

# Practical Guideline for Automated Programming Assignments Writing

José Daniel González-Domínguez  
Zenón José Hernández-Figueroa  
Juan Carlos Rodríguez-del-Pino  
Ján Skalka

[www.fitped.eu](http://www.fitped.eu)

2021

# Practical Guideline for Automated Programming Assignments Writing

## **Published on**

November 2021

## **Authors**

José Daniel González-Domínguez | University of Las Palmas de Gran Canaria, Spain

Zenón José Hernández-Figueroa | University of Las Palmas de Gran Canaria, Spain

Juan Carlos Rodríguez-del-Pino | University of Las Palmas de Gran Canaria, Spain

Ján Skalka | Constantine the Philosopher University in Nitra, Slovakia

## **Reviewers**

Anna Stolińska | Pedagogical University of Cracow, Poland

Dušan Junas | Teacher.sk, Slovakia

Cyril Klimeš | Mendel University in Brno, Czech Republic

Piet Kommers | Helix5, Netherland

Eugenia Smyrnova-Trybulska | University of Silesia in Katowice, Poland

Michal Švec | Teacher.sk, Slovakia

## **Graphics**

Marcela Skalková | Teacher.sk, Slovakia

David Sabol | Constantine the Philosopher University in Nitra, Slovakia

Erasmus+ FITPED

Work-Based Learning in Future IT Professionals Education

Project 2018-1-SK01-KA203-046382

Co-funded by the  
Erasmus+ Programme  
of the European Union



The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



Licence (licence type: Attribution-Non-commercial-No Derivative Works) and may be used by third parties as long as licensing conditions are observed. Any materials published under the terms of a CC Licence are clearly identified as such.

All trademarks and brand names mentioned in this publication and all trademarks and brand names mentioned that may be the intellectual property of third parties are unconditionally subject to the provisions contained within the relevant law governing trademarks and other related signs. The mere mention of a trademark or brand name does not imply that such a trademark or brand name is not protected by the rights of third parties.

© 2021 Constantine the Philosopher University in Nitra

**ISBN 978-80-558-1795-8**

## Table of Contents

<b>INTRODUCTION .....</b>	<b>8</b>
<b>SOFTWARE TESTING.....</b>	<b>10</b>
INTRODUCTION TO SOFTWARE TESTING.....	10
Advantages of software testing.....	12
Manual and automated testing.....	13
Fundamental types of testing.....	13
METHODS AND LEVELS.....	14
Testing methods.....	14
Testing levels.....	16
Advanced testing - non-functional types of testing.....	17
<b>AUTOMATIC EVALUATION OF SOURCE CODES.....</b>	<b>19</b>
<b>VIRTUAL PROGRAMMING LAB .....</b>	<b>22</b>
TYPICAL USERS.....	22
BASIC FEATURES .....	23
PROGRAM EXECUTION .....	23
VPL ACTIVITY IN MOODLE .....	24
Creating VPL activity.....	24
Test cases .....	28
Execution options.....	29
Requested files .....	30
Submissions report.....	31
Similarity.....	34
Test activity .....	37
Submission view .....	39
Assignment list in Virtual Programming Labs.....	42
VPL TEST CASE LANGUAGE.....	44
Basic definition .....	44
Input.....	44
Output .....	45
Multiple output checking .....	49
Penalizations and final grade .....	49
Advanced testing.....	50
Running another program .....	51
OUTPUT FILTERING AND FORMATTING .....	54
Filtering the raw output .....	54
Format indicators .....	56
<b>PRISCILLA SYSTEM .....</b>	<b>57</b>
CONTENT STRUCTURE .....	58
Content microlesson .....	60
Short answer .....	60
Choice of options.....	61
Multiple choice of options.....	62
Placing code snippets .....	63

Writing commands into code .....	64
Rearranging lines (of source code) .....	65
PROGRAM ASSIGNMENT .....	66
BIOTES assignments .....	66
Statically evaluated code.....	68
<b>ADVANCED VPL FEATURES .....</b>	<b>70</b>
EXECUTION FILES.....	70
EXECUTION RESOURCES LIMITS .....	71
FILES TO KEEP WHEN RUNNING .....	72
VARIATIONS .....	72
CHECK EXECUTION SERVERS .....	73
LOCAL EXECUTION SERVERS.....	74
THE BASED-ON FEATURE.....	74
ADDING SUPPORT FOR A NEW PROGRAMMING LANGUAGE .....	75
<b>CUSTOMIZING AUTOMATIC PROGRAM ASSESSMENT .....</b>	<b>76</b>
EVALUATION OUTPUT FORMAT .....	76
FORMATTING THE COMMENTS (FEEDBACK) .....	76
DETAILS OF RUNNING A TASK .....	76
<b>VPL ARCHITECTURE.....</b>	<b>78</b>
CONNECTIONS ACCEPTED BY EXECUTION SERVER .....	79
SECURITY ASPECTS.....	80
The "available" method.....	81
The "request" method.....	83
The "getresult" method.....	86
The "running" method.....	87
The "stop" method.....	87
TASK MONITORING.....	88
RETRIEVING THE VPL ACTIVITIES DEFINITION TO USE IT ON ANOTHER SYSTEM .....	89
<b>VPL SETTINGS IN PRISCILLA .....</b>	<b>91</b>
COURSE SETTINGS .....	91
JAVA .....	91
C/C++ LANGUAGE .....	95
PYTHON .....	96
PHP .....	98
<b>UNITTEST2VPL FRAMEWORK .....</b>	<b>102</b>
REQUIREMENTS .....	102
Produce suitable VPL feedbacks.....	102
Catch unexpected exceptions .....	102
Prevent from non-ending states .....	103
SOLUTION FOR PYTHON .....	103
Unittest2VPL Base Activity .....	103
Scripts .....	108

Internationalization and Localization .....	109
How to use .....	110
VPL activity configuration.....	112
PRISCILLA configuration .....	112
<b>SQLITETEST4VPL FRAMEWORK .....</b>	<b>114</b>
THE IMPLEMENTATION FILES .....	115
Scripts .....	115
Data preparation files.....	119
<b>JUNIT4VPL FRAMEWORK .....</b>	<b>122</b>
WHAT OFFERS JUNIT4VPL? .....	122
USING JUNIT4VPL.....	122
Basic use of JUnit4VPL, the OddEven problem .....	123
Testing a students class.....	125
ADVANCED TESTING CUSTOMIZATION .....	125
The Test annotation .....	126
The TestClass annotation .....	126
The ConsoleCapture class.....	126
JUNIT4VPL INTERNATIONALIZATION .....	127
<b>JUNITBASE FRAMEWORK .....</b>	<b>128</b>
EXAMPLE OF ASSIGNMENT .....	128
MYSOLUTION.....	129
STARTING CLASS .....	129
EVALUATE CLASS.....	131
SCRIPTS.....	133
<b>PRACTICAL INFO – xUNIT TESTING.....</b>	<b>134</b>
JUNIT.....	134
Testing in IntelliJ .....	135
Testing in Eclipse .....	140
Finish the test .....	144
ACCURACY IN TESTS.....	145
Real numbers in test.....	145
EXCEPTIONS .....	146
Division by zero .....	146
TESTING METHODS.....	148
<b>BIBLIOGRAPHY.....</b>	<b>150</b>



## Introduction

The ability to prepare algorithms for solving problems and rewrite them into program code is one of the necessary skills in finding work not only in the IT sector. Programming language courses are still the most challenging courses that students fail. The current approach to solving this problem is based on adapting the educational methodology to the habits of current students. Automated assessment represents a tool that automatically checks source code and provides feedback at a level defined by the instructor or module providing these operations.

The publication you are holding in your hands summarizes the authors' many years of experience in the field of automatic evaluation of source codes. Many software environments and modules have been created to assess the correctness of source code written in various programming languages. Some are narrowly oriented. Others can cover needs in many programming languages.

One of the successful solutions is the Virtual Programming Lab for Moodle, which currently works on more than 1,700 servers across educational organizations worldwide.

Based on this solution, modules for the PRISCILLA system were created within the FITPED project. PRISCILLA modules use this system but leave the LMS Moodle environment and offer a single-purpose and simple graphically oriented environment instead of Moodle complexity.

The publication is one of the results of the project Work-Based Learning in Future IT Professionals Education (ERASMUS+ Programme 2018, KA2, project number: 2018-1-SK01-KA203-046382).

The primary goal of the publication is the dissemination of project results in the field of automatically evaluated tasks development and use, which represent a significant contribution both in fulfilling the project objectives and in the didactics of learning programming.

The introduction of the publication presents basic information about the reasons and methods of testing followed by a general introduction to automatic evaluation of source code.

The following chapters are more practical. The third chapter presents VPL, its use in the LMS Moodle environment, explains the basic principles of creating assignments and familiarizes the reader with various types of programming assignments that can be made through the language for creating test cases. In the next chapter, the authors present the implementation of elements in the PRISCILLA environment and describe the types of questions and assignments that can be used to build programming skills in a newly developed system.

The following three chapters describe the structure and nuances of more complicated tasks that allow you to verify the correctness of complex tasks and enter the low-level evaluation process. At the same time, this group of chapters provides information suitable for users who decide to implement VPL in their software solution.

The VPL settings in the PRISCILLA section summarize the scripts needed to check the source code in Java, C, Python, and PHP.

The next chapters present advanced options for code verification and user message generation through frameworks from colleagues from the University of Las Palmas Grand Canaria and Constantine the Philosopher University in Nitra. They are dedicated to verifying the correctness of tasks with objects in Python and Java languages. A unique approach provides a framework for checking SQL entries, which allows you to check the correctness of queries sent to the database server.

The publication concludes with a chapter explaining the working principle of xUnit libraries designed for unit testing of programs. This chapter is intended for readers who have not yet encountered this type of testing and cannot absorb the content of the chapters presented frameworks without this knowledge.

Based on the content of the publication, it is possible to create different types of tasks not only within the platform developed for the needs of the project but also in other platforms supporting automated code evaluation.

The benefit of the publication is also a description of best practices for different programming languages because each of the languages has its own specifics that allow it to communicate with the educator more friendly and more accurately identify deficiencies in the code than when using universal approaches.

# Software Testing

## Introduction to software testing

Software testing is part of a more general verification and validation process, including static and dynamic validation techniques. Understanding these techniques is the main aim of this chapter.

### Basic testing methods

We can understand the concept of software testing as:

- the process of evaluating a system or its component(s) with the intent to find whether it satisfies the specified requirements or not,
- executing a system to identify any gaps, errors, or missing needs contrary to the actual requirements,
- intended to show that a program does what it is designed to do and discover program defects before it is used.
- a process of analysing a software item to detect the differences between existing and required conditions and evaluate the features.

Testing software often requires executing a program using artificial data. Testing can reveal the presence of errors, not their absence.

### Life cycle

Testing is a part of all kinds of software development life cycles. Different types of stakeholders are involved in the software testing process depending on the complexity of the project, used methodology and project management, the experience of the project team members. The following roles are often involved in the process:

- Software Tester
- Software Developer
- Project Manager
- End-User
- Software Quality Assurance Engineer
- Quality Assurance Analyst.

Software Development Life Cycle (SDLC), often called the Software Development Process, is a process used by the software industry to design, develop and test high-quality software. The SDLC aims to produce high-quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates. It has the following phases in general:

- Planning and requirements analysis
- Defining requirements
- Designing the product architecture
- Building the product
- Testing the product

- Deployment of the product
- Maintenance and further development of the product

There are numerous SDLC models, which are often called Software Development Process Models. They are suitable for different situations. Their effective use depends on the complexity of the software product, which should be developed. They differ in the series of steps, which ensure success in the process of software development. The following are the most popular SDLC models:

- Waterfall model
- Iterative model
- Spiral model
- V-model
- Agile model
- Rapid Application Development Model
- Prototyping model.

The software testing is included in all SDLC models to different extents. It can create a separate phase, or testing can create an inseparable part of various phases of the SDLC.

### **Verification and Validation**

Verification and validation are two very similar terms, which closely relate to the software testing topic. They differ in the following aspects:

Verification:

1. "Are you building it right?"
2. Ensures that the software system meets all the functionality.
3. Verification takes place first and includes checking for documentation, code, etc.
4. Developers do it.
5. It has static activities, as it includes collecting reviews, walkthroughs, and inspections to verify software.
6. It is an objective process, and no subjective decision should be needed to verify software.

Validation:

1. Validation addresses the concern: "Are you building the right thing?"
2. Ensures that the functionalities meet the intended behaviour.
3. Validation occurs after verification and mainly involves the checking of the overall product.
4. Testers do it.
5. It has dynamic activities, as it includes executing the software against the requirements.
6. It is a subjective process and involves personal decisions on how well the software works.

### **When to start testing?**

Testing should start as early as possible because an early start to testing reduces the cost and time to rework the product. The real start depends on the used development model. Testing is incorporated in every phase of SDLC:

- During the requirement gathering phase, the analysis and verification of requirements are also considered as testing.
- Reviewing the design in the design phase to improve the design is also considered testing.
- Testing performed by a developer on completion of the code is also categorised as testing.

### **When to finish testing?**

Any software can not be 100% tested. Testing is a never-ending process. Therefore, it is crucial to estimate how much testing is enough and consider the following circumstances:

- Deadline of the project
- Code coverage level of the source code
- Bug rate under a certain level
- Decisions of the project manager

### **Advantages of software testing**

Advantages of software testing can be defined as:

- Cost-effectiveness – early testing saves time and costs because the discovered problem does not affect the final implemented solution
- Software improvement – testing is a phase of all SDLC
- Automation reduces the testing time. However, it should be started after static validation, an inspection of the system.
- Software quality assurance helps measure the following software properties: functionality, reliability, usability, efficiency, maintainability, and portability.

Program Testing has the following parts

Validation testing demonstrates to the developer and the system customer that the software meets its requirements. A successful test shows that the system operates as intended.

Defect testing, which discovers situations in which the behaviour of the software is incorrect, undesirable or does not conform to its specification. It leads to defect testing. A successful test is a test that forces the system to perform incorrectly and so shows a defect in the system

Software inspection is a formal evaluation technique in which software requirements, designs, or codes are examined in detail by a person or a group other than the author to detect faults, violations of development standards, and other problems complementary to verification technique to testing. It represents a formal technique that involves formal or informal technical reviews of any artefact by identifying any error or gap.

Software inspection is considered a practical approach for discovering program errors. It is a static verification because it focuses on analysing the static system representation to discover problems. Simultaneously, it does not require the execution of a system and, therefore, it does not require additional costs for inspection of incomplete versions of a system. It can be applied to any software representation like the requirements specification, software architecture, database schema.

Software inspection can consider a broader set of quality attributes like portability, maintainability, compliance with standards. It can check conformance with a specification, not with the customers' requirements. On the other hand, it does not check non-functional requirements like performance, usability.

## Manual and automated testing

Software engineering defines the following two types of testing:

- Manual testing covers testing software manually, without any automated tool and scripts. Manual testing has several stages, which will be introduced later. The tester, in the role of end-user, tests the software to identify any unexpected behaviour. The tester uses test cases and test scenarios to ensure the completeness of tests.
- Automated testing (Test automation) requires the tester to write scripts and use specialised software to test the product. Test automation is a logical replacement for manual testing, in which constantly repeating routines occur. It allows running the test scenarios repeatedly and incrementally.

When to Automate? Test Automation is helpful in the following situations:

- project is large and complex,
- projects require testing the same areas repeatedly,
- requirements do not change very often.

There are many specialised systems, which allow automated testing. The following methodology can be used to decide if automated testing can be used:

- Identifying areas within the software for automation
- Selection of appropriate tool for test automation
- Writing test scripts
- Development of test suits
- Execution of scripts
- Create result reports
- Identify any potential bug or performance issues

## Fundamental types of testing

There are several types of testing:

- Smoke Testing
- Functional Testing
- Non-functional Testing

They all can be used on different levels of testing, which will be introduced later.

## Smoke Testing

Smoke testing, often called build verification testing, is a type of software testing that comprises a non-exhaustive set of tests, which try to ensure that the most critical functions of the software will work. The result of this testing is used to decide if a build is stable enough to proceed with further testing. This type of testing can uncover problems early. It can be used in integration, system and acceptance levels of testing.

## Functional Testing

Functional testing is a type of black-box testing. The software is tested using a set of tests with known inputs. The obtained outputs (results) are compared with expected ones. Functional testing has the following steps:

- The determination of the functionality that the intended application is meant to perform.
- The creation of test data based on the specifications of the application.
- The output based on the test data and the specifications of the application.
- The writing of test scenarios and the execution of test cases.
- The comparison of actual and expected results based on the executed test cases.

## Non-functional Testing

Non-functional testing involves testing important software non-functional requirements such as

- performance,
- security,
- user interface,
- compliance with standards.

## Methods and levels

The following basic methods can be used based on the level of knowledge of the internal structure of the software, which is tested.

- Black-box testing
- White-box testing
- Grey-box testing
- Agile testing
- Ad-hoc testing

## Testing methods

### Black-box Testing

Black box testing, known as behavioural testing, is testing without knowing the internal structure, design or implementation. The tester has no access to the source code, but he interacts with the user interface of the software product. She provides a set of inputs and examines the outputs.

The outputs must fulfil the tester's expectations. The main advantage of this approach is that black-box testing is suitable for large code segments. It does not require access to source code, shows how the software will be used by end-user, does not need testers with the knowledge of programming languages, operation systems and other whole SDLC.

The main disadvantages of the black-testing technique are limited coverage by tests, difficulties in designing test cases, and limited knowledge of the testers about the software product.

This method attempts to find incorrect or missing functions, interface errors, errors in data structures, behaviour, and performance. It applies to the integration, system and acceptance testing levels.

### **White-box Testing**

This testing method, also known as glass testing, requires the tester knows the internal structure, design or implementation of the software. In other words, the tester has access to the source code and can investigate the internal logic and structure of the code. This situation is simultaneously the main advantage of this method.

Moreover, it allows code optimising, refactoring and the maximal coverage of the code due to the knowledge of the code. On the other hand, this technique requires a skilled tester and specialised tools like code analyser and debugging tools. This method is applicable to unit, integration and system testing levels.

### **Grey-box Testing**

The grey-box testing method has limited knowledge of the internal structure and logic of the tested software product. The tester usually must design documents and the database. Therefore, she can write better test scenarios.

The combination of best practices of white box and black box methods is considered the main advantage of this method. It relies on interface definition and functional specifications. The tests are realised from the user's point of view, not a developer. It is primarily used at the integration testing level.

### **Agile Testing**

Agile testing represents a testing method, which follows the principles of agile development methods. This testing method does not require any special approach and techniques. It still needs all proven software testing methods and levels, but their use depends on the agile team's tester or developer decisions and other priorities.

Agile testing is built upon very simple, strong and reasonable processes like the process of conducting the daily meeting or preparing the daily build. Simultaneously, it attempts to leverage tools, especially for test automation, as much as possible. Testing itself is in the middle of interest. As a result, this method does not elaborate on any plan or documentation.

## **Ad-hoc Testing**

Ad-hoc testing, sometimes called Random Testing or Monkey Testing, is a software testing method without planning and documentation. All tests are conducted informally and randomly without any formal procedure or expected results.

This method is typically used during Acceptance Testing. Surprisingly, this method can be very useful in finding errors, which is hard to find using other more systematic, step-by-step approaches. The success of the method, therefore, depends on the creativity and previous experience of the tester.

## **Testing levels**

### **Unit Testing**

Unit testing is a level of testing where individual units/components are tested to verify that these units behave as expected. Unit testing belongs to the white-box testing method. A unit is the smallest tested part of the software, such as the OOP class method.

Unit testing is performed mainly by the developer, who can also be the author of the source code or other members of the development team. The developer uses test data. The main aim is to validate that part of the source code is correct in terms of requirements and functionality.

It is impossible to cover all source code and evaluate all possible execution paths of the software.

Unit testing has the following benefits:

- increase confidence in changing code,
- code is easier to reuse,
- development is faster,
- the cost of fixing a bug is smaller,
- small units are easier to understand.

### **Integration Testing**

Integration testing means testing of combined parts of the software intending to determine if the parts work correctly. The purpose of this second level of testing is to expose faults in the interaction between integrated units. Integration testing can use bottom-up, top-down and big bang approach.

While the first one begins with unit testing, followed by a combination of module testing and builds. In the second one, the modules are tested first and then the lower-level modules and units are tested. A big bang is an approach in which all or most units are combined and tested in one run. Integration testing uses any of the black-box, white-box or grey-box testing methods.

### **System Testing**

System testing is the third level of software testing. It tests the whole system to verify if it meets functional and technical specifications. After all the components are integrated, the software is tested with the aim to fulfil the specified quality standards.

This kind of test enables to test, verification, and validate business requirements and the software architecture in the environment, which is very close to the production environment. The black-box testing method is usually used for system testing.

### **Acceptance Testing**

Acceptance testing is closely joined to quality assurance. Acceptance testing is the fourth and last level of software testing. It verifies whether the software meets previously defined specifications and satisfies the business requirements of the customer.

A system is tested for acceptability. Acceptance tests are intended to point out any bugs, which will result in the software crash, but it also points out small mistakes, errors and differences. It usually uses the black-box testing method. It does not follow a strict procedure and is rather ad-hoc.

### **Alpha Testing**

Alpha test is realised by the development teams in the first stage of testing. It means that combined unit, integration and system testing are considered together as alpha testing. The software is tested for spelling mistakes, broken links.

### **Beta Testing**

The beta testing (also called pre-release testing) follows alpha testing after it has been successfully finished. A selected group of future users tests the software.

It is important to distribute the testing to a wide range of future users, who will test installation procedure, typography, navigation and flow of tasks, etc. Simultaneously, they provide important feedback, identify hidden problems and test their fixes.

## **Advanced testing - non-functional types of testing**

### **Regression Testing**

All changes in the software can cause problems in other areas of the software. For that reason, regression testing focuses on verification if the change has not resulted in another functionality violation. In other words, regression testing ensures that this change has not caused problems that the tests do not cover.

During regression testing, new test cases are not created, but previously created test cases are re-executed. Regression testing can be used during any level of testing, mainly during system testing.

### **Usability Testing**

Usability testing is a black-box technique used to identify errors and consequent implementations of the software improvements by observing the users' behaviour. This testing is done from a user perspective to find out if the software is easy to use.

It is focused on the efficiency of use, ability to learn, ability to memorise, errors and safety and satisfaction of the users. This type of testing can be performed during system and acceptance testing levels.

**Security Testing**

Security testing belongs to the critical and inevitable kinds of non-functional testing. It aims to identify any problems with the security and vulnerability of the software. Depending on the nature of the software, security testing tries to ensure integrity, availability, correct authorisation and authentication, and save software against different kinds of attacks and flaws.

**Portability Testing**

Portability testing is focused on testing software for reusability, transferring software between computers and different versions of operating systems and middleware.

**Compliance Testing**

Compliance testing, sometimes called conformance testing or regulation testing is a type of testing to determine the compliance of a system with internal or external standards. The type of testing conducted during compliance testing depends on the specific regulation/standard being assessed.

**Performance Testing**

Performance testing is non-functional testing focused on determining how the software performs in terms of responsiveness and stability under different conditions. It covers load tests, stress tests, endurance and spike tests.

While Load testing tests the behaviour of the software by application maximum load of input data, stress testing tests the behaviour under abnormal conditions like losing resources. Performance testing tries to identify any bottlenecks related to the software performance like network delay, load balancing between servers, database transaction delay using quantitative and qualitative measures. It tests the speed, capacity, stability and scalability of the software.

## Automatic Evaluation of Source Codes

Based on the summary of areas of knowledge and skills domains, the following levels of students' ability to program were identified:

- Problem domain understanding (focused on problem description using natural language or metalanguage; the need to use commands of a programming language is not necessary):
  - understanding the definition of problems expressed in natural language,
  - understanding the goal of a solution expressed in natural language,
  - understanding the description of the solution in natural language,
  - problem transformation from natural language to a limited area of metalanguage (not in programming language yet),
  - solutions obtained from metalanguage transformation to the natural language;
  - discuss a problem, explain the solution procedures.
- Programming language domain understanding (focused on understanding syntax and semantic):
  - understanding commands of programming language and their use,
  - understanding programming language fundamentals (variable, input, output),
  - knowing parameters and syntax of fundamental commands,
  - understanding semantic of algorithmic structures (usually sequence, conditions, cycles),
  - knowledge to find suitable command needed to the realisation of necessary activity.
- Transformation problem to the programming language and data structure domain:
  - understanding the concept of data structures,
  - knowledge of main data structures implemented in language and their use,
  - mental transformation of the problem to programming language design,
  - ability to use programming structures with a combination of data structures to solve the defined assignments,
  - selection of suitable problem-solving strategies.
- Source code understanding domain:
  - the ability to read foreign code,
  - verification of correctness and identification of errors in source code,
  - understanding boundary and unsolvable inputs,
  - manual testing and repairing code using checkpoint and output partial results.
- Subroutines used to make a program more understandable and more effective:
  - understanding why and how to divide the program into smaller parts,
  - understanding subroutines, parameters and types of parameters,
  - to make the first contact with the effectiveness of programs (not only) using subroutines.
- Use of development tools:
  - to use pre-compiler and help in the development environment,
  - understanding why and how to debug programs,
  - the ability to build applications,
  - the ability to program with a focus on performance and memory.

- Deep understanding (join of programming and problem-solving) domain:
  - experience of problem-solving application (optimisation, specialisation, abstraction),
  - good-practice adoption (design patterns, code writing, documentation building).
- Software development domain:
  - ability to select powerful and appropriate technology,
  - skills in the management of the team,
  - development planning and team performance calculating,
  - problem-solving skills building and using.

The presented complexity of the programming learning process is the cause of educational failures, student frustration and lack of motivation, so it is necessary to offer students different ways to achieve the goal.

The automated assessment of programming assignments has been practised since programming has been taught, especially in the introductory programming courses. The automatic assessment has different features, which are automatically assessed by different assessment tools and systems.

Automated assessment represents a tool that allows checking source code automatically and brings a new perspective on learning. According to many authors, automated assessments are beneficial for the following areas:

- the student gains immediate feedback whether the program is correct, and students can use their own pace;
- the teacher gains extra time, instead of time wasted by checking the assignment and identifying and re-explaining repeated errors in past;
- it is possible to teach large groups of students without increasing the demands on teachers, which apply especially in the case of MOOC courses;
- the learning process is more efficient and, due to the errors tracking, speed and quality of the solutions, the individual parts of the process can be fragmented, quantified and described (complicated topics, problematic examples, number of necessary attempts, etc.).

Implementing the systems that allow evaluating the practical exercises and assignments of programming code writing is a great challenge for modern education. They should be based on good practices of live coding systems, e.g. CodeWars, CodeCombat, CodingBat, HackerRank, SPOJ, Project Euler, etc.

Modern principles of automated assessment are based on two main approaches:

- **Static evaluation** is based on checking the form, structure, content, or documentation of the source code. This type of evaluation is based on validating source code without executing the program and analysing the textual expression of code and anomalies in it. Static evaluation can be enriched with rules aimed at validating the values of parameters defined in tasks assignments. Static evaluation is the first option for design-oriented languages (e.g. HTML, CSS) or languages with simple rules (e.g. SQL).
- **Dynamic evaluation** approaches use output results for validation on various levels. For defined inputs, the principles use automated testing or comparison of expected and achieved results:

- **The I/O approach** is the simplest method from the content developer's point of view, with minimal requirements for its capabilities. The author of the content defined the input values and expected outputs. Validation is based on comparing the values obtained from the students' program with the values defined as expected and correct. Pairs are defined as test cases and usually contain values for standard inputs, boundary inputs and exceptions. The approach offers strong advantages: the definition of test cases is high-speed; the same test cases can be defined and used for many programming languages. The disadvantages are missing functions to verify the internal structure of the source code (can be solved by extending static evaluation methods) and formatting problems that cause a mismatch between expected and received output.
- **Writing automated tests** as part of engineering software testing is a widely used and required approach in the software development process. Using unit testing is one of the necessary user skills of modern programmers (JUnit, CUnit, etc.). This approach is currently the most effective way to validate code, which tests the outputs or results of the program and focuses on checking its elementary parts (units), as are methods, procedures, algorithms, class states, etc. The ideal goal of unit testing is to verify each part of the written code and allow immediate repetition of testing after modifying any part of the code. The advantage of this approach is greater flexibility and more accurate identification of errors, or/and the mistakes explanation to the user. The disadvantage can be considered as the more arduous preparation of the validation itself through writing code. In the presented framework, writing tests is one of the important activities aimed at developing programming skills in students and test writing into new program tasks is the desired activity.
- **Acceptance testing** represents the highest level of testing based on the definition and combination of testing scenarios focused on the outcomes of the program activity. Individual parts of the test are accepted if they positively verify the expected results from the user's or customer's point of view (e.g. changes in class instances, changes in the database, changes in the content of the website, etc.). This type of test is usually used for more complex assignments that teach technological skills, machine learning, results of collaborative activities, etc.

The result of automated testing should not only inform about the correctness of the program. Users need to view syntax errors, compiler messages, and test cases with differences between expected and obtained outputs. Help is also a valuable tool to help write the program, or in the event of a user jam, the correct authoring solution can also be obtained.

## Virtual Programming Lab

Virtual Programming Lab (VPL) is the easy way to manage programming assignments in Moodle. Its features of editing, running, and evaluating programs make the learning process for students and the evaluation task for teachers easier than ever. The software is licenced under GNU GPL3.

Its salient features are:

- Enables to edit the programs source code in the browser
- Students can run interactive programs in the browser
- You can run tests to review the programs.
- Allows searching for the similarity between files.
- Allows setting editing restrictions and avoiding external text pasting.

These features make easier the learning process for students and the evaluation task for teachers.

The plugin VPL for Moodle requires some VPL-Jail-System to delegate the running and evaluation of the student's code. VPL-Jail-System serves as a stateless, secure, and isolated sandbox for running code.

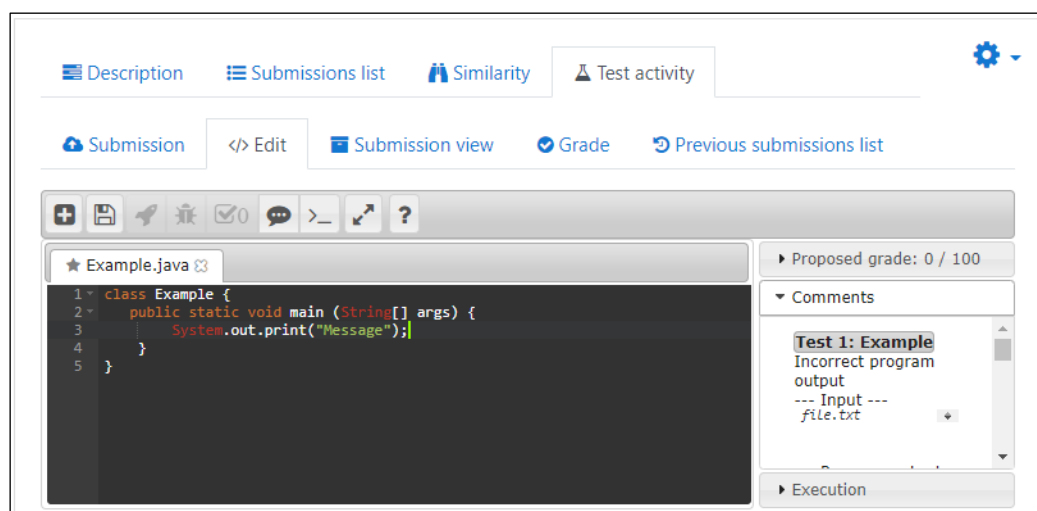
### Typical users

The primary users of VPL are teachers of universities and teachers of the last courses of high schools; currently (August 2021), at least 1700 servers around the world run VPL.

University teachers use VPL to take control of the programming tasks that students do, especially in courses with a large number of students.

Hight school teachers use VPL for introducing students to programming.

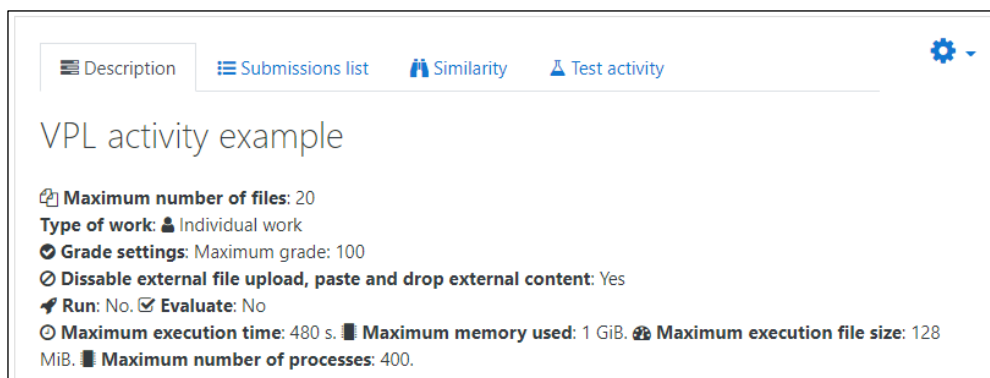
Students use VPL as an easy-to-use tool with zero installation and configuration, concentrating their efforts on the problem and not on the tools used.



## Basic features

If you don't know if VPL matches your needs, here is a list of some of the uses of VPL:

- Adequates for small to medium programming assignments tasks.
- Manages the student's submissions. Students and teachers can save time managing how, where, and when to submit and review programming assignments.
- Students can use VPL as a basic IDE online, allowing running their code with zero installation or configuration.
- A teacher can review, run, and evaluate students' submissions without a download code. Grading integrated with the Moodle grade book can be used.
- A teacher can set input/output tests easily to evaluate the student's code. The evaluation can lead to giving a final grade or advising the teacher in the grading process.
- The evaluation system is open to customization.
- Performing programming examines in controlled conditions with network access limits, password, and no external code introduction, etc.
- Search for code similarity.
- Helps teach classes due by showing, editing, and running code in the same environment students use.



## Program execution

The VPL execution service is responsible for receiving and controlling the execution of code. The execution can be terminated for four reasons:

- Finish their execution normally
- They are stopped when they deplete their assigned resources (time, memory, etc.)
- They are stopped on user requests (e.g. the user closes the browser). This monitoring is done through a WebSocket connection from the browser to the jail server.
- The Moodle server requests the stop of the task. Each student can only have one running task; if the user requests to execute another task, the previous one will be stopped.

After the end of a task, the work area used is cleaned.

The VPL module uses http+XMLRPC to communicate with the jail server. The browser uses WebSocket (ws:) for monitoring the task and the interactive execution. It is also possible to use HTTPS and WSS (secure connection). If communications from the browser with the Moodle server use HTTPS, most browsers require WSS to connect with the jail server. The use of HTTPS and WSS

required to have a certificate in the execution server. The recommended way is using certificates signed by a known Certificate Authority.

## VPL activity in Moodle

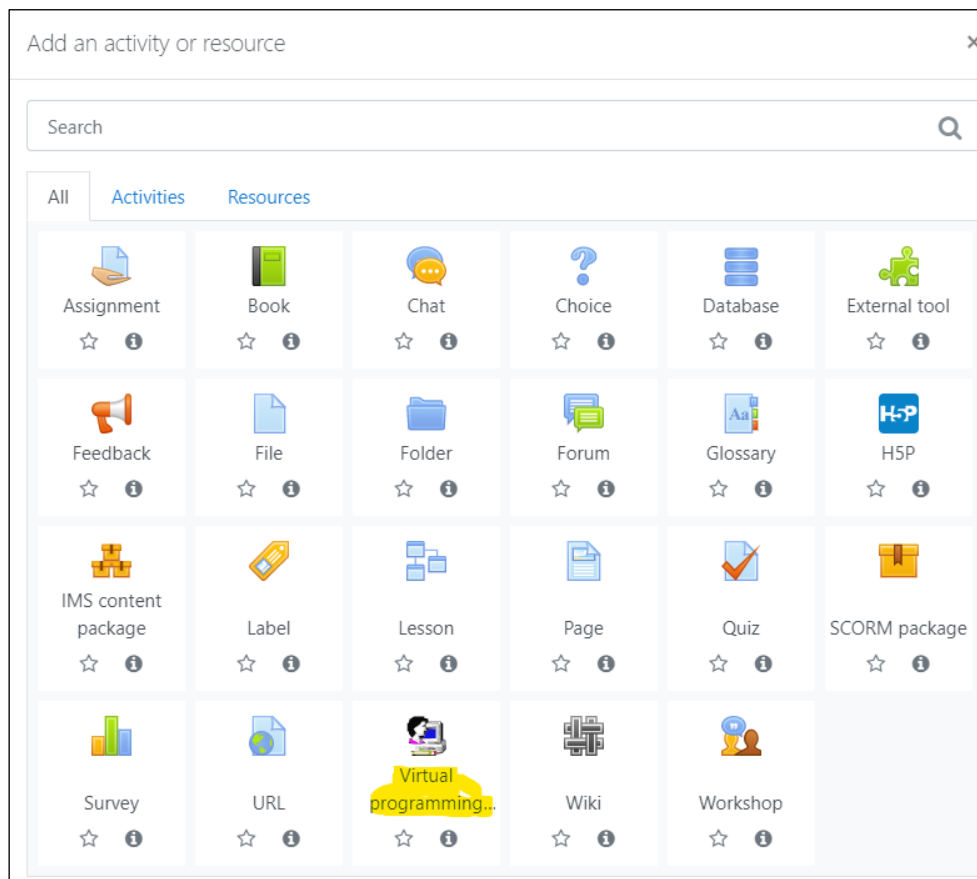
The following part is intended for teachers that want to start using VPL or need details of the basic VPL features. Teachers here will see how to create and configure programming activities for their courses and how VPL helps to monitor students' work.

### Creating VPL activity

To create a new VPL activity, you must set the course in editing mode by pushing the **Turn editing on** button as with other activity types. Then select the section where you want to add the activity and click on **Add an activity or resource** link.

[+ Add an activity or resource](#)

You have to select the Virtual Programming Lab activity.



All details of options you can set in this step will be shown below. After creating the activity, you can change these options at any time.

**Creating new VPL activity by duplicating another is an easy way to obtain a new activity preconfigured. VPL fully supports activity duplication.**

The simplest way to create new VPL activity contains the following steps.

- Start creating a VPL activity as shown at section start.
- Set the name of the activity. The system uses this name as the activity identification for users.
- Set the description. The description contains the details of the task the student must do.
- Set the due date. After the due date, students can not submit new code versions.
- Set the maximum number of files that the students can upload in each submission.
- If this activity gives grades to students, you must set the type of grade and the maximum grade.
- Save the new activity.
- Click on the activity name just created, and at the action menu, go to **Execution options** and set the execution actions (run/debug/evaluate) that you allow the students do.

The details of described steps and options of VPL activity follow.

### Name

The name of the activity is used to identify it. The name must be plain text. The name field is the only required field in this form.

### Short description

The short description is used to describe the activity when the full description is not available. The short description must be plain text.

### Full description

The full description is used as a detailed description of the activity. It is in HTML format and may contain images, mathematic equations, etc.

If you check the **Display description on course page** checkbox, the description will be shown on the course page just below the link to the activity.

### Submission period

Submission period

Available from

6

August

2021

09

31

☐ Enable

Due date

6

August

2021

09

31

☐ Enable

This setting allows limiting the time that students can submit files.

**Available from** set a time to start showing and usable the activity, before the set time the activity is not available for students. If not set, the activity will be usable if it is shown. If set and shown, the students can read the description before the **available from** but can not submit files.

**Note: When the start time (available from) is close to the current time (less than 5 minutes), the system will show the activity, and the students can access the description.**

**Due date** set a time to stop accepting submissions. If not set, while the activity is shown, the students submit files. After the due date, students can access the activity to see descriptions, download last submissions, etc.

### Submission restrictions

This option allows you to set restrictions primarily at the user interface or resource level (size and number of user files).

Submission restrictions

Maximum number of files

20

Type of work

Individual work

Dissable external file upload, paste and drop external content

Yes

This activity acts as example

No

Maximum upload file size

Select

Password

Click to enter text

Allowed submission from net

SEB browser required

No

SEB exam Key/s

**A maximum number of files** limits the number of files students can submit each time.

**Type of work** allows switching from individual work (the default) to **group work**. To activate group work, you must set a grouping of groups, each group with a team of students. Group work means that any student's submission belongs to the group. All group members can make submissions, access the last submission, and will get the same grade for the activity.

**Note:** *A student cannot belong to two or more groups of the grouping. Submissions belong to the group, and one student can be moved or removed from a group with no effect in submissions.*

**Disable external file upload, paste, and drop external content** – if the teacher sets this option, the students will not be able to upload files or paste code from external sources. The only way to write code is by typewriting in the IDE. The IDE still supports internal copy/paste functions.

**This activity acts as an example**, and it sets an option where the activity will be read-only. The teacher must write the example files in the Requested files. These files will contain the example of code that students can run or debug. Students can't submit or change the code.

**Maximum upload file size** – the teacher can set the maximum size of each file accepted in this activity.

**Note:** *Maximum upload file size may be affected by the option The based on feature.*

**Password** can be set by teachers for each activity. The student needs to enter the password to access any element of that activity (except the title). Entering the password gives permission to access the activity during the current session of the student, even if the key is changed. A common use is to give students the password to start the activity and change it once introduced, which prevents using the given password after the start of the activity.

**Note:** *Changing the password after all students access may give extra security.*

**Allowed submission from the net** – allows set which devices or networks can access the activity. This will prevent access to the activity from outside of the authorized networks. This feature may be **useful for exams**. The formats accepted are:

- xxx.xxx.xxx.xxx (IP). Example 1.2.3.4
- xxx.xxx.xxx.xxx/nn (number of bits in the mask). Example 1.2.3.128/4
- xxx.xxx.xxx.xxx yyy (IP range in the last group). Example 1.2.3.4 8
- xxx.xxx.xxx or xxx.xxx.xxx (incomplete address). Example 1.2.3

For allowing multiple networks or IPs, separate them by “,”. Example 1.2.3.4, 1.2.3.128/4, 1.2.3.4 8, 1.2.3

**SEB browser required to** support the use of the Safe Exam Browser (SEB).

**SEB exam Key/s** – if the key is set, the activity will require the Safe Exam Browser (SEB) with specific configuration by the **Browser Exam Key**. This feature may be **helpful for exams**. See <https://safeexambrowser.org>.

## Grade

In addition to the standard Moodle grade setting, there are options to control the students' evaluations.

Reduction by automatic evaluation	?	<input type="text" value="3%"/>
Free evaluations	?	<input type="text" value="2"/>

This control is done by penalizing the final grade based on the values of two options:

- **Free evaluations.** This option set the number of automatic evaluations students can request with no penalization.
- **Reduction by automatic evaluation.** Once the student expends all the free evaluations, this option is applied. The value can be a fixed value or a percentage. For each new automatic evaluation, the reduction is applied to the current grade. The percentage is applied to the current grade, not the maximum grade. A repeated assessment of the same submissions does not count.

### Example of use 1

With maximum grade set to 10, **Free evaluations** set to 3, and **Reduction by automatic evaluation** set to 0.5. A student with five evaluations and an initial grade of 8 will get 7 points.

$$8 - (5 - 3) * 0.5 = 7$$

### Example of use 2

With maximum grade set to 10, **Free evaluations** set to 2, and **Reduction by automatic evaluation** set to 10%. A student with four evaluations and an initial grade of 8 will get 6.48 points.

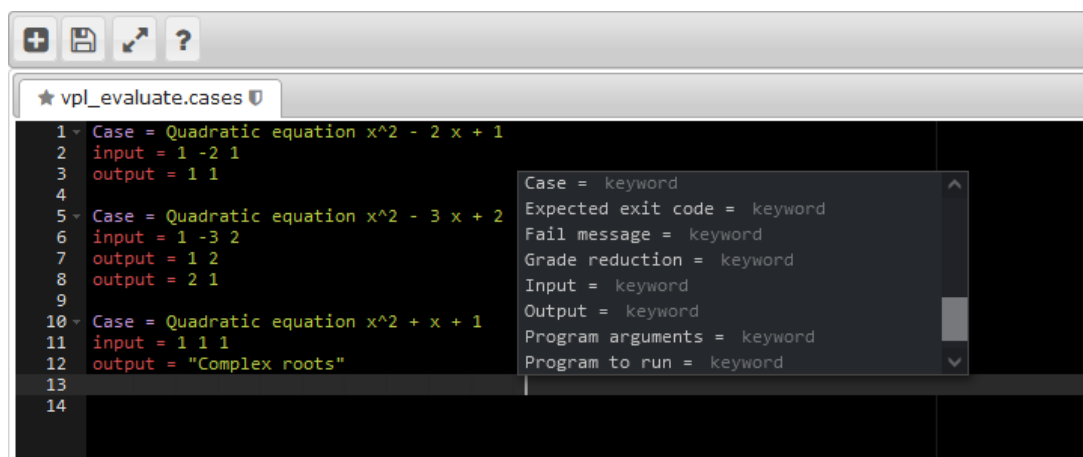
$$8 - (10\% \text{ of } 8) = 7.2$$

$$7.2 - (10\% \text{ of } 7.2) = 6.48$$

## Test cases

To use the feature of automatic program evaluation of VPL, teachers must populate the **vpl\_evaluate.cases** file going to Action menu ► Test cases.

**To create automated tests** teacher can use the auto-complete feature of the editor (Ctl+space), but notice that **the exact output match must go in double-quotes**. A detailed explanation of this feature will be presented later.



The figure above shows editing **vpl\_evaluate.cases** with two cases and auto-complete (*Ctrl-space*).

## Execution options

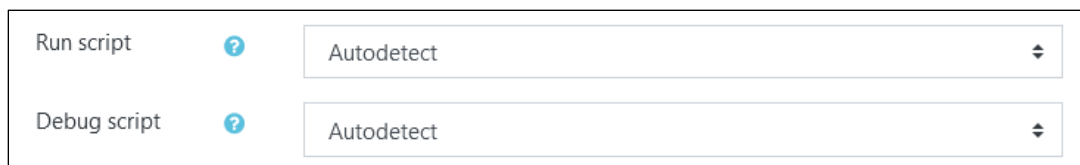
These options can be set going to **Action menu ► Execution options**.

**Based on** is a powerful feature that allows us to inherit the options and files of other VPL activities.

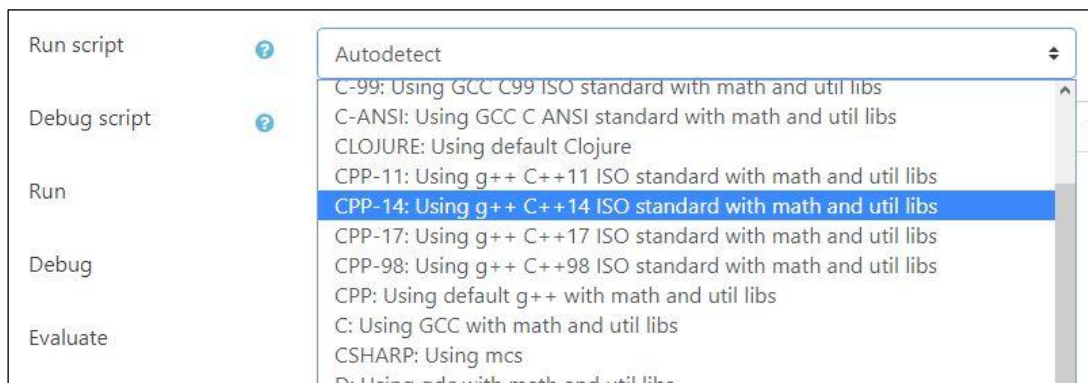


**Selecting programming language tools** allows selecting the tool used for running and debugging programs based on the filename extension of the submitted files. The system searches for a known extension following the order of submitted files. Once a known extension is found, the system uses the default associate tool.

This option allows setting the compiler/interpreter or debugger to use. Different versions are available for some programming languages, such as Python 2 or Python 3, C ANSI or C ISO 11, etc.



Selection of run and debug script.



Selecting run script.

- **Run:** The teacher must set to 'Yes' to allow the students to run programs in IDE.
- **Debug:** The teacher must set to 'Yes' to allow the students to debug programs in IDE.
- **Evaluate:** The teacher must set to 'Yes' to allow the students to run automatic evaluations.

**Note:** Users with grading capability can always run, debug, or evaluate.

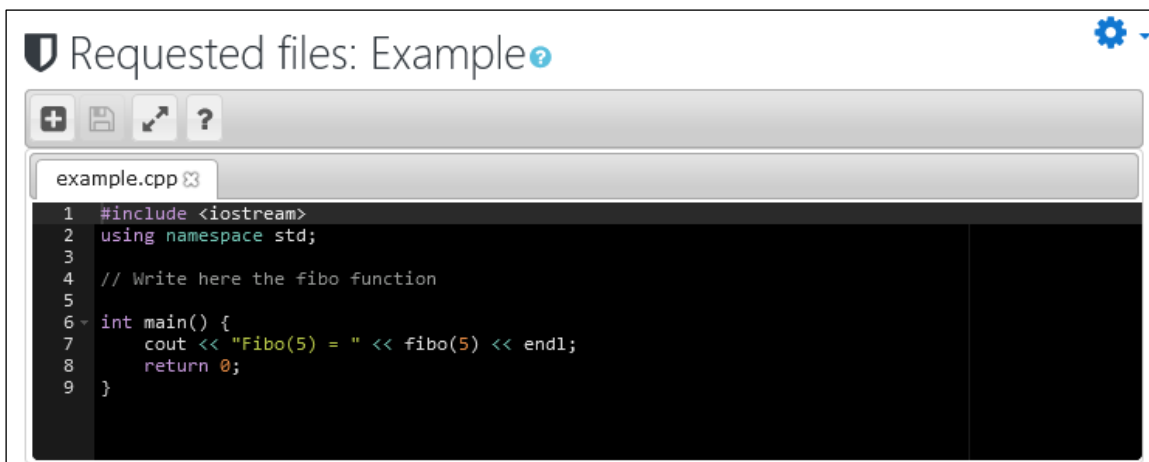
- **Evaluate on submission** - The submission is evaluated automatically when uploaded but not when saved in IDE.
- **Automatic grading** - If the evaluation result includes grading, they are used to set the grade automatically in Moodle grade book.

Run	No ▾
Debug	No ▾
Evaluate	No ▾
Evaluate just on submission	No ▾
Automatic grade	No ▾

## Requested files

This feature allows teachers to control the file names students submit and create activities in which the students must complete a task.

The teachers can set the names of the files that the students must submit and the initial content of these files. The teacher must use the editor to set the file names and their content. Notice that for some programming languages in VPL, the order of the files may be important.



```


1 #include <iostream>
2 using namespace std;
3
4 // Write here the fibo function
5
6 int main() {
7     cout << "Fibo(5) = " << fibo(5) << endl;
8     return 0;
9 }



```

The number of files must be less or equal to the max number of files set in the **Edit settings** of the activity. If the teacher doesn't set names for all the files, the unnamed files are optional, and the student can use any name.

This set of files are available for download with the description of the activity. The first time the student accessed the IDE, the editor loaded these files with their content. The IDE has an option to reset these files to their initial content.

### Example

 **Due date:** Saturday, 29 August 2020, 1:00 AM

 **Requested files:** example.cpp ( [Download](#))


**Note:** If the name of files changes after the students start working on the activity, they can need to move their code to the new files manually.

## Submissions report



The option **Submission list** reports the current situation of the students' tasks in the activity, allowing multiple actions such as accessing each student's submissions, evaluating, modifying submissions, automatic battery evaluation, viewing other reports, etc.

This report shows for each student the following columns:

- A sequence number to easily identify the number of students that match any criteria. If clicked, goes to edit a copy of the last submission in the teacher's workspace.
- The student photo - is omitted if there are too many students. If clicked, it goes to the student's profile.
- The student's full name.
- The last submission date and time – if the option is clicked, show the student's submission.
- A number of submissions that are saved in the system - if it is clicked, it goes to the student's previous submissions report.
- If available, the proposed or final grade. If available and clicked, it goes to student grade.
- If an assignment is graded manually, the name of the evaluator.
- If graded, the date and time of grading.

		First name ↓ / Surname ↓	Submitted on ↓	Submissions ↓	Grade ↓	Evaluator ↓	Evaluated on ↓	⚙️ ↓
1		[blurred]	Thursday, 21 May 2020, 5:28 PM	1	10.00 / 10.00	Juan Carlos Rodríguez del Pino	Wednesday, 15 July 2020, 11:01 PM	⚙️ ↓

The actions get by clicking are also available at the action menu at the end of each student's row.

		First name ↓ / Surname ↓	Submitted on ↓	Submissions ↓	Grade ↓	Evaluator ↓	Evaluated on ↓	⚙️ ↓
1		[blurred]	Thursday, 21 May 2020, 5:28 PM	1	10.00 / 10.00	Juan Carlos Rodríguez del Pino	Wednesday,	⚙️ ↓
2		[blurred]	Wednesday, 27 May 2020,	1	10.00 / 10.00	Jua Ro		

The report can be ordered as ascending or descending by any column.

The **submission selection** option allows to filter the students:

- **All:** Shows all students with or without submissions.
- **All submissions:** Select the students with submissions. This is the default option.
- **Not graded:** Select the students with submissions not graded.
- **Graded:** Select the students with submissions graded.
- **Graded by user:** Select the students with submissions graded by the current user.

Submission selection: All submissions

Evaluate: Choose... Choose... All All submissions Not graded Graded Graded by user

	Submitted on	Submissions	Grade	Evaluator	Evaluated on
1	Thursday, 21 May 2020, 5:28 PM	1	10.00 / 10.00	Juan Carlos Rodríguez del Pino	Wednesday, 15 July 2020, 11:01 PM

The **Evaluate** option allows to launch a batch evaluation of the selected students and submissions that match the criteria:

- **Not executed:** Option evaluates the submissions not previously evaluated.
- **Not grade:** Option evaluates the submissions not graded.
- **All:** Option evaluates all submissions.

Evaluate: Choose... Choose... Not executed Not graded All

	Submitted on	Submissions	Grade	Evaluator	Evaluated on
1	Thursday, 21 May 2020, 5:28 PM	1	10.00 / 10.00	Juan Carlos Rodríguez del Pino	Wednesday, 15 July 2020, 11:01 PM

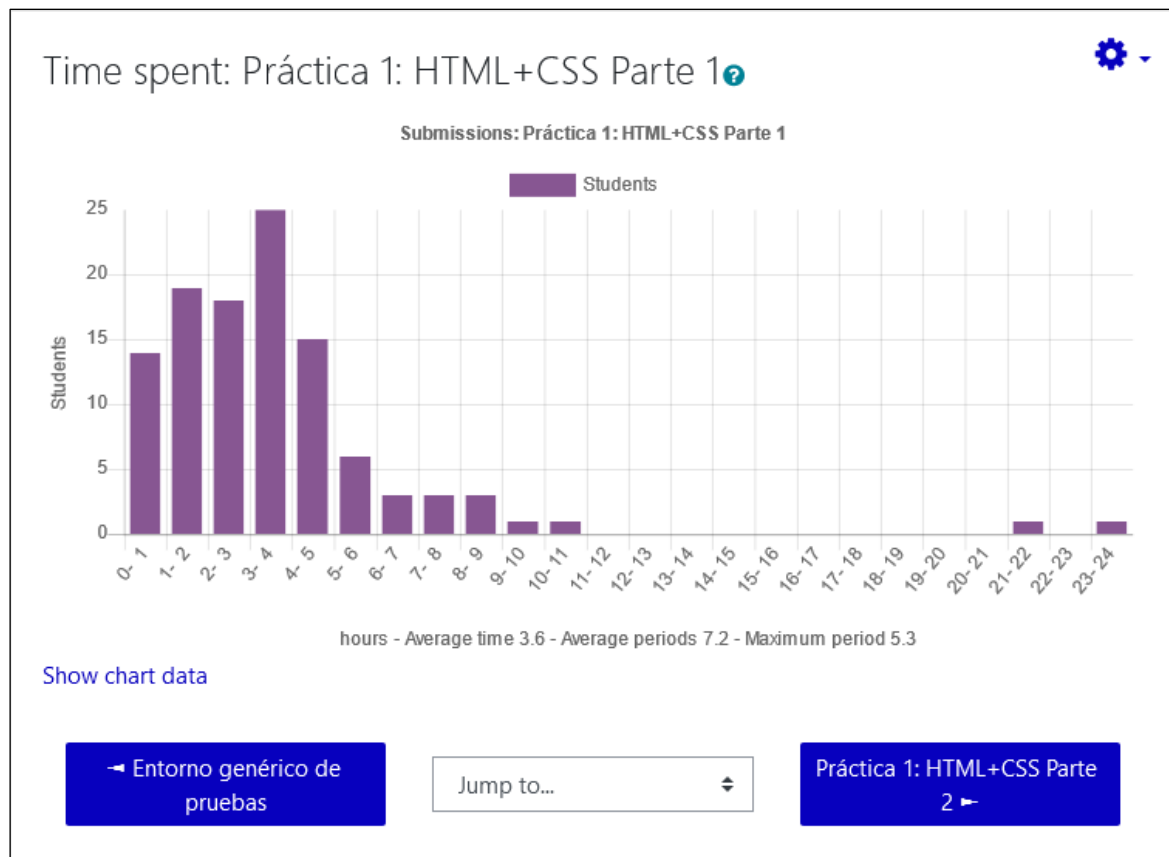
The action menu at the end of the header row allows access to the other reports and downloads that are shown below.

	First name / Surname	Submitted on	Submissions	Grade	Evaluator	Evaluated on
1	Juan Carlos Rodríguez del Pino	Thursday, 21 May 2020, 5:28 PM	1	10.00 / 10.00	Juan Carlos Rodríguez del Pino	Wednesday, 15 July 2020, 11:01 PM
2	Juan Carlos Rodríguez del Pino	Wednesday, 27 May 2020, 5:28 PM	1	10.00 / 10.00	Juan Carlos Rodríguez del Pino	Wednesday, 15 July 2020, 11:01 PM

Action menu options: Submissions, Assessment report, Download submissions, Download all submissions

### Time spent report

This graphic report is calculated based on the submissions saved and shows the number of students by hours spent in the tasks. It also shows the average of each student's work periods and the time spent in the largest period.



## Assessment report

This report is similar to the **Submission list** but shows the details of the automatic evaluation and final feedback, for each student shows the following field:

- A sequence number to identify easily the number of students that match any criteria. If clicked, goes to edit a copy of the last submission in the teacher's workspace.
- The student photo – is omitted if there are too many students. If clicked, it goes to the student's profile.
- The student's full name.
- If available, the proposed or final grade. If available and clicked, it goes to student grade.
- The automatic evaluation result and final assessment if available.
- Action menu to access the student's submission.

## Download submissions

This action downloads a ZIP file containing the last submission of each student. The zip file contains for each student a directory with name + id + username. The directory contains another directory with the name date+time of submission that contains the submissions files. There is another sibling directory with name date+time + '.ceg' that contains the files: compilation.txt, execution.txt, grade.txt, and gradecomments.txt.

An entry example:

```
Jone Doe 13 jone
2021-05-21-10-12-45
myclass.h
```

```

myclass.cpp
main.cpp
2021-05-21-10-12-45.ceg
compilation.txt
execution.txt
grade.txt
gradecomments.txt

```

The action **Download all submissions** contains all the saved submissions of the students.

## Similarity

This feature search for similarity in a set of files generating a report of similar pairs of files ordered from most to fewer similar. The basic set of files are the last submitted files in the activity, and the form allows adding other sets of files for searching.

The search uses a mix of three different metrics. Code changes affect in a different way to each metric.

Change Metric	Comments	Name of Identifiers	Code reorder	Systematic change (expression)	Complex change (expression)
Metric 1	Filtered	Filtered	Not affected	Slightly affected	Affected
Metric 2	Filtered	Filtered	Not affected	Affected by size of changes	Affected by size of changes
Metric 3	Filtered	Filtered	Affected	Affected by number of changes	Affected by number of changes

The table above presents the effects on the metrics of different code changes.

Scan options are supported by the parameter **Maximum output by similarity**, which is used as a cut point for the more similar selected pairs by metric and has no other influence.

The difference between a report setting **Maximum output by similarity** to 40 and another setting it to 45 is that the new report must be the same but with 5-15 new pairs with less similarity. Notice that if we have paired with the same similarity, the output may vary.

Similarity

Separate groups
 

All participants

Scan options
 

Maximum output by similarity
 

20

The option **Files to Scan** allows selecting the file names to compare. Notice that these files must be in the list of Requested files. Also, you can select to compare all files regardless of name or join all files on each student's submission.

Files to scan

☒ funcionales.scm
   
☐ All files
   
☐ Joined selected files

The option **Other sources to add to the scan** allows for incorporating more file sets into the search. An option enables adding another VPL activity to the search. Activities of other courses are available in this selection. Another option allows adding a zip file containing the set of files to add to the search.

By default, these files are not compared to each other. The last option allows to enable it. Using this option on an empty activity and adding a zip file containing the set of files to scan allows using this feature for search similarity on external files.


Other sources to add to the scan

Activity

Select

Zip file


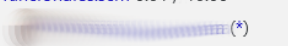



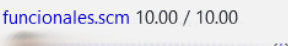


Choose a file...



You can drag and drop files here to add them.

☐ Scan similarities in added sources

The report orders the output from more to less similar pairs. Each report row shows the first filename and student, similarity rate, and second file and student. The similarity rate has the format of “metric1 percent | metric2 percent | metric3 percent | asterisk (1..3)”. As a per cent, each metric result also shows “\*” to indicate the metrics that set the pair in an order position less than **Maximum output by similarity**. After the ordered pairs, the report shows clusters of similar submissions.

Similarity		List of similarities found	
#	First name / Surname	Similar to	Cluster #
1	funcionales.scm 0.01 / 10.00  (*)	100 100 100*** funcionales.scm 0.01 / 10.00  (*)	4
2	funcionales.scm 0.01 / 10.00  (*)	100 100 100*** funcionales.scm 0.01 / 10.00  (*)	1
3	funcionales.scm 0.01 / 10.00  (*)	93 98 94*** funcionales.scm 10.00 / 10.00  (*)	2
4	funcionales.scm 0.01 / 10.00  (*)	87 98 94*** funcionales.scm 10.00 / 10.00  (*)	2

### Example of similarity report

The system tries to help the teacher compare pairs of files by showing them side by side and adding blank lines to align similar lines of code. You must click on the similarity rate of a pair of files to get this side by side report. Also, by clicking the (\*) near the student's name, an individual report is generated.

```
funcionales.scm                                     funcionales.scm
1  ((require errortrace) ;; Ofrece más información al produ | 1  ((require errortrace) ;; Ofrece más información al produ
2  (require racket/trace) ;; Permite seguir la ejecución d | 2  (require racket/trace) ;; Permite seguir la ejecución d
>>> 3
>>> 4 ;;Funcion fibonacci
=== 5
=== 6 (define (fibonacci num)
=== 7 (define (fibonacci n)
8  (cond
=== 8  (cond
9  ((= num 0) 0)
=== 9  ((= 0 n) 0)
10 ((= num 1) 1)
=== 10 ((= 1 n) 1)
11 (else (+ (fibonacci (- num 1)) (fibonacci (- num 2)))))
=== 11 (else (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))
12 )
=== 12 )
>>> 13 ;;(trace fibonacci)
>>> 14 ;;(display (fibonacci 7))
=== 15
>>> 16 ;;Funcion expo
=== 17
=== 18 (define (exponenciacion num exp)
=== 18 (define (exponenciacion b n)
19 (cond
=== 19 (cond
20 ((= exp 0) 1)
=== 20 ((= n 0) 1)
21 (else (* num (exponenciacion num (- exp 1)))))
=== 21 (else (* b (exponenciacion b (- n 1)))))
22 )
=== 22 )
>>> 23
>>> 24
>>> 25 ;;(trace exponenciacion)
>>> 26 ;;(display (exponenciacion 2 3))
=== 27
>>> 28 ;;Uso de listas
>>> 29 ;; funcion minimo
=== 30 (define (minimo lista)
=== 30 (define (minimo lista)
31 (cond
=== 31 (cond
32 ((null? (cdr lista)) (car lista))
=== 32 ((null? (cdr lista)) (car lista))
33 ((< (car lista) (minimo (cdr lista))) (car lista))
=== 33 ((< (car lista) (minimo (cdr lista))) (car lista))
34 (else (minimo (cdr lista)))
=== 34 (else (minimo (cdr lista)))
35 )
=== 35 )
>>> 36
>>> 37 )
=== 38 )
```

The figure above presents the example of side by side file comparison report

The side-by-side file report shows in the middle of the two files the differences between the two paired lines using a code:

- ===: The two lines are identical.
- <<<: The left line is not present in the right file.
- >>>: The right line is not present in the right file.
- ==#: The two lines without spaces are identical.
- =##: The two lines without alphanumeric are identical.
- ###: The two lines do not match any previous criteria.

Considering that the similarity report can give “**false positives**” and that the report has no external consideration that may affect the case. The use of similarity reports can follow these considerations and recommendations:

- The similarity report is **NOT** criteria to assure that plagiarism happened.
- The report is information for teachers but never must be used as a direct verdict.
- **Always** must be one or more teachers who judge the case.
- The teacher must have proofs based mainly on the code and not on the similarity report of VPL.
- Based on his own criteria, the only trustable report about the possibility of unfair behaviour is generated by a teacher.
- The similarity report can help the teacher to select what pair of submissions must be reviewed in detail.

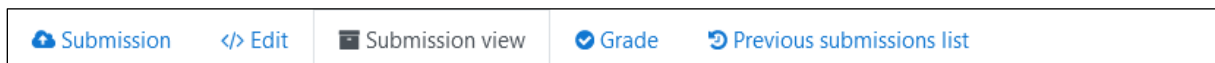
- A common way to study the report is to review the pairs from most to less similar and stop reviewing when enough adjacents “false positives” are found.

**Note:** In this context, a “false positive” is pair of files that the metrics indicate a high similarity, but an expert indicates that there is no reason to consider plagiarism.

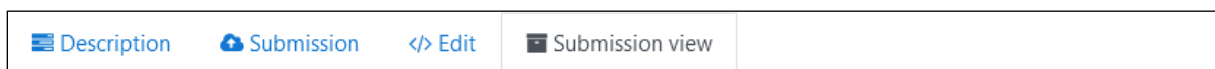
## Test activity

To test the activity, you can switch temporarily to the student role. But this way to test the activity will allow students to access the activity if you don’t temporarily **Restrict access** to the other students. However, VPL allows test activities without changing the role by accessing the menu **Edit settings -> Test activity**. The teacher can similarly access the activity as a student does but in a specific workspace for him. The difference is the **Previous submissions list**, **Grade**, and that teacher can use all options (run/debug/evaluate) regardless of the setting in **Execution options**.

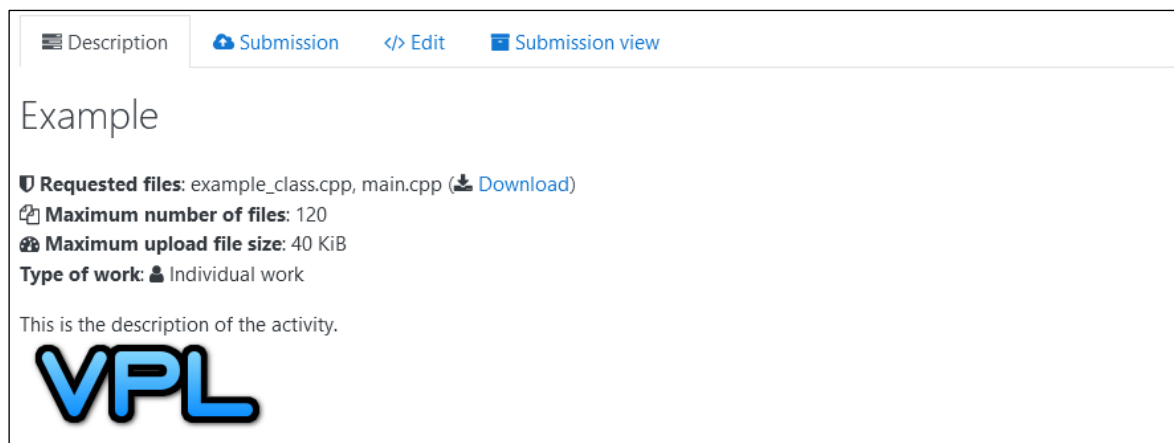
The activity menu for the teacher role looks like this:



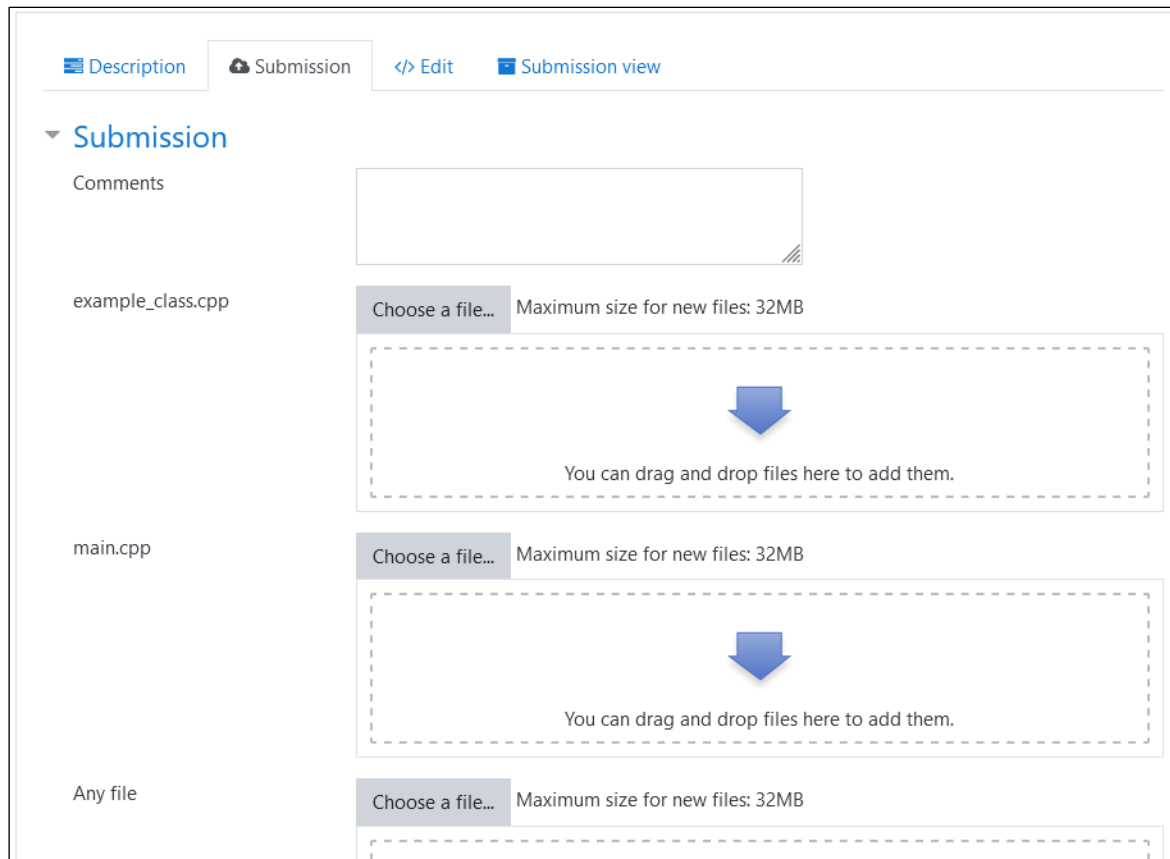
And the activity menu assigned to the student role is as follow:



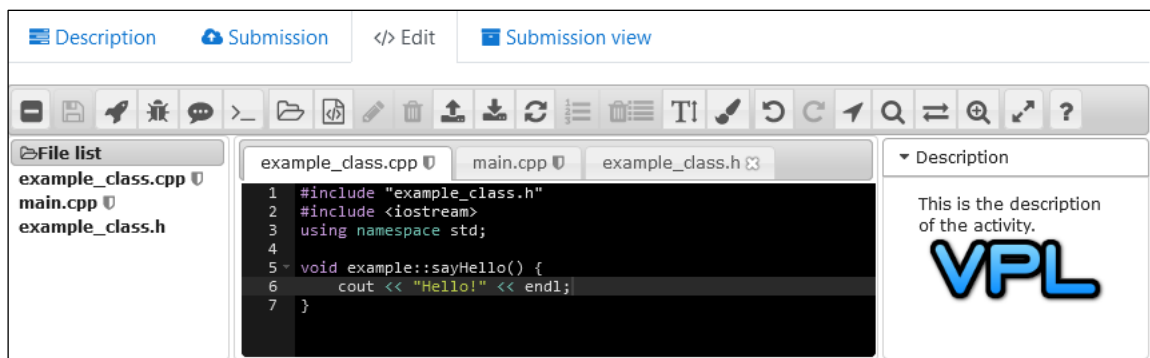
The part **Description** shows the description of the activity and the different settings that affect the task. The settings shown are different based on the role of the user. Teachers get more detailed information on settings.



The page **Submission** allows students to upload the files to submit. This interface has some drawbacks, and students cannot set directories for the files uploaded. Also, notice that this page may be restricted by the **Disable external file upload, paste, and drop external content option**. It may be more convenient to use the **Edit** page with a similar effect and more flexibility.

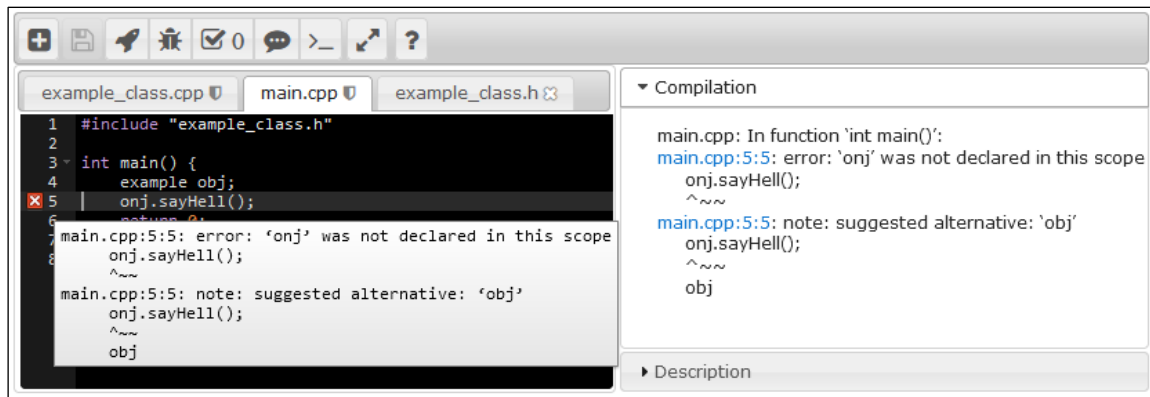


The page **Edit** gives access to students to edit, run, debug, or evaluate a new or previous submission. Saving here is equivalent to uploading a submission. This page allows teachers to test the activity or access a student's submission to run/debug/evaluate it or even change the student's submission when saving. The teacher can know if he is accessing a student's submission because, in that case, the menu shows the student's name.



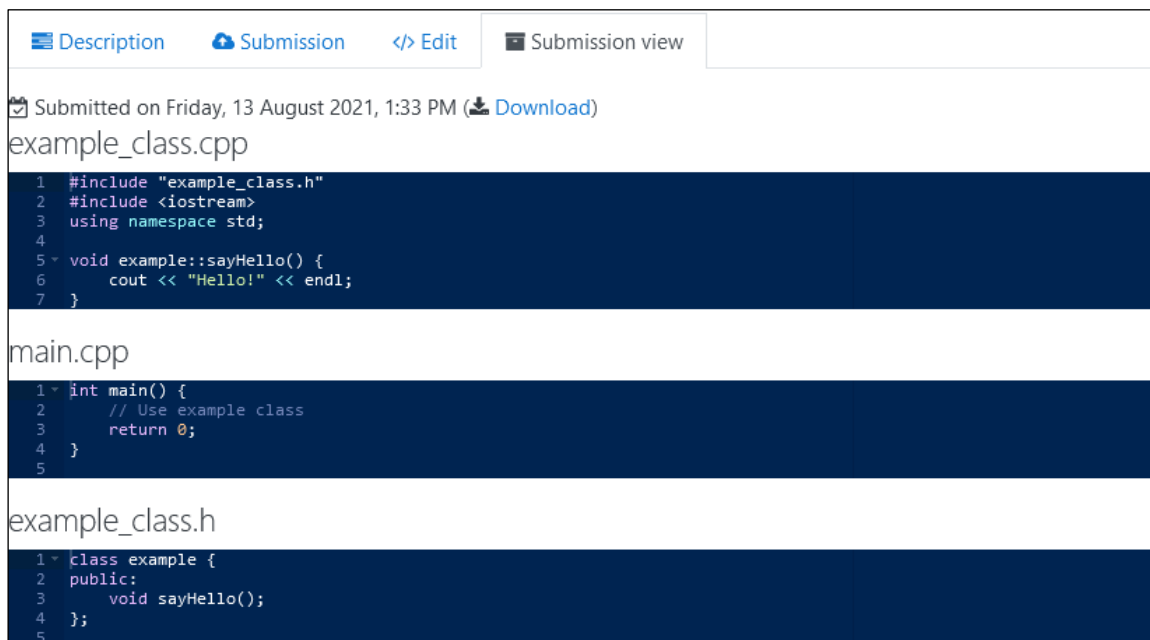
The figure above shows the example of editing files by the student role.

The next figure shows the example of running code by the student role.



## Submission view

The page **Submission view** shows students its last submission. It also allows to download the submission as a zip file or ask for automatic evaluation if available. This page also shows teachers the corresponding student's last submission.



The page **Grading a submission** allows teachers to grade a student's last submission manually. The page shows a form to introduce the grade and the comments (feedback) for the student. The form also shows different actions on the grade of the submission:

- **Grade:** This input must not be empty. The penalization (grade reduction) for asking automatic evaluations applies to the introduced value.
- **Comments:** This is a multi-line entry with feedback for the student. The content of this input is formatted when shown.
- **Grade button:** this button saves the grade to the Moodle grade book.
- **Grade & Next button:** This button, available on batch grade, saves the grade to the Moodle grade book and goes to the next student's last submission.

- **Copy button:** This button asks the system to copy the student's submission to the teacher's workplace. The teacher can there check the student's code by running, debugging, or fixing it.
- **Evaluate button:** This button asks the system to run the automatic evaluation for this submission. The automatic evaluation does not change a manual grade.
- **Calculate button:** This button asks the system to review the comment and calculate the grade based on its formatting.

Grade

Comments

- This is a header. A penalization can go at the end of the line. (-10)  
To calculate the grade with penalization use the "Calculate button"  
Regular text.  
Auto link format file\_name:line\_number => Agricultor.java:8  
>     This is a preformatted text  
>         Hello!  
- A second header. (-15.5)  
More regular text.

---

0.16 hours  
Submitted on Sunday, 14 June 2020, 12:39 PM ([Download](#)) (☒ Evaluate)

Agricultor.java

```

1 package mercadomayorista;
2
3 public class Agricultor extends Thread{
4     private String id;
5     private Mayorista m;
6     private int vendido;
7
8     public Agricultor(String id, Mayorista m){
9         this.id = id;

```

Bellow the form, the page shows information as the **Submission view** with the automatic evaluation, the manual evaluation, and the files submitted. Also, at the right of the form, the list of previous headers used is shown. These headers may be reused by clicking on them.

The student can see the grade and feedback in the Moodle grade book or the **Submission view**.

## Grade

Reviewed on Friday, 13 August 2021, 6:28 PM by Admin User

**Grade:** 74.50 / 100.00

### Assessment report [-]

**[!] This is a header. A penalization can go at the end of the line.**

To calculate the grade with penalization use the "Calculate button"

Regular text.



Auto link format file\_name:line\_number => [Agricultor.java:8](#)

*This is a preformatted text*

*Hello!*

**[!] A second header.**

More regular text.

 Submitted on Sunday, 14 June 2020, 12:39 PM ( Download) (☒ Evaluate)

### Agricultor.java

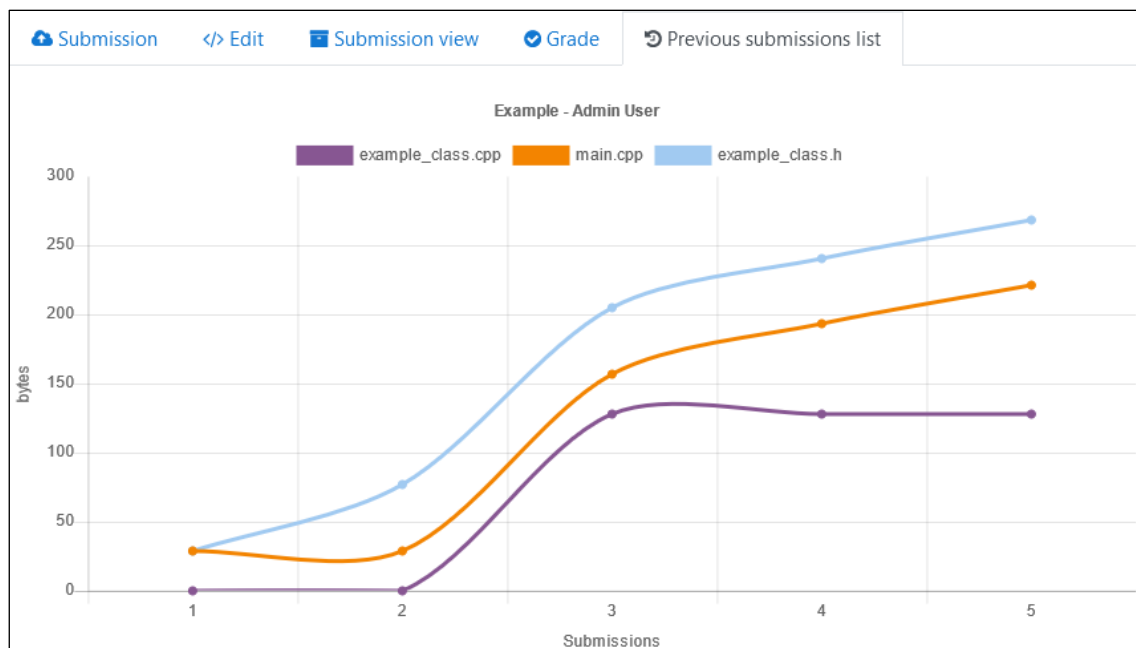
```

1 package mercadomayorista;
2
3 public class Agricultor extends Thread{
4     private String id;
5     private Mayorista m;
6     private int vendido;
7
8     public Agricultor(String id, Mayorista m){
9         this.id = id;
10    }
11    Auto link format file_name:line_number => Agricultor.java:8
12    this.vendido = 0;
13 }

```

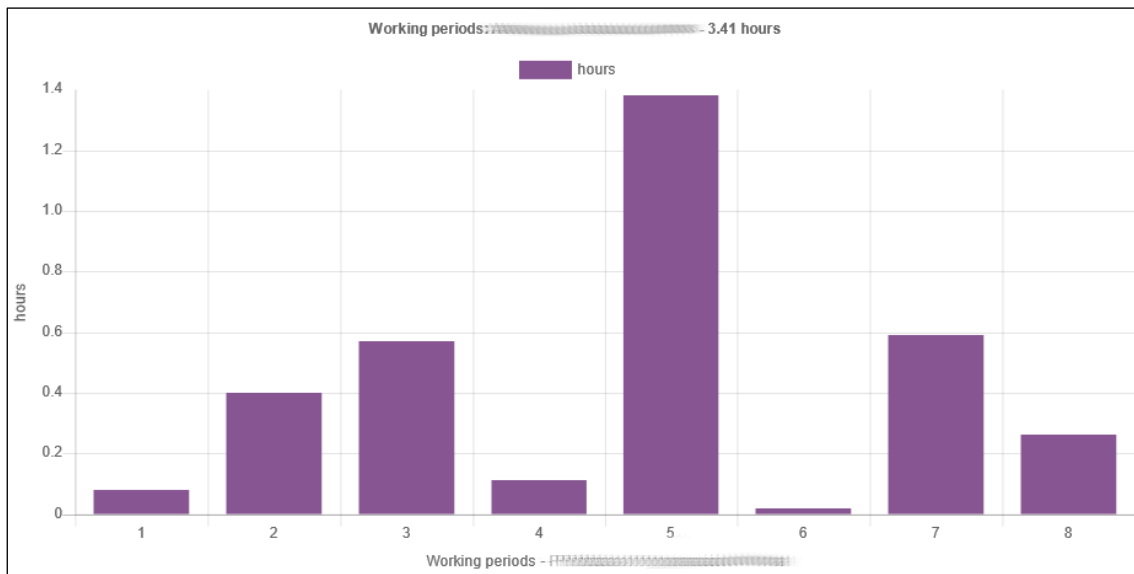
The page **Previous submissions list** shows the list of submissions saved in the system for a student and an activity. It also shows two reports obtained from analyzing these submissions.

This graphic report shows the evolution of the size in bytes of each file in each submission.



The following graphic report shows the student's time spent in this activity. The horizontal axis represents each period of the student's continuous work time, and the vertical axis is the time of

each period in hours. Notice that these parameters are calculated based on the submissions saved; then, they estimate the parameter, not its real value.



The last part of this page is the list of all submissions of the student order from newest to oldest. For each submission, the system shows:

- A sequence number.
- The submission date.
- The description that contains the names of the files, their size in bytes, and their number of lines.
- An action menu that allows to see or copy in the teacher's workspace any submission.

Also, there is the possibility to see this page with **More details**, meaning also shows the content of all the files of the student's submissions.

#	Date submitted	Description	Action
5	Friday, 13 August 2021, 2:03 PM	example_class.cpp 128b 8l, main.cpp 93b 8l, example_class.h 48b 5l	⚙️
4	Friday, 13 August 2021, 2:02 PM	example_class.cpp 128b 8l, main.cpp 65b 6l, example_class.h 48b 5l	⚙️
3	Friday, 13 August 2021, 1:55 PM	example_class.cpp 128b 8l, main.cpp 29b 4l, example_class.h 48b 5l	⚙️
2	Friday, 13 August 2021, 1:44 PM	example_class.cpp 0b 1l, main.cpp 29b 4l, ex	<div> <div>Submission view</div> <div>Copy</div> </div>
1	Friday, 13 August 2021, 1:40 PM	example_class.cpp 0b 1l, main.cpp 29b 4l, example_class.h 0b 1l	⚙️

Less detailed ▾

## Assignment list in Virtual Programming Labs

This report is only available from the action menu and shows the VPL activities in a course. To students, it shows its available activities. For teachers, it shows all the activities.

It shows the following data for each activity:

- **A sequence number.**
- **Section** - the section of the course where the activity appears. Clicking goes to the course section.
- **Name** - the activity name. Clicking goes to the activity.
- **Available from** - the date the activity is available for students.
- **Due date** - the date that the activity does not accept more submissions.
- **Submissions** - the number of students' submissions. Clicking goes to the list of students' submissions.
- **Graded** - the number of students' submissions that have been graded and not graded in parenthesis. Clicking goes to the list of students' submissions not graded.

Virtual programming labs

Section: All

Filter: Manual grading

#		Name ↑	Available from ↑	Due date ↑	Submissions	Graded
1		Prueba		Tuesday, 6 April 2021, 1:00 AM	1	0 (1)
2		Other example			62	0 (62)
3	Actividades a realizar	Example			0	0
4	Actividades a realizar	Práctica 1: HTML+CSS Parte 1	Wednesday, 6 February 2019, 12:00 AM	Friday, 21 February 2020, 8:00 PM	88	86 (2)
5	Actividades a realizar	Práctica 1: HTML+CSS Parte 2	Wednesday, 26 February 2020, 10:45 AM	Friday, 6 March 2020, 8:00 PM	85	85

The list of VPL activities in the course can be filtered in two ways: by selecting a section of the course and by selecting the state of the activity:

- **Open** - select the activities that, at this moment, students can submit files.
- **Closed** - select the activities that, at this moment, students can not submit files.
- **Time limited** - select the activities that have set a **due date**.
- **Time unlimited** - select the activities that have not set a **due date**.
- **Automatic grade** - select the activities that have grades and have set automatic grading.
- **Manual grading** - select the activities that have grades and have not set automatic grading.
- **Examples** - select the activities that are of type example.

## VPL Test Case language

The basic input/output test evaluation system (BIOTES) provided by VPL out of the box use special language. To evaluate the students' program, the evaluator writes the test cases in the **vpl\_evaluate.cases** file using this language.

The language uses statements with the format **"statement = value"**. The statement takes the whole line. Based on the statement type, the value can take only one line or spans multiple ones. A multiline value ends when another statement appears. The statement name is case insensitive.

Each test case definition includes a case name, the input we want to provide to the student's program and the output we expect. We can also configure other stuff, as the penalization for failed tests. VPL will run the evaluation applying the test cases and generating a report of failed cases and the mark obtained.

### Basic definition

The statement starts a new case definition and states the case description.

#### Format:

```
"Case = Test case description"
```

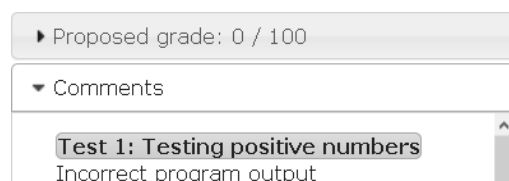
The case description occupies only one line. This description will appear in the report if the case fails.

Example:



```
vpl_evaluate.cases
1 Case = Testing positive numbers
2
3
```

Evaluation report showing the test case description:



► Proposed grade: 0 / 100

▼ Comments

Test 1: Testing positive numbers  
Incorrect program output

### Input

This statement defines the text to send to the student program as input. Each case requires one and only one input statement. Its value can span multiple lines.

#### Format:

```
"Input = text"
```

Example 1:

```
★ vpl_evaluate.cases
1 Case = Testing positive numbers
2 Input = 3 4 6 7
3
```

Example 2:

```
★ vpl_evaluate.cases
1 Case = Testing positive numbers
2 Input = 3
3 4
4 6
5 7
6
```

Example 3:

```
★ vpl_evaluate.cases
1 Case = Testing count words
2 Input = Text input of the student's program
```

Example 4:

```
★ vpl_evaluate.cases
1 Case = Testing count words
2 Input = Text input
3 of the
4 student's program
```

## Output

The output statement defines a possible correct output of the student program for the input of the current case. A test case can have multiple output statements and must have at least one. If the program output matches one of the output statements, the test case succeeds, else fails. There are four kinds of values for an output statement: numbers, text, exact text and regular expression.

### Format:

```
"Output = value"
```

The value of the output can span multiple lines.

### Checking only numbers

This type of output checks numbers in the response from the student's program, ignoring the rest of the text. To define this type of output check, you must use only numbers as values for the output statement. The output of the student's program is filtered, removing the non-numeric text. Finally, the system compares the resulting numbers of the output with those expected for the case, using a tolerance when comparing floating numbers.

Example 1:

```
★ vpl_evaluate.cases
1 Case = Testing count positive numbers
2 Input = 1 2 -3 5 6.5 -7
3 Output = 4
```

Student's program output that matches this definition:

```
>_ Console: connectio...
4

>_ Console: connectio...
The number of positives is: 4

>_ Console: connectio...
found 4 positive numbers

>_ Console: connectio...
4
is the
positive numbers
in the input
```

Example 2:

```
★ vpl_evaluate.cases
1 Case = Testing filter positive numbers
2 Input = 1 2 -3 5 6.5 -7
3 Output = 1 2 5 6.5
```

Student's program output that matches this definition:

```
>_ Console: connectio...
1 2 5 6.5

>_ Console: connectio...
1, 2, 5, 6.5

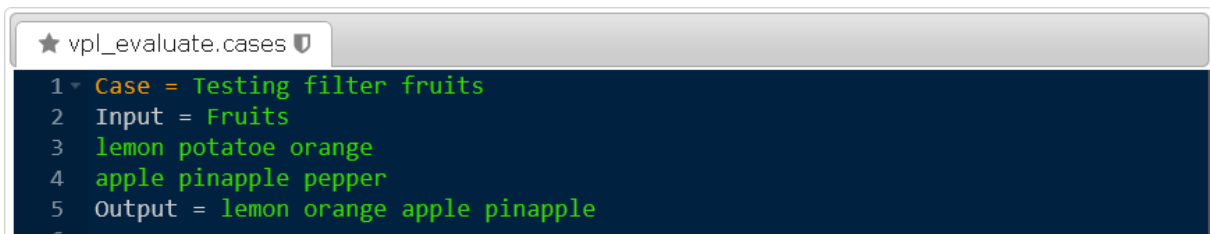
>_ Console: connectio...
One 1
Two 2
Three 5
Four 6.5

>_ Console: connectio...
The positive numbers are:
[ 1, 2, 5, 6.5]
```

## Checking text

This type of output is a **nonstrict text check** comparing only words in the output of the student's program. The comparison is **case-insensitive and ignores the punctuation marks, spaces, tabs, and newlines**. To define this type of output check, you must use text (may include numbers, but non only) non starting with a slash or being inside double-quotes. A filter removes punctuation marks, spaces, tabs, and newlines from the output of the student's program, leaving a separator between each word. Numbers are not punctuation marks, so they are not removed. Finally, the system compares case-insensitive the resulting text with the expected output.

Example:

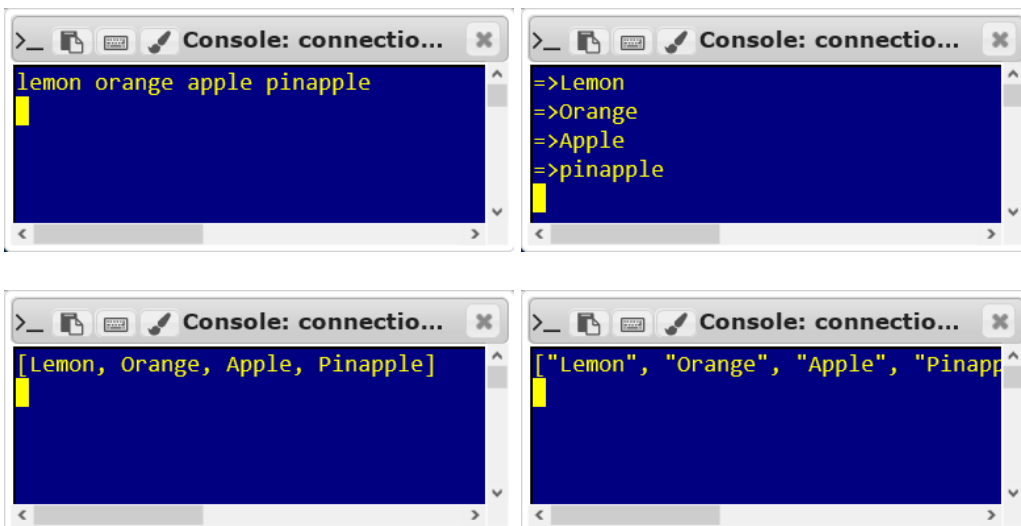


```

1 Case = Testing filter fruits
2 Input = Fruits
3 lemon potatoe orange
4 apple pinapple pepper
5 Output = lemon orange apple pinapple
6

```

Student's program output that matches this definition:



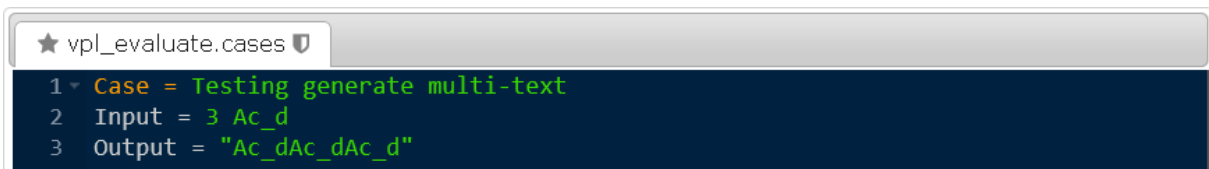
The four console windows show the following outputs:

- lmon orange apple pinapple
- =>Lemon  
=>Orange  
=>Apple  
=>pinapple
- [Lemon, Orange, Apple, Pinapple]
- ["Lemon", "Orange", "Apple", "Pinapple"]

## Checking exact text

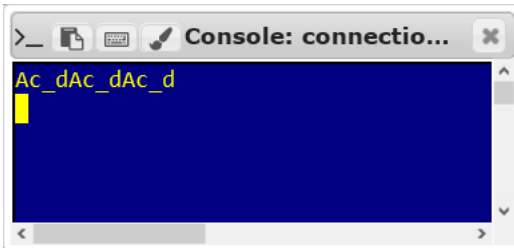
This type of output checks the exact text on the output from the student's program. To define this type of output check, you must use text enclosed in double-quotes. The system compares the output of the program with the defined output (removing double quotes).

Example 1:



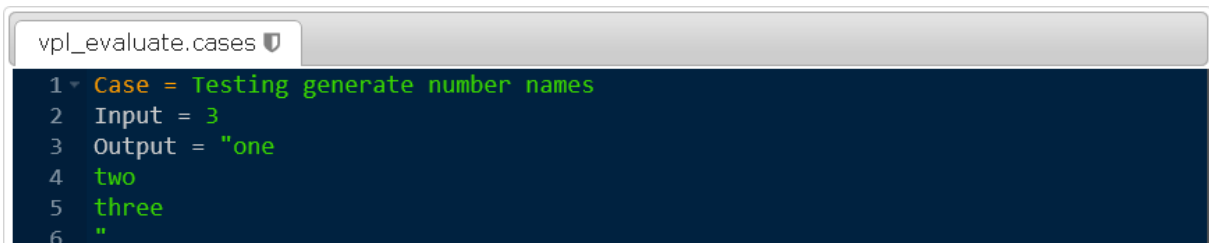
```
★ vpl_evaluate.cases
1 ▾ Case = Testing generate multi-text
2   Input = 3 Ac_d
3   Output = "Ac_dAc_dAc_d"
```

Student's program output that matches this definition:




```
>_ Console: connectio...
Ac_dAc_dAc_d
```

Example 2:



```
vpl_evaluate.cases
1 ▾ Case = Testing generate number names
2   Input = 3
3   Output = "one
4           two
5           three
6           "
```

Student's program output that matches this definition:

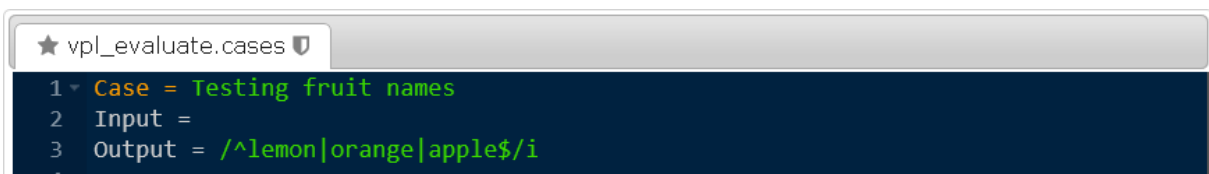


```
>_ Console: connectio...
one
two
three
```

### Checking regular expression

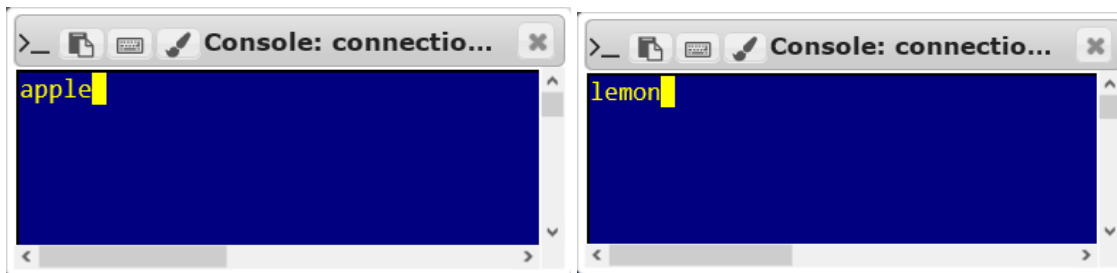
The evaluator can define this type of check, starting the output value with a slash "/" and ending with another slash "/" plus optionally one or several modifiers. This format is similar to JavaScript literal regular expression but uses POSIX regex instead.

Example:



```
★ vpl_evaluate.cases
1 ▾ Case = Testing fruit names
2   Input =
3   Output = /^lemon|orange|apple$/i
4
```

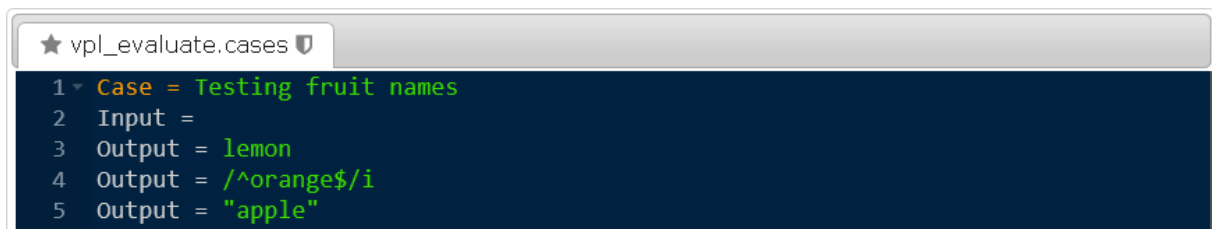
Student's program output that matches this definition:



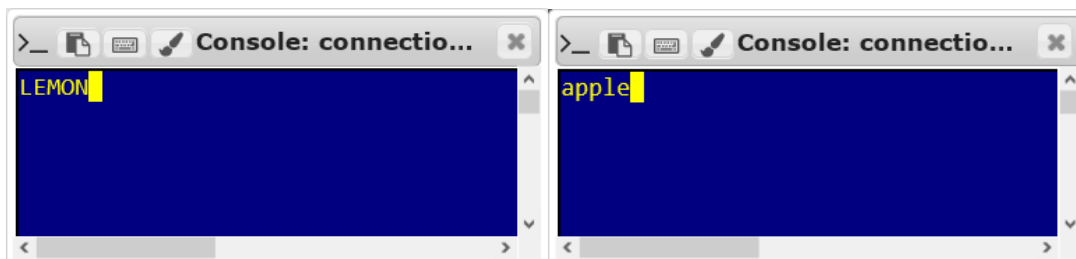
### Multiple output checking

The test case definition may contain multiple output statements, meaning that the case succeeds if any of them matches.

Example:



Student's program output that matches this definition:



### Penalizations and final grade

A test case fails if its output does not match with an expected value. By default, the penalty applied when a test case fails is the “grade\_range/number\_of\_cases”. The penalties of all failed test cases are summed to obtain the overall penalization. The final grade is the maximum mark, less the total penalization. The final grade value never is less than the minimum grade or greater than the maximum grade of the VPL activity.

#### Format:

```
"Grade reduction = [ value | percent% ]"
```

The penalty can be a percentage or a specific value. The final grade will be the maximum grade for the activity minus the overall penalization. If the result value is less than the minimum grade, the minimum is applied.

Example:

```
★ vpl_evaluate.cases
1 ▾ Case = Counting positive numbers
2   Input = -2 1
3   Output = 1
4   Grade reduction = 60%
```

Evaluation report for a wrong output

```
► Proposed grade: 40 / 100
▼ Comments
Test 1: Testing positive numbers
Incorrect program output
--- Input ---
-2 1

--- Program output ---
2

--- Expected output (numbers)---
1
```

## Advanced testing

### Controlling the messages in the output report

BIOTES adds to the report the details of the input, the output expected, and the output found when a test fails. When a test case fails and has a **Fail message** statement, the system shows that message instead of showing the default input/output report.

The **Fail message** statement allows the evaluator to hide the data used in the case. A student knowing the inputs and each output expected might code a solution that passes the tests without resolving the problem. If the fail message statement appears in a test case which fails, the report will only contain the message in this statement.

**Format:**

```
"Fail message = message"
```

Example:

```
★ vpl_evaluate.cases
1 ▾ Case = Counting positive numbers
2   Input = -2 1
3   Output = 1
4   Grade reduction = 60%
5   Fail message = The count is incorrect
```

Evaluation report for a wrong output

► Proposed grade: 40 / 100
▼ Comments
<p><b>Test 1: Counting positive numbers</b> The count is incorrect</p> <p><b>Summary of tests</b></p> <pre>+-----+   1 test run/ 0 tests passed   +-----+</pre>

## Running another program

The evaluator can use another program to test a different feature of the student's program. Among other possibilities, this allows running static/dynamic analysis of the student code, e.g., the evaluator can run `checkstyle`<sup>1</sup> to check the style of the student's java code. The **Program to run** allows, for this test case, replacing the program to run (the student's program) for another.

### Format:

```
"Program to run = path"
```

### Example:

```
★ vpl_evaluate.cases
1 ▾ Case = Testing Java code style
2   Grade reduction = 20
3   Program to run = checkJavaStyle.sh
4   input =
5   output = "OK"
6
7 ▾ Case = Testing x^2 - 3 x + 2 quadratic equation resolution
8   Grade reduction = 60
9   input = 1 -3 2
10  output = 1 2
11  output = 2 1
12
13 ▾ Case = Testing a quadratic equation resolution
14  Fail message = The output was incorrect
```

## Program arguments

This statement allows sending information as command-line arguments to the student program or the **program to run** if set. Notice that this statement is compatible with the input statement.

### Format:

```
"Program arguments = arg1 arg2 ..."
```

### Example 1:

This example shows how to use the **Program to run** and **Program arguments** statements to check if the student's program creates a file with a name passed as a command-line argument.

<sup>1</sup> <http://checkstyle.sourceforge.net/>

Example of using **Program arguments** statement and **Program arguments** statements:

```
vpl_evaluate.cases
1 Case = Testing file creation
2 Grade reduction = 60
3 Program to run = check_file_exists.sh
4 input = file1.txt OK
5 output = "OK"
6
7 Case = Testing file creation
8 Grade reduction = 60
9 Program to run = check_file_exists.sh
10 input = otherFile.dat Good
11 output = "Good"
12
13 Case = Testing file creation
14 Fail message = The file creation fails
15 Grade reduction = 60
16 Program to run = check_file_exists.sh
17 input = stranger.txt WellDone
18 output = "WellDone"
19
```

Code of the **check\_file\_exist.sh** script:

```
check_file_exists.sh
1 #!/bin/bash
2 # Removes $1 file is exists
3 if [ -f "$1" ]
4 then
5     rm $1
6 fi
7 # Runs student's program
8 ./vpl_test $1
9 # Checks if file $1 exists
10 if [ -f "$1" ]
11 then
12     echo "$2"
13 else
14     echo "The file $1 does not exist"
15 fi
16
```

Example 2:

The following example shows how the teacher can use the **Program to run** and **Program arguments** statements to evaluate a SQL query exercise using different data sets.

Example of using **Program to run** and **Program arguments** statements:

```
vpl_evaluate.cases
1 Case = Test query on Data Set 1
2 Grade reduction = 60%
3 Program to run = /usr/bin/sqlite3
4 Program arguments = data.db ".read tables.sql" ".read dataset1.sql" ".read answer.sql"
5 output = "Abel|bulharská
6 Mrkva|slovenská
7 Zelenina|česká"
8 Case = Test query on Data Set 2
9 Grade reduction = 60%
10 Program to run = /usr/bin/sqlite3
11 Program arguments = data.db ".read tables.sql" ".read dataset2.sql" ".read answer.sql"
12 output = "González|Spanish
13 Hernández|Italian"
14 Case = Test query on Data Set 3
15 Grade reduction = 60%
16 Program to run = /usr/bin/sqlite3
17 Program arguments = data.db ".read tables.sql" ".read dataset3.sql" ".read answer.sql"
18 output = "London|English"
```

### Expected exit code

This statement sets the expected exit code of the program case execution. The test case succeeds if the exit code matches. Notice that the test case also succeeds if an output matches.

### Format:

```
"Expected exit code = number"
```

### Example 1:

The following example shows how the evaluator can use the **Program to run** and **Program arguments** statements to evaluate a SQL query exercise using different data sets.

```
vpl_evaluate.cases
1 Case = Test query on Data Set 1
2 Grade reduction = 60%
3 Program to run = /usr/bin/sqlite3
4 Program arguments = data.db ".read tables.sql" ".read dataset1.sql" ".read answer.sql"
5 output = "Abel|bulharská
6 Mrkva|slovenská
7 Zelenina|česká"
8 Case = Test query on Data Set 2
9 Grade reduction = 60%
10 Program to run = /usr/bin/sqlite3
11 Program arguments = data.db ".read tables.sql" ".read dataset2.sql" ".read answer.sql"
12 output = "González|Spanish
13 Hernández|Italian"
14 Case = Test query on Data Set 3
15 Grade reduction = 60%
16 Program to run = /usr/bin/sqlite3
17 Program arguments = data.db ".read tables.sql" ".read dataset3.sql" ".read answer.sql"
18 output = "London|English"
```

The next example shows the possibilities of **Program to run** and **Program arguments** statements to execute different programs. The first case changes the name of a file, the second compiles the file, and the third runs the resulting program.

Example 2:

```
vpl_evaluate.cases
1 ▾ Case = Rename file
2   Program to run = /bin/mv
3   Program arguments = OtherExample.java1 OtherExample.java
4   input =
5   Output = /^$/
6
7 ▾ Case = Compile code
8   Program to run = /usr/bin/javac
9   Program arguments = OtherExample.java
10  input =
11  Output = /^$/
12
13 ▾ Case = Run the compiled program
14  Program to run = /usr/bin/java
15  Program arguments = OtherExample
16  Input =
17  Output = 2
18  Expected exit code = 3
19
```

## Output filtering and formatting

This chapter is intended for an expert audience interested in developing tools that can be integrated with VPL. VPL extracts the text for the assessment report and the proposed grade for an evaluation execution from the raw text output of such execution. To do this, VPL expects that the raw text output contains some labels to identify the relevant texts. These texts can also include some format indicators that VPL uses to format its report.

### Filtering the raw output

VPL processes the raw output of an evaluation execution, selecting the text for the report and the proposed grade. The text for the report is composed of comments which can be of two types: line comments and block comments. A line comment is a text contained in a line starting with the label “Comment :=>”, like:

```
Comment :=>>This text will appear in the report
```

Result:

▼ Comments
This text will appear in the report

A block comment is a text included between a line with the label “<|--” and a line with the label “--|>”. The labelling lines must not include any other text but the labels. A block comment looks like this:

```
<|--
This is a multiline text to appear as comment
in the evaluation report
--|>
```

Result:

▼ Comments
This is a multiline text to appear as comment in the evaluation report

The raw report may contain a combination of line comments, block comments, and other contents.

```
This text will NOT appear in the report
<|--
This is a multiline text to appear as comment
in the evaluation report
--|>
This text will NOT appear in the report
Comment :=>>This is a line comment in the report
```

Contents outside of a line or block comment are removed from the report.

Result:

▼ Comments
This is a multiline text to appear as comment in the evaluation report This is a line comment in the report

The proposed grade is set in a line starting with the label “Grade :=>>”. If more than one of these lines appear, the system uses the last. A report with a proposed grade line looks like this:

```
Grade :=>>10.5
Comment :=>>This text will appear in the report
Grade :=>>8.5
```

Result:

► Proposed grade: 8.5 / 100
▼ Comments
This text will appear in the report

The proposed grade found becomes the final grade if the automatic assessment option is set in the VPL activity configuration.

## Format indicators

The comments in the report of an activity evaluation may contain marks to get the best formatting. The allowed marks are as follows:

- Lines starting with “-” are titles.
- Lines starting with “>” are preformatted text “<pre>”.
- The rest of the lines are regular text and show as content related to the previous title.
- Expressions of the form “filename:number”, when **filename** is the name of one student's source file, generate automatic hyperlinks to the corresponding line of the source file. The text is also added at the corresponding editor gutter line as a tooltip.

You may add a penalty as a negative value between round brackets at the end of each title line. This value represents the penalty attributed to the associated comment and is hidden from the students. When the evaluator pushes the **calculate** button at the manual grade form, the system uses these penalties to reduce the maximum grade and shows the resulting value as the student grade. A reported comment that contains these formats, including a line title with a penalty, looks like:

```
<|--
-This is a major error. (-55)
This text will appear associate to the previous title
> +-----+
> |This is a preformatted text|
> |This is other text      |
> +-----+
Error in Fraction.java:4
--|>
Grade :=>>45
```

Result:

The screenshot displays the VPL evaluation interface. On the left, a code editor shows the file `Fraction.java` with the following code:

```
1 public class Fracction {
2     // ...
3     public static void main (String[] args) {
4         Fraction f1 = New Fraction(5, 6);
5         Fraction f2 = new Fraction(6, 5);
6         Fraction f3 = new Fraction(-21, 16);
7         Fraction f4 = new Fraction(8, 7);
8         Fraction fresult = f1.sum(f2.multiply(f3.sum(f4)));
9         System.out.println("Fraction result " +
10                             fresult.getNumerator() +
11                             "/" +
12                             fresult.getDenominator()
13     );
14 }
15 }
16
```

On the right, the evaluation summary shows a proposed grade of 45 / 100. Below this, the 'Comments' section contains the following text:

**This is a major error.**  
 This text will appear associate to the previous title

The comment is followed by a preformatted text block:

```
+-----+
|This is a preformatted text|
|This is other text      |
+-----+
```

Below the preformatted text, the error location is indicated: Error in [Fraction.java:4](#).

## PRISCILLA System

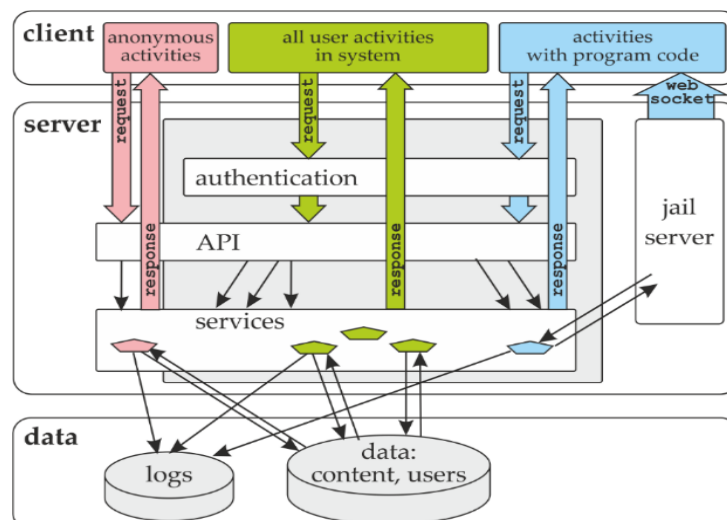
The positive experience with microlearning activities and exercises based on automated source code evaluation in LMS Moodle led to the software architecture proposal and subsequent software implementation of a system called **Priscilla** (PRogressive System for interaCtive (programming) Learning and Learning Assistance).

The front-end part can be implemented as a web, mobile or desktop application. The user's interaction with the application is fluent because the network traffic is very low after the first application launch in a web browser. The front-end part provides the educational content in three forms:

- Micro-content represents the content in the form of text, short source codes, images, etc. This type of activity is designed as an HTML container, and the content is transmitted as a package containing formatted text (headings, text, source code, images, tables, etc.).
- Microlearning activities are interactive objects that require the user to solve simple tasks. A typical example is filling in the correct code result, filling a gap in the code by typing or drag-and-dropping the right parts, reordering shuffled lines of source code, and so on. Interactive activities are combined with content activities (usually 1:1 or in favour of interactive activities) in lessons and chapters. Tasks in interactive activities are focused on the information contained in previous content activities – the content structure is developed concerning microlearning principles.
- Activities aimed at acquiring programming skills are focused on writing, executing, and validating the program code. The student completes the developed programs or writes complete codes in a user-friendly editor adapted to the selected language. After writing the code, the student sends the program to the validation system, which evaluates its correctness. The response may contain compiler errors (syntax errors) or code accuracy, depending on comparing the submitted code results with the expected results.

The communication between the front-end and the backend is provided via web services. This architecture allows the development of various front-end applications: web-client, mobile application, or desktop application. The communication is realised via REST API using application/JSON format.

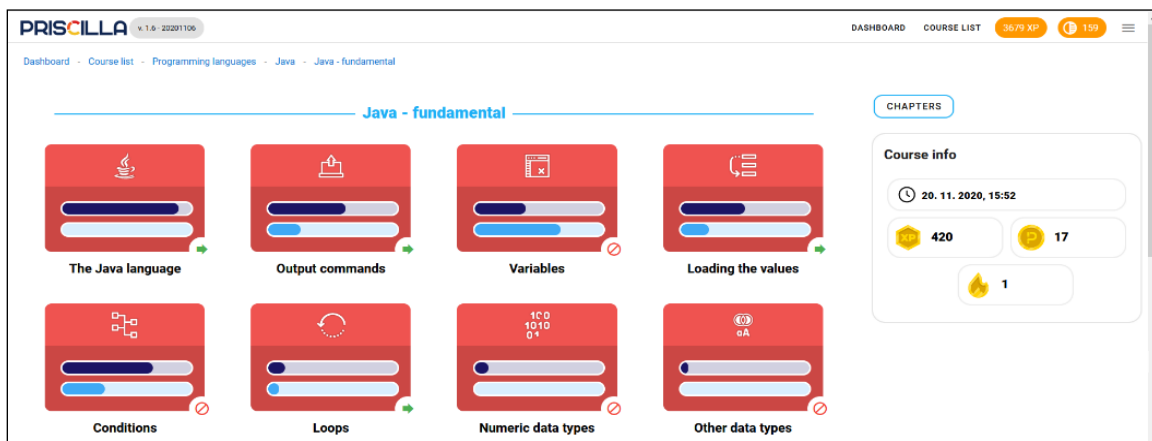
The following diagram presents an overall view of the communication between the individual elements of the system.



## Content structure

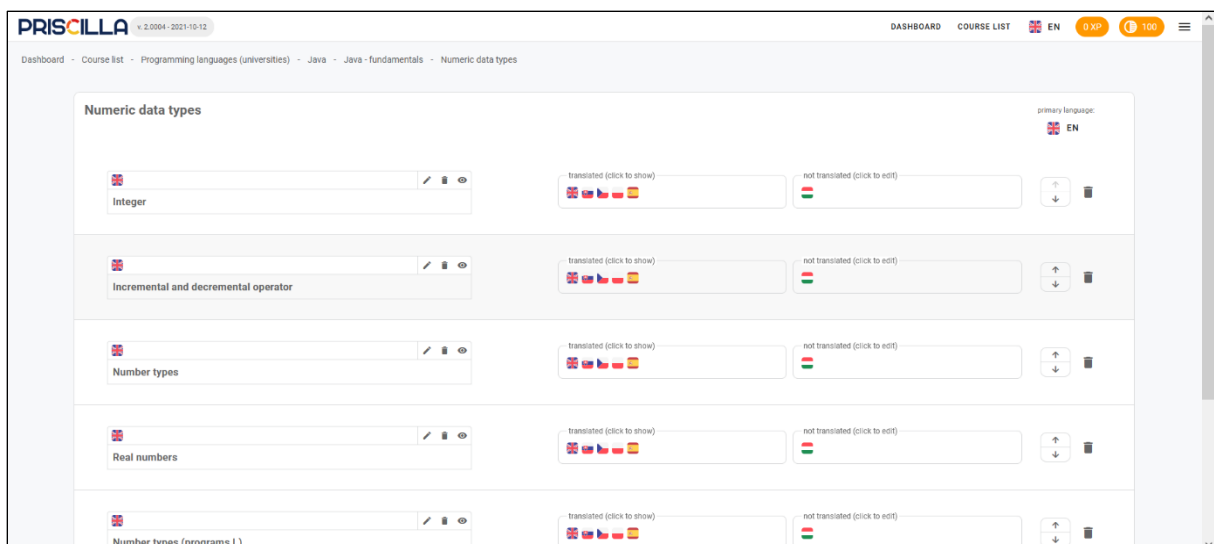
The essential idea in successful introductory programming courses is to leave students some freedom to choose the activities they should complete in the programming course. The programming courses were designed following the classical educational structures, and the order of defined chapters is in line with the didactics of teaching programming. Still, they do not force the student to proceed linearly. Almost every chapter contains a combination of tasks and programs, which students complete based on their preferences. Each task can be repeated as many times as a student needs. Students can return to the place of explanation of the issue, if necessary – the system's goal is not to evaluate but to teach.

The following figure presents chapters as the main parts of the courses.



The course consists of chapters; chapters contain lessons. Lessons are structured in the following views.

Admin view:



Every lesson contains tasks and program assignments.

## Admin view:

## User view:

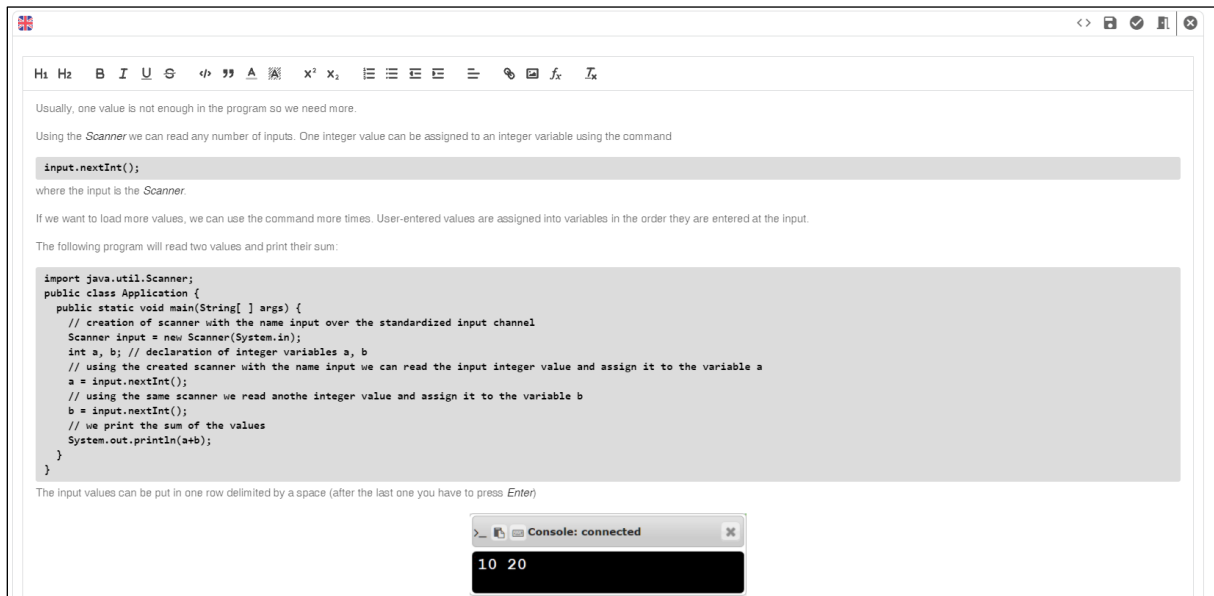
The most important activity is task and programming assignment (content) development. Support for building skills in several ways is based on a combination of different types of tasks. There are available the following task types covering the following activities:

- content microlesson
- short answer
- choice of options
- multiple choice of options
- placing code snippets
- writing commands into code
- rearranging lines of source code
- different types of writing programs based on VPL ideas.

Every question, microlesson or program can be translated into supported language. The list of languages depends on system settings. They are currently English, Spanish, Slovak, Czech, Polish, and Hungarian.

## Content microlesson

The content of microlessons is presented by HTML text with all standard supported objects (images, colours, links, bullets, codes, etc.).



Usually, one value is not enough in the program so we need more.

Using the `Scanner` we can read any number of inputs. One integer value can be assigned to an integer variable using the command

```
input.nextInt();
```

where the input is the `Scanner`.

If we want to load more values, we can use the command more times. User-entered values are assigned into variables in the order they are entered at the input.

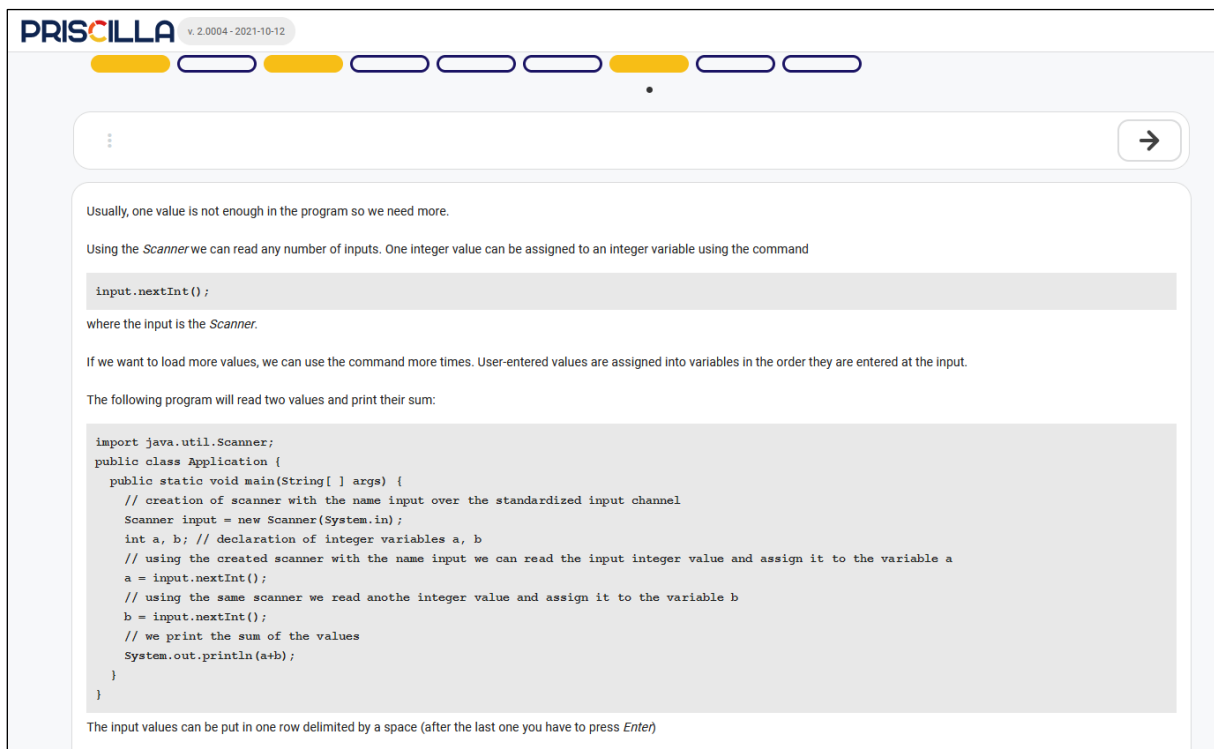
The following program will read two values and print their sum:

```
import java.util.Scanner;
public class Application {
    public static void main(String[] args) {
        // creation of scanner with the name input over the standardized input channel
        Scanner input = new Scanner(System.in);
        int a, b; // declaration of integer variables a, b
        // using the created scanner with the name input we can read the input integer value and assign it to the variable a
        a = input.nextInt();
        // using the same scanner we read another integer value and assign it to the variable b
        b = input.nextInt();
        // we print the sum of the values
        System.out.println(a+b);
    }
}
```

The input values can be put in one row delimited by a space (after the last one you have to press *Enter*)

Console: connected  
10 20

The result of content creator activity looks like this:



PRISILLA v. 2.0004 - 2021-10-12

Usually, one value is not enough in the program so we need more.

Using the `Scanner` we can read any number of inputs. One integer value can be assigned to an integer variable using the command

```
input.nextInt();
```

where the input is the `Scanner`.

If we want to load more values, we can use the command more times. User-entered values are assigned into variables in the order they are entered at the input.

The following program will read two values and print their sum:

```
import java.util.Scanner;
public class Application {
    public static void main(String[] args) {
        // creation of scanner with the name input over the standardized input channel
        Scanner input = new Scanner(System.in);
        int a, b; // declaration of integer variables a, b
        // using the created scanner with the name input we can read the input integer value and assign it to the variable a
        a = input.nextInt();
        // using the same scanner we read another integer value and assign it to the variable b
        b = input.nextInt();
        // we print the sum of the values
        System.out.println(a+b);
    }
}
```

The input values can be put in one row delimited by a space (after the last one you have to press *Enter*)

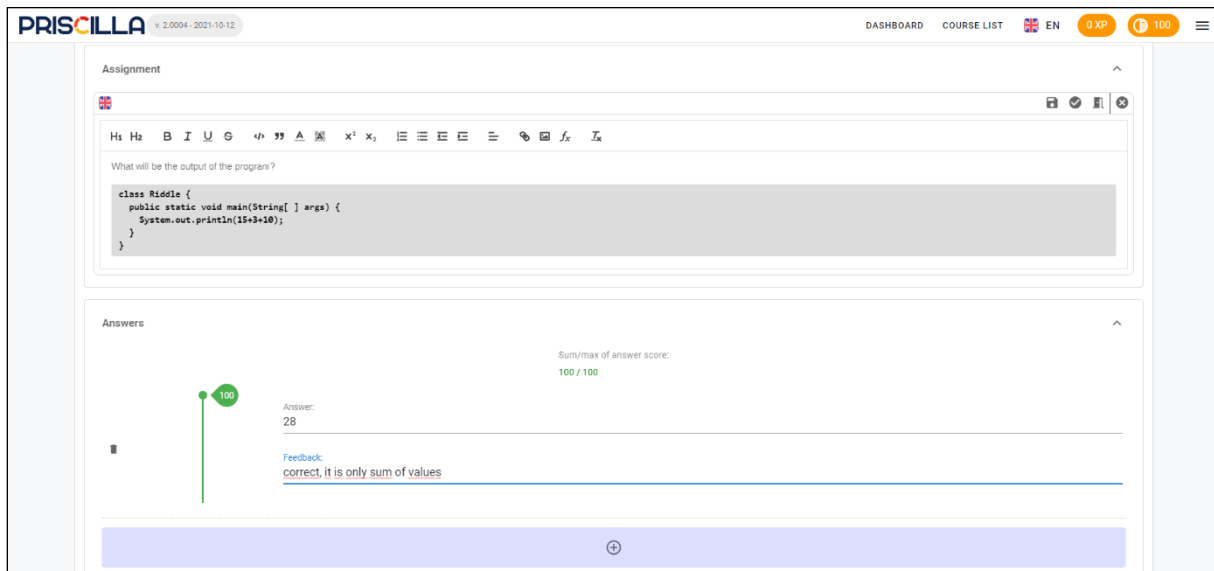
Console: connected  
10 20

## Short answer

The short answer is a type of question usable for obtaining simple (and short) answers, or results of programs or numerical values obtained, e.g. through the execution of expressions.

The content creator can define more correct answers, or some answers can be evaluated as not quite right. The per cent of correctness is defined together with expected answers and feedback.

Definition:

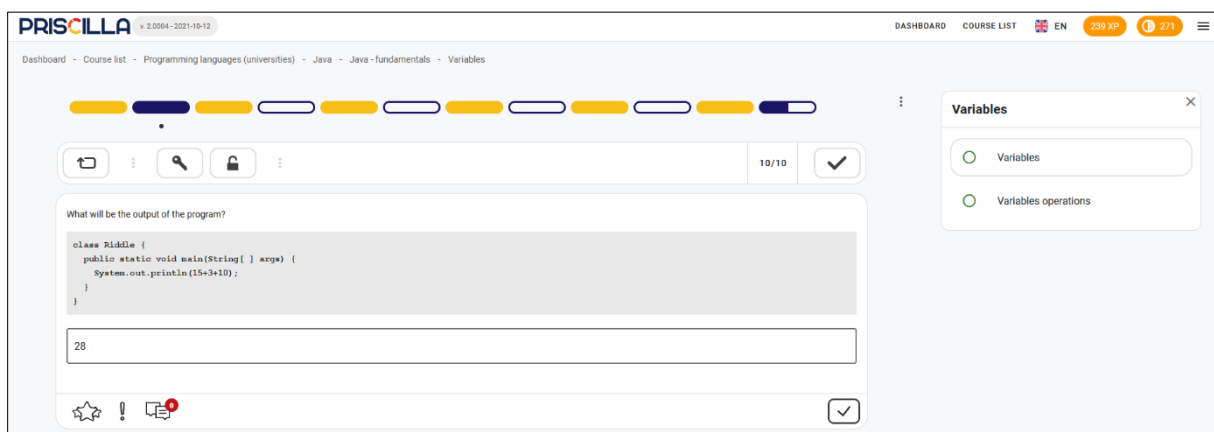


The screenshot shows the PRISILLA interface for an assignment. The top bar includes the PRISILLA logo, version 2.0004-2021-10-12, and navigation links for DASHBOARD, COURSE LIST, EN, 0 XP, and 100. The main content area is titled 'Assignment' and contains a text editor with a Java code snippet:

```
class Riddle {
    public static void main(String[] args) {
        System.out.println(15+3+10);
    }
}
```

Below the code, the question asks: 'What will be the output of the program?'. The 'Answers' section shows a score of 100/100 and the answer '28'. The feedback provided is: 'correct, it is only sum of values'.

User view:

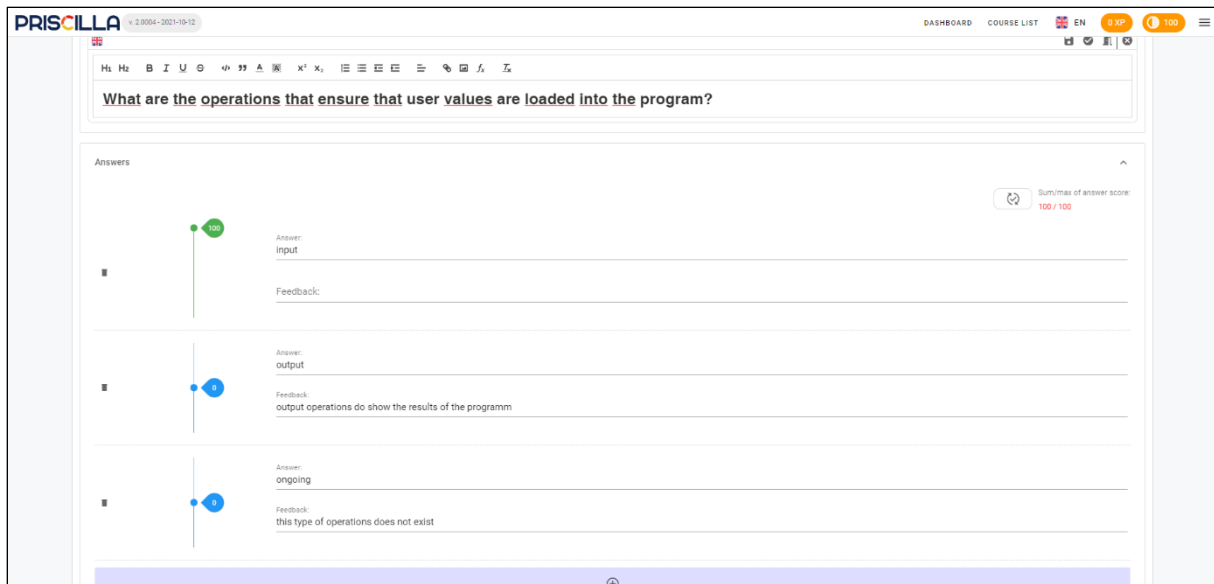


The screenshot shows the user view of the PRISILLA interface. The top bar includes the PRISILLA logo, version 2.0004-2021-10-12, and navigation links for DASHBOARD, COURSE LIST, EN, 239 XP, and 2/1. The main content area is titled 'Assignment' and contains a text editor with the same Java code snippet as the previous screenshot. Below the code, the question asks: 'What will be the output of the program?'. The answer '28' is entered in the input field. The 'Variables' sidebar on the right shows two options: 'Variables' and 'Variables operations'.

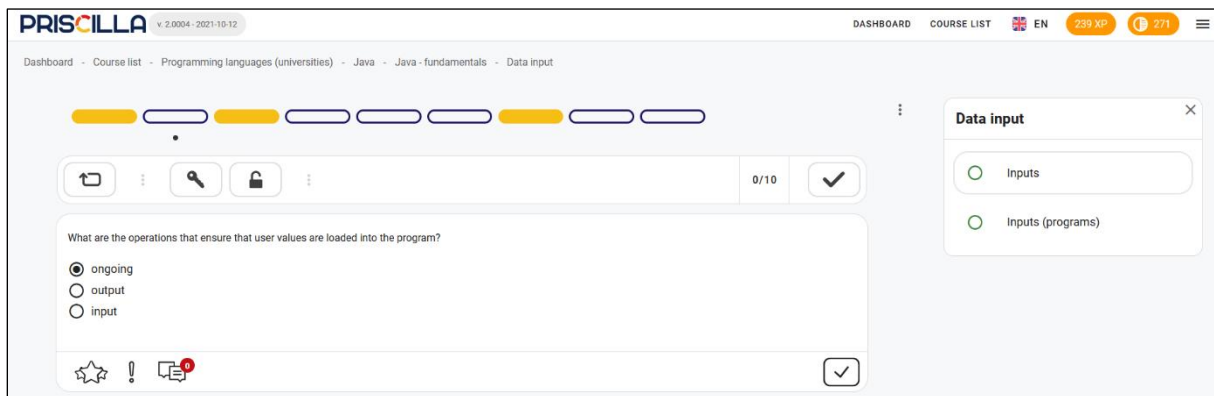
## Choice of options

This type of question can be used if only one of the pre-listed answers to the question is correct. Because the system is set up so that it is possible to evaluate even partially correct answers, it is possible to set a percentage of correctness for each option. The choice of methodology and rules of assessment depends only on the teacher. However, by default, for this type of question, one answer is expected to have a 100% rating and the others zero.

Definition:



User view:



## Multiple choice of options

This type of question can be used if one or more of the above answers are correct. The starting point for this type of question is to offer a different approach to accurate calculations. A content developer can penalize a bad answer or ignore it (set to zero), or some questions can be evaluated with a better score. Some wrong answers can be penalized by a worse score. The function of the system can recalculate partial scores for its harmonization.

## Definition:

PRISCILLA v. 2.0004 - 2021-10-12

DASHBOARD COURSE LIST EN 0 XP 100

Which statements are true?

Answers

Summan of answer score: 0 / 90

Answer: loop with condition on end will be executed at least once

Feedback:

Answer: loop with condition on beginning does not have to be run any times

Feedback:

Answer: loop with condition on end does not have to be run any times

Feedback:

## User view:

PRISCILLA v. 2.0004 - 2021-10-12

DASHBOARD COURSE LIST EN 239 XP 271

Dashboard - Course list - Programming languages (universities) - Java - Java - fundamentals - Loops

Which statements are true?

☐ loop with condition on end will be executed at least once

☐ loop with condition on end does not have to be run any times

☐ loop with condition on beginning will be executed at least once

☐ loop with condition on beginning does not have to be run any times

☐ loop with known number of repeats will be executed at least once

0/10

Loops

- Basic commands
- More about Loops
- For cycle (programs)
- Loops with conditions
- While loops (programs)

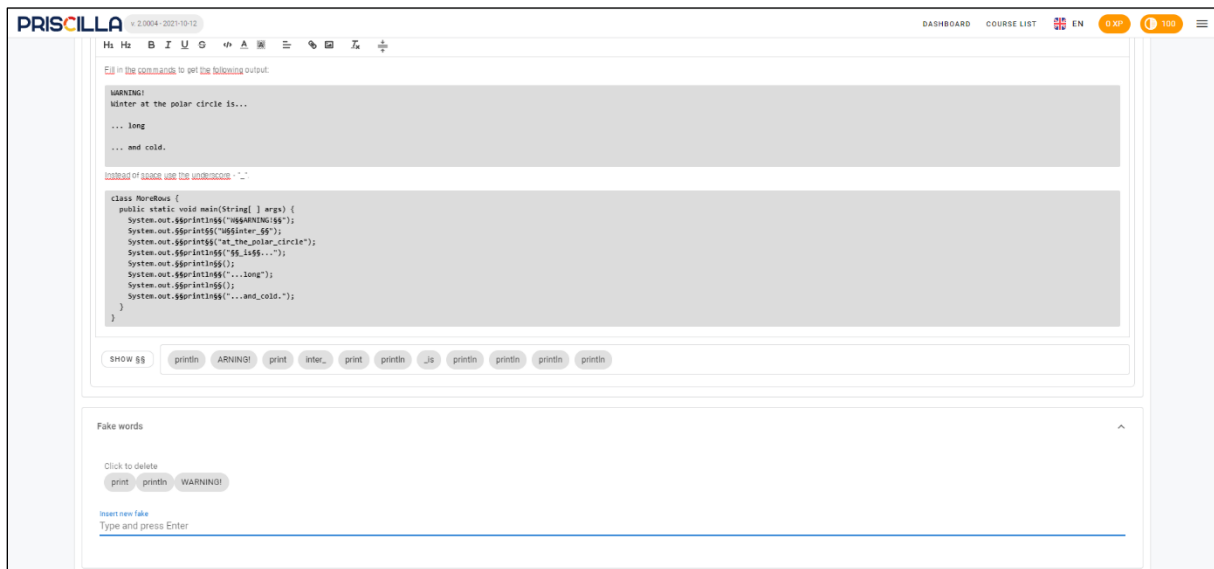
## Placing code snippets

This type of question allows the user to select code snippets and insert them into the program in the correct place. The part of the code that will be hidden from the user is inserted between the pairs of `$$` marks.

Solving the task can be complicated by adding incorrect options, which will appear to the user in a shuffled list along with the correct ones.

The task does not always have to be focused only on adding code to the program; it can be used to arbitrarily insert parts of the text into the content. Due to the complicated structuring of complex text, bullets and part of HTML objects are not supported.

## Definition:



## User view:

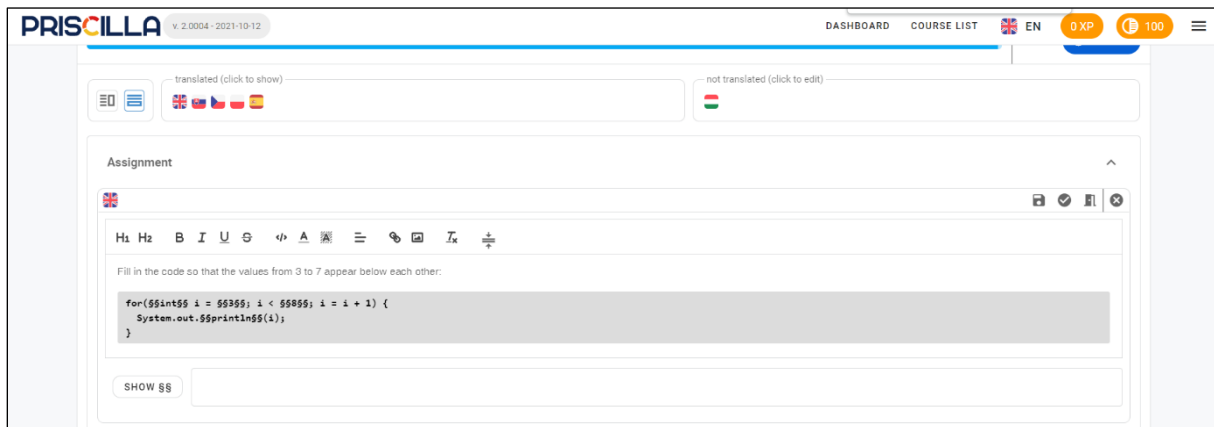


## Writing commands into code

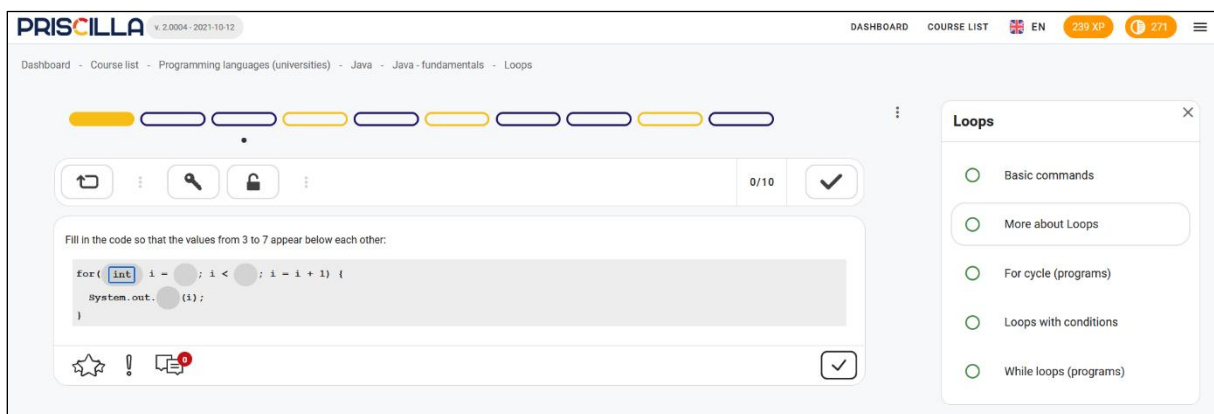
This type of question is similar to the previous one, except that we do not select the inserted code from the list, but we have to enter it manually.

Again, the hidden text inserted between the pairs of \$\$ marks and the type of task can be used to insert any content (not just parts of the program).

Definition:



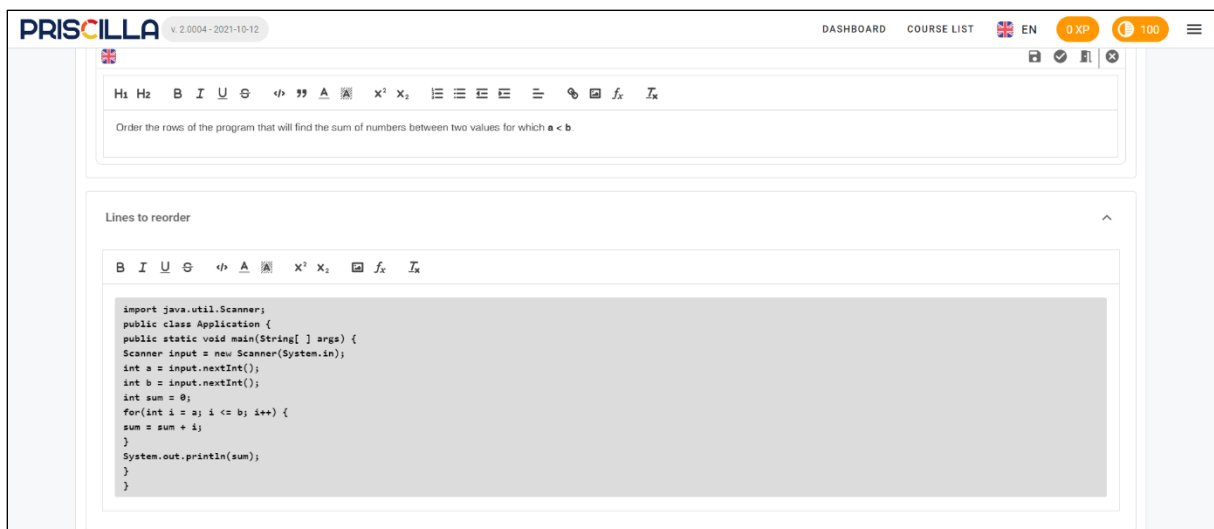
User view:



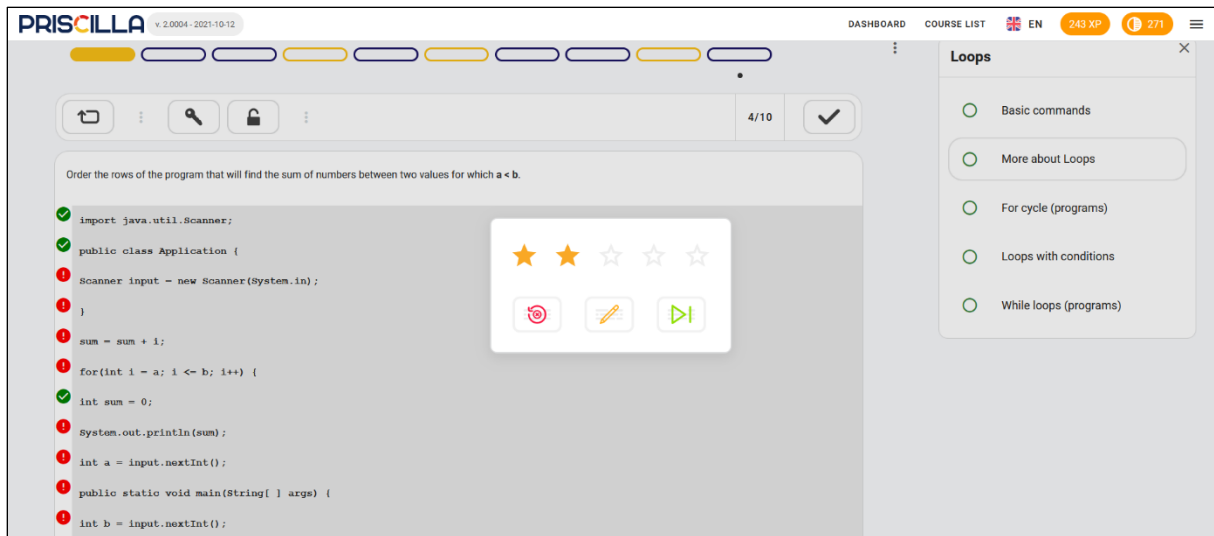
## Rearranging lines (of source code)

This type of question requires a row reordering. Lines can be source code or any text defined as paragraphs.

Definition:



User view:



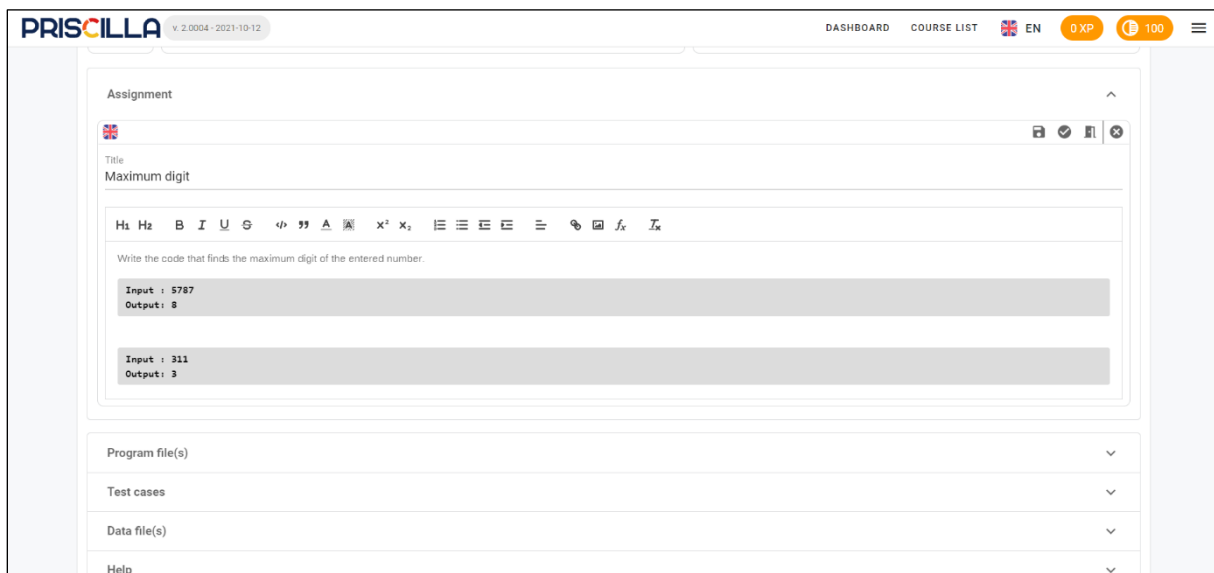
## Program assignment

### BIOTES assignments

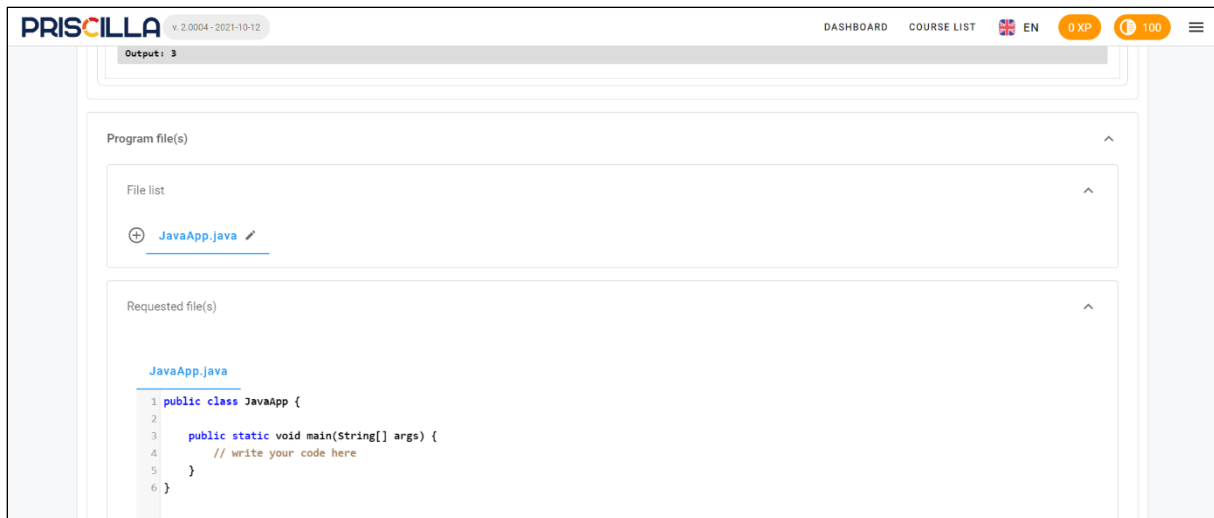
Programming tasks are based on the ideas of a Virtual Programming Lab for Moodle. The evaluation uses VPL infrastructure and the functionality of a jail server.

This section describes the creation of tasks based on BIOTES, while the functionalities described in the previous chapter (definition of inputs, various types of outputs, penalties, etc.) also work in the PRISCILLA system.

The content developer defines assignments, including examples of inputs and outputs displayed to the user directly in the assignment.



For the task, it is necessary to define a file name or a list of files that will be sent to the server evaluating the source code. It is advisable to set the file extensions to valid extensions – corresponding to the programming language.



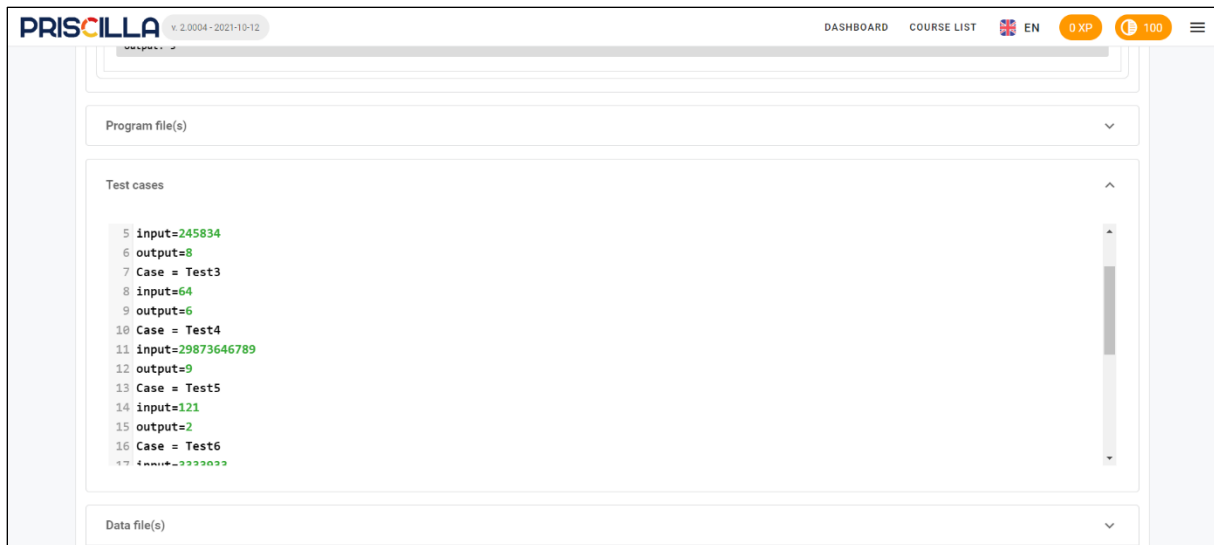
The **Requested file(s)** section defines the pre-prepared source code that will be displayed to the user together with the assignment.

Note that in some languages (such as Java), the file name must be the same as the class name defined in that file. For this reason, too, it is not advisable to leave the file displayed to the user blank.

The **Requested file(s)** also includes the **Author solution**, which will be displayed to students as a help or a ready-made solution. The functionality of the specified solutions can be verified directly in the system by buttons above the content of the author solution.

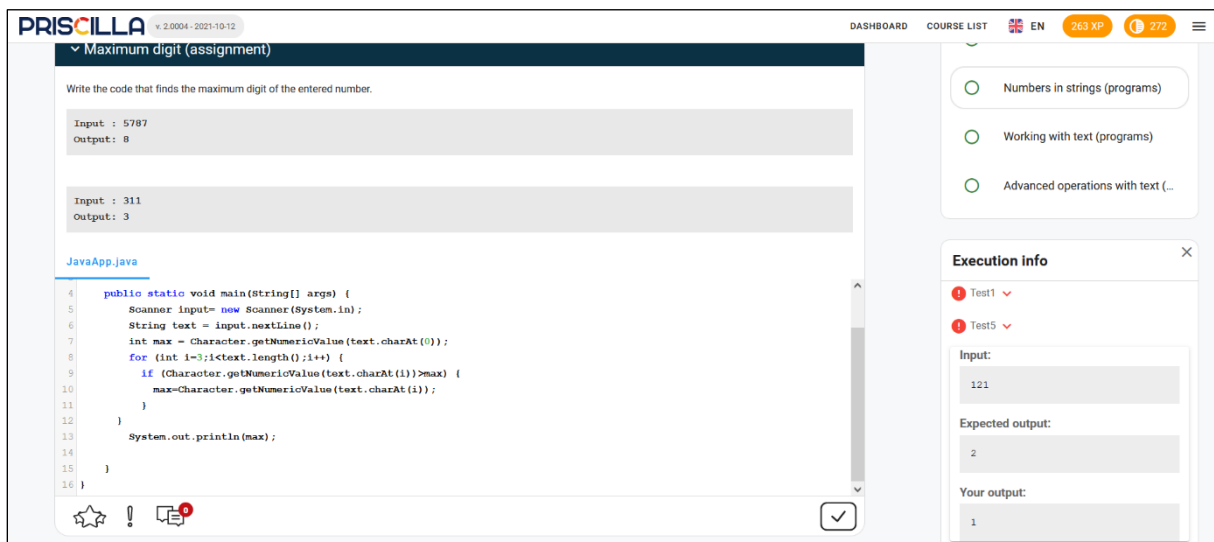


**Test cases** are defined and created in accordance with the rules specified for VPL in LMS Moodle.



The use of other object types such as files and configuration files will be explained below as part of the VPL.

User view:



The figure shows unsatisfactory results with the expected and obtained value of the program. These results are served as an aid to the user in identifying inputs with bad results.

## Statically evaluated code

A special category of source code is represented by languages such as HTML or CSS. It is impossible to evaluate the result in obtaining output based on executing a sequence of commands. In this case, text analysis was chosen as the basic evaluation approach.

For the text form of the document (parts of the code, web page, etc.), is it possible to define a series of rules that verify the fulfilment of the occurrence and/or position of specific text strings. The following rules were defined and applied:

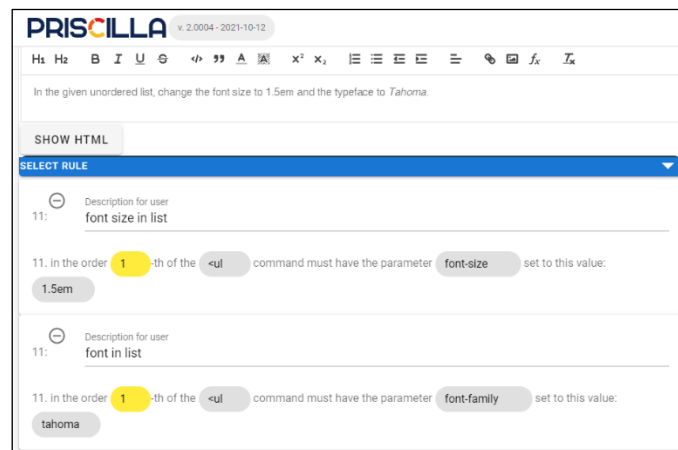
- The text must contain \$\$\$ minimum ### times
- The text can contain \$\$\$ maximum ### times
- The text must not contain \$\$\$

- The text \$\$\$\$ must follow \$\$\$
- The text \$\$\$ must be before \$\$\$
- The text \$\$\$\$ must not follow \$\$\$\$
- The text must contain the pairs in this order: \$\$\$\$ and \$\$\$\$
- The text must contain the triads in the following order: \$\$\$\$ and \$\$\$\$ and \$\$\$\$
- The text must contain the fours in the following order: \$\$\$\$ and \$\$\$\$ and \$\$\$\$ and \$\$\$\$
- The text must contain ##### times the text \$\$\$\$ between \$\$\$\$ and \$\$\$\$
- In the order #####-th of the \$\$\$\$ command must have the parameter \$\$\$\$ set to this value: \$\$\$\$
- Some of the occurrences of \$\$\$\$ must have the parameter \$\$\$\$ set to the value: \$\$\$\$

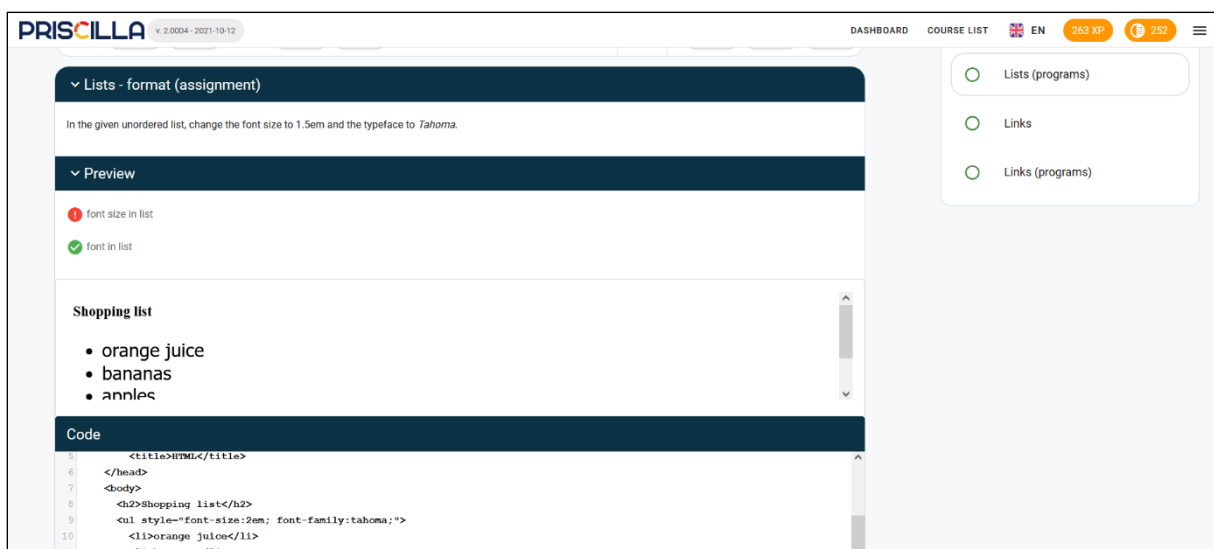
The \$\$\$ can be understood as a text string and ### as a number.

Following realisation can be presented as an example:

Definition:



User view:



## Advanced VPL Features

VPL assignments also provide additional settings to support many variations when checking programs. Teachers can customize the running and assessing process and more.

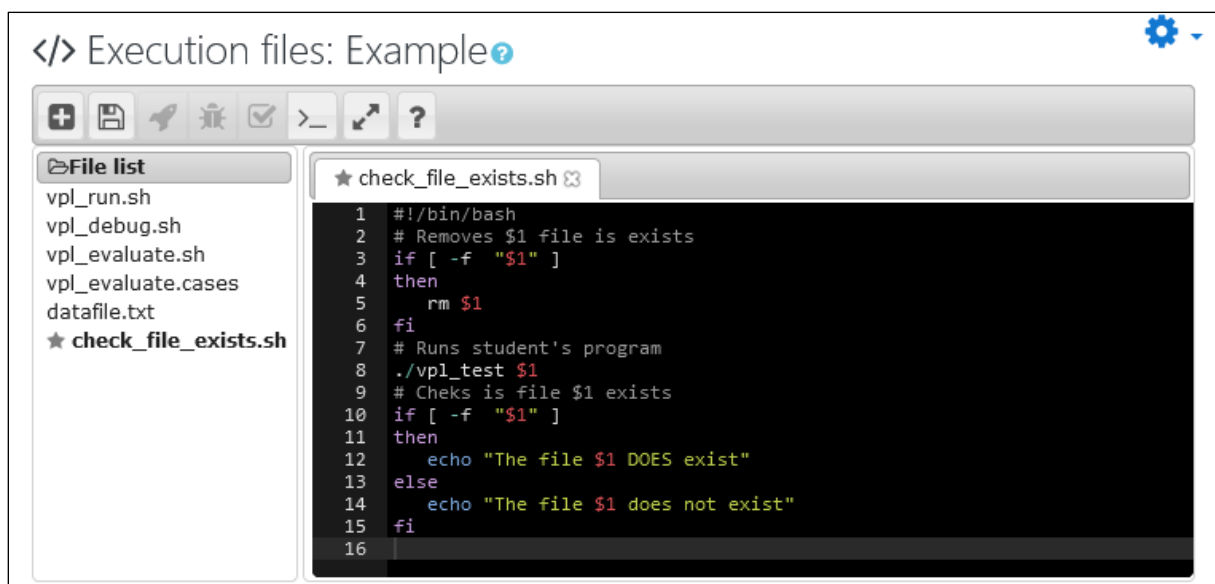
### Execution files

Execution files are a set of files that participate in the run, debug and evaluate tasks. The list includes scripting files, program test files, data files, etc. These files will go with the student's submitted files to run in an execution server. If a student's file has the same name as an execution file, the **execution file** is used. Some of the files have a predetermined purpose based on their name: `vpl_run.sh`, `vpl_debug.sh`, `vpl_evaluate.sh`, and `vpl_evaluate.cases`.

The scripts **`vpl_run.sh`**, **`vpl_debug.sh`**, and **`vpl_evaluate.sh`** if set, replaces the corresponding default action. These scripts carry out the compilation or preparation phase of the action. They aim to generate a file named `vpl_execution` or `vpl_wexecution`. These files must be a binary executable or a script beginning with `"#!/bin/sh "`. The system launches **`vpl_execution`** in a textual terminal and launches **`vpl_wexecution`** in a graphic terminal. The non-generation of one of these files impedes running the selected action. The **`vpl_debug.sh`** can not generate a `vpl_wexecution` file.

The file **`vpl_evaluate.cases`** defines the test cases used by the VPL input/output evaluation program.

Execution files setting has as interface an IDE that allows defining the files. This interface provides the predefined files named above and the run, debug and evaluate buttons allowing to test the effect of the **Execution files** in the last submission of the teacher.



### Files included and excluded

For security reason the run action removes **`vpl_debug.sh`**, **`vpl_evaluate.sh`** or **`vpl_evaluate.cases`** from the task. The debug action removes **`vpl_evaluate.sh`** or **`vpl_evaluate.cases`**.

All execution tasks includes two auxiliar scripts: **`common_script.sh`** and **`vpl_environment.sh`**.

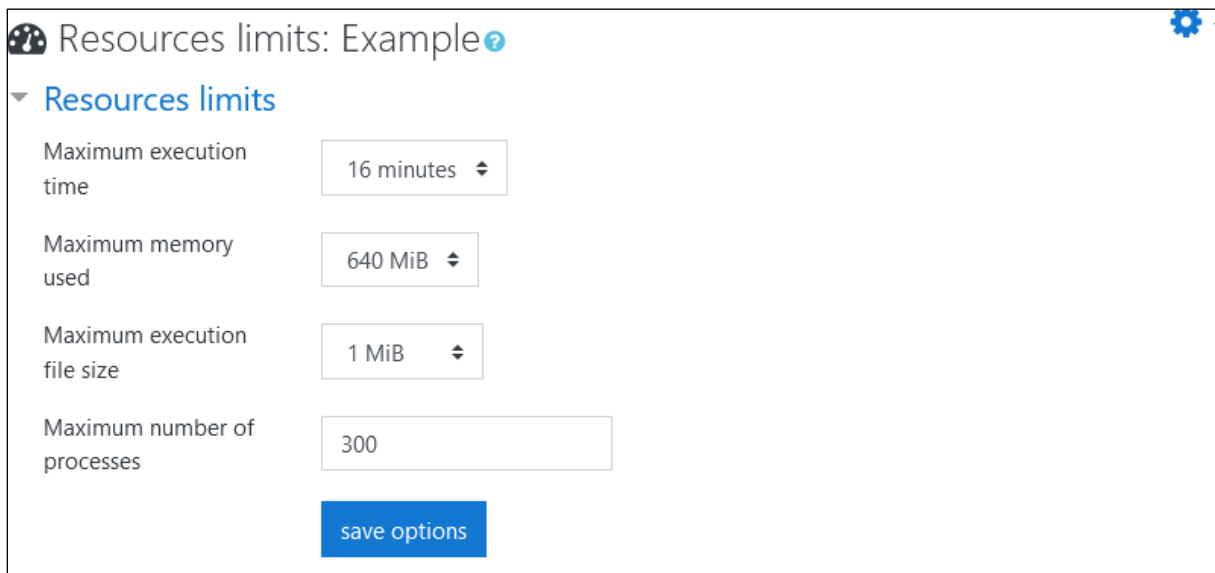
The **`common_script.sh`** script defines auxiliary functions for the other scripts. The **`vpl_environment.sh`** define shell environment variables with information about the task. The run, debug or evaluate script can use the following variables:

- LANG: used language.
- LC\_ALL: same value as LANG.
- VPL\_MAXTIME: maximum execution time in seconds.
- VPL\_FILEBASEURL: URL to access the files of the course.
- VPL\_SUBFILE#: each name of the files submitted by the student. # Ranges from 0 to the number of submitted files.
- VPL\_SUBFILES: list of all submitted files separated by space.
- VPL\_VARIATION: the identification of the variation assigned or empty.
- VPL\_VARIATION + id: where id is the variation order starting with 0, and the value identifies the variation assigned. These vars have a sense when using :ref: *the based on feature*.
- If the action requested is evaluation, then the following vars are added too.
- VPL\_MAXTIME: max time of execution in seconds.
- VPL\_MAXMEMORY: max memory usable.
- VPL\_MAXFILESIZE: max file size in bytes that can be created.
- VPL\_MAXPROCESSES: max number of processes that can be run simultaneously.
- VPL\_FILEBASEURL: URL to the course files.
- VPL\_GRADEMIN: Min grade for this activity.
- VPL\_GRADEMAX: Max grade for this activity.

## Execution resources limits

You can set limits for the execution time, the memory used, the execution files sizes, and the number of processes to be executed simultaneously.

These limits are used when running the scripting files **vpl\_run.sh**, **vpl\_debug.sh** and **vpl\_evaluate.sh** and the file **vpl\_execution** or **vpl\_wexecution** built by them.



**Resources limits: Example**

▼ **Resources limits**

Maximum execution time	16 minutes
Maximum memory used	640 MiB
Maximum execution file size	1 MiB
Maximum number of processes	300




**save options**

### How to measure the required resources for execution in an execution(jail) server?

In general, it is correct to leave the defaults value of resource limits. Suppose you want to restrict or extend the resources used in a particular activity. In that case, it is recommended to test a standard solution of the activity, varying the resource from higher to lower to find the correct value.

## Files to keep when running

For security reasons, after running the scripting files **vpl\_run.sh**, **vpl\_debug.sh** or **vpl\_evaluate.sh**, and before running the file **vpl\_execution** or **vpl\_wexecution** built by them, the unneeded **execution files** are removed. If some of these files need to be when running the student's program, the teacher must mark these files here, e. g. auxiliary libs or data files.

 Files to keep when running: Example 


▼ Files to keep when running




☐ vpl\_run.sh
 ☐ vpl\_debug.sh
 ☐ vpl\_evaluate.sh
 ☐ vpl\_evaluate.cases
 ☒ datafile.txt

save options

## Variations

A set of variations can be defined for an activity. These variations are randomly assigned to the students.

Here you can indicate if this activity has variations, put a title for the set of variations, and to add the desired variations.

 Variations: Example 


▼ Variation options

Use variations

Yes ▾

Variation title

Variations title

Save

Cancel













---

▼ Variation 1

Identification

V1

Description

This content is **added to the description** if variation 1 assigned.

Each variation has an identification code and a description. The students see the description of their assigned variation in the task description. The identification code of the student's assigned variation is set in `vpl_enviroment.sh` file to participate in the execution process.

Description
Submission
Edit
Submission view

## Example

**Requested files:** example\_class.cpp, main.cpp ([Download](#))

**Maximum number of files:** 120


**Maximum upload file size:** 40 KiB

**Type of work:** Individual work

**Variations title**

This content is **added to the description** if variation 2 assigned.

This is the description of the activity.



## Check execution servers

This report checks and shows the status of each execution server available for this activity. The report also shows the tasks running or just finished in the current course.

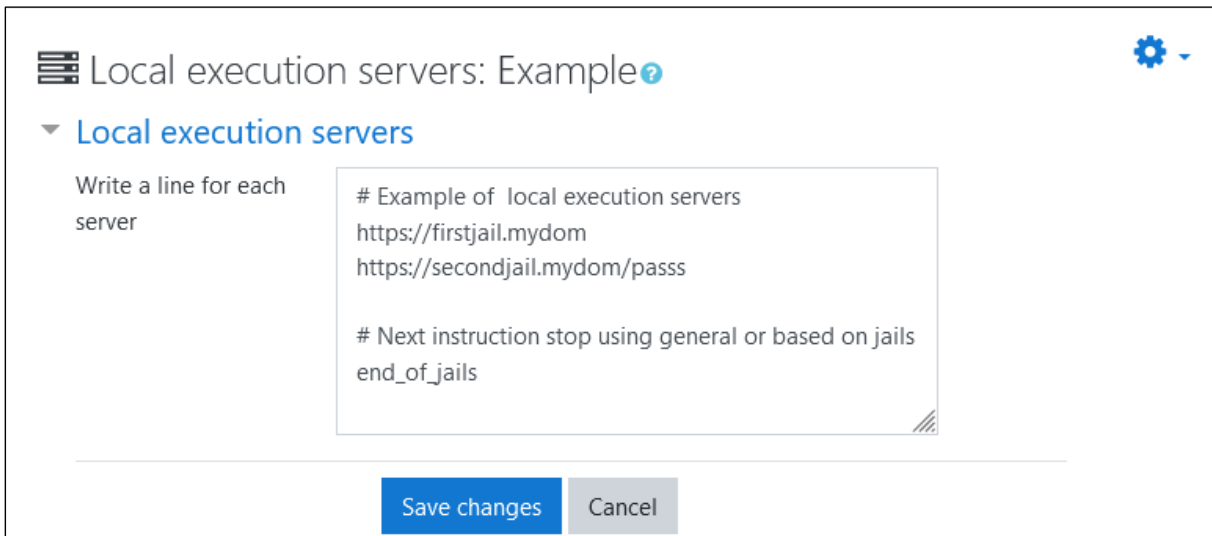
#	Server	Current status	Last error info	Last error date	Errors
1	https://demojail.dis.ulpgc.es	ready			0
2	https://otherjail.mydom [WARNING: not accessible from the internet]	request failed: Could not resolve host: otherjail.mydom	request failed: Could not resolve host: otherjail.mydom	Monday, 16 August 2021, 3:50 PM	2

#	User	Activity	Server	Starting from	Status
1	<a href="#">Admin User</a>	Example	https://demojail.dis.ulpgc.es	Monday, 16 August 2021, 3:50 PM	Running

## Local execution servers

VPL allows the setting of several execution servers. For each task requested, the system selects one of these execution servers to carry out the task. The local execution server allows adding new execution servers for the current activity. This option will allow having more execution power and fail tolerance for specific activities. It is also possible to set the only servers that will participate in the current activity by ending the list of local servers with a line containing “**end\_of\_jails**”. It allows having specific execution servers for specific activities that may have particular software needs.



Local execution servers: Example ?

Local execution servers

Write a line for each server

```
# Example of local execution servers
https://firstjail.mydom
https://secondjail.mydom/passs

# Next instruction stop using general or based on jails
end_of_jails
```

Save changes Cancel

## The based-on feature

This powerful feature allows activities to inherit the options and files of other VPL activities. This feature enables the development of generic activities that can be used as a basis for others. In many cases, the generic activities are not used as activities to be used by students; their purpose is to establish a common framework for other activities. The following items are inherited from the selected activity as based on:

- **Description.** The **based on** description is inherited. If the current activity contains a description, it is appended to the **based on** activity.
- **Resource limits.** The **based on** resource limits are inherited. Any resource limit set in current activity replaces inherited value.
- **Upload file size limits.** Current activity can replace inherited value.
- **Run/debug script.** Inherits the selection in **Execution options** of the run script and debug script if not set in the current activity.
- **Execution files.** All **execution files** are inherited. Current activity **execution files** add files and replace the ones inherited with the same name. Exceptionally, predefined scripting files (**vpl\_run.sh**, **vpl\_debug.sh**, and **vpl\_evaluate.sh**) are append to the inherited.
- **Local servers.** Inherits local servers and append local servers in current activity.
- **Variations.** Inherited and current variations are used to generate multi-variation activities.

### ▼ Execution options

Based on

Select



## Adding support for a new programming language

Adding support for a new programming language in VPL requires two things: the programming language compiler/interpreter ready to be used in an execution server and the scripts that allow preparing the student's files to run. The first requirement can be done following the instructions of the selected programming language tool. The second is done by customizing the **vpl\_run.sh** script for running and the **vpl\_debug.sh** for debugging. If planning using the Automated program assessment, customize **vpl\_evaluate.sh** is unneeded.

A simple way to start your own script is to take the default script of another similar programming language. The default scripts are located in `"/mod/vpl/jail/default_scripts/"` of the VPL plugin source code.

The purpose of the script is to compile or prepare the implementation of the action requested run/debug/evaluate. If the script succeeds, it will generate an executable file called **vpl\_execution** to be run on a textual terminal or **vpl\_wexecution** for execution in a graphics terminal.

Suppose you have a predefined activity customized to run a new programming language. All the new activities configured as **based on** the predefined one will also run the new programming language.

## Customizing Automatic Program Assessment

VPL is a flexible tool such that you can customize the automatic evaluation process. Following the **Execution files**, indications you can run your own tools to evaluate the student's code.

The customized evaluation must generate an output that follows the format below.

### Evaluation output format

When an automatic evaluation is done, the system process the execution output (standard output) to obtain the comments (feedback) and proposed grade.

The comment (feedback) format is as follows:

A comment (feedback) line would be a line starting with "Comment:==>". A block comment would be contained between a line containing only "<|--" and another with "--|>". The proposed grade is taken from the last line that begins with "Grade:==>".

If the automatic assessment is set, the proposed grade becomes the final going to the Moodle grade book.

### Formatting the comments (feedback)

The comments in the activity assessment have their own format:

- Lines beginning with "--" are titles.
- Lines starting with "> " are preformatted text "<pre>".
- The rest of the lines are considered content related to the previous title.
- Expressions of the form filename:number generate a hyperlink to the corresponding line of the file.
- At the end of each title line, optionally, you can add a negative discount. This value represents the discount attributed to the associated comment. This discount is never displayed to the students. This discount is used for the **calculate** button.

Example:

```
- Error: infinite loop (-10)
```

### Details of running a task

This section will try to explain what is happening backstage of a running task.

Connections when running task:

1. User clicks run button. Browser send request by === AJAX (json) === > to the VPL plugin in the Moodle server.
2. VPL plugin in Moodle Server prepare task and send data by === http/https (XMLRPC) === > to a selected execution server.
3. Execution Server starts the task and returns to VPL plugin in Moodle server the task identification.

4. VPL plugin in Moodle server returns to the browser the task identification.
5. The browser monitorize the task by connecting by `=== ws/wss === >` to the execution Server.
6. The browser may connect with the running program by connecting by `=== ws/wss === >` to the execution Server.

Process of a student's program execution (summarized). The following steps are performed:

1. The system takes the files submitted by the student
2. The system takes the files set by the teacher in **Execution files**. These files replace files of the student with the same name.
3. Depending on the action (run, debug, or evaluate), the system takes the customized or default script by detecting the programming language used based on the extension of file names.
4. In the case of evaluation, if there is no custom script, the VPL program for automatic assessment is also added. This program is based on the input and output of the program and requires that you specify the test cases in the file **vpl\_evaluate.cases**.
5. These collected files are sent to an execution server.
6. The plugin informs the browser that the execution has begun.
7. If the request is an evaluation, when the task is finished, the evaluation result is retrieved from the execution server.

## VPL Architecture

This chapter details the VPL internal architecture and the API of the execution servers. The VPL architecture is complex since the system requires the coordination of the three main software components:

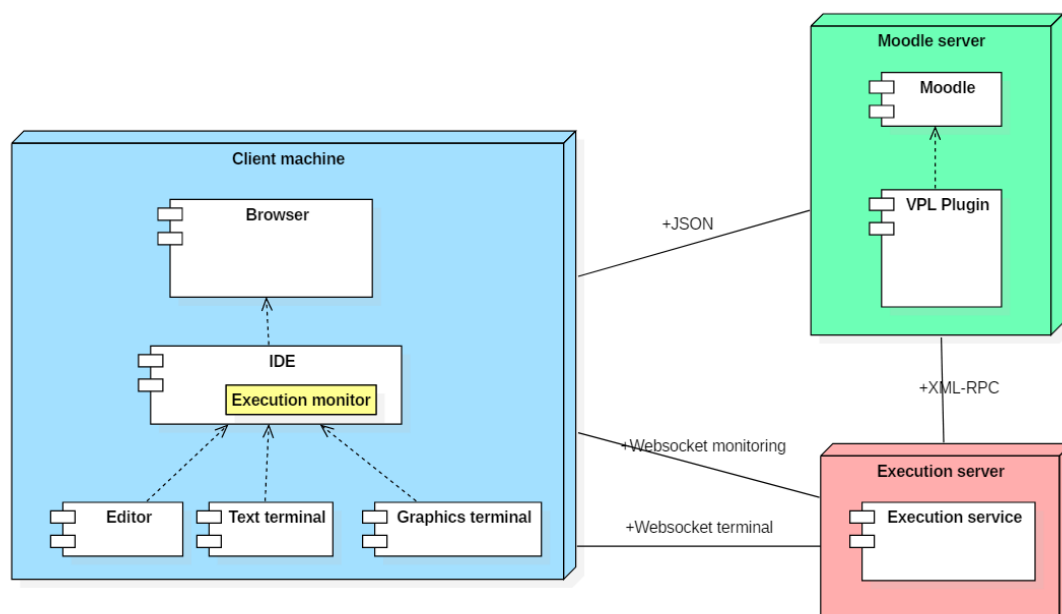
- the VPL Moodle plugin,
- the execution system
- and the integrated development environment (IDE).

The Moodle plugin is the core of the VPL system. It supports the persistent storage of information, integration with Moodle, and control of the execution system.

The IDE is composed of an execution monitor agent, a web code editor, a text terminal and a graphic terminal. It provides a modern development environment for multiple programming languages, with storage, execution, and evaluation provided by the Moodle plugin.

The execution system uses one or more execution servers providing a service for remote execution (execution service). The execution service is a component fully developed for VPL, which runs on a Linux operating system and executes programs in a safe and controlled environment, attending to the requests of the Moodle server and interacting with the IDE.

Virtual Lab Architecture is presented in the following figure.



The system uses three types of connections and data formats between their components:

- The client (Browser) connects with the VPL Moodle plugin using JSON over HTTP/HTTPS. The client sends requests to load and save files from the Moodle server, runs, debugs, and evaluates students' code, and retrieves evaluation results.
- The VPL Moodle plugin connects with the execution server using XML-RPC over HTTP/HTTPS. The plugin sends XML-RPC commands to the execution server to run tasks, get tasks results, stop tasks, etc.

- The client (Browser) connects with the execution server using raw/custom format over WebSocket. The client uses these types of connections to monitor execution tasks (this is required) and get remote access to the running program using a text or graphic terminal.

## Connections accepted by execution server

The execution service in VPL supports XML-RPC requests and WebSocket over HTTP/HTTPS. The WebSocket connections allow the bidirectional and direct connection between the browser and any other machine that supports the protocol. The WebSocket protocol does not require a dedicated port since it allows transforming an HTTP connection into WebSocket (through an initial negotiation using the HTTP protocol headers).

Suppose a browser uses secure connections to communicate with the Moodle server. In that case, the JavaScript code that runs on the resulting web pages must also use secure connections, even though the connections are WebSocket. This browser security requirement raises the need to support WebSocket Secure (WSS), the WebSocket encrypted protocol. It is achieved by adding support for HTTPS, which provides support for WSS as a side effect.

The execution service can execute both non-interactive evaluation tasks and interactive execution tasks (run or debug). The execution of those tasks is divided into several actions: execution request, retrieval of results request, and execution stop request. All these requests use HTTP or HTTPS in XML-RPC format.

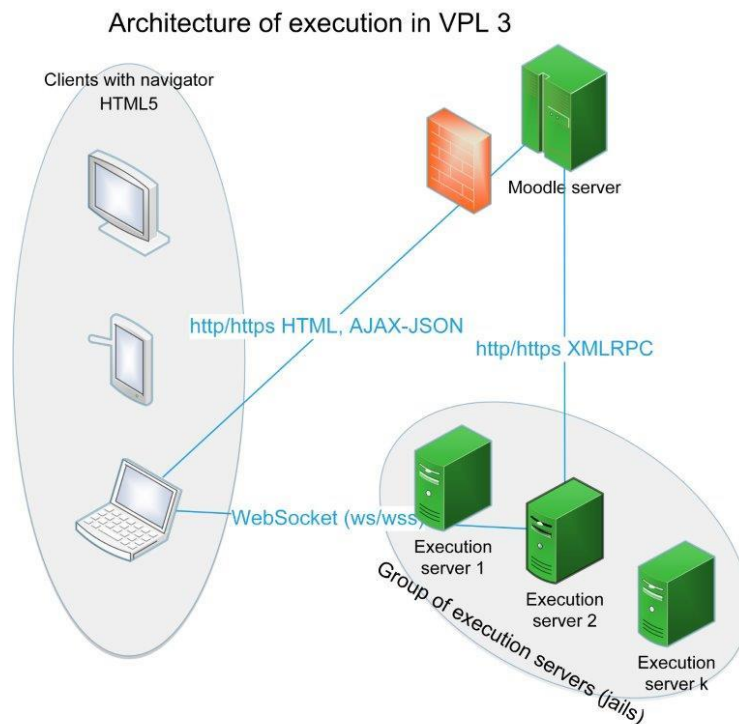
On the other hand, the monitoring and control of the state of the execution task and the interaction in case of interactive execution, either in a text or graphic terminal, are done by means of each WebSocket connection from the browser. This monitoring and control connect the browser with the execution server to inform the user of the state of the task and allow him to stop it. It should be borne in mind that the execution tasks are divided into two parts: compilation or preparation of the execution of the code and another execution.

The execution part varies depending on whether the requested task is run, debug, or evaluated. In the first two cases, the execution is interactive. In the third one, the execution takes place without the user's intervention and without the user having direct access to the evaluation result, which is processed to obtain a proposed grade and the annotations that support that grade.

In the run and debug tasks, the only difference is the script that is used. If the first compilation phase ends well, it goes to the interactive execution phase. To launch the execution, the browser is informed of the type of execution required through the monitoring and control channel (text or graphic terminal). So, it establishes the appropriate WebSocket connection.

This connection establishment triggers the execution process itself. After interaction with a text terminal, the inputs and outputs of the program are redirected to the established WebSocket connection. In the case of execution in a graphic terminal, a VNC server is started where the student's program will be executed.

The execution service in a graphic terminal establishes a raw connection with the VNC server to channel the RFB data stream between the server and the VNC client in the browser via the WebSocket connection.



## Security aspects

Security is an essential aspect of the system, so developers consider it carefully in each design and implementation phase of the proposed solution. Moodle takes care of the user authentication; however, VPL is in charge of the connections with the execution server. Execution servers can control the access from Moodle using two security elements:

The first one allows setting a key as part of the URL to access the service. The second one allows limiting by IPs or networks the machines that can request execution tasks.

This restriction only affects the start of the task. Once the request is accepted, the system uses other security mechanisms. A request generates specific credentials for task monitoring, interactive execution, and task management.

The monitoring and execution credentials are for single-use by the browser. The Moodle server only uses the management credential.

**Execution servers count failure requests statistics to mitigate denial-of-service (DoS) attacks by holding in quarantine for a while the IPs with a high rate of failures. XML-RPC commands**

Following commands from XML-RPC communication are supported:

- **available:** This is an optional command that can be used to know if the server has enough resources to execute a request.
- **request:** This is the command to request execution. The response to this request contains tickets required to follow the execution.
- **getresult:** This command returns the result of an evaluation.
- **running:** This command returns if execution is still running or has finished yet.
- **stop:** This command stops a running execution.

All VPL XML-RPC requests and responses use one parameter with a value of type struct. The XML-RPC struct is used to represent objects attributes or associative arrays in a programming language.

## The "available" method

### Description

This method requests the server to inform if it is available to run a task that requires a certain amount of memory. This request can be sent before a **request** command to know if the server has enough resources to execute a future **request**. This command is not a necessity to send a **request**. The **available** command returns a status value, indicating if the server is available and a detail of the server execution limits.

### "available" parameters

Attribute	type	Opt.	Description
<b>maxmemory</b>	int	No	Memory in bytes is required by the task to test if the system can support it

For PHP the code to generate the request may be:

```
$data = new stdClass();
$data->maxmemory = $maxmemory;
$encoding = array ( 'encoding' => 'UTF-8' )
$requestready = xmlrpc_encode_request( 'available', $data, $encoding);
```

### Example of XML-RPC method available

```
<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
<methodName>available</methodName>
<params>
  <param>
    <value>
      <struct>
        <member>
          <name>maxmemory</name>
          <value>
            <int>128000000</int>
          </value>
        </member>
      </struct>
    </value>
  </param>
</params>
</methodCall>
```

### Response to "available"

The response is an object with the following attributes:

Attribute	type	Description
<b>status</b>	string	"ready" for accepting the request "offline" for going offline

		"busy" for too busy for accepting the request
<b>load</b>	int	Number of the task currently running
<b>maxtime</b>	int	Limit of execution time in seconds defined at execution server configuration
<b>maxfilesize</b>	int	Limit of each file size in bytes defined at execution server configuration
<b>maxmemory</b>	int	Limit of memory in bytes used by a task defined at execution server configuration
<b>maxprocesses</b>	int	Limit of number of processes running in a task defined at execution server configuration
<b>secureport</b>	int	Reports the secure port used by the execution server

For PHP, the code to decode the response may be as

```
$response = xmlrpc_decode( $rawresponse, "UTF-8" );
if (is_array( $response )) {
    if (xmlrpc_is_fault( $response )) {
        $error = 'xmlrpc is fault: ' . s( $response ["faultString"] );
    } else {
        return $response;
    }
} else {
    $error = 'http error ' . s( strip_tags( $rawresponse ) );
}
return false;
```

### Example of XML-RPC available response

```
<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
  <params>
    <param>
      <struct>
        <member><name>status</name>
          <value><string>ready</string></value>
        </member>
        <member><name>load</name>
          <value><int>0</int></value>
        </member>
        <member><name>maxtime</name>
          <value><int>600</int></value>
        </member>
        <member><name>maxfilesize</name>
          <value><int>67108864</int></value>
        </member>
        <member><name>maxmemory</name>
          <value><int>2097152000</int></value>
        </member>
        <member><name>maxprocesses</name>
          <value><int>500</int></value>
        </member>
        <member><name>secureport</name>
          <value><int>443</int></value>
        </member>
      </struct>
    </param>
  </params>
</methodResponse>
```

```

    </struct>
  </param>
</params>
</methodResponse>

```

## The "request" method

### Description

This method requests the server to run a task. The request contains all the information necessary for the execution of the task. The server responds to the request before starting the task execution to avoid latencies. The response contains tickets to control the task execution.

The task execution has two phases: compilation and execution. The compilation phase has the basic function of generating a valid program and can be used to do another task as static analysis of code. The execution phase is interactive when running or debugging student code and is batch if evaluating. A monitor process supervises the execution of each task.

For more details about the monitor connection, see section **Task monitoring**.

The request may contain the following data:

- Files which will be sent to the execution server. The files contain the file name, including the path relative to the user's home directory and the file contents. The files can include the student's files, execution scripts and teacher's files. The files may be text in UTF-8 or binary encoded in base64.
- The list of files that the system must remove after the compilation phase and before the execution. By default, the system removes all teacher's files (advanced options->execution files in VPL).
- Limits of resources to use during the compilation and execution.
- The name of the script to start the compilation phase.
- If the task is interactive or batch.
- The language used by the user.

### "request" parameters

Attribute	type	Opt.	Description
<b>files</b>	struct	No	Each member of this struct represents a file. The name of each member is the name of a file, and its value is a string with the file content. The file content may be initially encoded in base64; see file encoding.
<b>filestodelete</b>	struct	No	Each member of this struct represents a file to be deleted. The name of each member is the name of a file to be deleted, and the value is not used. These files are removed after the compilation phase and before the execution phase
<b>fileencoding</b>	struct	Yes	Each member of this struct represents information about a file encoding. The name of each member is the name of a file, and the value is an integer. If the integer associated with the name is 0, the file content is text in UTF-8. If the

			value is 1 the file is encoded in base64.
<b>maxtime</b>	int	No	The maximum number of seconds that the compilation or execution is required to finish. If the <b>maxtime</b> is reached, the task is stopped.
<b>maxfilesize</b>	int	No	The maximum size a file created by the task can reach. This limit is recommended to be high. The OS control this limit. Some tasks may have an odd behaviour if they reach the limit.
<b>maxmemory</b>	int	No	The maximum number of bytes of memory that a task can use. The system stops the task if it uses more memory than the <b>maxmemory</b> .
<b>maxprocesses</b>	int	No	The maximum number of processes that the task can use. The OS controls this limit. Some tasks may have an odd behaviour if they reach the limit.
<b>lang</b>	string	No	A string that represents the LANG of the user. The task execution may use this environment variable. A common value is <b>en</b> .
<b>execute</b>	string	No	The name of the script to start the task. The VPL plugin uses <b>vpl_run.sh</b> , <b>vpl_debug.sh</b> or <b>vpl_evaluate.sh</b>
<b>interactive</b>	int	No	The 1 if the task is interactive or 0 if it is batch. VPL uses batch tasks when evaluating with the <b>vpl_evaluate.sh</b> script.

Example of XML-RPC method available

```
<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
<methodName>request</methodName>
<params>
  <param>
    <value>
      <struct>
        <member>
          <name>files</name>
          <value>
            <struct>
              <member>
                <name>vpl_run.sh</name>
                <value>
                  <string>...</string>
                </value>
              </member>
              <member>
                <name>eval/Main.java</name>
                <value>
                  <string>...</string>
                </value>
              </member>
              ...
            </struct>
          </value>
        </member>
        <member>
```

```

<name>filestodelete</name>
<value>
  <struct>
    <member>
      <name>vpl_run.sh</name>
      <value>
        <int>1</int>
      </value>
    </member>
    ...
  </struct>
</value>
</member>
<member>
  <name>maxtime</name>
  <value>
    <int>240</int>
  </value>
</member>
<member>
  <name>maxfilesize</name>
  <value>
    <int>67108864</int>
  </value>
</member>
<member>
  <name>maxmemory</name>
  <value>
    <int>469762048</int>
  </value>
</member>
...
<member>
  <name>execute</name>
  <value>
    <string>vpl_evaluate.sh</string>
  </value>
</member>
<member>
  <name>interactive</name>
  <value>
    <int>0</int>
  </value>
</member>
<member>
  <name>fileencoding</name>
  <value>
    <struct>
      <member>
        <name>vpl_run.sh</name>
        <value>
          <int>0</int>
        </value>
      </member>
      ...
    </struct>
  </value>
</member>
</struct>

```

```

    </value>
  </param>
</params>
</methodCall>

```

## Response to "request"

A response format is an object with the following attributes

Attribute	type	Description
<b>adminticket</b>	string	This ticket grants access to the XML-RPC methods related to the started task ( <b>getresult</b> , <b>running</b> and <b>stop</b> )
<b>monitorticket</b>	string	This ticket is needed to establish the WebSocket monitoring connection. This is a single-use ticket if the connection is lost cannot be re-established. See the section <b>Task monitoring</b> .
<b>executionticket</b>	string	This ticket is used in the WebSocket execution connection. This is a single-use ticket if the connection is lost cannot be re-established.
<b>port</b>	int	Server port used to accept HTTP requests.
<b>secureport</b>	int	Server port used to accept HTTPS requests.

## Example of XML-RPC request response

```

<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
  <params>
    <param>
      <struct>
        <member><name>adminticket</name>
          <value><string>112316513634721</string></value>
        </member>
        <member><name>monitorticket</name>
          <value><string>90173797645932</string></value>
        </member>
        <member><name>executionticket</name>
          <value><string>700988013259542</string></value>
        </member>
        <member><name>port</name>
          <value><int>80</int></value>
        </member>
        <member><name>secureport</name>
          <value><int>443</int></value>
        </member>
      </struct>
    </param>
  </params>
</methodResponse>

```

## The "getresult" method

### Description

This method requests the server to retrieve the result of a batch task. This request can be made when the monitor informs that the task has ended. For more details about the monitor connection, see section **Task monitoring**. The request only contains the adminticket

#### "getresult" parameters

Attribute	type	Opt.	Description
<b>adminticket</b>	string	No	The admin ticket returned in response to the <b>request</b> method.

#### Response to "getresult"

A response format is an object with the following attributes

Attribute	type	Description
<b>compilation</b>	string	Content the output of the compilation, standard output and error.
<b>execution</b>	string	Content the output of the execution, standard output and error.
<b>executed</b>	int	1 if the compilation generated a valid result and 0 if not. A valid result is a file named <b>vpl_execution</b> or <b>vpl_wexecution</b> .
<b>interactive</b>	int	1 if the execution was interactive and 0 if not.

The constant **JAIL\_RESULT\_MAX\_SIZE** limits the compilation and execution size. The default value is 32Kb.

### The "running" method

#### Description

This method requests the server information to know if a task is still running. Notice that this information can also be known through the control connection. The request only contains the **adminticket**

#### "running" parameters

Attribute	type	Opt.	Description
<b>adminticket</b>	string	No	The admin ticket returned in response to the <b>request</b> method.

#### Response to "running"

A response format is an object with the following attributes

Attribute	type	Description
<b>running</b>	int	1 if the task is still running, 0 if not

### The "stop" method

#### Description

This method requests to stop a running task. Notice that this can also be done through the control connection. The request only contains the adminticket

#### Method "stop" parameters

Attribute	type	Opt.	Description
<b>adminticket</b>	string	No	The admin ticket returned in response to the <b>request</b> method.

#### Response to "running"

A response format is an object with the following attributes

Attribute	type	Description
<b>stop</b>	int	Always set to 1

### Task monitoring

Once the server accepts a task execution request, the monitoring connection launches the creation of a process that monitors the evolution of the task. If a monitoring connection is not established after 5 seconds of the task request, the system stops the task. This timeout value **JAIL\_MONITORSTART\_TIMEOUT** is set in the **jail\_limits.h** source file of the execution daemon. The monitoring process stops the task also if the execution time limit or the memory limit are reached. The monitoring connection establishes a channel of communication with an external agent. If the external agent sends something to the monitoring process or the connection is lost or closed, the monitoring process stops the task. The monitoring process informs the agent about the state of the task, and commonly the agent reacts to the state changes. The external agent in VPL runs in the browser interacting with the end-user, the VPL Moodle plugin and the execution server.

The monitoring connection is a WebSocket connection established using a URL with the form:

```
[http|https]://ExecutionServerName/monitorTicket/monitor
```

A monitoring connection URL for an HTTPS connection to the server **demojail.dis.ulpgc.es** with a monitor ticket value 98723498124984 would be:

<https://demojail.dis.ulpgc.es/98723498124984/monitor>

The monitoring process sends messages to the external agent in a simple format "type:data".

Type of message	Description
<b>message:[text]</b> [text] is a text of any size.	This message, received from the monitoring process, is textual and commonly used to inform the end-user about the stage of the execution task and the time spent at that stage. VPL uses this information to show the state of compilation and execution to the user. The agent can ignore this information, does not need to take any action on it.
<b>compilation:[text]</b> [text] is a text of any size.	When the compilation phase ends, and the task is interactive, the monitoring process sends the compilation output to the agent using this format.

	<p>VPL uses this information to show the output of the compilation phase (commonly warnings and errors) to the end-user.</p> <p>The agent can ignore this information, does not need to take any action on it.</p>
<p><b>run:terminal</b> or <b>run:vnc:[password]</b></p> <p>The VNC client will use [password] to connect to the server.</p>	<p>If the compilation of an interactive task succeeds, the monitoring process sends a message to the agent requesting to start a WebSocket execution connection. This connection triggers the execution of the <b>vpl_execute</b> or <b>vpl_wexecute</b> file of the task at the execution server, starting the execution phase.</p> <p>run:terminal -&gt; a text terminal is expected at the end point of the connection.</p> <p>run:vnc:password -&gt; a VNC client is expected at the end point of the connection.</p> <p>The agent can delegate or start by itself a connection with the execution server with a URL of the form [http https]://ExecutionServerName/executionTicket/execution.</p>
<b>retrieve:</b>	<p>If the compilation of a non-interactive task succeeds, the monitoring process triggers the execution of the <b>vpl_execute</b> file of the task at the execution server, starting the execution phase. After the correct end of the execution phase, the monitoring process sends this message to the agent to inform that the evaluation has ended and the output is ready to be retrieved from the execution server.</p> <p>The reception of this message triggers at the VPL agent a request to the VPL Moodle plugin to send an XML-RPC <b>getresult</b> command to the execution server. VPL Moodle plugin saves the response of the command at the Moodle server (the compilation and execution result).</p>
<b>close:</b>	<p>The reception of this message informs the agent that the task has ended. The monitoring process will close the connection in a short time after it sends this message.</p>

## Retrieving the VPL activities definition to use it on another system

A simple way to export and use the VPL activities in another system may be saving for each activity:

1. The **description** in HTML format
2. The **requested files** with their content
3. The XML-RPC code generated for the evaluation action (method **request**) is generated using the **requested files** as the student's files to evaluate. This XML code will be used to request evaluations to the execution systems easily.
  - The new system can then show the description of the activity to the users.
  - It can offer the initial content of the requested files.
  - The system can also evaluate the code introduced by the users by replacing the new code into the saved XML and sending it to the execution server. Notice that the response of the evaluation (method **getresult**) must be properly formatted (see previous documentation **Filtering and Formatting VPL output**)

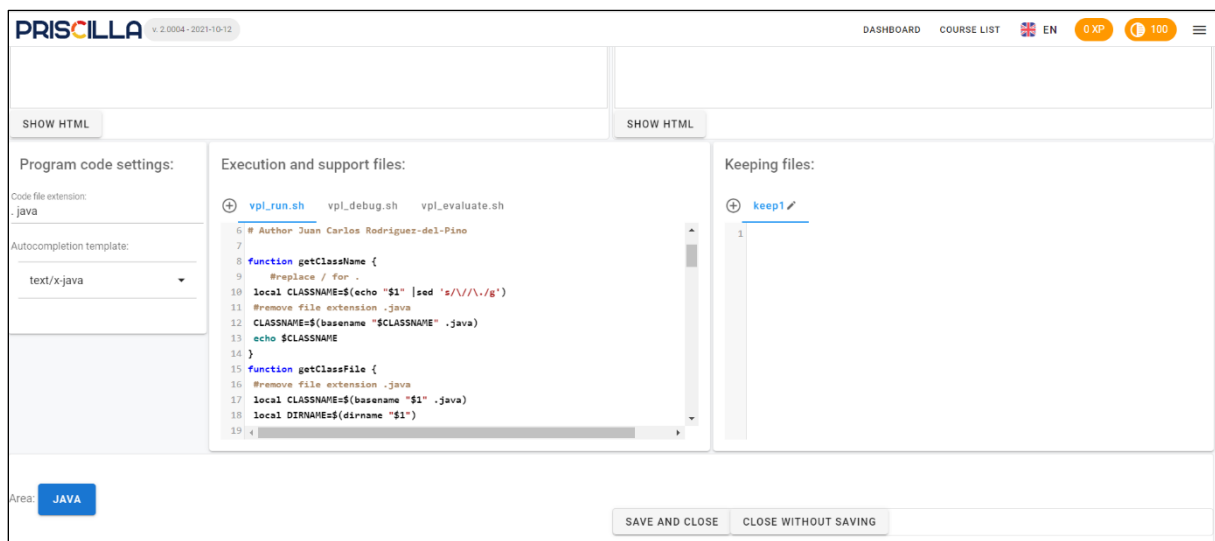


## VPL Settings in PRISCILLA

### Course settings

To run each program via VPL clones, it is necessary to appropriately define files that allow the preparation (compilation) of source code files and subsequent code execution. Along with files that execute programs, source code files, test cases, and possibly other files that may contain processed data or startup configuration elements go to the server.

Priscilla allows you to define these files at the course level or separately in each assignment as an advanced set of it. If the configuration files **vpl\_run.sh**, **vpl\_debug.sh**, **vpl\_evaluate.sh** that are part of the tasks contain text. They will replace the contents of parent files with the same names defined at the course level. The definition of files in the course settings is as follows:



There is also a standard file extension with source code and syntax highlighting + autocompletion template to provide a popup with commands in the editor.

For the following languages, we list the source code needed to run user code:

- Java
- Python
- C
- PHP

In the case of specific requests to run code in a given language, we will notify you.

### Java

#### Code file extension: java

When running the program, the source code file name must be the same as the class name defined. The running class must contain the public **main()** method, from which program execution starts.

```
public class JavaApp {
    public static void main(String[] args) {
```

```

        // write your code here

    }
}

```

## vpl\_run.sh

```

#!/bin/bash
# This file is part of VPL for Moodle - http://vpl.dis.ulpgc.es/
# Script for running Java language
# Copyright (C) 2015 onwards Juan Carlos Rodriguez-del-Pino
# License http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or later
# Author Juan Carlos Rodriguez-del-Pino

function getClassname {
    #replace / for .
    local CLASSNAME=$(echo "$1" | sed 's/\//\./g')
    #remove file extension .java
    CLASSNAME=$(basename "$CLASSNAME" .java)
    echo $CLASSNAME
}

function getClassFile {
    #remove file extension .java
    local CLASSNAME=$(basename "$1" .java)
    local DIRNAME=$(dirname "$1")
    echo "$DIRNAME/$CLASSNAME.class"
}

function hasMain {
    local FILE=$(getClassFile "$1")
    cat -v $FILE | grep -E "\^A\^@\^Dmain\^A\^@\^V\([L]java/lang/String;\)" &>
/dev/null
}

# @vpl_script_description Using default javac, run JUnit if detected
# load common script and check programs
. common_script.sh

check_program javac
check_program java
if [ "$1" == "version" ] ; then
    echo "#!/bin/bash" > vpl_execution
    echo "javac -version" >> vpl_execution
    chmod +x vpl_execution
    exit
fi
JUNIT4=/usr/share/java/junit4.jar
if [ -f $JUNIT4 ] ; then
    export CLASSPATH=$CLASSPATH:$JUNIT4
fi
get_source_files java
# compile all .java files

javac -Xlint:deprecation $2 $SOURCE_FILES
if [ "$?" -ne "0" ] ; then
    echo "Not compiled"
    exit 0
fi
# Search main procedure class
MAINCLASS=

```

```

for FILENAME in $VPL_SUBFILES
do
    hasMain "$FILENAME"
    if [ "$?" -eq "0" ] ; then
        MAINCLASS=$(getClassName "$FILENAME")
        break
    fi
done
if [ "$MAINCLASS" = "" ] ; then
    for FILENAME in $SOURCE_FILES
    do
        hasMain "$FILENAME"
        if [ "$?" -eq "0" ] ; then
            MAINCLASS=$(getClassName "$FILENAME")
            break
        fi
    done
fi
if [ "$MAINCLASS" = "" ] ; then
    # Search for junit4 test classes
    TESTCLASS=
    for FILENAME in $SOURCE_FILES
    do
        CLASSFILE=$(getClassFile "$FILENAME")
        grep "org/junit/" $CLASSFILE &> /dev/null
        if [ "$?" -eq "0" ] ; then
            TESTCLASS=$(getClassName "$FILENAME")
            break
        fi
    done
    if [ "$TESTCLASS" = "" ] ; then
        echo "Class with \"public static void main(String[] arg)\" method not found"
        exit 0
    fi
fi
cat common_script.sh > vpl_execution
echo "export CLASSPATH=$CLASSPATH" >> vpl_execution
if [ ! "$MAINCLASS" = "" ] ; then
    echo "java -enableassertions $MAINCLASS \$@" >> vpl_execution
else
    echo "java org.junit.runner.JUnitCore $TESTCLASS \$@" >> vpl_execution
fi
chmod +x vpl_execution
for FILENAME in $SOURCE_FILES
do
    CLASSFILE=$(getClassFile "$FILENAME")
    grep -E "javax/swing/(JFrame|JDialog|JOptionPane|JApplet)" $CLASSFILE &> /dev/null
    if [ "$?" -eq "0" ] ; then
        mv vpl_execution vpl_wexecution
        break
    fi
done

```

### vpl\_evaluate.sh

```

#!/bin/bash
# This file is part of VPL for Moodle
# Default evaluate script for VPL
# Copyright (C) 2014 onwards Juan Carlos Rodriguez-del-Pino

```

```

# License http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or later
# Author Juan Carlos Rodriguez-del-Pino <jcrodriguez@dis.ulpgc.es>

#load VPL environment vars
. common_script.sh
if [ "$SECONDS" = "" ] ; then
    export SECONDS=20
fi
let VPL_MAXTIME=$SECONDS-5;
if [ "$VPL_GRADEMIN" = "" ] ; then
    export VPL_GRADEMIN=0
    export VPL_GRADEMAX=10
fi

#exist run script?
if [ ! -s vpl_run.sh ] ; then
    echo "I'm sorry, but I haven't a default action to evaluate the type of submitted files"
else
    #avoid conflict with C++ compilation
    mv vpl_evaluate.cpp vpl_evaluate.cpp.save
    #Prepare run
    ./vpl_run.sh &>>vpl_compilation_error.txt
    cat vpl_compilation_error.txt
    if [ -f vpl_execution ] ; then
        mv vpl_execution vpl_test
        if [ -f vpl_evaluate.cases ] ; then
            mv vpl_evaluate.cases evaluate.cases
        else
            echo "Error need file 'vpl_evaluate.cases' to make an evaluation"
            exit 1
        fi
        mv vpl_evaluate.cpp.save vpl_evaluate.cpp
        check_program g++
        g++ vpl_evaluate.cpp -g -lm -lutil -o .vpl_tester
        if [ ! -f .vpl_tester ] ; then
            echo "Error compiling evaluation program"
            exit 1
        else
            cat vpl_environment.sh >> vpl_execution
            echo "./.vpl_tester" >> vpl_execution
        fi
    else
        echo "#!/bin/bash" >> vpl_execution
        echo "echo" >> vpl_execution
        echo "echo '<|--'" >> vpl_execution
        echo "echo '-$VPL_COMPILATIONFAILED'" >> vpl_execution
        if [ -f vpl_wexecution ] ; then
            echo "echo '======'" >> vpl_execution
            echo "echo 'It seems you are trying to test a program with a graphic user interface'" >> vpl_execution
        fi
        echo "echo '--|>'" >> vpl_execution
        echo "echo" >> vpl_execution
        echo "echo 'Grade :>>$VPL_GRADEMIN'" >> vpl_execution
    fi
    chmod +x vpl_execution
fi

```

## C/C++ language

### Code file extension: cpp

When running the program, the source code must contain the public **main()** method, from which program execution starts.

```
#include <iostream>
using namespace std;

int main(){
    // write solution here:

    return 0;
}
```

### vpl\_run.sh

```
#!/bin/bash
# This file is part of VPL for Moodle - http://vpl.dis.ulpgc.es/
# Script for running C++ language
# Copyright (C) 2012 Juan Carlos Rodriguez-del-Pino
# License http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or later
# Author Juan Carlos Rodriguez-del-Pino <jcrodriguez@dis.ulpgc.es>

#@vpl_script_description Using default g++ with math and util libs
#load common script and check programs
. common_script.sh
check_program g++
if [ "$1" == "version" ] ; then
    echo "#!/bin/bash" > vpl_execution
    echo "g++ --version | head -n2" >> vpl_execution
    chmod +x vpl_execution
    exit
fi
get_source_files cpp C
#compile
g++ -fno-diagnostics-color -o vpl_execution $2 $SOURCE_FILES -lm -lutil
```

### vpl\_evaluate.sh

```
#!/bin/bash
# This file is part of VPL for Moodle
# Default evaluate script for VPL
# Copyright (C) 2014 onwards Juan Carlos Rodriguez-del-Pino
# License http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or later
# Author Juan Carlos Rodriguez-del-Pino <jcrodriguez@dis.ulpgc.es>

#load VPL environment vars
. common_script.sh
if [ "$SECONDS" = "" ] ; then
    export SECONDS=20
fi
let VPL_MAXTIME=$SECONDS-5;
if [ "$VPL_GRADEMIN" = "" ] ; then
    export VPL_GRADEMIN=0
    export VPL_GRADEMAX=10
fi
```

```

#exist run script?
if [ ! -s vpl_run.sh ] ; then
    echo "I'm sorry, but I haven't a default action to evaluate the type of submitted files"
else
    #avoid conflict with C++ compilation
    mv vpl_evaluate.cpp vpl_evaluate.cpp.save
    #Prepare run
    ./vpl_run.sh &>>vpl_compilation_error.txt
    cat vpl_compilation_error.txt
    if [ -f vpl_execution ] ; then
        mv vpl_execution vpl_test
        if [ -f vpl_evaluate.cases ] ; then
            mv vpl_evaluate.cases evaluate.cases
        else
            echo "Error need file 'vpl_evaluate.cases' to make an evaluation"
            exit 1
        fi
        mv vpl_evaluate.cpp.save vpl_evaluate.cpp
        check_program g++
        g++ vpl_evaluate.cpp -g -lm -lutil -o .vpl_tester
        if [ ! -f .vpl_tester ] ; then
            echo "Error compiling evaluation program"
            exit 1
        else
            cat vpl_environment.sh >> vpl_execution
            echo "./.vpl_tester" >> vpl_execution
        fi
    else
        echo "#!/bin/bash" >> vpl_execution
        echo "echo" >> vpl_execution
        echo "echo '<|--'" >> vpl_execution
        echo "echo '-$VPL_COMPILATIONFAILED'" >> vpl_execution
        if [ -f vpl_wexecution ] ; then
            echo "echo '===='" >> vpl_execution
            echo "echo 'It seems you are trying to test a program with a graphic user interface'" >> vpl_execution
        fi
        echo "echo '--|>'" >> vpl_execution
        echo "echo" >> vpl_execution
        echo "echo 'Grade :=>>$VPL_GRADEMIN'" >> vpl_execution
    fi
    chmod +x vpl_execution
fi

```

## Python

### Code file extension: py

No special needs are defined.

### vpl\_run.sh

```

#!/bin/bash
# This file is part of VPL for Moodle - http://vpl.dis.ulpgc.es/
# Script for running Python language
# Copyright (C) 2014 onwards Juan Carlos Rodriguez-del-Pino

```

```
# License http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or later
# Author Juan Carlos Rodriguez-del-Pino <jcrodriguez@dis.ulpgc.es>

# @vpl_script_description Using python3 with the first file
# load common script and check programs
. common_script.sh
check_program python3
if [ "$1" == "version" ] ; then
    echo "#!/bin/bash" > vpl_execution
    echo "python3 --version" >> vpl_execution
    chmod +x vpl_execution
    exit
fi
get_first_source_file py
cat common_script.sh > vpl_execution
echo "export TERM=ansi" >>vpl_execution
echo "python3 $FIRST_SOURCE_FILE \"$@" >>vpl_execution
chmod +x vpl_execution
grep -E "Tkinter" $FIRST_SOURCE_FILE &> /dev/null
if [ "$?" -eq "0" ] ; then
    mv vpl_execution vpl_wexecution
fi
```

## vpl\_evaluate.sh

```
#!/bin/bash
# This file is part of VPL for Moodle
# Default evaluate script for VPL
# Copyright (C) 2014 onwards Juan Carlos Rodriguez-del-Pino
# License http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or later
# Author Juan Carlos Rodriguez-del-Pino <jcrodriguez@dis.ulpgc.es>

#load VPL environment vars
. common_script.sh
if [ "$SECONDS" = "" ] ; then
    export SECONDS=20
fi
let VPL_MAXTIME=$SECONDS-5;
if [ "$VPL_GRADEMIN" = "" ] ; then
    export VPL_GRADEMIN=0
    export VPL_GRADEMAX=10
fi

#exist run script?
if [ ! -s vpl_run.sh ] ; then
    echo "I'm sorry, but I haven't a default action to evaluate the type of submitted files"
else
    #avoid conflict with C++ compilation
    mv vpl_evaluate.cpp vpl_evaluate.cpp.save
    #Prepare run
    ./vpl_run.sh &>>vpl_compilation_error.txt
    cat vpl_compilation_error.txt
    if [ -f vpl_execution ] ; then
        mv vpl_execution vpl_test
        if [ -f vpl_evaluate.cases ] ; then
            mv vpl_evaluate.cases evaluate.cases
        else
            echo "Error need file 'vpl_evaluate.cases' to make an evaluation"
        fi
    fi
fi
```

```

    exit 1
fi
mv vpl_evaluate.cpp.save vpl_evaluate.cpp
check_program g++
g++ vpl_evaluate.cpp -g -lm -lutil -o .vpl_tester
if [ ! -f .vpl_tester ] ; then
    echo "Error compiling evaluation program"
    exit 1
else
    cat vpl_environment.sh >> vpl_execution
    echo "./.vpl_tester" >> vpl_execution
fi
else
    echo "#!/bin/bash" >> vpl_execution
    echo "echo" >> vpl_execution
    echo "echo '<|--'" >> vpl_execution
    echo "echo '-$VPL_COMPILATIONFAILED'" >> vpl_execution
    if [ -f vpl_wexecution ] ; then
        echo "echo '===='" >> vpl_execution
        echo "echo 'It seems you are trying to test a program with a graphic user
interface'" >> vpl_execution
    fi
    echo "echo '--|>'" >> vpl_execution
    echo "echo" >> vpl_execution
    echo "echo 'Grade :=>$VPL_GRADEMIN'" >> vpl_execution
fi
chmod +x vpl_execution
fi

```

## PHP

### Code file extension: php

Making the input channel available is necessary to provide an automatic evaluation of source code written in PHP. This operation is provided by the following lines, which are necessary to start each program in the PHP course:

```

<?php
// your input, do not remove this line
$x = (int) trim(fgets(STDIN));
// write your code here

```

### vpl\_run.sh

```

#!/bin/bash
# This file is part of VPL for Moodle - http://vpl.dis.ulpgc.es/
# Script for running PHP language
# Copyright (C) 2012 onwards Juan Carlos Rodriguez-del-Pino
# License http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or later
# Author Juan Carlos Rodriguez-del-Pino <jcrodriguez@dis.ulpgc.es>
# @vpl_script_description Using "php -n -f" with the first file or on serve if
index.php exists
# load common script and check programs
. common_script.sh
check_program php5 php
PHP=$PROGRAM
if [ "$1" == "version" ] ; then

```

```

echo "#!/bin/bash" > vpl_execution
echo "$PHP -v" >> vpl_execution
chmod +x vpl_execution
exit
fi
check_program x-www-browser firefox
BROWSER=$PROGRAM
if [ -f index.php ] ; then
    PHPCONFIGFILE=$(($PHP -i 2>/dev/null | grep "Loaded Configuration File" | sed
's/^[^\s/]*/' )
    if [ "$PHPCONFIGFILE" == "" ] ; then
        touch .php.ini
    else
        cp $PHPCONFIGFILE .php.ini
    fi
    #Configure session
    SESSIONPATH=$HOME/.php_sessions
    mkdir $SESSIONPATH
    #Generate php.ini
    cat >> .php.ini <<END_OF_INI

session.save_path="$SESSIONPATH"
error_reporting=E_ALL
display_errors=On
display_startup_errors=On
END_OF_INI
    #Generate router
    cat >> .router.php << 'END_OF_PHP'
<?php $path=urldecode(parse_url($_SERVER["REQUEST_URI"],PHP_URL_PATH));
$file='.'. $path;
if(is_file($file) || is_file($file.'/index.php') || is_file($file.'/index.html') ){
    unset($path,$file);
    return false;
}
$pclean=htmlentities($path);
http_response_code(404);
header(':', true, 404);
?>
<!doctype html>
<html><head><title>404 Not found</title>
<style>h1{background-color: aqua;text-align:center} code{font-size:150%}</style>
</head>
<body><h1>404 Not found</h1><p>The requested resource <code><?php echo "'$pclean'";
?></code>
was not found on this server</body></html>
END_OF_PHP
while true; do
    PHPPORT=$((6000+$RANDOM%25000))
    netstat -tln | grep -q ":$PHPPORT "
    [ "$?" != "0" ] && break
done
cat > vpl_wexecution <<END_OF_SCRIPT

#!/bin/bash
$PHP -c .php.ini -S "127.0.0.1:$PHPPORT" .router.php &
$BROWSER "127.0.0.1:$PHPPORT"
END_OF_SCRIPT
    chmod +x vpl_wexecution
else

```

```

get_first_source_file php
cat common_script.sh > vpl_execution
echo "$PHP -n -f $FIRST_SOURCE_FILE \${@}" >>vpl_execution
chmod +x vpl_execution
fi

```

## vpl\_evaluate.sh

```

#!/bin/bash
# This file is part of VPL for Moodle
# Default evaluate script for VPL
# Copyright (C) 2014 onwards Juan Carlos Rodr#iguez-del-Pino
# License http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or later
# Author Juan Carlos Rodr#iguez-del-Pino <jcrodriguez@dis.ulpgc.es>
#load VPL environment vars
. common_script.sh
if [ "$SECONDS" = "" ] ; then
    export SECONDS=20
fi
let VPL_MAXTIME=$SECONDS-5;
if [ "$VPL_GRADEMIN" = "" ] ; then
    export VPL_GRADEMIN=0
    export VPL_GRADEMAX=10
fi
#exist run script?
if [ ! -s vpl_run.sh ] ; then
    echo "I'm sorry, but I haven't a default action to evaluate the type of submitted files"
else
    #avoid conflict with C++ compilation
    mv vpl_evaluate.cpp vpl_evaluate.cpp.save
    #Prepare run
    ./vpl_run.sh &>>vpl_compilation_error.txt
    cat vpl_compilation_error.txt
    if [ -f vpl_execution ] ; then
        mv vpl_execution vpl_test
        if [ -f vpl_evaluate.cases ] ; then
            mv vpl_evaluate.cases evaluate.cases
        else
            echo "Error need file 'vpl_evaluate.cases' to make an evaluation"
            exit 1
        fi
        mv vpl_evaluate.cpp.save vpl_evaluate.cpp
        check_program g++
        g++ vpl_evaluate.cpp -g -lm -lutil -o .vpl_tester
        if [ ! -f .vpl_tester ] ; then
            echo "Error compiling evaluation program"
            exit 1
        else
            cat vpl_environment.sh >> vpl_execution
            echo "./.vpl_tester" >> vpl_execution
        fi
    else
        echo "#!/bin/bash" >> vpl_execution
        echo "echo" >> vpl_execution
        echo "echo '<!--'" >> vpl_execution
        echo "echo '-$VPL_COMPILATIONFAILED'" >> vpl_execution
        if [ -f vpl_wexecution ] ; then
            echo "echo '===='" >> vpl_execution

```

```
    echo "echo 'It seems you are trying to test a program with a graphic user  
interface'" >> vpl_execution  
fi  
echo "echo '--|>'" >> vpl_execution  
echo "echo" >> vpl_execution  
echo "echo 'Grade :=>>$VPL_GRADEMIN'" >> vpl_execution  
fi  
chmod +x vpl_execution  
fi
```

## UnitTest2VPL Framework

VPL provides two ways to evaluate activities:

- The simplest way is to evaluate the input/output of a program. We only have to fill a file (**vpl\_evaluate.cases**, in the execution files section of the activity) with the input we want to provide to the program and the output we expect. We can also configure other stuff, as penalization, when the output is not correct. Still, with only the input and the expected output, VPL can run the evaluation, applying the input and testing that the output matches the expected one.
- Advanced evaluation requires that the evaluator prepares some code to test the code submitted by the student, usually using the same programming language. This code could be entirely designed for each activity or based on some general, customizable framework. Such a framework could be written to take advantage of an existing unit test framework, as a unit test for Python. Of course, it could be written differently.

### Requirements

#### Produce suitable VPL feedbacks

##### Problem

VPL expects that the evaluation process of activity produces feedbacks using a pre-established texts format.

##### Discussion

The format required for writing VPL feedbacks is public. A test framework could produce the feedbacks text simply by means of output commands. When using a unit tests framework, the messages associated with the asserts that are usually used in the tests could, in most cases, be customized to adopt a format suitable to VPL

##### Resulting requirement

**The code designed to carry on advanced evaluation must produce suitable VPL feedbacks.** It is recommended to use a standard unit test framework and customize the asserts messages properly.

#### Catch unexpected exceptions

##### Problem

The tested code could raise an unexpected exception.

##### Discussion

An unexpected exception could interrupt the test execution without providing any suitable VPL feedbacks.

##### Resulting requirement

**Every unit test must be wrapped with an exceptions control block to catch any unexpected exception and provide proper VPL feedback.**

## Prevent from non-ending states

### Problem

The tested code could enter non-ending states such as infinite looping or when waiting for an input that will not arrive.

### Discussion

A VPL activity establishes a maximum execution time in its resource limits setup page. When the established time is reached, VPL kills the evaluation process of the activity, stopping all pending tasks. As a result, an infinite loop in a test could cause the loss of feedback for many other tests, resulting in a waste of time.

### Resulting requirement

**The test framework should include a timeout mechanism to stop only the tests that enter non-ending states, producing proper feedback about them and continuing with the rest until their normal finalization.**

## Solution for Python

The solution adopted for Python VPL activities is to provide a base activity, the `UnitTest2VPL Base` activity, designed to be used as base activity by any other VPL activity that makes tests using a unit test framework. It provides the tools to convert unit test results into suitable VPL feedback, including comments and grade calculation.

### Unittest2VPL Base Activity

The **Unittest2VPL Base Activity** is composed of: A Python script (file `main_evaluate.py`) to collect and run the tests and three Python classes (**VPLMessage**, **TestTimeout**, and **VPLTests**). It also defines its own `vpl_run` and `vpl_evaluate` bash scripts.

### Class VPLMessage

The class **VPLMessage** extends the Python's class `str` to represent valid VPL comments. A VPL comment requires a title, a detailed explanation, and a penalty, as it usually represents a failure in a test run on a task submitted to be graded. The detailed explanation and the penalty could be optional. Still, the **VPLMessage** class does not manage this issue, so the magic method `__new__` is defined with four arguments (apart of the mandatory `cls` argument): **test** (which binds the message with the unit test that raises the failure), **title**, **detail** and **penalty**.

```
class VPLMessage(str):
    """This class provides strings formatted for VPL report"""

    def __new__(cls, test, title: str, detail: str, penalty: float):
        """Adds to the class str some attributes required for VPL feedback"""
        obj = str.__new__(cls, title)
        obj.test = test
        obj.detail = detail
        obj.penalty = penalty
        return obj
```

```

def __str__(self):
    """Returns a string formatted for VPL report"""
    header = "Comment :=>>-{0}.({1})".format(str.__str__(self), self.penalty)
    body = ""

    if self.detail != None:
        lines = self.detail.split("\n")

        for line in lines:
            body += "Comment :=>>>{0}\n".format(line)

        self.test.updateGrade(self.penalty)

    return "\n" + header + "\n" + body

```

We must use the **\_\_new\_\_** method to construct a VPL message because **str** is an immutable class and does not permit to add attributes via the **\_\_init\_\_** method. We need to add the attributes **test**, **detail** and **penalty**, which will be used later to build the VPL message in the **\_\_str\_\_** method. This way, we can assure that grade for the assignment will be updated only when it is required to show the VPL comment because a fail has been raised. The **title** does not need to be added as an attribute because it is used to make the base **str** object.

### TestTimeout class

The **TestTimeout** class serves to build timeout objects which will be used in a context manager to end the execution of the tested code in case of non-ending sceneries as infinite loops.

```

import signal

class TestTimeout:
    def __init__(self, seconds, message, test):
        """Sets a timeout with a message to be shown when reached"""
        self.seconds = seconds
        self.message = message
        self.test = test

    def handle_timeout(self, signum, frame):
        """Raises a test fail when the timeout is reached"""
        self.test.fail(self.message)

    def __enter__(self):
        """Activates the timeout countdown"""
        signal.signal(signal.SIGALRM, self.handle_timeout)
        signal.alarm(self.seconds)

    def __exit__(self, exc_type, exc_val, exc_tb):
        """Deactivates the timeout"""
        signal.alarm(0)

```

The **\_\_init\_\_** method of the **TestTimeout** class requires as arguments the timeout time expressed in seconds, a VPL message (which includes the short description of the test, the detail of the timeout failure, and a proposed penalty for it), and the test itself.

The **TestTimeout** class sets the alarm when entering the managed context to be launched when the timeout is reached. The handle for this alarm raises a test fail with the pre-established VPL message.

If the end of the managed context is reached before the timeout, the alarm is deactivated by the `__exit__` method.

### VPLTests class

The **VPLTests** class is defined in the package **unittest2VPL**, which imports the modules **re** (regular expressions), **os** (operating system), **traceback**, **unittest**, **VPLMessage** and **TestTimeout**.

The **VPLTests** class extends the class **unittest.TestCase** and defines four class methods (**setUpClass**, **showGrade**, **updateGrade** and **tearDownClass**) and three instance methods (**message**, **timeout** and **runTest**).

```
import os
import traceback
import unittest
from vplmessage import VPLMessage
from testtimeout import TestTimeout
from localization import Localization

class VPLTests(unittest.TestCase):
    """Base class for testing of VPL activities written in Python"""

    show_grade = False
    __default_lang = "en"

    def set_penalty_percentage(self, percentage):
        self.penalty = self.grade_range * percentage / 100.0
```

The **setUpClass** method is executed only once before running the tests and initializes the default penalty to be applied for each test failure by dividing the grade range by the number of tests to run (note that the default penalty is a negative value). It also initializes the failed tests counter.

```
@classmethod
def setUpClass(cls):
    """Prepare the execution of the tests"""
    grade_min = float(os.getenv('VPL_GRADEMIN', '0'))
    grade_max = float(os.getenv('VPL_GRADEMAX', '10'))
    cls.grade_range = grade_max - grade_min
    cls.grade = cls.grade_range
    cls.tests = list(filter(lambda x: x.find("test_") == 0, dir(cls)))
    cls.number_of_tests = len(cls.tests)
    cls.default_penalty = - (grade_max - grade_min) / cls.number_of_tests
    cls.failed = 0
    cls.__addLocalizations()
    Localization.setLanguage(cls.__default_lang)
```

The **showGrade** method changes to **True** the **show\_grade** attribute, which determines if a proposed grade is shown at the end of the tests. The initial value for **show\_grade** is **False** and only can be changed to **True**.

```
@classmethod
def showGrade(cls):
    """Allows showing a proposed grade to the student"""
    cls.show_grade = True
```

The **updateGrade** method updates the proposed grade every time a test fails and increments the failed tests counter. Updating the proposed grade adds the penalty argument (supposed negative) to the grade class attribute.

```
@classmethod
def updateGrade(cls, penalty):
    """Updates the grade to be proposed with a penalization"""
    cls.grade += penalty
    cls.failed += 1
```

The **runTest** method:

1. Calls the **setUpClass** method.
2. Execute the unit tests.
3. Calls the **tearDownClass** method.

For each unit test:

1. Calls the test **setUp** method, if any.
2. Set the penalty for the test to the default penalty (this can be overridden by the test method, setting a custom penalty).
3. Configures a timeout context and run the test method inside it.
4. Catches any exception raised by the unit test, distinguishing between failure exceptions and any other unexpected exception and producing a proper VPL message in any case.
5. Calls the test **tearDown** method, if any.

```
def runTest(self, result = None):
    """Executes VPL tests on the submitted code"""

    self.setUpClass()

    for test_name in self.tests:
        if hasattr(self, "setUp"):
            getattr(self, "setUp")()

        try:
            self.penalty = self.default_penalty
            test = getattr(self, test_name)
            with self.timeout(3, test.__doc__):
                test()

        except Exception as e:
            if type(e) == self.failureException:
                if hasattr(test, "__doc__"):
                    message = str(e).replace(
                        "None",
                        Localization.localize(test.__doc__)
                    )
            else:
                message = VPLMessage(
                    self,
                    Localization.localize(test.__doc__),
                    Localization.localize(
                        #"Unexpected {} - {}\n-----\n{}",
                        #[type(e).__name__, str(e), traceback.format_exc()]
                        "Unexpected {} - {}",
                        [type(e).__name__, str(e)]
                    ),
                    self.penalty
                )
            print(message)
            if hasattr(self, "tearDown"):
                getattr(self, "tearDown")()

    self.tearDownClass()
```

## Scripts

The file named **main\_evaluate.py** contains a Python script to collect and run the tests.

```
import unittest
import tests

my_test = tests.Tests()
my_test.run()
```

The **vpl\_evaluate.sh** bash script is used to run the **main\_evaluate.py** script when the activity is going to be evaluated.

```
#!/bin/bash
cat common_script.sh > vpl_execution

echo "python3 main_evaluate.py" >>vpl_execution;
chmod +x vpl_execution
```

The **vpl\_run.sh** bash script runs the first file of the activity's submission. It is unnecessary because that is what VPL does by default.

```
#!/bin/bash
. common_script.sh
check_program python
cat common_script.sh > vpl_execution
echo "python3 $VPL_SUBFILE0" >>vpl_execution;
chmod +x vpl_execution
```

## Test class

The **test.py** file must import the **unittest2VPL** module defined in the **UnitTest2VPL Base activity**. It must also import any other module required to test the assignment and declare a class named **Tests** that will extend **unittest2VPL.VPLTests**. Test class will include the tests methods and any other method necessary to test the assignment.

## Test methods

The test methods will be written as normal **UnitTests** methods, with names beginning with the prefix **test**, but with two requirements:

- The first line must be a doc comment with a short description of the test to be used as the title of the VPL message if the test fails.
- You can use any of the assertion clauses available in **Unittest**, but always provide a custom message built using the **self.message** method to be a proper VPL message.

```
def test_00(self):
    """Initializing a square rectangle and testing area"""

    # Prepare the test
    length = 1.0
    width = 1.0
    expected_area = abs(length) * abs(width)
```

```

# Run code to test
my_rect = Rectangle(length, width)
real_area = my_rect.area

# Evaluate results
self.assertEqual(
    real_area,
    expected_area,
    self.message(
        self.shortDescription(),
        ("A rectangle with length = {} and width = {}\n"
         "Expected area is {} {}\n"
         "But real area seems to be {}".format(
            length, width, expected_area, real_area)
        )
    )
)

```

The test method could include, previous to the test itself, an assignation to the **self.penalty** attribute, only in the case when a failure will be penalized with a value different from the default.

If a proposed grade is going to be shown, the class method **tearDownClass** must be overridden with a new one to call the class method **showGrade** and then the superclass method **tearDownClass**. Any other required finalization action can be done between these two calls.

```

@classmethod
def tearDownClass(cls):
    cls.showGrade()
    super().tearDownClass

```

## Internationalization and Localization

Internationalization and localization of software are important issues in a global world. **Unitest2VPL** is prepared to give its feedback properly translated to any required language, based on the previous addition of translations for that language.

### The class Location

The main tool for the internationalization and localization of Unittest2VPL based activities is the Localization class provided by the localization module. The Location class has three class methods: **setLanguage**, **addLocale**, and **localize**. For all of them, the first parameter, *cls*, represents the Localization class, as is usual in Python:

- The **setLanguage** method has the form: **setLanguage(cls, lang)**, being **lang** a string representing a language. The method changes the current language to that represented by **lang**.
- The **addLocale** method has the form: **addLocale(cls, lang, key, value)**, being **lang**, **key**, and **value** three strings. The method adds the translation **value** of **key** for the language **lang**. The **value** parameter could include placeholders to be replaced by real values when key translation is required, as explained at the next point.
- The **localize** method has the form: **localize(cls, key, params = [])**, **key** a string and **params** a list of objects. The method returns the value previously registered for key by **addLocale** for the current language (set by **setLanguage**). Before return value, **localize** replaces its placeholders by string representations of the positional-corresponding objects in **params**. In

case of not localizing a translation of **key** for the current language, the **key** itself is returned as value.

The placeholders have the form "{}". An escaping backslash must be included to allow the inclusion of a group "{}" in a value string ("\\{}" → "{}"). See example:

- **value:** "Result is {} and must be {}. These curly brackets \{} will stay."
- **params:** [198.99, 190.0]
- **returned value:** "Result is 198.99 and must be 190.0. These curly brackets {} will stay."

## How to use

### Localization of the VPLTest class

The **VPLTests** class uses English as a base and default language and has all its feedback translated to Spanish. It is prepared to show localized feedback for any derived activity that uses localization. New localizations for **VPLTests**' feedback can be added to the private class method **\_\_Localizations**.

```
@classmethod
def _addLocalizations(cls):
    # Set the localizations for the test
    Localization.addLocale(
        "es",
        "Testing the area property existence",
        "Probando la existencia de la propiedad area"
    )
    Localization.addLocale(
        "es",
        "Testing that area is readonly",
        "Probando que la propiedad area es de solo lectura"
    )
    Localization.addLocale(
        "es",
        "area property is writable",
        "La propiedad area es modificable"
    )
    Localization.addLocale(
        "es",
        "Testing area value (1)",
        "Probando el valor de la propiedad area (1)"
    )
```

All the feedback messages printed by the **VPLTests** class are localized using the **localize** function of the class **Localization**. As can be seen in both the above and below figures, messages in the base language (English) are used as keys for localization.

```
print(
    Localization.localize(
        "Comment :=>> {} tests failed of {} tests runned",
        [cls.failed, len(cls.tests)]
    )
)
```

The default language is set at the **setUpClass** class method by calling the method **setLanguage** of **Localization** using the private variable **\_\_default\_lang** as a parameter. To change the default language, only the value of this private variable needs to be changed.

```
cls.failed = 0
cls._addLocalizations()
Localization.setLanguage(cls._default_lang)
```

```
class VPLTests(unittest.TestCase):
    show_grade = False
    _default_lang = "en"
```

### Localization of derived Test classes

A test class derived from **VPLTests** can localize its feedback. To do this, it must define its own **setUpClass** class method, which must first call the **VPLTests'** **setUpClass** class method and then add its own localizations using the **addLocale** method the **Localization** class, which must be imported previously.

```
from localization import Localization
...
class Tests(unittest2VPL.VPLTests):
    @classmethod
    def setUpClass(cls):
        super().setUpClass()

        # Set the localizations for the test
        Localization.addLocale(
            "es",
            "Time has not {} attribute",
            "El objeto de la clase Time no tiene atributo {}"
        )
    ...
```

The messages to be used as assessment feedback must be localized using the **localize** function of the **Localization** class. The test class does not print these messages, and the **VPLTest** base class prints them. The **Test** class establishes them as assertion fail messages or docstrings for the tests methods. The docstrings can't, and don't need, be localized using the **localize** function; this task is done at the **VPLTest** base class, the derived **Test** class only needs to add localizations for them in its **setUpClass** class method.

The default language set at the **VPLTests** base class can be overridden by including a call to the **setLanguage** method of **Location** in the **setUpClass** method of the derived **Test** class.

```
# Set the language selected by the user if any

rec_doc = rectangle.__doc__

if rec_doc != None:
    lang_esp = re.search(r"\(test-lang=(\w{2,})\)", rec_doc)

    if lang_esp != None:
        Localization.setLanguage(lang_esp.group(1))
    else:
        Localization.setLanguage("en")
```

An activity can allow users to use their own language. To do this, the user must include a special pattern like `"/(test-lang=es)/"` in the docstring of his module, and the Test class must include code to process that pattern in its **setUpClass** class method. (In the next figure, **rectangle** is the module developed by the user). If there is no localization for the language selected by the user, the alternate language is used. If no any exists, the default language is selected.

## VPL activity configuration

### Execution options

A VPL activity assignment must be based on the **UnitTest2VPL Base activity** (configured in **Execution options**).

▼ Execution options
 

Based on
 

UnitTest2VPL Base activity
 ▼

### Execution files

It must include a file named **test.py** in its **Execution files**.

```

1 import unittest2VPL
2 from rectangle import Rectangle
3
4 class Tests(unittest2VPL.VPLTests):

```

## PRISCILLA configuration

To set up recurring files for all program assignments, you can use the course settings where these files are defined and saved:

Program code settings:

Code file extension: .py

Autocompletion template: text/x-python

Execution and support files:

⊕ vpl\_run.sh vpl\_debug.sh vpl\_evaluate.sh

1 #!/bin/bash  
2  
3 . common\_script.sh  
4 check\_program python  
5 cat common\_script.sh > vpl\_execution  
6 echo "python3 \$VPL\_SUBFILE0" >>vpl\_execution;  
7 chmod +x vpl\_execution

Keeping files:

⊕ main\_evaluate.py unittest2VPL.py vplmessage.py

1 import unittest  
2 import tests  
3  
4 my\_test = tests.Tests()  
5 my\_test.run()  
6

Files are prepared to test specific tasks with test procedures, and localization is part of individual tasks. They can be added to the **Data files** or **Config files** section

PRISCILLA v 2.0004 - 2021-10-12

DASHBOARD COURSE LIST SK 0 XP 100

Program file(s) ▾

Config file(s) ▾

Data file(s) ▴

Data file list

tests.py addlocals.py

```
tests.py  addlocals.py
1 import re
2 import unittest2VPL
3 from localization import Localization
4 import addlocals
5 import rectangle
6 from rectangle import Rectangle
7
8 class Tests(unittest2VPL.VPLTests):
9     def test_01(self):
10         """Initializing a rectangle with length > width"""
11
12         # Prepare the test
13         length = 5.0
```

## SQLiteTest4VPL Framework

The SQLiteTest4VPL is a tool to create VPL activities that evaluate SQL easily. The system use script to generate the test cases using different datasets and obtain the correct answer by running a solution provided by the teacher on each data set.

The activity **SQLiteTest4VPL Base** is the root of the inherit tree and has, in the section **Advanced settings / Execution files**, the necessary scripts to generate the test cases and run the tests.

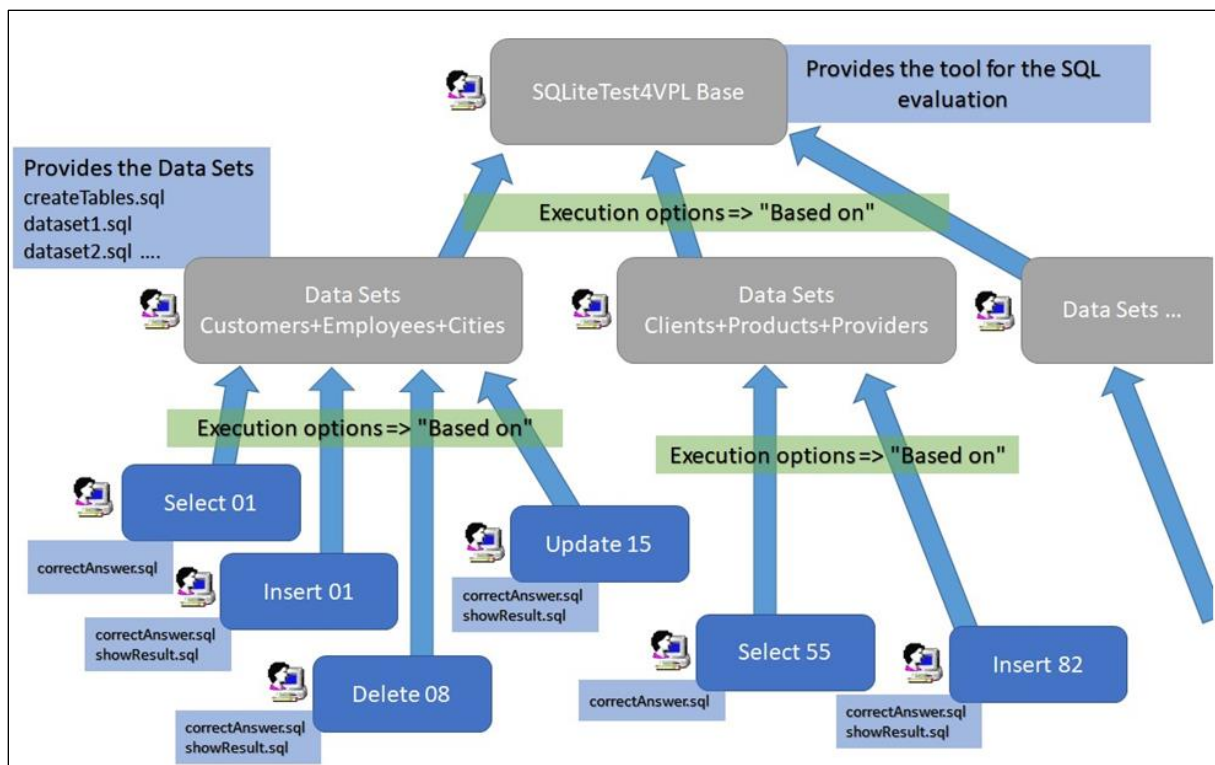
Each test case is based on a dataset. Each dataset is created by a sequence of SQL inserts in one or more tables.

The creation of tables must be done using SQL create commands that must be in a file called **createTables.sql**, in the section **Advanced settings / Execution files**.

Each dataset is populated by running insert SQL commands written in files named **dataset#.sql**, where '#' represents a number. The files must be in the **Advanced settings / Execution files** section. The files are processed consecutively, starting at the file named **dataset1.sql**, and the corresponding test cases will be generated according to the order established by those numbers.

Tables are created from scratch before running the dataset for each test case using the **createTables.sql** file.

Suppose we want to develop several VPL activities that use the same datasets. In that case, we will implement an activity that inherits from **SQLiteTest4VPL Base** in which we will include the file **createTables.sql** and as many **dataset#.sql** files as necessary. The process can be done easily by duplicating and modifying the activity **SQLiteTest4VPL Base Data Set Model**.



The activity **Data Set Customers+Employees+Cities** is an example of a base activity to provide datasets for final activities to be carried out by the student. Notice that the **SQLiteTest4VPL Base** and datasets activities are not intended to be used for the students, so they must be hidden.

We can create a new final activity, duplicating and modifying the activity **Activity tested with SQLiteTest4VPL** or other activity that uses the datasets we want to apply. It may be necessary to select others **based on** activities to use the correct datasets. In the **Advanced settings / Execution files** section, a final activity must include two files called **correctAnswer.sql** and **showResult.sql** (this one is not necessary if the solution is a select query).

The file **correctAnswer.sql** will contain the correct SQL commands to solve the activity, which will be used to generate the correct result prior to the execution of each test case to compare it with the result obtained by the execution of the student's response.

The file **showResult.sql** will contain SQL select commands to show the results of the execution of the student response (the state of the affected tables after the execution). The file should not be placed if the student's expected response is a select query. If the expected response is an insert, an update or a delete, the file **showResult.sql** MUST be included.

The activities **select01 v2**, **insert 01 v2**, **update 15 v2**, and **delete08 v2** are examples of final activities that inherit from the **Data Set Customers + Employees + Cities** activity, sharing the same dataset for the tests. They are modified versions of the activities **select01**, **insert 01**, **update 15** and **delete08** developed in the different sections of the course.

## The implementation files

The implementation contains the above files, while they can just as well be implemented in LMS Moodle or the PRISCILLA system.

## Scripts

### vpl\_run.sh

```
#!/bin/bash
# This file is part of VPL for Moodle - http://vpl.dis.ulpgc.es/
# Script for running SQL language (sqlite3)
# Copyright (C) 2018 Juan Carlos Rodríguez-del-Pino
# License http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or later
# Author Juan Carlos Rodríguez-del-Pino <jcrodriguez@dis.ulpgc.es>

# @vpl_script_description Using sqlite3
# load common script and check programs
. common_script.sh
check_program sqlite3
DBFILE=.vpl.db

#Generate execution script
cat common_script.sh > vpl_execution
#remove $DBFILE
function vpl_generate_execution {
    local SHOWRESULT
    local DATASET
    SHOWRESULT=vpl_showResult.sql
    CREATETABLES=vpl_createTables.sql
    local i
    for i in {1..1000}
    do
        DATASET="vpl_dataset${i}.sql"
```

```

    if [ -s "$DATASET" ] ; then
        grep -Ezq "VPL_fail_message_remove" "$DATASET"
        if [ $? -ne 0 ] ; then
            echo "rm $DBFILE &> /dev/null" >> vpl_execution
            echo "echo \"==== Data Set ${i}\" =====> vpl_execution
            echo "sqlite3 $DBFILE < $CREATETABLES" >> vpl_execution
            echo "sqlite3 $DBFILE < \"$DATASET\"> vpl_execution
            if [ -s "$SHOWRESULT" ] ; then
                echo "echo \"=> Initial state of the data set ${i}\">
vpl_execution
                echo "sqlite3 $DBFILE < $SHOWRESULT" >> vpl_execution
            fi
            echo "echo \"=> Execution of \"$VPL_SUBFILE0\" on data set ${i}\">
>> vpl_execution
            echo "sqlite3 $DBFILE < \"$VPL_SUBFILE0\"> vpl_execution
            if [ -s "$SHOWRESULT" ] ; then
                echo "echo \"=> Final state of the data set ${i}\">
vpl_execution
                echo "sqlite3 $DBFILE < $SHOWRESULT" >> vpl_execution
            fi
            echo "echo" >> vpl_execution
        else
            rm "$DATASET"
        fi
    else
        break
    fi
done
}

vpl_generate_execution
echo "sqlite3 $DBFILE" >> vpl_execution

chmod +x vpl_execution

```

## vpl\_evaluate.sh

```

#!/bin/bash
# This file is part of VPL for Moodle
# Default evaluate script for VPL
# Copyright (C) 2014 onwards Juan Carlos Rodriguez-del-Pino
# License http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or later
# Author Juan Carlos Rodriguez-del-Pino <jcrodriguez@dis.ulpgc.es>

#load VPL environment vars
. common_script.sh
if [ "$SECONDS" = "" ] ; then
    export SECONDS=20
fi
let VPL_MAXTIME=$SECONDS-5;
if [ "$VPL_GRADEMIN" = "" ] ; then
    export VPL_GRADEMIN=0
    export VPL_GRADEMAX=10
fi

#exist run script?
if [ ! -s vpl_run.sh ] ; then
    echo "I'm sorry, but I haven't a default action to evaluate the type of submitted files"

```

```

else
#avoid conflict with C++ compilation
mv vpl_evaluate.cpp vpl_evaluate.cpp.save
#Prepare run
./vpl_run.sh &>>vpl_compilation_error.txt
cat vpl_compilation_error.txt
if [ -f vpl_execution ] ; then
mv vpl_execution vpl_test
if [ -f vpl_evaluate.cases ] ; then
mv vpl_evaluate.cases evaluate.cases
else
echo "Error need file 'vpl_evaluate.cases' to make an evaluation"
exit 1
fi
mv vpl_evaluate.cpp.save vpl_evaluate.cpp
check_program g++
g++ vpl_evaluate.cpp -g -lm -lutil -o .vpl_tester
if [ ! -f .vpl_tester ] ; then
echo "Error compiling evaluation program"
exit 1
else
cat vpl_environment.sh >> vpl_execution
echo "./.vpl_tester" >> vpl_execution
fi
else
echo "#!/bin/bash" >> vpl_execution
echo "echo" >> vpl_execution
echo "echo '<|--'" >> vpl_execution
echo "echo '-$VPL_COMPILATIONFAILED'" >> vpl_execution
if [ -f vpl_wexecution ] ; then
echo "echo '===='" >> vpl_execution
echo "echo 'It seems you are trying to test a program with a graphic user
interface'" >> vpl_execution
fi
echo "echo '--|>'" >> vpl_execution
echo "echo" >> vpl_execution
echo "echo 'Grade :=>>$VPL_GRADEMIN'" >> vpl_execution
fi
chmod +x vpl_execution
fi

```

## SQLiteCase.sh

```

#!/bin/bash
# Copyright (C) 2019 Juan Carlos Rodriguez-del-Pino
# License http://www.gnu.org/copyleft/gpl.html GNU GPL v3 or later
# Author Juan Carlos Rodriguez-del-Pino <jcrodriguez@dis.ulpgc.es>
# DB Case for SQLITE
SQLITE=/usr/bin/sqlite3
DBFILE=.vpldata.db
CREATETABLES=vpl_createTables.sql
SHOWRESULT=vpl_showResult.sql
DATASET=$1
ANSWER=$2
SQLEXE=.sql_execution.sql
rm $DBFILE &> /dev/null
cp $CREATETABLES $SQLEXE
cat $DATASET >> $SQLEXE
cat $ANSWER >> $SQLEXE

```

```

if [ -s "$SHOWRESULT" ] ; then
    cat $SHOWRESULT >> $SQLEXE
fi
$SQLITE $DBFILE < $SQLEXE | sed "s/\\\"//g"

rm $DBFILE &> /dev/null
rm $SQLEXE &> /dev/null

```

### pre\_vpl\_run.sh

```

#!/bin/bash
. vpl_environment.sh
CASES=vpl_evaluate.cases
CORRECTANSWER=correctAnswer.sql
ANSWER=$VPL_SUBFILE0
SQLITECASE=SQLiteCase.sh
function rename_files {
    local CREATETABLES
    local SHOWRESULT
    CREATETABLES=createTables.sql
    SHOWRESULT=showResult.sql
    for i in {1..1000}
    do
        DATASET="dataset${i}.sql"
        if [ -s "$DATASET" ] ; then
            mv "$DATASET" "vpl_${DATASET}"
        else
            break
        fi
    done
    if [ -s "$CREATETABLES" ] ; then
        mv "$CREATETABLES" "vpl_${CREATETABLES}"
    else
        echo "ERROR: File createTables.sql missing"
    fi
    if [ -s "$SHOWRESULT" ] ; then
        mv "$SHOWRESULT" "vpl_${SHOWRESULT}"
    fi
    if [ -s "$SQLITECASE" ] ; then
        mv "$SQLITECASE" ".$SQLITECASE"
        SQLITECASE="./.$SQLITECASE"
    fi
}

function generate_sql_cases {
    rm $CASES &> /dev/null
    local FILENAME
    local i
    for i in {1..1000}
    do
        DATASET="vpl_dataset${i}.sql"
        if [ -s "$DATASET" ] ; then
            {
                echo "Case = Test $ANSWER on Data Set ${i}"
                grep -Ezq "VPL_fail_message_remove" "$DATASET"
                if [ $? -eq 0 ] ; then
                    echo "Fail message = The result is incorrect"
                fi
                echo "Grade reduction = 100%"
            }
        fi
    done
}

```

```

        echo "Program to run = $SQLITECASE"
        echo "Program arguments = $DATASET $ANSWER"
        echo -n "output =\""
        $SQLITECASE $DATASET $CORRECTANSWER
        echo "\"\"
        echo
    } >> $CASES
else
    break
fi
done
}
rename_files
if [ -f vpl_evaluate.sh ] ; then
    generate_sql_cases
fi
rm $CORRECTANSWER &> /dev/null
rm pre_vpl_run.sh

```

## Data preparation files

Scripts are common for all SQL assignments. Data preparation and individual tests are different for each assignment (in some cases, it is possible to use the same content of files – e.g. when defining the board and filling them).

### createTables.sql

This query creates tables according to SQLite syntax.

Example:

```

/*****
    Tables may exists. Drop Tables then here.
*****/
drop table if exists Customers;
drop table if exists Employees;
drop table if exists Cities;

/* Activate foreign_keys in sqlite3 */
PRAGMA foreign_keys = ON;

/*****
    Create Tables
*****/
create table Cities (
    Id integer PRIMARY KEY,
    Name text
);

create table Customers (
    Id integer PRIMARY KEY,
    Name text,
    Surname text,
    City_Id integer REFERENCES Cities(Id)
);

create table Employees (
    Id integer PRIMARY KEY,

```

```

    Name text,
    Surname text,
    JobTitleCode varchar(6)
);

create table EmployeesCopy (
    Id integer PRIMARY KEY,
    Name text,
    Surname text,
    JobTitleCode varchar(6)
);

create table CitiesCopy (
    Id integer PRIMARY KEY,
    Name text
);

create table CustomersCopy (
    Id integer PRIMARY KEY,
    Name text,
    Surname text,
    City_Id integer REFERENCES Cities(Id)
);

```

### dataset#.sql

The correctness of the student's query can be tested on many datasets (even empty ones). They must be numbered in the order 1-n.

Example:

```

insert into Cities (Id,Name) values (1,"Bratislava");
insert into Cities (Id,Name) values (2,"Nitra");
insert into Cities (Id,Name) values (3,"Trnava");
insert into Cities (Id,Name) values (4,"Katowice");
insert into Cities (Id,Name) values (5,"Telde");
insert into Cities (Id,Name) values (6,"Paris");
insert into Cities (Id,Name) values (8,"Galdar");
insert into Cities (Id,Name) values (10,"Krakow");

insert into Customers (Id,Name,Surname,City_Id) values (1, "Sofia","Bednar",10);
insert into Customers (Id,Name,Surname,City_Id) values (2, "Jan","Zachar",1);
insert into Customers (Id,Name,Surname,City_Id) values (3, "Nela","Walach",1);
insert into Customers (Id,Name,Surname,City_Id) values (4, "Witold","Nowak",5);
insert into Customers (Id,Name,Surname,City_Id) values (5,
"Katarzyna","Kowalska",6);
insert into Customers (Id,Name,Surname,City_Id) values (6, "Iwona","Bednarz",8);

insert into Employees (Id,Name,Surname,JobTitleCode) values
(1,"Jan","Kowalski","RKB012");
insert into Employees (Id,Name,Surname,JobTitleCode) values
(2,"Jan","Nowak","RKB003");
insert into Employees (Id,Name,Surname,JobTitleCode) values
(3,"Kamil","Wilmowski","RKB011");
insert into Employees (Id,Name,Surname,JobTitleCode) values
(4,"Olga","Milenka","RKB003");
insert into Employees (Id,Name,Surname,JobTitleCode) values
(5,"Angela","Wilga","RKB002");

```

In principle, two cases can be distinguished when evaluating the correctness of queries:

- Selection queries, where the evaluation of the result is a part of the answer - based on the output of data from the table (tables), we see whether the correct data were obtained.
- Action queries where the command changes the data, but we need to obtain information to know if the change was successful. We need to finish this process with a select query to get and check the final data in the database.

#### **correctAnswer.sql**

The file **correctAnswer.sql** is used for comparison of students queries and expected results. The text of queries can be different, but the important thing is the same outputs.

If the assignment requires an action query, the **showResult.sql** file is also required. The call of this file is provided by **SQLiteCase.sh** file (if **showResult.sql** exists)

Example of select:

Select all fields from countries ordered by Country

#### **correctAnswer.sql**

```
SELECT *  
FROM Countries  
ORDER BY Country;
```

Example of update:

Write a query that modifies a Capital name to Krakow from the Countries table, where the Country name is Poland.

#### **correctAnswer.sql**

```
UPDATE Countries  
SET Capital='Krakow'  
WHERE Country='Poland';
```

#### **showResult.sql**

```
SELECT *  
FROM Countries;
```

## JUnit4VPL Framework

The **JUnit4VPL** tool is a Java framework, fully integrated with the Virtual Programming Laboratory (VPL), mimics the well-known JUnit and seeks to meet the previously listed features to facilitate the evaluation of student code written in Java.

The direct use of **JUnit** would have multiple drawbacks, especially in the assessment, feedback, security, and internationalization. Notice that JUnit does not offer any customizable solution for these issues.

The users of **JUnit** usually test a single, known code when using this tool. The primary information obtained when a fail appears the line of the test code that triggers the error. The testing code state the details of the case tested and allows to reproduce the problem to find out the offending code. On the other hand, an evaluator needs to test many different codes which try to resolve the same problem with different approaches. These codes may have unusual errors due to the students' inexperience, including infinite loops, unexpected exceptions, security shortcomings and more.

### What offers JUnit4VPL?

The **JUnit4VPL** framework has the aim of evaluating students' code and showing a proper report to them. **JUnit4VPL** main features are:

- The grade range (minimum and maximum marks) can be specified. By default, the tool takes the grade range from the VPL activity.
- A text describing the test case that is being tested in the method can be specified. If the case fails, that text will appear in the report.
- A penalty may be set for each test fail. The penalty may be a fixed value or a per cent of the grade range. By default, the penalty is the grade range length divided by the number of test methods.
- The execution of test methods always follows the same order. The order of the execution is based on the lexicographic order of the name of the methods.
- The tool establishes a default timeout for every test method, but a different one can be set for each of them.
- The tool establishes a global timeout for the whole test suite.
- The tool is robust to fatal student code failures: infinite loop, stack overflow, exhaustion of threads, unexpected exceptions, etc.

### Using JUnit4VPL

**JUnit4VPL** attempts to be as similar as possible to JUnit, but some **JUnit** features were omitted, especially those that may allow altering the test results. For example, the **Assume** class has been omitted due to the possibility of using it to pass the entire test without checks. Asserts without messages have been removed too to fulfil the goal of always providing adequate feedback for assessments.

## Basic use of JUnit4VPL, the OddEven problem

The very simple problem **OddEven** requests the student to write a class **OddEven** with a static function **isOdd** that takes an integer parameter. The function **isOdd()** returns true if the number passed is odd and false if not.

The figure shows a class **TestOddEven** that tests the static method **OddEven.isOdd()** using JUnit4. The **TestOddEven** class use two methods, the first one for testing odd numbers and the other one for testing even numbers. Almost all asserts have a message that informs about the problem.

```

TestOddEven.java
1  import org.junit.Test;
2  import org.junit.runner.JUnit4;
3  import static org.junit.Assert.*;
4
5  public class TestOddEven {
6      @Test
7      public void testIsOdd() {
8          assertTrue("Testing isOdd with -1115", OddEven.isOdd(-1115));
9          assertTrue("Testing isOdd with -11", OddEven.isOdd(-11));
10         assertTrue("Testing isOdd with 1", OddEven.isOdd(1));
11         assertTrue("Testing isOdd with 3", OddEven.isOdd(3));
12         assertTrue("Testing isOdd with 999", OddEven.isOdd(999));
13         assertTrue("Testing isOdd with an odd number", OddEven.isOdd(13));
14         assertTrue("Testing isOdd with an odd number", OddEven.isOdd(21));
15     }
16     @Test
17     public void testIsEven() {
18         assertFalse("Testing isOdd with -12", OddEven.isOdd(-12));
19         assertFalse("Testing isOdd with -6", OddEven.isOdd(-6));
20         assertFalse("Testing isOdd with 0", OddEven.isOdd(0));
21         assertFalse("Testing isOdd with 2", OddEven.isOdd(2));
22         assertFalse("Testing isOdd with 100", OddEven.isOdd(100));
23         assertFalse("Testing isOdd with an even number", OddEven.isOdd(18));
24         assertFalse("Testing isOdd with an even number", OddEven.isOdd(444));
25     }
26     public static void main1(String[] args) {
27         JUnit4.main("TestOddEven");
28     }
29 }
30

```

We can use the above class to test the student code in the next figure. Notice that the student code is a dumb solution that always returns true.

```

▼ Execution

JUnit version 4.12
..E
Time: 0.008
There was 1 failure:
1) testIsEven(TestOddEven)
   java.lang.AssertionError: Testing isOdd with -12
       at org.junit.Assert.fail(Assert.java:88)
       at org.junit.Assert.assertTrue(Assert.java:41)

```

```

OddEven.java
1 public class OddEven {
2     public static boolean isOdd(int number) {
3         return true;
4     }
5 }
6

```

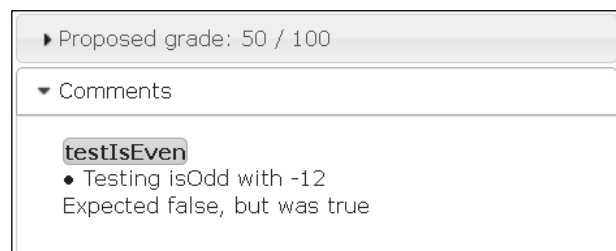
The result delivered by JUnit is not very useful for evaluation and feedback purposes. We can adapt the test to use **JUnit4VPL** changing **org.junit** the package name of JUnit for **es.ulpgc.junit4vpl** the package name of **JUnit4VPL**.

```

TestOddEvenVPL.java
1 import es.ulpgc.junit4vpl.Test;
2 import es.ulpgc.junit4vpl.runner.JUnitCore;
3 import static es.ulpgc.junit4vpl.Assert.*;
4
5 public class TestOddEvenVPL {
6     @Test
7     public void testIsOdd() {
8         assertTrue("Testing isOdd with -115", OddEven.isOdd(-11));
9         assertTrue("Testing isOdd with -11", OddEven.isOdd(-11));
10        assertTrue("Testing isOdd with 1", OddEven.isOdd(1));
11        assertTrue("Testing isOdd with 3", OddEven.isOdd(3));
12        assertTrue("Testing isOdd with 999", OddEven.isOdd(999));
13        assertTrue("Testing isOdd with an odd number", OddEven.isOdd(13));
14        assertTrue("Testing isOdd with an odd number", OddEven.isOdd(21));
15    }
16    @Test
17    public void testIsEven() {
18        assertFalse("Testing isOdd with -12", OddEven.isOdd(-12));
19        assertFalse("Testing isOdd with -6", OddEven.isOdd(-6));
20        assertFalse("Testing isOdd with 0", OddEven.isOdd(0));
21        assertFalse("Testing isOdd with 2", OddEven.isOdd(2));
22        assertFalse("Testing isOdd with 100", OddEven.isOdd(100));
23        assertFalse("Testing isOdd with an even number", OddEven.isOdd(18));
24        assertFalse("Testing isOdd with an even number", OddEven.isOdd(444));
25    }
26    public static void main(String[] args) {
27        JUnitCore.main("TestOddEvenVPL");
28    }
29 }
30

```

By default, **JUnit4VPL** will assign a penalty to each fail proportional to the number of test methods and take the name of the method as the title for the fail message. In this case, the dumb solution gets half of the grade range, and the message title is "TestIsEven".



The **JUnit4VPL Test** annotation has the elements **description** and **penalty**, allowing to customize the penalty and the title to show when an assert command fails. The description is a **String** that must describe the case checked. The **penalty** is a **String** that contents a constant value or a per cent. Using a per cent as a penalty may be more versatile than a constant value.

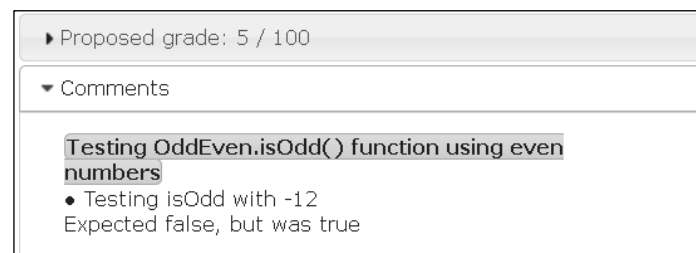
For example, if we want to penalize the fails as 95% of the range length, and to better detail what we are testing, the Test annotation of the test methods may look like this:

```

19  @Test (
20      description = "Testing OddEven.isOdd() function using even numbers",
21      penalty = "95%"
22  )

```

The evaluation report is more transparent and has a more appropriate grade than the previous one. The title of the message is verbose, and the grade is five of one hundred.



## Testing a students class

Testing classes differs from testing a static method mainly in the number of test cases needed and the object implicated in non-static methods. Often the object of the class to test has different states that may be as another dimension of a test case. Notice that when we call a method on an object, it is really another input of that method. A common approach is to prepare a set of objects with different states and use it when testing each class method. A specific test method creates this set of objects, frequently named fixture. JUnit supports this initialization with the tag **Before**. All the methods tagged as **Before** are running before running each method tagged as **Test**. **JUnit4VPL** support this tag and the related ones: **After**, **BeforeClass** and **AfterClass**. For security reasons, it is not recommended to initialize the fixture in the constructor of the test class.

```

1  package es.ulpgc.junit4vpl.examples;
2
3  import es.ulpgc.junit4vpl.*;
4  import static es.ulpgc.junit4vpl.Assert.*;
5
6  public class TestFraction {
7      private int[][] values = {{0, 2}, {5, 1}, {7, 3}, {2, 5}, {16, 4}, {6, 21}, {240, 180}};
8      private int[][] normalizedValues = {{0, 1}, {5, 1}, {7, 3}, {2, 5}, {4, 1}, {2, 7}, {4, 3}};
9      private Fraction[] fractions;
10     @Before
11     public void init() {
12         fractions = new Fraction[values.length];
13         for ( int i = 0; i < values.length; i++) {
14             fractions[i] = new Fraction(values[i][0], values[i][1]);
15         }
16     }
17     @Test(description = "Testing normalization after construction", penalty = "20%")
18     public void testNormalization() {
19         for ( int i = 0; i < values.length; i++) {

```

## Advanced testing customization

You only need to read this section if you find limitations or need a more detailed control of the basic customization of the Test annotation seen before. This section describes a more detailed use of the **JUnit4VPL** features that are different or extend the **JUnit** test framework. For more details, see the complete **JavaDoc** documentation. The main differences between **JUNIT4VPL** and Junit are:

- the new attributes of the **Test** annotation,

- the new **TestClass** annotation and
- the **ConsoleCapture** class.

## The Test annotation

**JUnit4VPL** modifies the Test annotation available in JUnit, adding the attributes **description** and **penalty**, as described above while accepting the **expected** and **timeout** attributes available in the original JUnit Test annotation. The **timeout** behaviour is slightly different because in **JUnit4VPL** a **timeout** always exists: if the **timeout** attribute of Test is not set, the **defaultTestTimeout** of **TestClass** is used. The timeout may need to be adjusted to do not surpass the global timeout.

## The TestClass annotation

**TestClass** is an annotation for classes that is not present in JUnit. **TestClass** sets global parameters to be applied to the test:

- **defaultTestTimeout** - sets the default timeout in milliseconds for each test method. The default value is 2000 and can be overridden for a method by defining the timeout attribute of Test.
- **globalTimeout** - sets the global timeout in milliseconds for the whole test suite. The default value is 30000, but the global timeout used is the minimum of **globalTimeOut** and the value set at the VPL activity options settings form. When a global timeout is reached, all pending tests are stopped, and penalization of 100% is applied.
- **timeoutPenalty** - sets the penalty to apply when a test method reaches its timeout. By default, the same penalization as for assertion fails is applied.
- **exceptionPenalty** - sets the penalization to apply when an unexpected exception is raised. By default, the same penalization as for assertion fails is applied.
- **expectedPenalty** - sets the penalization to apply when an expected exception is not raised. By default, the same penalization as for assertion fails is applied.

```

5  @TestClass (
6      defaultTestTimeout = 1000,
7      globalTimeout = 10000,
8      exceptionPenalty = "30%"
9  )
10 * public class TestFraction {

```

## The ConsoleCapture class

This class allows the creation of objects that can capture the standard output of the application. Capturing the output of the application has two goals: be able to check the output of the student's code, and avoid that the student's code interfering with the test report sent to VPL. **ConsoleCapture** has the following methods:

- **startCapture()** - saves and reassigns the out and err streams to new on-memory streams.
- **stopCapture()** - restores the saved out and err streams.
- **getCapturedOut()** - returns the text sent to the out stream from the last **captureStart()**.
- **getCapturedErr()** - returns the text sent to the err stream from the last **captureStart()**.
- **print(String text)** - sends the text to the saved out-stream (out of the capturing).

The best practice is to capture the streams as soon as possible and before starting the test. After ending the test, do not stop the capture: send the test report to VPL using the **ConsoleCapture** print method.

## JUnit4VPL internationalization

**JUnit4VPL** uses internationalised text to show from an object of **JUnitI18n** or a derived class. The default language is English, but other languages are available, as Spanish. You must call the static function **JUnitI18n.setLang()** with a new language object as a parameter to select an available language. The call to **setLang()** must be done before calling to **JUnitCore.runClasses()** or **JUnitCore.main()** methods to take effect.

To add a new language to **JUnit4VPL** it is necessary to extend the **JUnitI18n** class and override the methods that define the text to output. Some of the text strings are parametrized with one or two parameters that must be in the string. The current values replace the parameters when the text is used.

For example, the method **expectedButWas()** return a string with two parameters <expected> and <was>.

```
8 public class JUnitI18nEN extends JUnitI18n {
9     String expectedButWas() {
10         return "Expected <expected>, but was <was>";
11     }
```

The translated text for the Spanish language must also contain the two parameters. If **assertEquals("", 2, 3)** fails, the output text for English is "Expected 2, but was 3" and for Spanish is "Se esperaba 2, pero fue 3".

```
3 public class JUnitI18nES extends JUnitI18n {
4     @Override
5     String expectedButWas() {
6         return "Se esperaba <expected>, pero fue <was>";
7     }
8 }
```

The name of the new class must be **JUnit18nLC**, where LC is the language code capitalized and must belong to the **es.ulpgc.junit4vpl.i18n** package.

## JUnitBase Framework

The **JUnitBase** framework was proposed to simplify test preparation and create Java class assignments as quickly as possible. Although it is primarily debugged for this language, its use for other object languages represents only a minor modification of the code files; the philosophy remains.

The framework requires the correct solution (classes) as part of the testing process.

The principle of verifying the correctness of individual methods is to compare their results. It is also possible to compare the values of attributes if, for some reason, they are not defined as private.

The authors and students classes are compared after a sequence of (their) methods. The test class can generate the input values, the repeating of methods, the order of its calls, and other operations based on user/tester defined parameters.

### Example of assignment

Create a **Student** class that will have information about:

- first and last name
- the student's year
- the amount of the scholarship

Define a **constructor** with first name, last name, year and scholarship amount (integer), ensuring that the attributes are populated as follows:

- set the first and last name as they came in
- if the year is less than 1, generate an `IllegalArgumentException` with the text "too low year of study"
- if the year is greater than 5, generate an `IllegalArgumentException` with the text "too high year of study"
- If the scholarship is less than 0, generate an exception: `NumberFormatException` with text "negative scholarship"
- If the scholarship is less than 10000, generate an exception: `NumberFormatException` with the text "too expensive scholarship"

Define a **getInfo()** method that returns information about the student in the form of the name, surname, year, scholarship, e.g.:

```
first name last name, 5th, scholarship EUR
```

The following procedure presents an illustration of creating instances and calling methods:

```
public static void main(String[] args) {
    Student s1 = new Student("John", "Smith", 2, 5000);
    System.out.println(s1.getInfo()); // writes John Smith, 2., 5000 EUR
    // constructor ends with error: IllegalArgumentException: too high year of study
    Student s2 = new Student("John", "Doe", 8, 5000);
    // constructor ends with error: NumberFormatException: negative scholarship
    Student s3 = new Student("John", "Doe", 2, -5000);
}
```

## MySolution

MySolution class contains a complete and correct solution to the task. Due to its simplicity, probably no further comment is needed.

```
public class MySolution {
    private String name;
    private String surname;
    private int year_of_study;
    private int scholarship;

    public MySolution(String n, String sn, int yos, int sch) {
        name = n;
        surname = sn;
        if (yos < 1) throw new IllegalArgumentException("too low year of study");
        if (yos > 5) throw new IllegalArgumentException("too high year of study");
        year_of_study = yos;
        if (sch < 0) throw new NumberFormatException("negative scholarship");
        if (sch > 10000)
            throw new NumberFormatException("too expensive scholarship");
        scholarship = sch;
    }

    public String getInfo() {
        return name + surname + ", " +
            year_of_study + "., " + scholarship + " EUR";
    }
}
```

## Starting class

The starting class prepares a list of tests, defines the creation of input parameters, and summarizes the individual partial evaluations into the overall grade.

```
public class Main {
```

The constants for test names are defined for user information in which tests passed and failed.

```
    final static String TEST1_NAME = "constructor() valid data";
    final static String TEST2_NAME = "constructor() invalid data";
```

The tests details are defined as objects. Every tested object has a name and an importance, expressed as a percentage. For each element (method) test in the class, it is possible to perform any number of executions with different input values. In this case, pentuples representing the data for the constructor are defined:

- P - random positive integer value
- N - random negative integer value
- Z - zero
- R - random integer value
- X - the specific value specified after the X character

The method of generation can be modified at any time in the **Evaluate** class.

```

static Object[][] tests = {
    {TEST1_NAME, "20", new Object[][] {
        {"John", "Smith", "X1", "X2000"},
        {"Jason", "Bourne", "X2", "X5000"},
        {"Anna", "Green", "X3", "X0"},
        {"Jane", "Doe", "X4", "X10000"},
        {"Peter", "Pan", "X5", "X8000"},
    }},
    {TEST2_NAME, "80", new Object[][] {
        {"John", "Smith", "X0", "X2000"},
        {"Jason", "Bourne", "X12", "X5000"},
        {"Anna", "Green", "X3", "X-3000"},
        {"Jane", "Doe", "X4", "X12000"},
        {"Peter", "Pan", "X12", "X18000"},
    }},
};

```

The **main()** method goes through the individual tests, and the **grade** variable summarizes their results.

```

public static void main(String[] args) {
    long grade = 0;

    for(int i = 0; i < tests.length; i++) {
        int weight = Integer.parseInt((String)tests[i][1]);
        Object[][] cases = (Object[][])(tests[i][2]);
        // test name, weight, test cases
        grade += Math.round((processTest((String)tests[i][0], weight, cases)
            *100/cases.length) * weight/100);
    }

    System.out.println("Grade :=>>" + grade);
}

```

The **processTest()** method creates a new evaluation class and sends a test with input arrays. Catches errors and presents information about the progress of testing.

```

private static int processTest(String testName, int weight, Object[][] tests) {
    System.out.println("!--- Test: " + testName + " (" + weight + "%): ");
    Evaluate t = new Evaluate();
    String output = "";
    try {
        switch (testName) {
            case TEST1_NAME: output = t.testsAll(TEST1_NAME, tests); break;
            case TEST2_NAME: output = t.testsAll(TEST2_NAME, tests); break;
        }
    } catch (Exception e) { output = "" + e.getMessage(); }
    int correctCount = Integer.parseInt(output.substring(0, output.indexOf(";")));
    System.out.println(output.substring(output.indexOf(";") + 1));

    return correctCount;
}

```

## Evaluate class

The **Evaluate** class performs the execution of individual tests, compares the results of the student class and the sample class, and generates the output intended for the user.

```
public class Evaluate {
```

Definition of constants for generated output to support localization is as follow:

```
final String TEXT_ERROR = "failed";
final String TEXT_INPUT_VALUES = "input values";
final String TEXT_PROGRAM_OUTPUT = "program output";
final String TEXT_EXPECTED_OUTPUT = "expected output";
```

Value generator based on inputs from the **Main** class. The method type determines if integer or real values are generated.

```
public int getMyValue(String input) {
    String t = input.substring(0,1);
    int mx = 50;
    int val = 0;
    switch (t) {
        case "P": val = 1 + (int) (Math.random() * mx); break;
        case "N": val = -(int) (Math.random() * mx); break;
        case "Z": val = 0; break;
        case "R": val = -mx + (int) (Math.random() * (2 * mx + 1)); break;
        case "X": val = Integer.parseInt(input.substring(1)); break;
        default: val = 0;
    }
    return val;
}
```

Method for tests processing.

```
public String testsAll(String method, Object[][] cases) {
    String output = "";
    int correct_count = 0;
    String svalue1 = "", svalue2 = "", svalue3 = "", svalue4 = "";
    int ivalue1 = 0, ivalue2 = 0, ivalue3 = 0, ivalue4 = 0;
    double dvalue1 = 0, dvalue2 = 0;
```

Every line (array element) from the matrix with inputs is processed. String and integer parameters are inserted into variables and processed.

```
for (int i = 0; i < cases.length; i++) { // number of tests
    svalue1 = ((String) (cases[i][0]));
    svalue2 = ((String) (cases[i][1]));
    ivalue1 = getMyValue((String) (cases[i][2]));
    ivalue2 = getMyValue((String) (cases[i][3]));

    String result_test = "";
    String result_correct = "";
    String methods= "";
    MySolution correct;
```

```

Student tested;
try {
    switch (method) {
        case Main.TEST1_NAME:
        case Main.TEST2_NAME:

```

According to the author's solution, a new instance is created, and its contents are listed via the **getInfo()** method.

```

try {
    correct = new MySolution(svalue1, svalue2, ivalue1, ivalue2);
    result_correct = correct.getInfo();
} catch (Exception e) {result_correct = e.toString();}

```

A new instance is created according to the student solution, and its contents are listed via the **getInfo()** method.

```

try {
    tested = new Student(svalue1, svalue2, ivalue1, ivalue2);
    result_test = tested.getInfo();
} catch (Exception e) {result_test = e.toString();}
break;
}
} catch (Exception e) {
    result_test = "ERROR: " + e.toString();
}

```

The results are compared, and the system message is generated.

```

if (!result_test.equals(result_correct)) {
    output = output + "!--- result: " + TEXT_ERROR + "\n";
} else {
    output = output + "!--- result: OK\n";
    correct_count++;
}

output += "!--- " + TEXT_INPUT_VALUES + ": \n"
+ "values: "+ svalue1 + ", "+ svalue2 + ", "+ ivalue1 + ", "+ ivalue2;
output += "\n" + "!--- " + TEXT_PROGRAM_OUTPUT + ": \n"
+ result_test + " \n"
+ "!--- " + TEXT_EXPECTED_OUTPUT + ": \n"
+ result_correct + " \n";
}

```

The output and information about successful attempts are returned.

```

return "" + correct_count + ";" + output;
}
}

```

## Scripts

Customized scripts ensure that the correct classes are selected and run and that output is generated.

### vpl\_run.sh

```
#!/bin/bash
#load common script and check programs

. common_script.sh
check_program javac
check_program java
get_source_files java

#compile all .java files

export CLASSPATH=$CLASSPATH:/usr/share/java/junit4.jar
javac -Xlint:deprecation *.java

if [ "$?" -ne "0" ] ; then
    echo "Not compiled"
    exit 0
fi

cat common_script.sh > vpl_execution
echo "java -enableassertions -cp $CLASSPATH:/usr/share/java/junit4.jar
org.junit.runner.JUnitCore MyTest" >> vpl_execution
chmod +x vpl_execution
```

### vpl\_evaluate.sh

```
#!/bin/bash
#load common script and check programs

. common_script.sh
check_program javac
check_program java
get_source_files java

#compile all .java files

export CLASSPATH=$CLASSPATH:/usr/share/java/junit4.jar
javac -Xlint:deprecation *.java

if [ "$?" -ne "0" ] ; then
    echo "Not compiled"
    exit 0
fi

cat common_script.sh > vpl_execution
echo "java -enableassertions -cp $CLASSPATH:/usr/share/java/junit4.jar Main" >>
vpl_execution
chmod +x vpl_execution
```

## Practical Info – xUnit testing

XUnit is a typical tool representing libraries for testing elementary parts of a program. X- in the name means that it is a tool designed for various programming languages. E.g. JUnit is used for Java, CUnit is used for C, PHPUnit is used for PHP, etc.

The principle of working with xUnit libraries is similar in all environments. We illustrate the use of JUnit for testing Java applications in the two most used development environments - IntelliJ and Eclipse.

This part is aimed to presents methods and principles of xUnit testing. Specific environments are presented so that the reader can test the functionality and principle of assert methods in specific objects and parts of the application. It is expected that the presented procedures can be subsequently applied in situations aimed at testing student codes.

### JUnit

**JUnit** is a simple open-source framework for Java source testing ([www.junit.org](http://www.junit.org)). It is intended to verify if a piece of code works as is expected. It uses principles based on the comparison of expected and obtained outputs.

JUnit is a special tool that allows writing Java tests using a simple interface.

It can test functions, methods, classes, packages, subsystems and supports automated testing.

JUnit is intended to run tests that have already been prepared - after editing code and making changes.

#### Task:

Create a class Calculator with methods:

sum - adds two integers obtained as parameters and returns the result as an integer,

multi - multiplies two integers obtained as parameters and returns the result as an integer,

To create a new class is easy - we create it in a new project:

```
public class Calculator {  
  
    public int sum(int a, int b) {  
        return a + b;  
    }  
  
    public int multi(int a, int b) {  
        return a * b;  
    }  
}
```

To add tests, we can proceed in several ways:

- we can place the test files directly into a package with code (it is not the best solution)

- we can create a separate group for tests.

## Testing in IntelliJ

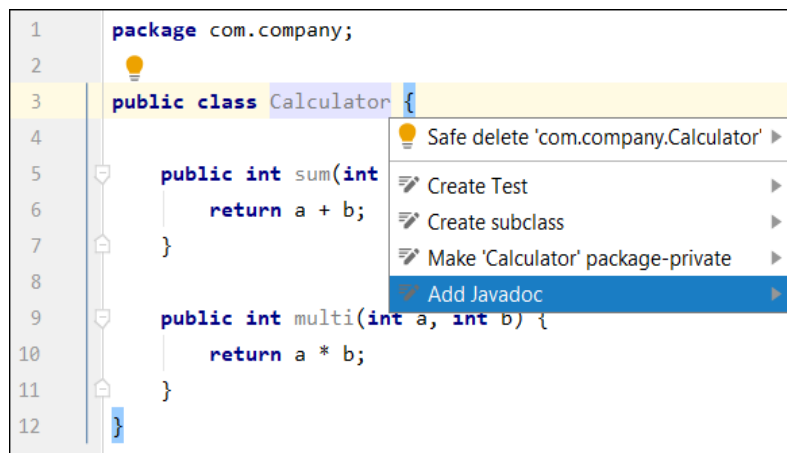
The libraries for JUnit are shipped with IntelliJ IDEA but are not included in the classpath of your project or module by default.

To add the necessary library to the classpath, you can use the general procedure of adding a dependency to a module. The corresponding libraries are in the following directories:

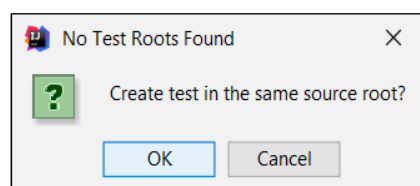
- JUnit libraries (**hamcrest-core-1.3.jar** and **junit-4.12.jar**): <IntelliJ IDEA directory>\lib.

IntelliJ IDEA can add the necessary library to the classpath automatically. The corresponding features are available when creating a test for a class or writing the code for a test.

To initialise tests, we should press **Alt+Enter** in the name of our class (or select **Show content action** in the popup menu).

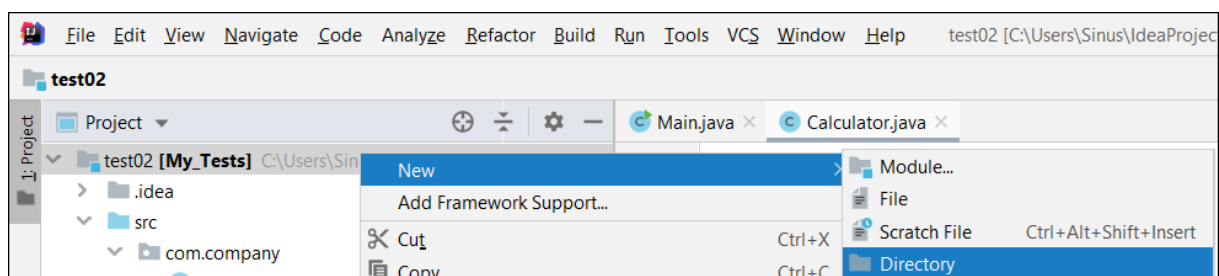


The warning after **Create test** option selection is that the application doesn't have a place for roots

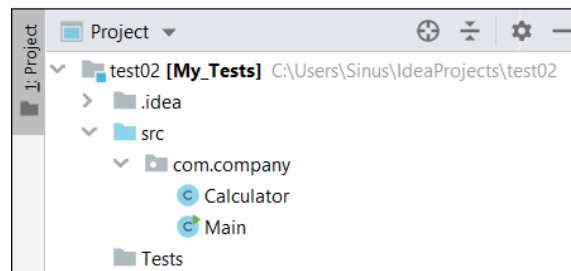


We cancel dialogue, and:

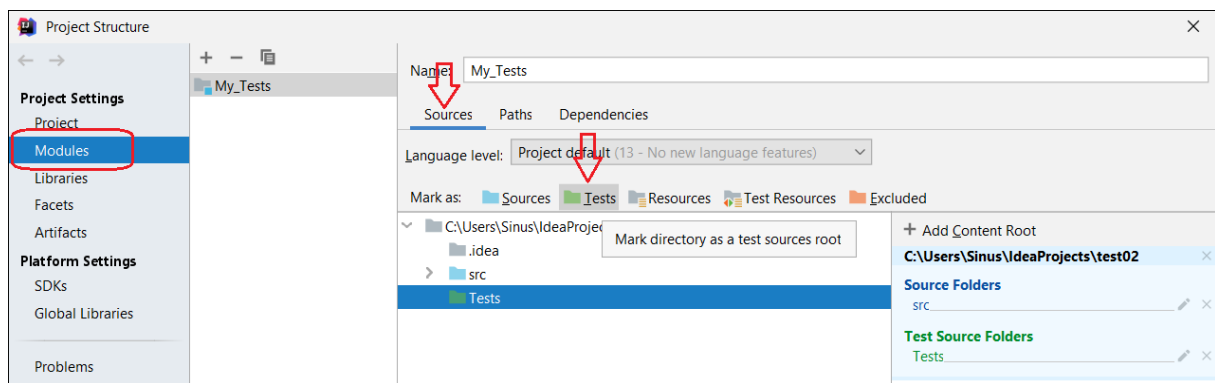
We create a new folder in the project structure (e.g. **Tests**)



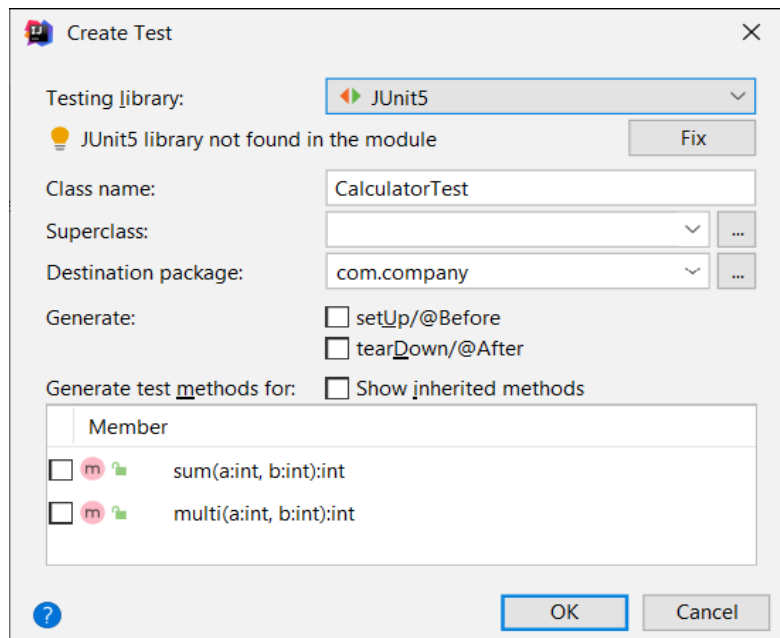
The result is a new folder:



We mark this created folder in project structure (**File -> Project structure**) in the **Module Group** on tab **Sources** to **Tests**:

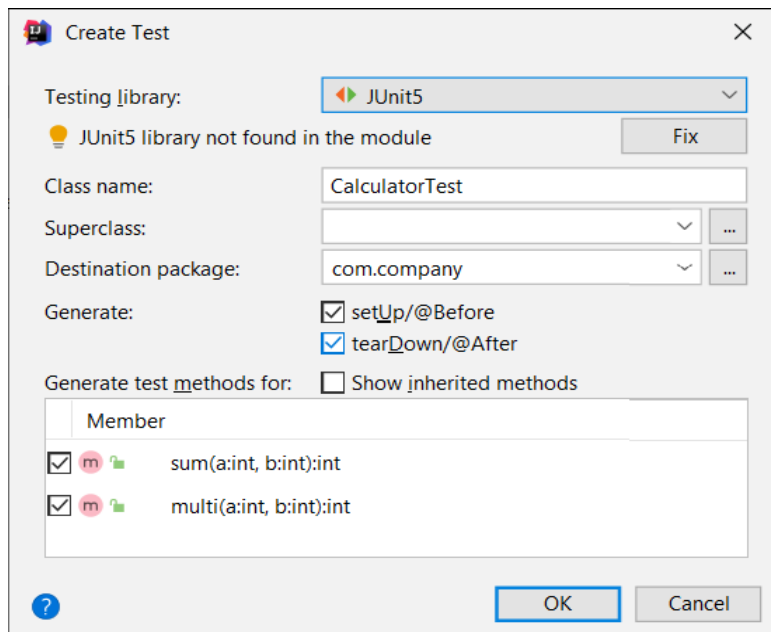


The next use of **Alt + Enter** opens a window for test parameters settings.



If we prepare the first test, we probably need to install the library and use the **Fix** button to solve the actual situation.

After install (the window is still opened), we set the following selections:



The result of the test dialogue activity brings some code:

```
package com.company;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {

    @BeforeEach
    void setUp() {
    }

    @AfterEach
    void tearDown() {
    }

    @Test
    void sum() {
    }

    @Test
    void multi() {
    }
}
```

Every test class consists of optional parts defined by notations (the method names are not important):

- **@BeforeEach** - must be performed **before each test** in the class to set the parameters needed for the test,
- **@AfterEach** - must be performed **after each test** in the class (e.g. reset parameters, etc.),
- **@Test** - the method of testing itself

The program has to work every time with an independent instance. We can achieve it with a new independent calculator created before every test.

We use the **@BeforeEach** notation method:

```
class CalculatorTest {
    Calculator calc;

    @BeforeEach
    void setUp() {
        calc = new Calculator();
    }
}
```

The use of the method with **@AfterEach** notation is not necessary. We can let it empty.

The tests are realised in methods with notation **@Test**. We can prepare its content ourselves, but the idea of test writing is to prepare easily understandable code - we use the methods in test class for testing in methods with the same name.

The most commonly used method for testing is the **AssertEquals** method, which compares the expected value with the result obtained from the tested class.

```
assertEquals(5, calc.sum(2, 3));
```

- the first parameter is the expected value
- the second parameter is the value obtained as a result of the test class

The code with tests for sum has the following form:

```
class CalculatorTest {
    Calculator calc;

    @BeforeEach
    void setUp() {
        calc = new Calculator();
    }

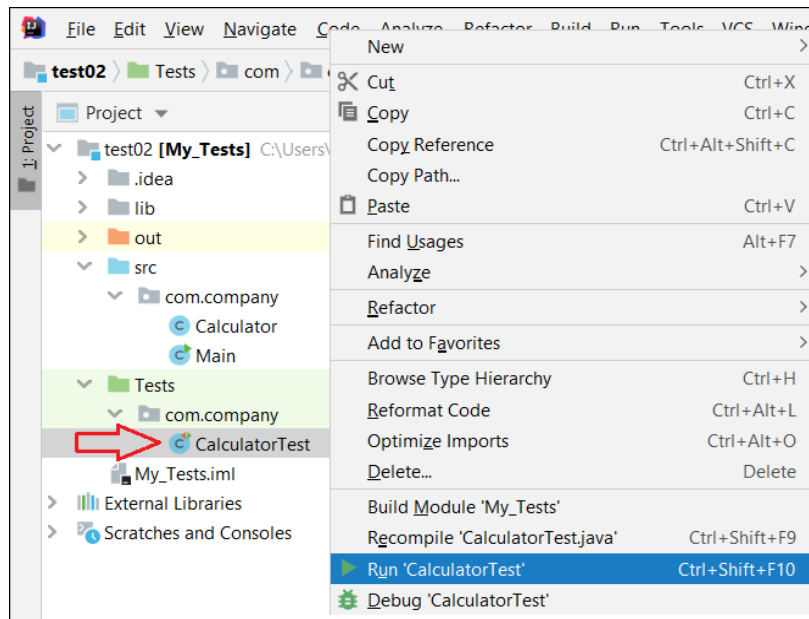
    @AfterEach
    void tearDown() {
    }

    @Test
    void sum() {
        assertEquals(5, calc.sum(2, 3));
        assertEquals(-3, calc.sum(-8, 5));
    }
}
```

We can add new test for method multi():

```
@Test
void multi() {
    assertEquals(6, calc.multi(2, 3));
    assertEquals(-40, calc.multi(-8, 5));
}
```

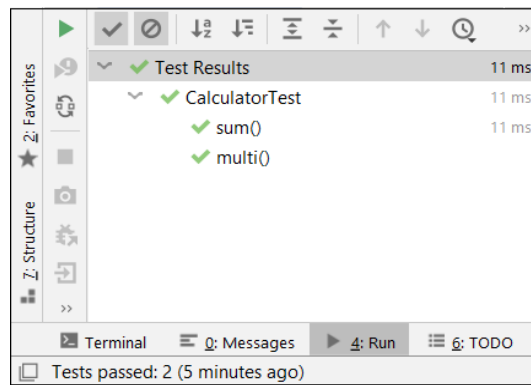
To run the created test, you have to start it as follow:



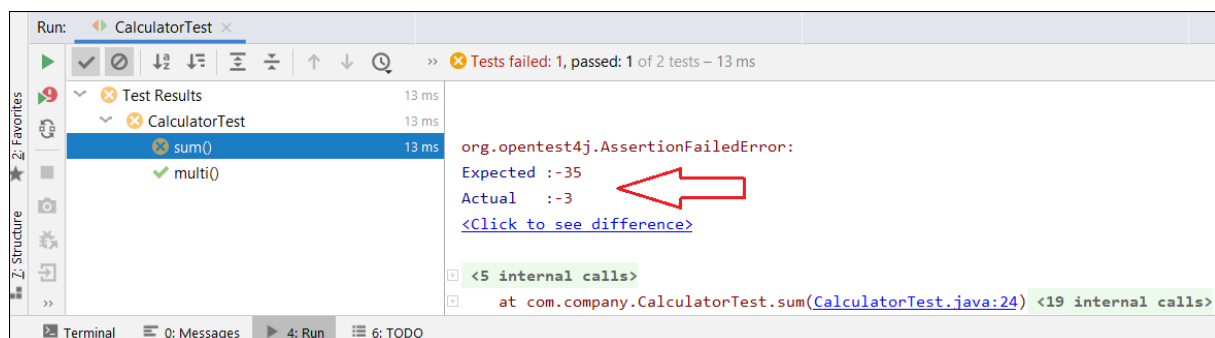
Using the context menu, you can **Run CalculatorTest**. Or you can run the application using the button on the toolbar.

The result of the run should be:

- all tests were passed successfully:



- some tests were failed:



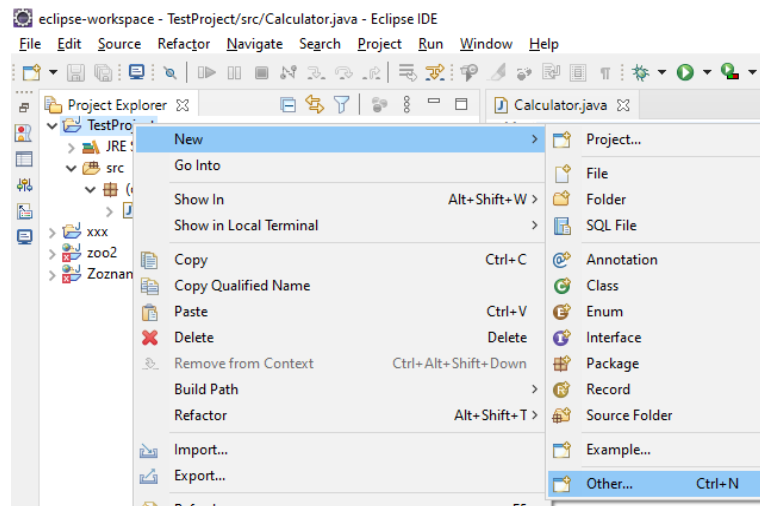
The environment shows you:

- expected value: value what user wrote as expected value of the tested function
- actual value: the value returned by the tested function

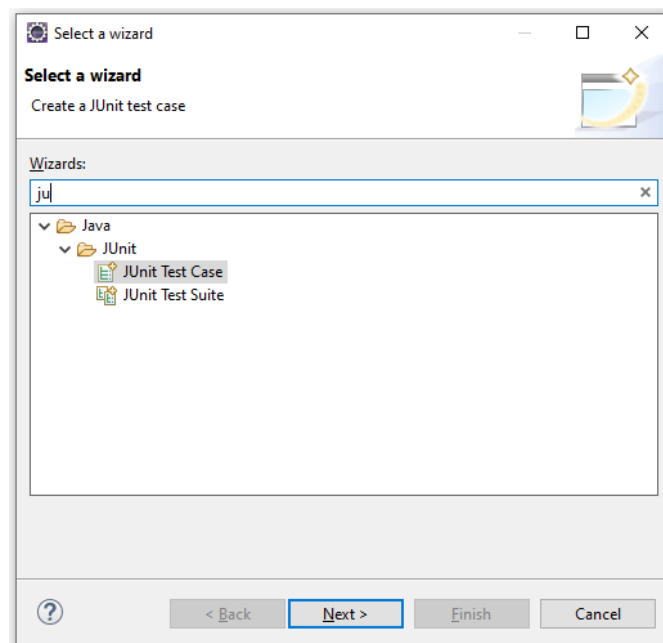
## Testing in Eclipse

The libraries for JUnit are integrated into Eclipse.

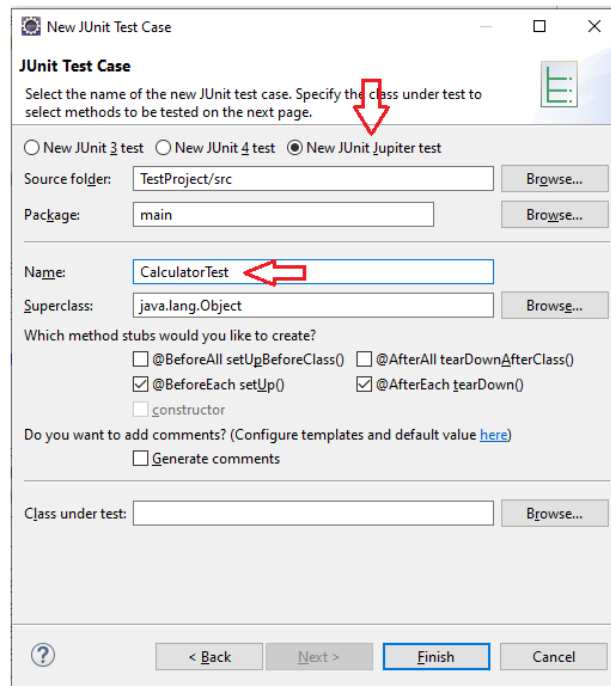
To add a new test case, you can select **New -> Other...** in the context menu of the project.



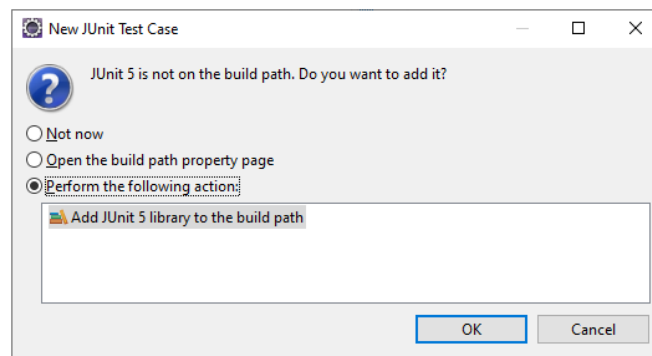
Select **JUnit Test Case** in the opened window...



... and set the name and package for the test class. You can add some methods of test classes too.



For using the JUnit library, you should add it to the project. Select **OK**.



The result of the test dialogue activity brings the short code:

```
package main;

import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {

    @BeforeEach
    void setUp() throws Exception {
    }

    @AfterEach
    void tearDown() throws Exception {
    }

    @Test
    void test() {
        fail("Not yet implemented");
    }

}
```

Every test class consists of optional parts defined by notations (the method names are not important):

- **@BeforeEach** - must be performed **before each test** in the class to set the parameters needed for the test,
- **@AfterEach** - must be performed **after each test** in the class (e.g. reset parameters, etc.),
- **@Test** - the method of testing itself

The program has to work every time with an independent instance. We can achieve it with a new independent calculator created before every test.

We use **@BeforeEach** notation method:

```
class CalculatorTest {
    Calculator calc;

    @BeforeEach
    void setUp() throws Exception {
        calc = new Calculator();
    }
}
```

The use of the method with **@AfterEach** notation is not necessary. We can let it empty.

The tests are realised in methods with notation **@Test**. We can prepare its content ourselves, but the idea of test writing is to prepare easily understandable code - we use the methods in test class for testing in methods with the same name – we create the method **sum()** with notation **@Test**.

The most commonly used method for testing is the **AssertEquals** method, which compares the expected value with the result obtained from the tested class.

```
assertEquals(5, calc.sum(2, 3));
```

- the first parameter is the expected value
- the second parameter is the value obtained as a result of the test class

The code with tests for **sum()** has the following form:

```
class CalculatorTest {
    Calculator calc;

    @BeforeEach
    void setUp() throws Exception {
        calc = new Calculator();
    }

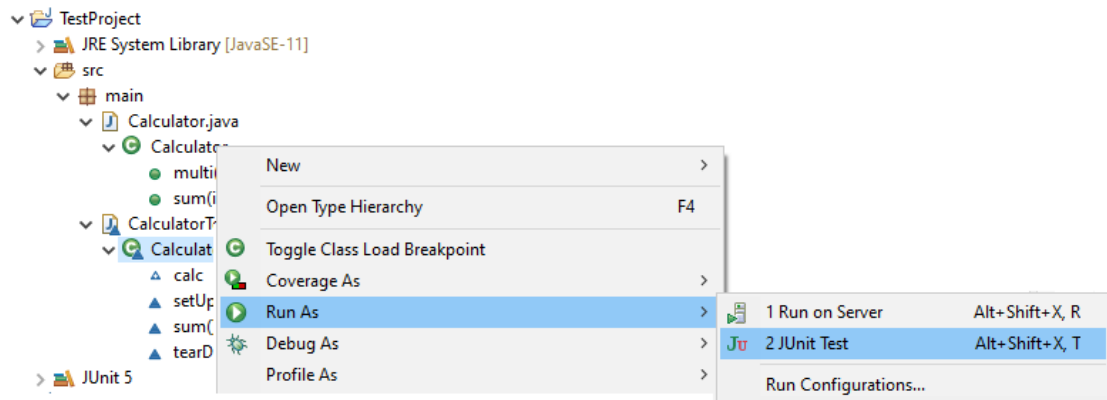
    @AfterEach
    void tearDown() throws Exception {
    }

    @Test
    void sum() {
        assertEquals(5, calc.sum(2, 3));
        assertEquals(-3, calc.sum(-8, 5));
    }
}
```

We can add a test for method **multi()**.

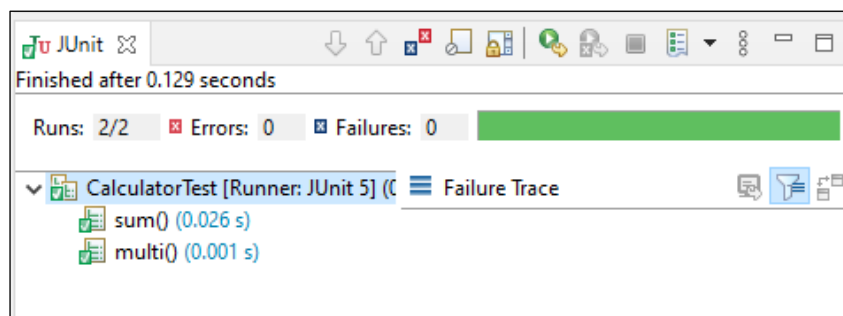
```
@Test
void multi() {
    assertEquals(20, calc.multi(2, 10));
    assertEquals(40, calc.multi(8, 5));
}
```

To run created tests, you have to start it using the context menu **CalculatorTest - Run as - JUnit Test**.

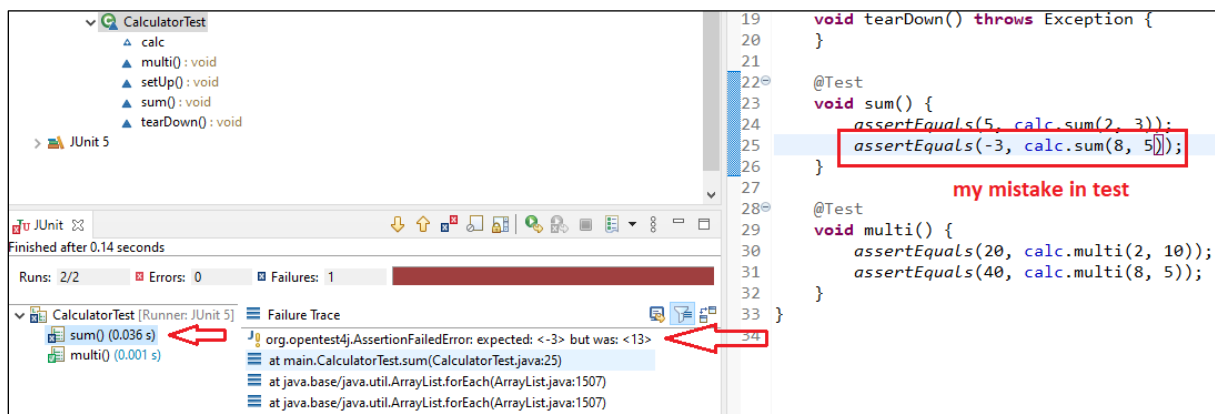


The result of the run should be in two forms:

- all tests were passed successfully:



- some tests failed:



The environment shows you:

- expected value: value what user wrote as expected value of the tested function
- actual value: the value returned by the tested function

## Finish the test

The following lines are independent of the used environment. Let's go to summarise our tests. We should add new asserts for the tested method – we should use some critical or limit values:

```
class CalculatorTest {
    Calculator calc;

    @BeforeEach
    void setUp() throws Exception {
        calc = new Calculator();
    }

    @AfterEach
    void tearDown() throws Exception {
    }

    @Test
    void sum() {
        assertEquals(5, calc.sum(2, 3));
        assertEquals(-3, calc.sum(-8, 5));
        assertEquals(0, calc.sum(0, 0));
        assertEquals(5, calc.sum(0, 5));
        assertEquals(-5, calc.sum(0, -5));
        assertEquals(10, calc.sum(-5, -5));
        assertEquals(0, calc.sum(-1, 1));
        assertEquals(2000000, calc.sum(1000000, 1000000));
    }

    @Test
    void multi() {
        assertEquals(8, calc.multi(2, 4));
        assertEquals(40, calc.multi(-8, -5));
        assertEquals(0, calc.multi(0, -5));
        assertEquals(-25, calc.multi(5, -5));
        assertEquals(1, calc.multi(-1, -1));
        assertEquals(1, calc.multi(1, 1));
        assertEquals(5000000, calc.multi(1000, 5000));
    }
}
```

The final test shows us that our tests passed, and we have **probably** the correct code.

If the expected value differs from the obtained value, an error (**AssertionError**) is generated, usually with a message why it is unsuccessful:

```
org.opentest4j.AssertionFailedError:
Expected :1
Actual   :0
<Click to see difference>

    at org.junit.jupiter.api.AssertionUtils.fail(AssertionUtils.java:55)
    at org.junit.jupiter.api.AssertEquals.failNotEqual(AssertEquals.java:195)
```

```
at org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:152)
at org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:147)
```

It should be noted that the system cannot identify that this is a tester error, and if the tester makes a mistake when entering the expected value, the test tool attributes a program error.

It should be noted that the test lists only a mismatch between the expected and the obtained value.

## Accuracy in tests

### Real numbers in test

How to modify the previous program and tests to use decimal/real numbers?

The original task was to create a **Calculator** class with methods:

- **sum** - adds two integers obtained as parameters and returns the result as an integer,
- **multi** - multiplies two integers obtained as parameters and returns the result as an integer,

The modified class has the following form:

```
public class Calculator {

    public double sum(double a, double b) {
        return a + b;
    }

    public double multi(double a, double b) {
        return a * b;
    }
}
```

The accuracy of real numbers processing is often a problematic part of calculation using digital computers.

According to this fact, the tests used in programming support the parameter accuracy as the third parameter in **assertEquals**, e.g.:

```
assertEquals(8, calc.multi(2, 4), 0.001);
```

If the result of the inspected method and the result set by the tester are diverged by less than the specified accuracy, the test is passed.

So, we prepare the tests for the modified **Calculator**:

```
class CalculatorTest {
    Calculator calc;

    @BeforeEach
    void setUp() {
        calc = new Calculator();
    }
}
```

```

@AfterEach
void tearDown() {
}

@Test
void sum() {
    assertEquals(5.5, calc.sum(2.5, 3), 0.0001);
    assertEquals(-3.01, calc.sum(-8.01, 5), 0.0001);
}

@Test
void multi() {
    assertEquals(6.25, calc.multi(2.5, 2.5), 0.0001);
    assertEquals(40, calc.multi(-8.0001, -5), 0.01);
    assertEquals(0, calc.multi(0, -5.99), 0.0001);
}
}

```

Look at to the second test in multiplication:

```

@Test
void multi() {
    assertEquals(6.25, calc.multi(2.5, 2.5), 0.0001);
    assertEquals(40, calc.multi(-8.0001, -5), 0.01);
    assertEquals(0, calc.multi(0, -5.99), 0.0001);
}
}

```

Even though the multiplication result is 40.0005, the test ignores the difference between the expected result (40) and the actual result. The difference is considered to be irrelevant because the acceptable deviation is 0.01.

Be careful:

if we allow big inaccuracy, the system will pass this test as well:

```
assertEquals(-5, calc.sum(-8, 5), 3);
```

## Exceptions

### Division by zero

Enrich the class **Calculator** to the division and solve the situation with division by zero.

We can add a new method to get quotient:

```

public class Calculator {

    public double sum(double a, double b) {
        return a + b;
    }

    public double multi(double a, double b) {
        return a * b;
    }
}

```

```
public double quotient(double a, double b) {
    if (b!=0)
        return a / b;
    }
}
```

The result of the new method is decimal value - it is defined via the type of method.

Therefore, returning some text in the form of "do not divide by zero" is quite problematic.

But, we can create a “managed” exception that can be caught and handled in the code using the Calculator.

The exception generation is a common method in work with classes and methods. We can apply it using keyword **throw**.

```
public double quotient(double a, double b) {
    if (b!=0)
        return a / b;
    else
        throw new IllegalArgumentException("zero division");
}
```

This exception is caught in application using:

```
public static void main(String[] args) {
    Calculator calc = new Calculator();

    try {
        calc.quotient(4,0);
    } catch (Exception e) {
        System.out.println(e.toString());
    }
}
```

The output is:

```
java.lang.IllegalArgumentException: zero division
```

Exceptions testing is important and common. This type of testing needs a special method to process the returned exceptions. The method **assertThrows** is used to assert that the supplied executable will throw an exception of the **expectedType**. If there is no exception of **expectedType**, the method will fail.

The definition of the **assertThrow()** consists of two parameters:

```
public static void assertThrows(Class<? extends Throwable> expectedType, Executable executable)
```

The second part is defined as executable. We can use the lambda notation and set this parameter as

```
() -> method()
```

The final code has following form:

```
assertThrows(IllegalArgumentException.class, () -> calc.quotient(2, 0));
```

When writing tests, keep in mind that we separately test the values for the standard result and the values giving the exception.

```
@Test
public void quotient_common() {
    assertEquals(2, calc.quotient(2, 1));
    assertEquals(2, calc.quotient(3, 1.5), 0.0001);
}

@Test
public void quotient_exception() {
    assertThrows(IllegalArgumentException.class, () -> calc.quotient(2, 0));
}
```

## Testing methods

In addition to the methods described above, we also have others available:

- **assertArrayEquals()** - return **true** if two arrays contain the same elements
- **assertNotEquals()** - return **true** if two values are **not** the same
- **assertNotNull()** - return **true** if the value is not null
- **assertNotSame()** - return **true** if two references don't address to the same object
- **assertSame()** - return **true** if two references address to the same object (compares with ==)
- **assertTrue()** - return **true** if result of expression or method is true
- **assertFalse()** - return **true** if result of expression or method is false

Example of assertions:

```
public class TestAssertions {
    @Test
    public void testAssertions() {
        String str1 = new String ("abc"), str2 = new String ("abc");
        String str3 = null;
        String str4 = "abc", str5 = "abc";

        int val1 = 5, val2 = 6;
        String[] expectedArray = {"one", "two", "three"};
        String[] resultArray = {"one", "two", "three"};
        //Check that two objects are equal
        assertEquals(str1, str2);
        //Check that a condition is true
        assertTrue (val1 < val2);
        //Check that a condition is false
        assertFalse(val1 > val2);
        //Check that an object isn't null
        assertNotNull(str1);
        //Check that an object is null
        assertNull(str3);
        //Check if two object references point to the same object
        assertSame(str4, str5);
        //Check if two object references not point to the same object
        assertNotSame(str1, str3);
        //Check whether two arrays are equal to each other.
```

```
    assertEquals(expectedArray, resultArray);  
  }  
}
```

## Bibliography

1. Vargas-Llave, O.; Mandl, I.; Weber, T.; Wilkens, M. Telework and ICT-based mobile work: Flexible working in the digital age, New forms of employment series, Publications Office of the European Union, Luxembourg, 2020, doi:10.2806/337167.
2. Kogan, M.; Klein, S. E.; Hannon, C. P.; Nolte, M. T. Orthopaedic education during the COVID-19 pandemic. *The Journal of the American Academy of Orthopaedic Surgeons*, Vol. 28, No. 11, 2020.
3. Gibson, A.; Bardach, S. H.; Pope, N. D. COVID-19 and the Digital Divide: Will Social Workers Help Bridge the Gap?, *Journal of Gerontological Social Work*, 2020, doi: 10.1080/01634372.2020.1772438.
4. Henriksen, D.; Mishra, P.; Fisser, P. Infusing Creativity and Technology in 21st Century Education: A Systemic View for Change. *Journal of Educational Technology & Society*, 2016, Vol. 19, No. 3 (July 2016), pp. 27-37.
5. Skalka, J.; Drlik, M. Automated Assessment and Microlearning Units as Predictors of At-Risk Students and Students' Outcomes in the Introductory Programming Courses. *Appl. Sci.* 2020, 10, 4566.
6. Tabanao, E. S.; Rodrigo, M. M. T.; Jadud, M. C. Predicting at-risk novice Java programmers through the analysis of online protocols. In *Proceedings of the Seventh International Workshop on Computing Education Research, ICER 2011*, Providence, RI, USA, 8–9 August 2011, doi:10.1145/2016911.2016930.
7. Kinnunen, P.; Malmi, L. Why Students Drop out CS1 Course? In *Proceedings of the Second International Workshop on Computing Education Research*, New York, NY, USA, 10–12 September 2006; pp. 97–108, doi:10.1145/1151588.1151604.
8. Othman, J.; Wahab, N. A. The Uncommon Approaches of Teaching the Programming Courses: The Perspective of Experienced Lecturers. *Computing Research & Innovation (CRINN)*, Vol. 1, November 2016.
9. Chen, Y.; Zhang, M. MOOC student dropout: Pattern and prevention. In *Proceedings of the ACM Turing 50th Celebration Conference-China*, 2017. pp. 1-6.
10. Skalka, J.; Drlik, M. Priscilla—Proposal of System Architecture for Programming Learning and Teaching Environment, *IEEE 12th International Conference on Application of Information and Communication Technologies (AICT)*. IEEE, 2018, pp. 1-6.
11. Fuller, U.; Johnson, C. G.; Ahoniemi, T.; Cukierman, D.; Hernán-Losada, I.; Jackova, J.; Lahtinen, E.; Lewis, T. L.; Thompson, D. M.; Riedesel, C.; Thompson, E. Developing a Computer Science-specific Learning Taxonomy, *ACM SIGCSE Bulletin*, 2007, Vol 37, No. 4, pp. 152-170.
12. Skalka, J.; Drlik, M. Educational Model for Improving Programming Skills Based on Conceptual Microlearning Framework. In *The Challenges of the Digital Transformation in Education. ICL 2018. Advances in Intelligent Systems and Computing*, vol. 916. Springer 2020.
13. Becker, B. A.; Quille, K. 50 years of CS1 at SIGCSE: A review of the evolution of introductory programming education research. In *Proceedings of the 50th ACM technical symposium on computer science education*, 2019. pp. 338-344.
14. Medeiros, R. P.; Ramalho, G. L.; Falcao, T. P. (2018). A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher Education. *IEEE Transactions on Education*, 2018, pp. 1–14. doi:10.1109/te.2018.2864133.
15. Murphya, E.; Crick, T.; Davenportc, J. H. An Analysis of Introductory Programming Courses at UK Universities. *The Art, Science, and Engineering of Programming*, 2017, 1(2), 23.
16. Queiros, R.; Pinto, M.; Terroso, T. Computer Programming Education in Portuguese Universities. In *First International Computer Programming Education Conference (ICPEC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
17. Krinke, J.; Storzer, M.; Zeller, A. Web-basierte Programmierpraktikamit Praktomat in Workshop Neue Medien in der Informatik-Lehre, Dortmund, 2002, pp. 48–56.
18. Morth, T.; Oechsle, R.; Schloß, H.; Schwinn, M. Automatische Bewertung studentischer Software in Workshop "Rechnerunterstütztes Selbststudium in der Informatik", Universität Siegen, 2007.
19. Edwards, S.; Perez-Quinones, M. A. Web-CAT: Automatically Grading Programming Assignments in *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*. ACM, New York, 2008. pp. 328 – 338.
20. Hoernecke, J.; Amelung, M.; Krieger, K.; Rosner, D. Flexibles E-Assessment mit OLAT und ECSpooler. In: Rohland, H., Kienle, A. & Friedrich, S. (Hrsg.), *DeLFI 2011 - Die 9. e-Learning Fachtagung Informatik*. Bonn: Gesellschaft für Informatik, pp. 127-138.
21. Striewe, M.; Goedicke, M.; Balz, M. Computer Aided Assessmentsand Programming Exercises with JACK, No. 28, *Institut für Informatik und Wirtschaftsinformatik (ICB)*, University of Duisburg-Essen, 2008.

22. Hass, B.; Yuan, C.; Li, Z. On the Automatic Assessment of Learning Outcome in Programming Techniques, IEEE 14th International Conference on Intelligent Systems and Knowledge Engineering (ISKE), Dalian, China, 2019, pp. 274-278, doi: 10.1109/ISKE47853.2019.9170370.
23. Agbo, F. J.; Oyelere, S. S.; Suhonen, J.; Adewumi, S. A systematic review of computational thinking approach for programming education in higher education institutions. In Proceedings of the 19th Koli Calling International Conference on Computing Education Research, 2019. pp. 1-10.
24. Caceffo, R.; Wolfman, S.; Booth, K.S.; Azevedo, R. Developing a Computer Science Concept Inventory for Introductory Programming. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16). Association for Computing Machinery, New York, USA, 2016, pp. 364–369, doi: 10.1145/2839509.2844559
25. Skalka, J.; Drlik, M.; Obonya, J. Automated Assessment in Learning and Teaching Programming Languages using Virtual Learning Environment. In IEEE Global Engineering Education Conference (EDUCON), IEEE., 2019. pp. 689-697
26. Parihar, S.; Dadachanji, Z.; Singh, P. K.; Das, R.; Karkare, A.; Bhattacharya, A. Automatic grading and feedback using program repair for introductory programming courses. In Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education, 2017, pp. 92-97.
27. Gao, J.; Pang, B.; Lumetta, S. S. Automated feedback framework for introductory programming courses. In Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education, 2016. pp. 53-58.
28. Batista, A. L. F.; Connolly, T.; Angotti, J. A. P. A framework for games-based construction learning: A text-based programming languages approach. In European Conference on Games Based Learning. Academic Conferences International Limited., 2016, p. 815
29. Duffany, J. L. Application of active learning techniques to the teaching of introductory programming. IEEE Revista Iberoamericana de Tecnologías del Aprendizaje, 2017, 12(1), pp. 62-69
30. Schez-Sobrinho, S.; Gomez-Portes, C.; Vallejo, D.; Glez-Morcillo, C.; Redondo, M. A. An Intelligent Tutoring System to Facilitate the Learning of Programming through the Usage of Dynamic Graphic Visualisations. Appl. Sci. 2020, 10, 1518.
31. Djelil, F.; Albouy-Kissi, A.; Albouy-Kissi, B.; Sanchez, E.; Lavest, J. M. Microworlds for learning Object-Oriented Programming: Considerations from research to practice. Journal of Interactive Learning Research, 2016, 27(3), pp. 247-266.
32. Karagiannis, I.; Satratzemi, M. Enhancing Adaptivity in Moodle: Framework and Evaluation Study. In: Auer M., Guralnick D., Uhomoibhi J. (eds) Interactive Collaborative Learning. ICL 2016., Advances in Intelligent Systems and Computing, vol 545. Springer, Cham, 2016. doi: 10.1007/978-3-319-50340-0\_52.
33. Rodríguez-del-Pino, J. C.; Rubio-Royo, E.; Hernández-Figueroa, Z. Virtual Programming Lab for Moodle with automatic assessment and anti-plagiarism features, Proceedings of the International Conference on e-Learning, e-Business, Enterprise Information Systems, and e-Government (EEE), The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012.
34. Vesin, B.; Mangaroska, K.; Giannakos, M. Learning in smart environments: user-centered design and analytics of an adaptive learning system. Smart Learning Environments, 5(1), 24. 2018.
35. Brusilovsky, P.; Malmi, L.; Hosseini, R.; Guerra, J.; Sirkiä, T.; Pollari-Malmi, K. An integrated practice system for learning programming in Python: design and evaluation. RPTel 13, 18 (2018). doi: 10.1186/s41039-018-0085-9
36. Buffardi, K.; Edwards, S. H. Introducing CodeWorkout: an adaptive and social learning environment. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education, 2014. pp. 724-724.
37. Wood, K. Top 25 Websites to Learn Coding for Free. Hostinger tutorials. on-line: <https://www.hostinger.com/tutorials/learn-coding-online-for-free/>, 2020.
38. Morris, S. 80+ Ways to Learn to Code for Free in 2020. skillcrush.com. on-line: <https://skillcrush.com/blog/64-online-resources-to-learn-to-code-for-free/>, 2020.
39. Hadjerrouit, S. Towards a blended learning model for teaching and learning computer programming: A case study. Informatics in Education-An International Journal, 2008, 7(2), pp. 181-210.
40. Zhang, J. An adaptive model customised for programming learning in e-learning. In 2010 3rd International Conference on Computer Science and Information Technology (Vol. 6). IEEE., 2010. pp. 443-447.
41. Bashir, G.M.M.; Hoque, A.S.M.L. An effective learning and teaching model for programming languages. Journal of Computers in Education, 3 (2016). pp. 413–437, doi: 10.1007/s40692-016-0073-2.
42. Chen, G. M. Programming Language Teaching Model Based on Computational Thinking and Problem-based Learning. In 2nd International Seminar on Education Innovation and Economic Management (SEIEM 2017). Atlantis Press., 2017. doi: 10.2991/seiem-17.2018.31.

43. Othman, M.; Othman, M.; Hussain, F. M. Designing prototype model of an online collaborative learning system for introductory computer programming course. *Procedia-Social and Behavioral Sciences*, 2013, 90, pp. 293-302.
44. Skalka, J.; Drlik, M. Conceptual framework of microlearning-based training mobile application for improving programming skills. In *Interactive Mobile Communication, Technologies and Learning*. Springer, Cham, 2018. pp. 213-224.
45. Appiahene, P.; Asante, G.; Kesse-Yaw, B.; Acquah-Hayfron, J. Raising students programming skills using appiahene gamification model. In *ECGBL 11th European Conference on Game-Based Learning*. Academic Conferences and publishing limited., 2017. pp. 14-21.
46. Khaleel, F. L.; Ashaari, N. S.; Wook, T. S. M. T.; Ismail, A. Methodology for developing gamification-based learning programming language framework. In *6th International conference on electrical engineering and informatics (ICEEI)*. IEEE, 2017. pp. 1-6.
47. Rojas-López, A.; Rincón-Flores, E. G.; Mena, J.; García-Peñalvo, F. J.; Ramírez-Montoya, M. S. Engagement in the course of programming in higher education through the use of gamification, *Universal Access in the Information Society*, 18(3), 2019. pp. 583-597.
48. Kordaki, M. A drawing and multi-representational computer environment for beginners' learning of programming using C: Design and pilot formative evaluation, *Computers & Education*, 2010, Vol. 54, No. 1, pp. 69-87.
49. Lee, D. M. C.; Rodrigo, M. M. T.; Baker, R. S. J. d.; Sugay, J. O.; Coronel, A. Exploring the Relationship between Novice Programmer Confusion and Achievement, In: *Affective Computing and Intelligent Interaction*. ACII 2011. Lecture Notes in Computer Science, vol 6974. Springer, Berlin, Heidelberg, pp. 175-184.
50. Krusche, S.; Seitz, A. ArTEMiS - An Automatic Assessment Management System for Interactive Learning, *SIGCSE '18 Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 2018. pp. 284-289.
51. Ambrosio, A. P.; Costa, F. M.; Almeida, L.; Franco, A.; Macedo, J. Identifying cognitive abilities to improve CS1 outcome, *Frontiers in Education Conference (FIE)*, 2011. IEEE, pp. F3G1-F3G7.
52. Ciancarini, P.; Missiroli, M.; Russo, D. Cooperative Thinking: Analysing a new framework for software engineering education. *Journal of Systems and Software*, 2019, 157, 110401.
53. López-Fernández, D.; P., Alarcón P.; Tovar, E. Motivation in engineering education a framework supported by evaluation instruments and enhancement resources. In *IEEE Global Engineering Education Conference (EDUCON)*. IEEE, 2015. pp. 421-430.
54. Tovar, E.; Soto, O. Are new coming computer engineering students well prepared to begin future studies programs based on competences in the European Higher Education Area?. *Frontiers in Education Conference*, 2009. FIE'09. 39th IEEE, pp. 1-6.
55. Bekki, J. M.; Dalrymple, O.; Butler, C. S. A mastery-based learning approach for undergraduate engineering programs. *Frontiers in Education Conference (FIE)*, IEEE, 2012, pp. 1-6.
56. Bloom, B. S. *Taxonomy of Educational Objectives. Cognitive domain*, 1956, Vol. 1, pp. 20-24.
57. Anderson, L. W.; Bloom, B. S. *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives*. Longman, 2001.
58. Zufic, J.; Jurcan, B. Micro learning and EduPsy LMS. In *Central European Conference on Information and Intelligent Systems*. Faculty of Organization and Informatics Varazdin., 2015. p. 115.
59. Jones, N. D.; Gomard, C. K.; Sestoft, P. Partial evaluation and automatic program generation. Peter Sestoft, 1993.
60. Selby, R. W.; Porter, A. A. Learning from examples: generation and evaluation of decision trees for software resource analysis. *IEEE Transactions on Software Engineering*, 1988, 14(12), 1743-1757.
61. Daly, C. RoboProf and an introductory computer programming course. *ACM SIGCSE Bulletin*, 1999, 31(3), 155-158.
62. Zheng, J.; Williams, L.; Nagappan, N.; Snipesm, W.; Hudepohl, J.; Vouk, M. On the value of static analysis for fault detection in software. In *IEEE Transactions on Software Engineering*, 2006, 32 (4), pp. 240-253.
63. Kafai, Y. B.; Resnick, They Have Their Own Thoughts: A Story of Constructionist Learning in an Alternative African-Centered Community School. In *Constructionism in Practice*. Routledge. 2012. pp. 259-272.
64. Palloff, R. M.; Pratt, K. *Collaborating online: Learning together in community*, John Wiley & Sons, 2010. Vol. 32
65. Wenger, E. *Communities of practice and social learning systems: the career of a concept*, Social learning systems and communities of practice. Springer, London, 2010. pp. 179-198.
66. Hunicke, R.; LeBlanc, M.; Zubek, R. MDA: A formal approach to game design and game research. In *Proceedings of the AAAI Workshop on Challenges in Game AI 2004*, Vol. 4, No. 1, p. 1722
67. Chou, Y. K. *Actionable gamification: Beyond points, badges, and leaderboards*. Packt Publishing Ltd., 2019.

68. Wise, A. F.; Vytasek, J. M.; Hausknecht, S.; Zhao, Y. Developing Learning Analytics Design Knowledge in the Middle Space: The Student Tuning Model and Align Design Framework for Learning Analytics Use, *Online Learning* 2016, Vol. 20, No. 2, pp. 155-182.
69. Skalka, J.; Drlík, M.; Obonya, J.; Capay, M. Architecture Proposal for Micro-Learning Application for Learning and Teaching Programming Courses. In *IEEE Global Engineering Education Conference (EDUCON)*. IEEE, 2020. pp. 980-987.
70. Brosig, F.; Huber, N.; Kounev, S. Architecture-level software performance abstractions for online performance prediction. *Science of Computer Programming*. 2014, 90(Part B, 0), pp. 71-92.
71. Drlík, M.; Skalka, J.; Švec, P.; Kapusta, J. Proposal of Learning Analytics Architecture Integration into University IT Infrastructure. In *IEEE 12th International Conference on Application of Information and Communication Technologies (AICT)*. IEEE., 2018, pp. 1-6.
72. Munk, M.; Drlík, M.; Benko, L.; Reichel, J. Quantitative and qualitative evaluation of sequence patterns found by application of different educational data preprocessing techniques. *IEEE Access* 2017, 5, pp. 8989-9004.
73. Drlík, M.; Švec, P.; Kapusta, J.; Munk, M.; Noskova, T.; Pavlova, T.; Smyrnova-Trybulska, E. Identification of differences in university e-environment between selected EU and non-EU countries using knowledge mining methods: project IRNet case study. *International Journal of Web Based Communities* 2017, 13(2), pp. 236-261.
74. Halvoník, D.; Kapusta, J. Framework for e-Learning Materials Optimisation. *International Journal of Emerging Technologies in Learning (IJET)*, 2020, 15(11), pp. 67-77.
75. Skalka, J. Data processing methods in the development of the microlearning-based framework for teaching programming languages. In *The 12th international scientific conference on Distance Learning in Applied Informatics, Praha, Wolters Kluwer*, 2018. pp. 503-512.



PRISCILLA



[priscilla.fitped.eu](http://priscilla.fitped.eu)