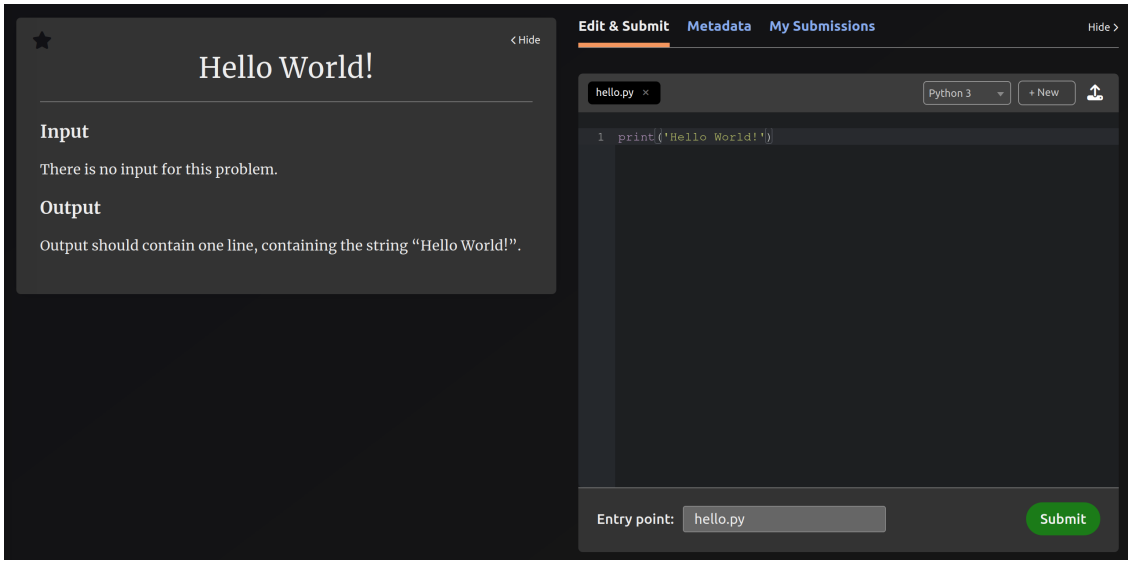
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Managing Code Evolution: Refactoring Challenges in Kattis' Code Judge

Design aspects of developing features in an online code judge

Bachelor's thesis in Computer science and engineering

Joshua Andersson
Alexander Gilabert
Rasmus Hulthe
Hugo Söderbergh

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Bachelor's 25

Managing Code Evolution: Refactoring Challenges in Kattis' Code Judge

Design aspects of developing features in an online code judge

Joshua Andersson
Alexander Gilabert
Rasmus Hulthe
Hugo Söderbergh

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg Sweden, 2025

Managing Code Evolution: Refactoring Challenges in Kattis' Code Judge
Design aspects of developing features in an online code judge

Joshua Andersson, Alexander Gilabert, Rasmus Hulthe, Hugo Söderbergh

Supervisor (Handledare): Mohannad Alhanahnah, Department of Computer Science and Engineering
Advisor: Fredrik Niemelä, Kattis
Examiner: Patrik Jansson and Arne Linde, Department of Computer Science and Engineering
Graded by teacher (Rättande lärare): Magnus Almgren, Department of Computer Science and Engineering

Bachelor's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Phone +46 31 772 1000

Cover: Problem "Hello World!" on Kattis.

Typeset in LaTeX

Gothenburg, Sweden 2025

Abstract

Online code judges are platforms that offer programming problems and assess submitted solutions based on correctness, efficiency, and other performance metrics. As online code judges evolve to support increasingly complex features, maintaining backward compatibility and secure system design becomes a significant challenge. This thesis investigates the practical implications of refactoring and extending the Kattis code judge to accommodate their new problem package format. By collaborating with Kattis, the project focusses on developing a selected set of key features from their upcoming 2023-07 problem format. Through a combination of architectural design and modular programming, the thesis proposes a solution to handle both the upcoming 2023-07 format and the old legacy format. The work also includes solutions for the integration of static validators, using markdown for problem statements and multi-pass problems. The project resulted in the addition of 5,716 and deletion of 1,070 lines of code, of which 1,967 of the additions and 644 deletions were merged into Kattis' codebase. The results of the project highlight the nuances of creating maintainable systems, identifying sacrifices in simplicity that are made to accomplish modular designs. The project also gives insight into the special considerations that need to be addressed when developing features from the 2023-07 format.

Sammandrag

Online-koddomare är plattformar som tillhandahåller programmeringsproblem och bedömer inskickade lösningar till problemen utifrån korrekhet, effektivitet och andra mått. I takt med att koddomare på nätet utvecklas för att stödja alltmer komplex funktionalitet blir det en betydande utmaning att bibehålla bakåtkompatibilitet och säker systemdesign. I samarbete med Kattis kommer denna avhandling undersöka de praktiska följderna av att refaktorera och vidareutveckla koddommaren Kattis för att stödja deras nya problemformat. Projektet fokuserar på att utveckla ett urval av de viktigaste funktionaliteterna som introduceras i det nya problemformatet *2023-07*. Med hjälp av kod arkitekturell design och modulär programmering föreslås en lösning för att hantera både det gamla *Legacy* och det nya 2023-07 formatet med hjälp av . Uppsatsen innehåller även lösningar för implementationen av statiska kodvalidatorer, hur man kan implementera problemformuleringar i Markdown och implementera multi-pass-problem. Projektet resulterade i ett tillägg av 5,716 och borttagning av 1,070 rader kod, varav 1967 av tilläggen och 644 av borttagningarna lades till i Kattis' kodbas. Projektets resultat främhäver nyanserna i att skapa underhållbara system och identifierar de uppoffringar i enkelhät som görs för att uppnå modulära designer. Uppsatsen redogör även för saker att tänka på när man implementera funktionalitet från formatet 2023-07.

Acknowledgements

First and foremost, we would like to express our gratitude to Kattis for making this project possible and for their valuable feedback on our code. We extend appreciation to our project supervisor Mohannad Alhanahnah for his continuous support and feedback throughout the course of the project. We would like to thank Arne Linde for assistance with logistics regarding the project.

Contents

List of Figures

List of Tables

Glossary

Config - Short for *config*uration.

LOC - Lines of code.

Online code-judge - A website that will receive a problem-solver-submitted program, run it and provide feedback on whether it fulfils requirements.

Problem Package - A set of files defining a programming problem for use in an online code-judge.

Problem-author - An individual who creates problem packages.

Problem-solver - An individual who solves problems defined by problem packages.

Pull Request - A tool provided by GitHub allowing developers to propose merging changes from one branch into another.

Regex - A regular expression – regex for short – is a string searching algorithm.

Render - The process of compiling instructions into a graphical output, typically intended for presentation to end-users.

Repository - A place to store code and the version history of said code.

Sandbox - A virtual environment with limited access to operating system features.

Sanitise - The process of removing malicious content from a piece of media.

Submission - A package of files submitted to an online code-judge by a problem-solver to solve a problem.

1 Introduction

Implementing changing requirements in legacy codebases presents challenges in refactoring and system design, often exposing hidden security risks. Through a collaboration with the company Kattis, this thesis investigates strategies to update their code-evaluation platform from a single-format system to a dual-format system, focusing on design and security aspects. The transition reveals trade-offs between extensibility and simplicity as well as highlighting hidden risks that can arise from implementing seemingly harmless functionalities, such as XSS attacks via Markdown.

1.1 Background

Kattis is a company that hosts a *code judge*, a web-based platform to automatically run and evaluate computer programs [1]. The platform allows for users to submit code solutions, which are automatically run against a suite of test cases to verify correctness and efficiency. A code judge can be used by students to practice algorithmic problem solving, by companies to screen candidates, or by universities to grade students' homework.

Kattis is in the process of developing a new *problem format* for their code judge to allow for a set of new features. A problem format is the specification that defines how a problem is defined, for example how the problem statement is stored or how the submitted programs will be evaluated. The Kattis Problem format is used by Kattis and other code judges such as Domjudge [2], PC^2 [3] and Omogenjudge [4]. By creating a new format, changes which break backwards-compatibility can be applied, such as changing the metadata for problems to allow for more detailed information about credits to authors, translators and testers. Other features include allowing problem statements to be written in Markdown, which was previously only available in LaTeX, to provide an alternative that can be considered "easier" to use [5].

Implementing the new format leads to technical challenges. The code judge has previously only supported one problem format, which will become known as the *Legacy format* [6], but with the introduction of the new *2023-07* format [7], this will lead to significant structural changes to Kattis' codebase. This is because the code is highly specialised to handle the Legacy format, with existing systems being hardcoded for the Legacy format, such as loading and validating the metadata for problems. The task of implementing the 2023-07 format therefore forces the choice between duplicating logic, which would make the code less maintainable, or refactoring it into a more modular system, which requires a larger upfront cost.

1.2 Purpose

There are two primary aims of this project: 1. To investigate strategies about refactoring Kattis' codebase to allow for both the Legacy format and the 2023-07 format by utilising design patterns such as the factory pattern or dynamic dispatch. 2. Identifying security risks about features in the 2023-07

format, such as using XSS to exploit Markdown statements, and ways to mitigate them by performing input sanitisation.

The approach to accomplish this is to implement a selection of key features from the 2023-07 format, including validation of metadata and problem statements in Markdown, primarily in the open-source project *Problemtools* [8]. This project is managed by Kattis and contains tools for rendering problem-statements and validating problem-instances. Some features like multi-pass problems will also be implemented in Kattis' backend to gain further insights into additional security considerations.

The findings of the project give insights to developers of similar code judge systems about modular design and mitigations to security risks. This can lead to more maintainable, well-designed and resilient systems in future code judges.

1.3 Objectives

The first objective is to assess the qualities of the solutions used to solve the different features, which is done with a combination of metrics and analysis. Firstly, a simple measure of the number of lines changed is brought up to give an indication of the magnitude of changes. Secondly, a qualitative analysis is performed by manually judging adherence to the design principles DRY and SOLID. This provides a foundation to examine the effects that maintainable code has on code size.

The second objective of the project is to find and mitigate risks when implementing the 2023-07 format. This is done by firstly identifying potential attack vectors through manual code review. Then secondly suggesting mitigation strategies such as recommending input sanitisation libraries like `nh3`.

1.4 Limitations

The scope of the project is limited in two ways: 1. To manage the timeline of 18 weeks together with the emphasis of the project on design, it was decided that a representative subset of the features from the 2023-07 format, such as metadata validation and Markdown statements are to be implemented, which allows for a deeper analysis. 2. The majority of the work is implemented in Problemtools instead of Kattis' backend, due to a combination of factors. Problemtools' size of 3,500 lines of Python code allows for a more granular analysis compared to the 15,500 lines of Python code in the backend. The backend's proprietary nature impacts the transparency of the results, so by working with Problemtools, which is open to the public, conclusions can be scrutinised by a wider audience. Problemtools emulate the backend to validate problems[1], but to consider backend-specific topics such as sandboxing, some work was done in the backend as well, such as implementing multi-pass problems.

The 2023-07 format was still a draft version during the project. Future versions of the format might receive minor changes, but the projects overall conclusions about refactoring and security aspects

[1] As a part of problem validation, example submissions are verified in a way that represents how they will be tested when uploaded to the actual platform.

remain applicable to similar systems.

As a part of the project we, the authors, perform a qualitative analysis when evaluating adherence to design principles. This implies a level of bias, which is complemented through code reviews by Kattis' developers as part of getting features merged into the codebase.

1.5 Thesis Outline

This thesis is structured as follows:

- **Chapter 1: Introduction** – This chapter gives an overview of the project, the purpose behind it, the objectives, as well as the limitation set by the authors. It provides the conditions for the report and highlights key areas.
- **Chapter 2: Technical Background** – This chapter introduces the concepts necessary for a comprehensive understanding of the project.
- **Chapter 3: Method** – This chapter describes the tools and approach towards reaching the goals outlined in the introduction.
- **Chapter 4: Results** – This chapter presents the results achieved after development.
- **Chapter 5: Discussion** – This chapter interprets the results, discusses the reasoning process behind the decisions presented in results and the possible impacts of them.
- **Chapter 6: Conclusion** – This chapter concludes the report, presenting a comprehensive and compact overview of the findings.

2 Technical background

This chapter begins by describing the general workings of code judges to provide a comprehensive understanding of the subject. A description of the two problem formats follows, containing relevant context about the formats and a providing brief overview. The validation process of submissions is described to give a comprehensive understanding of essential parts of the problem formats. This is proceeded by the description of relevant features of the 2023-07 format that are to be implemented. Problemtools is described, with a focus on technical details relevant to this project's findings. Finally, design principles used for analysis and design are presented with a brief explanation to give important context for the analysis in chapter 4.

2.1 Code Judge

An online code judge is a website that contains challenges that can be solved by submitting code [9]. These challenges are often algorithmic in nature. The code judge runs the submitted code in a controlled environment, providing it input, and analysing what the submitted program outputs [9]. If the output is deemed to be correct, the problem-solver will receive a `correct` verdict; otherwise, they will receive feedback on why their code did not pass. Some popular examples of online code judges are Codeforces [10], LeetCode [11], and Kattis [1].

From a problem-solver's perspective, a problem consists of a problem statement, sample inputs, outputs, and test data. The problem statement outlines the scenario in which the question arises, specifies the constraints on the program's input and defines the expected output of the program to be submitted. The sample cases provide examples of inputs that can be provided to the program, along with their expected output. Finally, the test data consists of the secret test cases that verify that the solution fulfils the requirements as specified in the problem statement. An example of a published problem is `Hello World`, as shown in figure 2.1 [12].

Kattis' code judge, as illustrated in figure 2.2 provides problems through their backend servers to their users, the problem-solvers. The problems need to be authored by problem-authors, who require tools for validating that the problem is well-defined, that the test cases work as expected, and that the problem statement is displayed properly. This functionality is provided by *Problemtools*, which is Kattis' toolset for providing problem-validation and basic problem-rendering. Problemtools is also used by Kattis' backend for its use in verifying the validity of problems, along with additional checks and security measures, before they can be uploaded for the problem-solvers.

2.2 Problem Format

Each problem is defined by a *Problem package*. Each package consists of the files and folders required to describe the problem [13]. In order for the code judge to parse a problem package, there needs to be a standardised format, known as a *problem format*, that defines this layout structure. The existence of the problem format helps both the online judge in parsing the problem package, and the

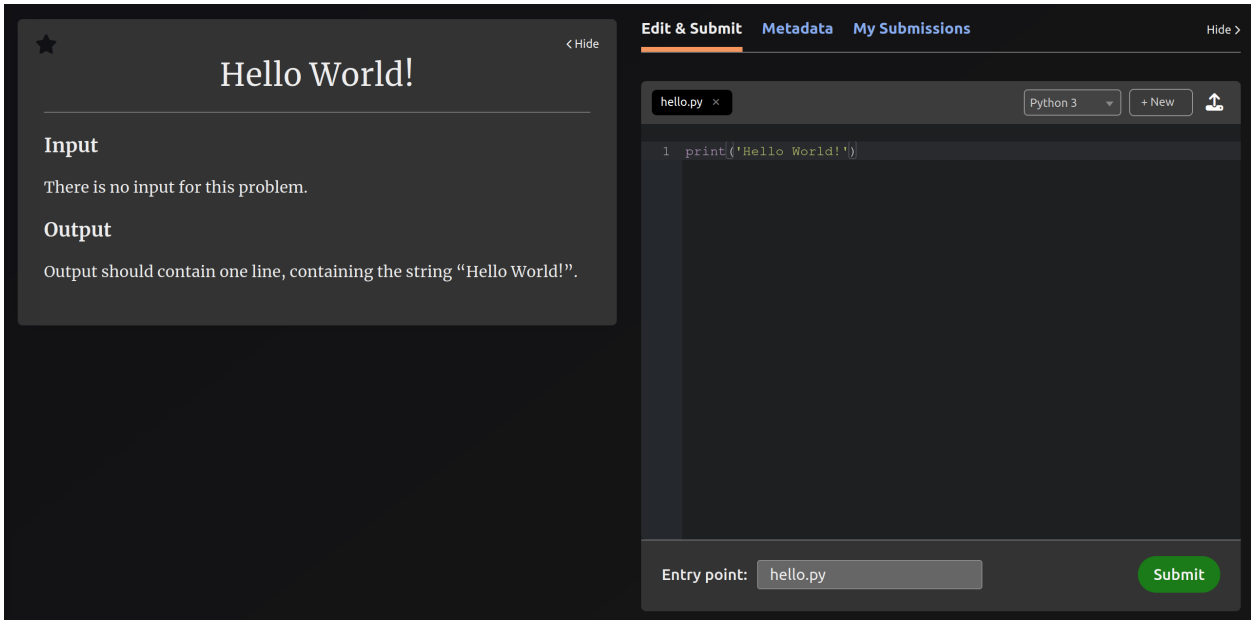


Figure 2.1: Problem "Hello World!" on Kattis. On the left side is the explanation of what the program should do, and on the right side is a code editor available in the browser.

problem authors by assuring them that their problem packages will work on Kattis' servers. Figure 2.3 illustrates the problem package structure for the Legacy format and the 2023-07 format.

2.2.1 History

Ever since Kattis was founded in 2010 [14], they have been developing a problem format. As the format developed, backwards compatibility was always ensured between updates. In 2023, Kattis decided to develop a new version of the format that would no longer be backwards compatible [15].

By not requiring backwards compatibility, enhancements and simplifications could be made. The earlier format is known as the *Legacy* format [6], and the new format is known as the *2023-07* format [15].

2.2.2 Overview

Through analysis of both the Legacy and 2023-07 format versions, the following core components were identified to be common to both format versions.

Problem Statement The text presented to problem-solvers, written in LaTeX.

Test Cases To verify that a solution works, multiple test cases will be specified by providing input data and answer data. These test cases are used by both the input and output validators.

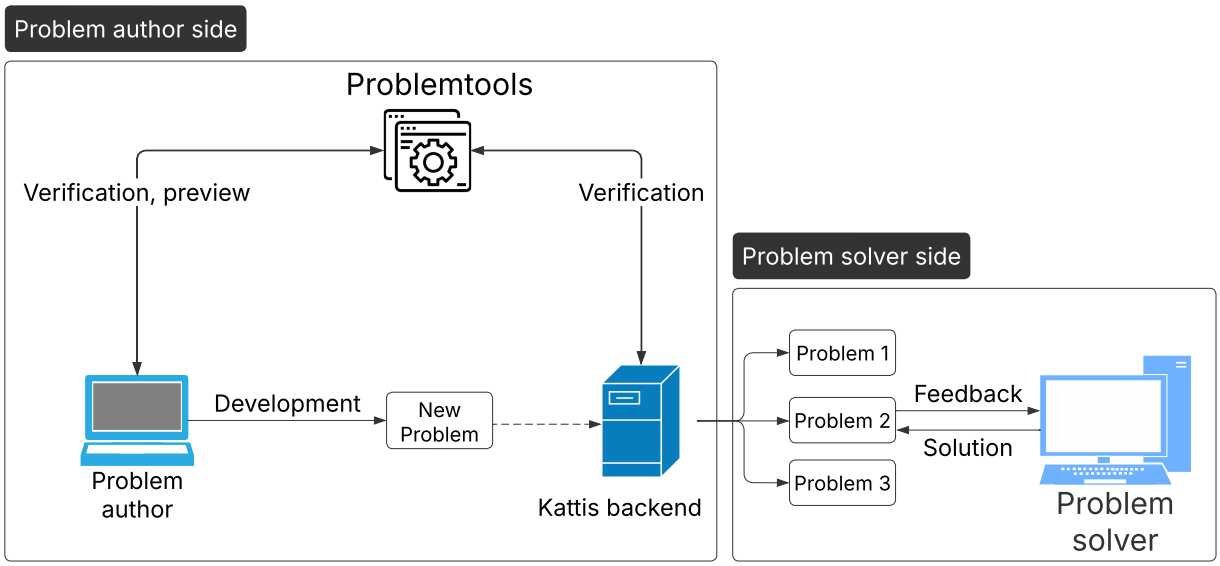


Figure 2.2: Figure of Kattis' interaction with problem-authors and problem-solvers. Both problem-authors and Kattis' servers will use Problemtools to verify the validity of the problem. The problem can then be uploaded to Kattis' servers and be accessed by problem-solvers, who can submit solutions and receive feedback.

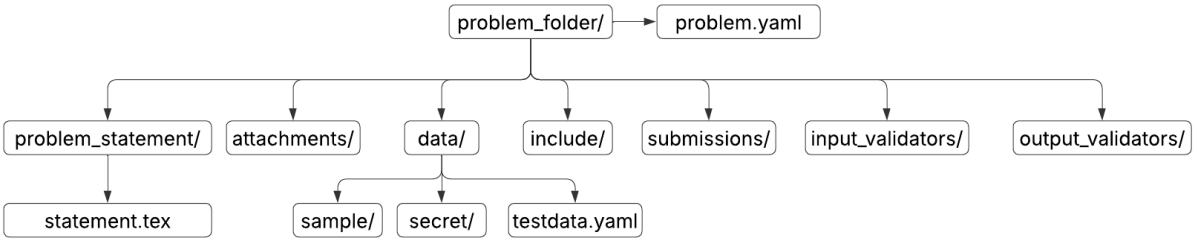
Output Validators To judge whether the output from a solution passes a given test case an output validator is used. The most common use case is to compare the solution's output, to the answer file – the output of the judge solution.

Input Validators To perform checks on each test case – making sure they can be parsed and does not contain invalid data for the problem – a program called an input validator is used.

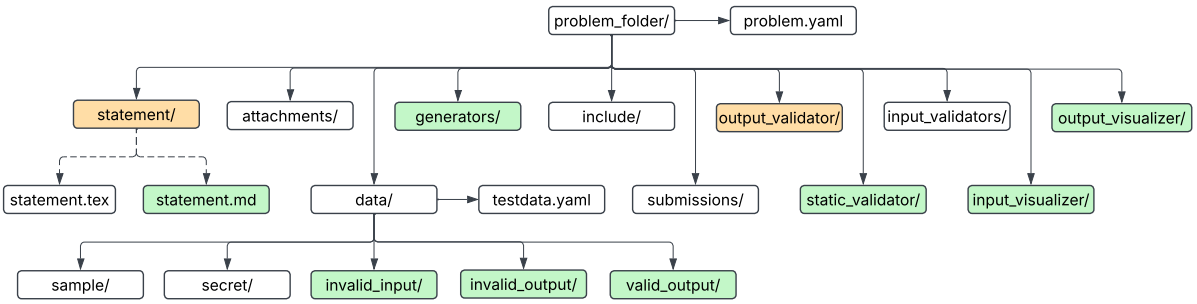
Graders[1] To combine verdicts of individual test cases or groups of test cases into one final verdict – for example by checking that all test cases passed – a grader is used.

Submissions Judge-written solutions to the problem. By running all submissions on the judging hardware and measuring their execution time, a time limit can automatically be computed, making the problem hardware-agnostic.

In addition to these parts, a file named `problem.yaml` is also present in the root directory of the problem package. This file is known as the *problem configuration* and contains technical options about how the problem is to be validated, such as if a custom output validator is provided or if a default output validator is used. It also contains metadata about the problem, such as who the problem-



(a) Structure of the Legacy problem package.



(b) Structure of the 2023-07 problem package. Green marks new folders, orange marks renamed folders.

Figure 2.3: Figures showing the folder layout of the Legacy format compared to the 2023-07 format.

authors are and the problem name.

2.2.3 Validation of submissions

In order to validate a submission, test cases need to exist. These test cases are made up of pairs of files, the input file and the answer file, with the file endings `.in` and `.ans` respectively [6]. The test data may be structured into groups and subgroups resulting in a tree-like structure, with the leaf nodes being individual test cases, as can be seen in figure 2.4.

The submission will be run on each of the individual test cases [6]. This means that the submission program will be executed, receiving the input file as the standard input[2], and outputting the answer to standard output. The output validator will consequently be run, receiving data from the input file, answer file and the output from the submission [6]. It will then signal to the judge system if the solution was correct or incorrect. It is also possible that the submission took too long to finish generating an answer, or that something unexpected happened when executing the output validator, in which case these errors will be reported [6].

[2]This is true for most problems, but if the problem is interactive, the standard input will actually be provided by the output validator instead.

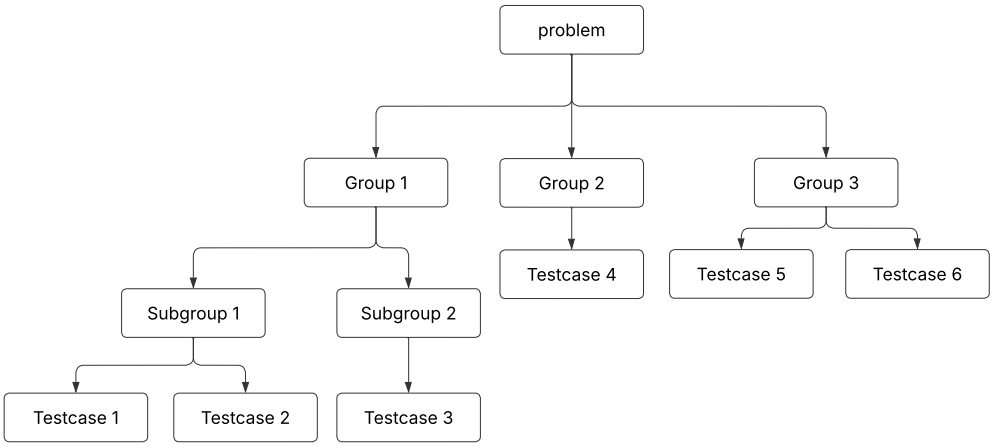


Figure 2.4: Visualisation of test case groups. The tree structure facilitates expressive groups of test cases and groups.

For each test case group, the grader will receive all results for test cases and subgroups and combine these results into one verdict [6]. This will be done for all groups until there is a single verdict for the entire problem, which will then be reported to the problem-solver.

Because most problems perform similar output validation and grading, there exists a standard output validator and grader which accomplish these common use cases [6]. The standard output validator compares the submission's output with the answer file. Depending on the configuration, it either requires an exact match, or allows a margin of relative or absolute error for floating-point results.

2.2.4 2023-07 format

The 2023-07 format builds upon the Legacy format by adding features and changing details about how parts work. The problem configuration has received changes in how information is stated, which often allows the problem-author to give more details than in the Legacy format. One example of this is how the information about the problem-authors is specified in the Legacy format compared to the 2023-07 format. In the Legacy format, a field named `author` was provided where the problem-author would state the authors as a string containing names separated by commas or the word "and". The 2023-07 format instead has a field named `credits` which contains the fields `authors`, `contributors`, `testers`, `translators`, `packagers`, `acknowledgements`. These fields in turn allow for multiple persons to be given for each aspect of the credits, where both name, email, orcid and Kattis usernames may be given for each person.

The 2023-07 format has specified a feature called static validators, whose purpose is to provide a verdict for a submission by analysing the submissions source code instead of its output. This is meant to be used in addition to the already existing output validator, to perform additional checks that will

allow for new types of problems, such as requiring the user to print "hello world" without being able to use the letter `h` in the source code of the program.

Most online judges evaluate submissions by executing the program once per test case, comparing the output against expected results stored in the test data folder [4], [8]. This single-execution model is extended in multi-pass problems, the same program may be executed multiple times with different inputs for a single test case. A crucial detail is that all these runs are independent, and no state is shared between runs. This approach enables novel problems, particularly ones where the challenge is related to implementing communication protocols. In such a problem, the code is executed in two passes. In the first pass, the submissions encode a message. The judge may then damage the message in some predetermined manner. In the second pass, the program decodes the potentially damaged message. While two-pass problems are the most common, the format supports an arbitrary number of passes, allowing for even more novel problems [15].

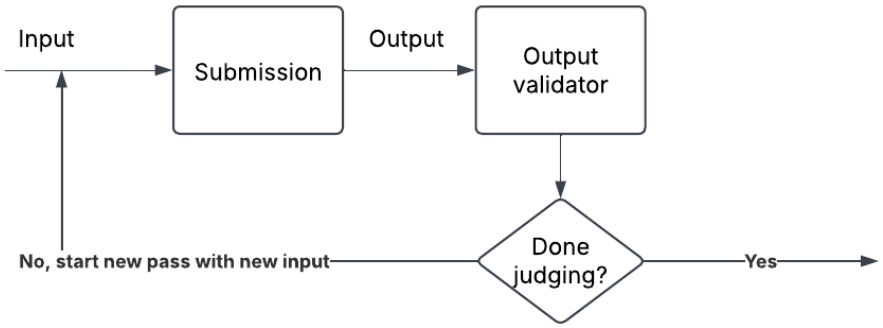


Figure 2.5: The control flow for running multi-pass problems. The output validator always produces a new testdata for the next pass, or gives a final verdict.

Stated more precisely, evaluating a multi-pass problem is a cyclic process, as illustrated in figure 2.5. First, the problem-solver's code – the solution – is run with input given by the testdata. The output of the program is then given to an output validator. The output validator may then signal that another pass should be run by creating the file `nextpass.in`. If created, the solution will then be rerun with `nextpass.in` as input. If `nextpass.in` is not created, the verdict of the output validator is considered to be the final verdict for the test case.

2.3 Problemtools

Problemtools is a suite of three Python programs developed by Kattis for problem creation and validation. The first of the three programs is verifyproblem [16], which is used to validate that a problem follows Kattis' Problem Format [13], [16]. The other programs are problem2html [17] and problem2pdf [18], which are used to preview how problem statements will render when uploaded on Kattis' website.

For contextual accuracy, Problemtools is described as it was prior to the modifications performed during the course of this project. The baseline version corresponds to the commit `05800557dddde9603e40306e53e9ec` dated 2024-11-11 [8]. This establishes a clear reference point for evaluating subsequent changes implemented during this project.

2.3.1 Problem2PDF and Problem2HTML

Problem2PDF and problem2HTML are both used to convert a problem statement to a document of type PDF or HTML, respectively. For both programs, the problem statement will first be preprocessed into a LaTeX file. This is done by starting with a template file that is filled in with the different parts of the problem statement such as the problem name and the sample test cases found in the test data.

When rendering to a pdf, the resulting output will then be converted to a PDF using the LaTeX compiler pdflatex [18]. When rendering to HTML, the statement will be given to the LaTeX compiler Plastex, which will render the LaTeX into an HTML-document [17].

2.3.2 Verifyproblem

Verifyproblem is a program written in Python that is around 2,000 lines of code [16]. The central class for verifying a problem is the `Problem` class, which instantiates a class for each part to be verified, such as validating problem statements.

These classes, which inherit the base class `ProblemAspect` to receive logging behaviour, perform the following checks:

Attachments Loads the attachments and makes sure there are no directories in the attachments folder.

Graders Ensures the grader programs are valid and checks that the problem is compatible to use grader programs if they exist. Provides a way for other classes to use the grader programs.

InputValidators Finds the input validator programs and makes sure they are valid. Runs the input validators on all test cases to make sure they are valid, and warns if input validators validate junk input.

OutputValidators Finds the output validator programs and makes sure they are valid. If output validators exist, ensures they should exist according to the problem configuration, and warns if they don't follow one of the recommended languages. Warns if the validators validate junk input. Provides functions for other classes to be able to use the output validators.

ProblemConfig Loads the problem configuration in `problem.yaml`, providing default values for some of the fields, and manually checks for some of the properties that should hold by using `if` statements.

ProblemStatement Finds all problem statements and makes sure that all the languages are unique. Runs problem2pdf and problem2html on each problem statement to make sure they are formatted in a valid way.

Submissions Loads all submissions from the submissions-folder and runs them against all test cases to verify that they get their corresponding verdict.

TestCaseGroup Checks configuration of test case groups and loads in test cases in the group. Provides some functionalities to interact with the test case group, like running a submission for the group.

Runner Manages the task of running test cases and test case groups. It keeps track of what program to run, which problem it is associated with, and the limitations for that problem. It has support for both single and multithreaded execution.

The validation starts with the `Problem` class creating a temporary directory to store intermediate results from verifying aspects of the problem. To resolve information-dependencies between verification-classes, they are instantiated in a predetermined order, and consequently stored as instance variables in the `Problem` class. Before checks are performed, all warnings and errors are reset, followed by the creation of an executor when multiple threads are used. Lastly, the classes are labelled into groups such as "validators", containing input- and output validators, and checked sequentially, reporting any errors to the user via the standard output stream.

2.4 Code Evaluation

An important part of this project was to evaluate the quality of software. This task is traditionally done in multiple ways, but static code analysis remains a central part. To help guide the analysis, adherence to common design principles can be checked which helps in order to perform a more methodic analysis.

2.4.1 Conventional methods

Evaluating the quality of code is traditionally done by experts with experience [19]. Although code analyser-tools have high potential, the knowledge, experience and judgement of experienced developers is indispensable for a reliable judgement. This means static analysis of the source code is a central part of evaluating the quality of software.

Another method of evaluating software quality of software is by using code metrics such as lines of code or cyclomatic complexity [19]. Because the evaluation using metrics usually involves relying on rigid thresholds, they are often insensitive to the specific context of the project [20].

2.4.2 Design principles

In order to perform static code analysis to evaluate the quality of code, the use of common design principles can be used. This project uses two widely used principles, DRY, "don't repeat yourself",

and SOLID, comprised of five principles for modular design [21], [22]. These were chosen due to their widespread use in the industry.

DRY discourages code duplication. In Kattis' context, shared logic, such as test case validation between Legacy and 2023-07 formats, should be centralised to ease future updates.

The SOLID principles are:

Single responsibility principle A class should only have one reason to change, for example the class loading the problem configuration should not also verify test cases.

Open close principle Modules should be able to be extended without changing the underlying implementation, for example via interfaces or inheritance.

Liskov substitution principle Subclasses, such as `InputValidators` to `ProblemAspect`, must be substitutable for their parent classes, meaning when `ProblemAspect` is expected, it should be able to use `InputValidators` instead.

Interface segregation principle Clients, such as problem validation submodules, should not depend on unused interface methods, such as Markdown specific methods in a generic problem statement interface.

Dependency inversion principle High level modules, such as problem validation, should not depend on low-level abstractions such as Legacy-specific problem configuration parsing.

3 Method

This chapter starts by describing the general workflow used to accomplish the projects goals, including design methods, code review and testing. This is followed by a description of tools used to ensure quality of code made, such as tools for static validation and continuous integration. Proceeding this are technical descriptions of how the parts of the project were implemented, which include: Markdown statements, static validators, problem validation refactoring, problem configuration parsing, and multi-pass problems.

3.1 Workflow

The workflow, as described in figure 3.1 consisted of several stages, which can be partitioned into two phases. The first phase is the planning phase, where features would be identified, selected and assigned to team members. The first stage, feature selection, consisted of comparing the 2023-07 format with the Legacy format to identify features and prioritise them according to availability and the perceived impact they would have on the code judge as a whole. A rough time estimate for each feature was also made based on analysing the existing code and estimating the magnitude of the changes required. In the second stage, the highest priority feature was given to a team member according to their availability and workload.

For each feature, the execution phase was carried out. The first stage in this phase was to identify and research relevant technologies to use in the design of the feature, either as libraries or as inspiration, such as JSON schemas for configuration parsing. Following the research phase, the team designed and implemented the feature by using the knowledge gained in the previous stage. As the solution was implemented, it would be tested and any issues would be fixed. After all tests passed and the code met internal quality standards, the team then created a pull request to Kattis' repository. Developers at Kattis and team members reviewed the code. If the code was deemed acceptable, the pull request would be approved and the code would be merged into Kattis' repository, else the feedback would lead to another iteration of development and testing.

3.1.1 Design

The project utilised the SOLID design principles to guide design choices, which have been shown to improve the quality of code [22]. These principles consist of the Single responsibilities principle, Open close principle, Liskov subtitution principle, Interface segregation principle, and Dependency inversion principle. DRY, which stands for "Don't Repeat Yourself" [21], is another principle that was used during the project.

The project also utilised common design patterns such as the factory pattern [23] to guide the design of systems. Design patterns help speed up development by providing design-level solutions for commonly occurring problems in software development [24]. They also aid in writing quality code since they are widely accepted solutions to these problems, and are recognised by many software developers.

3.1.2 Code review

After designing and implementing a feature, a pull request would be made to Kattis' repository. Developers at Kattis, as well as project members would give feedback on the code, and suggest changes to further increase quality and maintainability. This was an iterative process that would end in Kattis deeming that the code was at an acceptable level to merge into the codebase.

One common challenge during code review is a lack of documentation of the changes [25]. When making a pull request, a detailed message would be provided to ensure that the code review would be easier. This way, response time for pull request could be minimised which gave more time to focus on fixing the issues that arose.

3.1.3 Testing

To ensure that the code being produced met the existing requirements, it was tested in one of two ways. The first was that unit tests for specific behaviours would be created, and later run as a part of the continuous integration step when creating a pull request. This made sure individual components of systems worked as expected.

For some features, it was required to change the original implementation. To make sure that the code did not change functionally, it would be run against a large collection of old problem packages. That way, deviations between the results of the old version and the new version could be identified which indicated that there was an error in the code.

3.2 Tools

To successfully complete the project, several tools were used for version control, keeping track of tasks, and communicating both within the group and with the developers at Kattis.

3.2.1 Static code validation

Python is an interpreted language, which means that there can exist errors in the code that only trigger when the code is executed. If testing does not cover all code branches, there is serious risk that errors are first triggered in production. To mitigate this, multiple tools were used to analyse the code statically. The rationale behind executing so many different tools is that false positives can simply be ignored, while false negatives could be detrimental. The four tools used are as follows; First, Codefactor will run several tools to automatically scan the code for multiple types of issues. Second, Mypy was used to statically type check the Python code. It uses PEP484 to infer types, which can then be used to validate the type usages without needing to execute the code. Third, Pylint will scan the project's source code, analysing code style and searches for bugs, security vulnerabilities or anti-patterns. Finally, Flake8 serves a similar purpose as Pylint, but is less extensive.

3.2.2 Counting lines of code

In order to quantify the magnitude of the changes made in the project, the number of lines of code changed is measured. Unless stated otherwise, they are measured by using Github's automatically

generated line of code difference. This metric was chosen due to its ability to capture the effort in developing the system, including documentation and refactoring. In certain instances, a more exact view was required, in which case the command-line tool CLOC was used [26]. In these instances, it will be clearly mentioned.

3.2.3 Continuous integration

Continuous integration is a development methodology that is used along with version control systems. Every time new code is committed, multiple tests are run, ensuring errors or code quality issues are caught early.

During the course of the project, continuous integration was executed. To ensure code quality, Codefactor and Mypy were used. Cypress was used to automatically end-to-end test Kattis' frontend as part of CI/CD. Migra was used to compare the changes in new versions of the SQL schema and to generate the migration file needed to transform the old database schema to the new one. Finally, a suite of hand-crafted unit tests was also run by using Pytest.

Continuous integration tests were automatically executed using GitHub Actions whenever changes were pushed to the repository. This way, upon performing a pull request, these tools reported any immediate issues.

3.3 Implementing Markdown

In the Legacy format, problem statements could only be specified in LaTeX. In the 2023-07 format, statements may instead be provided in Markdown. At the start of the project, Markdown was not yet specified in the 2023-07 format, meaning that it became part of the project to specify the intended behavior. After specifying Markdown, it could be implemented.

3.3.1 Specification

Before implementing Markdown, it was necessary to define the intended behaviour. A particular challenge was the lack of a singular, strict specification for Markdown itself. If not clearly defined, there is a risk that some systems will use an uncommon flavour of Markdown, which would result in the system rendering many statements incorrectly. This was indeed a serious issue, as Fredrik Niemelä, one of the key contributors to both the 2023-07 and Legacy format emphasised in an interview that the format should be specified in such a way that the vast majority of valid problem packages should work on all systems implementing the Problem Package Format [27]. Therefore, multiple Markdown specifications were considered. The Markdown specification CommonMark was chosen due to its popularity, being used in projects such as Github, Reddit and Stack Overflow [28]. Even though it is very popular, prominent Markdown rendering libraries such as Pandoc and Python-Markdown are not CommonMark-compliant. Furthermore, after discussions with some problem-authors, it was determined that CommonMark alone is not feature rich enough to be able to express most statements. Therefore, the statement rendering system does not need to be fully compliant with CommonMark, but

must make a best effort to correctly render CommonMark-compliant statements. In order to exactly determine the necessary extensions to CommonMark, two approaches were used. First, discussions were held with existing problem-authors, where they were asked which features they commonly use. Second, several hundred existing Kattis problem statements were examined, where all visual elements not expressible in CommonMark were taken note of [29]. It was concluded that the extensions needed were tables, footnotes and support for rendering mathematical formulas. To specify how tables and footnotes should be formatted, Markdown Guide's extended syntax was used [30]. This syntax is not a specification, but a general description of the most common way that Markdown libraries allow the specification of tables and footnotes. This was chosen in the absence of a better alternative, as CommonMark does not specify tables or footnotes. In particular, they render correctly with Pandoc, the Markdown implementation used by Kattis. Finally, the implemention was required to support all mathematical formatting supported by MathJax. The motivation behind this decision is that both Kattis and Pandoc already use Mathjax, and it is widely supported. It was also important to mitigate security risks.

In order to reduce the security risks discussed in 3.3.3, the specification strongly recommends the use of sanitisers on any SVG files used, and the output of conversion from Markdown, such as PDF or HTML files. The reason to not specify the sanitisation more precisely, is as not to hinder the use of proper cybersecurity protection. At the same time, by mentioning the possibility of the use of sanitisers, problem-authors are informed that they should avoid writing Markdown that is unsafe enough to be potentially removed by sanitisers. The option of mandating that all HTML be interpreted as plaintext was also considered. Said option was not chosen, as Pandoc does not support this in a manner that ensures security.

3.3.2 Implementation

Kattis requested conversion from Markdown to HTML and PDF. To perform this conversion, the Pandoc library at version 2.9.2.1 was chosen due to its support for converting Markdown to HTML and PDF and its support for tables, footnotes and mathematical formatting [31].

To render Markdown to HTML, Pandoc is first used to convert the Markdown to HTML. Then, this HTML is inserted into an HTML template using simple Python string substitution. The template used will load in MathJax and display the problem title and id. Then, sample cases are inserted. The current behaviour in Kattis is that the samples cases are displayed below all content, except footnotes. To find where to insert them in the document, the code performs a search for the location of any footnotes if they are present. The system is not robust, as it only searches for a string that Pandoc currently generates for footnotes. A unit test was added to detect regressions caused by changes in Pandoc's HTML output format. Additionally, before samples are inserted, they are escaped using Python `html.escape` function. This is to avoid special characters such as `<` and `>` being parsed as HTML.

To render Markdown to PDF, a three-stage process is employed. First, Pandoc converts the Markdown

source into LaTeX. Next, the resulting LaTeX is processed. Because Pandoc does not support vertical lines in tables, a much-needed feature to be visually consistent, a simple regular expression is used to change all the LaTeX code for all tables to include vertical lines. Finally, the processed LaTeX is passed through Problemtools's existing LaTeX-to-PDF renderer. Rather than using Pandoc to produce a PDF file directly, this process serves two purposes: it results in statements that look more similar to traditional LaTeX statements and circumvents the difficulty in generating tables containing sample cases when going from Markdown to PDF. In fact, an initial non-robust draft for generating sample case tables resulted in over 100 lines of Python code.

The code was written procedurally without using object-oriented programming. The primary motivation for this choice is that during the rendering process mostly consists reading the statement source, then acquiring other data and using it to modify the statement. Thus, using object-oriented code for the different data sources such as samples, problem name, problem ID and sanitisation would presents a trade-off, where flexibility would be improved at the cost of greater complexity and amount of written code. It was argued that it's important for Kattis to maintain a consistent visual style across statements, and therefore, changes to statement code should be avoided. In fact, most statement rendering code had not changed in over 12 years. Therefore, the extra flexibility was deemed unnecessary.

3.3.3 Security in Markdown Processing

The goal of the following security analysis is to ensure that the problem statements in problem packages submitted by problem authors cannot compromise the integrity of Kattis' servers or its users. Kattis does thoroughly check the contents of problem statement before installing them. Their contents are rendered both in user browsers and once in the Backend during problem installation. As such, a systematic threat analysis was carried out with the aims of identifying and categorising attack vectors in Markdown rendering and conversion. Additionally, the security measure implemented for each issue is described and justified.

To structure this analysis, a threat modeling approach inspired by OWASP principles was applied [32]. The primary threat actor considered is a malicious problem author who is already trusted by Kattis, who then performs the attack by asking Kattis to install an otherwise valid problem package. The reason for this choice is because this is the only way to interact with the code related to problem rendering.

After the threat actor was identified, the potential attack surfaces were analysed in accordance with the "identify what can go wrong" question from OWASP. The analysis was carried out by analysing all reachable code related to problem rendering. The following attack surfaces were identified:

- **Frontend**: Possibly displaying unintended content, execution of code, or web requests in problem-solvers' browsers when rendering statements.

- **Backend**: Risks associated with converting Markdown to HTML or PDF, including exploitable

bugs in conversion tools that could allow code execution or external web requests.

In the frontend, the identified risk categories were consolidated into four primary types. The first category concerns cross-site scripting (XSS) attacks, where modifications to the appearance of the Kattis website outside the problem statement area could mislead users into clicking on links they would not otherwise follow [33]. The second category involves the execution of arbitrary code: if JavaScript is embedded within a problem statement, it may execute malicious code capable of compromising user data. The third category pertains to unauthorised web requests. If web requests are made to an attacker-controlled server, they could obtain the IP address of any problem-solver that loads the problem. Finally, the fourth category includes SVG files, which may contain arbitrary DOM content and, as a result, can serve as a vehicle for all of the aforementioned attack types.

To prevent XSS attacks, HTML and PDF were considered separately due to their differing characteristics. The primary risk in HTML was identified as XSS via malicious CSS or Javascript code from the problem statement. To mitigate this, the output HTML was sanitised using the nh3 Python library [34]. The library works by parsing the HTML, and then only keeping a small subset of HTML and CSS considered safe. The allowed subset was extended by the implementation to allow some HTML tags necessary to statements such as images and links, along with their associated properties. Additionally, some string were injected into the statement after sanitisation. To ensure safety, all such strings were escaped before insertion using Python's html.escape. The security was tested by compiling several malicious statements and ensuring all dangerous code was removed.

When considering PDF files, they were deemed not susceptible to XSS attacks, as the Markdown is converted to LaTeX before being converted to PDF. Because LaTeX does not support Javascript or CSS, it is removed when converting from Markdown to LaTeX [35]. However, as an additional security measure, all PDF files were sanitised using Ghostscript with the flag -dSAFER, including those generated by the Legacy format code for converting LaTeX statements to PDF. This decision was motivated by the fact that PDF files can contain Javascript, even if no exploit using this fact was found. A preventive approach was taken due to the intrinsic complexity of PDF files and associated tooling, together with the difficulty in completely ruling out this class of attacks.

Contrary to PDF files, SVG files present a very real and tangible threat. Sanitising SVG must be done separately from the sanitisation using nh3, as they are not embedded in HTML. Multiple SVG sanitisation libraries were considered, but they all deemed either insufficiently safe, or too large for Problemtools. For example, DOMPurify seemed safe enough, but required adding NPM, a significant dependency. Instead, the following sanitisation scheme was devised: SVG files are to be converted to the Magick Vector Graphics using ImageMagick, and then back to SVG. As MVG does not support scripting, this removes any embedded JavaScript or CSS while preserving the visual content. This approach was tested and showed promising results initially. By advisory from Kattis, this was not implemented due to the effort required to thoroughly test this approach. To avoid publishing code with a huge security vulnerability, SVG is disallowed until SVG sanitisation is implemented.

To prevent Markdown image tags from referencing external resources, the statement is first converted to the simpler format JSON using Pandoc. By traversing the JSON recursively, all images are found. Then, the validity of each image source is validated by ensuring that it points to a valid file path in the system, and that the extension of said file is either .png, .jpg or .jpeg. Notably, .svg is disallowed, as discussed in the last paragraph. Crucially, it is checked that this file path matches a strict regular expression that all files in a problem package must match [15]. The 2023-07 format states that all directory names must match the RegEx in listing 3.1.

```
^[a-zA-Z0-9]([a-zA-Z0-9_-]{0,253}[a-zA-Z0-9])?$
```

Listing 3.1: The Regex that all folder names in a problem package must match.

Additionally, all file names must match the RegEx in listing 3.2.

```
^[a-zA-Z0-9][a-zA-Z0-9_.-]{0,253}[a-zA-Z0-9]$
```

Listing 3.2: The Regex that all file names in a problem package must match.

In particular, these RegExs disallow any image sources containing the character ":", which excludes many kinds of URLs for which web browsers will make requests to external websites. The other kind identified were those that start with `//`. These were deemed to not be a threat, as `//` specifies a path beginning at the root of filesystem. It was deemed exceedingly unlikely that an attacker would be able to control a website that has the same URL as the link to an image file on the filesystem. Despite how unlikely the threat is, it was patched by checking that image source is relative, not absolute, forbidding those that start with `//`.

The risks in the backend identified primarily stemmed from the conversion of Markdown to HTML or PDF on Kattis' servers. It was deemed that nh3 and Ghostscript did not pose any risks, while Pandoc had a huge security flaw. Pandoc will automatically attempt to download externally linked images when converting directly from Markdown to PDF. No configuration option was found to disable this behaviour. This attack is already mitigated by instead using Pandoc to convert from Markdown to LaTeX instead.

3.3.4 Testing

The Markdown rendering system was primarily tested by running it on existing Markdown statements, ensuring that common usages work well. The statements were sourced from coding competitions, where both the problem statements, along with their respective Markdown source is released publicly, such as the IOI and EGOI. After rendering the source using Problemtools' Markdown rendering system, the outputs were inspected visually. The code was tested on more than 30 such statements. The tests were used to both test the correctness of the system, but also ensuring that outputs looked good. They were mainly examined using the following four criteria; First, each element should take an appropriate

amount of space in the page. For example, a 5×5 table or an image occupying an entire page would be considered incorrect. Second, the statements should render close to the original document in layout. Third, the statements should not contain obviously visually incorrect renderings, such as displaying raw LaTeX mathematical formatting. Finally, most subjective of all, they should look visually pleasing. Furthermore, the statements of some existing problems were manually translated into Markdown, both by the authors and people external to the project, and then compared to the LaTeX version. These were examined similarly to the IOI and EGOI statements, but a heavier emphasis was placed on the output being similar.

Additionally, unit tests were written to ensure that the security measures correctly sanitised malicious problem statements. There also were tests for XSS attacks, Javascript injection attacks and web requests. To obtain XSS examples, previous experience was used together with asking ChatGPT [36]. They were then manually validated to actually work. Some of which turned out to be non-functional, but some were achieved XSS in non-trivial manners. There were also tests to ensure that footnote detection and escaping special characters in the sample worked. The unit tests were run on 135 different versions of Pandoc to ensure robustness. The primary reason to test that many versions is to ensure that the code for footnotes works as intended.

3.4 Static Validator

A new feature introduced by the 2023-07 format was the static validator. Unlike some other components that require modification to existing code, the static validator presented an opportunity to implement a system that adheres to design principles with minimal dependence on other modules. This allowed for the emphasis on a modular, maintainable and clean design.

3.4.1 Design goals

The first goal of the design was to maximise the reuse of already existing code. In this context, this refers to the identification of similar features and using their implementation to construct a system that adheres to previous design choices.

The second goal was to create a system that was easy to maintain. To achieve this, a modular design was kept in mind. A consequence of a modular and maintainable design will be the facilitation of future extensions.

The third goal was testability and readability. This would be achieved by adhering to the SOLID principles [22]. By focusing on testability and readability, the system facilities understanding for future developers and makes it easier to discover errors.

3.4.2 Implementation

First, it was identified that the static validator needed code to implement a validation process and the ability to handle static validation test cases. To fulfil the goal of code reuse, the output validator was analysed to generalise the validation process and reuse the structure of the existing code.

As seen in figure 3.2, the process can be split into the following three phases: perform checks, executing the static validator and parsing and returning the feedback.

The validation process diagram was then used as a foundation to create skeleton code for the static validator. The implementation started with the code for output validators in order to reuse code. Then methods were modified to handle the static validator. To ensure that the static validator is provided correctly in the problem package, checks were implemented to compile the static validator, checking for errors.

The validation process was implemented in the validate method in accordance to already existing structure. Consequently, the code for parsing for the and possible feedback is reused. Parsing is done by examining the exit code of the static validator, the codes 42 and 43 signify an Accepted and Wrong Answer verdict respectively [37]. The method to acquire the feedback was then called in the parsing method, after which the judgement and feedback was returned from the validate method.

Static validators are invoked by defining static cases, as specified in the 2023-07 format [38]. Static test cases are implemented by providing the key `static_validation` in the file `test_group.yaml`, which may be provided for each test group. Under the static `static_validation` key, the inner key `args` may be provided, which will be passed as command-line arguments to the static validator.

To test implementation, a static validator was implemented for the "Hello World!" problem [39]. The validator only accepts solutions using the print function. This was done by searching the submissions source code for the string "print(", and only accepting solutions where it was present. To further test the implementation, this string was varied to search for the equivalent functions in solutions written in C++ and Java. To ensure that the command-line arguments were passed as intended, a check was implemented in the static validator asserting that the right amount of arguments were passed. Improper invocation of the static validator was then performed to verify that the checks indeed failed, and no edge cases were missed.

3.5 Refactoring Problem Validation

The `Problem` class was tightly coupled with the Legacy format, preventing its use for the 2023-07 format. As an example the 8 classes validating aspects of the format where manually initialised and assigned to instance variables to the `Problem` class. To solve this, a new design was made with the main purpose of making the `Problem` class more extendable, while keeping the present behaviour.

3.5.1 Design goals

The first objective of the new design was to increase the modularity of the `Problem` class to achieve the objective of making it more extendable. Here modularity refers to the reduction of coupling between the `Problem` class and a specific format, such as validating specific behaviours from the Legacy format.

The second objective was to increase clarity in the code. In the old implementation, it was unclear

what dependencies existed between the different parts of the format, as only the initialisation order was apparent from looking at the code. In order to protect against bugs involving initialisation order, making it clear what dependencies existed, such as test cases needed to be loaded after the problem configuration, was also a priority.

The third objective was to maximise the amount of code reuse in order to follow the DRY principle. There existed some repeated behaviours across all validation classes in the old codebase, such as setting up the name for logging of that part. With the new design, the aim was to reduce this to enforce consistency over all problem parts.

3.5.2 ProblemPart abstraction

An abstract base class `ProblemPart` was created to be used as a base class for all classes validating a part of a problem format. It was decided that all `ProblemPart` classes should need to define a part name, such as "statement" for the problem statement, which can be referred to for logging as well as to use as an identifier. This allows for an increased amount of code reuse which is the third objective.

Inheriting classes should define functions `setup` and `check`. The `setup` function can be used to load data relating to that part such as the data in the problem configuration. The `check` function is used to provide feedback on the validity of the state of that part. In addition to this, the function `start_background_work` could be used to utilise multithreading to speed up the `check` function, which was taken from the previous implementation.

To align with the second objective of clarity regarding dependencies between classes, a function `setup_dependencies` could be overwritten to return a set of classes that were depended upon. This information could then be used to ensure that all the dependencies are initialised first, allowing all the required data to be available in the `setup` method.

3.5.3 Changes to Problem class

The objective of modularising the `Problem` class was achieved through loading `ProblemPart` classes dynamically at runtime by using the strategy pattern. This way, different formats could be loaded by providing two different collections of classes to the `Problem` constructor, as long as the classes within the collections were compatible.

By utilising the information about setup dependencies, the `Problem` class loaded the classes and executed their setup methods in a topologically sorted order. To avoid classes accessing each other directly, the classes would return information as a dictionary from the `setup` function, which could be accessed by calling the `Problem.get` function with the class that had the information as argument.

Alternative approaches were considered but ultimately dropped due to the benefits of the chosen one. One other approach was to create an abstract base class for a problem, which would be overridden to specify the Legacy and 2023-07 format. This has the benefit of being simpler, as there is no need

to use topological sorting to initialise classes, and it is also easier to perform static validation since nothing is loaded at runtime. There are, however, major downsides such as reduced clarity and more code duplication in the form of loading similar classes in both specialisations which made the chosen approach more appealing.

3.6 Problem Configuration Parsing

Issues were identified in the old approach to validate the problem configuration about the thoroughness of checks and the quality of the code, as well as the output from the parsing step. The old code handled the verification of the problem configuration by performing a sequence of manual checks via if statements, for example checking that the license was not public domain if a rights holder was provided. This approach has several flaws, for example repeating a lot of code, and making code hard to reuse. It also makes it possible to look over important checks that should be made, for example checking the types of all fields in the configuration, which was the case in the old code. Another problem was the quality of the output from the parsing step, as the processing of the fields in the configuration was sometimes lacking, such as strings not being properly parsed into floats.

3.6.1 Design goals

The first goal was to ensure a thorough validation of both the Legacy format and the 2023-07 format. To ensure thoroughness, all fields in the format needs to be checked so that they have the correct type, and that they satisfy the limitations stated in the specification. In addition, all fields should be parsed properly, so that the types listed in the specification are the same as the types received after loading the problem configuration. All fields should receive any default values listed in the specification, such as "type" in Legacy being defaulted to "pass-fail".

The second goal was that the output of the parsed configuration would be structured such that working with the loaded configuration does not require structural checks. The specification has fields where there are multiple ways to represent similar data with different types. One example of this is the "name" field in the 2023-07 configuration, where a dictionary is expected from language-codes to the respective problem-title in that language. Instead of providing a dictionary, the problem-author may instead provide a single string, which will be interpreted as the title in English. This leads to checks needing to be performed every time the "name" field is accessed by later parts of the code, which decreases maintainability by increasing code duplication. To handle this, the goal of outputting the configuration to a single representation was made, which in the case of the "name" field means always representing it as a dictionary after the configuration is loaded.

The third goal was to improve the problematic aspects of the old design, including ensuring thoroughness, improving reusability of code, and making it easier to reason about the checks performed. Thoroughness can be ensured by making it more difficult to miss checks such as type checks when parsing the configuration. Reusability and the ability to reason about what checks are performed can be improved by designing a framework to use for validating all problem formats.

The fourth goal was to provide clear error messages to users. When an invalid configuration is loaded, it should be possible for the user to identify what part of the problem configuration was invalid and how. This is important since the messages will be shown to problem-authors, who might not be entirely familiar with Kattis' problem formats, making it critical that issues are clearly presented with a cause and location in the configuration file.

3.6.2 Approaches

When researching validation of YAML documents, JSON schemas were identified as a common method for validating both JSON and YAML documents. A JSON schema is a specification document that can be used to describe the allowed structures for a JSON document. They can also be used to validate YAML documents since YAML is similarly structured to JSON.

The problem with JSON schemas for implementing problem configuration validation are the first, second and fourth goals. JSON schemas do not modify the input document by default, making it impossible to apply default values or restructure the document into a single representation. In addition, the error messages from using JSON schemas are not always user-friendly, especially when the schema becomes more complicated with the use of sub-schemas. This is because they do not state the intention behind the two sub-schemas, but rather express that the document does not fit either one.

The first approach, known as the *general approach*, was to design a system similar to JSON schemas that would fix the issues for the use case of parsing the problem configuration. This system takes a specification document with keywords similar to the ones used by JSON schemas, as can be seen in figure 3.3a, but provides different keywords that can be used to put restrictions on the document while expressing the intention behind them. One example of this is the addition of a "forbid" keyword, which can be applied when an field has a certain value such as when "license" is "public domain", "right_holder" can not be anything other than the empty string. The system can be extended by adding parsing rules, which are classes that specify how a field will be parsed, which can be specified in the specification document to handle the restructuring of problem configurations.

As the general approach turned out to be quite complicated, needing 31 classes and 941 lines of code, a simpler approach was proposed by Kattis, known as the *simple approach*, which was also implemented. The suggestion was to simply extend the original approach to perform the additional behaviours such as checking the types and restructuring the output. This solves the first, second and fourth design goals, and by identifying shared fields between the Legacy and 2023-07 formats, it is possible to partly accomplish the third design goal of code reuse by using inheritance.

3.6.3 Implementation of general approach

The general approach for configuration validation was inspired by JSON schemas but extended with a custom specification format to enable restructuring of input, enforcing constraints, and delivering clear error messages. This system revolves around a specification document that defines the expected

structure, types, default values, and inter-property constraints of the configuration file. Parsing is done in multiple well-defined stages: resolving RegEx-based `match_properties`, parsing and validating types, injecting external data, resolving copy directives, and performing constraint checks.

Each configuration field is described by a set of keywords, such as `type`, `default`, `parsing`, `alternatives`, `required`, and `match_properties`. These keywords determine how the value is interpreted, validated, and transformed. For instance, fields with the `alternatives` property can trigger conditions using `require`, `forbid`, and `warn` directives, allowing for complex interdependencies between configuration properties.

For increased flexibility of handling properties, a parsing rule can be assigned per property, enabling handling of polymorphic types. One example of this is converting both a string and dictionary representation of the "name" field to the same standardised output. Parsing rules also support dependency ordering, which is essential for fields like `rights_holder`, which should be defaulted to the author as long as the license is not in the public domain, requiring `license` to be resolved before `rights_holder`.

The matching system uses match-strings with a consistent syntax across basic types, namely string, int, float and bool, and is implemented using the factory pattern. This system is used to implement the `alternatives`, `require` and `forbid` keywords, as they all require a way to describe collections of values for basic types.

Default values and cross-field references are resolved using a copy-from mechanism, evaluated in topological order to guarantee valid resolution. This makes it possible to assign a default to one field by copying its value from another field, which can be useful for example when copying `rights_holder` from the problem-author.

The system is accessed primarily via a `Metadata` class, which handles loading and verifying configurations using the specification. Supporting components like the `Path` class help traverse nested structures, and the matching and parsing subsystems ensure modular, extensible behaviour.

This architecture ensures that configurations are thoroughly validated, consistently structured, and easier to maintain or extend for new formats.

3.7 Multi-pass Problems

A major addition to the 2023-07 format is native support multi-pass problems, a problem type where problem-authors can run submissions in multiple independent phases. The problem type is explained in further detail in 2.2.4. This section details the technical implementations of the feature in the backend, highlighting how it improves over the constraints of an existing workaround to implement multi-pass problem in the Legacy format. Multi-pass support was also implemented in Problemtools, but is not discussed due to their similarity with the implementation in the backend. Key challenges

faced were language-agnostic compatability, information leakage across passes and integrating the feature into Kattis' existing codebase for executing code in sandboxes.

3.7.1 Legacy workaround

The Legacy format does not support multi-pass problems natively. However, via clever usage of the include folder, it is still possible to implement multi-pass problems. The first instance found of this workaround is in the problem *Avian* from the Nordic Olympiad in Informatics 2017 [40]. The problem is implemented as follows; In both Legacy and 2023-07, there exists a folder include. Before submissions are compiled and run, all files from the include folder are copied to the compilation and run folder. If subdirectories are specified, their contents is only copied if its name matches the programming language used for the submission to be judged. Using this, problem-authors can strategically create files that will take precedence in compilation or during runtime in order to ensure that execution is started in code written by the problem-author. To do this in C++, it suffices to add a file which contains the main method. In Python, execution is started in the file `main.py` Python. This code, called the `runner`, will then locate the problem-solver's code and run it. Since the runner is written by the judges and can start the problem-solver's code arbitrarily, it is possible to implement the multi-pass functionality.

The approach was analysed for limitations that needed to be addressed in the new implementation, of which five were identified. These are the following.

First, a runner has to be implemented in every language that the problem-author wants to support. It is infeasible to implement a runner for all the 52 languages currently supported by Kattis [41]. This is in contrast to interactive problems, where every language can be supported by writing one program and communicating via standard input and output. From experience, writing runners in multiple languages is typically bug-prone, as the large volume of code increases the risk of making mistakes and makes it harder to perform code review. In addition, the error rate is likely to increase if problem-authors write runners in languages they are unfamiliar with.

Second, through testing, it was discovered that on Kattis' platform, it is possible exfiltrate the runner's source code in some instances. Problem-solvers' submission have read access to the folder `src`, where the submission's source code is placed, along with all files from the include folder. By simply reading the runner's source code and printing it, the source code would be displayed in the user interface. The reason for it being displayed is because when sample cases are failed, the incorrect output is shown to the end user. This was deemed a serious problem, as problem-authors might prefer to keep the inner workings of the runner a secret.

Third, the runner cannot directly give a verdict to the judge system, as it is not an output validator. The problem Avian solved this by having the answer to each test case be a long, random string followed by the input. As contestants do not know this string, they usually cannot solve the problem by simply printing the same thing as the runner would have. Notably, this does not hold if the runner's source

code is successfully exfiltrated, in which case the problem can be trivially solved.

Fourth, preventing the problem-solver's code from sending arbitrary data across passes presents a challenge. For instance, a problem-solver could write information to a file during the encoding phase, and later read it during the decoding phase. One explored mitigation is to ensure that the runner deletes all files between passes. However, this solution lacks portability, as different implementations of the format could expose arbitrary directories for writing. Moreover, alternative data transmission channels might be difficult to prevent, such as environment variables, or starting a new process containing the data. It is therefore considered infeasible for a problem-author to write a completely safe runner.

Finally, as mentioned earlier, for runners to work, they must know which file the judge system will start execution from. In Python's example, the specification claims that execution will start in the file `__main__.py` [41]. This is in contrast to what Kattis actually does, as it starts in `main.py`. Such differences between implementation and specification make it difficult for problem-authors to implement runners, which may hinder adoption of the problem package format.

3.7.2 Implementation

When developing the solution, it was decided that the five limitations presented in the 3.7.1 subsection had to be addressed. The 2023-07 specification states that multi-pass is to be implemented as an extension of output validators [15]. This design solves most of the issues presented above. Output validators communicate through standard input and output as they are language-agnostic. Additionally, the third issue is also solved, as output validators may issue verdicts directly to the judging system. Finally, issue five is solved, problem-authors don't need to create files with highly specific names to make multi-pass judging work. The implementation will need to address the second and fourth issues related to exfiltrating the runner code and sending data between passes.

The system is implemented as follows; To run both the submision code and the output validator separate instances of IOI isolate is used, an isolated virtual environment [42]. By running programs inside IOI isolate, access to most system resources is severely limited, ensuring that the submission code cannot interact with the output validator or send data between passes. The default configuration of IOI isolate is insufficient, as the only available files in the virtual environment are the default files of a Linux distribution. Thus, to run the submission or output validator, relevant files need to be copied into the virtual environment. Instead of explicitly copying the files, a reference to a folder in the main filesystem is instead created, using a process called mounting. This choice facilitates debugging and increases performance. By constantly creating new virtual environments, the second and fourth problems, exfiltrating runner code or sending data across passes, are resolved.

Next, the file structure used when judging the submission is described, as illustrated in figure 3.4. To store data related to the submission, a directory with name equal to the ID of the submission is created. Inside this directory, the subdirectory src is created, containing the files of the submission. For every pass, a subdirectory is created with name equal to the index of the current pass $(1, 2, 3 \dots)$, in which the file input.txt is created, the input data for the current pass. When the submission is run, it has access to exactly two of these directories: one is named `src`, from which data can only be read and is a copy of `{submission_id}/src`. The other is `run`, where data can both be written to and read from, and is a copy of the directory for the current pass. This is illustrated in figure 3.5. By constantly creating a new run directory, no data persists across passes. The reason for creating new run folders instead of deleting old contents is because deletion is potentially dangerous, as exploits using symlinks could lead to deleted external the submission directory. The output validators function very similarly, except that it also has access to a special directory called the feedback directory, which is explicitly not cleared between passes, allowing the output validator to have persistent state.

Additionally, the specification supports both interactive and non-interactive multi-pass problems. The exact definition of interactive problem is not very relevant, as it suffices to know that interactive problems are run slightly differently compared to non-interactive problems. In order to be able support both problem types, the template method pattern was used, where a base class `multipassbase.py` was created with abstract methods for running setup logic and the submission code. Then, subclasses `multipass.py` and `multipassinteractive.py` implemented said abstract methods.

Putting it all together, the code will first create the directories for the submission code and output validator. Then, it will repeatedly run the submission code followed by the output validator in separate sandboxes. If the output validator did create a `nextpass.in` file in the feedback directory, its verdict is read and judging is halted. If `nextpass.in` is created, a new pass is started, and it is given as input to the submission code in the next pass. This process is repeated until a verdict is reached or the maximum number of allowed passes is performed.

3.7.3 Testing

Because of the monolithic nature of Kattis' code for judging submissions, it was difficult to write meaningful unit tests. Therefore, testing was primarily performed by creating multi-pass and submitting solutions to them. This was done by hosting a local version of Kattis using a virtual machine. After searching, only one existing multi-pass problem using 2023-07 format was found, called Alternative Encryption from the programming contest NWERC 2024 [43]. As the problem is not interactive and interactive multi-pass problems needed to be tested, an interactive multi-pass problem was developed. It is called Interactive Communication, authored by Joshua Andersson [44]. Using these problems, several bugs in the implementation were discovered and fixed. The code was also thoroughly reviewed multiple times to ensure correctness.

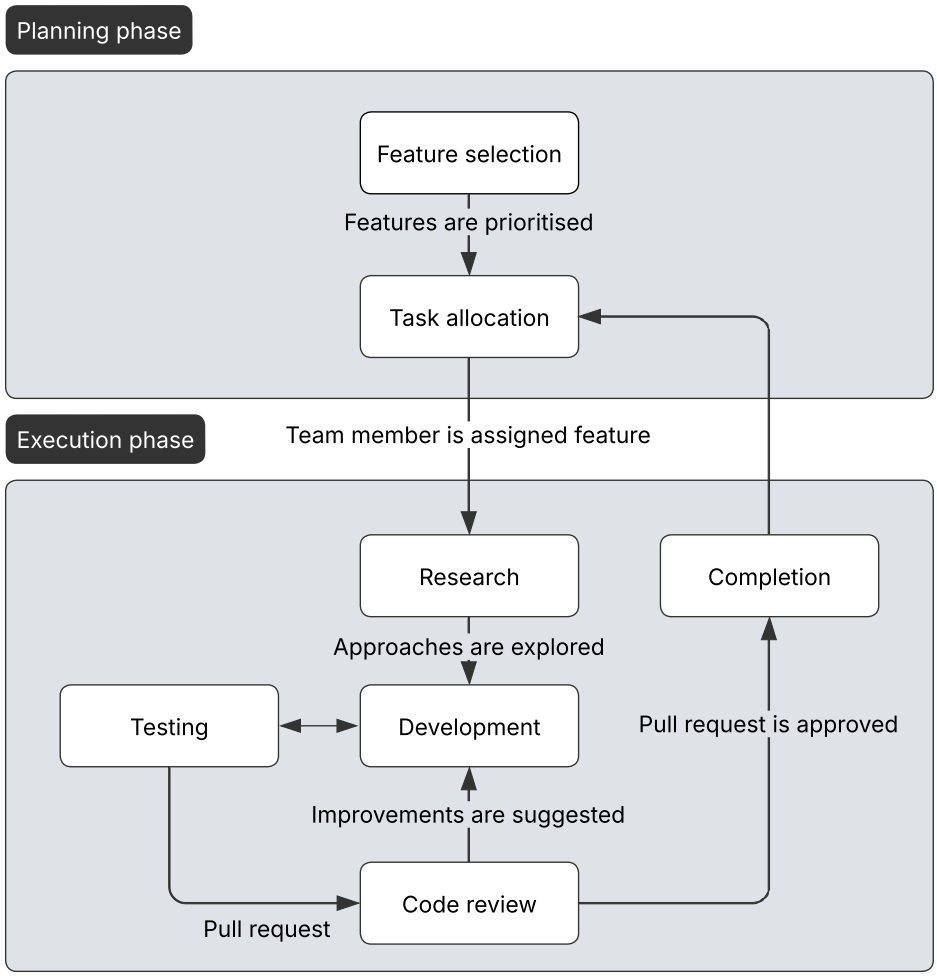


Figure 3.1: Flowchart describing the general workflow followed during the project.

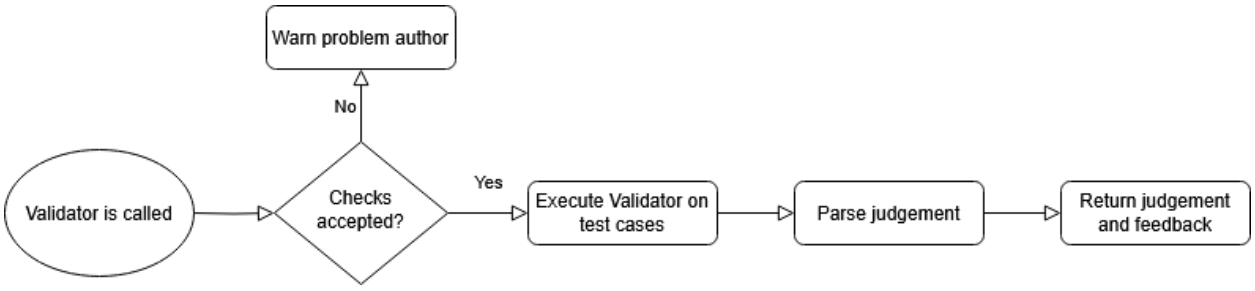


Figure 3.2: Generalised validation process implemented in the output validator.

```
# Specification document
type: object
required:
 - foo
properties:
  foo:
    type: int
  bar:
    type: float
    default: 3.14
```

(a) An example of a minimal specification document. Will check that configuration is an object with a property `foo` of type int, and optionally a property `bar` of type float. If `bar` is not present in the input, it will be present in the output with a value of 3.14.

```
# Configuration document
foo: 13
bar: 0.3 # optional
```

(b) Example of configuration document that will validate against the specification in 3.3a

Figure 3.3: Minimal example of inputs to the new configuration parsing system.

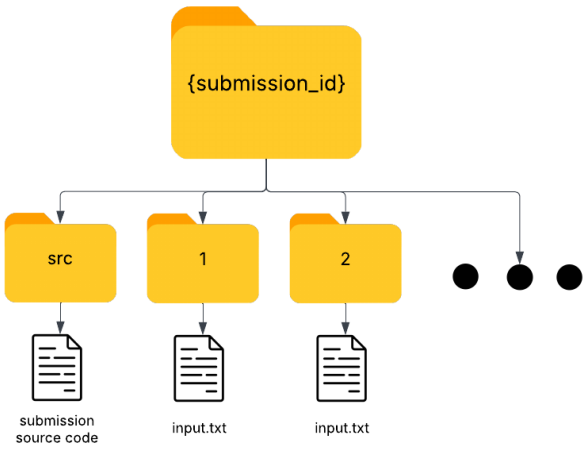


Figure 3.4: The file structure used to judge a submission. The directory src is always created. For every pass, one additional directory is created.

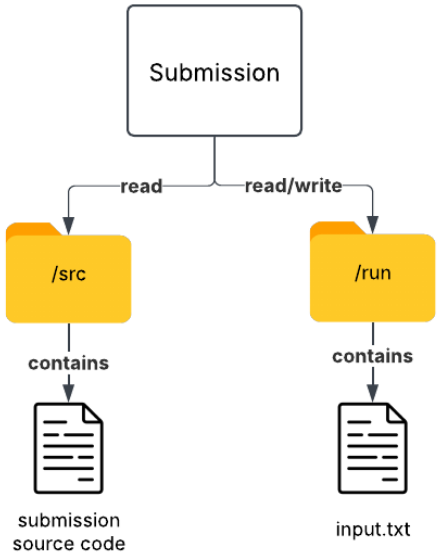


Figure 3.5: The files available to a submission when run. The `/run` directory is the only directory accessible for writing to the submission.

4 Results

This chapter presents the findings of the project. To start, the testing of the parts is presented to demonstrate the validity of the systems created. Following this, the findings about security for Markdown are presented. Finally, the solutions for static validators, problem validation, and configuration parsing are evaluated using design principles and judging metrics such as lines of code. A summary of the number of lines of code changed in total can be found in tables A.2 and A.3 in the appendix.

4.1 Testing

To verify that the code produced worked as intended, testing was performed by writing unit tests and running the implementations on several existing problems. In addition, the code was reviewed by experienced developers at Kattis before being merged into the codebase.

4.1.1 Problemtools problem abstraction

Since the function of the code was supposed to remain the same after the changes of the `Problem` abstraction, the main testing that was done was by running the program against existing problems to ensure that the same result was given as previously. In addition to this, one unit test was executed that existed since before the project. Problems that were tested can be found in the appendix A.

The code also passed the continuous integration tests, which include static code validation via Mypy as well as checks provided via Codefactor. Finally, the code was reviewed by developers at Kattis and approved to be merged into the codebase after minor updates.

4.1.2 Configuration parsing

To verify the validity of the general approach parsing system, a total of 26 unit tests were written and passed.

To further verify the system, it was tested by verifying 355 different existing Legacy problems created between 2017 and 2025 as seen in appendix A. The code passed the continuous integration tests, including static code validation with Mypy.

The simple approach passed the already existing unit test, as well as succeeding the 355 existing Legacy problems that were created between 2017 and 2025, as seen in appendix A.

4.1.3 Markdown statements

To ensure robustness, the implementation was tested in several ways. To test for logic errors, a total of 9 unit tests were created. All unit tests passed on over 135 Pandoc versions. To ensure the quality of rendered statements, over 30 statements were rendered and visually inspected. When rendered to HTML, all statements gave satisfactory results in regards to the size and placement of elements and were visually appealing. A comparison of an original and re-rendered IOI statement can be found in

appendix A. Results were poorer when rendering to PDF; in particular, statements rewritten from LaTeX to Markdown by individuals external to the project produced undesirable outputs. First, when statements are manually converted from LaTeX to Markdown, any used images will likely need to be manually rescaled, as Markdown does not support changing the image size. Additionally, some tables would be split up across pages and others would be missing vertical lines. In one instance, a table was missing horizontal lines, and the row order had been changed. Apart from tables, other visual elements rendered in the right place, took the right amount of space and were visually appealing.

The feature resulted in an addition of 1092, and a removal of 66 lines of code. After several rounds of feedback with Kattis, the code was merged into Problemtools. This was facilitated by following the DRY principle, which led to two major decisions. First, common logic between rendering to HTML and PDF was extracted into a common file called statement_util.py. Second, by reusing existing code to render LaTeX to PDF by converting the Markdown to LaTeX, much code duplication could be avoided. For example, the need to write difficult sample formatting code was entirely avoided. Analysing the code quality using all 5 aspects of SOLID is not relevant, as the code is written mostly procedurally. Despite being written procedurally, some principles still apply. First off, the Single responsibility principle was followed by composing the larger action of rendering a statement into smaller functions with a clearly defined purpose. Doing this allowed the functions to be unit tested and were easier to argue about in isolation. Analysing the rest of the principles is irrelevant due to the procedural nature of the code.

4.1.4 Static validator

The testing performed demonstrated important insights regarding expected behaviour of the implementation and its potential limitations. Through the systematic evaluation of the behaviour of the static validator, tests confirm that the implementation could handle user input files of multiple languages and worked properly. Specifically, by varying the string used when searching the source code it was asserted that the implementation was not dependent on any specific language. The implementation of checks on arguments in the created static validator ensured that proper invocation was implemented in the static validator class.

The limitations of testing include stress testing as well as unknown handling of more complex behaviour. Considering that the testing only was performed on small solutions there is no certainty that the static validator class would be able to handle larger solutions.

Another aspect of the limitations of the conducted testing was the lack of testing of more advanced syntax. A consequence of not performing tests regarding these aspects is the inability to conclude that the static validator class can handle these. Two examples of more advanced behaviour would be checking the for the existence of a properly implemented for-loop or switch-case in the source code.

4.2 Markdown Cybersecurity Attacks

When implementing the security features of the Markdown rendering system, several attacks were discovered. The system is resilient to all attacks that were found and those that are presented below in this section. Therefore, the system is considered fully secure based on current knowledge. Additionally, by adding Ghostscript sanitisation to all generated PDF, both from Markdown and LaTeX, the system as a whole is more secure than before the project. Although no known attacks were prevented using this extra sanitisation step, it adds another layer of security, hardening the system to future attacks. SVG files will not be shown in particular, as any attack listed below can be trivially embedded into an SVG.

4.2.1 XSS attacks

The first type of attack shown are those using statements embedded XSS. All were verified to work if they were not sanitized. The important keywords are colored blue. In all of the following instances except the one using the problem folder name, the Javascript can be replaced with CSS. The first type XSS attacks considered use the statement itself to inject XSS. Multiple examples of such XSS payloads are shown in listing 4.1.

```
<script>alert("Hello world!");</script>
<a href="#" onclick="alert('XSS')">Click me</a>
<svg onload=alert('XSS')></svg>
<a href="javascript:alert('XSS')">Click me</a>
<script>eval('\x61\x6c\x65\x72\x74\x28\x27\x58\x53\x53\x27\x29')</script>
<svg><script>alert('XSS')</script></svg>
<iframe src="javascript:alert('XSS')"></iframe>
<math><mtext><script>alert('XSS')</script></mtext></math>
```

Listing 4.1: XSS in statement payloads.

The second type of XSS attacks considered were by using strings injected into the sample after it was sanitised. The first string injected is the problem name. An attack using this is shown in listing 4.2.

```
problem_format_version: 2023-07
name: <script>alert("XSS");</script>
```

Listing 4.2: The problem.yaml file for XSS via problem name. While a UUID is technically necessary for it to be a valid problem.yaml file, it is omitted for sake of clarity.

The second string injected is the name of the problem folder. Not all approaches work in this instance, as folder names in Unix may not contain the character /, which is needed for the closing tag. If no closing tag was included, the resulting code would therefore be invalid – containing invalid Javascript – and thus no XSS. To circumvent this, an XSS payload that does not require a forward slash ('/')

was used, and is shown in listing 4.3. Notably, even if this string was not escaped, no payload was found that passes the RegEx applied to all folders in a problem package, defined in listing 3.1.

```
<svg onload=alert('XSS')>
```

Listing 4.3: The folder name for a problem which results in XSS.

The third and final type of strings injected are the sample cases. An example of such a sample is shown in listing 4.4.

```
<script>alert("XSS");</script>
```

Listing 4.4: The sample.in file for XSS via samples.

4.2.2 Web requests

If no checks are made on the image sources, the Markdown statement in listing 4.5 will make a web request when displayed in a web browser after it has been rendered.

```
![](https://upload.wikimedia.org/wikipedia/commons/a/a9/Example.jpg)
```

Listing 4.5: A statement which will make a web request.

If the folder RegEx defined in listing 3.1 is not used, but it is checked that the file pointed to by the source exists, it is still possible to perform a web request. This can be achieved by changing from `https://` to `http:` in the URL. The latest versions of Firefox (version 138.0.3) and Google Chrome (version 136.0.7103.114) will still make a web request, even with both forward slashes removed. Using this, a problem package can be structured such as in figure 4.1, which matches the source shown in listing 4.6. Thus, when Python checks whether a file exists corresponding to the image source, the operating system will claim that it exists. At the same time, when the HTML is loaded by a web browser, a web request will be performed.

```
![](https://upload.wikimedia.org/wikipedia/commons/a/a9/Example.jpg)
```

Listing 4.6: A statement which will make a web request if the problem package is structured as figure 4.1.

To circumvent the RegEx, the prefix `https:` can be replaced with `//`, resulting in the statement in listing 4.7.

```
![](//upload.wikimedia.org/wikipedia/commons/a/a9/Example.jpg)
```

Listing 4.7: A statement which will make a web request without using a colon.

4.2.3 Pandoc web requests

If Pandoc is used to convert from Markdown to PDF, it will automatically make a web request to all links present in images. It can be triggered by creating a file named `statement.md` containing the statement in listing 4.5 and then running `pandoc statement.md -o out.pdf`.

```
imgrequest
├── problem.yaml
└── statement
    ├── https:upload.wikimedia.org
    │   └── wikipedia
    │       └── commons
    │           └── a
    │               └── a9
    │                   └── Example.jpg
    └── problem.en.md
```

Figure 4.1: The folder structure necessary for the statement in listing 4.6 to pass validation.

4.3 Static Validator

The static validator is a new feature in the 2023-07 format. Due to its similarity with existing features such as output validators, the feature could be implemented without significant existing code. The additions and deletions are 184 and 6 respectively.

The static validator partially achieved its goals. The first goal of code reuse was partially achieved by the generalisation of the existing validation process. This was measured by the static validator sharing 100% of its methods with the Output Validator. The second goal, achieving a modular design, was partially achieved, as the system design does not depend on any other module. The third goal of creating a readable and testable design was also partially fulfilled by writing code with a clear focus on adherence to programming principles and modularity.

The reuse of existing code decreased development time and ensured consistency with existing design and structure. The consistency in style and design facilitated understanding for developers familiar with Problemtools. This ease of understanding facilitated acquisition of relevant feedback, allowing for faster code iteration.

The validate method violates the Single responsibility principle, as it handles the entire validation process: it reads the static validator flags from config files, invokes the static validator and parses its output. These spread responsibilities lead to a decrease in the code's readability. This issue stemmed from the generalisation of the validation process present in output validators.

The Liskov substitution principle is adhered to in the implementation of the static validator. This was verified by executing verifyproblem. Because verifyproblem makes usage of storing subclasses of the superclass ProblemPart for all parts of the validation process, including static validators, any trivial non-adherence to the Liskov substitution principle would be noticed. The adherence was further ensured by performing a comprehensive analysis, the result of which is that the ProblemPart could be substituted by the static validator.

In conclusion, the reuse of code led to the violation of several SOLID principles, primarily due to the assumption that the code was already following the principles.

4.4 Multi-pass

This section discusses the magnitude of changes and then analyses the code. First, multi-pass was implemented in both Problemtools and Kattis' backend. In the back-end, the implementation added 409 and removed 42 lines of code. The code was merged into Kattis' codebase after careful reviews. In Problemtools, the implementation resulted in an addition of 162 and removal of 43 lines of code. The multi-pass code in Problemtools was not considered for merging, as it depended on the new config parsing system, which ended up not being merged. The dependency was very weak, as the code only required for some system to be able to parse the metadata files in the 2023-07 format.

Using the template method pattern resulted in the code following DRY. By separating the code responsible for running submissions and output validators with the logic for creating directory hierarchies, moving files, checking for errors, and reporting verdicts, code duplication was avoided. In addition, adherence with the Single responsibility principle and the Open close principle is increased, as any subclasses wishing to introduce new ways of judging need only implement that logic. Furthermore, the Liskov substitution principle is upheld, as subclasses are designed to be fully interchangeable with the base class, maintaining consistent behaviour throughout the system. The Interface segregation principle and Dependency inversion principle are not very well-followed, as the methods depended on were created with the subclasses in mind. As an example, in `multipassinteractive.py`, the method `populate_submission_directory` simply does nothing, as interactive problems don't need to create an input file, while non-interactive ones do need to create it.

4.5 Problem Validation

The generalisation of the validation process for problems in Problemtools resulted in 346 lines added and 216 lines removed. This increases the size of Verifyproblem 130 lines, resulting in a 7% increase in code size.

The changes made by introducing the `ProblemPart` base class made it possible to generalise shared behaviour by for example setting the logging name for each part in the `ProblemPart` constructor, which was previously setup in similar ways across different parts. Repeated behaviours over multiple parts being reduced is beneficial according to the DRY principle.

The Single responsibility principle is slightly improved by letting the `Problem` class be more focused on managing `ProblemPart` classes. Previously, the `Problem` class contained logic to set up data related to validating the problem, such as containing variables for if the problem is interactive. This logic was moved to `ProblemPart` classes to keep the `Problem` class more focused on handling the setup and validation of `ProblemPart` classes. Adherence to the Single responsibility principle may be further improved by separating hardcoded checks such as validating the problem name and validating symlinks.

The Open close principle has received significant improvements via the introduction of the strategy pattern to the `Problem` class. By instantiating all the problem parts manually in the `Problem` class as was done previously, the behaviour could not be extended without directly modifying its behaviour. This has been solved by providing the `ProblemPart` classes to the constructor of the `Problem` class, allowing it to manage an arbitrary collection of `ProblemPart` classes, as long as they are compatible. This makes it possible to load multiple problem formats using the same `Problem` class, without changing the underlying implementation of the class itself.

Liskov substitution principle is followed, as there are no child classes that are not substitutable by the parent class. This is mostly due to the lack of a class hierarchy in the resulting code, because all inheritance is done via abstract classes. The previous code was already adherent to this principle however, so this is not an improvement to the previous version.

The Interface segregation principle is followed, because interfaces did not require irrelevant information to be implemented. `ProblemPart` was the only interface created, and required that a part name was specified. This was relevant for all `ProblemPart` classes, which means the principle was followed.

The Dependency inversion principle also received significant improvements by the introduction of the strategy pattern. Because the `Problem` class is a high-level abstraction of a problem to be validated, handling problem parts directly without a layer of abstraction is problematic according to the principle. This is fixed by the introduction of a `ProblemPart` abstraction, which makes sure the high-level `Problem` class does not rely directly on the more granular classes which verify aspects of the problem.

In conclusion, the more generalised code received significant improvements according to the DRY and SOLID principles, at the cost of a 7% increase in code size. This is further verified by Kattis' decisions to merge the changes into the codebase. Although the code grew in size, this can be partly attributed to the fact that some code was also created to start handling the 2023-07 format, such as creating command-line options to select format.

4.6 Configuration Parsing

Two approaches were tested for parsing configuration files: The general approach and the simple approach. These will be compared with each other to find the potential benefits and drawbacks of using the different systems. Although this gives a comprehensive view of the differences for their use

for specifically problem configuration files, it may be noted that the general approach has additional use cases, such as test case configuration files, meaning it has a wider use than the simple approach.

4.6.1 Complexity

The simple approach is significantly smaller than the general approach with only 508 lines of code compared to 941, counted using cloc, making the simple approach 85% larger than the general approach. In addition, the general approach also needs a YAML document to specify the layout of the configuration file, which is 173 lines for the Legacy format and 213 lines for the 2023-07 format. Including these, the general approach requires 1327 lines to be able to parse configuration documents for both formats, which is 161% more than the simple approach.

The general approach is also more complicated than the simple approach. One way to see this is to count the number of classes in the respective systems. In the simple approach, there exists a total of 3 classes, whereas the general approach is made up of a total of 31 classes. Complexity can contribute to systems being harder to maintain, as it requires more time for new developers to understand the workings of the system.

4.6.2 Adherence to design principles

The simple approach struggles to follow the DRY principle, as the fundamental design is to perform all the checks manually via if-statements, which are similar. The general approach is less problematic according to the DRY principle; Except for a null check that is performed at the start of the parsing functions in 16 out of 18 of the classes inheriting `Parser`, there is not a lot of repeated behaviour compared to the simple approach.

The Single responsibility, Liskov substitution and Interface segregation principles are all followed in both systems. This is because all classes in both systems have a single responsibility, use inheritance in acceptable ways, and do not rely on large interfaces.

The Open close principle is adhered to by the general approach, but is not followed at all in the simple approach. In the general approach, different configurations can be loaded by swapping out the underlying specification document, and can be further extended by implementing classes inheriting the `Parser` base class and adding them to the system. This is different compared to the simple approach, where it is impossible to load another configuration without rewriting large parts of the code.

The Dependency inversion principle is not adequately followed by the simple approach, but is followed by the general approach. This is because the level of abstraction in parsing a configuration document is higher than in checking individual fields, although, differing opinions may exist on the matter. In contrast, the general approach handles the task by performing parsing rules on a tree structure, which is seen as a more fitting level of abstraction for that level.

4.6.3 Summary of the two approaches

Both systems have certain flaws and benefits. The general approach is more complicated, which can lead to it being less maintainable. However, the general approach is more flexible and extendable than the simple approach and adheres better to design principles, making it more maintainable than the simple approach.

Ultimately, neither of the systems were accepted by Kattis' developers for use in Problemtools, opening the possibility that a better option might be possible than the tried approaches.

5 Discussion

This chapter presents commentary about the findings of the project, first discussing the effectiveness of the generalised problem validation. The general approach for configuration parsing is brought up to show that using DRY and SOLID might not always optimise maintainability. An evaluation of the solution proposed for static validators is made, along with potential improvements that could be done for testing them. Advantages and disadvantages are identified in the approach presented for Multi-pass problems. Tools used for rendering Markdown are evaluated, and security risks are summarised, along with especially difficult challenges. Finally, recommendations on future research about the 2023-07 format is suggested, covering natural continuations and topics outside the technical aspects of the format.

5.1 Other Design Principles

In order to evaluate the quality of code, the project used the design principles SOLID and DRY, but there are more design principles which were not evaluated. SOLID and DRY were chosen due to to their wide recognition, making them suitable for a quality analysis. If the project used a different set of principles such as KISS, "Keep It Simple Stupid", or YAGNI, "You Ain't Gonna Need It" it is possible that other results could have been reached.

Evaluating code quality is fundamentally a difficult task, making it one of the areas of improvement for future work. Utilising more metrics such as cyclometic complexity in addition to the static analysis could contribute to a more objective analysis of the code. This however comes at the cost of being able to implement less, as more time needs to be spent on the analysis itself.

5.2 Problem Validation

The project found that the strategy used to generalise the problem validation process significantly improved maintainability according to SOLID and DRY, at the cost of a 7% increase in code size. Kattis accepting the changes supports the notion that the increase in code size and complexity was a reasonable trade-off for the benefits of adherence to design principles such as increased modularity and extensibility in this use case.

The solution presented can therefore be viewed as an acceptable approach to create the framework of validating problems. Future works could explore the possibility of other approaches to handle the validation of formats, which may be simpler than the one presented in this thesis.

5.3 Configuration Parsing

The results indicate that both the general approach and the simple approach might not be optimal for the purpose of configuration parsing.

Another approach that could be explored is to use a hybrid approach, by first performing an initial structuring step to standardise the document into its most thorough form, such as turning the "name" field from a string to a dictionary. Then secondly using JSON schemas and existing libraries like `jsonschema` to validate types within that structure, such as validating that the "name" field is a dictionary with values of type string. Additional checks could be performed for more advanced checks such as ensuring that no rights holder exists if the problem is in the public domain. This might allow the code to be more concise, as a significant number of checks can be offloaded to external libraries. The fundamental issue of increasing modularity would likely still be an issue with this approach however, since this does not imply any systemic changes.

One possibility is that it might not be possible to produce code that is modular for this use case without it turning out to be complicated. The general approach would save lines of code compared to the simple approach given enough problem configuration files, but this raises the question if this will ever be required. If Kattis does not make multiple new updates to the format in the future, the downside of added size and complexity might be larger than the benefit that comes from generality, which might be the reason why the general approach was denied. This links to another principle YAGNI, which is short for "You Aren't Gonna Need It", which advocates for not creating complicated systems in order to satisfy future needs [45]. Thus, in contexts where requirements are stable, such as Kattis' problem formats, pragmatic simplicity might outweigh strict adherence to DRY and SOLID. This might however need to be reconsidered if the scope of the system expands.

5.4 Static Validators

The implementation of the static validator aimed to reuse code, exhibit modularity and clarity. The results indicate that this was only partially achieved.

The main findings relating to the act of code reuse demonstrate both positive and negative aspects. Reducing the amount of development needed to perform similar processes is generally advantageous. However, there is an inherent risk of the resurgence of the same issues as in the original code. To prevent this, developers are advised to examine the code to be reused thoroughly and evaluate which parts are appropriate to reuse, and which need to rewritten. Given these findings, it may have been relevant to consider creating a common superclass for output validators and static validators, given their high degree of similarity.

The results show that the Single responsibility principle is violated in the static validator. To improve the design, all the different behaviours in the validate method need to be extracted into smaller method. Doing this would improve the readability and maintainability of the static validator class. Additionally, adherence to the Open close principle would increased, which in turn facilitates future extensions.

Despite no advanced static validators being produced during the project, the format is flexible enough to support more complex behaviour. A promising approach not fully explored is to implement the

static validator by parsing the source code into an abstract syntax tree using libraries. Using the abstract syntax tree, conclusions can be made about the source code in a manner that is robust to small changes, such as whitespace usage. For example, using the abstract syntax tree, the keywords used by the program can be reliably obtained. At the same time, it is deemed likely that some conclusions cannot possibly be drawn about the source code, as they might imply a solution to the halting problem. Finally, it is believed that a promising middle ground between the complexity of abstract syntax trees and the simplicity of naive string searches is to use carefully-crafted regular expressions with limited scope.

5.5 Multi-pass problems

The class hierarchy in multi-pass presents both advantages and disadvantages. If a new type of multi-pass is to be added, it is going to be easier to implement, as the new implementation would hopefully only need to create a subclass that implements all the differences. However, it is deemed unlikely that this will work. Despite multiple attempts, no reasonable problem that would be used in a contest could be invented that cannot be implemented using an interactive multi-pass problem. Therefore, it was difficult to write an implementation anticipating change, as the changes will likely be very large. Despite this, it was argued that the usage of the template method pattern was beneficial, as it allowed a clearer distinction of responsibilities and not duplicating code between interactive and non-interactive multi-pass.

5.6 Markdown

Using Pandoc to render Markdown had both advantages and disadvantages. It was highly suitable in generating HTML, as it supports a Markdown flavour similar to CommonMark — requiring minimal effort to get good results for footnotes, tables, and mathematical formatting. However, it was less suitable for PDF conversion due to rendering tables in an unconventional way by default. This lead to a manual patch being implemented. This solution lacked robustness, highlighting the importance of thorough testing, especially given the fact that many issues with tables were discovered late in the project. Even though Pandoc was not perfect, paired with the library nh3 which was very well suited for the task of sanitisation, being strict, while also allowing customisability to allow specific elements, or to write arbitrary logic to allow exactly what is needed for the statement.

When it comes to security, most XSS and Javascript vulnerabilities could be prevented by using nh3 together with careful escaping of any inserted content. However, it turned out that preventing URLs to external websites was more difficult. Security was achieved through a strict RegEx on the image source, and by ensuring that the image source points to a file that exists. This highlights the general difficulty caused by the same data being interpreted in different ways. Pandoc's eagerness to perform web requests also highlights the importance of thoroughly testing all libraries used, as otherwise such behaviour could be missed.

5.7 Impact

This section discusses the impacts of the project in both academia and industry. First, the most direct impact of the project is the changes that were merged into Problemtools. Specifically, Markdown rendering and various code quality improvements were merged. Many online judges, including Kattis, use Problemtools during problem installation, which means that Markdown statements are supported on the Problemtools side. Additionally, the code quality improvements facilitate the development of new features from the 2023-07 format in Problemtools in the future. Second, other online judges will be able to use the findings of the report when developing their own backend. In particular, they can use both the analysis of requirements, the development choices and the results to make more informed decisions on how to implement their backend. If they deem the Markdown or config parsing implementations to be appropriate, it is easy for them to reuse the code. However, for features such as static validators and multi-pass, it is not recommended to use them for the backend, as they make no effort to sandbox any executed code. This is intentionally done, as it is outside the scope of ProblemTools. Finally, for wider academia, the project serves as a case study of the trade-offs that occur when developing new features for an old system, in particular that of an online code judge.

5.8 Ethical Considerations

The project's two main goals will be analysed separately. First, the act of investigating strategies to refactor Kattis' codebase to allow both formats is considered mostly good. It provided value to both Kattis and its users. At the same time, there is a risk of introducing security vulnerabilities, which may cause economic losses to Kattis or lead to the integrity of its users being compromised. This risk was mitigated by thoroughly testing code and having it be reviewed by Kattis employees before being added to their codebase. Second, investigating security risks and then publishing the findings is fundamentally risky, as malicious individuals may exploit any vulnerabilities found. To minimise this risk, only findings that had been patched in Kattis were disclosed. In addition, had any major vulnerabilities been discovered that are likely to affect other online judges, the authors would have contacted them well in advance of publishing. In conclusion, the project inherently carries risks, but by mitigating them via thorough testing and using safe publishing practices, the gains in knowledge and production of value far outweigh the risks.

5.9 Future Work

The project has implemented a subset of the features of the 2023-07 format. One way to further the goals of the project would be to implement the remaining features that the project did not manage to accomplish, such as the new submission handling. This would give further insights into implementation strategies for the format.

This project has only considered the technical aspects of the 2023-07 format, but another aspect that could be explored is the usefulness for companies, problem-solvers, problem-authors, and teachers. This could be done by surveying the desired groups utilising the platform, about the usefulness of different features. This would highlight the potential benefits of the new features.

6 Conclusion

This project was undertaken to design new features in an existing code judge, Kattis, and to evaluate design considerations such as maintainability and security. This was done by designing and implementing systems to handle selected features from the new 2023-07 problem format, namely static validators, Markdown statements, and multi-pass problems. In addition to this, systems for handling the general problem structure and to parse configuration files in the old Legacy format were redesigned to be generalised to both formats.

The results highlight the trade-off between abstraction and complexity when designing general systems, especially apparent when examining the configuration parsing system that was made. The project also showed that certain security measures must be taken when implementing some of the features, such as Markdown statements in the 2023-07 format. These findings illustrate how developers must keep in mind complexity, necessity, and practicality when building complex systems.

The analysis in this thesis focusses on system design, flexibility, and future extensions. It investigates how decisions based on programming principles can improve the implementation of features with regard to the aforementioned aspects.

Ultimately, the project gives insight into some of the aspects that need to be considered when implementing similar features. Because the Kattis problem format is open source, the findings can contribute to making the 2023-07 format more widely adopted by code judges using it.

Bibliography

[1] Kattis Inc., *Kattis: Online code-judge*, Accessed: 2025-03-01, 2025. [Online]. Available: https://www.kattis.com/.

[2] DOMjudge Team, *DOMjudge: A programming contest control system*, Accessed: 2025-05-19, 2025. [Online]. Available: https://www.domjudge.org/.

[3] PC^2 Team, *PC^2: Programming contest control system*, Accessed: 2025-05-19, 2025. [Online]. Available: https://pc2ccs.github.io/.

[4] Johan Sannemo, *Omogenjudge: An open-source online judge*, Accessed: 2025-05-19, 2023. [Online]. Available: https://github.com/jsannemo/omogenjudge.

[5] Y. Xie, *Bookdown: Authoring books and technical documents with R Markdown*. CRC Press, 2016, pp. 1–113, Cited by: 65. DOI: 10.1201/9781315204963. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85051606668&doi=10.1201%2f9781315204963&partnerID=40&md5=b550df3ede1c21fa8cee6de9f19065f8.

[6] Kattis, Accessed: 2025-03-16. [Online]. Available: https://www.kattis.com/problem-package-format/spec/legacy.html.

[7] Kattis. "2023-07 github commits." Accessed: 2025-05-06. (2025), [Online]. Available: https://github.com/Kattis/problem-package-format/commits/master/spec/2023-07-draft.md?after=3630378c732aedf5c04f3dc89a0929fd85a025f9+384.

[8] Kattis, *Problemtools: An open-source project*, Accessed: 2025-03-01, 2025. [Online]. Available: https://github.com/Kattis/problemtools/.

[9] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal, "A survey on online judge systems and their applications," *ACM*, p. 5, 2018. DOI: https://doi.org/10.1145/3143560.

[10] Codeforces Community, *Codeforces: Online code-judge*, Accessed: 2025-03-01, 2025. [Online]. Available: https://codeforces.com/.

[11] LeetCode, Inc., *Leetcode: Online code-judge*, Accessed: 2025-03-01, 2025. [Online]. Available: https://leetcode.com/.

[12] Kattis Inc. "Kattis hello world problem." Accessed: 2025-03-17. (2025), [Online]. Available: https://open.kattis.com/problems/hello.

[13] Kattis, *The kattis problem package format specficiation*. Accessed: 2025-05-19. [Online]. Available: https://www.kattis.com/problem-package-format/.

[14] Kattis. "Kattis: About us." Accessed: 2025-05-06. (2025), [Online]. Available: https://se.linkedin.com/company/kattis?trk=public_profile_experience-item_profile-section-card_subtitle-click.

[15] Kattis, Accessed: 2025-03-18. [Online]. Available: https://www.kattis.com/problem-package-format/spec/2023-07-draft.html.

[16] Kattis and collaborators. "Verifyproblem.py." Accessed: 2025-05-06. (2025), [Online]. Available: https://github.com/Kattis/problemtools/blob/develop/problemtools/verifyproblem.py.

[17] Kattis, *Problem2html.py: Latex-to-html problem statement generator*, GitHub repository, Accessed 2025-05-19, 2025. [Online]. Available: https://github.com/Kattis/problemtools/blob/develop/problemtools/problem2html.py.

[18] Kattis, *Problem2pdf.py: Latex-to-pdf problem statement generator*, GitHub repository, Accessed 2025-05-19, 2025. [Online]. Available: https://github.com/Kattis/problemtools/blob/develop/problemtools/problem2pdf.py.

[19] R. Plosch, H. Gruber, A. Hentschel, *et al.*, "The emisq method - expert based evaluation of internal software quality," in *31st IEEE Software Engineering Workshop (SEW 2007)*, 2007, pp. 99–108. DOI: 10.1109/SEW.2007.71.

[20] O. Masmali and O. Badreddin, "Towards a model-based fuzzy software quality metrics," in *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - MODELSWARD*, INSTICC, SciTePress, 2020, pp. 139–148, ISBN: 978-989-758-400-8. DOI: 10.5220/0008913701390148.

[21] W. Haoyu and Z. Haili, "Basic design principles in software engineering," in *2012 Fourth International Conference on Computational and Information Sciences*, 2012, pp. 1251–1254. DOI: 10.1109/ICCIS.2012.91.

[22] I. Oktafiani and B. Hendradjaya, "Software metrics proposal for conformity checking of class diagram to solid design principles," in *2018 5th International Conference on Data and Software Engineering (ICoDSE)*, 2018, pp. 1–6. DOI: 10.1109/ICODSE.2018.8705857.

[23] H. Mu and S. Jiang, "Design patterns in software development," in *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, 2011, pp. 322–325. DOI: 10.1109/ICSESS.2011.5982228.

[24] M. Ozkaya and M. A. Kose, "Designing and implementing software systems using user-defined design patterns," in *Proceedings of the 16th International Conference on Software Technologies - ICSOFT*, INSTICC, SciTePress, 2021, pp. 497–504, ISBN: 978-989-758-523-4. DOI: 10.5220/0010571404970504.

[25] E. A. AlOmar, H. AlRubaye, M. W. Mkaouer, A. Ouni, and M. Kessentini, "Refactoring practices in the context of modern code review: An industrial case study at xerox," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 348–357. DOI: 10.1109/ICSE-SEIP52600.2021.00044.

[26] A. Danial, *CLOC: Count Lines of Code*, https://github.com/AlDanial/cloc, Version 1.90, 2006.

[27] F. Niemelä, *Personal interview*, Personal Communication, Unpublished interview conducted by Joshua Andersson, May 9, 2025.

[28] A. John Gruber, *Commonmark*, Accessed: 2025-04-06, 2025. [Online]. Available: https://commonmark.org/.

[29] Kattis. "Problems." Accessed: 2025-03-19. (2025), [Online]. Available: https://open.kattis.com/problems.

[30] Matt Cone, *Extended syntax*, Accessed: 2025-04-06, 2025. [Online]. Available: https://www.markdownguide.org/extended-syntax/.

[31] John MacFarlane, *Pandoc*, Accessed: 2025-05-08, 2025. [Online]. Available: https://pandoc.org/.

[32] O. Foundation, *Threat modeling*, https://owasp.org/www-community/Threat˙Modeling, Accessed: 2025-06-02, 2024.

[33] OWASP, *Cross site scripting (xss)*, Accessed: 2025-05-12, n.d. [Online]. Available: https://owasp.org/www-community/attacks/xss/.

[34] messense. "Nh3." Accessed: 2025-03-19. (2025), [Online]. Available: https://nh3.readthedocs.io/en/latest/.

[35] *Document management — portable document format — part 1: Pdf 1.7*, Standard, International Organization for Standardization, 2008. [Online]. Available: https://www.iso.org/standard/51502.html.

[36] OpenAI, *Chatgpt (gpt-4)*, AI language model, Conversation held on 2025-04-08. Generated XSS examples for Markdown. Version: GPT-4o. Prompt: "i want to test that my sanitization system is working. give me some xss examples in markdown.", 2025.

[37] Kattis. "Reporting a judgement." Accessed: 2025-04-25. (2025), [Online]. Available: https://www.kattis.com/problem-package-format/spec/2023-07-draft.html#reporting-a-judgement.

[38] Kattis. "Problem package fromat 2023-07: Static validator." Accessed: 2025-05-02. (2025), [Online]. Available: https://www.kattis.com/problem-package-format/spec/2023-07-draft.html#static-validator.

[39] Kattis, *An example problem: Hello world!* Accessed: 2025-05-02. [Online]. Available: https://github.com/Kattis/problemtools/tree/develop/examples/hello.

[40] J. Sannemo and S. Lindholm, *Nordic olympiad 2017 tasks*, Github Repository, Accessed: 2025-05-19. [Online]. Available: https://github.com/nordicolympiad/nordic-olympiad-2017.

[41] Kattis, Accessed: 2025-03-30. [Online]. Available: https://www.kattis.com/problem-package-format/appendix/languages.html.

[42] M. Mareš, B. Blackham, and Other Contributors, *Isolate: Sandbox for competitive programming problems*, GitHub repository, Accessed 2025-05-19. [Online]. Available: https://github.com/ioi/isolate.

[43] T. Beuman, *Alternative encryption*, https://2024.nwerc.eu/test-session/problem-set.pdf, Programming problem from NWERC 2024 Practice Session. Problem materials available at https://2024.nwerc.eu/test-session/solutions.zip, 2024.

[44] J. Andersson, *Interactive communication*, https://github.com/Matistjati/kattis-markdown-examples/tree/master/interactivecommunication, Programming problem developed to test interactive multi-pass problems. Accessed 2025-05-19, 2025.

[45] M. Fowler. Accessed: 2025-05-19. (2015), [Online]. Available: https://martinfowler.com/bliki/Yagni.html.

A Appendix

A.1 Configuration Parsing Test Set

Competition	**Number of problems**
Keppnir 2024	34
Keppnir 2023	30
Keppnir 2022	30
Keppnir 2021	24
Keppnir 2020	33
Keppnir 2019	26
Keppnir 2018	20
Keppnir 2017	18
PO final 2025	8
PO katt 2025	3
PO lager 2025	3
PO långtävling 2025	4
PO onlinekval 2025	9
PO skolkval 2025	7
PO final 2024	7
PO katt 2024	3
PO lager 2024	3
PO långtävling 2024	4
PO onlinekval 2024	8
PO skolkval 2024	6
PO final 2023	7
PO katt 2023	3
PO lager 2023	3
PO onlinekval 2023	7
PO skolkval 2023	5
PO final 2022	6
PO katt 2022	3
PO lager 2022	3
PO onlinekval 2022	7
PO skolkval 2022	6
PO final 2021	6
PO katt 2021	3
PO lager 2021	3
PO onlinekval 2021	7
PO skolkval 2021	6
Total	355

Table A.1: Competitions with legacy problems that were used for testing the config parsing systems.

A.2 Markdown Statement: IOI Hieroglyphs

hieroglyphs
Day 2 Tasks
English (ISC)

Hieroglyphs

A team of researchers is studying the similarities between sequences of hieroglyphs. They represent each hieroglyph with a non-negative integer. To perform their study, they use the following concepts about sequences.

For a fixed sequence A, a sequence S is called a **subsequence** of A if and only if S can be obtained by removing some elements (possibly none) from A.

The table below shows some examples of subsequences of a sequence $A = [3, 2, 1, 2]$.

Subsequence	How it can be obtained from A
[3, 2, 1, 2]	No elements are removed.
[2, 1, 2]	[~~3~~, 2, 1, 2]
[3, 2, 2]	[3, 2, ~~1~~, 2]
[3, 2]	[3, ~~2~~, ~~1~~, 2] or [3, 2, ~~1~~, ~~2~~]
[3]	[3, ~~2~~, ~~1~~, ~~2~~]
[ ]	[~~3~~, ~~2~~, ~~1~~, ~~2~~]

Figure A.1: The beginning of the original statement of the task Hieroglyphs, used in IOI 2025.

Hieroglyphs

Problem ID: hieroglyphs

Hieroglyphs

A team of researchers is studying the similarities between sequences of hieroglyphs. They represent each hieroglyph with a non-negative integer. To perform their study, they use the following concepts about sequences.

For a fixed sequence A, a sequence S is called a **subsequence** of A if and only if S can be obtained by removing some elements (possibly none) from A.

The table below shows some examples of subsequences of a sequence $A = [3, 2, 1, 2]$.

Subsequence	How it can be obtained from A
[3, 2, 1, 2]	No elements are removed.
[2, 1, 2]	[~~3~~, 2, 1, 2]
[3, 2, 2]	[3, 2, ~~1~~, 2]
[3, 2]	[3, ~~2~~, ~~1~~, 2] or [3, 2, ~~1~~, ~~2~~]
[3]	[3, ~~2~~, ~~1~~, ~~2~~]
[ ]	[~~3~~, ~~2~~, ~~1~~, ~~2~~]

Figure A.2: The beginning of the statement of Hieroglyphs, rendered to HTML using Problemtools. Removing the header with the IOI logo, as compared to figure A.1 was a deliberate choice, as Kattis does not use headers.

A.3 Size of Code Changes

Part	**Additions**	**Deletions**
Problem abstraction	346	216
Problem abstraction deprecation removal	3	300
Statement folder name	117	20
Problem configuration general approach	2,628	143
Problem configuration simple approach	766	229
Markdown statements problemtools	1,092	66
Static validator	184	6
Multi-pass problems backend	409	42
Multi-pass problems problemtools	162	43
Other	9	5
Total	5,716	1,070

Table A.2: Summary of the magnitude of changes in the code base.

Part	**Additions**	**Deletions**
Problem abstraction	346	216
Problem abstraction deprecation removal	3	300
Statement folder name	117	20
Markdown statements problemtools	1,092	66
Multi-pass problems backend	409	42
Total	1967	644

Table A.3: Summary of the magnitude of changes in the code base that were merged.