TDK-thesis

Péter Kaszab

Automated evaluation of programming assignments with static code analysis

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

DEPT. OF SOFTWARE TECHNOLOGY AND METHODOLOGY

Author:

Péter Kaszab

Computer Science MSc

2. year

Supervisor:

Máté Cserép

Assistant Lecturer

Budapest, 2023

Contents

1 Introduction **3**

2 Technological and literature review **5**

2.1 Professional static analyzers 5

2.1.1 C/C++ analyzers 6

2.1.2 C# analyzers 7

2.2 Static code analyzers in higher education 8

2.2.1 Using professional tools for automatic evaluation 9

2.2.2 Develop custom solutions for the needs of students 10

2.2.3 Teach refactoring and improving existing source code 11

2.2.4 Analyzing existing submissions 11

3 Analyzing solutions from previous semesters **14**

3.1 Used analyzers and evaluated solutions 14

3.2 Results 15

3.2.1 C++ results 15

3.2.2 C# results 21

4 Integration with automated evaluator systems **26**

4.1 Requirements 27

4.1.1 Used analyzer tools 27

4.1.2 Integrate multiple solutions 27

4.1.3 Environment 28

4.1.4 Configuration 28

4.1.5 Presenting the results to the students 29

4.2 Implementation 30

4.2.1 Analyzer tools 30

4.2.2 Processing and storing results 33

4.3 Command-line interface 34
4.4 Graphical web-interface 35
4.4.1 Configuration 35
4.4.2 View the results of the code analysis 38

5 Test the implementation on live data 40
5.1 The reported diagnostics 41
5.1.1 Nullable reference types 43
5.1.2 Violation of conventions 43
5.1.3 Incorrect value is given to a variable or field 43
5.1.4 Loop execution times, conditional branches and recursion 44
5.1.5 Asynchronous operations 46
5.2 Changes in the students' behavior 46
5.2.1 Object-oriented programming 47
5.2.2 Web application development 47
5.2.3 Changes in the average number of uploads 48
5.3 Corrected errors 49
5.3.1 Object-oriented programming 49
5.3.2 Web application development 51
5.4 Feedback from the instructors 52
5.4.1 Attention paid to the reports 52
5.4.2 Clarity and relevance of the reports 53

6 Conslusion 55
6.1 Future work 56

Acknowledgements 57

A Enabled C# diagnostics during the live test of the implementation 58

Bibliography 63

List of Figures 67

List of Tables 68

List of Codes 69

Chapter 1

Introduction

Non-trivial runtime errors in programming submissions are often missed by students, instructors and automatic testers, because these kinds of errors are not always easy to find and reproduce. Furthermore, there are solutions with functionally correct and bug-free code that do not follow the conventions and guidelines of the given programming language. These errors can be avoided and the application of the given guidelines can be forced using static code analyzers.

In my thesis, I evaluated student submission from previous semesters written in C++ and C# by running static analysis on them. Then, I explored how the analyzer tools could be used to automatically evaluate student submissions. Finally, I tested my solution on live data that was collected during the spring semester of the 2022/2023 academic year. I used the open-source *Ericsson CodeChecker* software, which is mainly intended to analyze C/C++ projects, but it can also process the output of analyzers made for other programming languages. For the C# submissions, I used various public analyzer collections built on top of APIs provided by the *Microsoft Roslyn* compiler platform.

Based on the analysis result for the past solutions, I collected the most typical and interesting errors that could have been avoidable if the reported warnings of the tools had been available to the students. I presented them with code examples and detailed description in my thesis. With the usage of the previously mentioned tools, I created a prototype implementation as part of the open-source *TMS (Task Management System)* project, which is a web-based assignment and examination management system developed at *ELTE Faculty of Informatics*. After the implementation was finished, it was enabled for few groups at the faculty. The thesis also presents the reported errors, an analysis of the improvements made by students, and the feedback provided by the instructors from

this experiment.

I concluded that the result of static code analyzers can help students to improve the quality of their programs, and can also help teachers during grading in computer science education. Although, it is important to note the tools should be configured according to the requirements of the course and the skill-level of the students.

In Chapter 2, I introduce static code analysis, the used tools in this thesis, and review the previous applications of static code analysis in computer science education. Chapter 3 contains the description of most typical programming errors from the previous years. In Chapter 4, I outline the requirements for an automatic evaluator system that performs static code analysis on the submissions, and present my prototype implementation of such a system. The results from the trial of the system are presented in Chapter 5. Finally, Chapter 6 concludes the results of this research and sheds light on future research and development directions.

Chapter 2

Technological and literature review

2.1 Professional static analyzers

A *static analyzer* is a tool that check flaws in other programs in compile time without executing them. This can be achieved by evaluating the source code, the byte code or the binaries [1]. These tools can help developers to [2]:

- check styling and formatting problems;
- enforce the conventions and best practices of the given programming languages;
- find programming bugs;
- identify serious security problems;
- problems related to memory management;
- find error-prone code.

Language	Code analyzers
C++	Clang Tidy, Clang Static Analyzer, Cppcheck
C#	Analyzers based on the Microsoft Roslyn Compiler Platform APIs
Java	CheckStyle, Spotbugs, PMD
JavaScript/TypeScript	ESLint
Python	Pylint

Table 2.1: Examples of code analyzers categorized by programming languages

This thesis focuses on *static code analyzers*, that are available for most programming languages as standalone tools, see Table 2.1 for examples. Such analyzers can be categorized based on which techniques they use to detect potential problems [3]:

Pattern matching: it looks for certain constructs on lexical and syntactical levels.

Abstract interpretation: it executes the code on an abstract machine that approximates the original system.

Data-flow analysis: it tracks a set of values and their changes during execution.

Hoare logic: it uses a set of logical rules for reasoning about the correctness of the code.

Model checking: the code is represented as a finite state machine and the analyzer works on that machine.

They are usually configurable according to the needs of the given project (e.g., enabled rules, the reported severity of the problems, etc.). Some tools are extensible with custom rules and provide an API for developers [2]. Modern integrated development environments (IDEs) also include checks and fixes for the most common code-quality problems. This functionality is often implemented with the help of the aforementioned tools [4]. Furthermore, most of the IDEs and code editors can be extended with plugins, so new tools can be added later.

Nowadays, it is also common that code analyzers are included in the CI/CD pipelines alongside with test runners and build tools. In this way, developers get instant feedback about the quality of their code and the output of analyzers can be speed up the code-review process as well. Also, maintainers may require certain checks to pass before they accept a submitted change [5].

In the next two subsections, I present the professional analyzers that will be used in the rest of the thesis.

2.1.1 C/C++ analyzers

2.1.1.1 Cppcheck

Cppcheck [6] is a static analyzer tool that focuses on detecting bugs, dangerous coding constructs, and undefined behaviors. It uses pattern matching and bidirectional flow analysis to achieve this. Also, the developers put huge emphasis on minimizing the number of false positives.

2.1.1.2 Clang Static Analyzer and Clang Tidy

Clang is an open-source *C/C++/Objective C* compiler with modular architecture. Thanks to the modularity, all the functionalities are available as libraries. For instance, the parser of *Clang* can be used to build static code analyzer tools. Many related projects build to improve code quality. This research uses *Clang Static Analyzer* and *Clang Tidy* [3, 7, 8].

Clang Static Analyzer focuses on finding bugs in *C/C++/Objective C* programs with symbolic execution. A unique feature of this tool, that it can visualize the path to the given code segment which causes the potential issue.

Clang Tidy is a C++ linter. It can find programming errors, interface misuses and bugs. It uses AST and preprocessor matchers to examine the source code.

2.1.1.3 CodeChecker

CodeChecker [9] is an analyzer tooling that executes *Cppcheck*, *Clang Static Analyzer* and *Clang Tidy* on C/C++ programs. It supports several advanced features like incremental analysis, HTML and JSON report generation, false positive suppression, etc. A web-based report storage is also available, which helps to review findings in large codebases.

One of the most interesting feature of *CodeChecker* is the *Report Converter Tool* which can convert the output of any supported third-party analyzer tool to a *CodeChecker* report directory. Thus, the features of *CodeChecker* can be used for any programming language. Report Converter currently supports more than 20 code analysis tools, but it is easily extendable using the Python programming language.

2.1.2 C# analyzers

The *.NET Compiler Platform (Roslyn)* provides building tools for the .NET programming languages, such as C#. It is shared between the Microsoft .NET compiler and Visual Studio. It is easily extendable with third-party plugins, as it has well documented and maintained public APIs.

Most importantly, code analyzers can be written as such extension and can be distributed via *NuGet* which is a package manager for .NET [10, 11].

2.1.2.1 Microsoft.CodeAnalysis.NetAnalyzers

Microsoft.CodeAnalysis.NetAnalyzers [12] is a first-party set of analyzers created by Microsoft. It is enabled in all projects by default since .NET 5, but it can be still downloaded from NuGet as well. It can report design, documentation, globalization, interoperability, maintainability, naming, performance, usage, reliability, and security issues.

2.1.2.2 SonarAnalyzer CSharp

SonarAnalyzer CSharp [13] is a NuGet package containing more than 400 rules. It scans the project for bugs, code smells, vulnerabilities, and security hotspots. For certain problems, it provides quick fixes.

2.1.2.3 Roslynator

The *Roslynator* [14] project provides more than 500 Roslyn-based analyzers, code fixes and refactorings. The analyzers can detect issues from the following categories: formatting, style, simplification, readability, usage, maintainability, redundancy, naming, general, design, and performance. Fixes for errors are also available.

The developer also maintains a command-line tool (also referred as *Roslynator.Dotnet.Cli*) that allows to run any Roslyn analyzer (not limited to ones from the *Roslynator* project), fix issues, check styling and generate documentation.

2.2 Static code analyzers in higher education

Static analyzers can be used in education in order to help the learning process of the students and speed up the evaluation of the submission. Michael Striewe and Michael Goedicke [15] reviewed the static analysis approaches that can help in providing feedback to the submitted solutions. They highlighted the following requirements for a system that evaluates submissions with static analysis:

- checking for mistakes and violated coding conventions in syntactically correct source code;
- checking for source code that correct, but contains element that are not allowed in the context of the given subject or exercise;
- checking for missing code structures;

- giving hints on how to solve the previously mentioned issues.

In the following subsections, I review other solutions that utilize static code analyzers in higher education.

2.2.1 Using professional tools for automatic evaluation

2.2.1.1 Hyperstyle

Hyperstyle [4] is a tool that currently supports Java, Kotlin, JavaScript and Python, and it can be extended easily. It uses professional code analyzers (e.g. Pylint for Python), but it adapts them to the need of the students. The possible errors are split into the following categories:

- Code style
- Code complexity
- Error-proneness
- Best practices
- Minor issues

Hyperstyle gives grades for the solutions on a four level scale: excellent, good, moderate, bad. Additionally, it provides custom messages for some issues, because students often need more detailed errors messages than the output of the professional tool. However, the developers kept the original messages for simple issues that belong to the *Code style* and *Best practice* categories.

Hyperstyle currently integrated with *Stepik* and *Jetbrains Academy*, but it can be added to other LMS solutions. The software was evaluated with solutions from the previously mentioned platforms, written in Java and Python languages. The dataset was divided into two parts: before and after *Hyperstyle* was introduced. Their findings were:

- **for Java solutions**, the number of students who made fewer mistakes increased, and the number of who made six or more mistakes decreased;
- **for Python solutions**, the number of students without code quality issues increased four times and the number of students who made two or more errors decreased.

2.2.2 Develop custom solutions for the needs of students

2.2.2.1 FrenchPress

FrenchPress [16] is an Eclipse-plugin that is designed for students with intermediate knowledge of the Java programming language. It focuses on silent flaws that often get overlooked by students, because IDEs and compilers do not catch them. Categories of flaws:

- Misuse of fields (e.g., field used instead of a local variable)
- Misuse of public modifier (e.g., field variables should be private, and the student should write getters and setters)
- Misunderstanding of booleans (e.g., usage of integer instead of boolean, redundant expressions)
- Inappropriate for loop control (e.g., use instance variable as loop variable)

FrenchPress analyzes the AST, uses the Eclipse type hierarchy and does flow control analysis to detect the previously mentioned problems.

The advantage of the IDE integration, that the students can get feedback while they work in their code without leaving the development environment. The authors emphasize that the feedbacks should be relevant for the situation of the students and should be easy to understand. Also, it is important to minimize the number of false positives, as they could be more problematic than false negatives for inexperienced users.

In the end, the percentage of cases when *FrencPress* motivated the users to improve their programs varied from 36% to 64% depending on the course.

2.2.2.2 Formalize rules as logical functions

J. Walker Orr [17] proposes a rule-based tool for Java and Python that provides feedback on predefined rules. The checked design principles are formalized as logical functions, and they are applied to the subtrees of the abstract syntax trees. The models are implemented in Haskell, because of it is suitable for declarative programming. Also, the rules are designed to meet the needs of students. The system was hosted as a standalone web service, students could submit their solutions via an HTML form. There were no limitations to the number of uploads, and the execution of the tests was instant. Thus, this

increased the transparency of the grading progress. On average, the rate of design quality flaws dropped 47.84% on different assignments.

2.2.3 Teach refactoring and improving existing source code

The previously mentioned approaches are evaluating existing solutions or integrated to task management systems and IDEs.

In contrast, *Tutor* [18] is a tool that does not evaluate solutions, but it teaches students to improve existing programs. The users get functionally correct source codes that they have to improve and refactor further. During a tutoring session, the students refactor the programs step by step and get feedback about the correctness of their solutions.

2.2.4 Analyzing existing submissions

While the previously reviewed papers present solutions that provide feedback to students or grade their submissions automatically, analyzing datasets of existing submission can provide valuable information on several aspects. Checking older solutions can help to evaluate the code analyzer tools, and adapt their results to the needs of the students and teachers. Molnar et al. [19] evaluated Python assignments from an introductory programming course using *Pylint*. They also developed a custom tool that is able to analyze and visualize:

Perfomance of a single student over a semester for a given course. Therefore, it is possible to monitor the progress for a given student. Furthermore, reviewing reports for singular or groups of assignments can help teachers to understand common issues for many students.

All students' perfomance for a given assignment. So, it is possible to do a cross-selectional evaluation that targets all student submissions for a particular assignment.

Result from assignments corresponding to multiple students. This approach can be useful, because usually there are multiple seminars and laboratories with multiple instructors for a course, due to the large number of students. Course coordinators can check each group and identify the students' issues in them.

Keuning et al. [20] investigated code quality issues in Java programs. The analyzed source files were collected from the *Blackbox* database, which contains Java solutions

written in the *BlueJ IDE*. The database stores multiple versions of the source files, and also collects events and usage statistics from the *BlueJ IDE* (e.g. enabled plugins). However, the tasks and their requirements are not known. They used the *PMD* tool for analysis and categorized the errors into the following categories: flow, idiom, expression, decomposition, and modularization. Moreover, they included the *Copy/Paste Detector tool (CPD)* from the *PMD* project to detect duplicate code segments. Their findings were:

- Some detectable issues by *PMD* needed to be discarded, because they were too advanced for novice programmers, or they were too specific for a library or platform. Errors from categories like presentation and documentation were completely discarded.
- Empty if statements and singular fields were the most commonly fixed issues. Probably, because the initial uploads were not finished. Issues like too many fields or methods were fixed in less than 5% of the cases. So, the overall fixing rate was relatively low.
- Based on the available statistics about the used plugins, *CheckStyle*, *PMD* and *PatternCoder* were the most popular tools among students. However, these tools had a little effect on the number of issues in the final solutions.

Similarly to the previous paper, Edwards et al. [21] analyzed Java programs from four different courses for students with different skill-levels. The dataset contained nearly 10 million errors produced by 3,691 students. They used the *CheckStyle* and *PMD* open-source tools for static code analysis, but they created their own categories for errors: braces, coding flaws, documentation, excessive coding, formatting, naming, readability, style, and testing. Their finding were:

- The most common categories were documentation, formatting, style, and coding flaws.
- It is possible that some students ignore the warnings produced by the analyzers, if they are dominated by documentation and formatting issues. This factor should be considered when analyzers are used in automatic evaluator systems.
- During submissions, the number of errors were significantly decreased. However, the categories with the highest rates remained the same.

- Excessive coding and missing optional braces were the categories that took the longest time to fix. Although, there were no big differences between the times of the other categories. While documentation and formatting errors were the most common categories, they did not take the most time to fix.
- The coding flaws category could indicate that the student is struggling. Usually, the solutions with lower scores had more coding flaws in them.

Chapter 3

Analyzing solutions from previous semesters

To gain experience with the tool, introduced in Section 2.1.1 and 2.1.2, we tested them on C++ and C# solutions from previous semesters. The experiment showed that professionals analyzers found various programming errors that could have been avoidable if the reported warnings of the tools had been available to the students. The results in this section are already presented in our previous paper, which was recently accepted and will be published soon [22].

3.1 Used analyzers and evaluated solutions

The first batch containing 3433 solutions written in C++ were collected from the *Object-oriented programming* course. While the students have to develop command-line interface applications, they have to manage memory manually and use advanced object-oriented techniques, like polymorphism. In addition, 226 C++ projects were added from the *GUI programming with QT* course, where the participants have to develop complex graphical application with layered architecture (Model-View).

The C++ submissions were analyzed with *Clang-Tidy*, *Clang Static Analyzer* and *Cppcheck*. To run the previously mentioned tools and visualize their results, we have used *Ericsson CodeChecker*.

In the case of C# projects, 2148 programming submissions were collected from the *Event-driven programming* course, where students have to develop Windows Forms, WPF and Xamarin/MAUI graphical applications. Similarly to the *GUI programming with QT*

course, the usage of layered architecture (Model-View and Model-View-ViewModel) is mandatory. For these programs, we have used both first-party (*Microsoft NetAnalyzers*) and third-party (*Roslynator Analyzers* and *SonarAnalyzer CSharp*) analyzers. We performed analysis on the selected submission with *Roslynator.Dotnet.Cli*.

Table 3.1 summarizes the previously described tools, courses, and analyzed submissions.

Language	**Analyzer tools**	**Course**	**Submissions**
C++	Clang Tidy, Clang Static Analyzer, Cppcheck	Object-oriented programming	3433
		GUI programming with QT	226
C#	Microsoft NetAnalyzers, Roslynator Analyzers, SonarAnalyzer CSharp	Event-driven programming	2148

Table 3.1: Summary of the used analyzers and evaluated submissions

From the findings of the analyzers, we have selected the presented errors according to the following criteria:

- We have included the most common and typical errors.
- Some errors only occurred in a handful of submission, but they indicated serious design flaws or lack of understanding.
- We excluded styling errors. While code-styling is important, there were no enforced styling guidelines for the assignments. Also, these rules often require detailed configuration in real-world projects.
- For the C# programs a significant part of findings reported by the Roslyn-based analyzers were possible refactorings, those were also discarded.

3.2 Results

3.2.1 C++ results

In this section, we present the selected errors from the C++ solutions. Figure 3.1 shows the number of solutions from both courses where the those errors occurred.

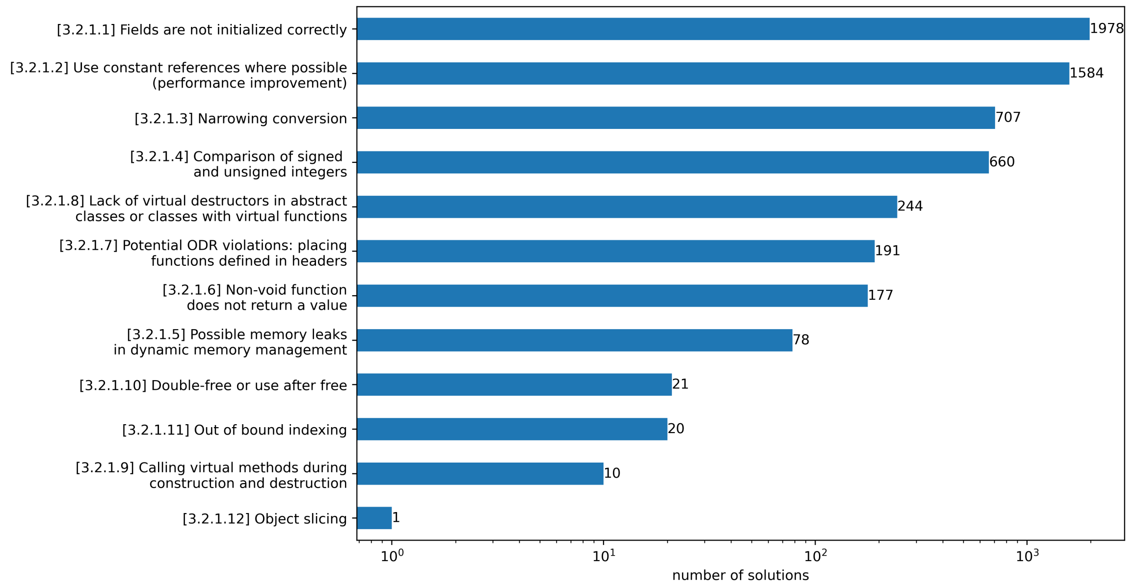


Figure 3.1: The number of C++ solutions with errors

3.2.1.1 Fields are not initialized correctly

Fields are usually not initialized automatically in C++. However, they could be initialized during debug compilation on some platforms, misleading students (as in Code 3.1). It is generally a good practice to give sensible starting values to fields during construction.

```
class Example {
private:
    int _x, _y, _z;
public:
    Example(int x, int y): _x(x), _y(y) {}
    void method() {
        if (_z > 0) { /* ... */ }
        // _z not guaranteed to be initialized to zero
    }
};
```

Code 3.1: Field initialization

3.2.1.2 Use constant references where possible (performance improvement)

While function `f` in Code 3.2 is functionally correct, it has two potential performance problems:

1. the `vec` vector is copied every time `f` is called;
2. the `curr` vector is copied in every iteration.

Using `const` references for the parameter and the loop variable improves performance of this program.

```
void f(vector<vector<int>> vec) {
    for(vector<int> curr : vec) { /* ... */ }
}
void f_improved(const vector<vector<int>>& vec) {
    for(const vector<int>& curr : vec) { /* ... */ }
}
```

Code 3.2: Performance can be improved with references

However, sometimes copying the values of the parameters is the desired behavior. Code analyzers are smart enough to give hints based on the context. So, these warning are only showed if marking the parameter to a `const` reference would not break the given program.

3.2.1.3 Narrowing conversion

Conversion from a wider data type to a narrower can lead to data loss (e.g., `float` →`int`) and/or integer overflow (e.g., `unsigned int` →`int` in Code 3.3).

```
void search(int elem, bool& found, int &ind) {
  found = false;
  for (unsigned int i = 0; i < vec.size() && !found; ++i) {
    if(vec[i] == elem) {
      found = true;
      ind = i; // Could cause integer overflow
    }
  }
}
```

Code 3.3: Possible integer overflow, because of narrowing conversion

3.2.1.4 Comparison of signed and unsigned integers

Direct comparison between signed and unsigned integers is not safe in C++. In most cases this error occurred, when students compared a loop-variable with a size of a container that has `std::size_t` type which is an unsigned integer type (Code 3.4). While this have not caused problems in the submitted solution, it is still considered a bad practice, because `vec.size()` can be greater than the maximum value of `int` on the given platform.

```
1 for (int i = 0; i < vec.size() /* std::size_t */ ; ++i) {}
```

Code 3.4: The maximum size of `int` might be smaller than `vec.size()`

Comparison of signed and unsigned integers could also be problematic if the signed integer value is negative. In Code 3.5, we would expect that it will print `0` as i is not greater than j, but the value of i is also cast to `unsigned int` and it underflows.

```
1 int i = -4;
2 unsigned int j = 5;
3 std::cout << (i > j) << std::endl; // Expected 0, but prints 1
```

Code 3.5: `int i` is casted to `unsigned int`

3.2.1.5 Possible memory leaks in dynamic memory management

Freeing allocated dynamic memory is often missed by students. Consider the `Stack` class in Code 3.6, where the writer of the code allocates memory for the array, but the destructor is missing. Thus, the memory will not be freed after s is not used anymore.

```
1 class Stack {
2 private:
3   int _top, _size;
4   int* _arr;
5 public:
6     Stack(int size): _top(-1), _size(size), _arr(new int[size]) {};
7     // ...
8 };
```

Code 3.6: Memory leak: missing destructor

3.2.1.6 Non-void functions does not return a value

Reaching the end of the body of a non-void function without returning a value is will not generate a compiler error by default, but it is an undefined behavior in C++. A good example of this, a stack class where the pop method of a stack that removes the item from the top of the stack and returns its value. The implementation in Code 3.7 of the pop method is error-prone, because the user of the class can call the method on an empty stack.

```
int Stack::pop() {
    if (!isEmpty()) {
        return _vec[--_top];
    }
}
```

Code 3.7: Empty stacks are not handled

3.2.1.7 Potential ODR violations: placing function in headers

One Definition Rule (ODR) means that non-inline functions and types must have only one definition in the entire program [23]. For instance, placing functions in headers can lead to ODR violations. This does not necessarily mean that the solution does not compile or run until it is only included in one source file. However, if the student had included it in two or more sources, the compiler would not have accepted the solution.

Consider the scenario illustrated in Code 3.8, while

- the `g++ main.cpp first.cpp` command will compile the program successfully;
- the `g++ main.cpp first.cpp second.cpp` command will fail.

```
/// Contents of helpers.h:
int square(int x) { return x * x; }

/// Contents of first.cpp:
#include "helpers.h"
void first_calculation() { int res = square(2); /* ... */ }

/// Contents second.cpp:
#include "helpers.h"
void second_calculation() { int res = square(3); /* ... */ }

/// Contents main.cpp:
// helpers.h is not included in main.cpp
```

Code 3.8: Functions in headers

3.2.1.8 Lack of virtual destructors in abstract classes or classes with virtual functions

It is possible that the student implemented all necessary destructors, but they are not marked as virtual when needed. In Code 3.9, if the destructor of `Base` is not marked as

virtual and `delete` is called on a variable with static type of `Base`, then the destructor of `Derived` will not be called.

```
struct Base {
    virtual void method() = 0;
    ~Base() {std::cout << "base "; }  // Should be virtual
};
struct Derived: public Base {
    void method() override {  }
    ~Derived() { std::cout << "derived "; }
};
void f() {
    Base* d = new Derived;
    delete d; // outputs: base
}
```

Code 3.9: Destructors should be `virtual`

3.2.1.9 Calling virtual methods during construction and destruction

During construction and destruction, the virtual call mechanism is disabled. Therefore, the implementation from the current class is used, as illustrated in Code 3.10 with the call of `f`. Calling virtual methods in the constructor is not necessarily a problem, but the student might not aware of the previously described behavior.

```
struct Base {
  Base() { f(); } // Prints base
  virtual void f() { std::cout << "base"; }
};
struct Derived: public Base {
  void f() override { std::cout << "derived"; }
};
```

Code 3.10: Virtual calls in constructors

3.2.1.10 Double-free and use after free

In C++ `delete` should be called only once for the same reference and the reference should not be used after `delete` called on it. Code 3.11 counts not only as double-free, but an infinite loop, because `~Example` will always get called again, recursively. This is a good example of how reported errors can also indicate lack of understanding from students.

```
struct Example {
    ~Example() { delete this; }
};
```

Code 3.11: Incorrect usage of `delete`

3.2.1.11 Out of bound indexing

Out of bound indexing is often missed by beginner programmers. It usually results in a *memory segmentation fault*. The provided example (Code 3.12) is relatively simple: the student made a small mistake and wrote <= instead of <. Fortunately, the used code analyzers can spot possible out of bound indexing in more complex scenarios.

```
int arr[10];
for (int i = 0; i <= 10; ++i) { arr[i] = i; }
```

Code 3.12: Out of bound indexing

3.2.1.12 Object slicing

Slicing happens when copying a derived object into a base object: the members of the derived object (both member variables and virtual member functions) will be discarded [24]. In Code 3.13, slicing object from type `Derived` to `Base` discards override `method`.

```
struct Base {
    virtual void method() { std::cout << "base"; }
};
struct Derived: public Base {
    virtual void method() { std::cout << "derived"; }
};
void f(Base obj) { obj.method(); }
int main() {
    Derived d;
    f(d); // prints base
    return 0;
}
```

Code 3.13: Object slicing

3.2.2 C# results

In this section, we present the selected errors from the C# solution. Figure 3.2 shows the number of solutions from both courses where the those errors occurred, categorized by

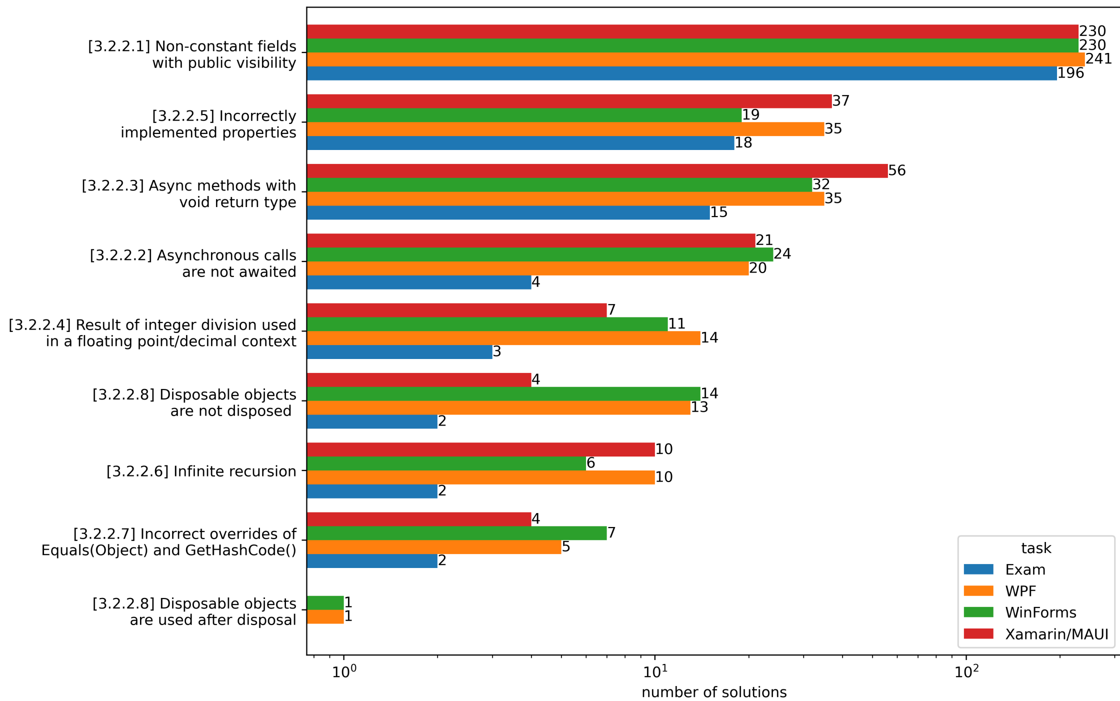


Figure 3.2: The number of C# solutions with errors

tasks. It is worth to note that the number of solutions containing the highlighted errors are really similar for the WinForms, WPF, and Xamarin/MAUI assignments. This is because the students have to develop the same software for all three tasks, and they are encouraged to reuse layers from their previous solutions. Exams are different, because student have to develop new applications from scratch, but reusing their existing materials is still allowed.

3.2.2.1 Non-constant fields with public visibility

Using public mutable fields are generally considered a bad practice and against guidelines in C#. There are several alternatives:

- mark the field `readonly` or `const`;
- use auto-implemented properties instead;
- make it private and access it with a property or method.

```
public class ClassName {
    public int Field;
}
```

Code 3.14: `Field` should not be public

3.2.2.2 Asynchronous calls are not awaited

In Code 3.15, `NewGameAsync` is an async function, but it is not awaited. Thus, the state of the `model` object might be incorrect when `AdvanceGame` is called.

```
1 GameModel model = new GameModel();
2 model.NewGameAsync();
3 model.AdvanceGame();
```

Code 3.15: `model.NewGameAsync()` is not awaited

3.2.2.3 Asynchronous methods with void return type

Asynchronous function should return `Task` or `Task<T>`, because they cannot be awaited and exceptions cannot be caught from them (Code 3.16). Event handlers are the only exceptions according to the Microsoft Learn guidelines, because they usually have to return `void` [25].

```
1 public async void LoadAsync(string filePath) {
2     FileContent = await File.ReadAllTextAsync(filePath);
3 }
```

Code 3.16: `LoadAsync` cannot be awaited

3.2.2.4 Results of integer division should not be assigned to floating point/decimal variables/parameters

In Code 3.17, if the result of `size / 2` is positive, then it is already floored because of the integer division. Therefore, calling `Ceiling` will not return the expected result.

```
1 int size = // ...
2 if ((int)decimal.Ceiling(size / 2) == x) { /* ... */ }
```

Code 3.17: Integer disivision

3.2.2.5 Incorrectly implemented properties

The getters and setters of the properties should access the correct backing fields. As shown in Code 3.18, the student may want to write a read-only property, but the setter is still present with an empty body. The correct solution would be a property without a

setter, because assigning a value to a property will not generate a compile-time error and the user may think that the property is writable. In contrast, if the setter is not present, then both the compiler and the IDE will show an error upon assignment.

```
1 private int property;
2 public int Property {
3     get { return property; }
4     set {}
5 }
```

Code 3.18: Read-only property: `set` should be omitted

3.2.2.6 Infinite recursion

A trivial example of this error, when the setter tries to assign the value to the property itself (Code 3.19). This may be the result of a typo in the source code, as backing fields often have the same name as the property, but with a different case. Calling the setter of such a property would lead to an infinite recursion.

```
1 private int property;
2 public int Property {
3     get { return property; }
4     set { Property = value; }
5 }
```

Code 3.19: Incorrectly implemented setter: infinite recursion

3.2.2.7 Incorrect overrides of Equals(object) and GetHashCode()

When overriding `Equals(object)` and `GetHashCode()` certain rules should be followed, such as:

- `Equals(object)` and `GetHashCode()` should be overridden in pairs.
- Classes directly extending `object` should not call `base` in `GetHashCode` or `Equals`. The implementation in `object` are based on object reference.

In Code 3.20, the student overrides both methods, but the `GetHashCode` calls the implementation from the `object` class.

```
class ClassName {
    public override int Equals() {
        // correct implementation
    }
    public override int GetHashCode() {
        base.GetHashCode(); // Calls GetHashCode from object
    }
}
```

Code 3.20: Incorrect override of `GetHashCode()`

3.2.2.8 Mistakes related to disposable objects

While C# has automatic garbage collector, the unmanaged resources taken by certain classes should be freed. For instance (Code 3.21), if a file opened by the `StreamReader` class, then it should be closed after usage. The `Dispose` should be called (or `Close` in this case), preferably in a `finally` block. A `using` statement or declaration would be an even better option, as it ensures the correct usage of disposable objects.

```
List<string> values = new List<string>();
StreamWriter sw = new StreamWriter("output.txt");
foreach (string line in values) {
    sw.WriteLine(line);
}
```

Code 3.21: The file is not closed after usage

It is also important that the objects cannot be used after they are disposed. In the example in Code 3.22, the `Dispose` method is automatically called after the execution leaves the block of the using statement. So, the returned `StreamReader` instance will not be able to read the file, as its methods will throw an `ObjectDisposedException`.

```
public StreamReader CreateReader(string filename) {
    using (StreamReader sr = new StreamReader(filename)) {
        return sr;
    }
}
```

Code 3.22: `sr` is returned after disposal

Chapter 4

Integration with automated evaluator systems

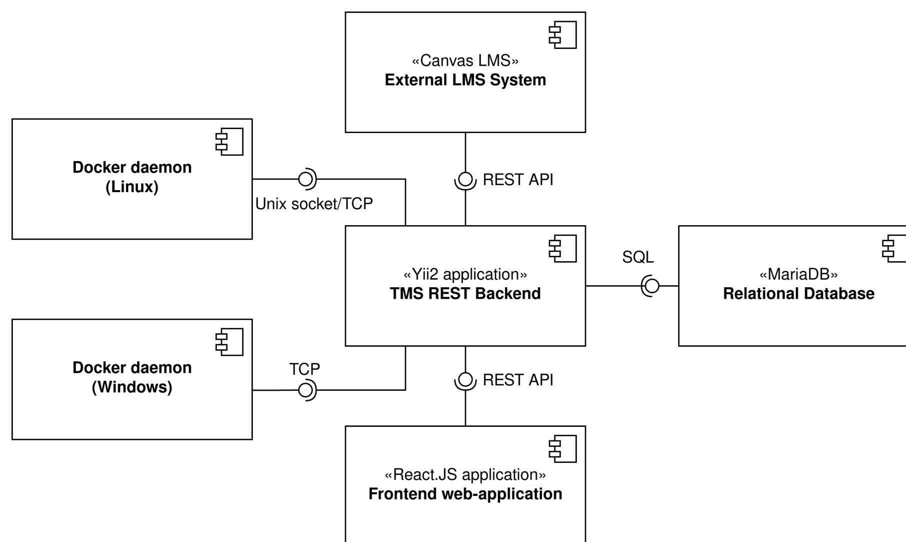


Figure 4.1: The relevant components of TMS

Based on the findings presented in Chapter 3, I believe that integrating static code analyzers to an automatic evaluator system would improve the quality of the student submissions.

I implemented this solution as part of the open-source *TMS* task management system developed at our university [26], which already contains a custom developed Docker-based automated evaluator and integrates the static analysis tool *CodeCompass* [27], but for code comprehension purposes. TMS can synchronize with the instances of the Canvas LMS to complement its functionality. Figure 4.1 illustrates the components of TMS relevant to this research.

The repositories of TMS are hosted and publicly available at GitLab: `https://gitlab.com/tms-elte`. The version presented in this thesis is tagged in all relevant repositories:

- **Yii2 REST Backend:** `https://gitlab.com/tms-elte/backend-core/-/tree/thesis/tdk-evaluator-static-analysis`
- **React.JS Frontend:** `https://gitlab.com/tms-elte/frontend-react/-/tree/thesis/tdk-evaluator-static-analysis`
- **Docker images:** `https://gitlab.com/tms-elte/docker-images/-/tree/thesis/tdk-evaluator-static-analysis`

4.1 Requirements

Before integrating automatic static code analysis capabilities into TMS, I outlined the requirements based on the reviewed technologies and literature (Chapter 2), the existing functionality of TMS, and the needs of the courses taught at *ELTE Faculty of Informatics*.

4.1.1 Used analyzer tools

In Section 2.2, I reviewed papers that utilize both existing professional and custom developed analyzers for teaching. While custom analyzers can be adapted more to the needs of the students, they require a lot more work to implement and maintaining them. This factor should be considered, especially in university projects. Furthermore, adding support for a new programming language may require a complete reimplementation of the tool if it is too dependent on the toolset of the given language.

Using professional code analyzers solves the previously mentioned problems. They are usually well-tested and already contain hundreds of checks for various programming errors. However, adapting them to the needs of the students might be challenging. In the following sections, this thesis will focus on professional analyzers that were described in Section 2.1.1 and 2.1.2.

4.1.2 Integrate multiple solutions

Instructors should be able to select from multiple analyzer solutions, and adding new tools should be easy, because many programming languages are taught in higher computer

science education. To achieve this, the integration should provide an extendable converter tool that takes the output of the various code analyzers and produces reports in a uniform format. Thus, it is enough if the LMS solution handles only this format.

4.1.3 Environment

Analyzer tools often require several dependencies (e.g, compilers) to be installed. The following alternatives were considered [28]:

Install analyzers and their dependencies to the host machine. This options has the least overhead. However, maintaining the installed tools and their dependencies can be hard, and it makes the evaluator system less portable.

Use virtual machines. Using virtual machines solves the problem of installing the right version of the dependencies, as images can be built and distributed for the different use cases. Virtual machines can also run a different OS than their host. Furthermore, they provide isolation from the host machine, and they can be reset that can be beneficial, when processing files from untrusted sources. However, such solutions demand more system resources and increase complexity.

Use Docker containers. With containers, applications can be packaged with their runtime environment. The overhead of running Docker containers is minimal compared to the virtual machines, because the applications in them usually run on the host OS [1]. As a trade-off, they provide less isolation compared to virtual machines. Docker containers can be a good compromise between installing the tools directly on the host machine and using virtual machines. With Docker, the containers can be created from images, that can be easily defined with Dockerfiles and distributed through Docker Hub or other registries.

4.1.4 Configuration

Analyzers should be configurable from the user interface, so instructors can adapt them to the needs of their students. The most important option is to disable or enable certain checks. Excluding some files from the analysis is also important, as certain project types have automatically generated files on them.

[1] Using a hypervisor might be still needed if the container based on a different OS than the host or the user wants a higher degree of isolation.

Apart from the detailed a configuration, a quick setup options should be also provided with sensible defaults. It can also be useful, if configurations can be shared between courses, so instructors can share them with each other. A configuration template system can solve the aforementioned problems.

4.1.5 Presenting the results to the students

Adapting professional analyzers to the needs of students was a recurring topic in the reviewed papers in Related Work.

Only show relevant findings. The majority of the reviewed papers in Section 2.2 concluded that, the results should be filterable, as not all reports indicate real problems or relevant to the given course or skill level. The same conclusion was drawn based on the results presented in Chapter 3, as the majority of the warnings reported by the used tools had to be discarded.

Remove duplicates. When using multiple analyzers, it is possible that more than one tools report the same problem. Those reports should be removed to reduce noise. A manual review of the available analyzers is needed to achieve this.

Prioritize findings. Prioritize the problems can help students to decide what parts of their solutions should be fixed first. Analyzers often report the severity and category of the given problem, these properties can be used when presenting the solutions to the students. Customizing these values can also be considered to meet the needs of the given course.

Provide additional information. Sometimes the messages attached to the findings can be too brief for beginner programmers. The documentation of the tools often contain more detailed description about the reported problem. Adding custom description and additional hints can be considered as well. However, maintaining the hints would require additional work from the instructors.

Mark the problematic code segments clearly. It must be clear where was the problem found in the uploaded codebase. It can be useful if students and instructors can view the problematic code segment without leaving the webpage of the LMS solution.

4.2 Implementation

I built my solution on top of *CodeChecker* (introduced in Section 2.1.1.3), because it provides advanced features for C/C++ projects and its output can be processed and stored easily in TMS. Furthermore, with the help of its *Report Converter Tool* the output of more than 20 analyzer tools can be converted to the same format. Thus, it is enough if TMS can process the output of *CodeChecker*.

The static analysis happens in Docker containers, it is integrated to the existing evaluator system of TMS. The automatic static code analysis uses the same environment as the tester components of the evaluator, so instructors who are already familiar with this system can easily enable the analysis for their courses.

The system regularly checks for new submissions from tasks with valid evaluator configuration. When there is a new untested solution, the system run the selected tools in Docker containers and converts the reports to a common report format if necessary. Finally, when the reports are available in the required format, TMS persists them for the given solution, and notifies the student about the completion of the analysis [22]. Figure 4.2 shows the previously described high-level workflow of the automated static analysis.

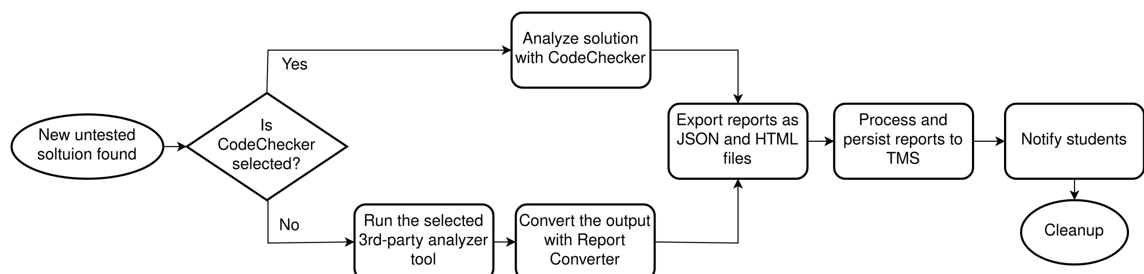


Figure 4.2: The high-level workflow of static analysis in TMS

4.2.1 Analyzer tools

As shown in Figure 4.2 the analysis workflow depends on the selected analyzer tool. The two cases differ in the number of started Docker containers, used tools and Docker images.

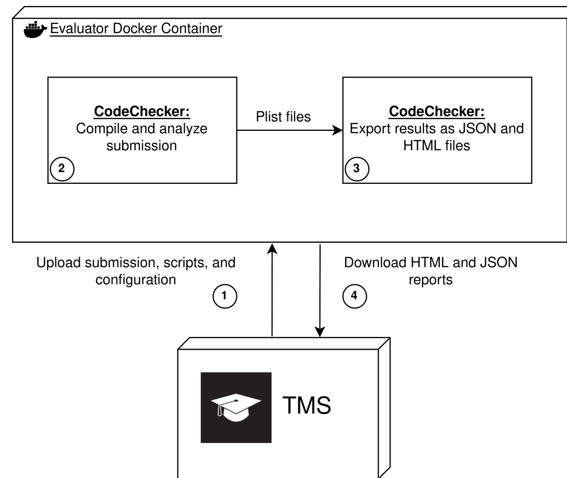


Figure 4.3: The CodeChecker workflow

CodeChecker (C/C++): TMS starts one Docker container based on the image configured for the given task on the evaluator environment page. This image must contain *CodeChecker*. After the container started, TMS uploads the submission, test files, configuration files and scripts to it. It runs *CodeChecker* on the source code, which produces a report directory containing Plist files. Then, the results are exported as static HTML files and a JSON file. These files are downloaded by TMS for further processing, and the container can be stopped. The process is illustrated in Figure 4.3.

Report Converter Tool: I decided to split the necessary tool in two different Docker images, as seen in Figure 4.4. The first container is configured based on the Docker image configured for the given tasks, the selected analyzer tool must be installed to it. After the necessary files are uploaded, the selected tool analyzes the submission. Then, TMS downloads the results and stops the container.

The results are uploaded alongside with the submission to a second container that only contains a minimal *CodeChecker* installation without the compiler toolset. This container is based on a preconfigured image. The *Report Converter Tool* produces a *CodeChecker* report directory, finally the results are exported in the same HTML and JSON format that can be processed by TMS.

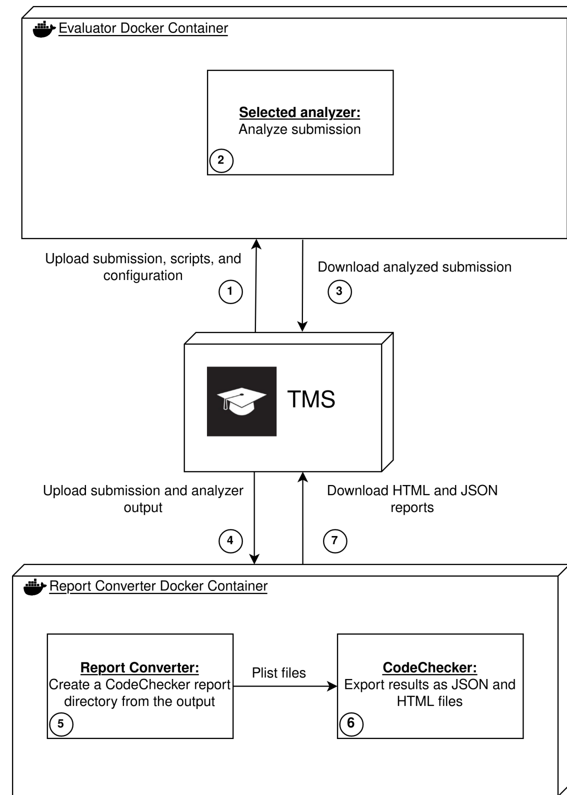


Figure 4.4: The Report Converter Tool workflow

While there is a slight overhead of using two different containers, this makes configuration easier for instructors. It also reduces the image size and complexity of the configured evaluator Docker image.

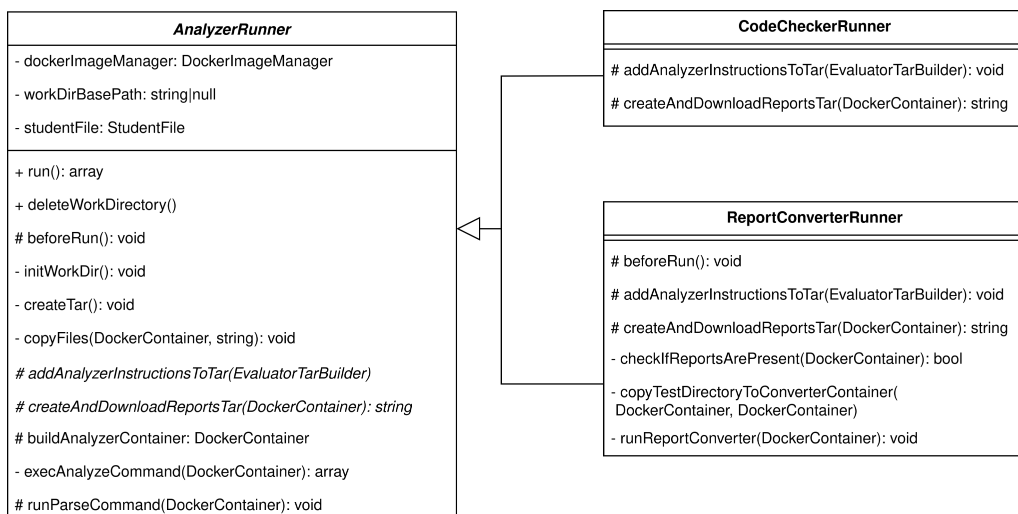


Figure 4.5: UML class diagram of the analyzer runners

Figure 4.5 shows the classes that implement the aforementioned workflows. Both workflows are implemented in the descendants of the `AnalyzerRunner` class, which con-

tains the common logic for analyzer runners. Since the workflows have really similar steps, most of the steps can be shared between them and only the differences must be implemented.

4.2.2 Processing and storing results

The reports from the CodeChecker JSON output file are processed and saved to the relational database used by TMS. The static HTML files are persisted to the disk, and can be viewed from the web interface of TMS.

4.2.2.1 Database

Figure 4.6 shows the entity relation of the tables where the results and configuration stored. The overall database structure of TMS is much more complex, so only the relevant tables are shown on the diagram.

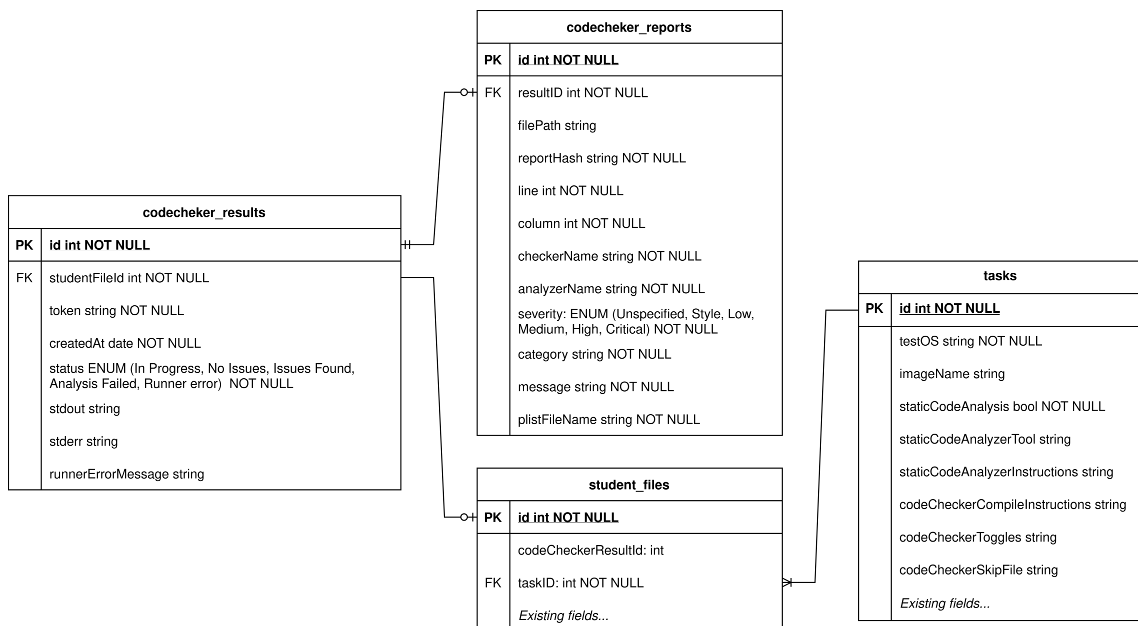


Figure 4.6: Entity relation diagram of the relevant tables

The evaluator settings (Docker Image settings, Docker OS, configuration files and scripts) for the given task are stored directly in the `tasks` table. For each task, the submissions uploaded by students are stored in the `student_files` table. The current implementation of TMS only stores the latest version of the submission for each student and task pair by default. In the `student_files` table, the `codeCheckerResultID` points to the results of the latest code analysis.

The details of the current and previous analysis runs are stored in the `codechecker_results` table. While the users can access the latest results only, all of them are persisted to help this research and future work. For the given run, the `codechecker_reports` table contains findings reported by the analyzer tools. The columns of this table are directly mappable from the JSON output of *CodeChecker*.

4.2.2.2 Persistence layer and notification sender

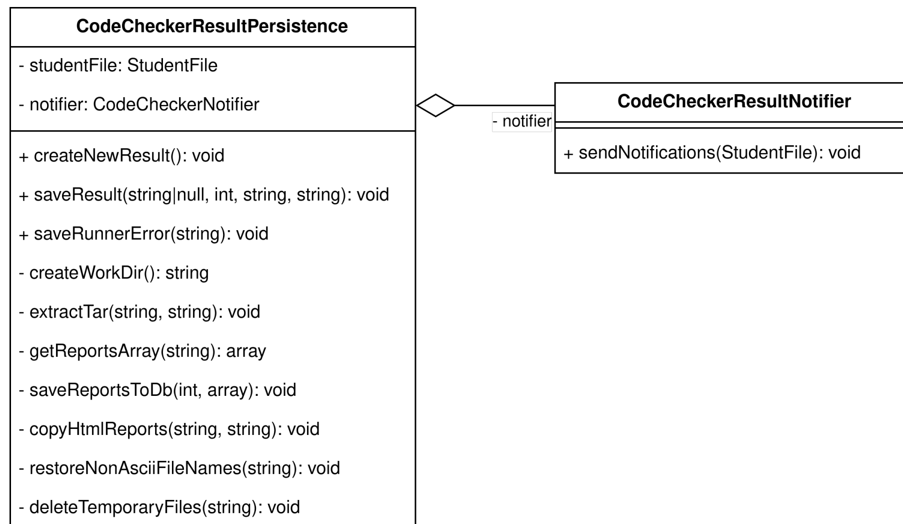


Figure 4.7: UML class diagram of the persistence layer

The responsibility of the `CodeCheckerResultPersistence` class is processing the results and managing the database (illustrated in Figure 4.7). It can create an empty result and attach it to the student file with the `createNewResult` method. After the analysis is run, the `saveResult` processes its result, saves the reports, and updates the status of the result. If there were errors related to TMS during the process, `saveRunnerError` can be used to update the status of the result store the error message for the users. When the status of a result changes, the `notify` method of `CodeCheckerResultNotifier` is called to send emails to the students and attach a comment to the Canvas submission if necessary.

4.3 Command-line interface

The static analysis can be initiated from the command-line interface of the backend, with the `yii code-checker/check <n>` command, where *n* is the maximum number of solutions to analyze. The command searches the submission without result, creates an

empty result for the submission, retrieves an instance of one of the descendant classes based on the evaluator configuration, runs the analysis, then calls the correct method of the persistence class to save the results. The benefit of this approach, that analysis can be easily run manually if needed, and it can be easily scheduled with any task scheduler solution.

The Docker images required for the Report Converter workflow can also be downloaded and updated from the command-line interface with the `yii code-checker/pull-report-converter-image (linux|windows)` command.

4.4 Graphical web-interface

The graphical interface of TMS is React.JS-based web application, written in TypeScript, which communicates through a REST API with the backend. This section provides a brief overview of the new graphical interface of the evaluator extended with the static code analysis capabilities.

4.4.1 Configuration

The instructors can configure the evaluator on the page of the given task by clicking on the *Automatic Evaluator* tab. The configuration page is divided into three sections, as seen in Figure 4.8.

Predefined configurations can be loaded from templates. Icons next to the template names indicate which templates modules (testing and static code analysis) will be active after the application.

4.4.1.1 Environment

Both the automatic tester and the static analyzers run in the configured Docker container. The user can set an image from Docker Hub or upload a custom Dockerfile. Additional files can also be uploaded, which will be added to the container before running the tests and analyzers. These files can be used for providing additional configuration or defining more complex test cases.

Figure 4.8: Overview of the evaluator configuration user interface

After the environment is configured, the instructor can turn on or off *Automatic tester* and *Static code analysis* independently.

4.4.1.2 Automatic tester

The automatic tester was already part of TMS before the static code analysis functionality was added, but the user interface was redesigned, in order to share the same Docker environment between the tester and static code analyzer.

To enable testing, the instructor must select the type of the application (console or web), and set the set compiler and run instructions. Furthermore, it can be customized whether students can see detailed error messages and existing solutions should be reeval-

uated. Finally, if run instructions are provided, then tests cases can be added by entering the arguments, input, and the expected output.

4.4.1.3 Static code analysis

Figure 4.9: Static code analysis settings with CodeChecker selected

First of all, the instructor must select the analyzer tool for the task. The required fields are different based on the selected option:

CodeChecker In this case, CodeChecker runs the static analysis with *Clang Tidy*, *Clang Static Analyzer* and *CppCheck*. Compiling the solution is required before, thus the instructor must provide the *compiler instruction*. It is also possible to exclude certain files from the analysis, the list of these files can be specified with the CodeChecker skipfile[2] format. Figure 4.9 shows an example configuration for *CodeChecker*.

[2] https://codechecker.readthedocs.io/en/latest/analyzer/user_guide/#skip

A 3rd party tool supported by CodeChecker Report Converter If the instructor selected a tool that requires report conversion, then a Bash (Linux) or Powershell (Windows) script must be provided, that runs the selected tool and places the analyzer output to the required location in the container. The exit code of the script is also considered when the evaluator tries to determine whether the analysis was successful. So, a non-zero exit must be returned if the tool found issues in the uploaded solution or the analysis failed. The skipfile format can also be used here to exclude files from the results. At the moment of writing this thesis, not all third-party tools supported by *CodeChecker* are available from the user interface.

4.4.2 View the results of the code analysis

Static code analysis reports

File (line, column):	stack.cpp (8, 1)
Checker name:	clang-diagnostic-return-type
Severity:	High
Category:	clang
Message:	non-void function does not return a value in all control paths
File (line, column):	stack.h (9, 46)
Checker name:	cppcheck-noDestructor
Severity:	Medium
Category:	warning
Message:	Class 'Stack' does not have a destructor which is recommended since it has dynamic memory/resource allocation(s).
File (line, column):	helpers.h (3, 5)
Checker name:	misc-definitions-in-headers
Severity:	Medium
Category:	misc
Message:	function 'square' defined in a header file; function definitions in header files can lead to ODR violations

Figure 4.10: List of reports

The results are displayed on the page of the given task on the student user interface, The list of reports ordered by severity: the more serious the issue, the higher it is in the list. Moreover, color codes also indicate the severity of the given issue (Figure 4.10).

The HTML report viewer (Figure 4.11) can be accessed from this list by clicking on the name of the files where the problem was reported. Thus, users can view the problems without downloading the solution. For certain analyzers, *CodeChecker* includes help links in the HTML reports. At the moment of writing this thesis, this feature was not configured for the Roslyn-based analyzers.

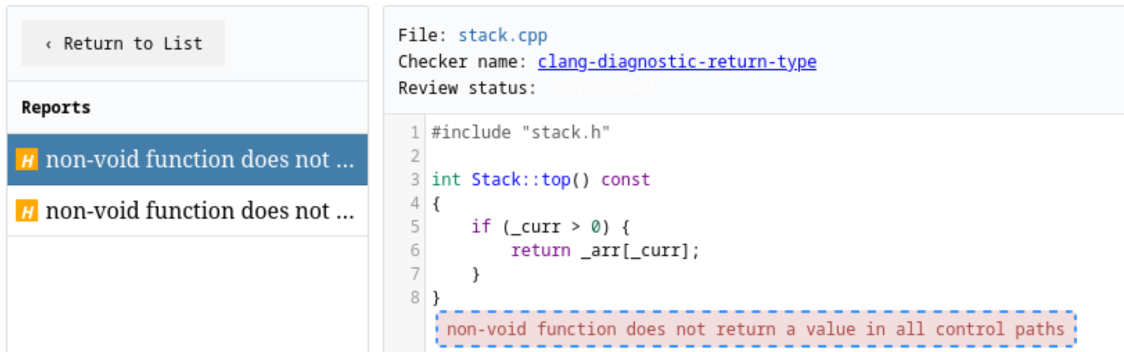


Figure 4.11: CodeChecker HTML reports

For instructors, the status of the analysis is displayed for each submission on the *Solutions* tab on the *Group* page. By clicking on the status, the reports are listed on a similar interface to the student task page. In addition to the results, instructors can also access the standard output and error messages produced by the analyzer tools, so they can debug the configuration if needed.

Chapter 5

Test the implementation on live data

To determine the efficiency of the implementation, static code analysis was enabled for *eight* groups of the *Object-oriented programming (OOP)* and *three* groups of the *Web application development (WEB)* course in the spring semester of the 2022/2023 academic year. We examined the reports which were created during this semester and collected feedback from instructors. Table 5.1 shows the number of solutions for both courses, grouped by assignments and exams, from **2023-03-30** to **2023-05-12**. We counted the submissions and their constituent uploads separately, so we could examine the improvements in the submissions. Uploads with invalid file formats, or failed status due to configuration errors, were excluded from the dataset.

Categories	OOP		WEB	
	Submissions	**Uploads**	**Submissions**	**Uploads**
Assignments	676	1101	17	45
Exams	137	303	-	-

Table 5.1: Number of submissions and uploads for each course and category

The *Object-oriented programming* programming course has been already introduced in Chapter 3: the students have to develop command-line application while using advanced object-oriented techniques such as polymorphism. Although, for the first time, this course is taught in C# from this semester.

The *Web application development course* course is also taught in C#. The students have to develop complex web applications using the ASP.NET Core framework. The assigned tasks range from MVC-based web applications to WebAPIs with WPF-based clients. This course is not mandatory, thus it has a lower number of students compared to

Object-oriented programming. It is also important to note that this course only has one exam at the end of semester, which has not yet written at the time of writing this thesis.

For both courses, we configured the *Roslynator.Dotnet.CLI* tool with the analyzer Nuget packages introduced in Section 2.1.2: *Microsoft NetAnalyzers*, *Roslynator Analyzers*, and *SonarAnalyzer CSharp*. We manually reviewed diagnostics provided by these tools, because enabling all of them would result in too much noise for the students. Furthermore, we included the warnings produced by the compiler. The complete list of active diagnostics and compiler warnings can be found in Appendix A.

5.1 The reported diagnostics

ID	Description	OOP	WEB
S1104	Fields should not have public accessibility	165	3
CS8600	Converting null literal or possible null value to non-nullable type.	98	1
CS8625	Cannot convert null literal to non-nullable reference type.	97	0
CA1001	Types that own disposable fields should be disposable	62	0
CS8603	Possible null reference return.	60	0
CS8618	Non-nullable field must contain a non-null value when exiting constructor. Consider declaring as nullable.	56	4
S2930	"IDisposables" should be disposed	46	0
S101	Types should be named in PascalCase	41	3
CS8602	Dereference of a possibly null reference.	37	5
S1643	Strings should not be concatenated using '+' in a loop	33	1
CS8601	Possible null reference assignment.	25	0
S2583	Conditionally executed code should be reachable	24	0
S3903	Types should be defined in named namespaces	22	0
CS8604	Possible null reference argument.	18	2
CA1710	Identifiers should have correct suffix	18	0
S3887	Mutable, non-private fields should not be "readonly"	14	0
CA1050	Declare types in namespaces	13	0
CS0649	Field is never assigned to, and will always have its default value	9	0
CS0436	Type conflicts with imported type	8	0
S1751	Loops with at most one iteration should be refactored	7	0
S2184	Results of integer division should not be assigned to floating point variables	7	0
CS0108	Member hides inherited member; missing new keyword	3	0
CS0162	Unreachable code detected	3	0

ID	Description	OOP	WEB
S1871	Two branches in a conditional structure should not have exactly the same implementation	3	0
S2259	Null pointers should not be dereferenced	3	0
S2201	Return values from functions without side effects should not be ignored	2	1
S1764	Identical expressions should not be used on both sides of a binary operator	2	0
S2190	Recursion should not be infinite	2	0
CS8619	Nullability of reference types in value doesn't match target type.	1	2
CS0109	Member does not hide an inherited member; new keyword is not required	1	0
CS0114	Member hides inherited member; missing override keyword	1	0
CS0642	Possible mistaken empty statement	1	0
CS0665	Assignment in conditional expression is always constant	1	0
CS1717	Assignment made to same variable	1	0
CS8629	Nullable value type may be null.	1	0
S1656	Variables should not be self-assigned	1	0
S1848	Objects should not be created to be dropped immediately without being used	1	0
S3400	Methods should not return constants	1	0
S3923	All branches in a conditional structure should not have exactly the same implementation	1	0
CA1711	Identifiers should not have incorrect suffix	0	3
S3168	"async" methods should not return "void"	0	2
CS1998	Async method lacks 'await' operators and will run synchronously	0	1
CS8605	Unboxing a possibly null value.	0	1
CS8632	The annotation for nullable reference types should only be used in code within a '#nullable' annotations context.	0	1
CS8765	Nullability of type of parameter doesn't match overridden member (possibly because of nullability attributes).	0	1

Table 5.2: The number of submissions where the reported diagnostics occurred for both courses

Table 5.2 shows the number of submission where the reported diagnostics occurred. In the following subsections, we highlight the most interesting findings and try to categorize the most typical errors without claiming completeness.

5.1.1 Nullable reference types

Huge part of the findings are from the compilers nullable reference types warnings. It is an optional feature of C# that tries to avoid situations where `System.NullReferenceException` is thrown. To achieve this, the compiler does static flow analysis to determine if a variable with reference-type is dereferenced before getting a non-null value. The developers should also mark their variables explicitly nullable if needed [29]. According to the instructors who took part in the experiment, the usage of this language feature was not enforced in all groups, and this is reflected in the number of diagnostics.

5.1.2 Violation of conventions

The most common error was *"Fields should not have public accessibility"* which was already described in Section 3.2.2.1. In such cases, the student should reconsider of what the user of the class can do with its fields. The fields should be completely hidden or properties should be used to access them.

Most of the styling and naming related rules were removed from the used ruleset, but some basic rules were left related to naming. Type names in C# should be written in *PascalCase*. Also, some types like exceptions should have a suffix that refers to their roles. For example, an exception related to validation should be named as `ValidationException` instead of `Validation`. In contracts there are cases, where suffixes are not recommended. Such as an enum for color values should be named `Color` instead of `ColorEnum`.

S3903 checks if all types are defined in namespaces. For simple programs, a violation of this rule would not cause serious problems, but polluting the global namespace is generally a bad idea. It can lead to name collisions, and types from the global namespace cannot be referenced from the code.

5.1.3 Incorrect value is given to a variable or field

Assigning the result of integer division to a floating point variable (*S2184*) was a fairly common mistake, which was already described in Section 3.2.2.4. In most cases, the students tried to calculate the average of values and forget about integer division, resulting in an incorrect result.

Assignment of constant conditional expressions in conditional expressions (*CS0665*) is most likely a mistake, where the student accidentally assigned a value to a boolean variable with = instead of comparing it to the expression it with ==. In this case, the value of the expression is assigned to the variable and this value is used by the if-statement.

Assignment made to same variable (*CS1717*) usually indicates that the student assigned a wrong value to a variable or field. For example, assigned a field to itself in a constructor instead of the given parameter, as shown in Code 5.1.

```
class Example {
    private int a, b;

    public Example(int a) { // Formal parameter int b was accidentally omitted
        this.a = a;
        this.b = b; // The this.b is set to its own previous value
    }
}
```

Code 5.1: The value of `this.b` is set to itself

When *"Return values from functions without side effects should not be ignored" (S2201)* was raised, it showed that the student was not aware that the function does not have a side effect and the return value should be used. LINQ functions are good examples, they do not modify the source, instead their return values should be used in by subsequent operations (Code 5.2).

```
// Incorrect
int[] arr = ReadIntArray();
arr.Where(i => i % 2 == 0);
return arr;

// Correct
int[] arr = ReadIntArray();
return arr.Where(i => i % 2 == 0);
```

Code 5.2: The return value of `Where` is ignored

5.1.4 Loop execution times, conditional branches and recursion

A loop with at most one iteration should be reviewed (*S1751*), because the student broke/returned too early from the loop, or defined a wrong exit condition or used it in a scenario where it acts as an if-statement.

S2251 is usually reported when a loop moves the counter to the wrong direction. For instance, the condition checks if the counter is greater than zero, but it is increased in iteration (Code 5.3).

```
for (int i = 10; i > 0; ++i) { /* ... */ }
```

Code 5.3: The counter is moved to the wrong direction

S2583 indicates that either a conditional branch will never be executed, or a part of a conditional expression is never evaluated. Code 5.4 shows a code example where the input was read incorrectly.

```
string foo = "";
// Missing input read logic
bool b = foo == "" || x > 0; // The first part is always true
if (foo != "") { /* Never executed */ }
```

Code 5.4: Unreachable conditional code

Returning early or calling `return`/`break` too early in loop can also lead to unreachable code segments (*CS0162*). It is also possible that the unreachable code segment is unnecessary, and should be removed.

Redundant branches in conditional structures (*S3923*) should be avoided, because it reduces the maintainability of the code. It is also possible, that the branches are did not mean to be the same, but the student implemented one of the branches incorrectly, which can lead to unexpected results when the program is tested.

When there are identical expressions on both sides of a binary operator, the expression is usually can be simplified. For instance, `a == a` will be always true, and the student probably wrote this conditional expressions by a mistake. Similarly, `2 - 2` is always `0`. Obliviously, there are operators where identical expressions are valid, like + or *. These operators are excluded from the analysis performed by *S1764*.

Warning *"Recursion should not be infinite" (S2190)* could indicate various problems. From incorrectly implemented setters that call themselves (described in Section 3.2.2.6) to incorrectly implemented recursive functions.

5.1.4.1 Polymorphism

In C# only `virtual` members can be overridden, and the programmer must explicitly indicate it with the `override` keyword. If the `override` keyword is omitted (*CS0114*), the new member with the same signature just hides the original. In such cases, the actually called member depends on the static type of the variable, and polymorphism is not work as excepted. If member hiding is intended, the programmer should indicate it with the `new` keyword (*CS0108*).

5.1.4.2 Performance and IDisposables

Concatenating strings with the + or += operators multiple times is not efficient in C#, because every time there is a new string created on the heap. The `StringBuilder` class should be used instead, which performs better in these situations (*S1643*).

The correct usage of classes which implement `IDisposable` is also enforced be the rules. We previously described the problems related to the `IDisposable` interface in Section 3.2.2.8.

5.1.5 Asynchronous operations

Diagnostics related to asynchronous operations only occurred in the submissions for the *Web application development* course. This shows that *Object-oriented programming* is more of an introductory course, while students who took *Web application development* have to use more advanced C# language features. We presented these diagnostics in Section 3.2.2.2 and 3.2.2.3.

5.2 Changes in the students' behavior

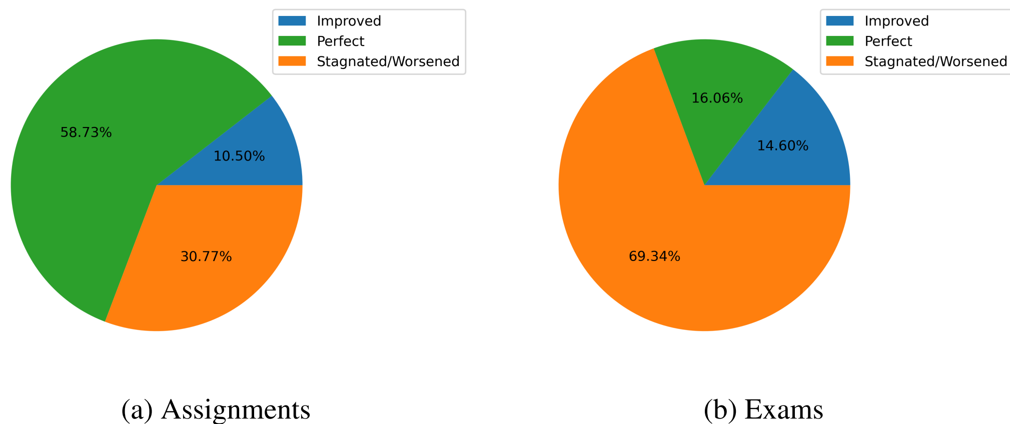
We categorized the submissions based on how the worst and the latest upload related to each other. Three categories were defined:

Perfect: the worst upload had zero problems, so either the submission had one perfect upload, or it stayed perfect during multiple uploads.

Improved: the worst upload of the submission has problems, but the number of problems was less in the latest upload than the worst one.

Stagnated/Worsened: the latest upload has more than zero problems, and the number of problems equal to the worst one. This means that code quality either stagnated or worsened during the uploads. It is worth to note, that the dataset contains unfinished uploads for the submissions, so there were cases where the students introduced more errors as the development of the submission progressed.

5.2.1 Object-oriented programming



(a) Assignments (b) Exams

Figure 5.1: *Object-oriented programming* submissions categorized

We examined the assignments and exams separately, because we assumed that students pay less attention to code quality during exams. This has been confirmed, by Figure 5.1 we can see that *58.73%* of the submission were classified into the *Perfect* category for assignments. In contrast, the *69.34%* of the exam submissions were categorized as *Stagnated/Worsened*, which means that code quality stagnated or worsened during the uploads.

We can also notice, that the improvement rate was relatively low in both cases, but the students were told that this is an experimental feature, and fixing the reported findings is not mandatory. Also, in case of assignments where the more than the half of the submissions were classified as *Perfect*, the low fixing rate is less problematic, than the exams where this number was *16.06%*.

5.2.2 Web application development

Figure 5.2 shows that while the number of perfect solutions was much lower compared to the OOP assignments, the improvement rate was much better. Probably, because of the

complexity of the assigned tasks. Also, only the *23.53%* of submissions were categorized into the *Stagnated/Worsened* category. However, two factors should be considered for this course: as it was already mentioned the number of students significantly lower compared to the *Object-oriented programming* course, and the enrolled students has more experience with the C# programming language.

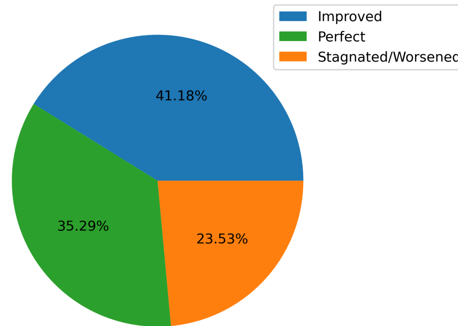


Figure 5.2: *Web application development* assignments categorized

5.2.3 Changes in the average number of uploads

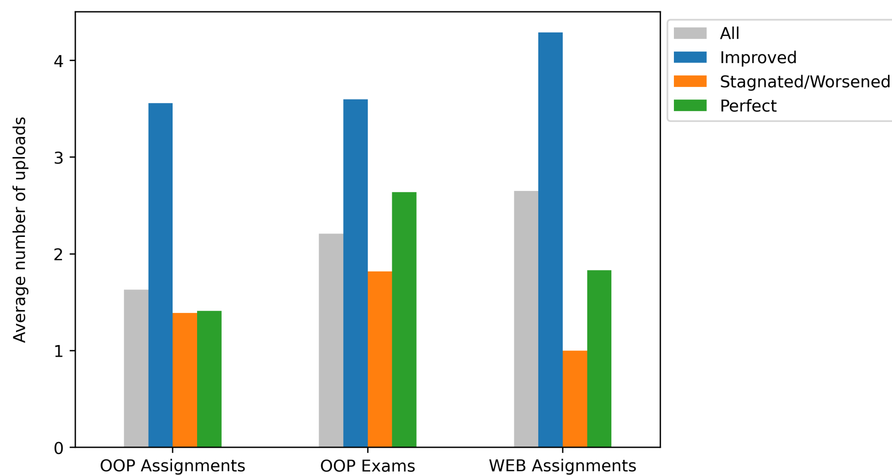


Figure 5.3: The average number of uploads for each category

In Figure 5.3 we can see that the average number of the uploads are significantly higher in the *Improved* category compared to the other categories and the average number of uploads for all submissions.

The current differences between the categories are normal, as it was expected that the improved category will have significantly higher numbers than the other categories. For static code analysis, an incremental approach with earlier uploads can help the students

to fix their errors as early as possible. In contrast to the automatic tester of the evaluator system, which usually fails for unfinished submissions. Although, the number of upload count requires further attention in the future. If instructors find that the students try to fix the warnings from the static analyzer and failed test cases from the automatic tester by brute force without actually understand their feedback, then an option for throttling or limiting the number of uploads should be considered.

5.3 Corrected errors

In this section, we review the number of diagnostics in the student's first, last, and worst uploads for each submission. The section focuses on submissions from *Improved* category introduced in the previous section, so we can assume that the student has taken into account the feedback from the analyzers. Both courses were examined separately due to the difference between the skill-levels of the students. The assignments and the exams were also separated, because the previous section showed that students pay less attention to fixing issues during exams.

5.3.1 Object-oriented programming

ID	Description	First	Worst	Last
S1104	Fields should not have public accessibility	61	66	18
CS8600	Converting null literal or possible null value to non-nullable type.	33	41	12
CS8618	Non-nullable field must contain a non-null value when exiting constructor. Consider declaring as nullable.	23	34	3
CS8602	Dereference of a possibly null reference.	18	24	6
CA1001	Types that own disposable fields should be disposable	18	23	18
CS8604	Possible null reference argument.	14	13	8
CS8603	Possible null reference return.	14	13	4
S2930	"IDisposables" should be disposed	13	16	6
S101	Types should be named in PascalCase	12	15	3
CS8625	Cannot convert null literal to non-nullable reference type.	11	13	6
CS8601	Possible null reference assignment.	6	6	2
S1643	Strings should not be concatenated using '+' in a loop	6	4	5
CS0649	Field is never assigned to, and will always have its default value	5	5	1
S3903	Types should be defined in named namespaces	5	5	0

ID	Description	First	Worst	Last
CA1050	Declare types in namespaces	4	4	0
S2583	Conditionally executed code should be reachable	4	4	0
S1751	Loops with at most one iteration should be refactored	2	2	0
S2184	Results of integer division should not be assigned to floating point variables	2	2	0
CS0436	Type conflicts with imported type	2	0	0
CA1710	Identifiers should have correct suffix	1	1	0
CS0162	Unreachable code detected	1	1	0
CS0642	Possible mistaken empty statement	1	1	0
S1871	Two branches in a conditional structure should not have exactly the same implementation	1	1	0
S3400	Methods should not return constants	1	0	0

Table 5.3: The number of reports in the first, worst and last uploads in the improved solutions from the assignments of *Object-oriented programming*

ID	Description	First	Worst	Last
S1104	Fields should not have public accessibility	82	82	66
CS8625	Cannot convert null literal to non-nullable reference type.	12	12	7
S2583	Conditionally executed code should be reachable	11	12	1
CS8600	Converting null literal or possible null value to non-nullable type.	11	11	4
CS8618	Non-nullable field must contain a non-null value when exiting constructor. Consider declaring as nullable.	6	6	2
CS8602	Dereference of a possibly null reference.	5	5	4
CS8603	Possible null reference return.	5	5	1
S1751	Loops with at most one iteration should be refactored	2	2	2
CS0665	Assignment in conditional expression is always constant	2	2	0
S2184	Results of integer division should not be assigned to floating point variables	2	2	0
CA1001	Types that own disposable fields should be disposable	1	1	1
CS1717	Assignment made to same variable	1	1	1
CS8619	Nullability of reference types in value doesn't match target type.	1	1	0

Table 5.4: The number of reports in the first, worst and last uploads in the improved solutions from the exams of *Object-oriented programming*

Tables 5.3 and 5.4 show the improvements in the assignments and exams of the *Object-oriented programming*. We can notice that diagnostics related to the usage of classes that implement the `IDisposable` were often left in the final solution. The correct usage of this concept was not enforced in this course, so either this should be enforced in the future or the analyzer configuration should be changed for this course. The same applies to *S1643* which is a rule that recommends the usage of `StringBuilder` instead of repeatedly concatenating string with the + operator.

The enforcement of the usage of nullable reference types can also be argued, because it was optional, and it is a relatively new language feature. However, correctly applying it can prevent potential problems.

In both cases, the most common error was fields with public visibility (*S1104*). However, there was a relatively big difference between the assignments and the exams, regarding the number of occurrences in the last upload. Probably, because this is just a violation of a convention, and it does not result in a malfunction.

Less common, but more serious issues were fixed in both cases, but we can also notice that students more likely to fix their issues in an assignment than an exam.

5.3.2 Web application development

ID	Description	First	Worst	Last
CS8632	The annotation for nullable reference types should only be used in code within a '#nullable' annotations context.	23	23	0
S101	Types should be named in PascalCase	10	10	10
CS8602	Dereference of a possibly null reference.	6	8	2
CS8618	Non-nullable field must contain a non-null value when exiting constructor. Consider declaring as nullable.	5	10	1
CA1711	Identifiers should not have incorrect suffix	4	4	1
S3168	"async" methods should not return "void"	4	4	0
S1104	Fields should not have public accessibility	1	3	0
CS8619	Nullability of reference types in value doesn't match target type.	1	1	1
CS8765	Nullability of type of parameter doesn't match overridden member (possibly because of nullability attributes).	1	1	1
CS1998	Async method lacks 'await' operators and will run synchronously	1	1	0
CS8600	Converting null literal or possible null value to non-nullable type.	1	1	0

ID	Description	First	Worst	Last
CS8605	Unboxing a possibly null value.	1	1	0
S1643	Strings should not be concatenated using '+' in a loop	1	1	0
S2201	Return values from functions without side effects should not be ignored	1	1	0

Table 5.5: The number of reports in the first, worst and last uploads in the improved solutions from the exams of *Web application development*

Due to the lower number of students and the higher complexity of the tasks, there were warnings that only occurred in one or a few students' submission. We can see that even the most common warnings has a better fixing rate, and warnings related to nullable reference types are also fixed at most times in the final uploads.

5.4 Feedback from the instructors

At the end of the research, the five instructors who took part in the experiment were asked to fill a form about their experience with the system. This section summarizes their answers to the questions that were relevant to the research. Apart from describing their experiences, they were asked to rate some aspects of the systems from *1* to *5*, where *1* was the worst and *5* was the best option.

5.4.1 Attention paid to the reports

All the instructors introduced the system as an optional feature to their groups. Taking part in the experiment and fixing the reported warnings was not mandatory. The students were also told that they can ignore certain reports if they do not agree with them. While most of the instructors reviewed the reports, the depth of the review varied between them. Ultimately, the presence of reports did not have a major impact on the final grades, according to the given answers and the average of the ratings given for this aspect, which was *2.40*. This is due to the experimental state of the system.

Judging from the previously mentioned circumstances, it was expected that students paid even less attention to the reported problems. The instructors rated the attention from the students only *2.00* on average. The behavior of the students across the groups was not changed significantly after the tools were introduced, according to the instructors.

Although, they noticed that students who took the time to fix the reports had more uploads compared to the others. Probably, this would be more noticeable if the usage of the tool was mandatory. All in all, the experience of the instructors from this aspect was really similar to the presented results in Section 5.2.

5.4.2 Clarity and relevance of the reports

The ratings regarding the relevance of the reports were mixed, where the average was *3.20*. Perhaps, because the same ruleset, which is described in Appendix A, was used for both courses. This shows the need for different rulesets for different courses or even groups. On one hand, warnings can be distracting for the students in case of the *Object-oriented programming* course, which is taught at the second semester. On the other hand, the same warnings could have more relevance in case of the *Web application development* course, where the enrolled students are in their fourth or sixth semester.

The clarity of the reports was rated good (*4.40*) by the asked instructors. However, there was one documented case when a student applied the wrong solution that silenced the analyzer warnings. Diagnostic *S1643* checks if the student concatenates strings in loops with the + or += operators, instead of using the `StringBuilder` class. It turned out that, the analyzer only checks for the usage of the aforementioned operators, and it does not detect other solutions with equally bad performance. For instance, it can be silenced with the usage of string interpolation, which can still lead to unnecessary heap allocations in a loop, as shown in Code 5.5. Thus, the student would think that the problem is solved, but only the analyzer missed it.

```
1  string[] arr = new string[] { /* Array items... */ };
2
3  // Incorrect solution, detected by the analyzer
4  string str = "";
5  foreach (string item in arr) { str += item; }
6
7  // Incorrect solution, missed by the analyzer
8  string str = "";
9  foreach (string item in arr) { str = $"{str}{item}"; }
10
11 // Correct solution
12 StringBuilder sb = new StringBuilder();
13 foreach (string item in arr) { sb.Append(item); }
14 string str = sb.ToString();
```

Code 5.5: String concatenation: a mistake missed by the analyzer

Two important conclusions can be drawn from this case. First, when introducing such a tool, the warnings of the analyzers and the fixes made by the students should be reviewed regularly to check if the introduction of the system has the expected effect and really helps the students. Second, an additional hint for this issue would have been useful and helped the student to choose the most optimal solution.

The description of the warnings produced by the system was the same as the ones that can be found in the tables of Appendix A at the moment of writing this thesis. Furthermore, additional documentation links were not presented on the user interface for the Roslyn-based analyzers. Both the aforementioned incident, both the answers from the instructors showed that the detail of the feedback should be improved in the future.

Chapter 6

Conslusion

The thesis consists of two major scientific contributions. In the first part of the research I evaluated more than 5000 student submissions written in C++ and C# by running static code analysis on them. I have found violations of conventions and various programming bugs which could have been filtered with static analysis, but were overlooked by the teachers, probably due to the high number of student submissions they had to evaluate and grade. In these cases, the feedback provided by the analyzers could help students to fix their mistakes before the deadlines and learn from them. Furthermore, the usage of these tools would allow a more thorough assessment by instructors and speed up the grading process.

After I tested the code analyzers on previous solutions, I decided to integrate *Ericsson CodeChecker* software to the automatic evaluator system of *TMS (Task Management System)*, and adapt it to the needs of computer science education. I created a simplified interface, where instructors can easily configure *CodeChecker* or other analyzers that are supported by its *Report Convert Tool*. I also intended to present the findings of the tools in a legible format for the students. My solution was tested on live data in the spring semester of the 2022/2023 academic year. I created a custom configuration for the used analyzer tools, because both the reviewed literature and my previous experiment showed that the default configuration of the tools produce too many warnings for the students. Overall, the system analyzed more than 800 student submissions written in C# with more than 1400 individual uploads. I presented the findings from the period when the tool was active, and examined how it changed the behavior of the students. I also collected and evaluated feedback from instructors who took part in the experiment.

Based on the results of the experiments, I concluded that automated static code analy-

sis with the right configuration helps both the students and instructors. During the experiment, fixing the reported problems was optional, thus the improvement rate was mixed between groups and task categories even in the same course. So, the impact of the system could be more significant in the future if the usage of the tool is uniformly required for all groups.

6.1 Future work

In the future, we are planning to enable static analysis for other courses at *ELTE Faculty of Informatics*. The impact of the tool could be investigated in more detail when data is available from multiple semesters and courses.

The detail of feedback should also more customizable in the future, as the ideal level of feedback can depend on the skill level of students. For introductory courses, it would be useful if instructors could attach hints for the most typical errors to ensure that students will find the most optimal solution for the problem. In contrast, it is possible that some instructors would prefer to completely hide the findings from the students and use it only during grading.

This research focused on the usage of existing professional analyzer tools. Developing custom solutions or extending the used ones with custom rules can be also considered in the future. Thus, some errors relevant to a certain course that are not covered by the existing tools could be avoidable. Also, instructors would be able to enforce custom requirements, such as the usage of certain language elements or architectural patterns.

Acknowledgements

The writing of this thesis was supported by a student research scholarship from ELTE Faculty of Informatics.

I am grateful to my supervisor, Máté Cserép, for his guidance during the research and the Software Technology Lab courses. I must also thank to instructors who took part in the experiment and provided feedback about my solution: Balázs Csendes, Botond Geönczeöl, Dávid Magyar, Ha Tuan Nguyen and Zsombor Hartmann.

Appendix A

Enabled C# diagnostics during the live test of the implementation

I reviewed the compiler warnings, and diagnostics from the used analyzer Nuget packages and manually enabled them in the used configuration.

The warnings of the compiler were enabled with a few exceptions [30]. The following warnings were disabled:

- Warning related to unused elements: CS0067, CS0168, CS0219, CS0414, CS8321
- Warnings related to the comments: CS1570, CS1571, CS1572, CS1573, CS1574, CS1580, CS1581, CS1584, CS1587, CS1589, CS1590, CS1591, CS1592, CS1607
- Messages with lower and higher severity than warning were also discarded. Less serious messages were not relevant, and error messages are already reported when the automatic tester tries to compile the solution.

The complete list of diagnostics from the Nuget packages can be found in Table A.1, A.2, and A.3. The IDs and descriptions of the diagnostics were extracted from the used packages [12, 13, 14].

Microsoft NetAnalyzers	
ID	**Description**
CA1001	Types that own disposable fields should be disposable
CA1024	Use properties where appropriate
CA1036	Override methods on comparable types
CA1050	Declare types in namespaces
CA1061	Do not hide base class methods
CA1069	Enums values should not be duplicated
CA1507	Use nameof to express symbol names
CA1508	Avoid dead conditional code
CA1710	Identifiers should have correct suffix
CA1711	Identifiers should not have incorrect suffix
CA2000	Dispose objects before losing scope
CA2200	Rethrow to preserve stack details
CA2214	Do not call overridable methods in constructors
CA2226	Operators should have symmetrical overloads
CA2245	Do not assign a property to itself
CA5363	Do Not Disable Request Validation
CA5391	Use antiforgery tokens in ASP.NET Core MVC controllers

Table A.1: Enabled diagnostics from *Microsoft NetAnalyzers*

Roslynator Analyzers	
ID	**Description**
RCS1075	Avoid empty catch clause that catches System.Exception.
RCS1204	Use EventArgs.Empty.
RCS1210	Return completed task instead of returning null.
RCS1215	Expression is always equal to true/false.
RCS1242	Do not pass non-read-only struct by read-only reference.

Table A.2: Enabled diagnostics from *Roslynator Analyzers*

SonarAnalyzer CSharp	
ID	**Description**
S100	Methods and properties should be named in PascalCase
S101	Types should be named in PascalCase

ID	Description
S1048	Destructors should not throw exceptions
S1104	Fields should not have public accessibility
S1163	Exceptions should not be thrown in finally blocks
S1206	"Equals(Object)" and "GetHashCode()" should be overridden in pairs
S1226	Method parameters, caught exceptions and foreach variables' initial values should not be ignored
S1244	Floating point numbers should not be tested for equality
S1643	Strings should not be concatenated using '+' in a loop
S1656	Variables should not be self-assigned
S1696	NullReferenceException should not be caught
S1751	Loops with at most one iteration should be refactored
S1764	Identical expressions should not be used on both sides of a binary operator
S1848	Objects should not be created to be dropped immediately without being used
S1862	Related "if/else if" statements should not have the same condition
S1871	Two branches in a conditional structure should not have exactly the same implementation
S1944	Inappropriate casts should not be made
S2114	Collections should not be passed as arguments to their own methods
S2123	Values should not be uselessly incremented
S2183	Integral numbers should not be shifted by zero or more than their number of bits-1
S2184	Results of integer division should not be assigned to floating point variables
S2190	Recursion should not be infinite
S2201	Return values from functions without side effects should not be ignored
S2222	Locks should be released on all paths
S2225	"ToString()" method should not return null
S2234	Parameters should be passed in the correct order
S2251	A "for" loop update clause should move the counter in the right direction
S2252	For-loop conditions should be true at least once
S2259	Null pointers should not be dereferenced
S2275	Composite format strings should not lead to unexpected behavior at runtime
S2328	"GetHashCode" should not reference mutable fields
S2345	Flags enumerations should explicitly initialize all their members
S2551	Shared resources should not be used for locking
S2583	Conditionally executed code should be reachable
S2674	The length returned from a stream read should be checked
S2688	"NaN" should not be used in comparisons
S2757	"=+" should not be used instead of "+="
S2761	Doubled prefix operators "!!" and "~~" should not be used

ID	Description
S2857	SQL keywords should be delimited by whitespace
S2930	"IDisposables" should be disposed
S2931	Classes with "IDisposable" members should implement "IDisposable"
S2934	Property assignments should not be made for "readonly" fields not constrained to reference types
S2952	Classes should "Dispose" of members from the classes' own "Dispose" methods
S2955	Generic parameters not constrained to reference types should not be compared to "null"
S2995	"Object.ReferenceEquals" should not be used for value types
S2996	"ThreadStatic" fields should not be initialized
S2997	"IDisposables" created in a "using" statement should not be returned
S3005	"ThreadStatic" should not be used on non-static fields
S3168	"async" methods should not return "void"
S3169	Multiple "OrderBy" calls should not be used
S3172	Delegates should not be subtracted
S3244	Anonymous delegates should not be used to unsubscribe from Events
S3249	Classes directly extending "object" should not call "base" in "GetHashCode" or "Equals"
S3256	"string.IsNullOrEmpty" should be used
S3263	Static fields should appear in the order they must be initialized
S3343	Caller information parameters should come at the end of the parameter list
S3346	Expressions used in "Debug.Assert" should not produce side effects
S3397	"base.Equals" should not be used to check for reference equality in "Equals" if "base" is not "object"
S3400	Methods should not return constants
S3449	Right operands of shift operators should be integers
S3453	Classes should not have only "private" constructors
S3456	"string.ToCharArray()" and "ReadOnlySpan<T>.ToArray()" should not be called redundantly
S3464	Type inheritance should not be recursive
S3466	Optional parameters should be passed to "base" calls
S3598	One-way "OperationContract" methods should have "void" return type
S3603	Methods with "Pure" attribute should return a value
S3610	Nullable type comparison should not be redundant
S3655	Empty nullable value should not be accessed
S3869	"SafeHandle.DangerousGetHandle" should not be called
S3887	Mutable, non-private fields should not be "readonly"
S3889	Neither "Thread.Resume" nor "Thread.Suspend" should be used
S3903	Types should be defined in named namespaces
S3923	All branches in a conditional structure should not have exactly the same implementation

ID	Description
S3926	Deserialization methods should be provided for "OptionalField" members
S3927	Serialization event handlers should be implemented correctly
S3949	Calculations should not overflow
S3981	Collection sizes and array length comparisons should make sense
S3984	Exceptions should not be created without being thrown
S4143	Collection elements should not be replaced unconditionally
S4158	Empty collections should not be accessed or iterated
S4159	Classes should implement their "ExportAttribute" interfaces
S4210	Windows Forms entry points should be marked with STAThread
S4260	"ConstructorArgument" parameters should exist in constructors
S4275	Getters and setters should access the expected fields
S4277	"Shared" parts should not be created with "new"
S4428	"PartCreationPolicyAttribute" should be used with "ExportAttribute"
S4583	Calls to delegate's method "BeginInvoke" should be paired with calls to "EndInvoke"
S4586	Non-async "Task/Task<T>" methods should not return null

Table A.3: Enabled diagnostics from *SonarAnalyzer CSharp*

Bibliography

[1] Alexandru G Bardas et al. “Static code analysis”. In: *Journal of Information Systems & Operations Management* 4.2 (2010), pp. 99–107. URL: https://core.ac.uk/download/pdf/6552448.pdf.

[2] Ivo Gomes et al. “An overview on the static code analysis approach in software development”. In: *Faculdade de Engenharia da Universidade do Porto, Portugal* (2009). URL: https://paginas.fe.up.pt/~ei05021/TQSO%20-%20An%20overview%20on%20the%20Static%20Code%20Analysis%20approach%20in%20Software%20Development.pdf.

[3] Maurizio Martignano and IT Spazio. “A new Static Analyzer: The Compiler”. In: *ADA USER* 40.2 (2019), pp. 99–103. URL: https://www.ada-switzerland.ch/doc/auj/auj-40-2.pdf#page=27.

[4] Anastasiia Birillo et al. “Hyperstyle: A Tool for Assessing the Code Quality of Solutions to Programming Assignments”. In: *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2022. Providence, RI, USA: Association for Computing Machinery, 2022, pp. 307–313. ISBN: 9781450390705. DOI: 10.1145/3478431.3499294. URL: https://doi.org/10.1145/3478431.3499294.

[5] Kiran K et al. “An Approach to basic GUI-enabled CI/CD pipeline with Static Analysis tool”. In: *Journal of University of Shanghai for Science and Technology* 23 (June 2021), pp. 683–693. DOI: 10.51201/JUSST/21/05317.

[6] Daniel Marjamäki. *CppCheck*. URL: https://cppcheck.sourceforge.io/ (visited on 02/23/2023).

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

//icai.uni-eszterhazy.hu/icai2017/uploads/papers/2017/final/ICAI.10.2017.23.pdf.

[8] Ted Kremenek. “Finding software bugs with the clang static analyzer”. In: *Apple Inc* (2008). URL: https://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf.

[9] Ericsson Ltd. *CodeChecker*. URL: https://codechecker.readthedocs.io/ (visited on 02/25/2023).

[10] Jürgen Sundström. *Assessment of Roslyn analyzers for Visual Studio*. 2019. URL: https://www.diva-portal.org/smash/record.jsf?pid=diva2:1351980.

[11] Manish Vasani. *Roslyn Cookbook*. Packt Publishing Ltd., 2017.

[12] Microsoft. *Overview of .NET source code analysis*. URL: https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/overview?tabs=net-6 (visited on 04/04/2023).

[13] Sonar. *Code Quality and Security for C# and VB.NET*. URL: https://github.com/SonarSource/sonar-dotnet (visited on 04/04/2023).

[14] Josef Pihrt. *Roslynator*. URL: https://github.com/JosefPihrt/Roslynator (visited on 04/04/2023).

[15] Michael Striewe and Michael Goedicke. “A Review of Static Analysis Approaches for Programming Exercises”. In: *Computer Assisted Assessment. Research into E-Assessment*. Vol. 439. Springer. July 2014, pp. 100–113. ISBN: 978-3-319-08656-9. DOI: 10.1007/978-3-319-08657-6_10. URL: https://link.springer.com/chapter/10.1007/978-3-319-08657-6_10.

[16] Hannah Blau and J. Eliot B. Moss. “FrenchPress Gives Students Automated Feedback on Java Program Flaws”. In: *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’15. Vilnius, Lithuania: Association for Computing Machinery, 2015, pp. 15–20. ISBN: 9781450334402. DOI: 10.1145/2729094.2742622. URL: https://doi.org/10.1145/2729094.2742622.

[17] J. Walker Orr. “Automatic Assessment of the Design Quality of Student Python and Java Programs”. In: *arXiv e-prints* (2022). DOI: 10.48550/ARXIV.2208.12654. URL: https://arxiv.org/abs/2208.12654.

[18] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. "A Tutoring System to Learn Code Refactoring". In: *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. SIGCSE '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 562–568. ISBN: 9781450380621. DOI: 10.1145/3408877.3432526. URL: https://doi.org/10.1145/3408877.3432526.

[19] 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.

[20] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. "Code Quality Issues in Student Programs". In: *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE '17. Bologna, Italy: Association for Computing Machinery, 2017, pp. 110–115. ISBN: 9781450347044. DOI: 10.1145/3059009.3059061. URL: https://doi.org/10.1145/3059009.3059061.

[21] Stephen H. Edwards, Nischel Kandru, and Mukund B.M. Rajagopal. "Investigating Static Analysis Errors in Student Java Programs". In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ICER '17. Tacoma, Washington, USA: Association for Computing Machinery, 2017, pp. 65–73. ISBN: 9781450349680. DOI: 10.1145/3105726.3106182. URL: https://doi.org/10.1145/3105726.3106182.

[22] Péter Kaszab and Máté Cserép. "Detecting programming flaws in student submissions with static source code analysis". In: *Studia Universitatis Babes-Bolyai, Series Informatica* (2023). Accepted for publication.

[23] D Quinlan et al. "Support for Whole-Program Analysis and the Verification of the One-Definition Rule in C++". In: *Static Analysis Summit 2006*. June 2006. URL: https://www.osti.gov/biblio/898014.

[24] Clang Team. *LLVM - Clang-tidy - cppcoreguidelines-slicing*. URL: https://releases.llvm.org/13.0.0/tools/clang/tools/extra/docs/clang-tidy/checks/cppcoreguidelines-slicing.html (visited on 02/25/2023).

[25] Microsoft. *Async return types (C#)*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-return-types (visited on 02/23/2023).

[26] ELTE. *TMS – Task Management System*. URL: https://tms-elte.gitlab.io/ (visited on 02/27/2023).

[27] Zoltán Porkoláb et al. "Codecompass: an open software comprehension framework for industrial usage". In: *Proceedings of the 26th Conference on Program Comprehension*. 2018, pp. 361–369. URL: https://dl.acm.org/doi/abs/10.1145/3196321.3197546.

[28] Ken Cochrane, Jeeva S. Chelladhurai, and Neependra K Khare. *Docker Cookbook - Second Edition*. Packt Publishing Ltd., 2018.

[29] Microsoft. *Nullable reference types (C# reference)*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/nullable-reference-types (visited on 05/12/2023).

[30] Microsoft. *C# Compiler Errors*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-messages/ (visited on 05/11/2023).

List of Figures

3.1 The number of C++ solutions with errors 16
3.2 The number of C# solutions with errors 22

4.1 The relevant components of TMS 26
4.2 The high-level workflow of static analysis in TMS 30
4.3 The CodeChecker workflow 31
4.4 The Report Converter Tool workflow 32
4.5 UML class diagram of the analyzer runners 32
4.6 Entity relation diagram of the relevant tables 33
4.7 UML class diagram of the persistence layer 34
4.8 Overview of the evaluator configuration user interface 36
4.9 Static code analysis settings with CodeChecker selected 37
4.10 List of reports 38
4.11 CodeChecker HTML reports 39

5.1 *Object-oriented programming* submissions categorized 47
5.2 *Web application development* assignments categorized 48
5.3 The average number of uploads for each category 48

List of Tables

2.1 Examples of code analyzers categorized by programming languages . . . 5

3.1 Summary of the used analyzers and evaluated submissions 15

5.1 Number of submissions and uploads for each course and category 40

5.2 The number of submissions where the reported diagnostics occurred for both courses 42

5.3 The number of reports in the first, worst and last uploads in the improved solutions from the assignments of *Object-oriented programming* 50

5.4 The number of reports in the first, worst and last uploads in the improved solutions from the exams of *Object-oriented programming* 50

5.5 The number of reports in the first, worst and last uploads in the improved solutions from the exams of *Web application development* 52

A.1 Enabled diagnostics from *Microsoft NetAnalyzers* 59

A.2 Enabled diagnostics from *Roslynator Analyzers* 59

A.3 Enabled diagnostics from *SonarAnalyzer CSharp* 62

List of Codes

3.1 Field initialization . . . 16
3.2 Performance can be improved with references . . . 17
3.3 Possible integer overflow, because of narrowing conversion . . . 17
3.4 The maximum size of `int` might be smaller than `vec.size()` . . . 18
3.5 `int i` is casted to `unsigned int` . . . 18
3.6 Memory leak: missing destructor . . . 18
3.7 Empty stacks are not handled . . . 19
3.8 Functions in headers . . . 19
3.9 Destructors should be `virtual` . . . 20
3.10 Virtual calls in constructors . . . 20
3.11 Incorrect usage of `delete` . . . 21
3.12 Out of bound indexing . . . 21
3.13 Object slicing . . . 21
3.14 `Field` should not be public . . . 22
3.15 `model.NewGameAsync()` is not awaited . . . 23
3.16 `LoadAsync` cannot be awaited . . . 23
3.17 Integer disivision . . . 23
3.18 Read-only property: `set` should be omitted . . . 24
3.19 Incorrectly implemented setter: infinite recursion . . . 24
3.20 Incorrect override of `GetHashCode()` . . . 25
3.21 The file is not closed after usage . . . 25
3.22 `sr` is returned after disposal . . . 25
5.1 The value of `this.b` is set to itself . . . 44
5.2 The return value of `Where` is ignored . . . 44
5.3 The counter is moved to the wrong direction . . . 45
5.4 Unreachable conditional code . . . 45

5.5 String concatenation: a mistake missed by the analyzer 53