

UNIVERSITY OF DEBRECEN

FACULTY OF INFORMATICS

CodeDOG - Automated Feedback and Assessment for Programming Assignments

Dr. Péter Jeszenszky

Senior Lecturer

Dr. Róbert Tóth

Senior Lecturer

Bakhomious Basta

Computer Science BSc

Bírálatra való beküldéséhez hozzájárulok

Debrecen

2024

Acknowledgments

I would like to thank each and everyone who offered their support during the development of this project.

First of all, my supervisors, Mr. Jeszenszky, or Jeszy for he never left a question of mine unanswered even if they were sometimes idiotic, and his enthusiasm about the project since I first introduced the idea. Mr. Tóth, or Robi, for his constant encouragement, and support not only during the development of this project but during my demonstrator semesters as well and finally enduring my sense of humor, bad memes, and silly development mistakes. One more, Mr. Adamkó for some of the ideas he gave me during the very early stage of the development, and the summer professional scholarship research that I made that contributed heavily to the theoretical background for this paper.

Secondly, to my parents, it seems that I have been a very troublesome child but they still supported my decisions. Well, look Mom and Dad, I am actually graduating now. My mom also joked about mentioning her by name—for very patriarchal reasons that we cannot bear our mother's name—so Mariam, here is your request.

Thirdly, to my friends for keeping up with my depressive episodes throughout last year, I will not mention them by name because they would know themselves and honestly, I have the memory of a goldfish.

Finally, to my guitars and music, for they have saved my life numerous times and for my belief that the littlest of things around us can help us find a cause in this world.

Mischief managed!

Bakh

Contents

1	Introduction	2
1.1	Problems with automatic feedback tools	3
1.2	Proposed solutions	4
2	Analysis of Automatic Assessment Tools	5
2.1	ProgCont	5
2.2	Moodle	6
2.3	check50	9
2.4	HackerRank	10
3	Architecture and Operation of CoDExe	12
3.1	Authentication flow	12
3.2	Exam Application	13
4	CodeDOG – Alternative Automatic Assessment	14
4.1	codedog-common	14
4.2	codedog-exec	16
4.3	codedog-cparser	19
4.4	codedog-server	20
4.5	User Interface	22
5	Syntax Analysis or Parsing	28
5.1	Context-Free Grammars	28
5.2	Parse-Trees	29
5.3	Revisiting codedog-cparser	30
6	Static Code Analysis	33
6.1	Applications of Static Analysis	33
6.2	Approximate Answers	34
6.3	CodeDOG's Future	35
7	Conclusion	37
7.1	Current Results	37
7.2	Future Goals and Further Development Opportunities	38
	Bibliography	40

1. Introduction

The monotonous process of marking students' source codes has become a sort of an issue; staring at source codes that produce one result specified by the teacher, the process of testing the output, checking the source code against predefined criteria, and then giving the final verdict. For instance, in C programming language source codes, with the limited support of automatic code runners and tedious setups in modern IDEs; we would have to compile the file, run it, provide input, and analyze the deficiencies. A process that could be simply automated with a script. Lots of tools are available for these kinds of processes. However, as they vary in their use cases, they come with a great deal of limitations.

In this paper, we report on a project we recently launched, specifically for *High-level programming languages 1* course taught at the University of Debrecen which uses C as its main practice language. Our goal was to develop a tool that instructors can easily use to set criteria and provide feedback on the students' source codes written in C programming language. As a result, we were successful in automating the code-checking process that can be efficiently used to provide feedback to the students on their source codes. We also present the technologies we used to achieve a satisfactory reliable result and our examination of currently available solutions regarding automatic feedback.

C can be a troublesome language to work with, it has been referred to as a high-level assembler. Nevertheless, we cannot deny the fact that it retains its central position. As technology has been advancing over the years, some developers might argue that this language is dead, we overlook the fact that we are bound by memory. The other overlooked fact is that many of the modern programming languages have their compilers or interpreters written in C, at least partially, perhaps the most known example is *CPython* [2]. It also helps new programmers—in our case, freshmen students—grasp the idea of memory. Arguably, C is very easy to learn, it offers only straightforward, single-thread control flow: tests, loops, grouping, and subprograms [3].

Freshmen have always been criticizing the decision to teach C instead of its successor C++, while the latter has a steep learning curve. The migration from procedural paradigm to object orientation would be overwhelming in the case of these two languages specifically. C/C++ are robust languages, yet with the latter having a three-year release schedule [4], it would be hard to introduce to first-year students. Moreover, concepts like pointers, memory management, structs, and more are exceptionally easier to teach in the scope of C.

With this simplicity, comes the fact of tedious work to produce an outcome as it is a relatively “low-level” language. This creates the challenge of automatically checking the students'

source codes and sets the huge and main drawback of the existing solutions, the complete dependency on runtime execution.

1.1. Problems with automatic feedback tools

Perhaps the most crucial question any programmer would ask before embarking on a development journey is, does this already exist? If yes, the next question would be, at what cost? The answer to the former question is yes, the answer to the latter will lead us to address the problems. After analyzing the existing solutions, the experience was unsatisfactory, ranging from problematic setups, limitations, back-doors, and financial difficulties as well.

Most of the available solutions—while being completely efficient, easy to use, and have fancy UIs—for our specific task have the main limitation of depending on runtime execution feedback. In some cases, this can be efficient. However, we are talking about a small set of source codes. In an ideal world, everything can be deducted from runtime execution but this ideal world only exists in the category of `Hello World!` programs. Mostly a simple `printf()` statement would grant the user the satisfaction of seeing: “Solution Accepted”. In our case, this would be insufficient. What if we want to check if a function has been declared? The grading needs to be done on what we propose to call **Block Level**. Figure 1.1 shows an example, it would be helpful to identify every function as a block first, i.e., check the existence of the function signature with the correct properties.

```
int rectangle_area(int a, int b) {  
    return a * b;  
}  
  
int rectangle_perimeter(int a, int b) {  
    return 2 * (a + b);  
}  
  
int square_area(int a) {  
    return rectangle_area(a, a);  
}  
  
int square_perimeter(int a) {  
    return rectangle_perimeter(a, a);  
}
```

Figure 1.1: Various functions in C

1.2. Proposed solutions

Diving to the root of programming language concepts, and perhaps the core foundation of modern linguistics. The proposed solution was to use *Grammars*. A *Grammar* can be described as a set of instructions about how to write statements that are valid for that programming language. The instructions are given in the form of rules that specify how characters and words can be put one after the other, to form valid statements (also called sentences) [5].

Grammars are used to organize compiler front ends [6], hence using a syntax definition we can generate a parse tree. This grants us a syntactical and lexical analysis of the source code provided, hence we can use it to extract specific and minute information. This would help us achieve the block level identification and much more. This encourages us to dive a little towards compiler design and use compiler-component generators or, in simpler terms, a parser generator. The approach depends on parse-tree listeners, the key is to have the parser create a parse tree and then walk it to trigger application-specific code [7]. With multiple available solutions, perhaps the most commonly known and widely used is *ANTLR* [8]. *ANTLR* is quite a fascinating tool, with its Java implementation, it was easy for us to integrate it within our application.

We also cannot deny the necessity of runtime execution it is still crucial for C specifically. As we can discover multiple errors and bugs with static code analysis [9], we still fail shortly for deeper stuff that can be properly identified during runtime, memory leaks for example. Our decision to implement this came to creating a sandbox to execute the students' codes, enforcing a lot of security measures. On the contrary to *ProgCont* [10], one of the systems that were previously introduced at the University of Debrecen, our application proved to be faster and more accurate.

Our other main focus was implementing our service in a clean and scalable approach to add support to more programming languages using the appropriate technologies. In the current scope of the project, we are offering a functional prototype limited to C language source codes and the possible launch of a long-term project. Therefore, combining both techniques and using the correct tools, we were able to achieve satisfactory results after testing our implementation with *High-level programming languages I* 2022–2023 lab test source codes and 2023–2024 lab test as well.

2. Analysis of Automatic Assessment Tools

In this chapter, we would like to examine and analyze some pre-existing tools that we tested while implementing our solution, whether for inspiration or to identify the main drawbacks that should be avoided.

2.1. ProgCont

The ProgCont system [10] was primarily designed for conducting programming competitions. The primary task of the system was to automatically and objectively evaluate solutions submitted for programming tasks [11]. It was then adapted to support programming-related closed exams. A list of available assignments adapted for *High-Level Programming Languages 1-2* courses can be found on the website.

The basic workflow of ProgCont consists of a provided structure or a rule to be followed, with an explanation regarding the problem (Figure 2.1). The task typically required reading standard input, processing the input data, and then writing the result to the standard output. This allowed for fast-checking the output of codes and the system provided immediate feedback to the user with one of the following statuses: *Accepted*, *Rejected*, *Runtime Error*, *Compilation Error*, *Time Limit Exceeded*, *Formatting Error*.

The screenshot shows the ProgCont exercise interface for the task "Prime or Not?". The interface is dark-themed with white text. It includes a title, a problem description, sample input and output, and a link to a solution.

Prime or Not?

Write a program that helps you determine whether the elements of a sequence of numbers are all prime numbers.

The input consists of several lines. Each line contains integers; if there are more numbers, they will be separated from each other by exactly one space character within the line. The first number of each line (n) specifies the number of numbers to be examined in that line. For each line, your program must determine whether the additional n numbers are all prime numbers. The end of the input is indicated by a line where $n = 0$. As a hint, we mention that the smallest prime number is 2.

For each line, your program should write „YES“ to the standard output if all numbers in the line are prime numbers, or „NO“ if there is at least one number among the numbers that is not prime.

Sample Input

```
1. 1 1
2. 1 2
3. 5 2 3 5 7 11
4. 5 2 3 4 5 7
5. 5 3 11 5 7 2
6. 5 11 7 3 9 5
7. 0
```

[download as text file](#)

Output for Sample Input

```
1. NO
2. YES
3. YES
4. NO
5. YES
6. NO
```

[download as text file](#)

One possible solution is available in: [main.c](#).

Acknowledgement This work was supported by the construction EFOP-3.4.3-16-2016-00021. The project was supported by the European Union, co-financed by the European Social Fund.

Figure 2.1: ProgCont exercise interface

The other crucial problem with the ProgCont system is how slow it is. Typically it would take a lot of time to provide feedback to the user. Additionally, the system provides concise feedback regarding the user's source code, a list of statuses is insufficient for the learning and educational process, rather than, the desirable and helpful comprehensive feedback. As a student, I would want to see what I did wrong and how I can correct it. In best cases, where the system is not stuck in limbo (Figure 2.2), we would receive a feedback response¹ (Figure 2.3). We would also draw attention to a crucial point in using this system during examinations, ProgCont would not allow us to add restrictions on non-classroom IP addresses, hence making cheating a lot easier. The ProgCont architecture has many different, domain-specific applications that are designed for various scenarios. Our description is based on the public, non-authorized instance (<https://progcont.hu>), and the last known version of the authorized exam application (<https://exam.progcont.eu>). However, the application has already provided efficient ways to identify non-local IP addresses as its admin page provided logs for the instructors. On the other hand, instructors had to know the set of the local PC's IP addresses to identify them.

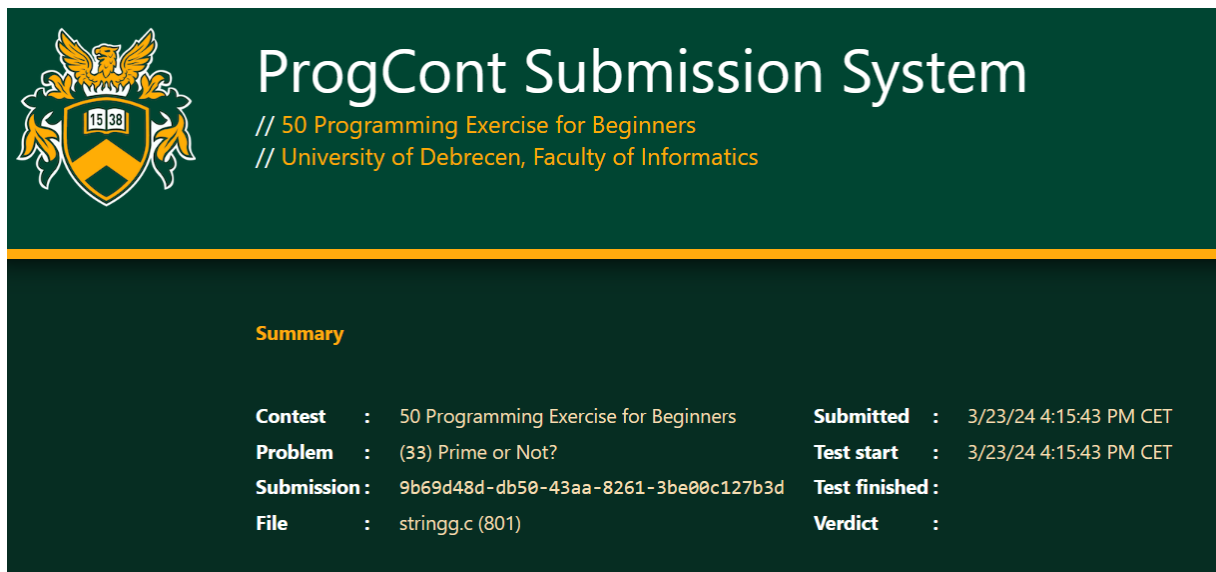


Figure 2.2: *ProgCont* feedback graphical user interface

2.2. Moodle

Moodle is the most popular open-source web-based e-learning platform used internationally by numerous educational institutions, including the University of Debrecen. It has adequate functionalities to manage attendance, course materials, assignments, and -most importantly for our topic- automatic assessment.

¹The "code" value was omitted.

```
{
  "id": "424472",
  "user": "Anonymus",
  "language": "C",
  "time": "2014-05-21 15:08:26.166372",
  "status": "ERES",
  "result": "0%",
  "code": "...."
}
```

Figure 2.3: A JSON feedback from *ProgCont*

At our university, there are primarily two Moodle instances:

- E-learning: <https://elearning.unideb.hu>
- Exam: <https://exam.unideb.hu>

E-learning is mainly used for managing course materials, assignments, and submissions, while Exam was specifically created for examinations. However, Moodle's exam functionality can be utilized within the e-learning system as well. Using the Exam system would create extra administrative work, for instance, you have to enroll all the students in a separate course enlisted on the Exam portal. E-learning exams have been utilized over the years to use *Safe Exam Browser* (SEB) [12] specifications. SEB is a web browser environment to carry out e-assessments safely. The software turns any computer temporarily into a secure workstation. It controls access to resources like system functions, other websites, and applications and prevents unauthorized resources from being used during an exam.

2.2.1. Quiz Activity

The *Quiz* activity allows us to create online quizzes that can meet many teaching needs, from simple, multiple-choice knowledge tests to complex, self-assessment tasks with detailed feedback. Questions are created and stored separately in a question bank and can be reused in different quizzes [13].

The quiz activity supports *CodeRunner*, a question type that allows authors to set programming questions in which the student's answer is code in some programming language, which is graded by running it [14]. It currently supports Python 3, C, C++, Java, PHP, JavaScript (NodeJS), Octave and Matlab (Figure 2.4).

CodeRunner uses a sandbox machine called Jobe server [15], it supports the running of

small compile-and-run jobs in a variety of programming languages. The interface is available via a RESTful API. As the previously mentioned tool, *CodeRunner* uses standard output evaluation. On the contrary, it offers some details to the user, not always sufficient and sometimes cryptic (Figure 2.5). It is worth noting that, in some cases, sharing the expected result between students will allow other students to adjust their solutions accordingly. The pre-defined test cases being visible to the user would be a major downside of using *CodeRunner* as a main assessment tool.

Nevertheless, the experience of using the answer block has proven to be one of the worst experiences among students, as a student myself, this gave a huge limitation on my code writing ability. Programmers tend to rely heavily on integrated development environments (IDEs), which offer freedom and ease of grasping the language syntax, due to their majestic features like code completion [16].

Question 1

Correct

Mark 1.00 out of 1.00

Flag question

Write a Python3 function `sqr(n)` that returns the square of its numeric parameter `n`.

For example:

Test	Result
<code>print(sqr(-3))</code>	9
<code>print(sqr(11))</code>	121

Answer:

```
1 def sqr(n):
2     return n * n
```

Test	Expected	Got	
<code>print(sqr(-3))</code>	9	9	✓
<code>print(sqr(11))</code>	121	121	✓
<code>print(sqr(-4))</code>	16	16	✓
<code>print(sqr(0))</code>	0	0	✓

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00.

Figure 2.4: An example of usage of *CodeRunner*

Precheck only				
	Test	Expected	Got	
✖	adjust=@(x) sprintf("%.6f",x); n=3; disp(adjust(solve(n)));	1.833333	4.833333	✖

Figure 2.5: *CodeRunner* feedback example

2.3. check50

check50 [17] was first introduced to be used within Harvard’s *CS50: Introduction to Computer Science*² course. It is a tool for checking student code against predefined problem sets, it allows the teachers to automatically grade code on correctness and to provide automatic feedback while students are coding. *check50* is built with Python, it runs in the command line and provides the flexibility of writing custom checks in YAML format (Figure 2.6)³ and the usage of its API (Figure 2.7). On top of checking the codes, it also submits them to the course’s dedicated repository. The tool is designed to be used by several instructors for their specific needs. However, it is troublesome to set up *check50* in a local environment, as it depends on *GitHub*, which is not the case for many courses. The approach to use this tool would require some major adjustments to the code but that would easily break the CI/CD pipeline. As extensible as it sounds, we cannot argue that this extensibility is not bounded. It is worth mentioning that it supports unit testing execution for Python and REST API—written with Flask—endpoints testing.

```
check50:
  files: &check50_files
  - !exclude "*"
  - !require caesar.c

submit50:
  files: *check50_files
```

Figure 2.6: *cs50* example of a yaml check

Consequently, it is a **command-line tool**, and the best use case scenario is within a ded-

²<https://p11.harvard.edu/course/cs50-introduction-computer-science>

³Other examples and problems can be found here: <https://github.com/cs50/problems>

```
@check50.check(compiles)
def encrypts_a_as_b():
    """encrypts "a" as "b" using 1 as key"""
    check50.run("./caesar 1").stdin("a")
    .stdout("ciphertext:\s*b\n", "ciphertext: b\n").exit(0)
```

Figure 2.7: *cs50* example of API usage

icated codespace. *Codespaces* [18] is a highly customizable cloud-based development environment, as it supports multiple code editors: *Visual Studio Code*, *JetBrains*, and *Jupyter*.

Using *Codespaces* can be beneficial, but as a student who previously took this course out of their own leisure, the availability of all the required packages and tools made installing them locally later on for personal projects a cryptic challenge. For all we know, the initial step for any programmer is learning how to set up local environments, manage environment variables, and deal with package manager tools, for example, `apt` and `winget`. This is one of the main reasons why we encourage using locally installed IDEs, help students set up their personal computers, and provide detailed guides on it. Moreover, the tool only runs on Linux, on their homepage they clearly state that Windows support is available using WSL [19]. A typical usage example: `check50 <owner>/<repo>/<branch>/<check>`

With closer inspection of *check50*'s implementation, we can quickly realize the tedious setup—other than the obvious dependency on runtime in the case of C source codes—it requires an extremely detailed specification to be able to provide feedback. The API, in itself, is a pretty example of clean architecture [20]. While they offer their API in Python (widely known for its easy syntax) and also clear documentation, the approach itself is not generic enough. Not to mention, the dependency on Linux distributions is also a drawback; it can be easily solved with containerization technologies like *Docker* [21].

In conclusion, as much as *check50* might be the strongest candidate here, it still falls short in usability and portability; it requires a specific dedicated environment and repetitious check writings.

2.4. HackerRank

HackerRank [22] is a technology hiring platform that is the standard for assessing developer skills. Their platform can be used for multiple purposes, it was adapted for multiple online courses on various educational platforms via their *HackerRank Tests* [23] service. It supports different types of questions and different types of validation, including runtime evaluation and unit tests. They also allow full access to projects via their IDE powered by Visual Studio

Code. For C it uses the C11 standard with GCC compiler⁴.

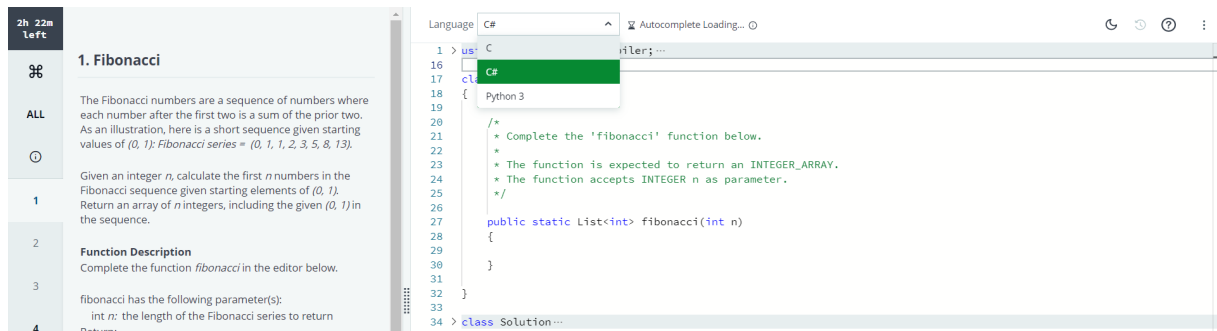


Figure 2.8: *HackerRank Test*

HackerRank offers nice educational kits for interview preparation, their main goal and use case. However, licensing it is costly and might be a financial challenge. The adaptation of such a platform for classroom usage would not be a good decision, considering that it also does not offer a better analysis than what other free solutions offer. However, their range of supported programming languages is vast and various.

Most developers who has hiring experience had at least one *HackerRank Test*, it is not a very smooth experience. Sometimes the checks are poorly written, for example, writing unit-tests that do not follow the unit testing principles [24]. In addition, if the test was defined to be conducted with their IDE, the test cases cannot be hidden, they can be easily spotted in the project directory and sometimes even edited. When checking with runtime execution, the test cases visibility can be controlled but in some instances—it is even reported online on various tech forums—they just fail arbitrarily. In case of hidden test cases, candidates would fail to understand the problem with their code, unless it is reported that there was a faulty test case. The reasoning behind it is, of course, the poor setup; given the complexity of such an application.

Developers have been arguing over the years about whether *HackerRank* is a good tool, diving also into its psychological effect and their experiences using it in their hiring process [25]. The “natural environment” *HackerRank* deems to offer is countered by the fact that an online editor is not a natural development environment. Specifically when *HackerRank* disables copy and paste on its page which makes testing in a proper shell and editor (IDEs) hard or keeps warning the user if the tab is out of the screen’s focus. This leads us to a similar problem to a previously discussed drawback of using *CodeRunner* but with a greater limitation.

⁴We use the C99 standard.

3. Architecture and Operation of CoDExe

In this chapter, we briefly present *CoDExe* environment currently being used in multiple courses in the university to manage examinations instead of *Moodle*. We started using the system at the University of Debrecen in 2023–2024, we had a better experience with it than *Moodle* for multiple reasons including administration logistics and secure exam conditions [26]. It is a **scalable** environment, very similar to a micro-services approach, that supports multiple applications integration within our central application. Thus, we can consider every service as an independent application.

3.1. Authentication flow

To provide a safer examination environment, the students are authenticated in person using their *UniPass* cards [27] and a QR-code for the current session (Figure 3.2). It benefits us with IP tracking, facilitation of administrative processes, and ensuring fair chances among students. Moreover, the scanning is done platform-independent via web application implementation that uses the device's camera—ideally, a mobile phone— then sends the authentication request via a RESTful API (Figure 3.1). Finally, the student could access the exam and start at the appropriate starting time.

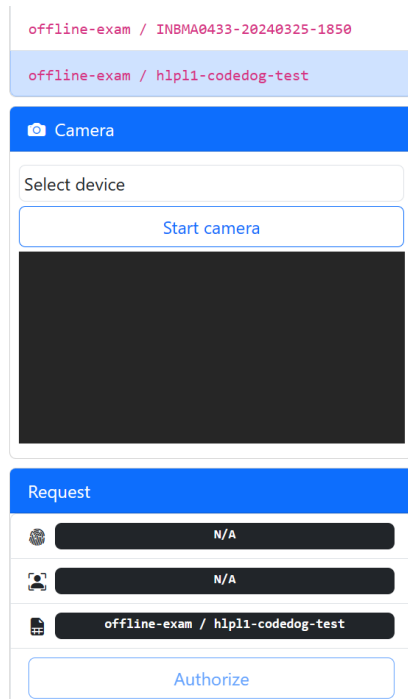


Figure 3.1: Mobile QR-code scanner with relevant activities (exams)



Figure 3.2: Session QR-code

3.2. Exam Application

From our experience, the new exam application saved us a tremendous amount of time; we no longer had to hand exam sheets, check the attendance—as it can be automated using the authentication service previously mentioned—and collect submissions. All of this can be done with simple steps.

The application supports the rendering of exam sheets (Figure 3.3) to the students' screens using Markdown markup language. Moreover, on top of the standard upload, copying, and pasting solutions, students can submit the solutions via a Maven plugin by just providing the `sessionId` in the command line. This lets the student focus more on finishing their solution and decreases the tiresome and frustrating window changes, they can do everything on one screen. We have noticed students window splitting their screen to have both the exam specification open and their IDE side-by-side.

The screenshot displays the CoDExe exam admin panel. At the top, there is a navigation bar with 'CoDExe | Exam | Admin', 'Exams', 'Students', and a 'Logout' button. Below the navigation bar, there are tabs for 'Edit', 'Students', 'Sessions', and 'Results'. The main content area is divided into two sections: 'Properties' and 'Submission collector'.

Properties

Name	Value
ID	INBPA0211-20240403-1600
Start	2024-04-03T16:06:00+02:00
End	2024-04-03T16:46:00+02:00
CodeDog available	☒ true ☐ false Whether the CodeDOG evaluation service is available for the students.
Retro available	☒ true ☐ false Whether the exam can be accessed in application CoDExe Exam Retro after its end.

Submission collector

☒ STANDALONE_TXT ☐ STANDALONE_C ☐ MERGED_SQL

The solution of each exercise goes into a separate file having the `txt` extension.

Solve as many exercises as you can. You should create a new source code for each exercise and follow the instructions. The content of each source code can be copied and pasted here as many times as you want to. The test will lock automatically when the test ends. All the unsaved solutions will be dropped.

Sample input for `EXERCISE-01`

Select it, then `CTRL + C`.

```
4 10 63 45 -68
10 55 -67 93 -26 -55 0 40 81 -92 50
10 -12 -47 -61 -97 30 -57 -37 -35 40 -91
6 65 -61 27 0 61 46
7 -95 -2 84 -67 -58 -77 95
7 6 62 34 -86 -76 -80 -54
6 52 -12 -28 -46 46 -91
10 -67 94 -38 98 -68 3 -28 57 -77 38
7 54 29 31 10 77 -10 64
```

Save

Figure 3.3: *CoDExe* exam admin panel side by side with Markdown markup and rendering

4. CodeDOG – Alternative Automatic Assessment

When designing CodeDOG, our goal was to not only create this alternative to address the burdens that we previously discussed in Chapter 2, but also exploit the *CoDExe* architecture and scalability, following the same architectural patterns to introduce our solution.

We were very careful to consider the environment and some of the students' concerns about using such a service. Vitally, we designed our service to be scalable, portable, clean, and modular [28, 20]. As previously mentioned in Chapter 1, this prototype was adapted as an initial step towards a bigger application, having the chance to live test it with students in a fully realistic exam environment to test C language source codes.

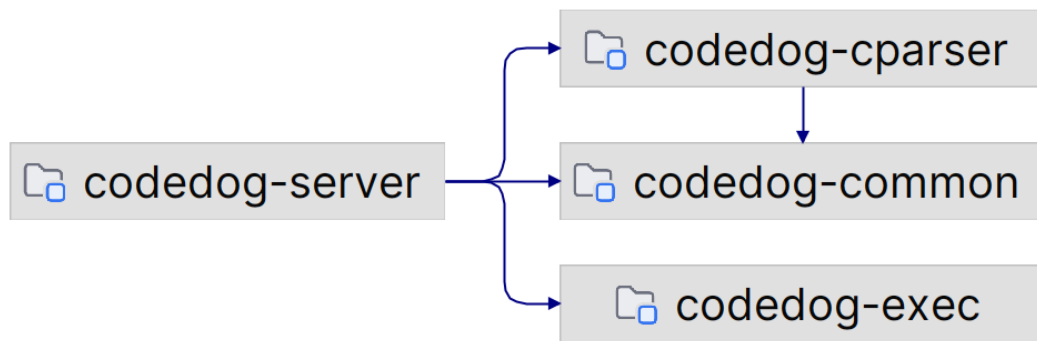


Figure 4.1: CodeDOG modules

The modular structure of the application allowed us to easily manage dependencies and introduce new features. In the next sections, we dive deeply into each module, we will briefly cover `codedog-cparser` as an introduction to Chapter 5.

4.1. `codedog-common`

This module is the crucial skeleton of the application, which handles the database entities, DTOs, and internal structuring of the code. Utilizing multiple design patterns for scalability and ease of feature development. Using Figure 4.2 as a reference, we will start deconstructing and analyzing the design patterns.

From the top of the hierarchy, the `Output` abstract class is the base class we use for output results classes, such as `COutput` which is the specific output we defined for C.

Moving on to `CaseConfiguration`, this interface provides us with the blueprint of any case configurations we want to introduce. Within `COutput`, there are three inner classes; each of which is adapted to a specific case: *structs*, *Runtime Output* and *Functions*.

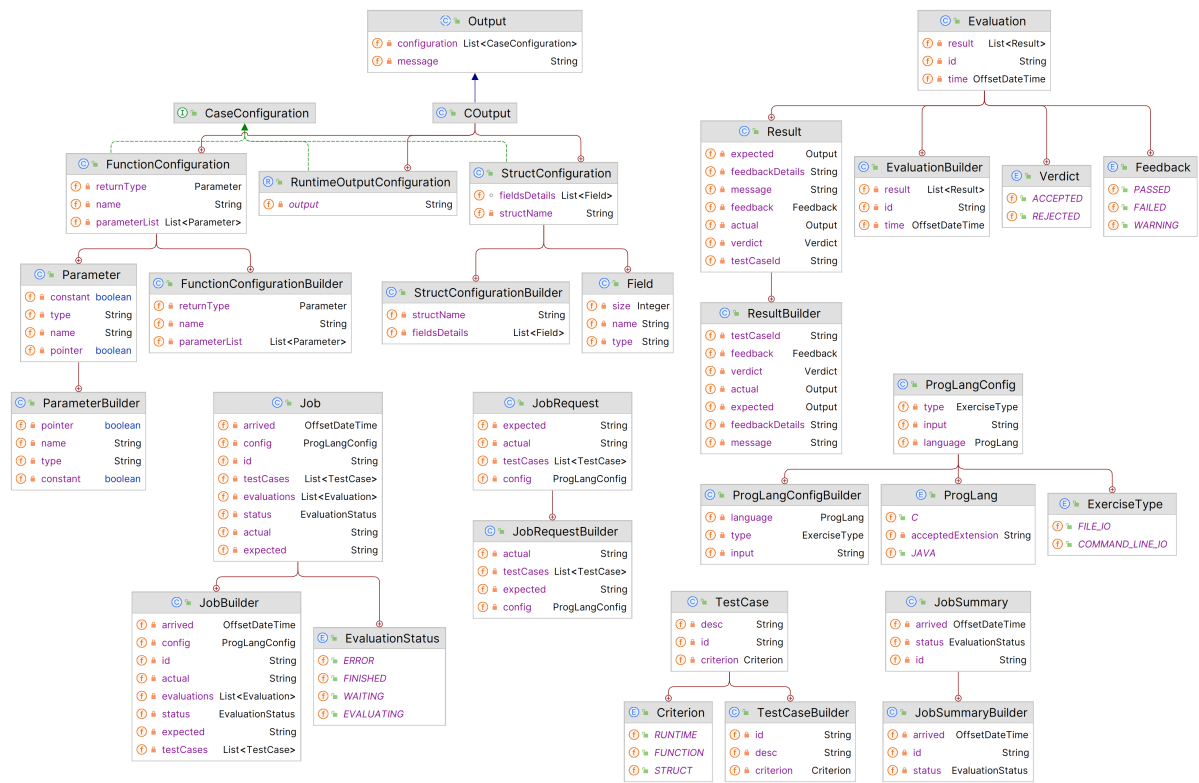


Figure 4.2: *codedog-common* structure

In almost all entity classes, we use the **Builder** creational pattern. The classic example to explain the importance of this is building a house [29] and the dilemma of which constructor should be used. The builder pattern facilitates this, as it enforces a step-by-step process to construct a complex object as a finished product. We rely on it heavily as we sometimes need to build objects at different stages in the application.

Finally, the *Job* class is the core of our application. This is the entity that we store in the database, our choice was *MongoDB* for its wonderful document-oriented data storage [30]. We can use this class to describe the whole idea of our application in one sentence as it contains all the essential information we need. We will revisit this class when we talk about the *codedog-server* module.

Our usage of object-oriented programming and its design patterns, following both the *Gang of Four's patterns* [29] and *Uncle Bob's Clean Architecture principles* [20] allowed us to design this application as extensible as possible. It is quite easy to add support for more programming languages and their relative case configurations. Abstraction is everything, perhaps even in real life.

4.2. codedog-exec

The execution module encapsulates our approach for runtime execution. As previously discussed in Chapter 1, it is difficult to scrap the idea of runtime evaluation for C codes. Lots of information about the code quality and usability can only be detected at runtime. Using tools like *clang* [31], *gdb*, *gcc* [32], and *valgrind* [33]; grants us the access to this information. These tools are available on Unix based systems. However, the first challenge was the portability—beforehand we examined tools like *check50* in Chapter 2 which only ran on Linux—this led us to use **containerization**.

Containerization is the packaging of software code along its dependencies to create a single lightweight executable that runs consistently on any infrastructure. Software developers create and deploy container images—that is, files that contain the necessary information to run a containerized application. Developers use containerization tools to build container images based on the Open Container Initiative (OCI) image specification. OCI is an open-source group that provides a standardized format for creating container images. Container images are read-only and cannot be altered by the computer system [34, 35, 36].

The most popular, de facto standard in software development and engineering is *Docker*. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings. Container images become containers at runtime and in the case of *Docker* containers—images become containers when they run on Docker Engine. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and production.

Using a DockerFile (Figure 4.3) and Docker Compose, we successfully built an *Ubuntu* image with the required environment to compile and execute the source codes.

```
FROM ubuntu:latest
RUN apt-get update && apt-get install -y clang valgrind
WORKDIR /src
```

Figure 4.3: DockerFile used in the project

A traditional docker-compose file is then used to build the image and its container for the application. Using this approach, we can install unlimited tools and set up environments that we might need to add support to other programming languages; emphasizing the scalability and portability of our service.

Using Docker Java API [37], we built our execution module to connect and execute commands inside the container (Figure 4.4). The technical details of the implementation can be

summarized in a few points:

- The Docker Client is initialized by establishing a connection via Apache HttpClient 5.
- The `containerId` is then grabbed from the available container list—this happens in the service’s constructor.

© DockerClientServiceConfig		
ⓕ 🔒	<code>CONTAINER_NAME</code>	String
ⓕ 🔒	<code>containerId</code>	String
ⓕ 🔒	<code>log</code>	Logger
ⓕ 🔒	<code>DOCKER_CLIENT</code>	DockerClient
Ⓜ 🔒	<code>buildClient()</code>	DockerClient
Ⓜ 📁	<code>getOutput(String)</code>	Map<String, String>
Ⓜ 📁	<code>getContainerId()</code>	String
Ⓜ 📁	<code>exitContainer()</code>	void

© Util		
ⓕ 📁	<code>SUCCESS_CODE</code>	String
ⓕ 📁	<code>EXIT_CODE</code>	String
ⓕ 📁	<code>STDOUT</code>	String
ⓕ 📁	<code>STDERR</code>	String

Figure 4.4: codedog-exec implementation of *Docker Client*

4.2.1. clang

The choice of `clang` over `gcc` was a very clear decision for us from the start. `clang` is designed as an API from its inception, allowing it to be reused by source analysis tools, refactoring, IDEs as well as for code generation. The system is highly reusable compared to its predecessor `gcc`. It is also much faster and uses far less memory than `gcc` while still capturing more information about the original source code after compilation using their Abstract Syntax Trees (ASTs) approach, for instance if you write ‘`x-x`’ in your source code, the `gcc` AST will contain ‘`0`’, with no mention of ‘`x`’ [38].

The crucial feature that made us use `clang` was its expressive diagnostics, including but not limited to: column numbers and caret diagnostics, range highlighting for related text, precision in wording, and fix-it hints [39]. This was advantageous; to provide more details about the compilation stage to the students (Figure 4.5).

Furthermore, Clang Static Analyzer (CSA) performs context-sensitive and inter-procedural analysis; designed to be fast to detect common mistakes. The fascinating part is the ability of CSA to detect even potential memory leaks without checking it at runtime—

```
$ gcc-4.2 -fsyntax-only -Wformat format-strings.c
format-strings.c:91: warning: too few arguments for format
$ clang -fsyntax-only format-strings.c
format-strings.c:91:13: warning: '.*' specified field precision is missing a matching 'int' argument
printf("%.d");
      ^
```

Figure 4.5: gcc vs. clang output

although its precision can be debatable—we cannot recall any other static analyzers as powerful as clang [40]. It is available within the clang suite along with debug info tuning, only via command-line arguments.

```
#include <stdlib.h>
int process(void *ptr, int cond) {
    if (cond)
        free(ptr);
}
int entry(size_t sz, int cond) {
    void *ptr = malloc(sz);
    if (ptr)
        process(ptr, cond);
    return 0;
}
```

Figure 4.6: Example CSA input containing a memory leak

```
memorytest.c:15:9: warning: Potential leak of memory pointed to
by 'ptr' [unix.Malloc]
    return 0;
    ^
1 warning generated.
```

Figure 4.7: Example CSA output containing a memory leak

4.3. codedog-cparser

In this section, we briefly cover our parsing service module; Chapter 5 includes a detailed view of the methodology. We mainly used *ANTLR* as our parser of choice for its Java implementation, ease of use, and adaptability to multiple programming languages using *.g4 (Grammar files) [7]. Using the ANTLR 4 Maven plugin, we can generate Java source files for any grammar of choice. In our case, for the C language grammar [41].

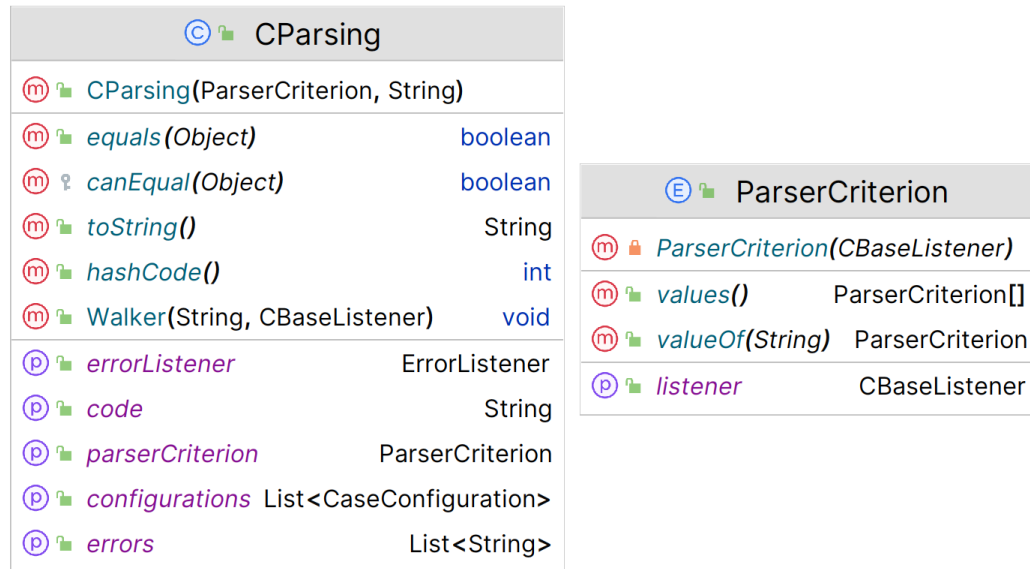


Figure 4.8: codedog-cparser structure

Figure 4.8 shows the structure of the service. It utilizes Lexers and Parsers from ANTLR 4 to generate a **parse tree** for the code given based on a certain criterion. It handles the initialization of a tree walker with the appropriate **listener**. We implemented some listeners

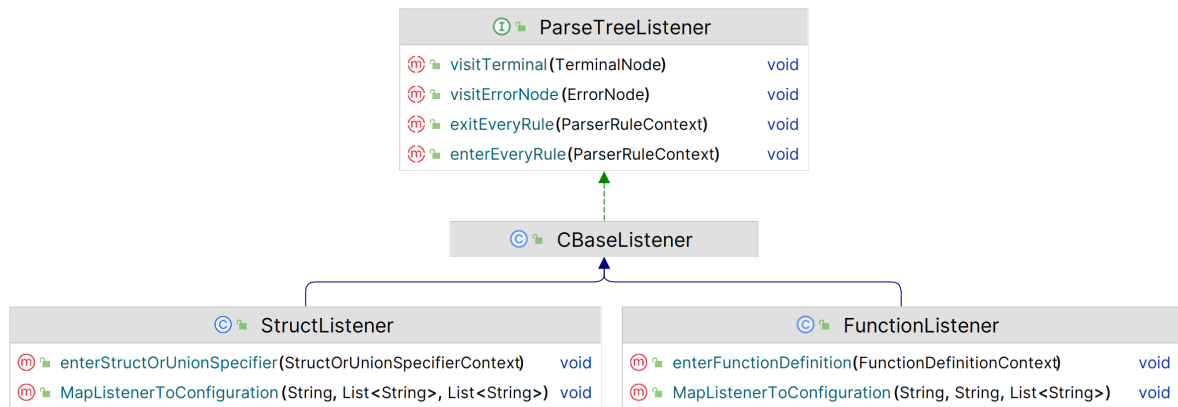


Figure 4.9: Listeners based on ANTLR

for functions and structs, these listeners will listen to the appropriate declarations and return the required information (Figure 4.9). We also created some helper methods to map the output from the parser to its corresponding configuration from `codedog-common`.

4.4. `codedog-server`

Though architects like to be able to strategically plan for the future, the constantly changing software development ecosystem makes that difficult. Since we can't avoid change, we need to exploit it.

–*Building Evolutionary Architectures*

by Neal Ford, Rebecca Parsons, and Patrick Kua

To tie everything together we decided to use a RESTful API, keeping in mind API-first design. API-first design is an approach where developers and architects consider the functionality of their service and design an API in a consumer-centric manner. The API consumer could be a mobile application, another service, or even an external customer. In the field of software architecture, there are a handful of terms that are incredibly difficult to define. The term API, which stands for application programming interface, falls into this categorization, as the concept first surfaced as many as 80 years ago. However, we can define an API as an abstraction of the underlying implementation. On the other hand, REpresentation State Transfer (REST) is a set of architectural constraints, most commonly applied using HTTP as the underlying transport protocol. As the number of APIs grew, it quickly became necessary to have a mechanism to share the shape and structure of APIs with consumers. This is why the OpenAPI Initiative was formed by API industry leaders to construct the OpenAPI Specification (OAS). OpenAPI specifications are JSON- or YAML-based representations of the API that describe the structure, the domain objects exchanged, and any security requirements of the API. [42]

There are various frameworks to build RESTful APIs, our choice was *Spring Boot*. It makes it easy to create stand-alone, production-grade Spring-based Applications that you can “just run” [43]. Following the regular Model-View-Controller (MVC) architecture, we implemented controllers to expose the end-points on which we receive a `JobRequest` (Figure 4.2), and then we process everything in the services.

We segregated each service to do one job, to have a clean reusable code:

- `CParsingService`: Uses the `codedog-cparser` API to parse the given code based on a given criteria.
- `CExecutionService`: The implementation of the *DockerClient* from `codedog-exec`

module, specifically written to execute C related commands to compile and execute the source code with specific error handling.

- **ExecutorService:** This service ties all the other services together in one simple service. It serves as the connection point between all the other service implementations and the main one.

`JobService` is the main service that is directly used by the RESTful API `JobController` to receive and process the jobs being sent on the API endpoint. It has a scheduler that processes the `WAITING` jobs (Figure 4.2) to avoid overloading the server. This implementation allows us to expand and scale the service to process a larger amount of requests if we need to, using an event store or a stream-processing platform, e.g., *Apache Kafka*. Moreover, the current implementation of the system cannot be considered as **Reactive Programming**. However, this can be easily managed with *Spring WebFlux* and *Project Reactor* [43]. For the scale of this prototype, this would have just caused us extra work.

4.4.1. Quick overview of Reactive Programming

In a typical service, a thread is created for each request to be handled. Each thread requires resources, and as such, the number of threads that an application can manage is limited. As a somewhat simplified example, if an app can service 200 threads, that application can accept requests from up to 200 discrete clients at once, but no more; any additional attempts to connect to the service must wait for a thread to become available. Reactive programming was created to address this scalability crisis. [43]

The Reactive Manifesto [44] states that reactive systems are: *Responsive, Resilient, Elastic* and *Message driven*.

In a nutshell, the four key points of reactive systems as listed combine to create (at the macro level) a maximally available, scalable, and performant system requiring the fewest resources possible to do the job effectively.

The reason we mention reactive programming here is that—and as addressed before—it is a potential paradigm to investigate for *CodeDOG*'s future implementations and goals. As we mentioned before, we are building this application to be scalable. For the current scale of this prototype, we do not need to handle a huge amount of clients and the speed of the application is pretty sufficient compared to the other implementations we discussed in Chapter 2. However, when the time comes, the application might be used by a lot of people, possibly other universities, or even be adapted as a general solution for automatic grading for any use case.

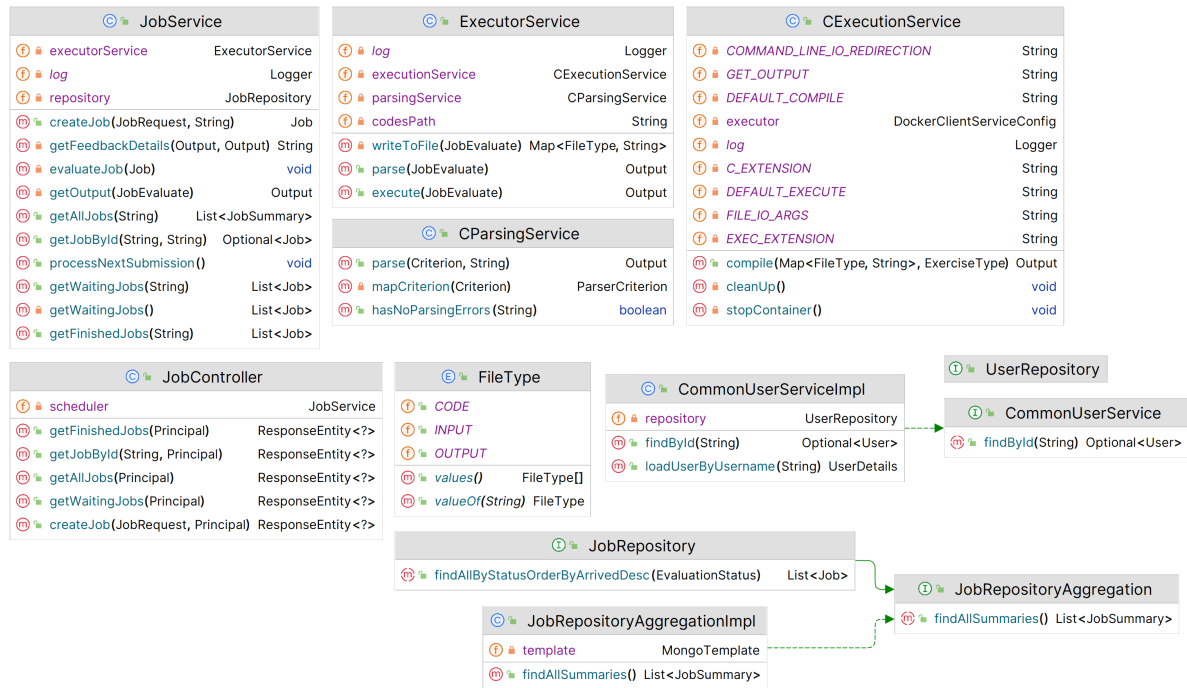


Figure 4.10: codedog-server architecture

4.5. User Interface

As previously mentioned, we integrated CodeDOG with the coDExe exam service. The instructor or exam admin provides the criteria in YAML format from the user interface (Figure 4.11). Also the availability of CodeDOG's feedback to the students is controlled by the instructor. The feedback can be seen for each submission from the exam admin results panel, which can also be used to send the requests for CodeDOG feedback (Figure 4.12).

Property	Value
configuration	<pre>input: abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789 a b c d e f g h</pre>
expected	<pre>#include <stdio.h> #include <stdlib.h> #include <string.h> #include <ctype.h></pre>
testCases	<pre>- id: "test-fun" desc: "Function Output" criterion: "FUNCTION" - id: "test-exec" desc: "Runtime Output"</pre>

Figure 4.11: CodeDOG properties

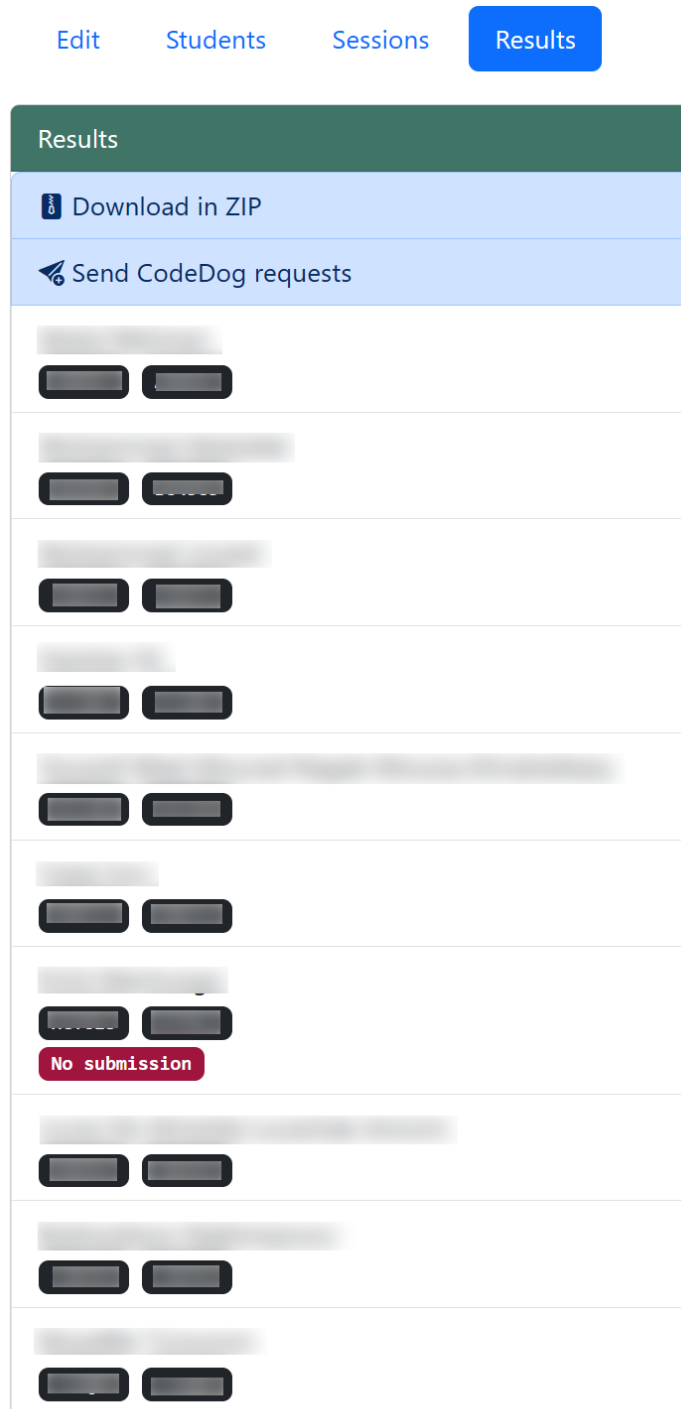


Figure 4.12: CoDExe exam admin results panel

The UI can show the warnings generated by the compiler (Fig 4.13), errors generated at runtime (Figure 4.14), missing functions in the source code (Figure 4.16), and if the code failed to compile or parse (Figure 4.15). The feedback details are sent to the front-end in Markdown format which facilitates the rendering.

Compilation

#test-funCOMPILES

No syntax errors were found.

#test-execCOMPILES

No syntax errors were found but code compiled with warnings

Evaluation

#test-funPASSED

All requirements have been met.

#test-execWARNING

Code compiled with warnings: Warning:

```
660dd9599ceb1802438f9652.c:34:11: warning: implicit declaration of function 'gets' i
      while(gets(line) != NULL)
              ^
660dd9599ceb1802438f9652.c:34:22: warning: comparison between pointer and integer ('
      while(gets(line) != NULL)
                   ~~~~~~ ^ ~~~~~
2 warnings generated
-
/usr/bin/ld: /tmp/660dd9599ceb1802438f9652-dd3c9f.o: in function `main':
<unknown>:
34: warning: the `gets' function is dangerous and should not be used.
```

No further syntax errors found. Code passed all other checks.

Figure 4.13: CodeDOG compiled with warnings example

Evaluation

#test-funPASSED

All requirements have been met.

#test-execWARNING

Code compiled with warnings: Warning:

660dd95a9ceb1802438f965f.c:30:20: warning: format specifies type 'int *' but the arg
scanf("%d",numbers[i]);
 ~ ^~~~~~
1 warning generated.

Error at RUNTIME:

/bin/bash: line 1: 1603 Segmentation fault (core dumped) ./660dd95a9ceb1802438

Missing:

• Runtime Output:

2
4
5
3
2
3
4
4
3
2

Figure 4.14: CodeDOG compiled but had a runtime error example

Compilation

#test-fun

SYNTAX ERROR

Error: Code is not compilable (failed to parse):

- line 7:5 no viable alternative at input
'False\nFalse\nTrue\nFalse\nFalse\nFalse\nFalse'

#test-exec

SYNTAX ERROR

Error: Code is not compilable (failed to parse):

- line 7:5 no viable alternative at input
'False\nFalse\nTrue\nFalse\nFalse\nFalse\nFalse'

Evaluation

#test-fun

FAILED

Error: Code is not compilable (failed to parse):

- line 7:5 no viable alternative at input
'False\nFalse\nTrue\nFalse\nFalse\nFalse\nFalse'

#test-exec

FAILED

Error: Code is not compilable (failed to parse):

- line 7:5 no viable alternative at input
'False\nFalse\nTrue\nFalse\nFalse\nFalse\nFalse'

Figure 4.15: CodeDOG failed to compile/parse example

Compilation

#test-fun

COMPILES

No syntax errors were found.

#test-exec

SYNTAX ERROR

No further syntax errors found.

Evaluation

#test-fun

FAILED

Missing:

- **Function** `count_of_decreasing_neighbours`, returning `int` with parameters:
 - `numbers` of type `int*`
 - `length` of type `int`

#test-exec

FAILED

No further syntax errors found.

Figure 4.16: CodeDOG missing functions example

5. Syntax Analysis or Parsing

In instruction in the language the following process will occur: the learner names the objects; that is, he utters the word when the teacher points to the stone.—And there will be this still simpler exercise: the pupil repeats the words after the teacher—both of these being processes resembling language.

—*Philosophical Investigations*
by Ludwig Wittgenstein

A language, whether natural or artificial, is a set of strings of characters from some alphabet. The strings of a language are called **sentences** or statements. The syntax rules of a language specify which strings of characters from the language's alphabet are in the language [5]. We began this chapter with a quote from Wittgenstein, to describe the problem of learning a language but the more general problem is with describing a language's syntax. We can ambiguously define a language as a set of instructions just like Wittgenstein did, but in the realm of programming languages, the learning factor is different; i.e., we cannot point at a variable and say: It is a variable, the learner's question would be: What is a variable? What is the rule for defining a variable? That is typically when we start to introduce **syntax** rather than pointing out lines of code.

The general solution to this problem was defined by researchers and programmers over the last decade to achieve a simple approach, cut down the cake into smaller bites. Formal descriptions of the syntax of programming languages often do not include descriptions of lowest-level syntactic units, we call these units **lexemes**. These lexemes are partitioned into groups—for example, the names of variables, methods, classes, and so forth in a programming language form a group called identifiers. Each lexeme group is represented by a name, or **token**. So, a token of a language is a category of its lexemes. One can think of programs as strings of lexemes rather than of characters. This is the actual first phase of a compiler called *lexical analysis*, shortly, for each lexeme, the lexical analyzer produces as output a *token*. [45, 6]

In this chapter we provide a gentle theoretical introduction then we expand on how we implemented our `codedog-cparser` module using *ANTLR*.

5.1. Context-Free Grammars

A grammar can be regarded as a device that enumerates the sentences of a language [5]. In the mid-1950s, Noam Chomsky, arguably the father of modern linguistics (among other things), described four classes of generative devices or grammars that define four classes of

languages. Two of these grammar classes, named *Context-Free* and *Regular*, turned out to be useful for describing the syntax of programming languages. The forms of the tokens of programming languages can be described by regular grammars. The syntax of whole programming languages, with minor exceptions, can be described by context-free grammars. It is important to note that Chomsky applied his theory to natural languages, but not until later, John Backus and Peter Naur introduced what is known as **Backus-Naur Form** (BNF). BNF became the new formal notation for specifying programming language syntax, nearly identical to Chomsky's context-free grammar. BNF can also be regarded as the metalanguage for programming languages. For simplicity, we are going to refer to context-free grammars simply as grammars. [46, 45]

5.2. Parse-Trees

One of the most attractive features of grammars is that they naturally describe the hierarchical syntactic structure of the sentences of the language they define. These structures are called **parse-trees**. Every internal node of a parse-tree is labeled with a non-terminal symbol; every leaf is labeled with a terminal symbol. Every sub-tree of a parse tree describes one instance of an abstraction in the sentence [45]. Formally, given a grammar, a *parse-tree* is a tree with the following properties [6]:

- The root is labeled by the start symbol.
- Each leaf is labeled by a terminal or by ϵ .
- Each interior node is labeled by a non-terminal.
- If A is the non-terminal labeling of some interior node and $X_1, X_2, \dots, X_n$ are the labels of the children of that node from left to right, then there must be a production $A \rightarrow X_1 X_2 \dots X_n$ each stand for a symbol that is either a terminal or non-terminal. As a special case, if $A \rightarrow \epsilon$ is a production, then a node labeled A may have a single child labeled ϵ .

It is important to note that a grammar can have more than one parse tree generating a given string of terminals, such is said to be **ambiguous**. Since a string with more than one parse-tree usually has more than one meaning, we need to design unambiguous grammars for compiling applications or to use ambiguous grammars with additional rules to resolve the ambiguities. It is also mathematically impossible to determine whether an arbitrary grammar is ambiguous [6, 45]. However, several characteristics are useful in determining whether a grammar is ambiguous [45]:

- Whether the grammar generates the sentence with more than one leftmost derivation.

- Whether the grammar generates a sentence with more than one rightmost derivation.

5.3. Revisiting codedog-cparser

In Chapter 4, we briefly introduced our parsing module and the usage of *ANTLR* as our main language recognition tool. However, we introduced terms like listeners, lexers, and tree walkers—without a proper explanation. Now that we initialized a theoretical background we can embark on a little journey.

ANTLR v4 is a powerful parser generator that you can use to read, process, execute, or translate structured text or binary files. *ANTLR* is widely used in the industry to build all sorts of languages, tools, and frameworks. Including and not limited to Oracle’s SQL developer, NetBeans IDE, and HQL language. [7]

From a grammar, *ANTLR* generates a parser for that language that can automatically build **parse trees**. It also automatically generates **tree walkers** that you can use to visit the nodes of those trees to execute application-specific code.

ANTLR provides support for two tree-walking mechanisms in its runtime library. By default, *ANTLR* generates a parse-tree listener interface that responds to events triggered by the built-in tree walker.

5.3.1. Parse-Tree Listeners

To walk a tree and trigger calls into a listener, *ANTLR*’s runtime provides class `ParseTreeWalker`. To make a language application, we build a `ParseTreeListener` implementation containing application-specific code that typically calls into a larger surrounding application. *ANTLR* generates a `ParseTreeListener` subclass specific to each grammar with enter and exit methods for each rule. As the walker encounters the node for rule `assign`, for example, it triggers `enterAssign` and passes it to the `AssignContext` parse-tree node. After the walker visits all children of the `assign` node, it triggers `exitAssign`.

In CodeDOG, we rely heavily on parse-tree listeners. Let us consider the function `rectangle_area()` from Figure 1.1, as we previously discussed, we would call this a **block**. In this block, we have a couple of information we want to collect:

- Function’s return type
- Function’s parameters

The visualization facilitates the programming process, *ANTLR Lab* [47] is a great tool that aims at experimenting and testing grammars with an input. By taking a look at the visualiz-

ation of the parse-tree (Figure 5.1), we can easily recognize the rules that we would need to write the listeners for (Table 1).

Function name	directDeclarator
Return Type	typeSpecifier
Parameters	parameterList

Table 1: Function information items and their corresponding rules

From each rule, we can extract whatever information that we need and tokenize it to our specific usage. In our case, we implemented listeners to listen on entering a `functionDefinition`, from there we can identify all the pieces of information from the respective rules as seen in Table 1. Then we map this information to our `codedog-common` classes respectively. Similarly, `structs` are handled the same way, we just listen to `structOrUnionSpecifier` as the context instead, and then we handle the mapping. Moreover, we also implemented a custom `ErrorListener` which parses the errors that can be encountered while generating a parse-tree; for instance, a missing semi-column. This enabled us to statically test the code first for any syntactical errors before compiling.

Furthermore, the usage of parse-trees would enable us to provide code metrics to estimate the code complexity by analyzing the depths of sub-trees and other techniques. This would be the perfect reliable solution for an infamous problem with C codes, in addition to compilers and static code analysis tools, yet, the problem's complexity would make this a topic for another paper.

5.3.2. Unit Testing

A question might arise to the reader, why we did not use unit tests? Unit testing C code can be particularly challenging due to several factors. First, C code often relies on external libraries and system calls, making it difficult to isolate the code being tested. Additionally, tightly coupled code—where different parts of the program are closely interconnected—poses a significant challenge for unit testing. Legacy code, lack of built-in testing features, and resource constraints in embedded systems further complicate the process. Furthermore, handling segmentation faults and error conditions requires careful consideration during testing.

Unlike languages with built-in testing frameworks (e.g. Java with *JUnit*), C does not have native support for unit testing. However, developers have put time and effort into developing support for unit testing, a particular project from *ThrowTheSwitch* called *Unity* [48] was developed to test C codes. As much as it is a brilliant project, simple, and easy to set up—it is quite obscure and we do not expect instructors or users of our project to be familiar with

it, not to mention that it cannot be automated easily; in my opinion, that would also require another paper on its own.



Figure 5.1: Parse Tree for `rectangle_area()` from Figure 1.1

6. Static Code Analysis

Static program analysis is the art of reasoning about the behavior of computer programs without actually running them. This is useful not only in optimizing compilers for producing efficient code but also for automatic error detection and other tools that can help programmers [49].

The importance of static analysis, in our opinion and to our project, is perhaps the one road that we should take to enhance our application. In this chapter, we will give a very friendly and abstract theoretical background on static analysis and how can we benefit from it.

6.1. Applications of Static Analysis

Throughout the years, static analysis has been used to optimize compilers. More recently, it has proven useful also for bug-finding and verification tools, and in IDEs to support development.

Analysis for program optimization

A few examples of properties that we need to find out to generate efficient code[49, 50]:

- Does the program contain dead code, i.e., is a function or a block reachable from the `main`? If so, the code size can be reduced.
- Is the value of some expression inside a loop the same in every iteration? This would cause redundant computations
- Does the value of a variable depend on an input or can it be pre-computed at runtime?
- What are the bounds of a variable, i.e., what data are we storing in it? For example, we can utilize `short` for very small numbers—this would guide a choice of the representation of such a variable.

The runtime behavior of a software can be introduced as a formal definition, *semantics*. The **semantics** of a program is a description of its *runtime behaviors*. We call semantic property any property about the runtime behavior of a program. Hence, checking if a software will run as intended is equivalent to checking if this software satisfies a semantic property of interest. Program analysis can be applied wherever understanding program semantics is important or beneficial [50]. Software developers are the biggest beneficiaries, as they can use analysis for quality assurance, locate errors, and maintain legacy software. Language processors can also benefit from program analysis, to translate the input codes into optimized

programs. Automatic tutoring systems for teaching programming can use program analysis to hint students toward a direction to amend their faulty programs. We cannot claim that with coDExe and CodeDOG we have a complete tutoring system. However, it seems to be the goal with the development of such applications (or macroservices), nevertheless, it showed a huge optimization compared to the previous e-learning systems used at the University of Debrecen.

Analysis for program correctness

Analysis tools that have been designed to detect errors target generic correctness properties that apply to most or all programs written in specific programming languages. In unsafe languages like C, such errors sometimes lead to critical security vulnerabilities. In more safe languages like Java, such errors are typically less severe [49, 50]. Examples of such properties are:

- Does an input exist that leads to a null pointer dereference, division-by-zero, or arithmetic overflow?
- Are all variables initialized before they are read?
- Are arrays always accessed within their bounds?
- Can there be dangling references, i.e., the use of pointers to memory that has been freed?
- Are all assertions guaranteed to succeed? Assertions express program-specific correctness properties that are supposed to hold in all executions.
- Assure *typestate* correctness properties from specific libraries.
- Can the program deadlock? This is often a concern for multi-threaded programs that use locks for synchronization.
- What types of values can a variable have? This is vital for dynamically typed programming languages, for example, JavaScript, Python, and OCaml.

6.2. Approximate Answers

Regarding correctness, programmers routinely use testing to gain confidence that their programs work as intended. Ideally, we need guarantees about what our program may do for all possible inputs, and we want these guarantees to be provided automatically, that is, by programs. A *Program Analyzer* is such a program that decides whether or not other programs have a certain property.

Reasoning behind the behavior of programs can be extremely difficult, even straight-line programs are difficult to reason about. This problem has been conjectured for almost a century, until **Rice's theorem** came to formulation [50].

Theorem 1 *Let L be a Turing-complete language, and let P be a nontrivial semantic property of programs of L . There exists no algorithm such that, for every program $p \in L$, it returns **true** if and only if p satisfies the semantic property P .*

Let us simplify this a little, a Turing-complete system is a system in which a program can be written that will find an answer—although with no guarantees regarding runtime or memory—so, in principle, it could be used to solve any computation problem. Most modern programming languages are Turing-complete because they each implement all the features required to run programs like addition, multiplication, if-else conditions, return statements, ways to manipulate data (CRUD operations), and so on. Rice's theorem informally states that all interesting questions about the input/output behavior of programs are **undecidable**. This encourages approximative answers, while it is impossible to build an analysis that would correctly decide a property for any analyzed program, it is often possible to build analysis tools that give useful answers for most realistic programs with more precise approximations (colloquially referred to as *full-employment theorem*) [49].

6.3. CodeDOG's Future

While we do not intend to design our own static analyzer, we had to introduce it in theory first. Some other concepts like completeness and soundness are not going to be discussed in this paper. For our purpose, static analysis can be used for bug finding, type analysis, error search, and more.

In static analysis itself, lots of problems are *simplified* as it approximates conservatively semantic properties of programs before their execution [50]. We can incorporate lots of these tools in our implementation:

Clang Static Analyzer

The *Clang* C/C++ compiler comes with a static analyzer which can be used to find bugs using path-sensitive analysis. The path-sensitive analysis is a technique that explores all the possible branches in code and records the code paths that might lead to bad or undefined behavior, like uninitialized reads, use after frees, pointer leaks, and so on. It comes with a wide variety of checkers, for example, deadcode. [40]

Spoon

Spoon is an open-source library to analyze, rewrite, and transform Java source code. It parses source files to build a well-designed abstract syntax tree (AST) with powerful analysis and transformation API. *Spoon* provides a static analysis at the source code level using a processor that analyzes all AST elements of a given type, one by one. It also provides a Runtime Reflection API. [51]

Infer

Infer is a static analysis tool for Java and C/C++/Objective-C code. It produces a list of potential bugs that can be intercepted before shipping to production. Infer is deployed within Facebook and it is running continuously to verify select properties of every code modification for the main Facebook apps for Android and iOS, Facebook Messenger, Instagram, and other apps. It is simple, can be used from the command line and has a vast variety of analyses and issue types. [52]

7. Conclusion

We tested CodeDOG in one of the exams for *High-level programming languages 1* at the University of Debrecen. This test has shown us the result of this implementation, before showing us the benefits, it showed us the flaws. With this result, we can determine the road to take with the development, enhancements, and expansion possibilities.

7.1. Current Results

After heavy testing the server with multiple requests, it can handle it properly. However, due to some human errors, we noticed that sometimes the server can get stuck. Please notice that a production environment is very different from local testing, we have to adapt to the host's capabilities if we cannot get better specifications. After analyzing the errors, we deduced that we need to enhance the request handling to be asynchronous. Simply, asynchronous programming allows a computer to perform tasks without waiting for each one to finish before starting the next. In Java, this can be achieved with multiple libraries or with built-in thread operations. However, since we are using *Spring Boot*, this functionality comes out of the box with the magic of annotations and a few tweaks.

Furthermore, Docker is a robust tool, it facilitates the implementation of sandboxes but it uses too much memory, this can be controlled with more fail-safes for memory management and CPU usage. Another possibility is to migrate to use *alpine* [53] instead of *ubuntu*, the main reason is the massive reduction in the virtual image size compared to *ubuntu*.

The server also can be enhanced to store the input and test case specifications with higher reusability and adaptability, this can be achieved in multiple ways. Since we are using *MongoDB*, we can store the test cases and inputs in a separate document and reuse them accordingly. This would reduce the amount of data we are storing in a single Job, and allow faster modification and better auditing control.

On the bright side, the service's outputs facilitated the checking for the teachers, they can easily identify the errors as it is shown for them if the code passed all the checks required like: compilation, runtime output, and functions existence. We left the grading—the final result or verdict delivered to the student—to be a human job for now. We cannot suddenly introduce this tool to be the main grading judge for the students for various reasons; students need to get more familiar with the environment, and we have to account for some human errors like typing mistakes in variable or function names—which we had implemented but we decided to use strict equality for the exam—, and a psychological factor; students expect a fair grading and it cannot be otherwise under any circumstances. So before we can allow

this service to provide a grade, it has to be tested with more exams, due to the lack of time at the time of writing this paper, we only had one chance to test the service. For the time being, the service is a good indication and tool for its users, whether the teachers or the students, to have small feedback about their codes. As we showed, we provide concise but thorough feedback to the students including the warnings, runtime errors, and missing blocks in their codes.

7.2. Future Goals and Further Development Opportunities

As we previously mentioned, we need to enhance this service to account for the problems we encountered during the testing. However, these problems are not the end of this cycle.

At the moment, we only support C language source codes. For this particular language, we want to use more static analysis results from dedicated tools other than the standard `clang` static analysis that occurs at the compilation stage. We mentioned a couple of candidate tools that are easy to implement within our system and easy to use as well. We would love to add some code complexity metrics analysis as well, which is possible with the help of our parsing module.

The next steps include introducing the support for Java source codes. As an initial step, we plan to support one class exercises, i.e., exercises that can be solved with one Java class file with the help of static code analysis using tools like *Spoon* [51]. Then, we can have more complex exercises/exams to be handled. We can also introduce the support of unit tests with automatic generation, this would require having an internal *mavenizing* system, i.e., convert into a *Maven* project, and execute our analysis and testing services on it [54]. Not to mention, we would also plan to introduce support for Python, JavaScript, C++, and other used programming languages at the university, each with the appropriate approach.

At the university, there is an existing service that is used to evaluate SQL exercises [55], we used a similar architecture in CodeDOG as this service. It would be beneficial to migrate this service to be used as a CodeDOG module, hence, we can have a centralized evaluation service for multiple types of exercises/exams.

Finally, we also plan to provide an actual grading, not just feedback. This will only be possible after more testing, enhancements regarding code complexity and metrics, and most importantly the familiarization of the students with the platform. We have a huge database of programming problems [56] that we can use to make this service available to the students.

The modularity of CodeDOG's architecture allows infinite possibilities in terms of grading and code checking. The main purpose of CodeDOG, after all, is to provide this grading using an appropriate scientific background, whether it is static code analysis, parsing, or

even other techniques that we have not covered in this paper. In this paper, we presented—and insisted on calling—a prototype that we plan to slowly build into a much better product with the mentioned enhancements.

Bibliography

- [1] *Unified Modeling Language (UML) version 2.5.1*. <https://www.omg.org/spec/UML/2.5.1/PDF>. Accessed: 26/03/2024.
- [2] *CPython*. URL: <https://github.com/python/cpython> (visited on 24/03/2024).
- [3] Brian W. Kernighan and Dennis M. Ritchie. *C Programming Language*. Pearson, 1998. ISBN: 9780131103627.
- [4] *Standardization: Standard C++*. URL: <https://isocpp.org/std> (visited on 23/03/2024).
- [5] Noam Chomsky. ‘On certain formal properties of grammars’. In: *Information and Control* 2.2 (1959), pp. 137–167.
- [6] Alfred Aho, Jeffrey Ullman, Ravi Sethi and Monica Lam. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006. ISBN: 9780321486813.
- [7] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013. ISBN: 9781934356999.
- [8] *ANTLR*. URL: <https://www.antlr.org/> (visited on 23/03/2024).
- [9] Mayur Pandey and Suyog Sarda. *Llvm cookbook*. Packt Publishing Limited, 2015.
- [10] *Programmming Contests*. URL: <https://progcont.hu> (visited on 23/03/2024).
- [11] Tamás Kádek, Márk Kósa, Anett Orosz and Róbert Tóth. ‘Új fejlesztési irányok programozási feladatok megoldáskiértékelő rendszereihez’. In: *INFODIDACT 2017*. 2017. ISBN: 9786158060813.
- [12] *Safe Exam Browser*. URL: <https://safeexambrowser.org> (visited on 23/03/2024).
- [13] *Moodle Documentation - Quiz activity*. URL: https://docs.moodle.org/403/en/Quiz_activity (visited on 23/03/2024).
- [14] *CodeRunner Moodle plugin*. URL: https://moodle.org/plugins/qtype_coderunner (visited on 23/03/2024).
- [15] *JOBE*. URL: <https://github.com/trampgeek/jobee> (visited on 23/03/2024).
- [16] *IntelliSense in Visual Studio Code*. URL: <https://code.visualstudio.com/docs/editor/intellisense> (visited on 23/03/2024).
- [17] *check50*. URL: <https://cs50.readthedocs.io/projects/check50/en/latest/index.html> (visited on 24/03/2024).
- [18] *GitHub Codespaces*. URL: <https://github.com/features/codespaces> (visited on 24/03/2024).
- [19] *Windows Subsystem for Linux*. URL: <https://learn.microsoft.com/en-us/windows/wsl/about> (visited on 24/03/2024).
- [20] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson, 2017. ISBN: 9780134494326.

- [21] *Docker*. URL: <https://www.docker.com/> (visited on 24/03/2024).
- [22] *HackerRank*. URL: <https://www.hackerrank.com/> (visited on 24/03/2024).
- [23] *HackerRank Tests*. URL: <https://candidatesupport.hackerrank.com/hc/en-us/articles/4402913821843-Getting-Familiar-with-HackerRank-Tests> (visited on 24/03/2024).
- [24] Vladimir Khorikov. *Unit Testing Principles, Practices, and Patterns*. Manning, 2020. ISBN: 9781617296277.
- [25] Daniel Reynaud. *The Problem with Using HackerRank as a Programmer Screening Tool*. 2017. URL: <https://dreynaud.fail/the-problem-with-hackerrank-as-a-programmer-screening-tool/> (visited on 24/03/2024).
- [26] Márta Nagy. 'QR-kód alapú azonosítás és ráépülő alkalmazások oktatási célra'. Spring TDK 2023.
- [27] *UniPass Card System*. URL: <https://en.unipass.hu/oldal/promo> (visited on 25/03/2024).
- [28] *Guide to Working with Multiple Modules in Maven 4*. URL: <https://maven.apache.org/guides/mini/guide-multiple-modules-4.html> (visited on 25/03/2024).
- [29] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN: 9780201633610.
- [30] *MongoDB*. URL: <https://www.mongodb.com/> (visited on 26/03/2024).
- [31] *Clang Compiler User's Manual*. URL: <https://clang.llvm.org/docs/UsersManual.html> (visited on 26/03/2024).
- [32] *GCC online documentation*. URL: <https://gcc.gnu.org/onlinedocs/> (visited on 26/03/2024).
- [33] *Valgrind User Manual*. URL: <https://valgrind.org/docs/manual/manual.html> (visited on 26/03/2024).
- [34] *What is containerization?* URL: <https://aws.amazon.com/what-is/containerization/> (visited on 26/03/2024).
- [35] *What is containerization?* URL: <https://www.ibm.com/topics/containerization> (visited on 26/03/2024).
- [36] *Use containers to Build, Share and Run your applications*. URL: <https://www.docker.com/resources/what-container/> (visited on 26/03/2024).
- [37] *Java Docker API Docs*. URL: <https://github.com/docker-java/docker-java/tree/main/docs> (visited on 26/03/2024).
- [38] *Clang vs Other Open Source Compilers*. URL: <https://opensource.apple.com/source/clang/clang-23/clang/tools/clang/www/comparison.html> (visited on 26/03/2024).

- [39] *Clang - Expressive Diagnostics*. URL: <https://opensource.apple.com/source/clang/clang-23/clang/tools/clang/www/diagnostics.html> (visited on 26/03/2024).
- [40] Vince Bridgers. *Using the Clang Static Analyzer*. https://llvm.org/devmtg/2020-09/slides/Using_the_clang_static_analyzer_to_find_bugs.pdf. 2020 LLVM Developers' Meeting, Accessed: 27/03/2024.
- [41] *Grammars-v4*. URL: <https://github.com/antlr/grammars-v4> (visited on 26/03/2024).
- [42] James Gough, Daniel Bryant and Matthew Auburn. *Mastering API Architecture*. O'Reilly Media, 2022. ISBN: 9781492090632.
- [43] Mark Heckler. *Spring Boot: Up and Running*. O'Reilly Media, 2021. ISBN: 9781492076988.
- [44] *The Reactive Manifesto*. URL: <https://www.reactivemanifesto.org/> (visited on 29/03/2024).
- [45] Robert Sebesta. *Concepts of Programming Languages*. Pearson, 2015. ISBN: 9780133943023.
- [46] Theodore Norvell. 'A Short Introduction to Regular Expressions and Context Free Grammars'. In: *Memorial University of Newfoundland - Software Engineering 7893* (2002).
- [47] *ANTLR Lab*. URL: <http://lab.antlr.org/> (visited on 30/03/2024).
- [48] *Unity*. URL: <https://github.com/ThrowTheSwitch/Unity> (visited on 30/03/2024).
- [49] Anders Møller and Michael I. Schwartzbach. *Static Program Analysis*. <https://cs.au.dk/~amoeller/spa/spa.pdf>. Accessed: 31/03/2024.
- [50] Xavier Rival and Kwangkeun Yi. *Introduction to Static Analysis*. The MIT Press, 2020. ISBN: 9780262043410.
- [51] Renaud Pawlak et al. 'Spoon: A Library for Implementing Analyses and Transformations of Java Source Code'. In: *Software: Practice and Experience* 46 (2015), pp. 1155–1179. DOI: 10.1002/spe.2346. URL: <https://hal.archives-ouvertes.fr/hal-01078532/document>.
- [52] *Infer*. URL: <https://fbinfer.com/> (visited on 31/03/2024).
- [53] *alpine - Official Image*. URL: https://hub.docker.com/_/alpine (visited on 06/04/2024).
- [54] Bruno Pereira Cipriano, Nuno Fachada and Pedro Alves. 'Drop Project: An automatic assessment tool for programming assignments'. In: *SoftwareX* 18.101079 (2022).
- [55] *SQLe | Kiértékelő szolgáltatás*. URL: <https://codexe.inf.unideb.hu/sql-evaluation/> (visited on 06/04/2024).

- [56] Tamás Kádek, Márk Kósa and János Pánovics. *Informatikai versenyfeladatok*. URL: <https://gyires.inf.unideb.hu/GyBITT/04/> (visited on 06/04/2024).