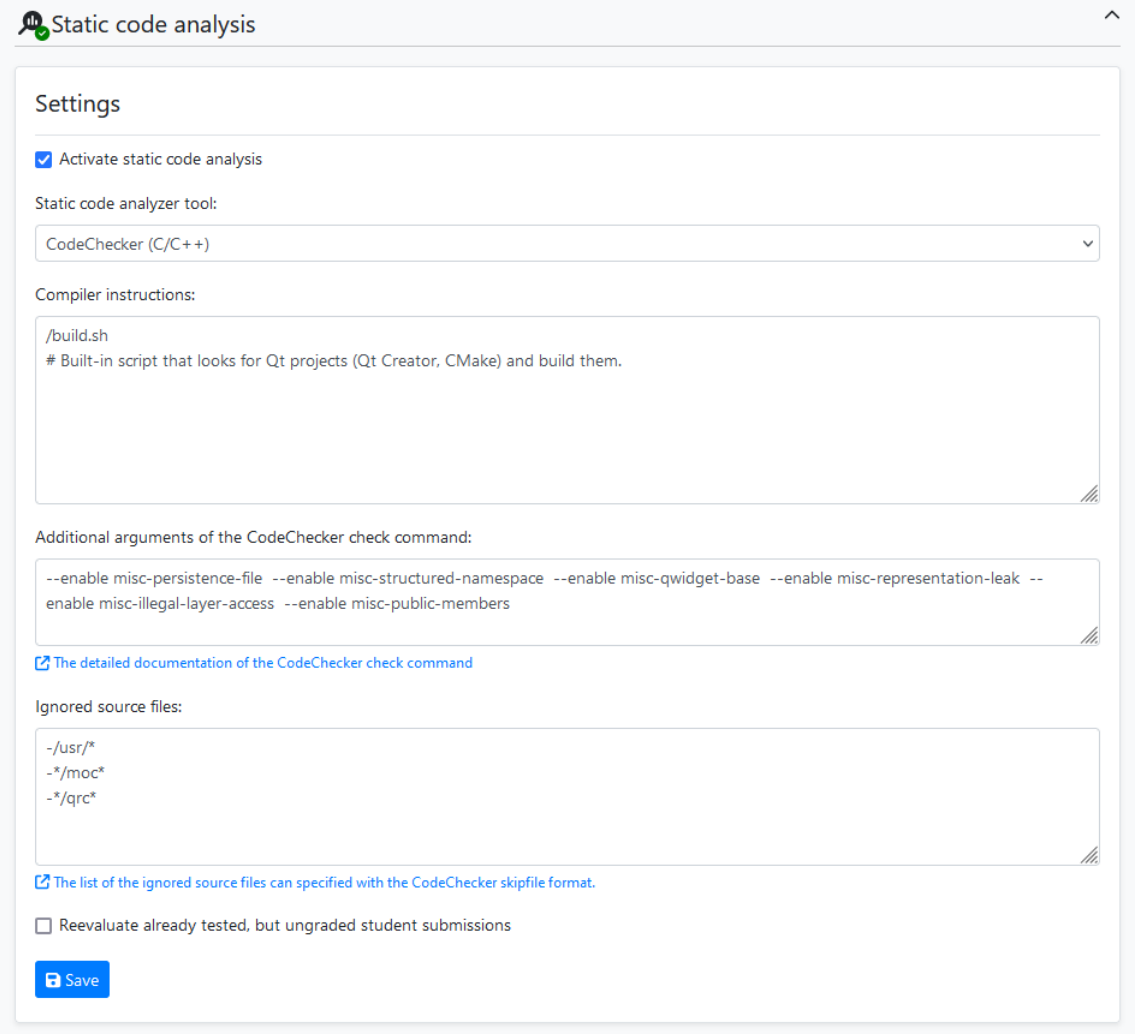

Figure 4.2: Static code analysis settings in TMS

The llvm-project is a very large repository, but to keep the size of the Docker

[2]The merge request of the Docker image is available at https://gitlab.com/tms-elte/docker-images/-/merge_requests/9

image relatively small, the capability of the scripts was used by doing a so-called 2-stage Docker build, where in one – temporary – container the software is built, and then only the binaries of the build result are copied to the second, final image. Later this is used to create containers, where the static code analysis is executed.

After the Linux / Qt5 (architecture) template is loaded in the TMS application, in the settings section for the static code analysis the default parameters are set, as displayed in Figure 4.2. The CodeChecker tool runs 3 static analyzer tools (cppcheck, clang-tidy and clang-static-analyzer), all with a predefined set of checkers. To enable the custom checkers as well, they are directly provided in the corresponding field (additional arguments). The `build.sh` file that is also installed in the Docker image is responsible for building the Qt5 applications that are submitted. If `.pro` files are found, QMake is executed, or as a fallback, in case a `CMakelists.txt` file exists, cmake is used to process the code. If neither of them is present, or the compilation fails, TMS displays the "Analysis Failed" status.

If the compilation is successful, and the tool detected problems, "Issues Found" is displayed at the submission. In this case, all the findings are listed for the submitted assignment. For each one, the exact location of the problem, the name of the checker and the diagnostics message is shown. A few examples can be seen in Figure 4.3.

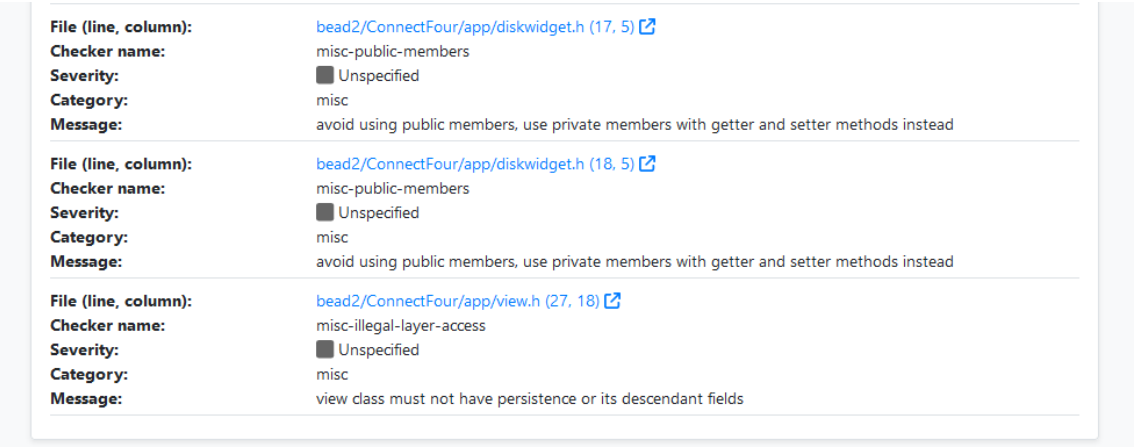

Figure 4.3: Custom static code analysis results in TMS

By clicking on the link of the file, another view is opened, where the error is visualized in the code, allowing the students or teachers to inspect the issue in more details. Figure 4.4 shows an instance of an illegal layer access warning.

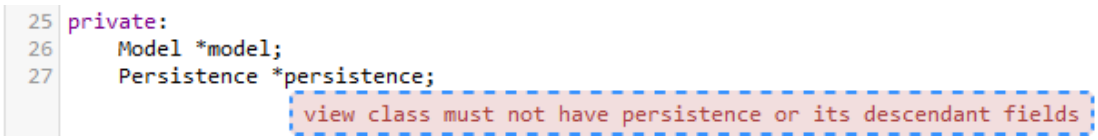

Figure 4.4: Detailed analysis result in TMS

Chapter 5

Conclusion and future work

The different custom clang-tidy checkers that were implemented to check the model-view-persistence architecture of the student assignments in the GUI programming with Qt course detected real architectural programming errors in the codes of the students, proving that it is possible to check certain architectural rules with clang-tidy. The identification of these issues provides information to the lecturers about the most significant weak points in the submitted assignments. Being integrated into TMS, from the next semester both the students and the teachers will be able to see the result of the checkers, allowing the students to improve their work even more and guiding the teachers to the occasionally well-hidden structural errors.

The correct separation and containment of the layers is important to have a structurally correct GUI program. Adhering to the rules in this case does not restrict rather than helps during the implementation. Understanding the relationships between the different components is another area that the checkers identified as a confusing idea for some.

Requiring standard naming of the components on the one hand would guide the students on having all the layers in their programs, and also making them clearer, on the other hand it would ease and speed up the detection of the components for the teachers as well as for the custom checkers. This way less false positive and false negative cases are expected, as the detection of the components would be more stable. If, for example, all persistence components would have in their name the persistence text, it would be clear at first glance, where to look for this component to check its behavior.

The results of the QWidget base checker shed light on the issue, that several

students had trouble understanding this specific concept of Qt, namely that deriving from `QWidget` is only necessary when the class in question is a GUI component. For backend parts, either no or `QObject` base is sufficient, depending of the purpose.

The public members checker is not tightly connected to architecture, rather than identifying a generally bad programming practice. As this checker resulted in the most warnings, and even though they were valid findings, it can make the clang-tidy output superfluous, thus it should be considered whether to include this checker in the analysis.

5.1 Future work

Although in this study the clang-static-analyzer tool is not used, it could be considered in the future. Additional checkers besides the already existing custom clang-tidy checks would detect further possible programming errors, focusing more on the relationships between the components and how they operate together.

A limitation of clang-tidy, being only able to analyze the translation units individually made in impossible to detect the file and directory structure of the project. As seen in the different student assignment projects, many of them used one directory for all files, while others placed the different components into different directories. If clang-static-analyzer is capable of identifying this structure, it could also detect the correct architecture.

As a more advanced approach, if the usage of namespaces would be required for the different components, as they are often used in industrial code bases, the correct structure can be already checked with the structured namespace checker.

As the custom checkers are now integrated into TMS, they will be able to identify issues in the future semesters, further enhancing the quality of the students' assignments.

Bibliography

[1] Arthur-Jozsef Molnar, Simona Motogna, and Cristina Vlad. "Using static analysis tools to assist student project evaluation". In: *Proceedings of the 2nd ACM SIGSOFT International Workshop on Education through Advanced Software Engineering and Artificial Intelligence*. EASEAI 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 7–12. ISBN: 9781450381024. DOI: 10.1145/3412453.3423195. URL: https://doi.org/10.1145/3412453.3423195.

[2] Nathaniel Ayewah et al. "Using Static Analysis to Find Bugs". In: *IEEE Software* 25.5 (2008), pp. 22–29. DOI: 10.1109/MS.2008.130.

[3] Brittany Johnson et al. "Why don't software developers use static analysis tools to find bugs?" In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 672–681. DOI: 10.1109/ICSE.2013.6606613.

[4] *Static program analysis*. URL: https://en.wikipedia.org/wiki/Static_program_analysis (visited on 03/30/2024).

[5] *Can Static Analysis replace Code Reviews?* URL: http://swreflections.blogspot.com/2014/09/can-static-analysis-replace-code-reviews.html (visited on 03/30/2024).

[6] B. Chess and G. McGraw. "Static analysis for security". In: *IEEE Security and Privacy* 2.6 (2004), pp. 76–79. DOI: 10.1109/MSP.2004.111.

[7] *Rice's theorem*. URL: https://en.wikipedia.org/wiki/Rice%27s_theorem (visited on 04/06/2024).

[8] *Halting problem*. URL: https://en.wikipedia.org/wiki/Halting_problem (visited on 04/06/2024).

[9] *The Need for Fourth Generation Static Analysis Tools for Security – From Bugs to Flaws*. URL: https://owasp.org/www-pdf-archive/OWASP-AppSecEU08-Lebanidze.pdf (visited on 03/30/2024).

[10] Gábor Horváth and Norbert Pataki. "Clang matchers for verified usage of the C++ Standard Template Library". In: *Annales Mathematicae et Informaticae*. Vol. 44. 2015, pp. 99–109.

[11] Michael Striewe and Michael Goedicke. "A review of static analysis approaches for programming exercises". In: *International Computer Assisted Assessment Conference*. Springer. 2014, pp. 100–113.

[12] Tomche Delev and Dejan Gjorgjevikj. "Static analysis of source code written by novice programmers". In: *2017 IEEE Global Engineering Education Conference (EDUCON)*. 2017, pp. 825–830. DOI: 10.1109/EDUCON.2017.7942942.

[13] *ELTE Közérdekű, nyilvános adatok*. URL: https://www.elte.hu/kozerdeku (visited on 04/27/2024).

[14] *Extra Clang Tools 19.0.0git documentation*. URL: https://clang.llvm.org/extra/index.html (visited on 04/21/2024).

[15] *Clang-Tidy*. URL: https://clang.llvm.org/extra/clang-tidy/ (visited on 04/28/2024).

[16] *JSON Compilation Database Format Specification*. URL: https://clang.llvm.org/docs/JSONCompilationDatabase.html (visited on 04/19/2024).

[17] *lit - LLVM Integrated Tester*. URL: https://llvm.org/docs/CommandGuide/lit.html (visited on 04/30/2024).

[18] *Abstract syntax tree*. URL: https://en.wikipedia.org/wiki/Abstract_syntax_tree (visited on 04/30/2024).

[19] *Matching the Clang AST*. URL: https://clang.llvm.org/docs/LibASTMatchers.html (visited on 04/30/2024).

[20] *AST Matcher Reference*. URL: https://clang.llvm.org/docs/LibASTMatchersReference.html (visited on 04/30/2024).

[21] *Clang Static Analyzer*. URL: https://clang-analyzer.llvm.org/ (visited on 04/30/2024).

[22] *Checker Developer Manual*. URL: https://clang-analyzer.llvm.org/checker_dev_manual.html (visited on 05/02/2024).

[23] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. "A study of the difficulties of novice programmers". In: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE '05. Caparica, Portugal: Association for Computing Machinery, 2005, pp. 14–18. ISBN: 1595930248. DOI: 10.1145/1067445.1067453. URL: https://doi.org/10.1145/1067445.1067453.

[24] Bence Babati and Norbert Pataki. "Analysis of Include Dependencies in C++ Source Code." In: *FedCSIS (Communication Papers)*. 2017, pp. 149–156.

[25] Bence Babati et al. "Static analysis toolset with Clang". In: *Proceedings of the 10th International Conference on Applied Informatics*. 2017, pp. 23–29.

[26] *Software design pattern*. URL: https://en.wikipedia.org/wiki/Software_design_pattern (visited on 05/02/2024).

[27] *Model–view–controller*. URL: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller (visited on 05/02/2024).

[28] *The llvm-project*. URL: https://github.com/llvm/llvm-project (visited on 04/30/2024).

[29] *Using a Designer UI File in Your C++ Application*. URL: https://doc.qt.io/qt-6/designer-using-a-ui-file.html (visited on 04/19/2024).

[30] P. Kaszab and M. Cserép. "Detecting Programming Flaws in Student Submissions with Static Source Code Analysis". In: *Studia Universitatis Babeș-Bolyai Informatica* 68.1 (2023), pp. 37–54. ISSN: 2065-9601. DOI: 10.24193/subbi.2023.1.03. URL: https://www.cs.ubbcluj.ro/~studia-i/journal/journal/article/view/87.

[31] Péter Kaszab. "Automated evaluation of programming assignments with static code analysis". MSc thesis. ELTE Eötvös Loránd University, Faculty of Informatics, 2023.

[32] *Docker*. URL: https://docs.docker.com/get-started/overview/ (visited on 05/15/2024).

List of Figures

2.1 Number of students at ELTE, Faculty of Informatics 8

3.1 Model-view-controller architecture 26
3.2 Model-view-persistence illegal layer accesses 31
3.3 Processing of a student assignment 36
3.4 Statistics summary 37
3.5 Proportion of errors per checker 38

4.1 Automatic evaluator templates in TMS 44
4.2 Static code analysis settings in TMS 45
4.3 Custom static code analysis results in TMS 46
4.4 Detailed analysis result in TMS 47

List of Codes

2.1 Typical usage of clang-tidy (specify checkers) 12
2.2 Typical usage of clang-tidy (apply fixes) 12
2.3 Output snippet of the modernize-use-using checker 13
2.4 Typical usage of clang-tidy (passing compilation options) 13
2.5 Typical usage of clang-tidy (compilation database) 13
2.6 Typical usage of clang-tidy (headers) 13
2.7 Suppressing clang-tidy warnings 14
2.8 Creating a new clang-tidy checker 14
2.9 Example C++ file 17
2.10 AST of the example C++ file 18
2.11 Bind matcher 19
2.12 Retrieve binded node 19
2.13 Example C++ file 20
3.1 Examples where the QWidget base checker binds matches 29
3.2 Examples where the persistence file checker binds matches 30
3.3 Examples where the representation leak checker binds matches 31
3.4 Examples where the illegal layer access checker binds matches 32
3.5 Examples where the structured namespace checker binds matches . . 32
3.6 Examples where the public members checker binds matches 33
3.7 False positive compiler error without build 35
3.8 QWidget base checker warnings 39
3.9 Persistence file checker warnings 39
3.10 Representation leak checker warnings 40
3.11 Illegal layer access checker warnings 40
3.12 Public members checker warnings 41