



## ASSIGNMENT OF MASTER'S THESIS

**Title:** PCSS – System for Automatic Evaluation of Submitted Computer Programs  
**Student:** Bc. Tomáš Hauk  
**Supervisor:** Ing. Martin Kačer, Ph.D.  
**Study Programme:** Informatics  
**Study Branch:** Web and Software Engineering  
**Department:** Department of Software Engineering  
**Validity:** Until the end of winter semester 2016/17

### Instructions

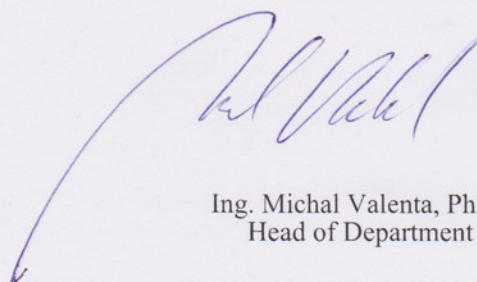
PCSS is a system for automatic evaluation of computer programs submitted by students during programming competitions. The system consists of components communicating with a remote API. The goal of this work is to finish the pilot implementation of PCSS.

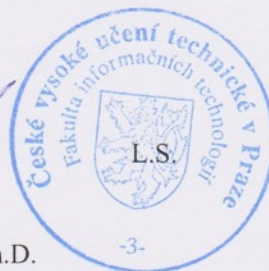
- Study the current state of existing components of PCSS and their API.
- Fix the functionality of components that are currently not working as expected, namely the evaluation component.
- Design the missing components (and their APIs) necessary to completely evaluate a program.
- Implement all such components to the point allowing to evaluate submitted programs in C and Java.
- Document and test your design and implementation.

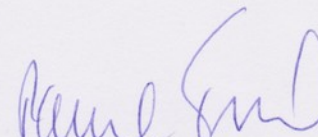
The result should be a functional system that allows to submit programs with a web user interface, compile them, run against test data, evaluate the correctness, and provide a scoreboard based on given criteria.

### References

Will be provided by the supervisor.

  
Ing. Michal Valenta, Ph.D.  
Head of Department



  
prof. Ing. Pavel Tvrđík, CSc.  
Dean

Prague September 10, 2015



CZECH TECHNICAL UNIVERSITY IN PRAGUE  
FACULTY OF INFORMATION TECHNOLOGY  
DEPARTMENT OF SOFTWARE ENGINEERING



Master's thesis

## **PCSS – System for Automatic Evaluation of Submitted Computer Programs**

*Bc. Tomáš Hauk*

Supervisor: Ing. Martin Kačer PhD.

11th January 2016



---

## **Acknowledgements**

Foremost, I would like to express my sincere gratitude to my advisor Ing. Martin Kačer PhD., for his support, guidance and helpful advices. I would also like to thank my family and close friends for their patience and support during the course of writing this thesis.



---

# Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 11th January 2016

.....

Czech Technical University in Prague  
Faculty of Information Technology

© 2016 Tomáš Hauk. All rights reserved.

*This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).*

### **Citation of this thesis**

Hauk, Tomáš. *PCSS – System for Automatic Evaluation of Submitted Computer Programs*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.



---

# Abstrakt

Náplní této práce je pilotní implementace systému pro automatické vyhodnocování úloh, zejména v programování. Hlavními požadavky na systém jsou modularita a rozšiřitelnost, jež zajistí možnost použití systému i pro méně typické případy. Výsledkem je funkční systém, který nabízí možnost zabezpečeného vyhodnocení zdrojových kódů v programovacích jazycích C, C++ a Java a dále nabízí webové rozhraní pro běh obecné soutěže nebo školního kurzu v programování.

**Klíčová slova** vyhodnocovací systém, soutěže v programování, ICPC

---

# Abstract

The aim of this work is the pilot implementation of a system for automatic evaluation of programming assignments. The main system requirements are modularity and extensibility that will allow to use this system in less typical situations. The result of this thesis is a functional system that is able to automatically evaluate source codes in the C, C++ and Java programming languages and also offers a web user interface for operating general contest or school course in programming.

**Keywords** evaluation system, programming contests, ICPC



---

# Contents

<b>Introduction</b>	<b>1</b>
Theses related to PCSS . . . . .	2
<b>1 Review of the original system architecture</b>	<b>3</b>
1.1 Domain terminology . . . . .	3
1.2 Initial state of the system . . . . .	5
<b>2 Analysis and design</b>	<b>9</b>
2.1 Collection and analysis of use cases and system features . . . . .	9
2.2 System architecture . . . . .	11
2.3 CML component . . . . .	12
2.4 Judge component . . . . .	15
2.5 Web interface . . . . .	22
2.6 Scoreboard component . . . . .	24
2.7 Proxy component . . . . .	25
2.8 Other system modules . . . . .	25
<b>3 Implementation</b>	<b>31</b>
3.1 Judge component . . . . .	34
3.2 WebUI component . . . . .	40
3.3 Scoreboard component . . . . .	41
<b>4 Testing</b>	<b>43</b>
4.1 Automated testing . . . . .	43
4.2 Manual testing . . . . .	45
4.3 CML testing . . . . .	47
<b>Conclusion</b>	<b>49</b>
Future work . . . . .	50

<b>Bibliography</b>	<b>51</b>
<b>A Acronyms</b>	<b>55</b>
<b>B Contents of enclosed CD</b>	<b>57</b>
<b>C Deployment instructions</b>	<b>59</b>
C.1 CML deployment . . . . .	59
C.2 Judge deployment . . . . .	60
C.3 Scoreboard deployment . . . . .	60
C.4 Web Interface deployment . . . . .	61
<b>D Documentation</b>	<b>63</b>
D.1 Component configuration . . . . .	63
D.2 System configuration . . . . .	67
D.3 Extending system functionality . . . . .	73
<b>E Web interface screenshots</b>	<b>77</b>

---

## List of Figures

1.1	Raw simplified domain diagram of PCSS system. . . . .	4
1.2	Data processing in PCSS. From [1]. . . . .	5
1.3	Illustration of basic system components. . . . .	6
2.1	Illustration of common system process. Taken from [1]. . . . .	10
2.2	Illustration of the Judge sub components tree and the orchestrator/unit design. . . . .	18
2.3	Illustration of the interface between the orchestrated unit and the actual implementation of the algorithm. . . . .	18
2.4	Domain model of the contest data. . . . .	26
2.5	Illustration of the internal representation of the contest directory in the CML component. . . . .	29
3.1	Illustration of modules communication. . . . .	32
3.2	System modules and their dependencies. . . . .	33
3.3	Top level Judge process. . . . .	34
3.4	Illustration of the validation process. . . . .	35
3.5	Illustration of the structure defining the submission processing algorithm with two-step validation. . . . .	37
3.6	Illustration of the scoring algorithm. . . . .	39
C.1	Illustration of deployed system components. Taken from [1] . . . . .	62
E.1	Contest detail with list of problems. . . . .	77
E.2	Web interface main page with contest list. . . . .	78
E.3	Problem submissions and problem details and stats. . . . .	78
E.4	Logs of the compilation process. . . . .	79
E.5	Contest scoreboard. . . . .	79



---

# Introduction

PCSS (stands for „Programming contest scoring system”) is a system for automated evaluation of submissions of either programming contest or a school course. Its final version however should be able to operate other types of contests such as best design contest or any other of non-programming oriented contests.

The motivation for creating a new evaluation system is based on the fact that the currently available ones (known to the author and researched in the theses mentioned below) have either some functional issues or lack certain desired features.

The most common major issue is the very specific orientation of the system on some specific purpose, e.g. particular programming contest like mentioned ACM ICPC. The rules of the contest are reflected in the system implementation in such a way that e.g. implementation of some other scoring approach is not possible. Some systems for contest management are also not designed to manage multiple contests at one time and usually aren't suitable for running in a different „domain context” like school course (and vice versa).

Other practical issue is the system availability and this is related to system extensibility. Although most of the systems are available for free, not all of them have also available their source codes. Lack of certain features (such as support of e.g. Java language) might make them unusable for some purposes and usage scenarios.

The system implemented in this thesis aims be an attractive alternative to the currently existing evaluation systems by offering features and solving problems described above. The system is designed with modularity in mind and none of its components is presuming anything about the actual „domain context” in which the systems runs, yet the system seems to be well suited for managing contest, school course or run as an publicly available archive of programming assignments. The system will be released as an open source.

This thesis builds on the work done in the past (complete list below), mainly on the thesis [1] that aimed to design the top level system architecture.

The core result of the work (except the architecture) was the implementation of the unique and system specific data repository, so called CML. Since the results of the mentioned thesis seemed plausible, the CML component was used in this implementation without major changes and the main focus in this thesis was on the implementation of components responsible for actual program evaluation and web user interface.

## Theses related to PCSS

This is the fifth thesis related to the PCSS system, in version designed in [1] (there exists an old system of the same name that is predecessor of this system). Following list is a brief summary of the work, that has been done in the past.

**Secure program execution** Is bachelor thesis [2] that has been successfully defended in 2012, implementing framework for secure program execution.

**PCSS** My bachelors thesis [1], successfully defended in 2013. The aim of the thesis was to design and implement the initial version of the PCSS system but the core of the thesis was unique data repository of the system (so called CML component) — other implemented components were in state of functional prototypes.

**PCSS web interface** Thesis [3] successfully defended in 2013 that aimed to implement the web interface for the PCSS system for the purposes of ACM ICPC programming contest [4].

**Testing of PCSS** Bachelors thesis [5] that was testing the initial implementation of the PCSS CML component. Successfully defended in 2015.



---

# Review of the original system architecture

Since the research has been already done, as is shown in relevant works listed above, and the work assignment states that the system should be built on author's previous thesis [1], author undertook research only when solving partial problems of the system design or implementation (can be found in appropriate sections of the thesis). This chapter will make just a brief overview of the initial state of the system as implemented in [1].

## 1.1 Domain terminology

Since the final version of the system is planned to be used in different contexts, this section will briefly address the terminology problems of this feature for purposes of the thesis.

In the context of the ACM ICPC contest, the raw domain model of the system will look as follows:

- There is one *contest* in the system.
- The main users of the system are *teams* that are solving *contest problems* that are given.
- Each contest has a set of predefined *problems* to be solved.
- The output of the team (source codes) which solves a problem is called a *submission*.
- The evaluated submission of the team is called a *judgment*.

Another context example in which the system can be used is automatic evaluation of school assignments. In such case, the raw domain model will look as follows:

## 1. REVIEW OF THE ORIGINAL SYSTEM ARCHITECTURE

---

- There is a set of courses in the system.
- The main user of the system is student who is signed up for particular course
- Each course has set of problems or homeworks which the student should solve
- The solution of the problem (created by student) is called submission
- The evaluated submission can be called judgment, grade or some point gain etc.

Based on the examples above it is clear that the actual model remains the same, but the terminology changes slightly for the purposes of a given context. For purposes of better clarity of this thesis, the author uses exclusively the terminology that has been mentioned first (ACM ICPC context). The raw domain model is illustrated on the diagram 1.1.

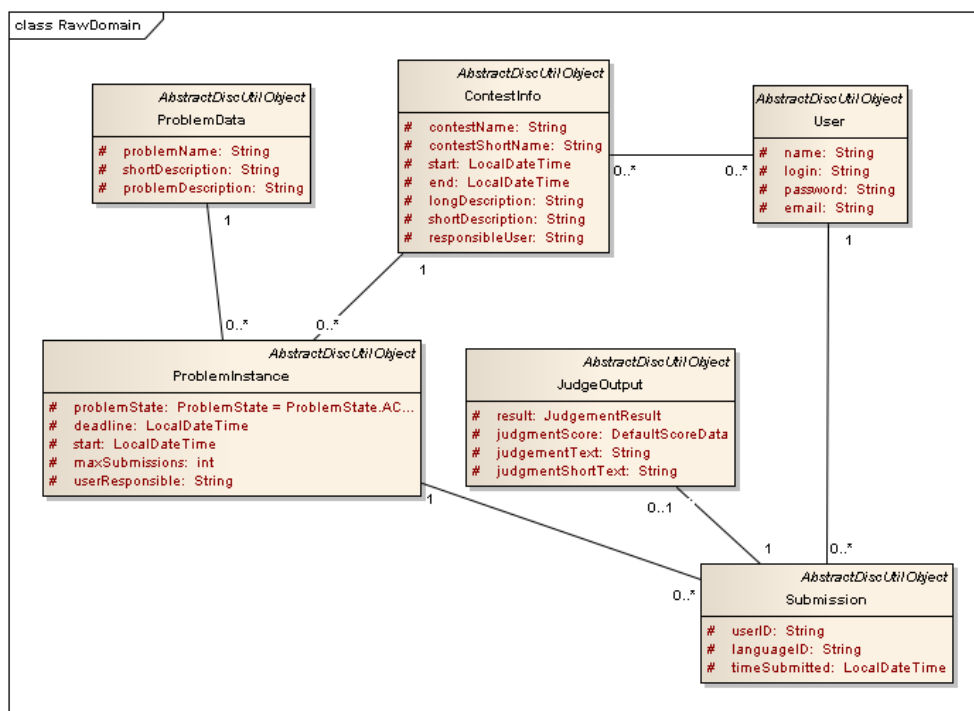


Figure 1.1: Raw simplified domain diagram of PCSS system.

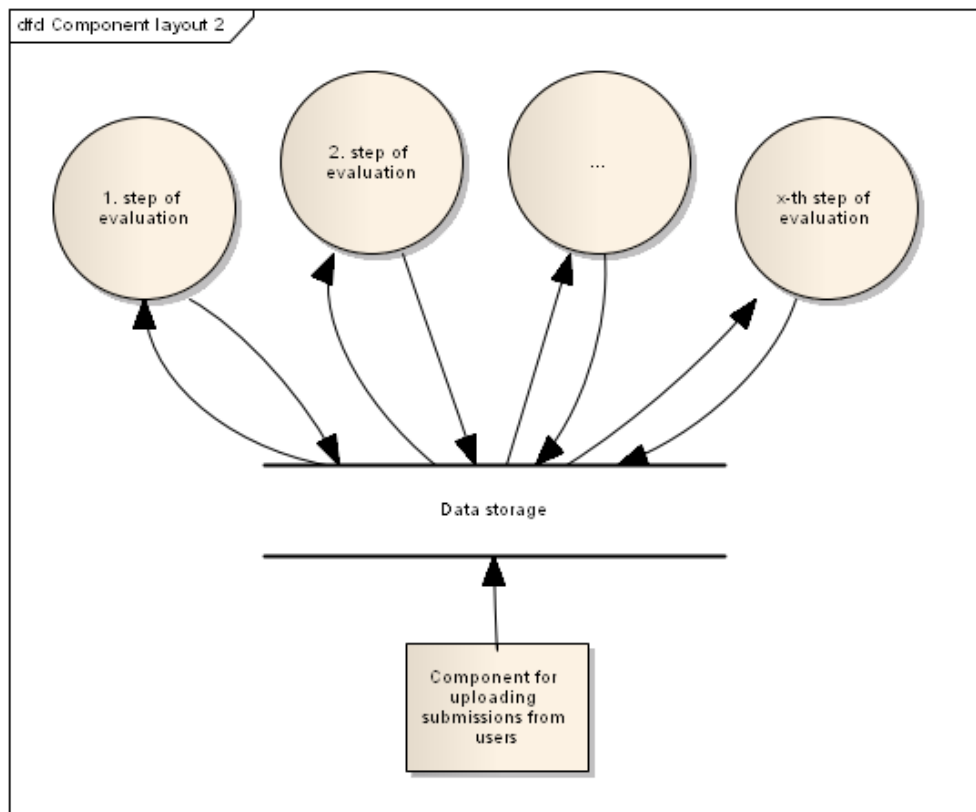


Figure 1.2: Data processing in PCSS. From [1].

## 1.2 Initial state of the system

The PCSS system is component oriented. Communication between components is based on the REST principle [6] with the use of HTTP protocol [7]. Core component of the system is called CML which serves as a unique data repository and as a mediator between most of the other components of the system. Other components are responsible for specific tasks — in most cases, they download unprocessed data from the CML and upload the processed data back. The basic structure of the system and its components can be found on diagram 1.3. Short description of components implemented in the initial version of the system follows.

### 1.2.1 CML

This is the core component of the PCSS system responsible for data management (retrieve and store) and synchronization of the other components work. In typical case, each system component responsible for some particular task

## 1. REVIEW OF THE ORIGINAL SYSTEM ARCHITECTURE

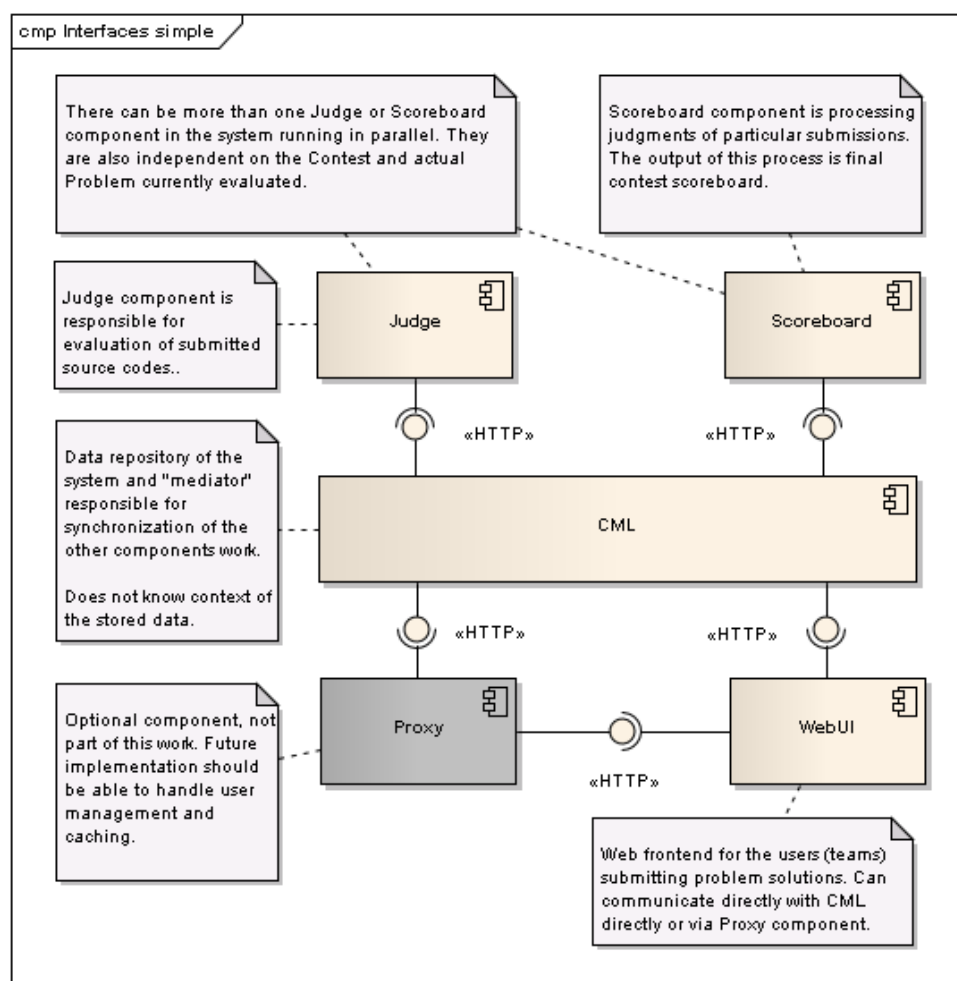


Figure 1.3: Illustration of basic system components.

will take its input data from the CML and the processed result will upload back to CML, where it can be used as an input for another processing (this approach is illustrated on diagram 1.2). The data in the CML should be accessed exclusively by the other dedicated components (i.e. access from web browser should be denied).

The CML component was designed to be independent on the context of the data being stored which means that it can be used for other systems than PCSS. The current implementation does not support management of data relations and effective data searching.

During the process of the CML design, author made an observation that data in this type of system are (or should be) typically only created and read, but not updated (only „interpreted” differently). Even if some submission will be reevaluated multiple times (e.g. because of a technical issue), every

other previous judgment will stay in the system in the original form and the system will just correctly decide which one is the actual one. In case of contest scoreboard for example, the system creates new data entity representing the scoreboard each time that the scoreboard needs an update (new submission judgments were made) and the actual one is the newest one. In a certain sense, the data entities stored in CML component are forming a „history” of the contest.

As the consequence of the observation above, the data entities stored in the CML component are **immutable** by default. Thus, the effective data updates are not supported. In case that the intervention in the stored data is really necessary (e.g. in case of some technical issues), the changes can be done via appropriate HTTP methods (see below), but updating the data on routine basis should be viewed as a certain violation of the CML philosophy.

Internally, the data are stored on the local file system in the multipart format [8] which is also used in the body of the HTTP requests. The current implementation is not efficient (using global pessimistic data locking) nor secure (can be fixed via application server SSL settings).

Data entities are accessed via HTTP protocol as „resources” (REST principle). Standard implemented methods are GET (access data entity or directory), PUT (create new entity) and POST (create new versioned entity). For synchronization purposes there were implemented LOCK and UNLOCK methods (these methods can operate on the whole data tree). For data updates or deletion purposes there were implemented FREEZE, CLONE, RESTORE and DISCARD methods.

The implementation details and discussion of design decisions can be found in [1].

### 1.2.2 Judge

Judge component is responsible for evaluation of the user submissions. Input data are submission and the problem definition, output of the process is called judgment result. This component communicates directly with CML.

The initial version of the component was able to evaluate source codes of the C/C++ programming languages in a non-secure manner. Each contest instance needed at least one own Judge component but could have more Judge components running in parallel.

### 1.2.3 Scoreboard

This component is responsible for generating the contest scoreboard. Input data are the best submissions of the problem for each user (team) in the contest, the output is processed scoreboard of the contest. Main idea behind this component is ability to post-process the results in specific way and to

provide optimization due the CML. This component communicates directly with CML.

The initial version of the Scoreboard component was able to evaluate the submissions only for the purposes of the ACM ICPC contest and was tightly bind to the contest (i.e. each contest instance required its own scoreboard component).

### 1.2.4 Proxy

The idea of the Proxy component was taken from the previous version of the PCSS system. The purpose of this component is to provide user management functionality, submission timestamping and caching (e.g. ability to accept the submission even if the CML is unreachable for some reason). This component serves as a mediator between the user interface and the CML and also provides same API as the CML itself.

The component in the previous version of the system was only timestamping the submissions, no user management or caching were implemented.

### 1.2.5 Web interface

This component provides user interface for showing contest data (problem definitions, submission results, scoreboard) and uploading submissions. This component can communicate either with CML or with the Proxy component.

The previous version of this component was implemented in the [3], is strictly tied to the ACM ICPC contest style and the interface to the PCSS system was not implemented (i.e. the project was in the state of prototype working only with mock data and mock CML client).

---

# Analysis and design

This chapter describes the process and results of the system analysis and design along with appropriate discussion.

Collection of the use cases can be found in section 2.1. Section 2.2 contains quick discussion of the system architecture. Analysis and design of particular components are described in the following sections: CML 2.3, Judge 2.4, web interface 2.5, Scoreboard 2.6 and Proxy 2.7. Section 2.8 contains discussion of the supporting system modules which are providing API to the CML (HTTP clients) and other system features shared by other components.

## 2.1 Collection and analysis of use cases and system features

Since the collection and analysis of use cases and required system features was performed in every work related to PCSS system (see introduction chapter for complete list), this section will provide just a quick overview and basic discussion of the main use cases which were gathered from the mentioned works, work assignment and from consultations between the author and the supervisor of the thesis. Detailed analysis can be found in the thesis that were referred to. The illustration of basic system evaluation process is illustrated on diagram 2.1.

### 2.1.1 Minimum set of use cases from the main user perspective

The main use cases of the system were gathered from the work assignment where they are described from the perspective of the user which is in role of the student or team who submits source code which will be automatically evaluated. To this user, the system must provide following minimal functionalities the web interface:

## 2. ANALYSIS AND DESIGN

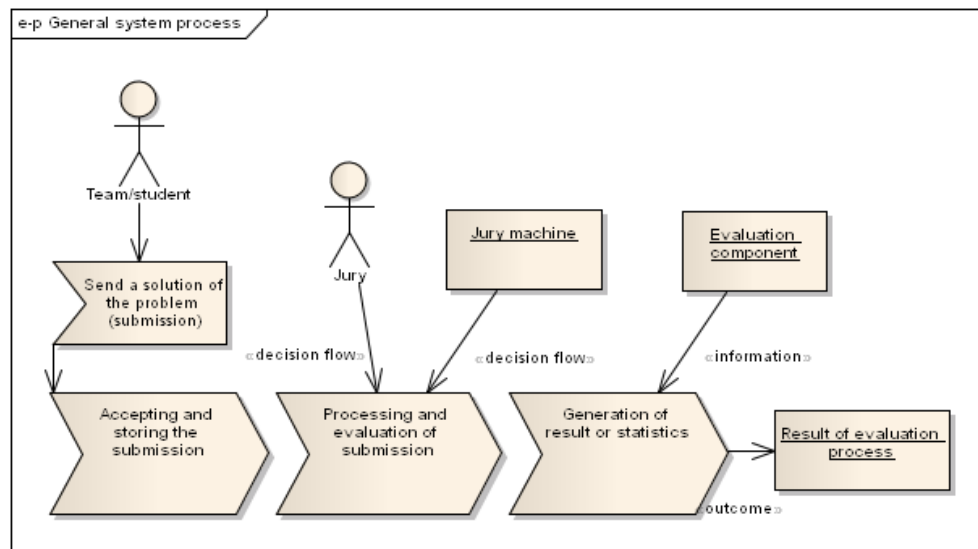


Figure 2.1: Illustration of common system process. Taken from [1].

- Ability to log in to user account
- Show contests, to which the user has access to
- Show contest description
- Show list of contest problems
- Show contest scoreboard
- Show problem description and assignment details
- Show all submissions of user of particular problem
- Show all submission of user of particular contest
- Show scoreboard of particular problem
- Submit new submission in C/C++ or Java programming languages

The list above is the minimum set of use cases, which are necessary to implement, so that system can be used at least for the purposes of the school course. For the purposes of the ACM ICPC contest for example, there would be needed at least one more user role of the Jury, who will be able to e.g. see the submissions of all users, send contest announcements or respond to the clarification requests of the teams. Detailed analysis of such context can be found in [3], where the ACM ICPC web interface prototype was implemented.



### 2.1.2 Required system features

The minimal list of system features looks as follows:

- ability to accept submission of a user
- compile (process) the submission source codes in C/C++ and Java programming languages and create executable (interpretable) file
- perform validation of executable (interpretable) file in a secure way (the execution must prevent corruption or compromising of system data and prevent system intentional and unintentional crash)
- Evaluate the result of the validation process (create judgment)
- Create contest scoreboard from available judgments
- Provide way to create contest data (contest, users, problems etc. . .)
- Provide web interface with features described in 2.1.1.

The list above seems pretty straight forward and does not need more discussion.

## 2.2 System architecture

The original system design is based around a data repository with specific REST API (CML) and components, whose communication is done through the CML (further review of the original architecture and its implications in 2.3). Even though this approach, communication API and mentioned system implications (like object immutability) can be viewed a little bit unorthodox, author is of opinion, that this approach will work for given purpose of the evaluation system, considering both drawbacks and benefits of the approach. Another aspect worth mentioning is the positive feedback this architecture received in the process of defense of the work [1]. Therefore, CML and its API remains the „heart” of the system.

Alternative solution of the one described above would be to discard the CML component from the system and utilize some more „mature” existing solutions, such as some kind of XML database. This approach would probably yield better results in the short-term, but for the current purposes of the systems looks the CML approach sufficient and more interesting in perspective of the future development of both PCSS system and CML itself.

Consequently, the system architecture remains the same and the core of this work will be design and implementation of the specific components, namely Judge (see 2.4) and user web interface (see 2.5) components, which are responsible for submission evaluation and user interaction.

### 2.3 CML component

The main data repository of the system, CML, represented core of the authors previous work. Review of this component showed, that except for some efficiency and security issues provides this component sufficient functionality for the system and therefore no major modification in the implementation or the CML API will be needed. Throughout the entire course of work, the component was stable and behaved as expected, with exception of few bugs, which author fixed immediately. Thus, the conclusion from the architecture section 2.2 holds.

Nevertheless, the CML is still at the state of the prototype and its current implementation can't match other more mature database systems. In the sections below the author discusses current CML drawbacks and their possible solutions, which he observed and gathered during the work with this component in the system. Found issues are concerning the CML security (2.3.1), efficiency (2.3.2) and effective ad-hoc data changes (2.3.3). Some other minor issues are mentioned in 2.3.4.

#### 2.3.1 CML security

Current implementation of the CML does not provide any security mechanisms. Because the system architecture is designed in a way, that user never communicates with CML directly, but only through specific components (even administrators), author sees three possible ways how to solve this issue:

- of the component by the valid SSL certificate, which practically means deploying CML on server which handles the request authorization and implementing the authentication mechanisms to the CML API module (described in 2.8)
- deploying the CML to the server, which is not accessible to public network (security based by network setup and by the fact, that the potential attacker does not know the internal data structure and exact server location).
- combination of the previous two solutions.

Since the pilot deployment won't be in a security critical context (either small school course with approximately 50 students or leisure time programming problems solving), first option should be sufficient. As the consequence, the development was focused on other parts of the system.

### 2.3.2 CML efficiency

The efficiency and optimization of the CML implementation were not addressed yet. The analysis and discussion of the possible solutions follows.

First minor problem is component parallel access. Current implementation of the CML is locking whole data tree by the global lock, which means, that the CML is currently unable to utilize the server parallel processing capabilities. The solution of this problem is either implementing a nontrivial algorithm for locking specific subtrees or transition from the raw file-storage approach to some underlying NoSql/Xml database (see [9] for possible solutions).

Second efficiency issue is related to the web interface component (2.5) and fragmented data model, which is mainly based on convention and direct links uses only when necessary (which is in detail discussed in 2.8.3). This issue lies in the need to display large amount of related data on one page, which can practically mean many HTTP requests (request for each needed data entity). Even redesign of the data model (e.g. not having all problem submission in one directory but having directory for each user for each contest problem) might not solve the issue entirely.

Quite a quick and simple solution to this problem would be to implement some kind of caching mechanism, which can rely on already implemented CML API header „if-modified-since”. In this scenario, the component would download the object in the first request and then it would just check, if the object haven't been modified since the last download, which should be a rare case, since the CML is designed to store immutable objects (changes are happening rarely in exceptional situations, such as fixing technical issues). This approach however won't reduce the number of the HTTP requests, only the amount of transferred bytes.

The only solution of this efficiency problem (provided, that the API base will remain the same) seems to be implementation of mechanism for obtaining more entities in a single HTTP request at once. Best yet, design and implement some mechanism, how to request some entities in specific subtrees (e.g. submission data object relative path will contain user ID somewhere and CML might return only those objects, which satisfies this condition). The actual implementation might be based on some existing solution (e.g. XPath language [10]) or some simple CML specific solution. Even more, this approach could further combined with the solution in previous paragraph, so only changed entities would be actually returned. On the other side, while this approach will make the system quite efficient, it represents certain kind of deflection from the pure and clean REST principle and introduces additional complexity on the CML API level as on the implementation level as well. For this purpose, the reintroduction of the multipart format (which was replaced by pure XML in this version) might be considered (the response to the request for multiple data entities would contain multiple XML data structures, composed in multipart).

As was mentioned in the CML security section 2.3.1, the system pilot implementation would probable serve only to few tens of users and for this purpose should be the implementation sufficient enough. Moreover, both mentioned efficiency problems are related mainly to the UI component. Both other implemented components in this version of the system (Scoreboard 2.6 and Judge 2.4) are for each evaluation cycle sending about 10 HTTP requests to the CML, which is comparatively less to the mentioned web interface component.

### 2.3.3 Data changes

Current version of the CML component still does not support ad-hoc data changes (changes in data for purposes of elimination some kind of a contest error or bug).

For the purpose of fixing some „expected” or „common” problems (such as reevaluating some set of submissions, which were affected by some technical issue), the implementation of specific component would seem as the best solution, since it would allow the contest judges or administrators to perform these tasks comfortably and quickly through some UI.

For the purposes of performing some more complex data interventions, author proposes the following approach. First, the method FREEZE would be performed on the affected data in the CML component. These data would be cloned (CLONE method) to the new location and downloaded to the local machine of administrator. Downloaded data (which will be initially in the form of set of XML files structured in the local file system) could be transformed to one XML structure, on which could be performed required data modifications (e.g. with help of XPath). The modified XML would be than converted back to the set of XML files, those could be uploaded back to the CML and then methods RESTORE and UNFREEZE would be performed.

Approaches described above seems much more plausible thanks to the utilizing of the XML as opposed to the previous multipart format (which is certainly less flexible). Reliable implementation is unfortunately beyond scope of this work.

### 2.3.4 Other miscellaneous CML issues

In the work [5], was found, that the current CML component does not work on the Windows platform. The mentioned work did not discovered why exactly, but since the Unix based operating systems seems better suited for deployment of PCSS, it was decided to solve this issue in the later version of the system.

Another minor issue is semantics of the versioning API of CML component. The original plan was to use the versioning API only for purposes of creating new version of the same object, e.g. scoreboard (there is one data entity representing scoreboard, but in different versions and the currently valid one

is the one with the highest version). But at the same time, there is need to differentiate the submissions of the one particular user to the same problem in contest (i.e. to have a directory, where there are all submissions of user for given problem and the actual submission files must have different name). For this purposes, the current implementation uses the versioning mechanism, which has misleading semantics with comparison of the one described above, because the versioned submission objects are not the same one (compare to scoreboard), where only the highest version is the actual, but they are entirely different objects, using the versioning only for differentiating them by name. The implementation of the second functionality would be probably the same (e.g. will lead to directory with submissions, which will contain directory "submission.1", "submission.2" etc), but, as stated above, there is semantic difference, which might be confusing and in the future versions of the API could be addressed and solved.

The last possible remark to the CML component is, that replacing the current data storage approach (raw files on disc accessed by relative path) by some kind of NoSql/XML database might still be an option, which might solve some of the issues described above (namely efficiency and ad-hoc changes). Cost for this change however is another technology, on which is system dependent on and increased complexity of the CML component. The discussion and research on this issue can be found in [1].

## 2.4 Judge component

This section describes the design and analysis process related to the Judge component, which is responsible for evaluating the submitted source codes. The state of the original version of the Judge component is described in chapter 1. List of required features and issues to solve follows, the analysis, design process and appropriate discussion can be found in the subsections below.

The minimal list of features, which must the Judge component provide looks as follows:

- Download the right submission from CML, using the synchronization mechanisms provided by CML API
- Compile the downloaded source codes
- Run the compiled program against test data
- Evaluate the results and upload them back to the CML (again with help of synchronization mechanisms).

### 2.4.1 Desired component features

During the consultations with supervisor and analysis of the previous state of the component were found following major issues or questions to be solved, respective answered. Found problems are presented and discussed below.

#### 2.4.1.1 Languages support

Supporting all features of the programming languages seems to be a non-trivial task, since every language has its own specific features. Languages might use different build mechanisms, e.g. for Java Ant, Maven or Gradle. They also have a different approach, when it comes the packaging or module organization. Lastly, they might have some specific language constrains, e.g. again for Java, the source file must contain only one top level class and the name of the file containing the class must be derived from the actual name of the contained class (the *javac* compiler might refuse to compile the source code, if the requirements are not met). The old version of the Judge component did not support other language than C/C++.

#### 2.4.1.2 Dynamic Judge configuration and contest independence

When discussing possible configuration strategies between author and supervisor, it turned out that it would be very nice, if the Judge component would be independent on the contest and if it would be able to handle different problem configurations dynamically. For example, the component might be able to first evaluate source code for the purposes of the ACM ICPC contest (which means download, compile, validate, upload submission) but than it should be able to evaluate the submission from another contest, which would consist of downloading two submissions representing chess solving programs, which would be validated against each other (better chess playing program contest). The original Judge architecture was not designed in any way for such purpose.

#### 2.4.1.3 Judge sub-components modularity

Another problem with the old Judge component is its approach to modularity and reusability of the components it has available for different purposes (different compilers, validators, modules for plagiarism checking etc). Or in another words, answering the question „How hard it will be to add new sub-component or module or functionality to the Judge component and how hard it will be to reuse it and integrate it with the existing ones?“. This is also a architecture aspect not considered in the old version of the Judge component.

#### 2.4.1.4 Secure execution of programs

Last feature, which should the Judge component provide, is the ability to run the submitted problems in a secure way. This means, that the execution

of the program should not lead to the crash of the system (or the Judge component) it runs on and that the submitted program should not be able to interfere (or cheat) the evaluation process, by e.g. reading the validation data or providing the user with inappropriate information by abusing e.g. signals etc. Since developing such feature (minimally for C/C++ and Java) is clearly out of scope of this work, some existing is expected to be used and integrated somehow into the component.

### 2.4.2 Programming languages

The complete solution of the languages problem from the list above is clearly beyond the scope of this work and, at least in this version of the system, should be sufficient to support C/C++ and Java languages in some simple form, so that it will provide similar functionality to the systems like DomJudge [11], *PC<sup>2</sup>* [12] or other similar systems.

For the current purposes, the author has decided to stick to the minimal language support. The system will accept solution in form of the single file for Java or C/C++ language. No build mechanisms or other advanced features will be supported.

### 2.4.3 Dynamic judge configuration

The solution of the dynamic Judge configuration (second item on the list in 2.4 section) was found in form of a Spring framework [13] and designing flexible interface between the Judge sub-components. And the key for the interface design was found in the realization, that the process of submission evaluation can be abstracted to the tree-like structure, through which the processed data (submission) will flow in the „pre-order” similar manner and where:

- The nodes of the tree will represent the „local” logic of the process and will be responsible for controlling the data flow of its child nodes. Such component might just pass the data from first child to the next one and when done, pass it to the parent or it can implement some more sophisticated logic like conditional flow, iterative flow or other operations. These sub-components were internally named **orchestrators**.
- The leaves of the tree will contain a sub-component, which will utilize the strategy design pattern and will contain actual implementation of some algorithm (compilation, validation etc) and will be responsible for converting the data, which flow through the tree, to some form needed by the contained algorithm implementation. These sub-components were internally named **units**.

The example of the tree is illustrated on diagram 2.2. The diagram 2.3 illustrates the interface between the unit and the actual algorithm implementation.

## 2. ANALYSIS AND DESIGN

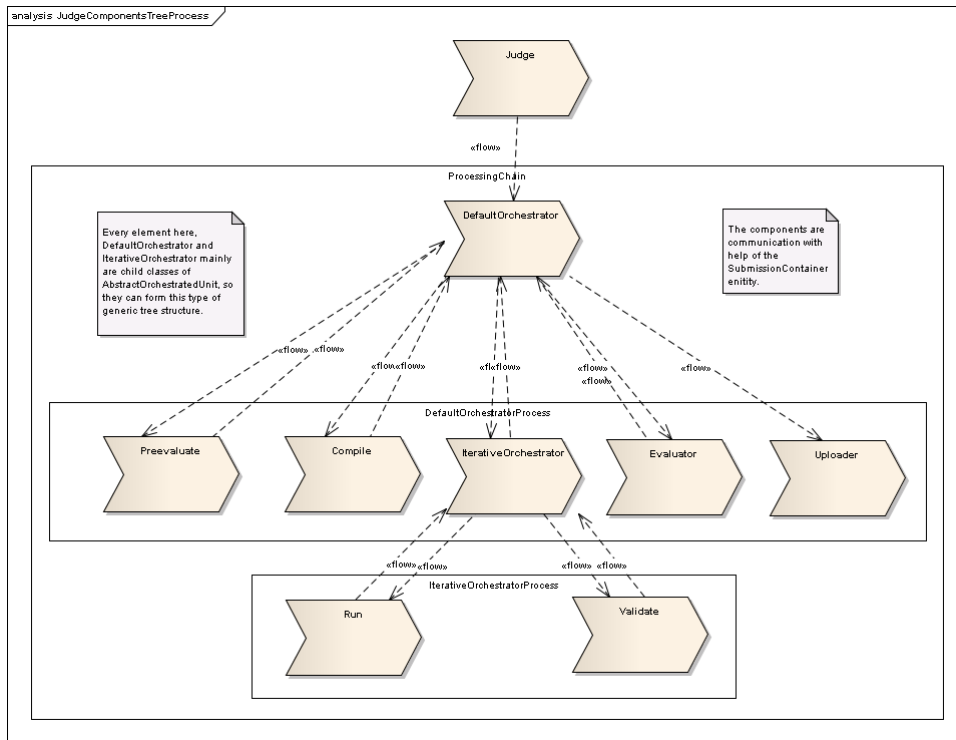


Figure 2.2: Illustration of the Judge sub components tree and the orchestrator/unit design.

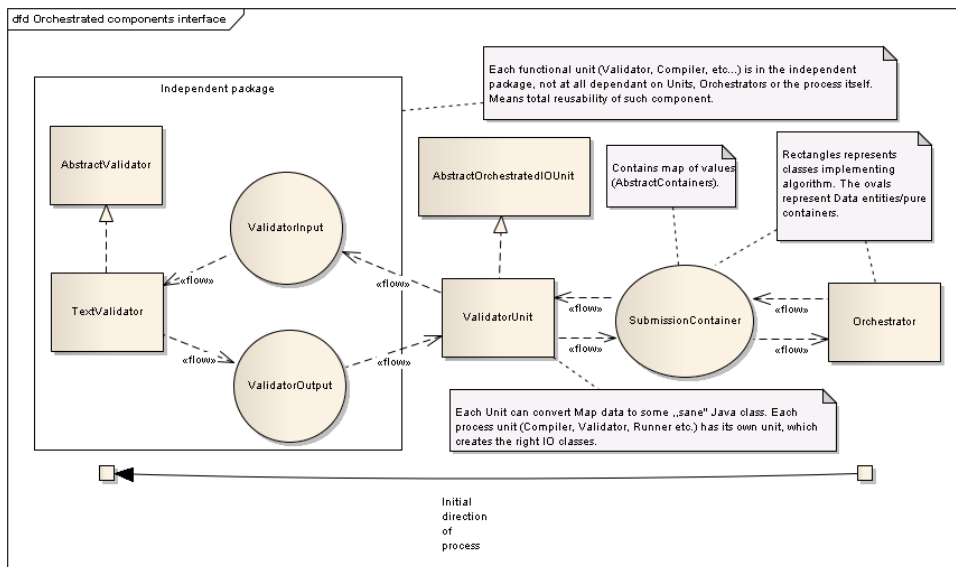


Figure 2.3: Illustration of the interface between the orchestrated unit and the actual implementation of the algorithm.



#### 2.4.4 Judge modularity

The third found issue with the old Judge component is the actual implementation of the interface between the subcomponents, which will determine the process of adding new sub-component and possibilities for their integration or other aspects like parallelization. In the old Judge were the sub-components communicating as the Java classes. In this situation, for adding the new component or functionality would be needed to implement new class or whole package, manually integrate them to the existing component and recompile whole package.

Following possible communication solutions between Judge subcomponents were found:

- as Java classes (old way)
- via HTTP directly
- via HTTP through local CML
- via local file system, emulating the CML locking principle

At first, it seemed quite tempting to utilize the existing API of the CML, either by locally deployed CML or emulating the CML API at local file system (for this purposes the Java NIO library [14] would be used). The advantage of this approach would be the recursive application of existing API on the lower level of the system, which would lead to somehow „clean and pure” system architecture — the component would be able to communicate with the file system, local CML, remote CML or any other component with the same API. But the big downside of this approach (aside from probable efficiency issues) would certainly lead to a very difficult configuration and maintainability (in comparison with the alternatives), because the subcomponents would be actually different processes.

The final solution presented itself naturally when solving the previous issue of the Judge architecture, namely when Spring and tree like component structure were designed. With this approach, adding new component would just mean writing the new module or class, writing new Spring configuration file utilizing this class and either recompile the Judge with new dependency or just update the JVM classpath.

#### 2.4.5 Secure execution

The last problem mentioned in the list in section 2.4 is the ability of the Judge component to execute the submitted program in a secure way. This feature should have been already implemented in the previous version of the system, utilizing the results of [2] (SRun component), but the author was unsuccessful to integrate the results of mentioned work to the previous version of the Judge

component or run the component independently from the command line (see [1] for details).

In this work, the author made another attempt to use the SRun component, but was unsuccessful again. The process was similar as described in [1] and it seemed, that there was a progress found, but the last error found was the system error „Requested resource is not available”, which happened at the forking section of the program. Solution of this problem was not found either by modifying the source code itself, nor in the software documentation.

It seemed, that attempts to use this components are very time consuming and the success is not guaranteed. The found solutions to this situation were either consultation (either with the author of the component or with someone with higher knowledge of low level C++ and Unix systems) or finding and using another one.

At this point, author decided to made a research of other existing solutions. The examined software was either from the other existing programming contest systems or general purpose sandboxing components or approaches. After a consultation with a supervisor, one special requirement on this component was identified in the form of a license choice — using the component should not end up in need to extend the license on the whole project (using e.g. GNU license [15] distinguishes between software composition and aggregation and in the latter case, at least the Judge component would have to be published under the GNU, if the secure component would be used).

The incomplete list of the software researched contains (totally unusable solution are omitted):

- timeout script [16]
- proot [17]
- component from DomJudge system [11]
- CMS [18]
- Moe [19] and Isolate component [20]

Author believes, that the most suitable one from the list above, is the Isolate utility [20] used in the (currently obsolete, but being rewritten) Moe contest system [19]. The authors of this system and utility are Martin Mareš (Charles university in Prague), Tomáš Gavenčíak and Bernard Blackham. The Isolate component provides various features, has simple installation and deployment and was practically used in programming contests. List of interesting features and settings follows:

- the program is executed in chroot environment, to which is possible to map specific system directories and files for various access.

- provides ability to limit resources (time, memory, disc block writes, process etc)
- can run in parallel in one system and provides support for system control groups
- because of the command line API, its GNU license means no problem, since it is the software composition and not aggregation.

The installation is very simple — it just needs to compile the program and set the SUID bit, than its ready to be used. The usage example for executing the Java program is illustrated on listing 2.1.

```

1 ./isolate --init
2 ./usr/bin/isolate --dir=/tmp/compiler --dir=/etc -e --mem=20 --
   time=4 --wall-time=4 --extra-time=2 --processes=32 --run /usr/
   bin/java -- -cp /tmp/compiler CorrectIO
3 ./isolate --cleanup

```

Listing 2.1: Example usage of the Isolate utility for Java

As stated, the isolate utility was tested in real programming contests. Regarding its security, authors claim [21] [22], that they did not encounter any security issues and bugs for the C or C++ programming languages. From the other compiled languages, they have got good experience with Pascal, for which they found only problem with slowness of the Pascal IO libraries. As for the interpreted languages, the authors were quite thoroughly testing the Isolate utility for purposes of executing C# programs (Mono platform [23] with some special hacks), where they did not encountered any problems, but do not guarantee the safety of the execution.

The authors of Isolate also tested Java, but encountered problems with the JVM, which consumes quite a lot of resources, e.g. namely processes. The consequence of this is, that it is currently impossible to restrict the executed Java program to create these processes, since the JVM wont start without them (in other words, more than one thread is needed to execute the Java „Hello World” program). The author of this work also encountered a strange issue, that it seems to be not possible to execute the Java with Isolate with less than 3000000kb of assigned memory (or at least, set the „-mem 3000000” parameter). On the authors testing system (Arch Linux 64bit [24], Intel i5, 4Gb RAM, Oracle JVM [25]), setting the parameter „-mem 250000” will lead to the JVM crash („Not enough memory for code segment.”). Author performed short experiments with JVM memory settings, but without any success. This issue should be kept in mind, when using the Isolate in real environment and certainly deserves more investigation. But for the current purposes, deploying the Judge component on the system with more than 4Gb RAM should not present any problem as for running it in the ACM ICPC contest context. The consequence however is, that under this circumstances, it

wont be possible to use the system for e.g. purposes, where memory handling may be a criterion for accepting or refusing solution, at least in the Java programming language.

One little drawback in the Isolate utility was found in the handling of memory limits. Author believes, that for the most purposes, the best strategy how to handle memory requests of program, which are crossing the given limits, is to kill the program and inform the user, that the test crashed on the memory exceeded error. By experimental attempts was found however, that the Isolate utility does not kill the program, but just refuses to allocate the requested memory — e.g. the C *malloc* function will return null. While this is certainly not error, the final program behaviour is dependent on whether the user checks, if the memory was returned and the final code or the system behaviour is not strictly defined (i.e. the user may not be informed, that the problem with the program was inappropriate memory handling).

At last, author would like to point out few advantages of using the Isolate instead of the SRun component:

- Isolate was tested in real environment, while SRun was not
- Isolate is clearly disproportionately easier to use, deploy and maintain, than the SRun (compare the SRun installation instructions from [2] and the process of installing Isolate above).

## 2.5 Web interface

This section describes the process of design and analysis of the component, which represents the user web interface for the main system user. The requirements on the component are stated in 2.5.1, review of the old state from [3] in 2.5.2 and design process is described in the 2.5.3.

### 2.5.1 Web interface requirements

The minimal set of requirements on this component is described in 2.1.1. The component should be preferably implemented in Java, since the rest of the system is implemented this language and it will allow to use some of the available functionalities like CML API library (see 2.8).

After a discussion it turned out, that it would be nice, if this component could be universal to certain extent, so that it would be possible to use this component in different contexts, as was discussed in the introduction chapter. Quick analysis showed, that these contexts share most of the functionality (upload source code and show related data), but main difference is displaying the score or results. For example, ACM ICPC basically computes score on boolean base (ACCEPTED, WRONG\_ANSWER, and than handling ties), but the school course can compute the final score in the percentual or „sum

of points” manner, where there can be different score value thresholds for accepting the solution as solved. Also, the actual score computing should be linked to the contest and not the actual problem itself, so that same definitions of the problems would be reusable between contests.

The universality described above also opens the question of a user management. While for the school course, the user management and authentication would be best to be done through the directory services, in case of the public programming problems library is this approach inapplicable and some more suited mechanism for user management will be needed.

### 2.5.2 Old state review

The previous web interface component was designed and implemented in [3]. It was written in Java using the Wicket framework [26].

The main problem found in this solution was, that it was designed and implemented without the universality principles described above in mind. The web interface in that version is designed for the purposes of the ACM ICPC contest, so that it e.g. does not support multiple ongoing contests at once or is implementing some ACM ICPC specific features (clarifications, judges, ICPC style scoreboard etc).

The last found complication is, that the implementation uses the obsolete data model, used in the previous version of PCSS and does not use any „translation” layer, which would make the UI independent on the old PCSS project.

Based on observations from this section, the author was of the opinion, that it would be best to implement new web interface from scratch, with the new evolved requirements in mind.

### 2.5.3 Design

First design decision lied on choosing the implementation framework. The author of this work has quite a little experience with writing the web interfaces (at this time), so little research and discussion were needed. Because learning a new framework from scratch concealed danger of bad framework usage, bad design decisions and seemed too time-consuming for the purpose, the author choose the Java Server Faces framework [27], with which he was familiarized during the related JavaEE course.

The problem with need of different scoreboard display algorithms was solved by designing abstract interface, which allows to create universal data structure, which must be interpreted prior to generating the score results to the user. More details in 3.1.2.

The proper solution of the user logging strategy was found in the Proxy (see 2.7) component from the previous version of the system. But as the correct implementation seemed beyond scope of this work and unnecessary for the

pilot version of the system, it was decided, that for the current time being will be users handled by the UI component and the user creation or management will be done via appropriate XML objects in the CML component.

After solving previous problems, the author begun to design the actual web interface structure. The process went as follows:

- analysis of the use cases 2.1.1
- creating task list from use cases
- creating lo-fi prototypes of the UI
- performed research of the UI of similar existing systems (Progtest [28], DomJudge [11], previous UI [3])
- revision of the lo-fi prototypes
- implementation of hi-fi prototype in HTML/Bootstrap [29]
- final implementation

Process details and created artifacts (like result of the task analysis or lo-fi prototypes) are omitted for time and space save reasons.

The final design is based on the assumption, that the user will spend most time in the area of the application, which represents one contest and with this concept in mind was designed the application navigation and data layout. The screenshots can be found in appendix E.

Aside from the requirements above, author implemented additional features, like contest superuser user role, logs from the evaluation process and other. See section 3.2 for more details.

## 2.6 Scoreboard component

The purpose of this component is to generate the data structure, which contains aggregated data representing the best submissions of the users in the contests. Motivation for implementing this component is firstly to optimize the data retrieval from CML when viewing scoreboard from the web interface and secondly can the scoreboard (if not now, than in the future) perform some kind of post processing of the result.

The component in the old version of the system was quite small and simple — its task was to download new submissions, incorporate them to the previous scoreboard version and upload the result back to CML.

Since the Judge component was now designed with contest and problem independence in mind (see 2.4), it seemed to be a good idea to apply the same concept to the new version of the scoreboard component. Thanks to the new scoring implementation (see 3.1.2) is the scoring logic abstracted from this

component and for scoreboard generation seems to be sufficient to use the prepared API for interpretation of the score and comparison the results. The implementation of the scoreboard for the particular contest only could also bring configuration complications, since the scoreboard component would need to be executed multiple times for the given contests or the running scoreboard components would have to be configured somehow to know contests, which they are supposed to handle. In comparison with the single (or parallelized) scoreboard component concept (and with help of the abstracted scoring API), this solution seems to yield no added benefit.

The final design of Scoreboard component is similar to the old one, with the difference that the algorithm of scoring is abstracted to separate module and the is now contest independent. See section 3.3 for implementation details.

## 2.7 Proxy component

The purpose of this component from previous version system is described in chapter 1. Because at design phase of the web interface was decided, that users will be handled by the UI itself (see 2.5), there seems to be no reason for using this component and therefore the current version of the system will not utilize this component.

## 2.8 Other system modules

This section contains analysis and design of the system data model and other system modules. The domain model design in described in the subsection 2.8.1 and found design and implementation issues are described in the subsection 2.8.3.

### 2.8.1 Domain model

Quick review of the domain model from the previous version of the system showed, that the old model can be used with just a few minor modifications, which reflect some additional system features (e.g. new scoring API or problem independence on processing structure). The new version of the domain model is illustrated on diagram 2.4. For the purposes of clarity hides the diagram some data entities, which are needed for the actual implementation.

### 2.8.2 CML API and data storage format

The previous version of the project contains API, which provides easier access to the CML data entities and other CML features (like synchronization). The original version of the API was three-layered. The first layer handled the CML communication on the HTTP level, the second on the data object level (performed conversion and initialization from/to Java class) and third

## 2. ANALYSIS AND DESIGN

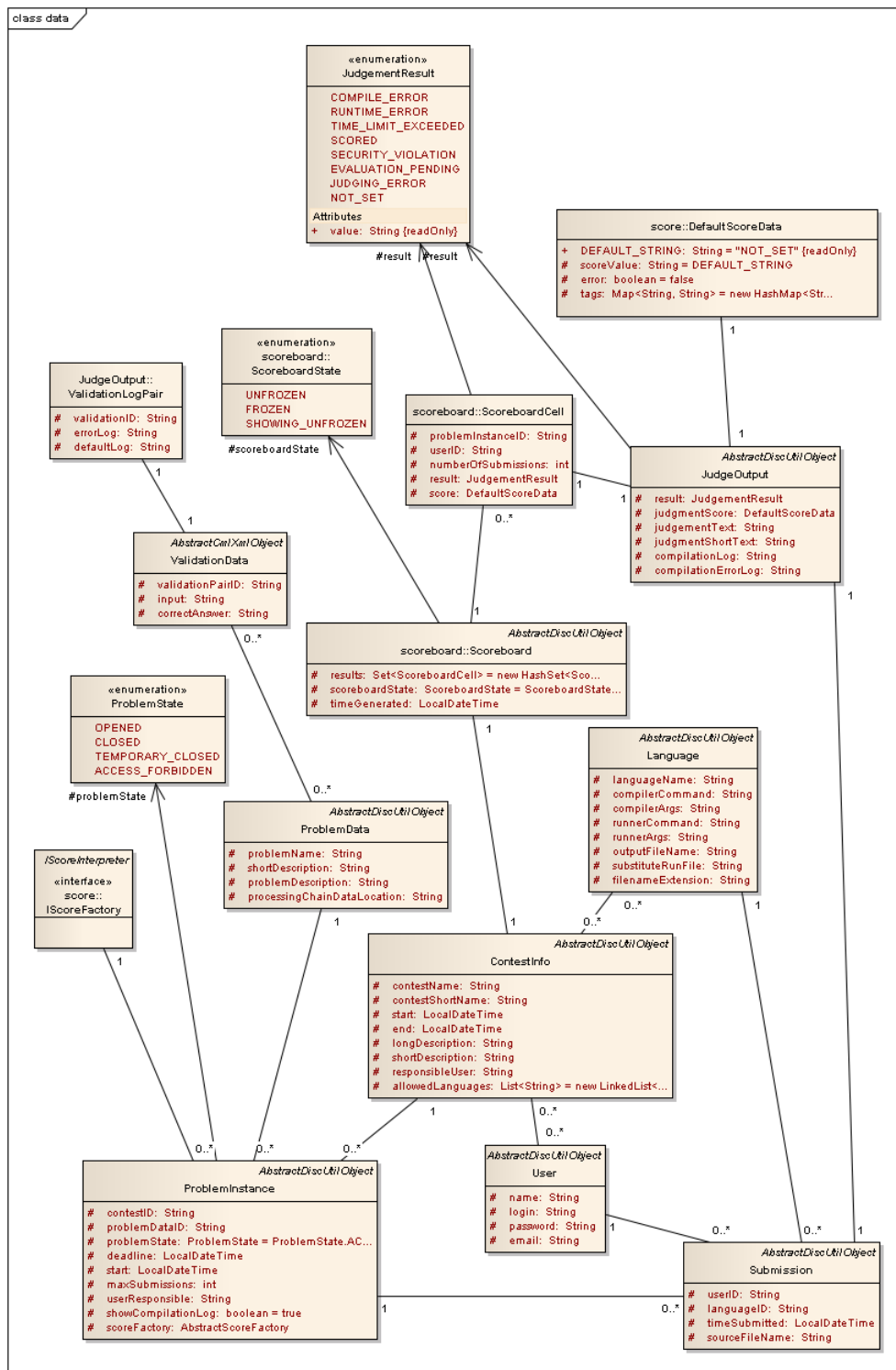


Figure 2.4: Domain model of the contest data.



layer provided support for component specific features (e.g. locking actually processed submission in Judge component).

Analysis showed, that the API based is sufficient for the current purposes, just one more layer was added (after the second one), which aggregates some functionalities shared by components and implemented in the original third layer.

Minor issue was found concerning the data storage format. The previous version of the system stored data entities in the text files, which contained data in multipart format. This approach was questioned in the previous version of the work — while it seems quite nice from the API perspective (utilizing standard HTTP only), it presents certain complications, since this format is quite hard to work with (for e.g. purpose of the manual data intervention) as opposed to the XML file. After a discussion, it was decided to migrate to the XML format of stored data. This change should also ease the transition from the raw file storage to some kind of NoSql/Xml database in CML component, if such need will be discovered in future.

### 2.8.3 Issues with relations mapping

During the work, major issue, related to the specific CML nature, namely object immutability and requirement to implement data object relations, was found, and the real actual impact was discovered quite late in process of implementation of the web interface. Because the object stored in CML should not be modified (except for rare exceptions), best solution to approach the relations problem was to rely on conventions rather than actual links. In that way, the data object, which contains the e.g. contest data, does not contain neither list of contest problems or submissions, but they are found by convention in their appropriate directories. Relations are handled by the appropriate component CML client, which will e.g. receive relative path of the contest on input and, with help of some relations mapping object, will return list of submissions (optionally filtered or post-processed in some way).

The actual problem is the implementation of the data mapper object, which handles relations of the data entities and should ideally be flexibly configurable and independent on the actual data contest (in case, that the CML data repository would be used for other purposes than PCSS system). Analysis showed, that with the current CML API, there is need to handle mapping of relative path of five different objects (see diagram 2.8.3 with structure of the contest directory for better illustration):

**Named file** Under the relative path is stored the appropriate XML file. Example entity is XML representing final judgment of submission.

**Named directory** Under the relative path is stored some named directory, which just helps to structure the data tree and does not contain any

actual data object. Example object is directory „problem-instances” in the contest directory.

**Versioned file** This object is used with the CML POST method. The actual relative path contains path and file name without version and CML returns specific version (typically the actual newest one) of the data object. Example object is XML file representing scoreboard in contest.

**Versioned composite object** This object is similar to the versioned file, but the actual relative path of the object is relative path of a directory without version, which contains some XML data entity and arbitrary amount of other objects as well. On request, the mapping API should return the XML object data entity, of which relative path will be computed from the relative path of the parent versioned directory (i.e. map relative path of versioned directory to relative path of the contained XML data entity). Such object in current implementation is represented by the submission object.

**Named composite object** This object is practically the same as the previous „versioned composite object”, but here the parent directory of the actual data entity is not versioned, but named specifically. In this case, that mapping entity must convert relative path of the directory which contains some XML entity (and other objects) to the relative path of the actual data XML object. Example object in current implementation can be the contest object.

The final version of the mapper object should be configurable in a way, which will allow to create the model of the file system (tree structure), and which will define which type of object is stored where and which type of entity is expected, when requesting an object on given relative path. For the configuration purposes would be probably used the Spring framework.

The benefits of the datamapper object as described above are:

- Separation of the mapping functionality from the actual component CML clients.
- Flexibility of the file system, which final structure would be flexibly configurable.
- easier use of the CML component for purposes other than the PCSS system (consequence of the previous item)

While author has a quite good idea of how to implement API described above (implementing factory for each object type and compose them to tree-like structure, which would be traversed with some kind of appropriate visitor),

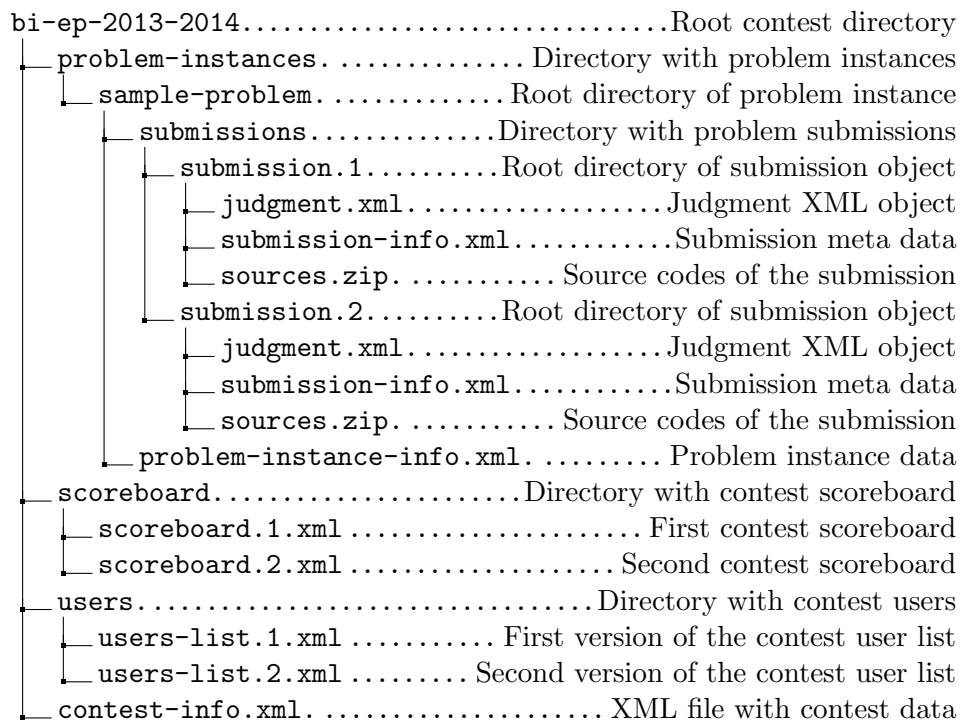


Figure 2.5: Illustration of the internal representation of the contest directory in the CML component.

actual, complete and reliable implementation seems to be out of scope of this work, since it would also be necessary to rewrite clients for all three main components of the current system implementation (Judge, Scoreboard and web interface).

Alternative solution would be to resign on some specific file system features, which would be easier to implement. The final result however would miss some of the features like named directory (i.e. the root directory with contest would not be named by the contest, as is illustrated on diagram 2.8.3, but would be named like „contest.1”, „contest.2” etc). That would ultimately lead to worse orientation in the file structure when browsed manually (by the user and not machine).

The current implementation uses hard-coded mapping approach and author is of opinion, that this implementation flaw should be addressed before implementation of the other components, which will rely on the CML API.



---

# Implementation

This chapter describes and discusses the process of realization of the system. Author will focus on discussion and system features introduction here, since the implementation details could be found in the documentation in the appendix of this document.

As was defined in the requirements, the system was implemented in the Java programming language, using the Maven [30] build system. The illustration of implemented system modules and module dependencies can be found on diagram 3.2. Illustration of modules communication can be found on diagram 3.1. The short description of modules follows, more details can be found in the sections below.

**CML** Server component, which serves as system data repository. Except bug fixes, the author did not perform any significant changes in this component.

**Judge** This package contains the Judge component, which is responsible for evaluating the user submissions (source codes). The analysis was described in 2.4, implementation is discussed in 3.1.

**WebUI** Contains the web interface for submitting solutions and viewing results. The analysis can be found 2.5, implementation discussion in 3.2.

**Scoreboard** Contains the component, which is responsible for processing and generating the contest scoreboard. The analysis can be found 2.6, implementation discussion in 3.3.

**CML Data** Contains various classes shared across the system and its components. In this module, author implemented the necessary domain model and simple abstraction of the data entities relations. This package also contains the scoring algorithm, since it is used in Judge, Scoreboard and WebUI components. The analysis can be found in 2.8. Discussion of implementation is omitted.

### 3. IMPLEMENTATION

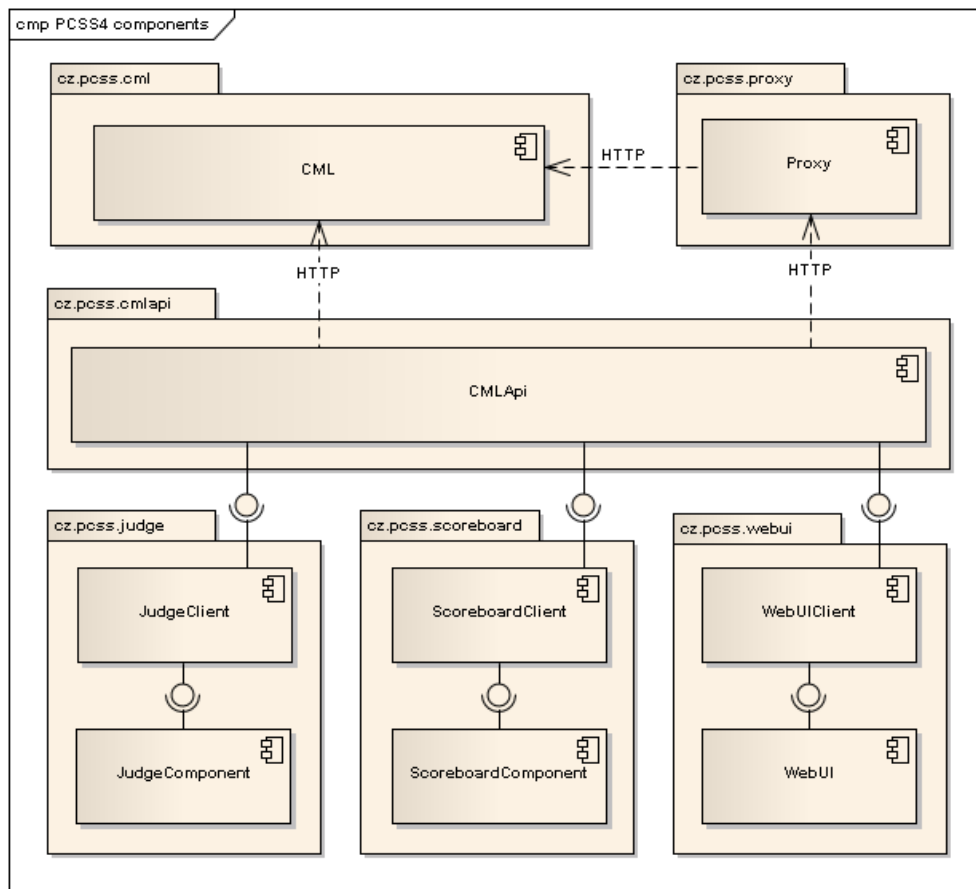


Figure 3.1: Illustration of modules communication.

**CML API** Classes in this module are providing API for communicating with the CML component. As described in the analysis 2.8, author reimplemented the original clients to the 3-layers, which provides shared API for CML communication by other components. Additionally were implemented changes, which were consequence of migration from multipart to XML format of the Data and fixed some bugs from the previous version of the API.

**Proxy** Contains proxy component, which was implemented in the previous version of the system and was not used in the current one. More details can be found 2.7.

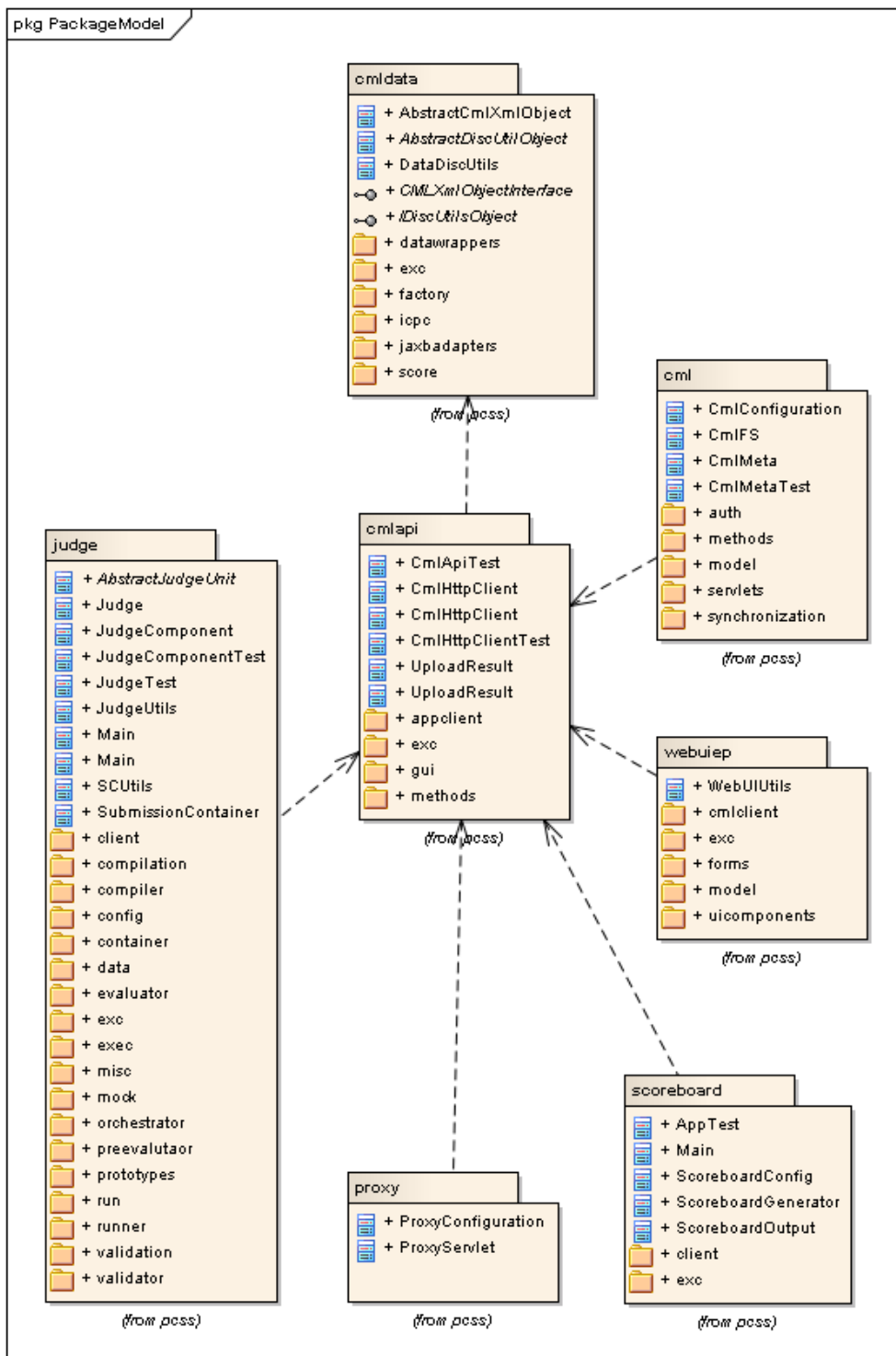


Figure 3.2: System modules and their dependencies.

### 3.1 Judge component

The Judge component was implemented as was described in the section 2.4. The quick overview of implemented features follows, their discussion can be found below.

- The Judge component is independent on the contest and the problem (i.e. the actual process dynamically changes with the evaluated problem).
- The component executes the submitted programs securely with help of the isolate utility.
- The component is currently able to evaluate submitted programs in C, C++ and Java programming languages.
- The Judge component was designed with the modularity needs in mind. Adding new component would usually mean to just implement one interface and add the class to the class path of the module.

The top level Judge process is illustrated on diagram 3.3.

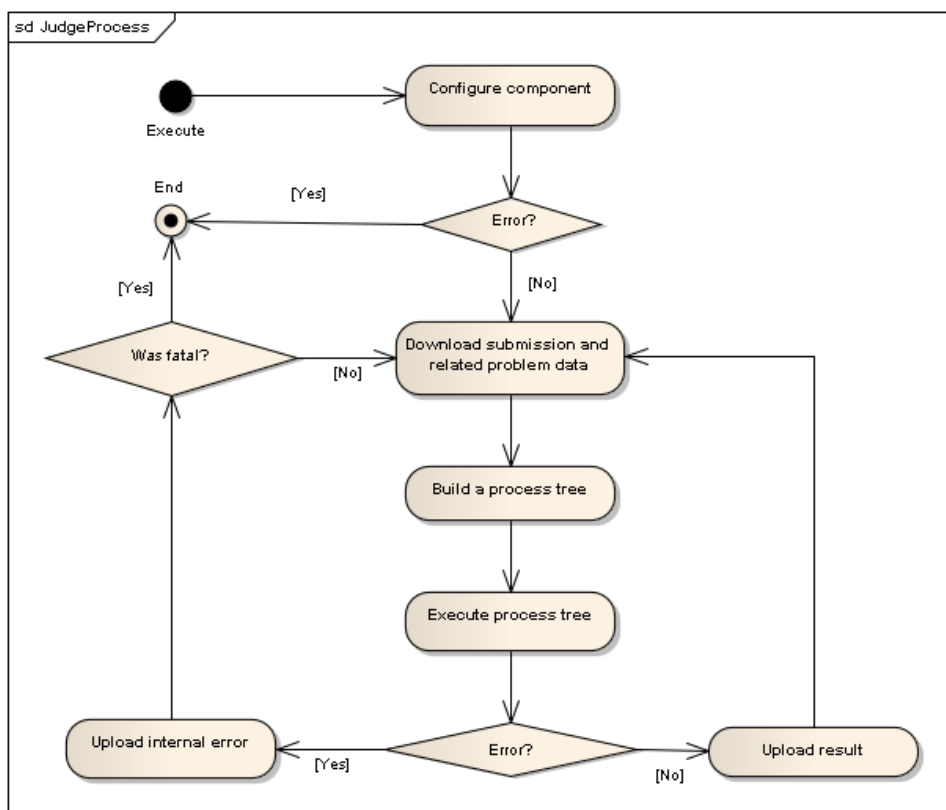


Figure 3.3: Top level Judge process.



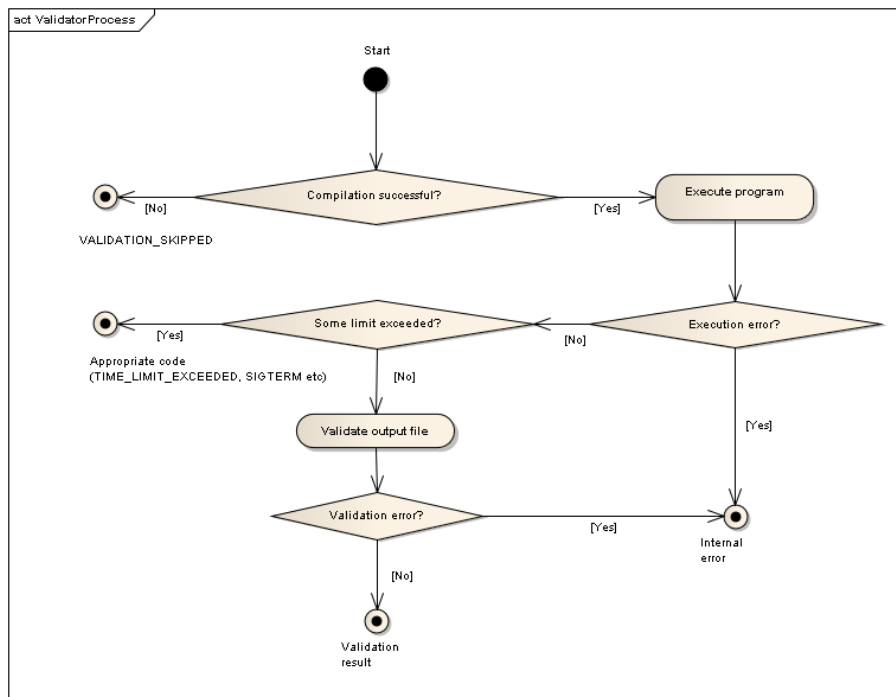


Figure 3.4: Illustration of the validation process.

### 3.1.1 Implementation of dynamic behaviour

As stated before, for implementing the dynamic behaviour of the component was used the Spring framework. The actual representation of the process tree is stored and downloaded from the CML component. After the initialization, the Judge component will create entity called „SubmissionContainer”, which is basically map of stored values of different datatypes. This map is then passed to the root of the tree, which should be usually *DefaultOrchestrator* component (see below). There are currently implemented three types of components for purposes of the processing tree — *Orchestrators*, *Units* and actual processing algorithm components. Their description can be found below.

#### 3.1.1.1 Orchestrators

Orchestrators are handling the data flow and contains list of assigned components (other orchestrators or units), implements the *IOrchestrator* interface.

**DefaultOrchestrator** This orchestrator just passes the work from one assigned component to other, in defined sequential order.

**IterativeOrchestrator** This orchestrator will pass the SubmissionContainer to the assigned components given number of times, also in sequential order.

#### 3.1.1.2 Units

Units are representing an interface between Judge component and independent processing algorithm. They contain particular implementation of some algorithm (strategy design pattern), handle conversion from and to data input of the particular algorithm and store result in `SubmissionContainer`. May handle exceptions, if needed. Units are implementing the `IOrchestratedUnit` interface)

**DefaultCompilerUnit** Generates needed input and output for classes, which implements the `ICompilerInterface`.

**DefaultValidatorUnit** Generates the input and output data containers for classes implementing the `IValidator` interface. The validation data are downloaded from the CML component.

**DefaultEvaluatorUnit** Generates input and output data containers for classes that implements the `IEvaluator` interface.

#### 3.1.1.3 Algorithm implementations

These classes are implementing some specific algorithm and are independent on the Judge component and the whole system itself.

**DefaultCompiler** Implements the `ICompilerInterface`. Serves for compilation of programs. The output of this component is either successfully compiled program or `CompilationError` result.

**DefaultValidator** Implementation of the `IValidatorInterface` and contains classes, which are responsible for secure execution and validation of the compiled program (strategy pattern again). The process may be skipped, if the validation failed. See diagram 3.4 for more detailed algorithm illustration. In the current implementation, the only validation strategy implemented is binary validator.

**DefaultEvaluator** Implements the `IEvaluatorInterface`. This class performs generation of the final judgment of submission (provided, that none previous component crashed). Current implementation will set the final Judgment result (enumeration) and will assign the final score (see 3.1.2). Finally, the evaluator will upload the submission data entity to the CML component along with another data entity called „Evaluation request”, which contains the relative path to submission.

The component diagram for the most common evaluation scenario „compile – validate – evaluate” is illustrated on figure 3.5.

### 3.1. Judge component

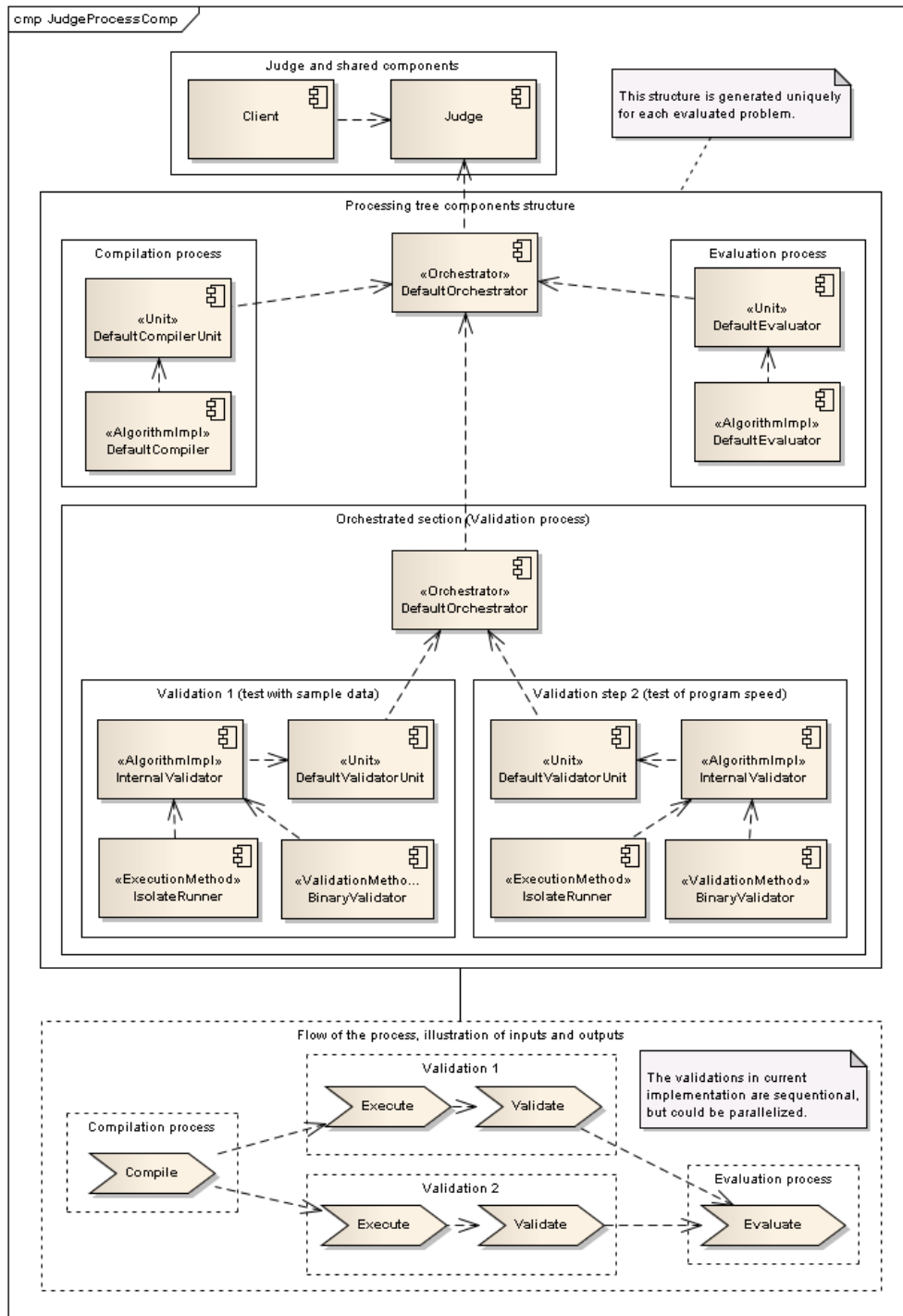


Figure 3.5: Illustration of the structure defining the submission processing algorithm with two-step validation.

#### 3.1.2 Scoring algorithm

As stated, it should be possible to use the system in different contexts and therefore we can assume, that the different contexts will have different scoring mechanisms and algorithms. For example, the ACM ICPC contest will show score in the „total solved problems” manner, while the system for evaluation of school homeworks might work on weighted percentual principle.

Because of this universality, author decided to implement the scoring mechanism using the interpretation principle. The data entity, which represents the instance of the contest problem, will have access to class implementing the `IScoreFactory` interface. This class will be able to produce some universal data entity (created from list of validation results), which will be stored in the final judgment. Anyone, who will need to compare two results (in this version the Scoreboard and Web interface components), will first give them to the original (or any other if need be) factory class, which will create interpreted result, which contains methods to e.g. compare two score results, determine if the scoring result is maximal or minimal possible etc.

This approach will allow to abstract the actual scoring algorithm from Judge, Scoreboard and WebUI and should allow implementation of different scoring algorithms. Another advantage of this approach is, that if some parameter of the scoring will change, it might not be necessary to reconstruct the whole scoreboard from scratch, since the changed parameter will manifest itself in the process of the interpretation.

The process and interface illustration can be found on diagram 3.6. In the current implementation of the system, author implemented two following scoring algorithms:

**Percentual scoring** This method will go through list of validations and will compute the weighed percentile of correct validations. The problem will be marked as solved in case, that the final value will be equal or higher than given threshold.

**Pedantic boolean scoring** This method will go through the list of validations and will set the problem as solved only in case, that all validations were correct and no error (like exceeded time etc) was found.

#### 3.1.3 Synchronization

Because it can be expected, that in the system will run more Judge components in parallel, there is need to synchronize them, so that the work wont be evaluated multiple times and there wont be any collisions when uploading the Judgments.

The synchronization mechanism is provided by the CML component in the form of subtree locking mechanism (see [1] for details). The work to be

### 3.1. Judge component

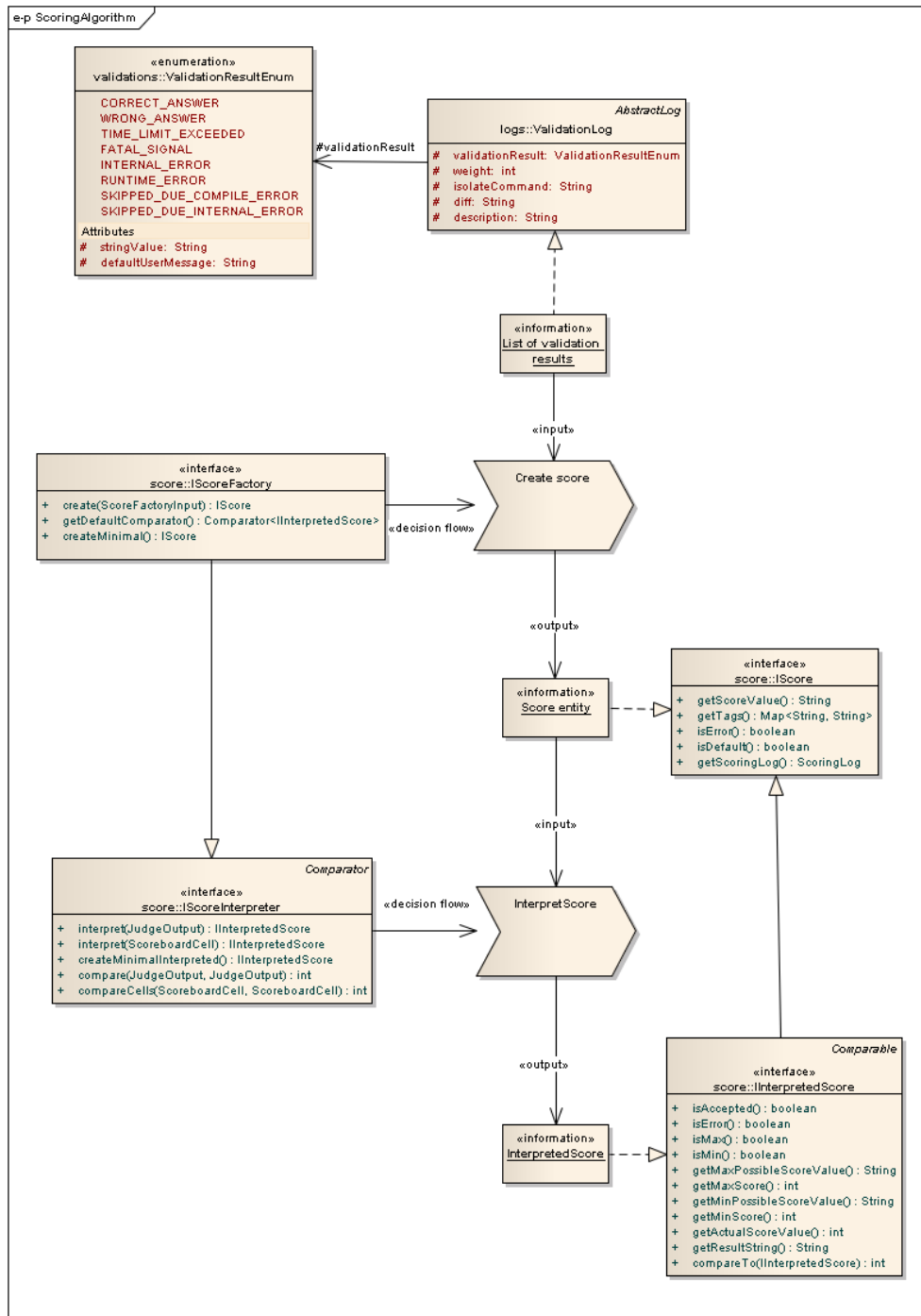


Figure 3.6: Illustration of the scoring algorithm.

evaluated is stored in the data entity called „Evaluation request” (contains relative path to the submission waiting for the evaluation) and each Evaluation request has data entity called „Evaluation ack”. When component asks CML for work, it will attempt to lock some evaluation request, which does not have its ack. If succeeds, than it will download it, process it, upload Judge output along with evaluation ack and then will release the lock. The whole process is optimized with help of the „modified-since” header, so that the CML will return only those evaluation requests, which the Judge component haven’t received yet.

## 3.2 WebUI component

As stated in analysis, the web interface was written from scratch. The component was implemented in Java, using the Java server faces framework. The design was created with help of Twitter Bootstrap.

The final implementation supports all requirements as defined in the subsection 2.5.1. After consultation with the supervisor, following use cases were added for all users:

- Showing logs of related process (e.g. compilation error log, validation logs etc).
- Ability to submit the solution as file and also as pure text.
- Refuse the user to see contest details, see contest problem and submit solutions if the contest did not started or the contest ended.
- Refuse the user to submit the solution if the maximum count was reached or the problem is closed (was set so by admin, problem did not start or problem ended).

Additionally was added one special user role (person responsible for contest), which has following privileges:

- can see all submissions of all users
- can submit any solution any time to any problem to particular contest, even if the limit was reached
- when submitting the solution, the solution will be evaluated normally, but the result will not be shown in the scoreboard of normal users.
- will have access to logs of the evaluation process, which are hidden to normal user

### 3.2.1 Result

The current version contains some flaws and has some specific features, which, as author believes, should be mentioned.

Firstly, the author did not test the web interface in the real environment yet and there is a possibility, that the component might have some performance issues, because of the architecture of the CML component and HTTP protocol. While it seems no issue in the Judge or Scoreboard components (because the actual evaluation algorithm will surely take no longer than about 10 HTTP requests to initialize the process), in the web interface, where there is need to show large quantities of related data (even if the user will ignore most of them), this might be an problem.

The discussion and possible solution of this performance problem is also described in 2.3.2, will be probably handled by some caching mechanism. The current implementation uses minimal caching, in which it does not download the same data entity for the request more than once.

Second issue lies in fact, that the current version of application does not use any mechanism for securing the passwords of the users — they are stored in the plain-text form in the CML component. This issue was addressed, but author decided not to secure them in this implementation. First reason is, that the CML will run somewhere in the private network and will communicate only with other system components — direct access will be denied. Second reason is, that the user management will be eventually implemented in the Proxy component, but in the later version of the system. Even though the state is not ideal, it seems to be secure enough for some small pilot deployment and therefore will be this issue solved in the future versions of the system.

Last remark concerns the implementation of the application logic. In standard web applications build in Java server faces (or JavaEE) is controller part of the application handled by Java server faces beans and the application logic by Enterprise beans (EJB). But since the current version of the application provides quite a little application logic and the use of the EJB would bring just another technology dependency to the project, without any found benefit, the author decided not to use the EJB and implement even the application logic layer in the with the help of Java server faces beans.

## 3.3 Scoreboard component

Because there were too many changes made in the system from the previous one, the author decided to rewrite the Scoreboard component from scratch and use just a relevant code snippets from the previous version.

As stated in analysis section of this component, the Scoreboard is as Judge component independent on contest and on the actual scoring algorithm, which is used for computing results. And also, as case of the Judge component,

there can be more than one Scoreboard component in the system, so the synchronization is needed.

The synchronization mechanism basically the same as described in the Judge component (see 3.1.3), with the difference, that in the role of the evaluation request is here the evaluation ack and in the role of the evaluation ack used in Judge is here the scoreboard ack. In this way, the scoreboard checks for any new evaluations of submissions by checking new Judge component evaluation ack and when the submission was processed and new scoreboard was generated and uploaded, the scoreboard will also upload the scoreboard ack, so that the submission wont be processed twice.

As for the process itself, the input data are the new submission, old version of the contest scoreboard and the scoring algorithm (stored in the problem instance data entity). The Scoreboard will let the scoring algorithm interpret the score and compare it with last result of the user for given problem. If the result is better, the scoreboard get updated with new score. If not, just a counter with submission count of the user for particular problem is incremented and the new scoreboard is uploaded to the CML component.

#### 3.3.1 Space efficiency considerations

The current implementation uses the versioning mechanism provided by the CML component for the scoreboard storing. This may result in unnecessary space usage in the CML component in case of the long-running contests, because as the consequence, there is currently stored every version of the contest scoreboard, but read is usually only the last one.

The solution of this problem may be to implement some mechanism to limit amount of versioned objects in the CML component.



---

# Testing

This section describes tests, which author performed in the current version of the application. The section 4.1 describes automated testing of the Judge component. The section 4.2 describes the manual testing of system features through the web interface. The section 4.3 contains quick remark about the state of the CML component.

## 4.1 Automated testing

Automated tests were written only for the Judge component. The functionality of the WebUI and Scoreboard components were tested manually (see the 4.2).

Author implemented 24 unit tests for the Judge component, each testing some component implementation in the black-box manner and then there are 5 tests, which tests the Judge component as a whole against mock CML client.

For the purposes of testing, the author implemented simple Mock instances, which represent one sample problem and 5 submissions, representing 5 basic source code types:

- accepted in C++
- wrong answer in C++
- accepted in Java
- time limit exceeded in C++
- memory limit exceeded in C++

The performed component and Judge tests are described in the following composite list:

- The tests for the compiler includes:

- Test of compilation of „Hello world” program in C++, where is tested, that the binary exists, compiler returned code 0 and that standard and error outputs of program execution exists
- Compilation of Java „Hello world” program, where is tested that given class file exists, *javac* returns the code 0, and the standard and error output logs exists
- Tests of C++ „Hello world” program, which tests if the binary exists, is executable and after execution returns the correct „Hello world” string
- Same test as the previous one, but in custom set output directory
- Tests of basic two units:
  - The `DefaultCompilerUnit` correctly initializes the input data container from generic `SubmissionContainer`
  - The `DefaultOrchestrator` unit passes the container correctly to 3 mock sub-units

Tests of default execution interface wrapper (used for execution of programs for compilation and validation purposes):

- The executed binary C++ „Hello world” program is correctly executed and returns correct string
- Same as above, but in custom working directory
- Same as previous ones, but with extended testing of the directory content (binary, output files etc. . .)
- Test, that the working directory is correctly cleaned, after execution
- Tests for the `Isolate` wrapper are testing five binaries of the sample contest problem. Tested values are result codes `OK`, `TIME_LIMIT_EXCEEDED`, `SIG_TERM` and `INTERNAL_ERROR` of binaries, representing following cases:
  - correct answer in C++
  - wrong answer in C++
  - correct answer in Java
  - time limit exceeded in C++
  - memory limit exceeded in C++ (ends up in segfault due to the `Isolate` behaviour)
- The tests of the validator is also based on submissions mentioned above, tested values as `ACCEPTED` and `WRONG_ANSWER` of submissions:

- correct C++ program
- correct Java program
- invalid (wrong answer) C++ program
- The tests for the Judge component and default process tree contains test of 6 basic submissions, which is tested for not throwing any exception, not crashing and returning the correct results (tested are Judge output codes SCORED/COMPILE ERROR and score output ACCEPTED/WRONG\_ANSWER), for submissions, which represents the following cases:
  - correct answer in C++
  - wrong answer in C++
  - correct answer in Java
  - time limit exceeded in C++
  - memory limit exceeded in C++ (ends up in segfault due to the Isolate behaviour)
  - uncomilable code in C++

## 4.2 Manual testing

The system was also manually tested through the web interface to ensure, that the stated requirements on the system are fulfilled and system provides those features. The tests were performed on top of the sample data, which can be found on CD. These data contains:

- 4 contests (2 in progress, one finished and one which has not started yet)
- one problem definition to which all contest problem instances are linked
- 4 users, from one of them is the user responsible for the contest
- other necessary data to run the system (e.g. language definitions etc)

On these data and final version of the system, following tests were performed:

- User does not see any contest, unless logged in, user also cant access any data on any contest subpage (e.g. problem page) unless logged in
- User can log in with his credentials
- After logout, user is redirected to the index page and the cache is cleared (tested from the contest page, problem page and scoreboard page of running contest, see below)

- User sees list of all contests, to which has access to
- For the contest, which did not started yet:
  - User cant access contest main page, scoreboard or any other contest sub-page, unless is in privileged role
  - If in privileged role, he can access all contest data and upload submissions, even though the submissions max count is reached
  - The Judge and Scoreboard component are processing the submissions of privileged user, as if the contest was running
  - The submissions of privileged user are processed correctly for the Percentual scoring algorithm, by both scoreboard and Judge components, with correct results (compile error, wrong answer, time limit exceeded, memory limit exceeded, partially correct and correct in C++ language, in Java language for the correct solution). All tested submissions were accepted either via file or via text input.
  - The logs of the processed submissions are correct (i.e. are displaying correct compile input or output, source code, isolate command, validation result, diff in case of partially correct submission and formula for scoring computation in case of Percentual factory). The source code can be downloaded as file.
  - The scoreboard shows correct results (submissions count and percentual result) in accordance with the Percentual scoring factory algorithm.
  - The web interface will not crash and will display correct empty data entities, if the components Judge and Scoreboard are not running (i.e. empty scoreboard, minimal score etc).
- For the contest which has ended:
  - User can view contest description, problems, problems description, contest scoreboard and his submissions with their detailed log, to which has access to
  - User cant upload new submissions, unless he is the privileged user
  - For the privileged user does the contest behave as if the contest is in progress (can view everything, can submit)
  - Privileged user has access to all contest submissions
  - Unprivileged user cant access page with all contest submissions
  - The uploading of the submissions behaves correctly (same results as in the contest which did not started for privileged user, see above)
  - The privileged user sees the whole scoreboard, the unprivileged user sees the scoreboard without the results of the privileged user

- The scoreboard corresponds to the expected output and are correctly sorted.
- For the contest, which is in progress
  - Every user sees all his submissions and their appropriate logs, contest main page, problems descriptions, scoreboard
  - Every user can submit, if the problem is in progress, is not closed and the maximum submissions count is not reached
  - Privileged user can submit always
  - Privileged user sees all submissions of all users in the contest
  - Unprivileged user cant access page with all contest submissions
  - The privileged user sees the whole scoreboard, the unprivileged user sees the scoreboard without the results of the privileged user
  - Tested submissions behave correctly, as in the case of privileged user in the contest, which did not started yet.
  - The scoreboard corresponds to the expected output and are correctly sorted.

### 4.3 CML testing

As stated before, the thesis [5] has performed some thorough testing of the CML component. Author briefly examined the work and found out, that the component might contain indispensable amount of bugs in the CML API, although the result might quite easily reflect the not ideal state of the CML API documentation.

Nevertheless, author did encounter only minimal amount of minor bugs in the current work and fixed them immediately — after the fixes, the CML component behaved as expected and no other problems were found. Since the CML component was not the primary aim of this work, author did not investigate the found issues further, but in the future, maybe before further CML development, this issue should be addressed and results of the mentioned work reviewed.



---

# Conclusion

In this thesis, the functional pilot implementation of the PCSS evaluation system was created. The system is ready to be tested in some real environment, preferably in the smaller school course or programming contest. For these usage scenarios, only minimal additional implementation should be needed (specific scoring algorithm and missing validation methods for specific problems). Because of the „smaller nature” of the event, it will be also easier to address any unexpected system issue that might arise.

From the author’s perspective, the strongest part of the system is its modularity, extensibility and flexibility. The logic of the system’s processes seems to be well separated, Judge component is highly configurable and the architectural style seems to be based on simple and clearly understandable principles. Because of the mentioned features, the result seems to fulfill the requirement on system „domain independence”.

The main system drawback seems to be the implementation part of the system and related system „immaturity” — i.e. the system currently does not offer as many features as it could. The main reason of this is the fact that author decided not to use any result of the theses from the previous years related to the system, namely the framework for secure program execution [2] and the web interface for the ACM ICPC contest [3]. This decision cost the author time that could have been invested in the other parts of the system (probably system infrastructure and Judge component).

Quite a lot of time was also consumed by using the immature CML data repository that meant e.g. manual handling of the data relations in the implementation stage of the work. Using some existing solution with some object related mapping such as Hibernate [31] would probably yield faster and more comfortable work with data model. (Nevertheless, using the CML component as the system data repository is, from the author’s point of view, still better for the system in the long term and CML drawbacks will be hopefully solved by future development.)

The implementation also contains certain sections with which the author is not entirely satisfied. They do not present problem in the terms of system functionality, but the author would like to rewrite them nevertheless at the first opportunity (as an example can be used the configuration of programming languages).

Overall is the author quite content with the final result. Author also hopes that he will be able to contribute to the system development in the future and that the system will also attract other contributors, since (as can be seen from the text above or from the following section) there is a lot work, that can be done via other bachelor's or master's theses, or some school courses projects as well.

## Future work

As for the future development, there are numerous ways to enhance the system. From the view of the system infrastructure, the author believes that the most crucial part is the implementation of the data model. The current implementation is not configurable and, mainly from the developer's perspective, is very difficult to work with. This fact is reflected in the CML API module. Because all of the data processing components of the PCSS are dependent on this system part, author believes that unnecessary delay in dealing with this issue might bring complications or unnecessary work in the future.

Issue related to the one mentioned above is the absence of some tool for creating the system data entities (e.g. contest, problem etc...).

CML component itself is also far from finished, even though practically usable and reliable for the purposes of the current implementation of PCSS. The component would mainly benefit from solving the security and efficiency issues. The API of the CML seems sufficient, the only possible bigger enhancement here seems to be the ability to filter and return more than one data entity. Review of the [5] that was testing the CML API would be also helpful. Another possible enhancement is the implementation of some separate component (tool) that would allow to utilize the CML API responsible for the manual data intervention.

A lot of work can be done in the Judge component. From the perspective of the offered features, numerous modules might be implemented, such as mentioned plagiarism checker, module for static code analysis, support for more languages, more validation methods...

As for the user web interface, the main drawback is lack of usability testing that might reveal some design flaws or suggest enhancements for better user experience. As in the CML, this component would also benefit from implementation of some form of caching mechanism. Some tasks and UI features could be also moved from the server to the client (AJAX) that would also lead to better user experience.



---

## Bibliography

- [1] Hauk, T.: *Základní komponenty vyhodnocovacího systému PCSS*. Bachelors work, Czech technical university in Prague, 2013.
- [2] Šilhavý, J.: *Základ systému pro automatické hodnocení programů*. Bachelors work, Czech technical university in Prague, 2012.
- [3] Benák, T.: *Webové rozhraní systému vyhodnocování programátorských soutěží*. Bachelors work, Czech technical university in Prague, 2013.
- [4] ACM ICPC International Collegiate Programming Contest. January 2016. Dostupné z: <http://icpc.baylor.edu/welcome.icpc>
- [5] Dmitriy, B.: *Automated Test of of Programming Contest Evaluation System*. Bachelors work, Czech technical university in Prague, 2015.
- [6] Fielding, R. T.: *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000. Dostupné z: <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
- [7] Fielding, R.; Gettys, J.; Mogul, J.; aj.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), Červen 1999, updated by RFCs 2817, 5785, 6266, 6585. Dostupné z: <http://www.ietf.org/rfc/rfc2616.txt>
- [8] Borenstein, N.; Freed, N.: MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies. RFC 1341 (Proposed Standard), Červen 1992, obsoleted by RFC 1521. Dostupné z: <http://www.ietf.org/rfc/rfc1341.txt>
- [9] NoSql. January 2016. Dostupné z: <http://nosql-database.org/>
- [10] XML Path Language (XPath) 2.0. January 2016. Dostupné z: <http://www.w3.org/TR/xpath20/>

## BIBLIOGRAPHY

---

- [11] DomJudge. January 2016. Dostupné z: <http://domjudge.sourceforge.net/>
- [12] PC2. January 2016. Dostupné z: <http://pc2.ecs.csus.edu/>
- [13] Spring framework. January 2016. Dostupné z: <http://projects.spring.io/spring-framework/>
- [14] JavaSE7 NIO library. January 2016. Dostupné z: <https://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html>
- [15] GNU General Public License v3. January 2016. Dostupné z: <http://www.gnu.org/licenses/gpl.html>
- [16] Timeout script. January 2016. Dostupné z: <https://github.com/pshved/timeout>
- [17] Proot. January 2016. Dostupné z: <http://proot.me/>
- [18] CMS Contest Management System. January 2016. Dostupné z: <http://cms-dev.github.io/>
- [19] The Moe Contest Environment. January 2016. Dostupné z: <http://www.ucw.cz/moe/>
- [20] Isolate. January 2016. Dostupné z: <http://www.ucw.cz/moe/isolate.1.html>
- [21] Mareš, M.; Blackham, B.: A new contest sandbox. *Olympiads in Informatics*, ročník 6, 2012: s. 100–109. Dostupné z: <http://mj.ucw.cz/papers/isolate.pdf>
- [22] Mareš, M.: Moe — Design of a Modular Grading System. *Olympiads in Informatics*, ročník 3, 2009: s. 60–66. Dostupné z: <http://mj.ucw.cz/papers/eval2.pdf>
- [23] Mono — Cross platform, open source .NET framework. January 2016. Dostupné z: <http://www.mono-project.com/>
- [24] Arch linux. January 2016. Dostupné z: <https://www.archlinux.org/>
- [25] JavaSE downloads. January 2016. Dostupné z: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- [26] Apache Wicket. January 2016. Dostupné z: <http://wicket.apache.org/>
- [27] Java Server Faces Framework. January 2016. Dostupné z: <http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html>

- [28] Progtest. January 2016. Dostupné z: <https://progtest.fit.cvut.cz/>
- [29] Bootstrap. January 2016. Dostupné z: <http://getbootstrap.com/>
- [30] Apache Maven Project. January 2016. Dostupné z: <http://maven.apache.org/>
- [31] Hibernate. January 2016. Dostupné z: <http://hibernate.org/>
- [32] GlassFish Server Open Source Edition. January 2016. Dostupné z: <https://glassfish.java.net/>



## Acronyms

**ACK** Acknowledgement

**ACM** Association for Computing Machinery

**API** Application program interface

**CML** Data repository of PCSS

**EJB** Enterprise Java Bean

**GUI** Graphical user interface

**HTML** Hyper Text Markup Language

**HTTP** Hypertext Transfer Protocol

**ICPC** International Collegiate Programming Contest

**PCSS** Programming contest scoring system

**REST** Representational State Transfer

**XML** Extensible markup language

**UI** User interface



---

## Contents of enclosed CD

example-data .....	directory with example data
├─ configurations .....	example component configurations
├─ data .....	example CML data directory (contest, problems)
├─ sources .....	source codes of example problems
exe .....	the directory with executables
src .....	the directory of source codes
├─ isolate .....	directory with the Isolate utility sources
├─ pcss .....	directory with PCSS source codes
├─ thesis .....	the directory of $\text{\LaTeX}$ source codes of the thesis
text .....	the thesis text directory
├─ thesis.pdf .....	the thesis text in PDF format
└─ readme.txt .....	the file with CD contents description





---

## Deployment instructions

Sections in this chapter are describing process of deployment of the PCSS system (there is section for each component). Note, that no component can run without the CML component, therefore the deployment of the CML is mandatory.

The configuration file and example data templates can be found on the CD. See section D.1 for details of component configuration.

The example of deployment is illustrated on diagram C.1.

### C.1 CML deployment

1. Build the CML component if not built already.
2. Setup and run appropriate application server, which supports basic JavaEE and Java server faces framework. Author of this work used the Glassfish application server [32].
3. Create new directory for the CML component on the local filesystem, in which the CML will store the data entities. In this directory, create the directory named „data”.
4. (Optional) In to the CML directory, copy the example initial „data” directory from the „example-data” folder on the CD.
5. Modify the „cml-config.properties” java properties file (located in project resources in „CML/src/main/resources”), so that it will contain correct path to the created CML directory (the directory created in step 3, which contains the „data” directory)
6. Deploy CML on the application server and check, that the application runs correctly on the appropriate URL. Upon access, the contents of the „data” directory should be returned in the form of the list.

## C.2 Judge deployment

First, there is need to setup the isolate utility, than the Judge component itself.

### C.2.1 Setting up the Isolate utility

- Compile the isolate utility on the CD (src dir) via the `make` command
- Set the SUID bit to the generated binary
- Test, that the isolate works via added bash scripts `test-c.sh` and `test-java.sh` (the correct result is string of type „You entered value: ...). The C script should always work, the Java script might need adjustment based on the system. Consult the isolate documentation in case of trouble.

### C.2.2 Deploying Judge

1. Build the component if not built already.
2. Update the Judge configuration XML file (example structure and settings available in file „default-judge-config.xml” located in directory with example configurations). Main parts here are the URL of the CML component and path to the Isolate binary from step C.2.1
3. Deploy the configuration file either in a remote location (must be accessible via URL) or to a local file system
4. Execute the Judge component by one of the following commands.
  - `java -jar Judge.jar /home/user/.config/judge-cfg.xml`
  - `java -jar Judge.jar http://server.com/cfg/judge.xml`

## C.3 Scoreboard deployment

1. Build the Scoreboard component, if not built already.
2. Update the Scoreboard configuration XML file (example structure and settings available as file „default-scoreboard-config.xml” in the directory with example configurations).
3. Deploy the configuration file either in a remote location (must be accessible via URL) or to a local file system.
4. Execute the Judge component by one of the following commands:

- `java -jar Scoreboard.jar /home/user/.cfg/scoreboard.xml`
- `java -jar Scoreboard.jar http://localhost/scoreboard.xml`

## C.4 Web Interface deployment

1. Build the project, if not built already.
2. Modify bean „applicationSettings” in the „WEB-INF/faces-config.xml” file to suit the needs of the deployment (most importantly it must contain correct URL of the CML component).
3. Deploy the project to the application server, which is able to handle JavaEE and Java server faces technologies (author was again using the Glassfish server).

### C. DEPLOYMENT INSTRUCTIONS

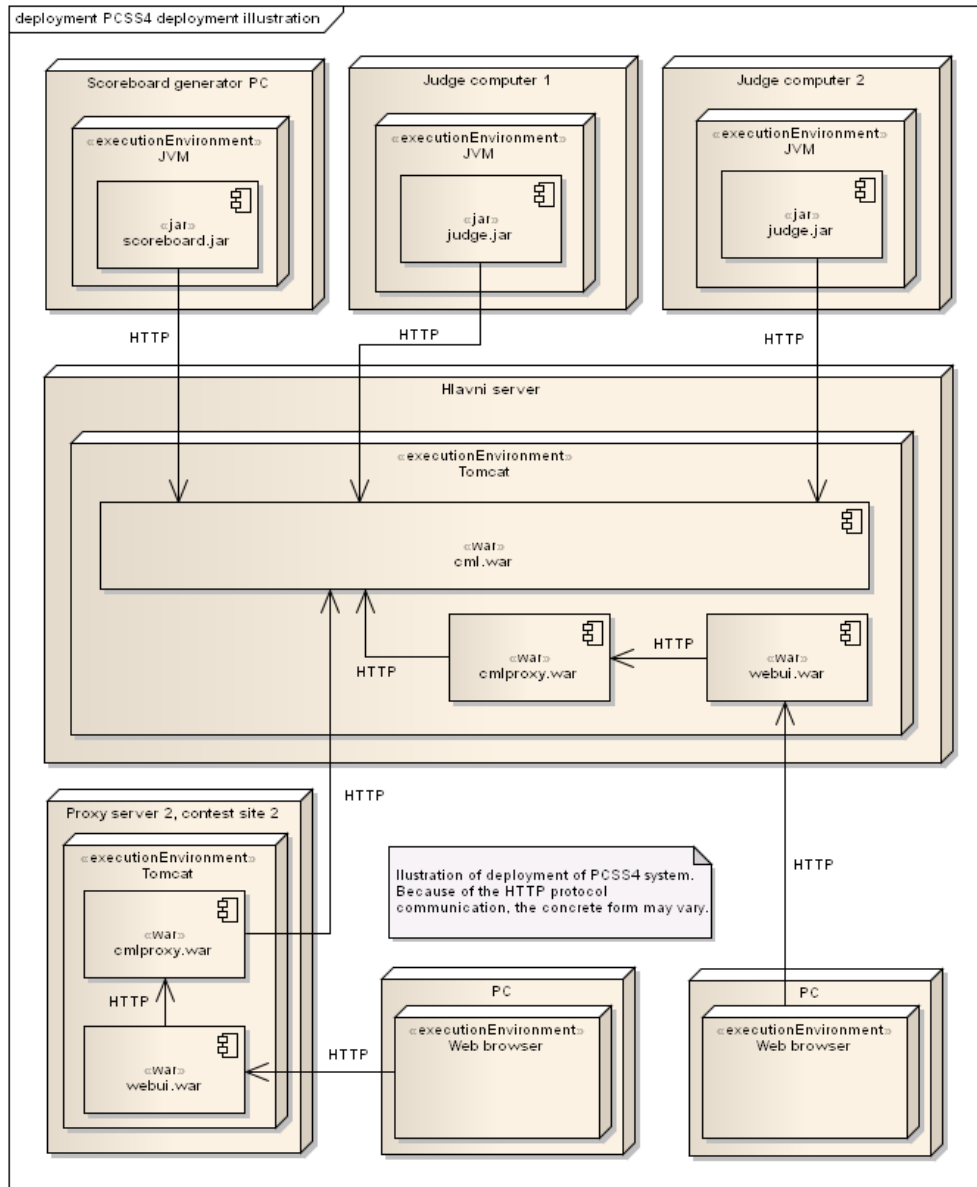


Figure C.1: Illustration of deployed system components. Taken from [1]

---

# Documentation

This chapter contains documentation of important parts of the system.

## D.1 Component configuration

The subsections below are describing process and the actual format of files that are used for the configuration of system components.

### D.1.1 CML configuration

The CML component is configured by the „cml-config.properties” file located in the „src/main/resources” project directory. File is in the Java properties format (key value principle). The configuration file contains only one parameter called „dataDirPath”, which must contain path to the to the CML reserved directory on local file system.

The CML component must have read and write access to the mentioned reserved directory.

In the CML directory, there must exist another directory called „data”. Files in this directory can be accessed via CML API. The files in this directory may be of any type, but because of the current implementation of the versioning mechanism, they cannot have any extension.

For more information, see [1] which will contain actual information, except changes mentioned in this text.

#### D.1.1.1 Example configuration

- The CML directory is set to „/home/user/.cml”
- The stored data entities of the CML component are stored in the directory „/home/user/.cml/data”

- parameter of the configuration file „dataDirPath” will be set to path „/home/user/.cml”.
- CML will be deployed on the local application server and will be accessible via URL: „http://localhost:8080/cml”
- Let there be a data entity stored in „/home/user/.cml/data/cfg/file”, this file will be accessible for e.g. GET method on the URL (of the CML): „http://localhost:8080/cml/cfg/file”

### D.1.2 Judge configuration

The Judge component is configured via XML file. This file is passed to the component either as URL (in case, that it is stored on a remote location) or as a path to file on local filesystem. The following configuration options are currently supported (example configuration file on listing D.1):

**cmlURI** This contains URL of the CML component. The Judge component will quit, if the connection fails.

**evaluationRequestsLocation** This parameter contains relative path of the CML directory, in which are stored versioned entities called „Evaluation requests” which are representing input data for the evaluation process.

**isolateCommand** Contains absolute path to the binary executable (or sym-link to the binary) of the Isolate utility on local file system. Component must have rights to read and execute this file. The file must have set an SUID bit.

**isolateLimits** This configuration section contains currently supported „limits” settings of the Isolate utility. The parameters are (see Isolate documentation [20] for more details):

**memLimit** Maximum memory limit for the program executed by isolate, in kB. Corresponds to the Isolate „-mem” switch.

**timeLimit** Maximum time limit in seconds for the executed program, should count only the real CPU time. Corresponds to the „-time” switch”.

**timeWallLimit** Wall-clock time limit (real time of the execution), measures time even if the program lost the CPU. Use this to handle e.g. sleeping programs. Corresponds to the ”-wall-time” switch.

**extraTime** When a time limit is exceeded, wait for extra time seconds before killing the program. Corresponds to the „-extra-time” switch.

**discQuota** Set disk quota to a given number of blocks and inodes. Corresponds to the „-quota” switch.

**processesCount** Maximum number of processes which the executed program is allowed to create. You should set more than 1 for running Java programs, since JVM will create multiple processes even for simple „Hello World” program. Corresponds to the „processes” switch.

**isolateSettings** This parameter contains a string of Isolate settings which are passed to the Isolate as a command line parameter. Use this to e.g. map specific parts of the host file system to the executed program. The final Isolate command will look as follows:

```
isolate <ISOLATE-SETTINGS> --run binaryFile
```

**mainOrchestratorBeanID** Contains name of the bean in the Spring configuration file which defines the process tree of evaluation in Judge component. See D.2.2.3 for details.

**waitingTime** Represents number of seconds the Judge component waits before sending next work request to the CML component in case that no work was downloaded on the last request.

```

1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <judgeConfiguration>
3   <cmlURI>http://localhost:8080/cml</cmlURI>
4   <evaluationRequestsLocation>/evaluation-requests</
evaluationRequestsLocation>
5   <isolateCommand>/usr/bin/isolate</isolateCommand>
6   <isolateLimits>
7     <memLimit>3000000</memLimit>
8     <timeLimit>4</timeLimit>
9     <timeWallLimit>8</timeWallLimit>
10    <extraTime>2</extraTime>
11    <discQuota>64</discQuota>
12    <processesCount>32</processesCount>
13  </isolateLimits>
14  <isolateSettings>--dir=/etc -e</isolateSettings>
15  <mainOrchestraorBeanID>mainOrchestrator</mainOrchestraorBeanID
>
16  <waitingTime>5</waitingTime>
17 </judgeConfiguration>

```

Listing D.1: Example configuration file of the Judge component

As stated in the 2.4, the actual evaluation algorithm is dependent on the problem itself and the component is dynamically configured by the related Spring configuration file. Details of this configuration can be found in D.2.2.3.

### D.1.3 Scoreboard configuration

Configuration of the Scoreboard component is done by single XML configuration file, which can be passed either by the path to file on local file system or as an URL in remote repository at the start of the program.

The configuration file currently supports two parameters, the example configuration file can be found on listing D.2.

**cmlURL** This contains url of the cml component. the Scoreboard component will quit, if the connection fails.

**waitingTime** Represents number of seconds the Scoreboard component waits before sending next work request to the CML component in case that no work was downloaded on the last request.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <scoreboardConfig>
3   <cmlURL>http://localhost:8080/cml</cmlURL>
4   <waitingTime>10</waitingTime>
5 </scoreboardConfig>
```

Listing D.2: Example configuration file of the Scoreboard component.

### D.1.4 WebUI configuration

The configuration of this component is done via the standard „faces-config.xml” file, which is located in the „WEB-INF” directory of the web application.

The configuration itself is done through the application scoped bean named *applicationSettings*. The example configuration file is on listing D.3, the currently supported configuration parameters are:

**cmlURI** This contains URL of the CML component. the WebUI component will crash, if the connection fails.

**adminMail** Mail of the web application administrator (shown on the error page).

**maxSubmissionFileSize** Maximum allowed size of the submission file in kilobytes. The upload will be refused, if the file size will exceed this threshold.



```

1 <?xml version='1.0' encoding='UTF-8'?>
2 <faces-config version="2.2"
3     xmlns="http://xmlns.jcp.org/xml/ns/javaee"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
6     http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd">
7 <managed-bean>
8   <managed-bean-name>applicationSettings</managed-bean-name>
9   <managed-bean-class>
10     cz.pcss.webui.model.ApplicationSettings
11   </managed-bean-class>
12   <managed-bean-scope>application</managed-bean-scope>
13   <managed-property>
14     <property-name>cmlURI</property-name>
15     <value>http://localhost:8080/cml</value>
16   </managed-property>
17   <managed-property>
18     <property-name>adminMail</property-name>
19     <value>admin@pcss4.cz</value>
20   </managed-property>
21   <managed-property>
22     <property-name>maxSubmissionFileSize</property-name>
23     <value>1024</value>
24   </managed-property>
25 </managed-bean>
</faces-config>

```

Listing D.3: Example configuration file of the WebUI component

## D.2 System configuration

This section describes configuration parameters and format of system data (e.g. contest, problem...).

The subsection D.2.1 covers description of configuration files and data entities independent on the contest or problem definition (e.g. users). Configuration of problem definitions and process of creating new ones is described in D.2.2. The subsection D.2.3 describes configuration of contest related data and process of adding new contest to the system.

### D.2.1 System data configuration files

This section describes the data entities used by system that are independent or shared among contests. Before setting the contest and the problem definitions itself, following data entities should be already present in the system:

- User data entities (D.2.1.1)
- Supported programming languages configurations (D.2.1.2)

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <user>
3   <email>hauktoma@fit.cvut.cz</email>
4   <login>hauktoma</login>
5   <name>Tomas Hauk</name>
6   <password>password</password>
7 </user>
```

Listing D.4: Example of XML representing user.

### D.2.1.1 Users

XML files that are representing users are stored as „shared/users/username” in the sample data. Fields of the XML are self-explanatory and can be found on listing D.4.

### D.2.1.2 Languages

The XML objects representing programming language settings are stored in the „shared/languages/language-name” file. Example XML for the C++ language is on listing D.5, for Java on D.6. The supported parameters are:

**LanguageName** Name of the language.

**FileNameExtension** Extension of the source code file.

**CompilerCommand** Path to compiler binary executable.

**CompilerArgs** Compiler arguments. The Judge can substitute correct values for „workingDirectory” (compiler directory) and „files-to-compile” (list of files to be compiled).

**sourceFileName** Name of the source file — submitted program. If not set, it wont be possible to submit java programs in text form.

**isolateCommand** Command for the isolate utility. This will be appended to the Isolate command after the „-run” switch (see [20] for details). Supports the „compiled-file” substitution.

**OutputFileName** The file name generated by the compiler. Must be set because of the Java compiler constraints.

## D.2.2 Problem configuration files

This subsection describes configuration files and configuration process of defining new problems. The problem definitions are independent on the contest and scoring approach and thus can be shared among the system contests.

To add new problem definition to the system, one must:

```

1 <Language>
2   <CompilerArgs>-Wall -o b.out ${files-to-compile}</CompilerArgs>
3   <CompilerCommand>/usr/bin/c++</CompilerCommand>
4   <FileNameExtension>cpp</FileNameExtension>
5   <isolateCommand>${compiled-file}</isolateCommand>
6   <LanguageName>CPlusPlus</LanguageName>
7   <sourceFileName>io.cpp</sourceFileName>
8 </Language>

```

Listing D.5: Example XML describing C++ programming language settings.

```

1 ?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 \begin{verbatim}
3 <Language>
4   <CompilerArgs>-d ${working-directory} ${files-to-compile}</
   CompilerArgs>
5   <CompilerCommand>/usr/bin/javac</CompilerCommand>
6   <FileNameExtension>java</FileNameExtension>
7   <isolateCommand>/usr/bin/java -- -cp ${compiled-file-dir}
   CorrectIO</isolateCommand>
8   <LanguageName>Java</LanguageName>
9   <OutputFileName>CorrectIO.class</OutputFileName>
10  <RunnerCommand>/usr/bin/java</RunnerCommand>
11  <sourceFileName>CorrectIO.java</sourceFileName>
12 </Language>
13 \end{verbatim}

```

Listing D.6: Example XML describing Java programming language settings.

- create data objects representing the validation data (see D.2.2.1).
- create data object containing basic problem data (see D.2.2.2)
- create the Spring configuration file representing the processing tree for the Judge component (see D.2.2.3).

Prerequisite of adding a new problem is having configured at least one programming language definition (see D.2.1.2).

### D.2.2.1 Validation data

The validation data are represented by pair of two files named *input* and *output* (these names are mandatory). The *input* file will be given to the submitted program upon execution on standard input. The *output* file contains any data that can be used by the particular validator that will perform validation of the output generated by the program. Each pair has its own dedicated directory that is by default located in the „shared/validation-data” directory The

dedicated directory can have arbitrary name and must contain both correctly named files.

### D.2.2.2 Problem definition

This XML object represent the top level problem definition. These data objects are by default stored in the „shared/problems/” directory. The actual file might have an arbitrary name.

The problem definition currently supports following parameters (example file can be found on listing D.7):

**problemName** Name of the problem as will be shown to the user.

**problemShortDescription** Short description (summary or intro) of the problem.

**problemDescription** The description of the problem and task assignment details.

**processingChainDataLocation** Relative CML path of the Spring configuration file, which represents the processing algorithm for the Judge component (see D.2.2.3).

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <problemData>
3   <problemDescription>description</problemDescription>
4   <problemName>Sample problem</problemName>
5   <processingChainDataLocation>/shared/judge-component-settings/
6   default-process-tree</processingChainDataLocation>
7   <shortDescription>short description</shortDescription>
8 </problemData>
```

Listing D.7: Example XML representing the problem definition.

### D.2.2.3 Judge processing tree

The actual algorithm is described in form of the Spring configuration file. This file contains the actual structure of the algorithm whose components are formed by set of *units* and *orchestrators* (described in 2.4.4).

This file is by default located in the „shared/judge-components-settings” directory and can have an arbitrary name. This file can be theoretically shared between different problem definitions.

The example file can be found on the enclosed CD. This example contains tree which contains one compiler, 3 instances of binary validator and one evaluation unit.

The top level bean, which will serve as the root of the processing tree must be named as defined in the *mainOrchestratorParam* of the Judge configuration file (see D.1.2).

### D.2.3 Contest configuration files

This section describes the format and supported configuration options of the contest related data along with the process of creating new contest.

To add new contest, one must:

- create data object representing basic contest info (see D.2.3.1).
- create list of users of the contest (see D.2.3.2).
- create data object representing problem instance (which is linked to the problem definition created in D.2.2.2). Problem instance object is described in D.2.3.3.

Before adding new contest, the system should have access to data objects representing at least one user (see D.2.1.1) and at least one problem definition (see D.2.2.2).

#### D.2.3.1 Contest info

This XML file describes the basic contest settings and is located in the root contest directory (e.g. „contests/contest-name/contest-info” in case of default data). The file contains following settings (example file at listing D.8):

**allowedLanguages** Relative paths to the data entities that represent allowed languages for this contest.

**ContestName** Name of the contest.

**contestShortName** Short name of the contest.

**ContestStart** Date and time of the contest start. The contest will be accessible only to user responsible for contest (see below) before this time and date.

**ContestEnd** Date and time of contest end. The user interface will refuse to accept submissions of users after this date and time (except for the responsible user, see below).

**ShortDescription** Short contest description.

**LongDescription** Long contest description.

**responsibleUser** Relative CML path of the XML object, which represents user. This user has many privileges in this contest, such as ability to submit always (regardless of contest or task time constraints), can see all submissions in the contest of all users and his submissions are not displayed in the scoreboard for regular users.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <ContestInfo>
3   <allowedLanguages>/shared/languages/cplusplus /shared/
   languages/java</allowedLanguages>
4   <ContestName>Effective programming 1, ZS 2000/3000</ContestName
   >
5   <contestShortName>BI-EP1 ZS 2000/3000</contestShortName>
6   <ContestEnd>2030-02-18T06:10</ContestEnd>
7   <longDescription>Long description here</longDescription>
8   <responsibleUser>/shared/users/hauktoma</responsibleUser>
9   <shortDescription>Short description here</shortDescription>
10  <ContestStart>2000-03-20T10:10</ContestStart>
11 </ContestInfo>
```

Listing D.8: Example contest info XML object

### D.2.3.2 Contest users

Is XML versioned object (located in „contests/contest-name/users/contest-users.1” by default). This object contains list of relative CML paths of the XML objects that are representing users participating in contest in role of the „team”. Example file is on listing D.9.

```
1 o?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <usersInContest>
3   <users>/shared/users/anrbard</users>
4   <users>/shared/users/hauktoma</users>
5   <users>/shared/users/mmuunas</users>
6   <users>/shared/users/vlastsvc</users>
7 </usersInContest>
```

Listing D.9: List of contest users (teams)

### D.2.3.3 Problem instance

This XML object represents the instance of the problem that is linked to the object with actual problem definition (see D.2.2.2). This object is by default stored in „contests/contest-name/problem-instances/problem-name/problem-instance”. The example XML can be found on listing D.10, the supported parameters are:

**ScoreFactoryImplementation** This parameter represents implementation of some class implementing the *IScoreFactory* interface that is responsible for handling score (creating, interpreting). See D.3.1 for details.

**start** Time and date of the problem start. Access to the problem definition and submission before this time and date will be denied to regular user.

**deadline** Time and date of the problem end. Submissions from regular users will not be accepted after this limit.

**maxSubmissions** Maximum submission count that is regular user allowed to submit (privileged user can submit always).

**problemDataID** Relative path to the XML object representing the actual problem definition and that contains necessary data for e.g. evaluation process (see D.2.2.2 for details).

**problemState** Represents state of the problem. Currently supported values are:

**OPENED** Standard state, no restrictions.

**CLOSED** No matter how deadline is set, the system does not accept new submissions for this problem.

**TEMPORARY\_CLOSED** Problem does not accept submissions, but only temporary (e.g. technical issues). This option is technically the same as the previous one but with different semantic meaning.

**ACCESS\_FORBIDDEN** Problem does not accept submissions and does not allow to see the problem details. (But it is possible to see, that the problem instance exists in some particular contest.)

**showCompilationLog** Boolean value. True means, that regular user will see a compilation log.

```

1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 <problemInstance>
3   <percentualScoreFactory>
4     <threshold>80</threshold>
5   </percentualScoreFactory>
6   <deadline>2017-03-10T10:12</deadline>
7   <maxSubmissions>5</maxSubmissions>
8   <problemDataID>/shared/problems/bubbles</problemDataID>
9   <problemState>OPENED</problemState>
10  <showCompilationLog>true</showCompilationLog>
11  <start>2015-03-20T10:12</start>
12 </problemInstance>

```

Listing D.10: Example XML representing problem instance in the contest.

## D.3 Extending system functionality

The following subsections are describing process of adding new functionality to the system.

### D.3.1 Adding new scoring approach

Implementation of the scoring algorithm can be found in the module *CM-LLData*, in the package „cz.pcss.cmldata.score” (and „impl” subpackage). Implementing new Scoring factory will mean implementing the abstract class *AbstractScoreFactory* (most importantly method creating the universal data structure and method used for interpreting it). The implementation details and requirements should be apparent from the implementation of the factories already implemented.

After implementation of the new scoring factory, there is need to add reference of new class to the *ProblemInstance* class as an JAXB XML annotation (*@XmlElementRef*). The class is located in the „cz.pcss.cmldata.icpc.data” package.

If the above process was successful than the system will be able to use the new Scoring factory by adding it the *Problem instance* data structure (see D.2.3.3). No other intervention should be needed and the implementation should correctly handle submissions evaluation as well as presenting correct scoreboard to the user in the web interface.

### D.3.2 Adding new validator

The implementation currently supports only exact binary validation. The actual validation logic is currently implemented such that there is top-level validation component implementing the *IValidator* interface (currently only class *InternalValidator*) that is responsible for executing the program in secure way and generating results. The actual validation method implementation is separated from this class (strategy pattern) and is represented by the class implementing the *IValidationMethod* interface.

To implement new validation method, there is need to:

- implement the *IValidationMethod* interface and its only method *validate*, and
- create new Spring configuration file utilizing the new validation method (see D.2.2.3), i.e. inject this new class to the *InternalValidator* instance.

### D.3.3 Judge subcomponents

The architecture of Judge subcomponents was described in 2.4.4.

Adding new *Orchestrator* (component responsible for data flow in the processing tree) means implementing the abstract class *AbstractOrchestrator*. Implemented class can be than used in the Spring configuration file (see D.2.2.3) and can be used either as the root or inner node of the processing tree.

Adding new *Unit* (component, which is responsible for data preprocessing for the actual algorithm implementation) means implementing the *AbstractOrchestratedIOUnit* interface. The new implemented unit can be used in the



Spring configuration file (see D.2.2.3) and can be used as a leaf in the actual processing tree (i.e. will be assigned to some Orchestrator).

As stated in 2.4.4, the classes implementing the actual processing algorithm are totally independent on the system itself. Implementing new processing algorithm (e.g. static source code analysis, plagiarism checker...) will mean implementing the new algorithm and appropriate unit, which will handle preprocessing of data needed for the particular task. The new unit will be than utilized in the Spring configuration file of the processing tree (see D.2.2.3) and through it, the new algorithm will be injected into the unit.



# Web interface screenshots

The screenshot displays the web interface for the contest "Effective programming 1, ZS 2000/3000". The interface is divided into several sections:

- Left Sidebar:** Contains navigation options such as "Main contest page", "List of problems", "Sample problem", "Squares and Circles", "Bubbles", "Scoreboard", "My submissions", and "All contest submissions".
- Main Content Area:**
  - Header:** "Effective programming 1, ZS 2000/3000 BI-EP1 ZS 2000/3000".
  - Introduction:** A paragraph explaining the contest's focus on algorithmic and data structure problems.
  - Harmonogram:** A table listing the schedule of events, including lectures, exercises, and contests.
  - Hodnocení:** A table showing the scoring system for various problems, including the number of problems, points per problem, and maximum possible points.
- Right Sidebar:**
  - Contest links:** A list of links to sample problems, scoreboard, and other resources.
  - Timer:** A countdown timer showing "5151 days, 16 hours, 0 minutes remain to end".
  - Contest info:** A section providing details about the contest, including the name, short name, start and end dates, and the person responsible.

Figure E.1: Contest detail with list of problems.

## E. WEB INTERFACE SCREENSHOTS

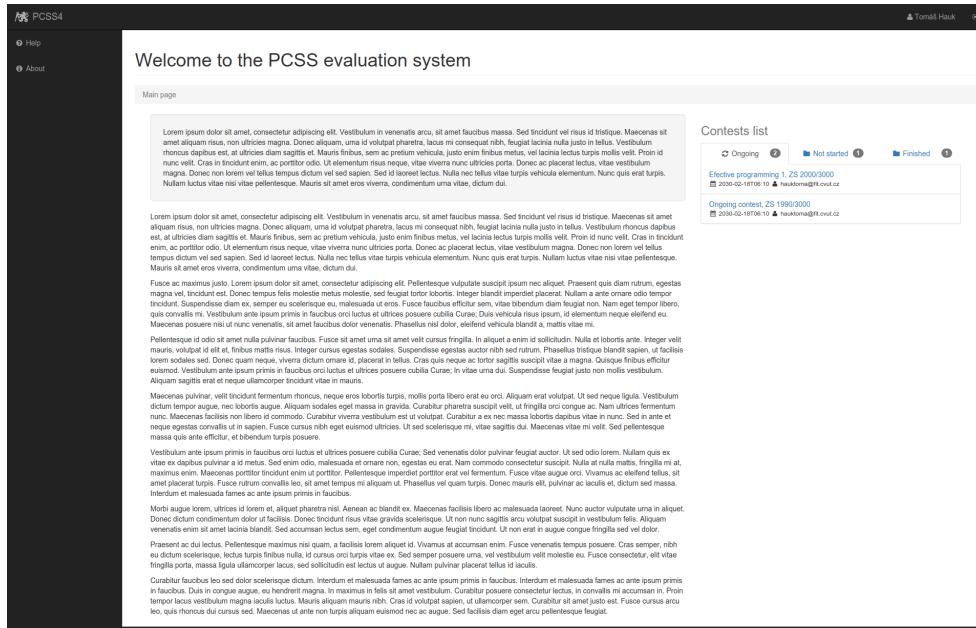


Figure E.2: Web interface main page with contest list.

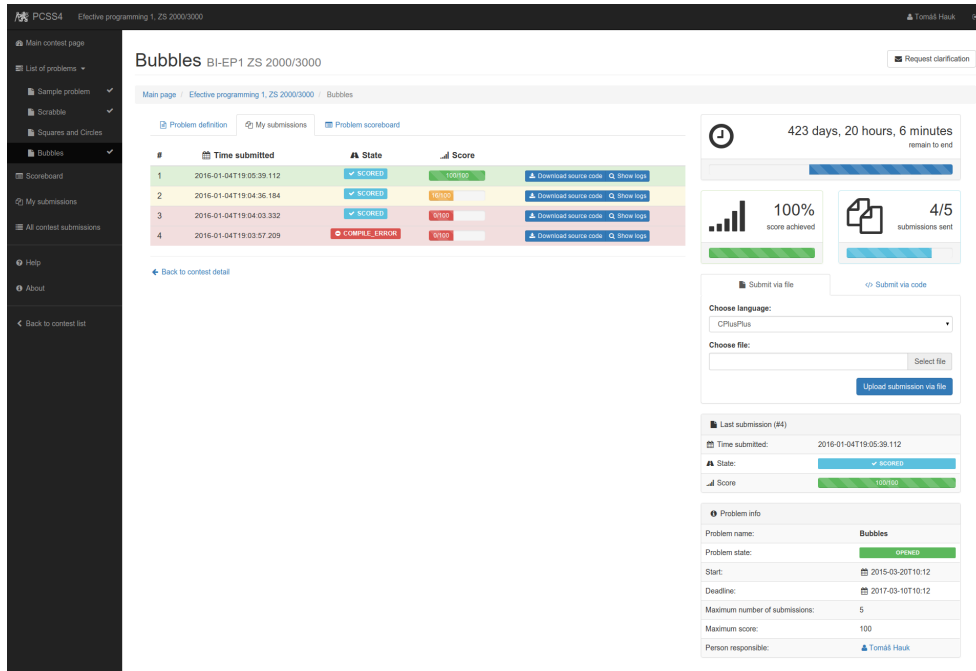


Figure E.3: Problem submissions and problem details and stats.

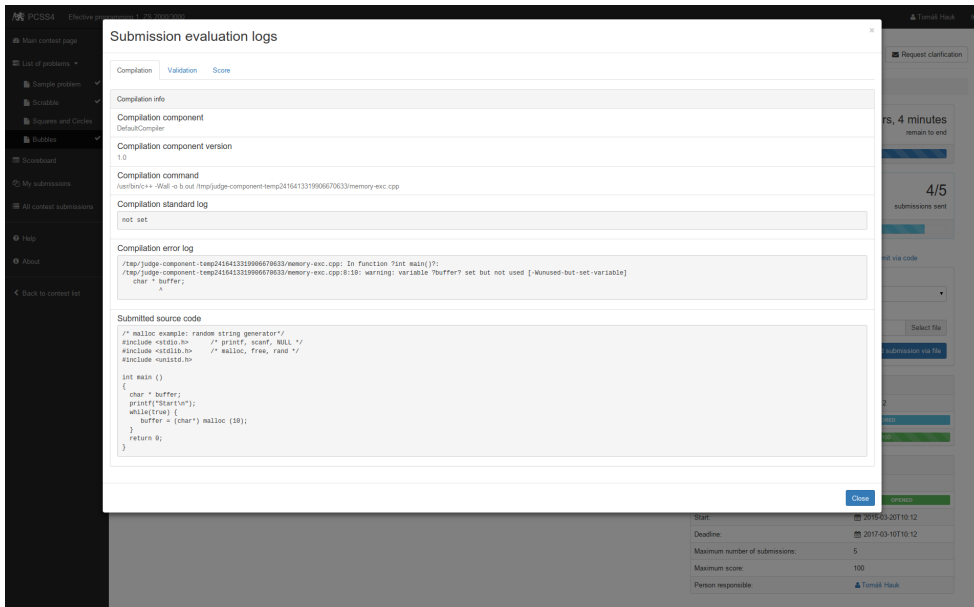


Figure E.4: Logs of the compilation process.

The screenshot shows a 'Scoreboard' for the contest 'Elective programming 1, ZS 2000/3000'. The scoreboard table is as follows:

#	User	Bubbles		Sample problem		Squares and Circles		Scrabble		TOTAL %
		Submissions	Result	Submissions	Result	Submissions	Result	Submissions	Result	
1.	[Tomáš Hauk]	1/5	16%	30/5	100%	1/5	100%	4/5	100%	79.00%
2.	Vlastimil Švorc	0/5	0%	0/5	0%	0/5	0%	0/5	0%	0.00%
3.	Michael Unnamed	0/5	0%	0/5	0%	0/5	0%	0/5	0%	0.00%
4.	Andrew Barak	0/5	0%	0/5	0%	0/5	0%	0/5	0%	0.00%

At the bottom right, it says 'Last scoreboard update on 2016-01-04 19:05:40.663'.

Figure E.5: Contest scoreboard.