FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatik

Development of Manual Assessment for Programming Exercises in the Orion Plugin

Martin Dunker

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatik

Development of Manual Assessment for Programming Exercises in the Orion Plugin

Entwicklung manueller Bewertung für Programmieraufgaben im Orion-Plugin

Author:	Martin Dunker
Supervisor:	Prof. Dr. Bernd Brügge
Advisor:	Dr. Stephan Krusche
Date:	September 15, 2021

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, September 15, 2021 Martin Dunker

Abstract

Artemis is a learning platform used to conduct programming courses with programming exercises. Students submit their code and the system automatically runs tests on it. Nonetheless, manual assessment is still necessary to ensure fair grading. Reviewers are currently assessing the code using Artemis's online code editor. While this code editor can show the code, advanced functionality like code completion or debugging is missing. If reviewers wish to execute the students' code, they need to manually download it into their IDE, causing a media disruption.

The same problem is already solved for students with the Orion plugin for Artemis, which integrates Artemis into the IntelliJ IDE, automating the download and import of programming exercises. In this thesis, we extend the plugin to support the manual assessment. The plugin should enable reviewers to automatically download the students' code, making available all features of the IDE. Reviewers can create assessment comments with Orion, enabling them to perform the assessment without leaving the IDE.

Zusammenfassung

Artemis ist eine Lernplatform, mit der Programmierkurse und Programmieraufgaben abgehalten werden. Studenten geben ihren Code ab und das System führt automatisch Tests darauf aus. Nichtsdestotrotz ist eine manuelle Korrektur erforderlich, um eine faire Bewertung zu gewährleisten. Korrigierende bewerten derzeit mittels Artemis' Online-Code-Editor, der den Code zwar anzeigen kann, dem aber erweiterte Funktionen wie Autovervollständigung oder Debugging fehlen. Wenn Korrigierende den Code der Studenten ausführen wollen, müssen sie ihn manuell in ihre IDE herunterladen, was einen Medienbruch verursacht.

Das gleiche Problem ist für Studenten durch das Orion-Plugin für Artemis gelöst, das Artemis in IntelliJ integriert und dabei den Download und Import von Programmieraufgaben automatisiert. In dieser Arbeit erweitern wir das Plugin, damit es auch die manuelle Korrektur unterstützt. Korrigierende sollen mit dem Plugin automatisch den Studentencode herunterladen können, damit alle IDE-Funktionen für ihn verfügbar werden. Sie können Korrekturkommentare in der IDE erzeugen, sodass die Korrektur durchgeführt werden kann, ohne die IDE zu verlassen.

Contents

API Application Programmer Interface

Artemis Automatic Assessment Management System for Interactive Learning

CIS Continuous Integration System

GUI Graphical User Interface

IaaS Infrastructure-as-a-Service

IDE Integrated Development Environment

JCEF Java Chromium Embedded Framework

UI User Interface

UML Unified Modeling Language

URL Uniform Resource Locator

VCS Version Control System

SDK Software Development Kit

SGI Structured Grading Instruction

TUM Technical University of Munich

Chapter 1

Introduction

In order to effectively teach increasing numbers of students[1], the Department of Informatics at the Technical University of Munich (TUM) developed the Automatic Assessment Management System for Interactive Learning (Artemis) to provide a capable, scalable platform to conduct large courses [KS18]. One of the various available exercise types on the platform are programming exercises. A Version Control System (VCS) like Git[2] supplies these exercises and allows students to download a template and upload their solutions. Uploads are automatically built and tested with predefined tests using a Continuous Integration System (CIS), providing immediate feedback to the students. With this feedback, students can interactively improve their solution until they are satisfied with the automatic test result.

While solving exercises, students need to switch between Artemis's client application in their browser to view the problem statement, their Integrated Development Environment (IDE) to write code, their VCS client to upload the code, and back to the browser to view the results. The Orion plugin[3] for IntelliJ[4] improves this process by integrating the Artemis client into a browser inside the IDE, using IntelliJ's Git plugin to clone and push the code, and providing the build results directly inside the IDE [Ung20]. This enables students to solve programming exercises without leaving the IDE at all.

[1] See Appendix A
[2] `https://git-scm.com/`
[3] `https://github.com/ls1intum/Orion`
[4] `https://www.jetbrains.com/idea`

1.1 Problem

In addition to purely automatic assessment via tests, manual assessment of programming exercises is required. This is due to the creative nature of programming, which leads to many possible solutions submitted by students, making it nearly impossible to create tests that are able to precisely assess all of them. For tests based on the output of a program, only small typos can lead to many test failures and reduce the score disproportionately [IS15, chapter 1]. Conversely, some properties, like the efficient implementation of an algorithm or the correct usage of language features, are difficult to determine automatically [ZKF11, chapter 1], with a faulty solution still giving the correct result. Additionally, some edge cases, like code that does not compile or terminate, cannot be handled by automatic tests [IS15, chapter 1]. All of these cases require manual assessment. The ability for reviewers to give qualitative feedback is recommended for automatic assessment systems [Pie13, chapter 6].

In Artemis, reviewers assess the code manually using an integrated code editor [Mon17]. The editor enables the reviewers to browse and view the code and to leave assessment comments; further details are described in Section 3.1. While examining the code is sufficient for smaller exercises, larger exercises oftentimes require the reviewers to edit, execute, and debug the code to fully understand and assess it. Such execution is currently not directly supported by Artemis.

If reviewers still wish to run the students' code, they first need to select to download the repository in Artemis and then clone it using their VCS client. Then they need to configure the repository by renaming it correctly and moving it to the right directory within the test repository, which they also need to clone once per exercise. After the configuration, they can open the code in their IDE and edit, execute, and debug it there, using the full support of the IDE. To add the resulting assessment comments, however, they need to return to Artemis's web editor.

1.2 Motivation

The described steps to run the students' code require up to four different programs and thereby cause a media disruption. Especially the steps required to download and configure the submissions are purely mechanical, repetitive, and could be automated. The time and concentration needed to perform them could better be spent on the actual assessment. At TUM, this is worsened due to the number of students rising disproportionately faster than the number of research assistants, reducing the available working time per student[5].

Additionally, the required effort discourages reviewers from running the code locally. For partially correct solutions, reviewers then revert to a purely manual assessment through examining the code, which can lead to inconsistent and insufficient feedback [BBL16, chapter 1] and yields the problem of reviewers being likely more prone to overlooking edge cases [EKN$^+$11, p. T3J-4]. If, for example, a submission does not compile due to a minor error, reviewers currently must decide to either omit the automatic feedback and assess manually, sacrificing quality, or to perform the required steps to download the submission, fix the error locally, and then run the tests, sacrificing time. It is, however, preferable to not have to choose, enabling the reviewers to edit and retest the submissions without additional effort.

1.3 Objectives

The aim of this thesis is to remove the media disruption and additional effort described in Section 1.1 by integrating manual assessment into Orion. Reviewers should be able to perform the assessment of a programming exercise entirely inside IntelliJ without leaving the IDE. To achieve this, Orion needs to automate the download and configuration of relevant repositories. After downloading, the code should be available to all features of the IDE like editing, code completion, or debugging, as well as the ability to rerun the automatic tests.

Reviewers should also be able to perform the manual assessment similar to the current process in Artemis. They need to be able to view the problem statement, current result, and assessment instructions. Additionally, Orion must display assessment comments and enable the users to add new comments, edit existing comments, and delete comments. Other features of Artemis's assessment workflow, like highlighting the students' changes and enabling Structured Grading Instructions (SGIs), should also be supported.

1.4 Outline

This thesis follows the major software engineering development activities as described by Brügge and Dutoit [BD09, p. 14].

Chapter 2 – Related Work describes IntelliJ's GitHub plugin with similar functionality.

[5] See Appendix A

Chapter 3 – Requirements Analysis explains how manual assessment is performed in the current system, defines the precise requirements for the assessment in Orion, and analyzes these requirements using system models.

Chapter 4 – System Design shows the architecture of the proposed system by presenting system models of Artemis's client and the Orion plugin as well as their deployment.

Chapter 5 – Object Design details how the design goals of the system have been implemented on code level.

Chapter 6 – Summary presents which requirements have been met and summarizes the conclusion of the thesis.

Chapter 2

Related Work

Similar to Artemis's assessment process, GitHub[1] also features comments on code lines while reviewing so-called pull requests. If a developer wants to add any changes to a repository, they usually need to create a pull request, which suggests to merge these changes into the project. Other developers can then view and discuss the changes, leaving both general comments and comments referring to a specific line of code. After a sufficient number of developers has approved the pull request, a project maintainer can merge the changes into the repository.

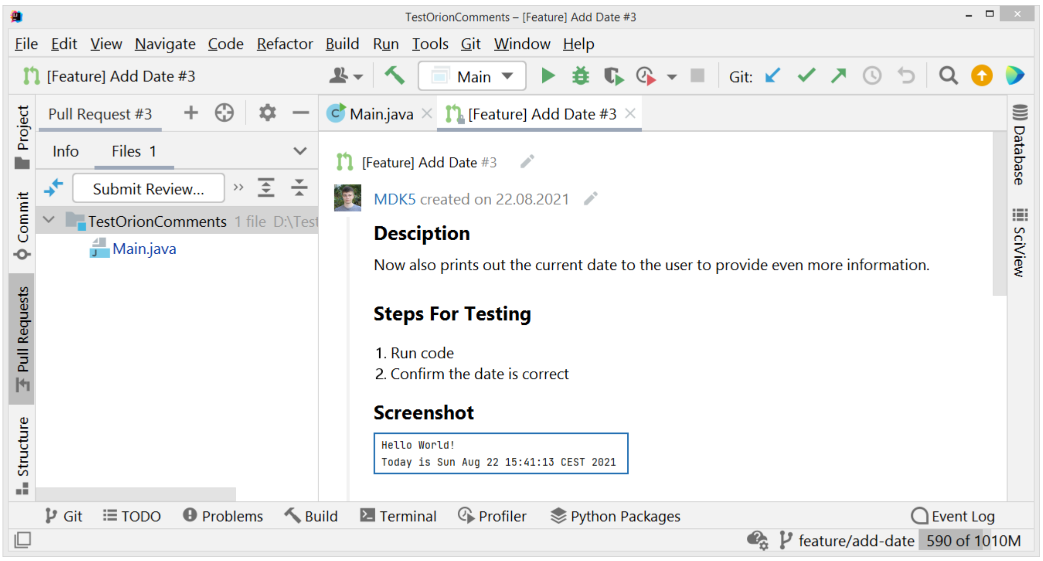
For better integration, IntelliJ provides a GitHub plugin[2], which, among other things, allows to view and comment on pull requests within the IDE. Upon selecting a specific pull request, the plugin provides an overview as shown in Figure 2.1. On the left side it shows a list of all changed files, while the main editor contains the description of the request, similar to what the web client shows. When selecting a changed file, a separate editor is opened (see Figure 2.2). This editor shows the code of the file and highlights the changes. Reviewers can then add new comments by clicking the "plus" button on the left gutter of the editor (1) and add text to it (2). Comments may also have replies (3). Each comment also has buttons to edit and delete them (4).

These comments are already close to the desired functionality in Orion. While the code cannot be directly reused since IntelliJ plugins cannot access classes from other plugins but only from the base platform, its source code is publicly available[3] and studying it provides valuable insights into how the underlying library needs to be used.

[1] `https://github.com`

[2] `https://www.jetbrains.com/help/idea/github.html`

[3] `https://github.com/JetBrains/intellij-community/tree/master/plugins/github/src/org/jetbrains/plugins/github`



[Feature] Add Date #3

MDK5 created on 22.08.2021

Descption

Now also prints out the current date to the user to provide even more information.

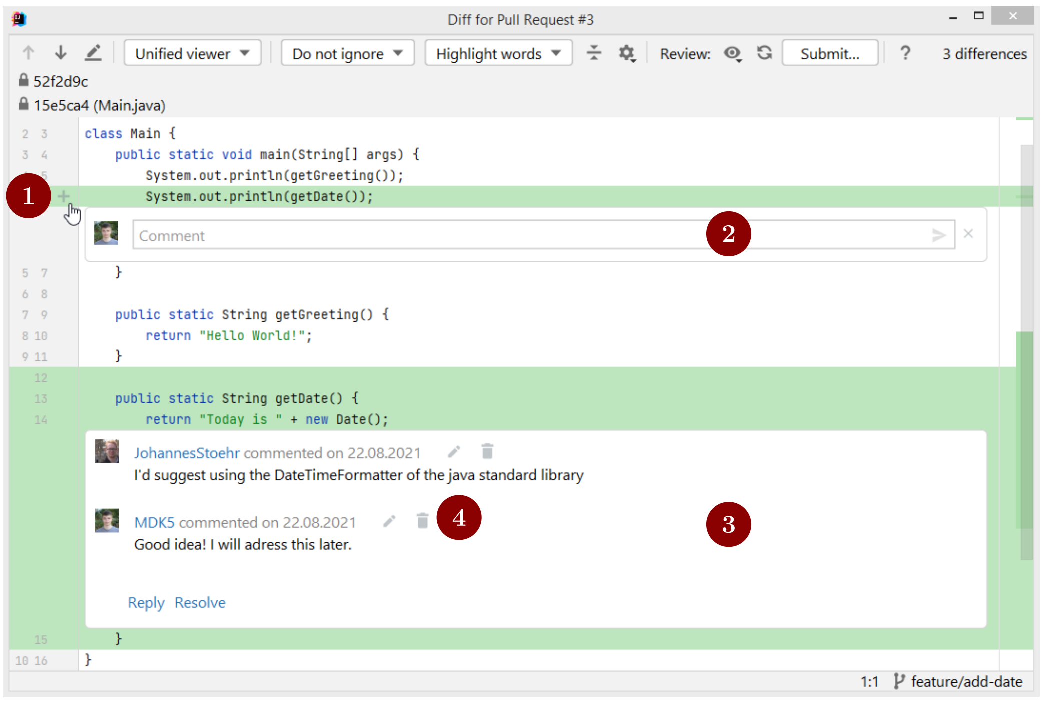
Steps For Testing

1. Run code
2. Confirm the date is correct

Screenshot

```
Hello World!
Today is Sun Aug 22 15:41:13 CEST 2021
```

Figure 2.1: Screenshot of the pull request overview of the GitHub plugin, font size enlarged



```
class Main {
    public static void main(String[] args) {
        System.out.println(getGreeting());
        System.out.println(getDate());
    }

    public static String getGreeting() {
        return "Hello World!";
    }

    public static String getDate() {
        return "Today is " + new Date();
    }
}
```

JohannesStoehr commented on 22.08.2021

I'd suggest using the DateTimeFormatter of the java standard library

MDK5 commented on 22.08.2021

Good idea! I will adress this later.

Figure 2.2: Screenshot of the review editor of the GitHub plugin, font size enlarged

Chapter 3

Requirements Analysis

As described in Section 1.3, this thesis aims to improve the manual assessment in Artemis by integrating it into the Orion plugin and enabling reviewers to edit and execute the students' code without further configuration. In this chapter we discuss the proposed system from a user's point of view, following the Requirements Analysis Document Template of Brügge and Dutoit [BD09, p. 194]. We first explain the features of the current system that are relevant for this thesis in Section 3.1, then define the requirements for the proposed system in Section 3.2, and finally analyze the requirements using various models in Section 3.3.

3.1 Current System

In this section we describe features of Artemis and Orion which are relevant for this thesis. We show how programming exercises are generally performed in Section 3.1.1, how the assessment works in detail in Section 3.1.2, and how Orion currently interacts with Artemis in Section 3.1.3.

Artemis is used to conduct programming courses, therefore all non-administrative features are organized in courses. A course consists of exercises, lectures, and exams. There are multiple types of exercises, e.g. text exercises, modeling exercises, quiz exercises, and programming exercises. This thesis focuses only on the latter type. Users can interact with courses in one of the following four roles:

- **Students** participate in exercises and exams. They can view the problem statement, submit their solution, receive a score, and receive a grade at the end of the course.

- **Tutors** review student code. They can access the assessment dashboard, view the example solutions, tests and assessment instructions of exercises, and create manual results for submissions.
- **Instructors** have every permission in the course. They can create and delete exercises, lectures and exams, override any assessment, and view every result.
- **Editors** are privileged tutors who help creating and maintaining exercises, but do not need full instructor access. They can create and edit exercises but not delete them, nor are they able to view all student results [GSS21, chapter 3]. The role has no impact on the assessment process, therefore we will not distinguish between instructors and editors in this thesis.

While the assessment is performed by tutors in most courses, some courses are managed only by instructors with the assessment also performed by the instructors. We will refer to the user performing the assessment as reviewer. Reviewers are either instructors, editors, or tutors.

3.1.1 Artemis's Programming Exercise Workflow

The conduction of a programming exercise with manual assessment follows a general workflow, which is shown as a Unified Modeling Language (UML) activity diagram [BRJ05, chapter 19] in Figure 3.1. Initially, a course instructor needs to set up the exercise. This includes writing a problem statement and assessment instructions as well as setting various settings, e.g. the name, start date, due date, and assessment due date of the exercise. Artemis then automatically creates three repositories on the VCS server: template, solution, and test. The template repository contains the code any student will receive at the start of their exercise, the solution repository contains an example solution, and the test repository contains automatic tests that will provide immediate feedback to the students. Artemis also configures build plans on the CIS server that will automatically test template and solution. The instructor can then clone these repositories and commit their code.

After the start date has passed, the exercise becomes visible for the students. Students can choose to start the exercise. Artemis then creates a participation for the student by setting up a personal repository containing a copy of the template repository as well as a build plan to test the student's code. Students solve the exercise by either editing their code in the browser using Artemis's integrated code editor [Mon17] or by cloning their repository, programming locally, and committing their changes. Whenever changes are

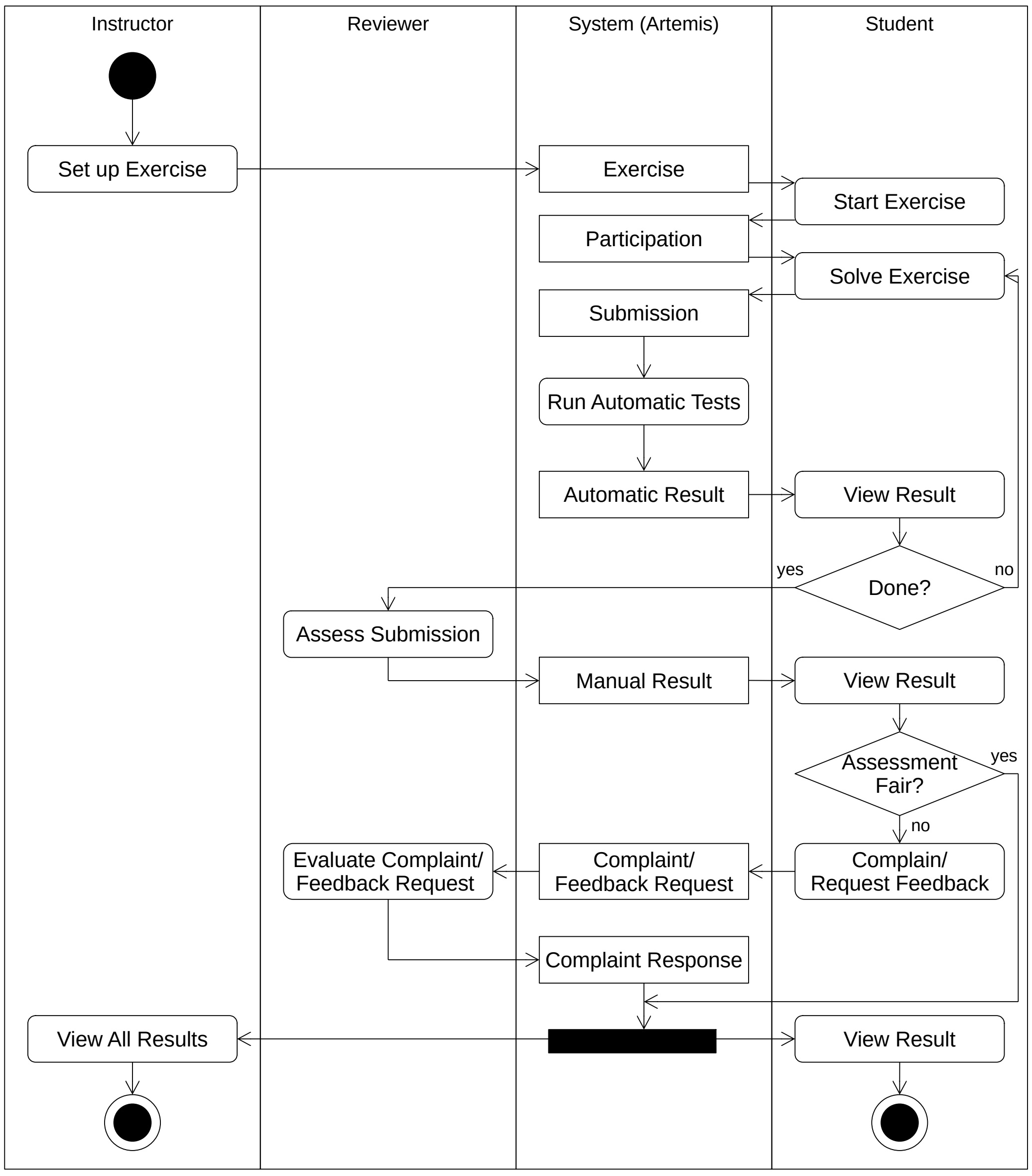


Figure 3.1: Current System: Activity Diagram showing the general programming exercise workflow with manual assessment enabled

submitted, Artemis creates a new submission and triggers the build plan to run the automated tests for the submitted code. This generates an automatic result which is shown to the student. The student can then choose to further alter the code and resubmit until they are done with their participation or the due date has passed. Each submission receives a new automated result.

Reviewers start assessing the submission after the due date has passed. They review the student's latest submission and its automated result based on the assessment instructions defined by the instructor and add their own feedback comments, adding or deducting points. This improved feedback is stored as a new manual result. After the assessment due date, the manual result is presented to the student. The student is able to review their result and, if they think their feedback is unfair or incomplete, can create either a complaint to ask for a different score or a feedback request to ask to elaborate the comments. These requests are then evaluated by a second, different reviewer, who can override the previous result with a new, improved result and write a response for the complaint or feedback request. The final feedback is presented to both the student and the instructor; the instructor can then use the results to determine the student's score or grade.

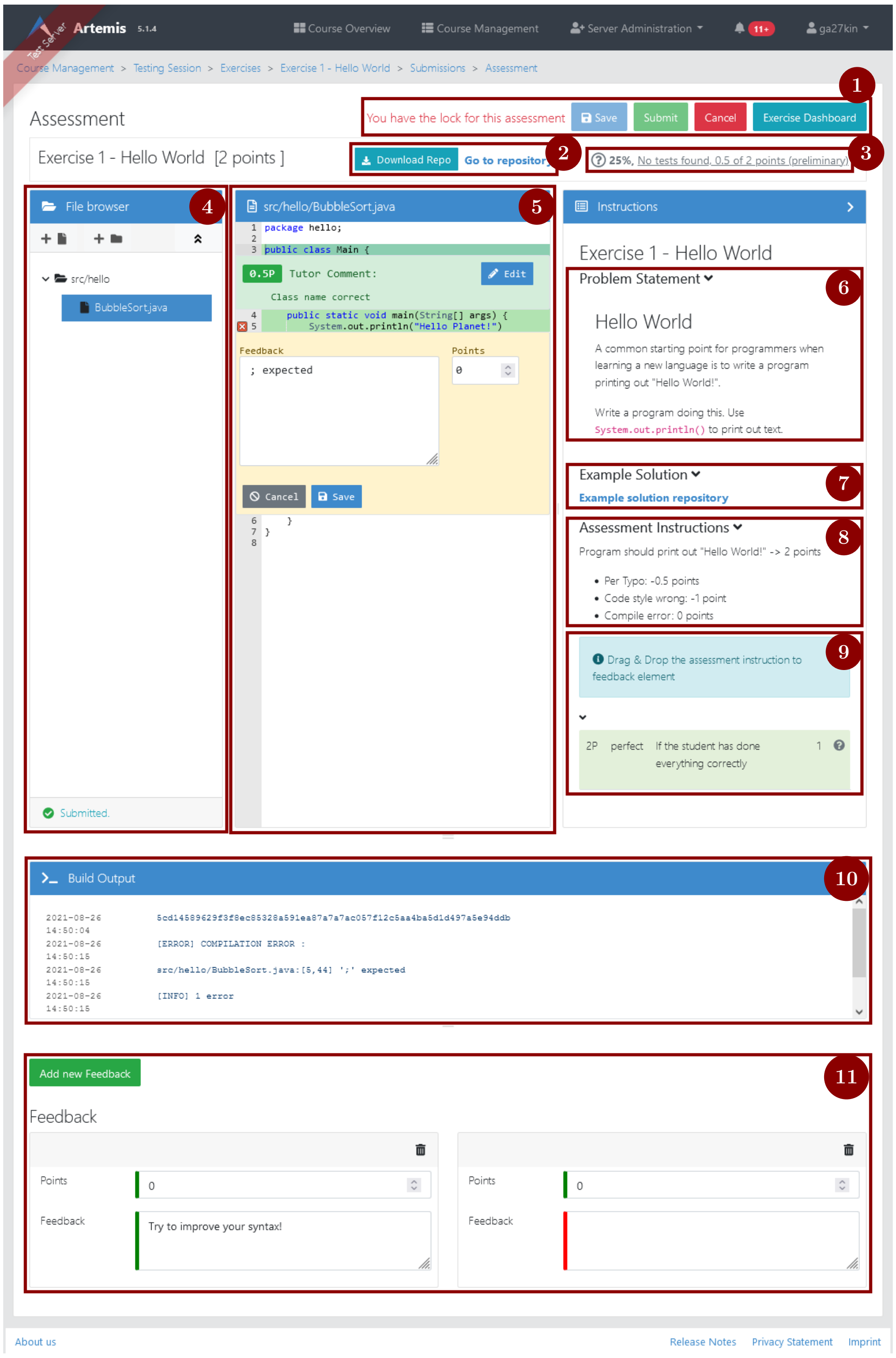
3.1.2 Artemis Manual Assessment Workflow

In order to assess a submission in the workflow described in the previous section, reviewers need to navigate to the assessment dashboard of the exercise and then select to assess a submission. This opens the assessment view of a submission as shown in Figure 3.2. The top right corner (1) contains control buttons to save, submit, or cancel the assessment as well as information about the lock ensuring each submission is only assessed by one reviewer at a time.

The current result (3) with its score is located below the controls. Clicking on it opens a pop-up showing all assessment comments, both automatic and manual. The output of the last build is also shown below (10). Next to the result are buttons to download the submission (2) in case the reviewer wants to take a close look at the repository or download the code.

The center is filled with the online code editor [Mon17]. On its left side (4), reviewers can browse the student's files. On the right side, the editor shows the problem statement (6), a link to the example solution (7), the assessment instructions (8), and the SGIs [Elh20] (9). These are suggestions for feedback comments prepared by the instructors. Reviewers can simply drag-and-drop them onto a new comment to apply them. This unifies the feedback and accelerates the assessment process.

In the middle of the editor (5), the selected file is shown. The editor



You have the lock for this assessment

Exercise 1 - Hello World [2 points]

25%, No tests found, 0.5 of 2 points (preliminary)

```
package hello;

public class Main {
```

0.5P Tutor Comment:

Class name correct

```
    public static void main(String[] args) {
        System.out.println("Hello Planet!")
```

Feedback: ; expected

Points: 0

```
    }
}
```

Exercise 1 - Hello World

Problem Statement

Hello World

A common starting point for programmers when learning a new language is to write a program printing out "Hello World!".

Write a program doing this. Use `System.out.println()` to print out text.

Example Solution

Example solution repository

Assessment Instructions

Program should print out "Hello World!" -> 2 points

- Per Typo: -0.5 points
- Code style wrong: -1 point
- Compile error: 0 points

Drag & Drop the assessment instruction to feedback element

2P	perfect	If the student has done everything correctly	1

```
2021-08-26          5cd14589629f3f8ec85328a591ea87a7a7ac057f12c5aa4ba5d1d497a5e94ddb
14:50:04
2021-08-26          [ERROR] COMPILATION ERROR :
14:50:15
2021-08-26          src/hello/BubbleSort.java:[5,44] ';' expected
14:50:15
2021-08-26          [INFO] 1 error
14:50:15
```

Feedback

Points	0
Feedback	Try to improve your syntax!

Points	0
Feedback	

Figure 3.2: Current system: screenshot of the assessment view. The style has been slightly adapted to avoid scroll bars.

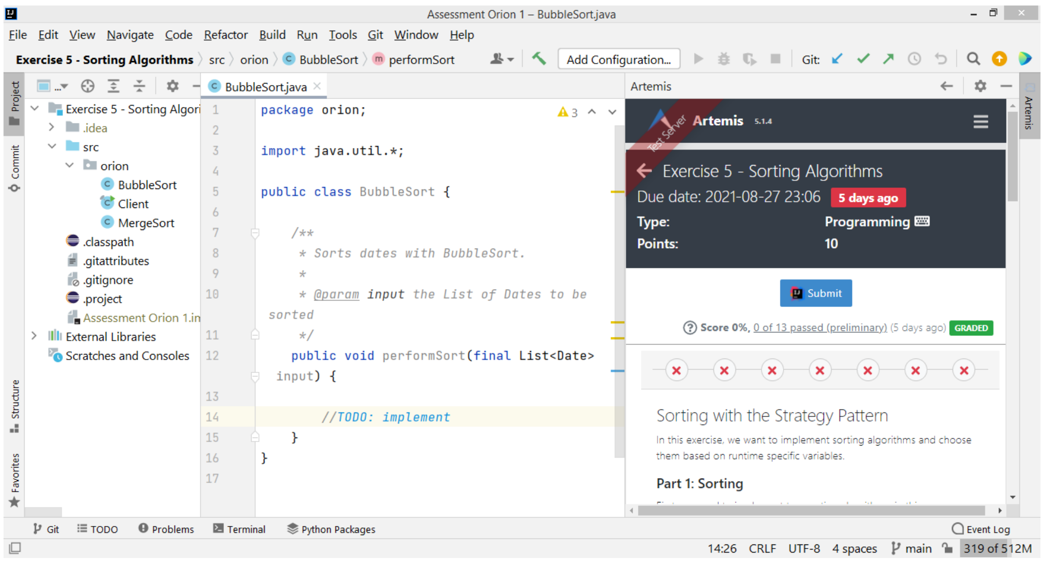
highlights which lines have been added by the student (in green). Reviewers can leave inline feedback comments referencing a specific line [lCY21]. A comment consists of both a text and a score and is colored depending on whether the score is positive or negative. Reviewers can also create and edit general assessment comments not referring to a specific line using the buttons at the bottom (11).

3.1.3 Orion

The Orion plugin for IntelliJ was developed by Ungar [Ung20] in order to improve the steps *Start Exercise* and *Solve Exercise* from Figure 3.1 for students as well as *Set up Exercise* for instructors. It is installed into the IDE IntelliJ, where it offers a special tool window, containing an integrated browser. With this browser, users can interact with Artemis from within their IDE.

If a student opens an exercise with Orion, instead of getting presented a button to receive a link to their personal repository which they can use to clone it, they get a button that automates that. This button directly downloads the repository, sets up an IntelliJ project for it, and opens it. Students can then use the project regularly, e.g. open and edit files. The Orion window shows the problem statement as well as a button to submit their changes, as shown in Figure 3.3, with the Orion tool window on the right, the editor in the center and the file browser to the left. When clicking the submit button, Orion automatically commits and pushes all changes to Artemis and then connects to the output of the automatic build process to show it in the regular IntelliJ build view, as if it was a local build. Students do not need to manually interact with the repository and also don't need an external browser for Artemis. With Orion they can solve programming exercises only using in IntelliJ.

Similarly, Orion helps instructors to create exercises. After configuring the settings, instructors can open the exercise in Orion, which automatically downloads template, test, and solution repository, configures them as modules in an IntelliJ project, and opens that project. They can then edit the files while Orion provides an editor for changing the problem statement, a button to automatically commit and push all changes, and a button to run the current tests locally without any further configuration. This view is shown in Figure 3.4 with the three repositories as modules in the file browser on the left side, the editor in the middle, and the Orion tool window to the right.


Assessment Orion 1 – BubbleSort.java
File Edit View Navigate Code Refactor Build Run Tools Git Window Help
Exercise 5 - Sorting Algorithms › src › orion › BubbleSort › performSort
Add Configuration...
Exercise 5 - Sorting Algori
.idea
src
orion
BubbleSort
Client
MergeSort
.classpath
.gitattributes
.gitignore
.project
Assessment Orion 1.in
External Libraries
Scratches and Consoles
BubbleSort.java
Artemis
Artemis 5.1.4
Submit
Git TODO Problems Terminal Python Packages
Event Log
14:26 CRLF UTF-8 4 spaces main 319 of 512M

```
package orion;

import java.util.*;

public class BubbleSort {

    /**
     * Sorts dates with BubbleSort.
     *
     * @param input the List of Dates to be
 sorted
     */
    public void performSort(final List<Date>
  input) {

        //TODO: implement
    }
}
```

Exercise 5 - Sorting Algorithms
Due date: 2021-08-27 23:06 5 days ago
Type: Programming
Points: 10

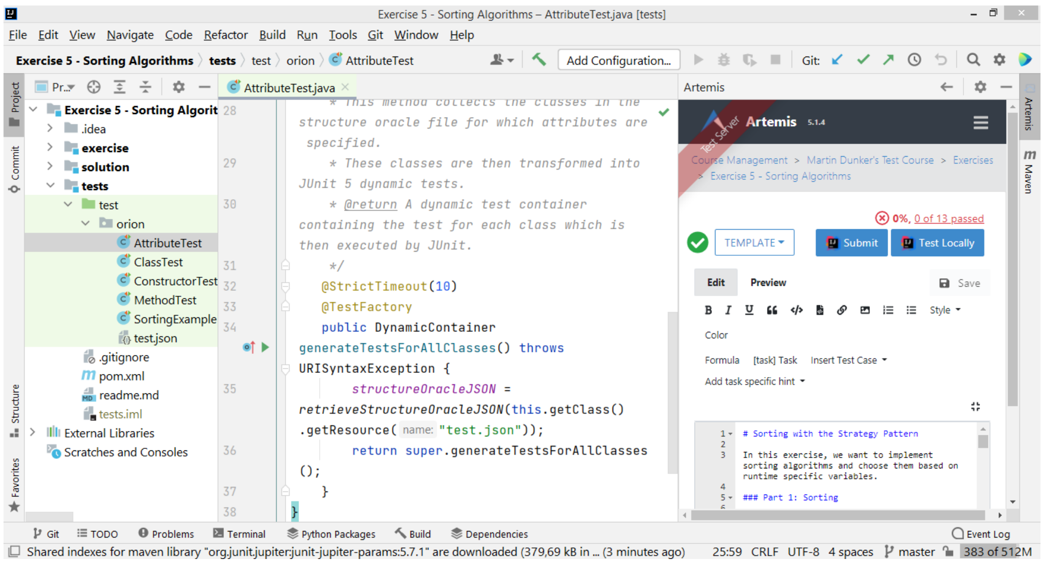
Score 0%, 0 of 13 passed (preliminary) (5 days ago) GRADED

Sorting with the Strategy Pattern

In this exercise, we want to implement sorting algorithms and choose them based on runtime specific variables.

Part 1: Sorting

Figure 3.3: Current System: Screenshot of IntelliJ with Orion from a student's perspective while solving an exercise


Exercise 5 - Sorting Algorithms – AttributeTest.java [tests]
File Edit View Navigate Code Refactor Build Run Tools Git Window Help
Exercise 5 - Sorting Algorithms › tests › test › orion › AttributeTest
Add Configuration...
Exercise 5 - Sorting Algorit
.idea
exercise
solution
tests
test
orion
AttributeTest
ClassTest
ConstructorTest
MethodTest
SortingExample
test.json
.gitignore
pom.xml
readme.md
tests.iml
External Libraries
Scratches and Consoles
AttributeTest.java
Artemis
Artemis 5.1.4
Course Management > Martin Dunker's Test Course > Exercises > Exercise 5 - Sorting Algorithms
0%, 0 of 13 passed
TEMPLATE
Submit
Test Locally
Edit
Preview
Save
Color
Formula
[task] Task
Insert Test Case
Add task specific hint
Git TODO Problems Terminal Python Packages Build Dependencies
Event Log
25:59 CRLF UTF-8 4 spaces master 383 of 512M

```
     * This method collects the classes in the
 structure oracle file for which attributes are
  specified.
     * These classes are then transformed into
 JUnit 5 dynamic tests.
     * @return A dynamic test container
 containing the test for each class which is
 then executed by JUnit.
     */
    @StrictTimeout(10)
    @TestFactory
    public DynamicContainer
 generateTestsForAllClasses() throws
 URISyntaxException {
        structureOracleJSON =
 retrieveStructureOracleJSON(this.getClass()
 .getResource( name: "test.json"));
        return super.generateTestsForAllClasses
 ();
    }
}
```

```
# Sorting with the Strategy Pattern

In this exercise, we want to implement
sorting algorithms and choose them based on
runtime specific variables.

### Part 1: Sorting
```

Shared indexes for maven library "org.junit.jupiter:junit-jupiter-params:5.7.1" are downloaded (379,69 kB in ... (3 minutes ago)

Figure 3.4: Current System: Screenshot of IntelliJ with Orion from an instructor's perspective while setting up an exercise

3.2 Proposed System

This section explains the precise objectives of this thesis by defining the functional requirements in Section 3.2.1 and the nonfunctional requirements in Section 3.2.2.

3.2.1 Functional Requirements

This section covers the functional requirements of the system, which describe how the system interacts with other systems and the user [BD09, p. 119]. The requirements are split into two parts: First we describe how the system should interact with the user in order to **download and execute student submissions,** later we define how reviewers add assessment comments for **manual assessment in Orion.** The requirements are adapted from Young [lCY21] since the manual assessment in Orion shares its characteristics with the manual assessment of programming exercises in Artemis.

Download and Execute Student Submissions

FR1.1 **Start Assessment of Exercise:** Reviewers must be able to start the assessment of a programming exercise in Orion. Orion should automatically download the automatic tests and configure them.

FR1.2 **Assess Submission:** Reviewers must be able to start assessing a student's submission and download it in Orion. The submission must be anonymized to allow for a blind assessment. During the assessment, the submission must be unavailable for other reviewers to ensure one submission is only assessed by one reviewer at a time.

FR1.3 **View Code in IDE:** Reviewers must be able to view the student's code. The view must be configured to provide all of IntelliJ's IDE support, including syntax highlighting, auto-formatting, and code completion.

FR1.4 **Highlight Changes:** Reviewers must be able to identify the student's changes quickly. Orion must highlight them relative to the initial template code, which was provided by the instructor.

FR1.5 **Edit and Execute Student's Code:** Reviewers must be able to edit their local copy of the student's code and execute the automatic tests on it. The configuration to run the tests must be provided automatically. This enables reviewers to test local fixes in order to improve their understanding of the student's code.

FR1.6 **Support Feedback Requests and Complaints:** Assessment in Orion should also be available for reviews of complaints and more feedback requests.

Manual Assessment in Orion

FR2.1 **View Problem Statement and Assessment Instructions:** Reviewers must be able to view the problem statement of the assessed exercise as well as the assessment instructions as defined by the instructor.

FR2.2 **View Result:** Reviewers must be able to view the student's current build result and score as reported by the CIS system and shown to the student after submitting.

FR2.3 **View Feedback:** Reviewers must be able to view the current feedback comments, both automatic and manual. Inline feedback must be displayed inline.

FR2.4 **Add, Edit, and Delete Feedback:** Reviewers must be able to add, edit, and delete feedback comments. Feedback comments can be either general, referring to a file, or inline, referring to a certain line of code. A comment must have a text and a score.

FR2.5 **Save Assessment:** Reviewers must be able to save their feedback comments. This should store the local comments on the Artemis server. The feedback needs to be compatible with Artemis's current feedback format and must be indistinguishable from feedback added via the web client.

FR2.6 **Cancel Assessment:** Reviewers must be able to cancel their assessment. This should remove all their pending comments and make the submission available for assessment for other reviewers again.

FR2.7 **Use Structured Grading Instructions:** Reviewers must be able to use SGIs in Orion. The instructions must be shown and it must be possible to drag-and-drop them into a new feedback.

3.2.2 Nonfunctional Requirements

This section lists the nonfunctional requirements. These are requirements not directly associated with the behavior of the system [BD09, p.120]. They are categorized using the FURPS+ model [Gra92] as described by Brügge and Dutoit [BD09, p-120-121], including the category constraints.

Usability

NFR1.1 **Usage Complexity:** The amount of interactions required to run the student's code after starting the assessment should be reduced from the current minimum of eight interactions[1] to a minimum of one interaction[2].

NFR1.2 **Usage Complexity:** The amount of interactions required to add assessment comments after starting the assessment should not increase from the current minimum of two interactions[3].

NFR1.3 **Graphical User Interface (GUI) Consistency with Artemis:** The GUI elements added in Artemis should look similar to the current view. The color scheme, button descriptions, and positions should match the ones of the current system. Reviewers accustomed to the current system should be able to immediately adapt to the proposed system.

NFR1.4 **GUI Consistency with IntelliJ:** GUI elements added in IntelliJ should follow the IntelliJ UI Guidelines[4] to be consistent with the GUI of the IDE.

Reliability

NFR2.1 **Download Large Repositories:** The proposed system should be able to download student repositories up to a size of 10 megabytes when using default settings and terminate gracefully if the limit is exceeded.

NFR2.2 **Illegal Input:** The proposed system should prevent illegal input and ensure only valid feedback is sent to the server. All feedback referencing files needs to reference an existing file and all feedback scores need to be valid numbers.

[1] Two clicks to view the repository's URL and copy it, at least two clicks to clone it, at least three clicks to copy the repository into an IntelliJ project, and one click to start the test run configuration

[2] One click to start the already provided and selected run configuration

[3] One click on the line to add the comment and one click to save it

[4] `https://jetbrains.github.io/ui`

Performance

NFR3.1 **Communication Between Artemis Client and Server:** The proposed system should not slow down the communication between the Artemis client and Artemis server. All operations that are already available in Artemis should not require more data being sent between the client and server if invoked from Orion.

NFR3.2 **Load Time:** The duration between clicking the button to start assessing a submission and the assessment becoming available should be at most three seconds longer than downloading the submission using a regular browser.

NFR3.3 **Save Time:** The duration between clicking the button to save a feedback in Orion and the feedback being stored in the Artemis client should be at most one second longer than saving the same feedback in the web client directly.

Constraints

NFR5.1 **Platform:** The features need to extend the existing Orion plugin[5]. The plugin is written mostly in Kotlin[6] and needs to use the IntelliJ platform Software Development Kit (SDK)[7].

NFR5.2 **Platform:** Changes in the GUI on the web client need to be integrated into the current Artemis client[8], which is written in Angular[9].

3.3 System Models

In this section we provide various models to describe the proposed system. We present scenarios in Section 3.3.1, derive use cases from the requirements in Section 3.3.2, we define the analysis object model in Section 3.3.3, the dynamic model in Section 3.3.4, and show the changes of the user interface in Section 3.3.5.

[5] `https://github.com/ls1intum/Orion`
[6] `https://kotlinlang.org`
[7] `https://plugins.jetbrains.com/docs/intellij/welcome.html`
[8] `https://github.com/ls1intum/Artemis`
[9] `https://angular.io`

3.3.1 Scenarios

In this section we present one visionary and two demo scenarios. They describe one specific interaction of a user with the system [BD09, p. 126]. Visionary scenarios describe a potential future system, demo scenarios describe workflows that are possible with the proposed system. In the scenarios, the proposed system is split into three subsystems: The user's IDE IntelliJ, the Orion plugin installed into it, and the Artemis client running in Orion's integrated browser. We present one visionary scenario

Visionary Scenario: Offline Assessment

This scenario has one participating actor, the tutor Alice. Alice has opened IntelliJ with Orion and has navigated to Artemis's assessment dashboard for the exercise "Introduction" in the course "Introduction to Software Engineering" in the integrated browser.

On the dashboard, Artemis presents buttons to start the assessment of a new submission as well as a button to start the assessment for multiple submissions. Alice selects to assess 10 new submissions, which Orion downloads. Alice then goes on her way home and disconnects from the internet. Orion notices the disconnect and switches to offline mode, showing its own GUI instead of Artemis, displaying the previously downloaded submissions and offering buttons to assess them. While riding home on the train, Alice assesses the submissions by navigating to them and adding assessment comments. Orion stores these comments locally. Once home, Alice reconnects with the internet. Orion notices this, loads Artemis again, and automatically synchronizes the local feedback with Artemis.

Demo Scenario: Tutor Edits and Tests Submission

This scenario has one participating actor, the tutor Alice. She has opened the assessment dashboard for the course "Introduction to Software Engineering" in Orion.

In the dashboard, Artemis shows a list of all exercises for the course. Alice selects the exercise "Introduction", which has unassessed submissions. Artemis then orders Orion to open this exercise in assessment mode. Orion downloads the required repositories for the exercise into a new project and instructs IntelliJ to open that project. After opening the new project, Orion displays Artemis's exercise assessment dashboard for the open exercise. Alice gets presented a list of her assessments in the exercise as well as buttons to continue or start new assessments.

Alice clicks the button to open a new assessment. Orion then downloads a new submission into the open project and opens the submission page for it. This page contains the problem statement of the exercise, the assessment instructions, and the current test results and grade. Alice opens several files of the submission in IntelliJ and looks at the test results. She suspects the submission has wrongly initialized one variable with 1 instead of 0 but is not entirely sure. In order to make sure, she edits the file with her fix and then selects to execute the tests in IntelliJ with a configuration provided by Orion. IntelliJ runs the tests and shows that the submission passes more tests now. Alice uses this information to edit the score accordingly.

Demo Scenario: Tutor Adds and Removes Feedback Comments

This scenario has one participating actor, the tutor Alice. She has opened the assessment of a submission in IntelliJ with Orion. Orion shows the submission page of the submission in its integrated browser and displays all current inline feedback.

Alice reviews the code and sees that the student has correctly implemented a method. She wants to add a new inline feedback for this method, so she clicks a button in the editor on the line she wants to add the feedback to. Orion then creates an empty feedback in that line. Alice enters “Well done” as feedback text, sets the score associated with the feedback to “2”, and clicks save. She then realizes the student has made a mistake in the method above and wants to remove her previous feedback for that line. She clicks the edit button on the feedback and then the delete button. Orion then removes the feedback. She decides she is done with the submission and clicks the save button in Artemis. Orion then overrides the feedback in Artemis with the local feedback.

3.3.2 Use Case Model

This section covers the use cases of the proposed system, which are generalized scenarios [BD09, p. 130]. They show the relationship between users and the requirements. This section is split into the same two parts as the functional requirements: We first show the use cases that describe how to **download and execute student’s submission** and afterwards how to perform **manual assessment in Orion**.

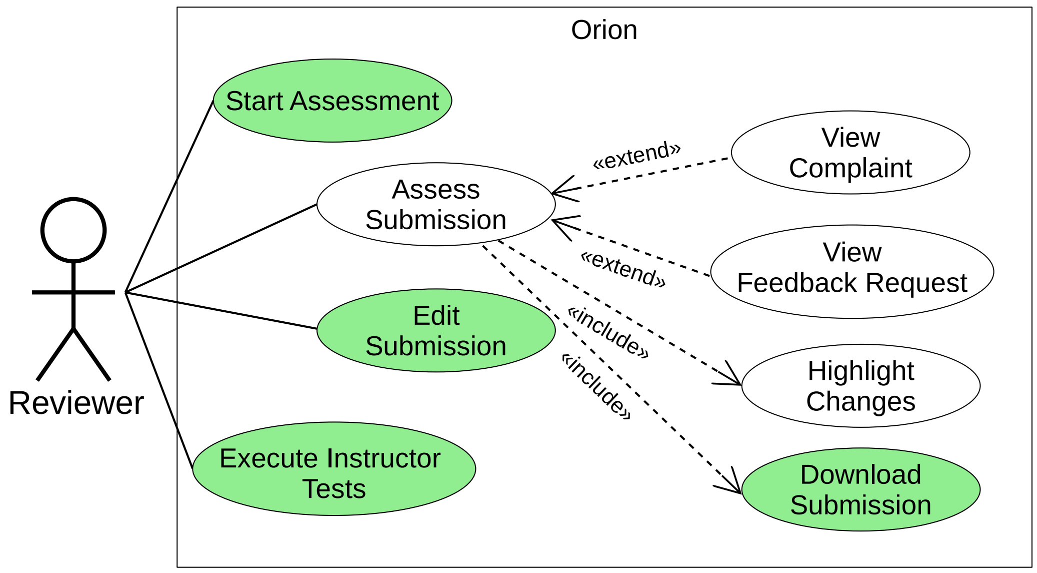


Figure 3.5: UML use case diagram displaying use cases associated with downloading and executing the students' submissions. Added use cases are colored **green**.

Download and Execute Student Submissions

Figure 3.5 shows the use cases as UML use case diagram [BRJ05, chapter 17]. Reviewers can start the assessment of an exercise (FR1.1) and afterwards view the students' submissions (FR1.3). Viewing the submission requires to download it (FR1.2); the view also highlights the code changes (FR1.4). Viewing complaints and more feedback requests (FR1.6) are special cases of viewing normal submissions where, in addition to the submission, the complaint or feedback request is shown. Highlighting the code and reviewing submissions, complaints, and feedback requests are already existing use cases in Artemis. While viewing the submission, reviewers can also edit it and execute the instructor tests on it (FR1.5).

Manual Assessment in Orion

The use cases for the manual assessment in Orion are shown in Figure 3.6. All use cases are already present in Artemis and only need to be ported to Orion. While viewing the submission, reviewers can view the problem statement and assessment instructions (FR2.1), the result (FR2.2), and the current feedback (FR2.3). They are also able to update the feedback, that is to add, delete, and edit feedback comments (FR2.4). Additionally, they can use SGIs (FR2.7), which also update the feedback. We describe this use case in more detail in Table 3.1 with the template from Brügge and Dutoit [BD09, p. 129]. Lastly, reviewers can save (FR2.5) or cancel (FR2.6) their assessment.

Use case name	Use Structured Grading Instructions
Participating actors	Initiated by `Reviewer`
Flow of events	1. The `Reviewer` selects to add a new feedback comment at a specific line. 2. `Orion` displays a new feedback comment at the given line. 3. The `Reviewer` chooses a structured grading instruction to fill into the feedback. 4. `Orion` sets the detail text and score of the new comment to the values provided by the grading instruction. 5. The `Reviewer` selects to save the new comment. 6. `Orion` saves the newly created comment in Artemis.
Entry condition	• The `Reviewer` has opened an assessment in `Orion`. • The `Reviewer` has a file from the submission open in an editor.
Exit condition	• The new comment has been saved by Artemis.
Quality requirements	• Saving the new comment should take at most 2 seconds.

Table 3.1: Use Structured Grading Instructions as a use case detail description

3.3.3 Analysis Object Model

The analysis object model of the proposed system is shown in Figure 3.7 as UML class diagram [BRJ05, chapter 8]. It depicts the relationships and taxonomies of the problem. The diagram is based on Young [lCY21, p. 33].

An assessment always belongs to a *Programming Exercise*. The exercise has, among other properties, a problem statement, assessment instructions, a

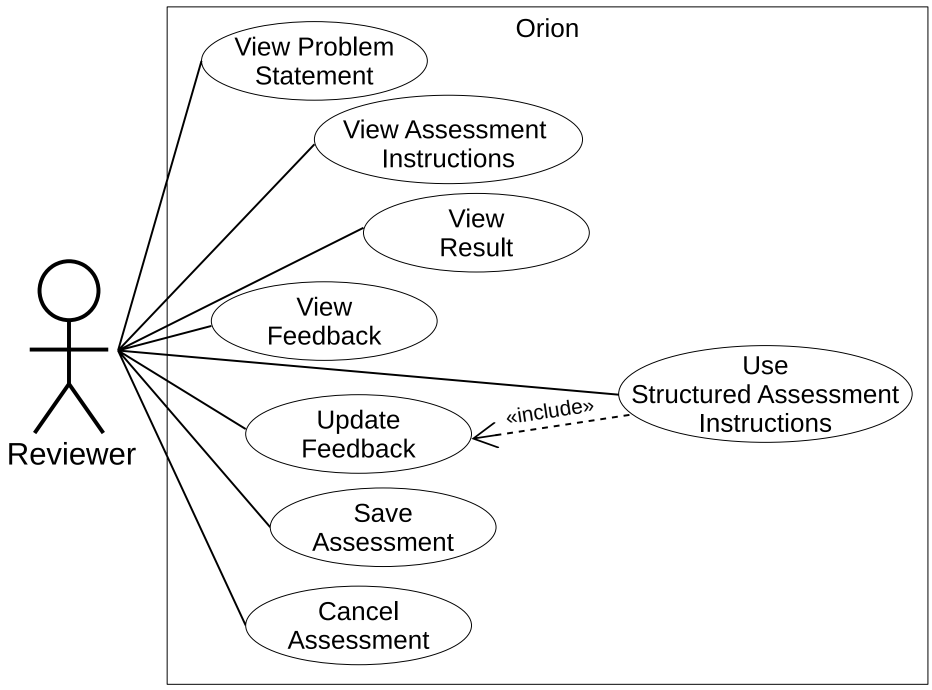


Figure 3.6: UML use case diagram displaying use cases associated with manual assessment in Orion.

start and end date, and three *Repositories*: template, solution, and test. The exercise also has an arbitrary amount of *Participation*s belonging to it, with a new participation created for every student starting the exercise. Each *Participation* is associated with a unique *Repository* storing the student's code; each student repository also belongs to exactly one participation. The *Repository* consists of *File*s and has a Uniform Resource Locator (URL) allowing to both download its contents and commit changes to it.

Each *Participation* manages an arbitrary amount of *Submission*s, each representing one submitted change. Each *Submission* receives a *Result* that gives feedback about the score of the submission. Artemis creates an automated result for each submission by executing the instructor tests. Additionally, reviewers can create results by performing a manual assessment. In order to ensure fair, blind grading, submissions are anonymized; to ensure one submission is only assessed by one reviewer at a time, they can be locked or unlocked.

Each *Result* consists of an arbitrary amount of *Feedback Comment*s. Each comment has a detail text and a score assigned to it. The score of the *Result* is calculated as the sum of the scores of its comments; the result also needs to be validated, ensuring all comments have a text and score. *Feedback Comment*s

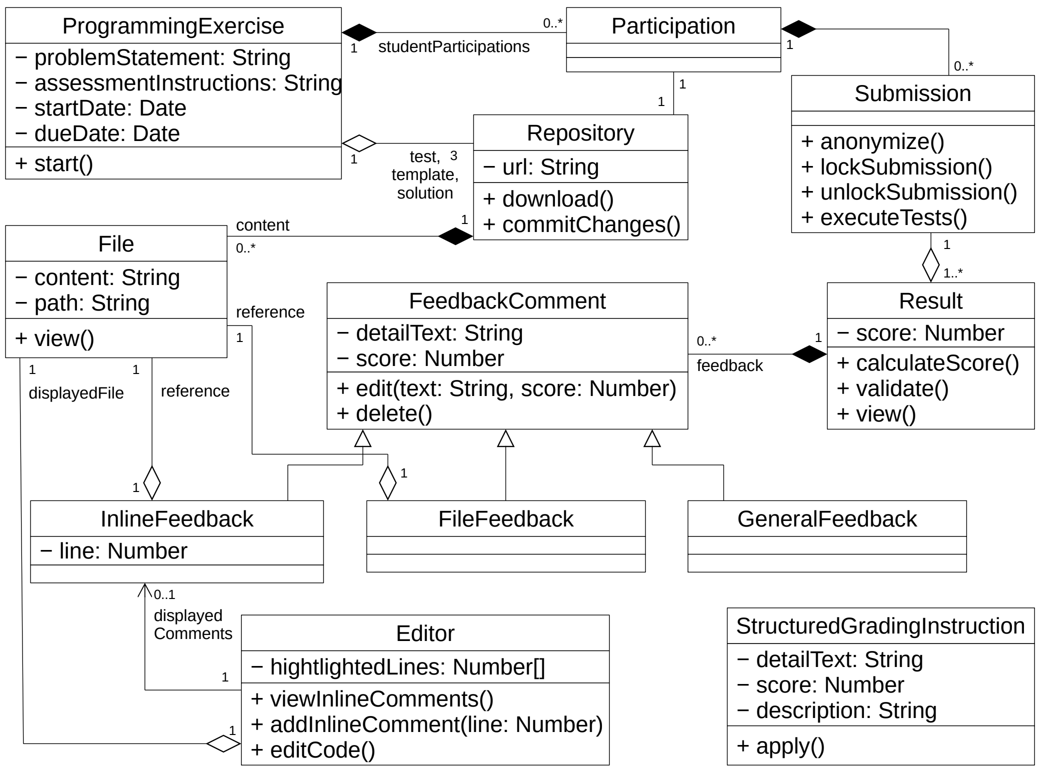


Figure 3.7: UML class diagram depicting the analysis object model, based on Young [lCY21, p. 33]

are either *General Feedback*, *File Feedback*, referencing a file of the repository, or *Inline Feedback*, referencing a specific line of a file. Comments can also be created using a *Structured Grading Instruction* which has a predefined detail text and score as well as a description explaining when it should be applied.

Lastly, a *File* can be displayed in an *Editor*. The editor highlights the changes by the student and displays all inline comments referencing the opened file. It also allows to edit the file or add new inline comments.

3.3.4 Dynamic Model

This section models the proposed assessment process as an UML activity diagram [BRJ05, chapter 19] in Figure 3.8. At first, the reviewer needs to start the assessment of the exercise. The system then either downloads or opens the exercise, depending on whether it has been downloaded before already. After the exercise is opened, the reviewer starts assessing a submission. This causes Orion to download a submission and highlight the code changes.

Afterwards the main assessment process starts, which consists of the re-

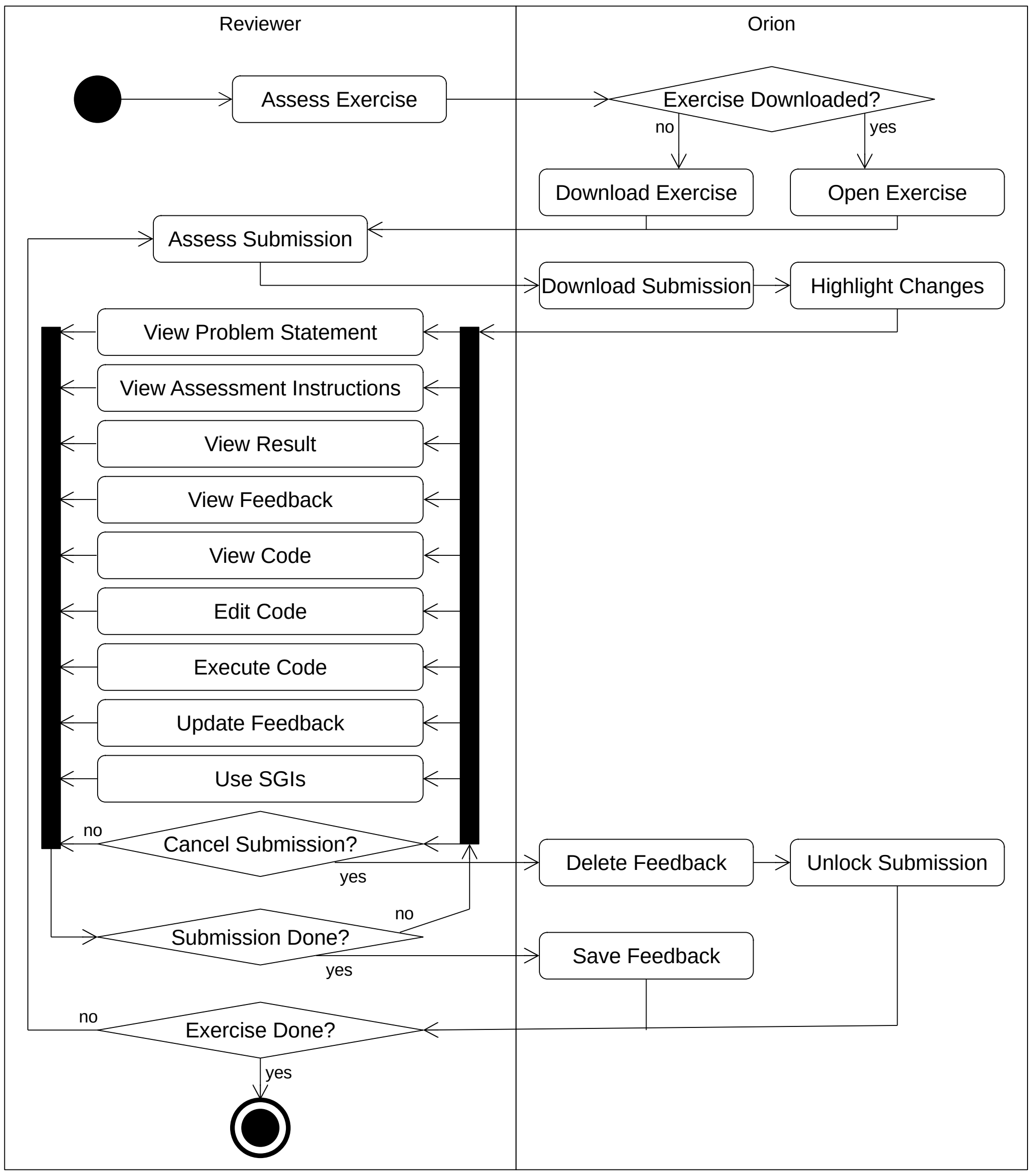


Figure 3.8: UML activity diagram of the proposed assessment process

viewer performing the following actions in any order: view the problem statement, assessment instructions, result, feedback, and code, edit and execute the code, update feedback, and use SGIs. The reviewer can also decide to cancel the assessment of the submission at any point. In that case, Orion deletes all current feedback and unlocks the submission, allowing other reviewers to assess it. The reviewer continues with these actions until they decide they are done with the submission. Orion then saves the feedback. After either saving the feedback or cancelling, the reviewer can decide to either continue assessing the exercise and start assessing a new submission or to stop, in which case the process ends.

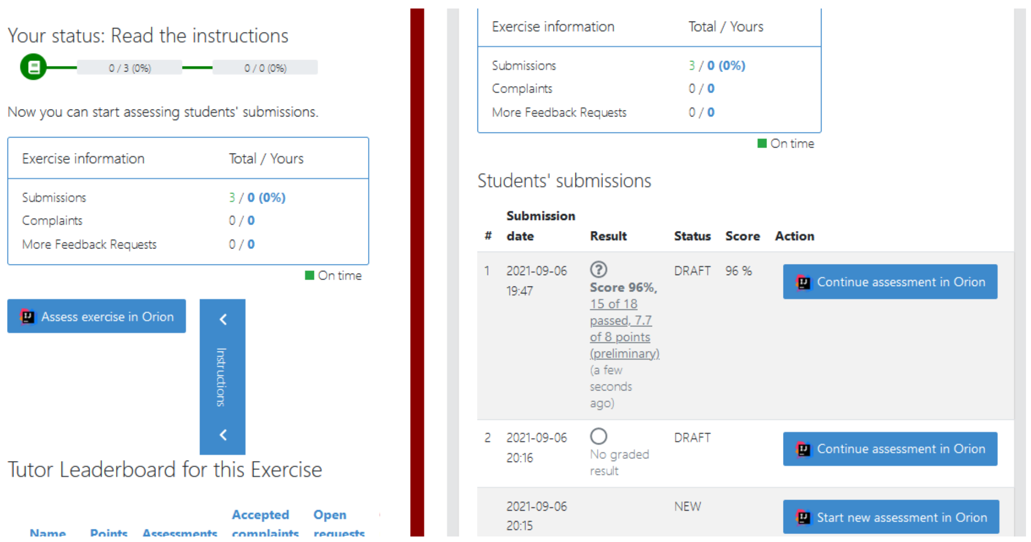
3.3.5 User Interface

After modeling the requirements using different methods, we present the proposed changes of the GUI. Due to the system being finished before this thesis, we use screenshots of the implemented system instead of mock-ups.

After selecting the exercise they want to assess, reviewers get presented the exercise assessment dashboard. If they connect using Orion, they get presented the views shown in Figure 3.9. The view on the left is presented to reviewers if the exercise is currently not opened in Orion. Instead of the submissions, reviewers get a button that, when clicked, downloads or opens the exercise. If they have opened the selected exercise, they see the right view, showing a list of all of their assessments as well as a button to start a new assessment. This view differs from the current system with the buttons not opening the code editor but instead triggering the assessment download in Orion. When IntelliJ is opened with an assessment project without a current submission, Orion automatically navigates to this page.

IntelliJ's interface during the assessment process, after downloading a submission, is shown in Figure 3.10 and Figure 3.11. On the left side (1 in both figures) there is the IntelliJ file browser, allowing to select the student's files and test files. The middle (2 in both figures) is occupied by IntelliJ's code editor. In Figure 3.10 it is in edit mode, behaving like a regular editor with the reviewer being able to edit the files. In Figure 3.11 it is in assessment mode, being read-only but instead displaying assessment comments and enabling reviewers to add new inline comments. The comments look similar to Artemis's GUI, fulfilling NFR1.3. Reviewers can switch the editors using the bar at the bottom (3 in both figures). If reviewers want to execute the code, they can use the provided run configuration and click the "run" button at the top (4 in both figures).

On the right side, Orion displays Artemis's Orion version of the assessment detail view/code editor, Artemis's view is shown in Figure 3.2. Orion



Your status: Read the instructions

0 / 3 (0%) 0 / 0 (0%)

Now you can start assessing students' submissions.

Exercise information	Total / Yours
Submissions	3 / **0 (0%)**
Complaints	0 / **0**
More Feedback Requests	0 / **0**

🟩 On time

Tutor Leaderboard for this Exercise

Name	Points	Assessments	Accepted complaints	Open requests

Exercise information	Total / Yours
Submissions	3 / **0 (0%)**
Complaints	0 / **0**
More Feedback Requests	0 / **0**

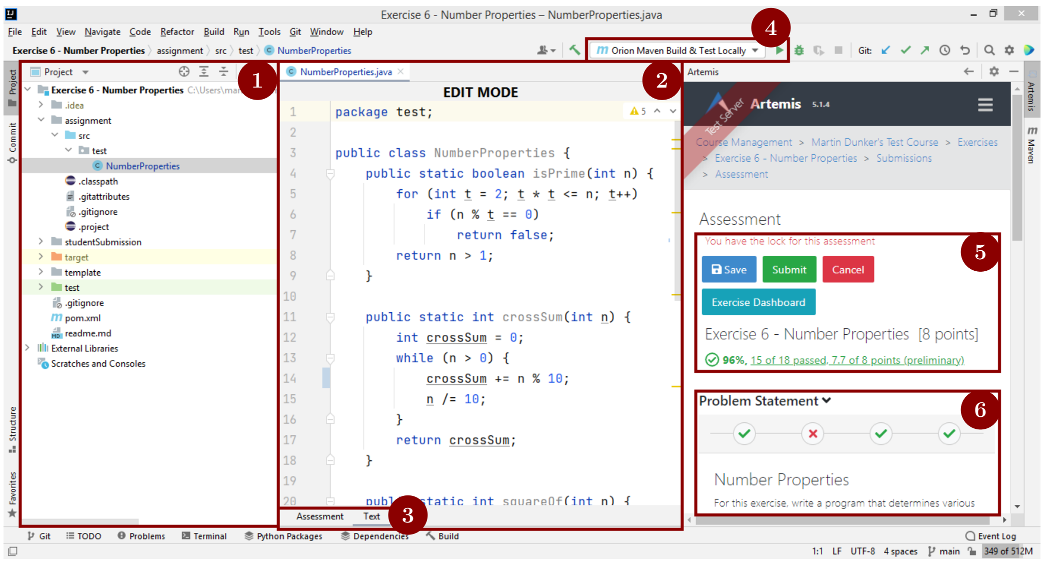
🟩 On time

Students' submissions

#	Submission date	Result	Status	Score	Action
1	2021-09-06 19:47	**Score 96%,** 15 of 18 passed, 7.7 of 8 points (preliminary) (a few seconds ago)	DRAFT	96 %	Continue assessment in Orion
2	2021-09-06 20:16	No graded result	DRAFT		Continue assessment in Orion
	2021-09-06 20:15		NEW		Start new assessment in Orion

Figure 3.9: Proposed System: Screenshots of the exercise assessment dashboard when connecting via Orion.

displays the same information excluding the code editor since viewing the code is done in IntelliJ. At the top (5 in Figure 3.10) there are buttons to save, submit, or cancel the assessment as well as the current result of the student. Below follows the problem statement (6 in Figure 3.10) and, after scrolling down, the assessment instructions (5 in Figure 3.11) and the general feedback (6 in Figure 3.11) as well as a button to add new general feedback.


Exercise 6 - Number Properties – NumberProperties.java
File Edit View Navigate Code Refactor Build Run Tools Git Window Help
Exercise 6 - Number Properties 〉 assignment 〉 src 〉 test 〉 NumberProperties
Orion Maven Build & Test Locally
Git:
Project
Exercise 6 - Number Properties C:\Users\mar
.idea
assignment
src
test
NumberProperties
.classpath
.gitattributes
.gitignore
.project
studentSubmission
target
template
test
.gitignore
pom.xml
readme.md
External Libraries
Scratches and Consoles
NumberProperties.java
EDIT MODE
Assessment
Text
Git
TODO
Problems
Terminal
Python Packages
Dependencies
Build
Artemis
Artemis 5.1.4
Test Server
Course Management > Martin Dunker's Test Course > Exercises > Exercise 6 - Number Properties > Submissions > Assessment
Save
Submit
Cancel
Exercise Dashboard
Problem Statement
Event Log
1:1 LF UTF-8 4 spaces main 349 of 512M
1
2
3
4
5
6

```java
package test;

public class NumberProperties {
    public static boolean isPrime(int n) {
        for (int t = 2; t * t <= n; t++)
            if (n % t == 0)
                return false;
        return n > 1;
    }

    public static int crossSum(int n) {
        int crossSum = 0;
        while (n > 0) {
            crossSum += n % 10;
            n /= 10;
        }
        return crossSum;
    }

    public static int squareOf(int n) {
```

Assessment

You have the lock for this assessment

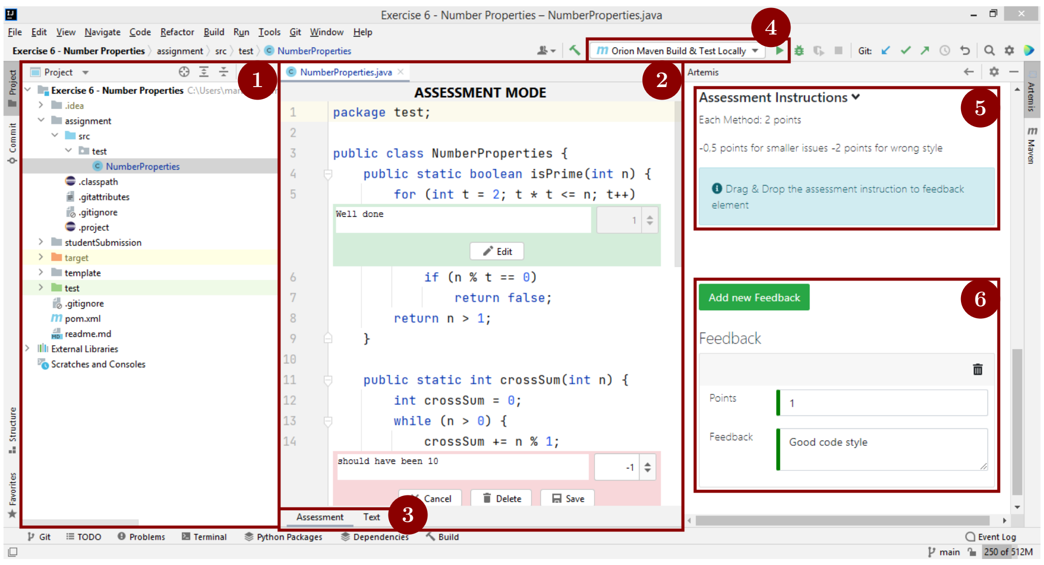
Exercise 6 - Number Properties [8 points]

96%, 15 of 18 passed, 7.7 of 8 points (preliminary)

Number Properties

For this exercise, write a program that determines various

Figure 3.10: Proposed System: Screenshots of IntelliJ with Orion during the assessment with the editor in edit mode.



```java
package test;

public class NumberProperties {
    public static boolean isPrime(int n) {
        for (int t = 2; t * t <= n; t++)
```

```
Well done                                        1
```

```java
            if (n % t == 0)
                return false;
        return n > 1;
    }

    public static int crossSum(int n) {
        int crossSum = 0;
        while (n > 0) {
            crossSum += n % 1;
```

```
should have been 10                             -1
```

Each Method: 2 points

-0,5 points for smaller issues -2 points for wrong style

Drag & Drop the assessment instruction to feedback element

Feedback

Points: 1

Feedback: Good code style

Figure 3.11: Proposed System: Screenshots of IntelliJ with Orion during the assessment with the editor in assessment mode.

Chapter 4

System Design

This chapter describes how the requirements map to the solution domain, following the system design document template by Brügge and Dutoit [BD09, chapter 6–7]. We give an overview of the architecture in Section 4.1, derive design goals from the requirements in Section 4.2, model the subsystem decomposition in Section 4.3, and explain the hardware/software mapping in Section 4.4.

4.1 Overview

Following the constraints from Section 3.2.2, the proposed system needs to be integrated into the existing Artemis and Orion systems. The general architecture of the system is shown in Figure 4.1 as a UML component diagram using the syntax of Brügge and Dutoit [BD09, chapter 6.4].

Users interact with Artemis via the *Artemis Client*. The client is written in Angular[1] and uses a layered architecture style [BMR$^+$96, chapter 2.2] with the following two layers. The Orion subsystem in the client is part of the same layers.

- The **User Interface (UI) Layer** consists of Angular components [Fre20, chapter 17] that manage the user interaction by providing the GUI. The style is done using Bootstrap[2]. It registers user interactions and relays them to the service layer.

- The **Service Layer** performs the client side business logic and handles the user interactions. If necessary, it sends requests to the server to retrieve data.

[1] `https://angular.io`
[2] `https://getbootstrap.com`

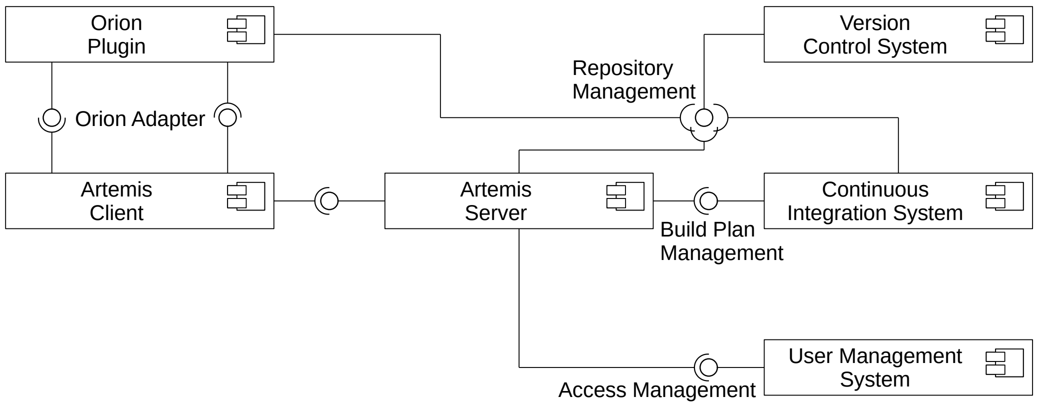


Figure 4.1: UML component diagram of the general architecture of Artemis and Orion adapted from Münch [Mü16, p. 44].

The *Artemis Server* is written in Java[3] using the Spring framework[4] and a MySQL[5] database. Artemis's server requires a *VCS* to manage the repositories for programming exercises and a *CIS* to perform the automatic test runs. Additionally, access management is handled by a separate *User Management System*.

The *Orion Plugin* displays the Artemis client using the Java Chromium Embedded Framework (JCEF) implementation provided by IntelliJ[6] and connects to it using the *Orion Adapter* [Ung20]. The plugin is organized using IntelliJ's services[7] without following a specific architecture style. User interactions get triggered in the Artemis client, sent through the adapter and then relayed to the relevant service.

4.2 Design Goals

In this chapter we derive design goals from the nonfunctional requirements from Section 3.2.2 and prioritize them. We also discuss conflicts between design goals and how we solve them.

Usability Since this thesis mainly aims to improve the assessment process for reviewers, usability has highest priority. We need to create a GUI simple enough to be capable of conforming to the limit of interactions to

[3] `https://www.java.com`
[4] `https://spring.io/`
[5] `https://www.mysql.com`
[6] `https://plugins.jetbrains.com/docs/intellij/jcef.html`
[7] `https://plugins.jetbrains.com/docs/intellij/plugin-services.html`

perform certain tasks defined by NFR1.1 and NFR1.2. Additionally, the GUI needs to be familiar to reviewers being used to the current assessment process, therefore the components need to look similar to the ones currently in use (NFR1.3). The GUI also needs to seamlessly integrate with IntelliJ (NFR1.4).

Usability vs. Readability [BD09, p. 245] Due to the different display size of Orion's integrated browser, Artemis's GUI needs to be rearranged to fit the limited space. Further altering the GUI improves usability, but also requires more code, reducing its readability since more special cases for different screen sizes are needed. In these cases we prioritize the usability, sacrificing readability if necessary.

Performance In order to make use of the usability improvements, the performance of the system must not be notably worse than the current process, otherwise the speedup by the automation would be nullified. The system must conform to the performance requirements NFR3.1, NFR3.2, and NFR3.3.

Performance vs. Maintainability and Portability In order to optimize performance, a large interface between Orion and Artemis is required to separately optimize every operation, e.g. by handling the addition, deletion and editing of feedback comments separately and only transferring the changed details. Such a large interface, however, is also more difficult to maintain and decreases portability since the systems are more coupled and less flexible. As long as we stay within the required performance thresholds, we prefer to sacrifice a bit of performance in order to keep the interface smaller and therefore easier to maintain, e.g. by performing an update operation transferring all feedback comments for every change.

Extensibility Some components of Artemis's client are changed for users connecting via Orion. The two variations of the components should be only loosely coupled; Artemis's components should not be dependent on Orion but instead provide generic extension points. This allows for easier extensions with e.g. future plugins by removing the direct coupling to Orion. It also improves modifiability since Artemis's components can be changed independent from Orion.

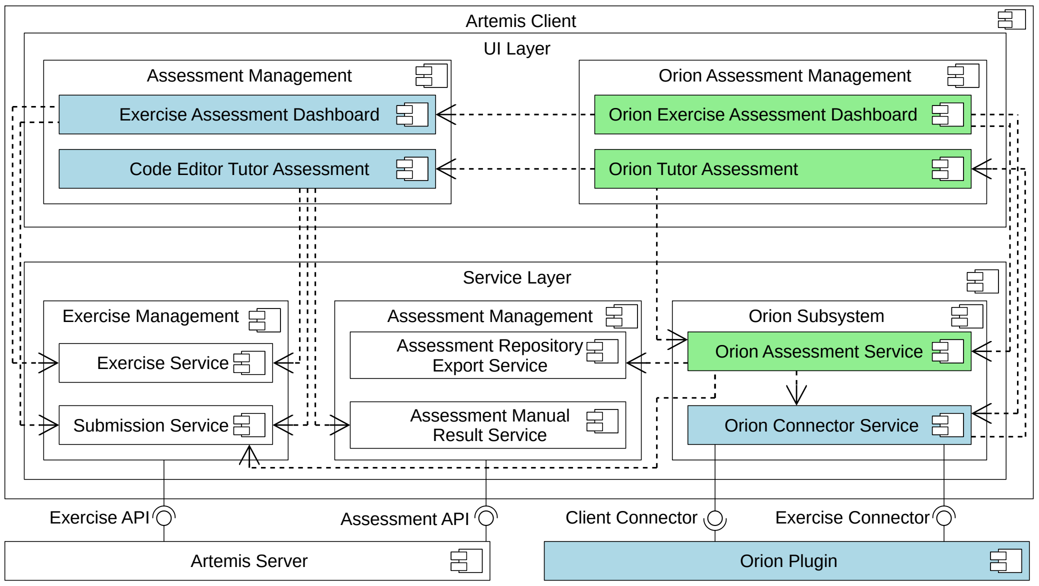


Figure 4.2: UML component diagram of the relevant parts of the Artemis client. Added components are colored green, modified components are colored blue.

4.3 Subsystem Decomposition

This section explains the detailed decomposition of the Artemis client in Section 4.3.1 and of the Orion plugin in Section 4.3.2.

4.3.1 Client Decomposition

The decomposition of the relevant client components is shown in Figure 4.2 as a UML component diagram. On the UI layer, two Artemis components are important for the manual assessment process of programming exercises: the *Exercise Assessment Dashboard* lets the user manage their assessments for an exercise by providing a list of all assessed submissions, both finished and unfinished, offering to start new assessments, and displaying general information about the exercise, e.g. the total assessment progress. The *Code Editor Tutor Assessment*, shown in Figure 3.2, allows to perform the actual assessment by displaying the student's code in a code editor and letting the reviewers leave assessment comments. Both of these components need to be modified to provide extension points.

These extension points are then used by the newly created *Orion Exercise Assessment Dashboard* (shown in Figure 3.9) and *Orion Tutor Assessment* (shown in Figure 3.10 and Figure 3.11 on the right side) which replace the assessment in the code editor with the assessment in Orion. The nature of these extension points is elaborated on in Section 5.1. Business logic like the preparation of downloading submissions is delegated to the newly created *Orion Assessment Service.* This service then delegates to existing services like the *Submission Service* to retrieve submissions to assess and the *Assessment Repository Export Service* to get a download file of the student's code, reusing the same methods that are used by Artemis's current components. The services then either delegate to further services or retrieve the requested data from the server using the *Exercise API* or *Assessment API.*

The communication with the Orion plugin is handled by the *Orion Connector Service*, which serves as a façade [GHJV07, chapter 4.5] for all Orion subsystems in the client [Ung20, chapter 5.3.1]. The service is extended with the new operations required for the assessment. Operations triggered in the components, like starting and downloading a new assessment, first get processed in the *Orion Assessment Service* and are then relayed to the *Orion Connector Service*, which sends the data to the plugin through the *Exercise Connector.*

Updates from the plugin, like an update of the feedback comments, are sent from the plugin through the *Client Connector.* The *Orion Connector Service* relays them to the *Orion Tutor Assessment*, which further relays them to its *Code Editor Tutor Assessment*, treating the updates as if they were made by the code editor.

4.3.2 Orion Decomposition

The decomposition of the relevant components in the Orion plugin are shown in Figure 4.3 as a UML component diagram. The communication from the Artemis client concerning exercises is handled by the *Exercise Connector.* All new operations like the download of assessments and submissions have been added to it. The operations are then handled by the *Orion Exercise Service*, which processes the requests and then delegates to the *Orion Git Adapter* to perform the download operation and the *Exercise Registry* to store Artemis's exercise data. The service and the registry have been extended to also manage assessment projects. The configuration of the repositories is handled by the *Instructor Project Creator* for instructor projects or by the newly created *Tutor Project Creator* for assessment projects.

The assessment comments are handled by the *Orion Assessment Service.* The feedback is initialized by Artemis through the *Exercise Connector* and any local updates get sent to Artemis with the *Client Connector.* In order to display the feedback, the new *Orion Editor Provider* has been added. Whenever a file belonging to an assessment is opened, the provider opens the *Orion*

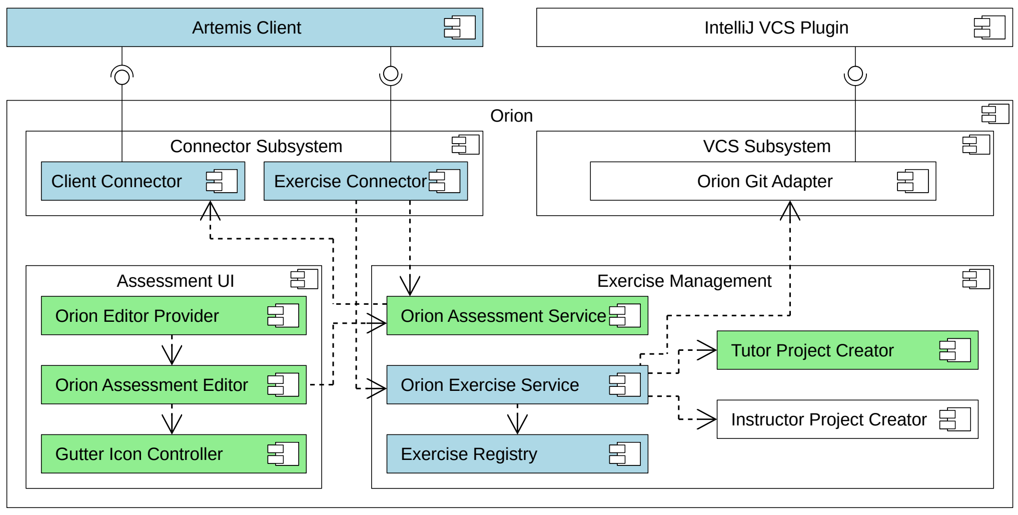


Figure 4.3: UML component diagram of the relevant parts of the Orion plugin. Added components are colored **green**, modified components are colored **blue**.

Assessment Editor as well as the normal editor. The *Orion Assessment Editor* then queries the relevant feedback from the *Orion Assessment Service* and displays it inline. The editor also uses the *Gutter Icon Controller* which displays the icons to add new feedback comments. Adding, editing or deletion of comments is then synchronized with the *Orion Assessment Service*, relayed to the *Client Connector* and from there sent to Artemis.

4.4 Hardware/Software Mapping

Figure 4.4 displays the hardware/software mapping of Artemis and Orion as UML deployment diagram [BRJ05, chapter 26], adapted from Ungar [Ung20], since the deployment of the plugin did not notably change.

The *Artemis Server* is deployed in the *University Data Center* along with its *Version Control Server*, the *Continuous Integration Server*, and the *User Management Server*. These servers interact as described in Section 4.1. The *Artemis Server* also supports integrating an external *Course Platform*. The *Continuous Integration Server* delegates its build to its internal *Build Agents*, but can also use remote build agents from an *Infrastructure-as-a-Service (IaaS) Provider* to increase performance.

The reviewer interacts with the system through *IntelliJ*, which needs to be installed on the *Reviewer Machine*. In *IntelliJ*, *Orion* is installed as a

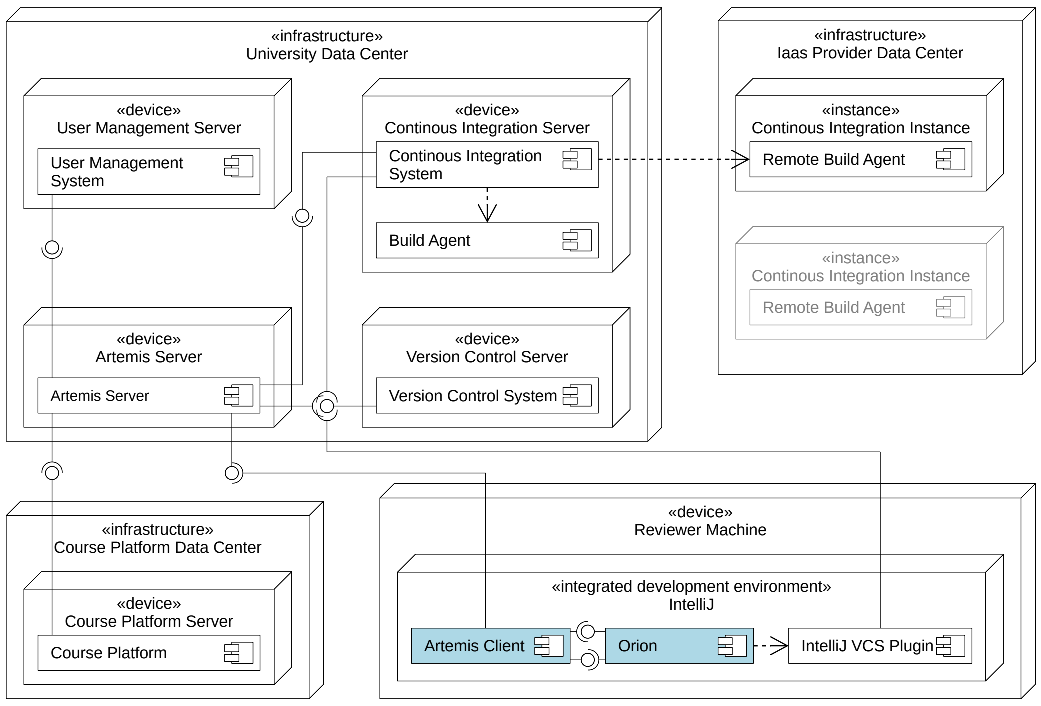


Figure 4.4: UML deployment diagram of Artemis's and Orion's infrastructure, adapted from Ungar [Ung20]. Modified components are colored blue.

plugin. *Orion* displays the *Artemis Client* using the integrated browser and communicates with it through the Orion adapter. It also delegates to the *IntelliJ VCS Plugin*, which can interact with the *Version Control Server* to download repositories.

Chapter 5

Object Design

This sections discusses details of the implementation that optimize the system model. We show how we achieved low coupling between Artemis's and Orion's subsystems in the Artemis client in Section 5.1 and how we prevented the system from reaching invalid states during loading in Section 5.2.

5.1 Decoupling Orion from Artemis

Orion has some dedicated code in the Artemis web client, e.g. to display the Orion specific buttons and the required services to connect them to the plugin. One of the affected components is the `CourseExerciseDetailsComponent`. We first explain the purpose of this component in Section 5.1.1, then describe how the current system implements these extensions and why this approach is flawed in Section 5.1.2, and finally present a refactoring to address these flaws in Section 5.1.3.

5.1.1 Overview

The `CourseExerciseDetailsComponent` is the component shown to students while solving an exercise, see Figure 3.3. It presents the problem statement as well as the current result. At the top of the component it displays the `ExerciseDetailsStudentActions`. For programming exercises, these actions can be in one of three states:

- If the student did not yet start the exercise, a button to start the exercise is shown, which, when clicked, creates the participation.
- After starting the exercise, the component shows two buttons: One to receive a link for the repository to clone it, and one to open the online code editor to solve the exercise.

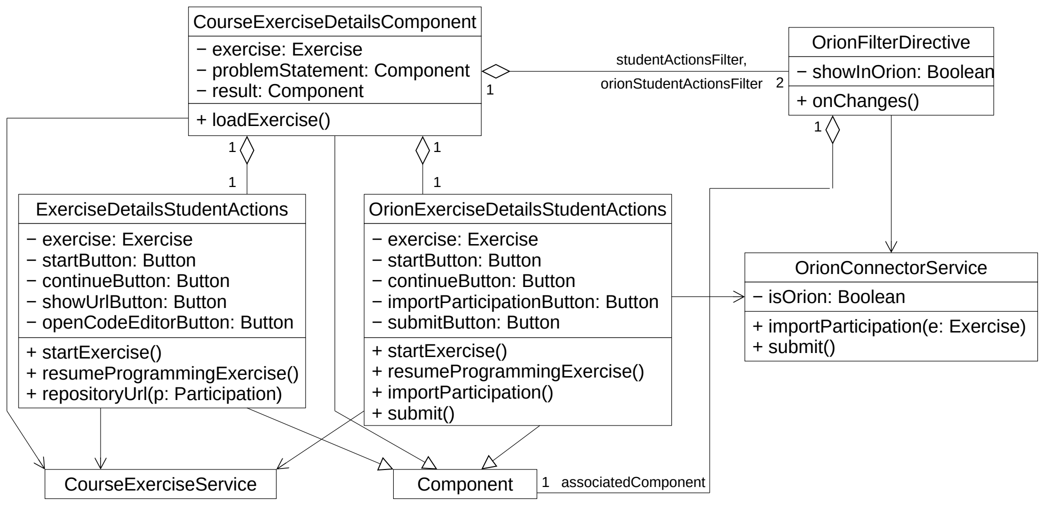


Figure 5.1: UML class diagram of the previous implementation of the `CourseExerciseDetailsComponent` and associated components and services

- If the participation did get inactive, a button to continue the exercises is displayed, which reactivates the participation.

If the user connects using Orion, the actions still have a start and continue button, however, instead of the repository link and the code editor, two special buttons appear:

- If the opened IntelliJ project is not the one belonging to the exercise, a button to download or reopen the exercise in Orion is shown.
- If the exercise is open in Orion, the submit button is displayed.

5.1.2 Current Approach

The current system has two different ways of implementing these extensions into the client, which are both present in the example. The relevant components, directives, and services are modeled as a UML class diagram [BRJ05, chapter 8] in Figure 5.1.

The first approach is present in the `ExerciseDetailsStudentActions` and the `OrionExerciseDetailsStudentActions`. Both components can be used interchangeably and have all required buttons as well as the implementation of their click handlers. Button clicks get processed and then forwarded to the `CourseExerciseService` or, for Orion calls, to the `OrionConnectorService`.

The second approach is present in the `CourseExerciseDetailsComponent`. It has both variations of student actions as child components. To organize which of the two student actions gets displayed, it attaches an `OrionFilterDirective` to both of them, setting `showInOrion` for the `ExerciseDetailsStudentActions` to false and for the `OrionExerciseDetailsStudentActions` to true. The filter directive then connects to the `OrionConnectorService` to receive whether the user uses Orion. Using this information and the internal `showInOrion`, it sets its associated component to either visible or invisible.

Both of these approaches are flawed. The first approach generates code duplication. In this case, the start exercise and continue exercise buttons and the handling of their clicks are duplicated in both components. This worsens maintainability and also yields the high risk of a developer overlooking one of the places and only changing one instance of the duplicated code, potentially breaking the other instance. The second approach creates a coupling between the `CourseExerciseDetailsComponent` and the `OrionConnectorService`, indirectly through the `OrionFilterDirective`, even though the Orion classes belong to their own, different subsystem. This also worsens maintainability since now changes in one subsystem potentially impact other, unrelated subsystems. Additionally it increases the components' complexity since they needs to be aware of all potential variations.

5.1.3 Refactored System

A refactoring to a more generic solution is required to address the decreased maintainability and to achieve the design goal extensibility. The new, refactored classes are modeled in Figure 5.2. In order to get rid of the static link to Orion, we introduce a new directive, the `ExtensionPointDirective`. It has two references to any component, a default component and an override component. If the override is undefined, the default component is rendered, otherwise the override is rendered. It also allows to pass any arbitrary context to the override component. The directive is similar to Angular's `ngTemplateOutletDirective`[1], the difference being that Angular's directive does not support any default, it only either displays one component or nothing.

Instead of using the `OrionFilterDirective`, the `CourseExerciseDetailsComponent` now only provides a generic extension point for the student actions. It sets the default to the `ExerciseDetailsStudentActions` and leaves the override undefined, displaying the regular Artemis components. For Orion users a different component is created, the `OrionCourseExer-`

[1] `https://angular.io/api/common/NgTemplateOutlet`

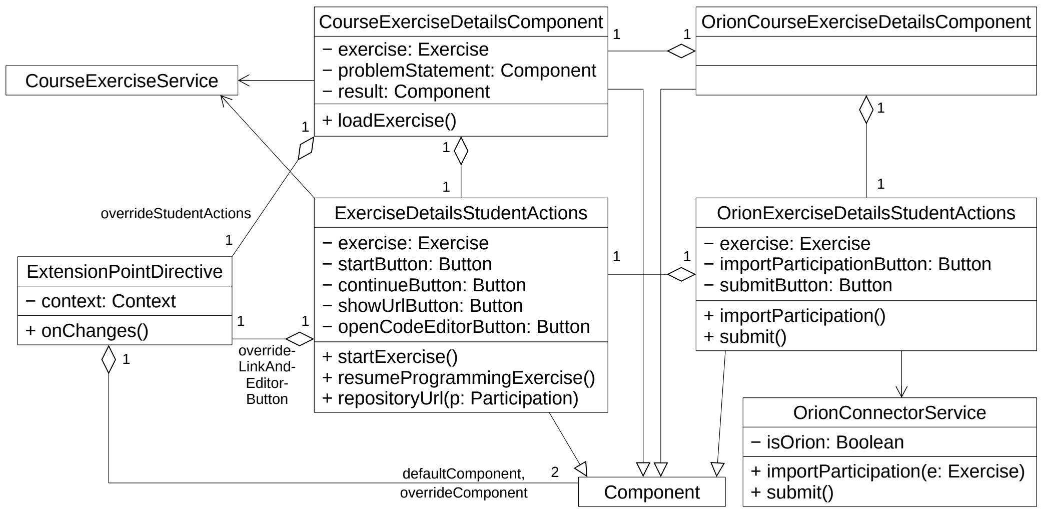


Figure 5.2: UML class diagram of the refactored implementation of the `CourseExerciseDetailsComponent` and associated components and services

`ciseDetailsComponent`. This component only displays the regular `Course-ExerciseDetailsComponent`, but sets its extension point's override to the `OrionExerciseDetailsStudentActions` by injecting it using Angular's content projection [Fre20, chapter 17][2]. This triggers the extension point to instead display the Orion buttons.

Similarly, the `ExerciseDetailsStudentActions` provide an extension point to override the buttons to show the download URL or open the code editor. The `OrionExerciseDetailsStudentActions` then display the regular actions but set the override to their own import and submit button. They no longer need a copy of the start and continue buttons since it can now reuse them from the regular actions.

This refactoring addresses both issues of the previous approaches: By providing generic extension points we remove the coupling between the regular Artemis components and Orion. This is easily extendable, since any future extending component can reuse the same extension points without any changes. It also removes the duplication by allowing the Orion components to easily reuse the shared code. Changes in Artemis's components automatically apply to the Orion components too, unless the extension point itself is altered. The distinction between Orion users and normal users is now not done by the components but instead by Angular's router [Fre20, chapter 25-

[2] see also `https://angular.io/guide/content-projection`

27][3], which decides which of the components get displayed. This solution with extension points, while only illustrated in one example in this section, can be used throughout the client for every component Orion uses.

5.2 Preventing Invalid States

While loading new submissions or while starting IntelliJ, several operations run simultaneously: users can open editors and already view files during the initialization of Orion's integrated browser. This could lead to erroneous states if users manage to add feedback comments before the feedback from Artemis has been loaded, potentially creating two different versions of feedback. This section describes how the implementation of the involved classes ensures no such feedback conflict can happen.

The initialization of the relevant classes is shown in Figure 5.3 as a UML communication diagram [BD09, p. 59] (also referred to as collaboration diagram [BRJ05, chapter 18]). Upon starting `IntelliJ` with an assessment project, the IDE simultaneously starts creating the `OrionBrowserService` and reopens all previously opened files, which, for files from the assessment, also creates an `OrionAssessmentEditor`.

The editors are created first, due to the browser requiring more complex initialization, which can take several seconds or more, depending on the system load. At the end of initialization, the `OrionAssessmentEditor` queries the `OrionAssessmentService` for inline feedback belonging to the opened file. At this time, the service is not yet initialized and returns nothing. The editor therefore skips the creation of its `GutterIconController`, disabling the creation of new comments. Until the browser is loaded, reviewers can view the files, but cannot interact with the assessment.

When the browser finishes its creation, it automatically loads the `OrionTutorAssessment` in the Artemis client. This component loads the current feedback stored on the server and sends it to the `OrionAssessmentService` via the Orion adapter as described in Section 4.3.1. The service initializes its local feedback and afterwards requests all open `OrionAssessmentEditors` to reinitialize. The editors query the service again. This time, since the feedback has been initialized, the service returns the correct feedback. The editors display the feedback belonging to their opened file and initialize their `GutterIconController` to enable reviewers to add new comments.

With this strategy we forced the parallel initialization into the correct order by disabling the creation of new comments until synchronization with Artemis is ensured. This prevents the system from reaching a problematic

[3] see also `https://angular.io/guide/router`

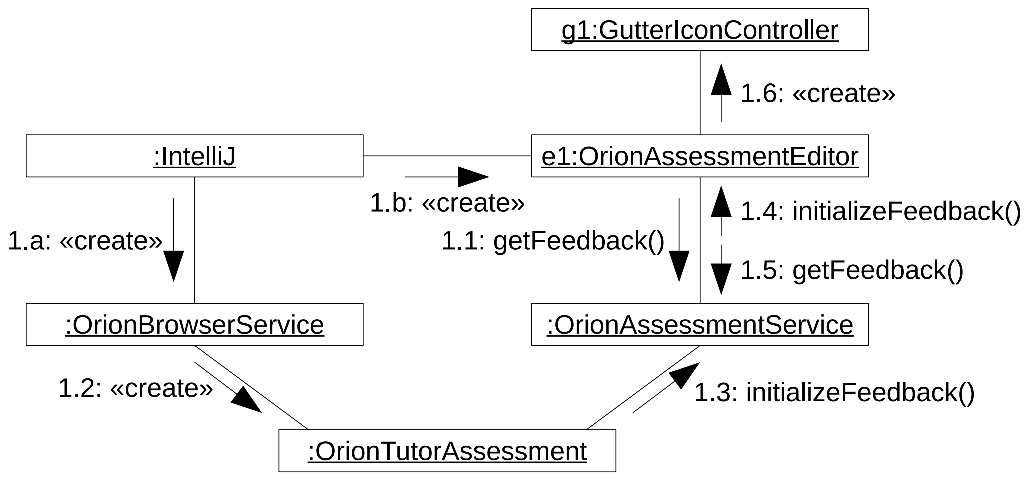


Figure 5.3: UML communication diagram of the initialization of the Orion browser and Orion assessment editors.

state with the local feedback getting out of synchronization, potentially creating different, conflicting versions of the same feedback.

Chapter 6

Summary

We summarize our findings in this chapter by first listing the status of our features in Section 6.1, describing our conclusions in Section 6.2 and giving an outlook on future work in Section 6.3.

6.1 Status

This section describes the development status of all functional requirements from Section 3.2.1 and nonfunctional requirements from Section 3.2.2. We evaluate the status using the following scale:

● **Implemented**. The requirement is entirely fulfilled.

◐ **Partially Implemented**. The requirement is only fulfilled partially with some additional work required.

○ **Not Implemented**. The requirement is not fulfilled, future work is necessary.

Table 6.1 lists the status of the functional requirements concerning how to download and execute student submissions. The requirements are mostly implemented. Reviewers can start their assessment in Orion with Orion downloading and configuring the required repositories. They can then also download submissions of students, view the code in IntelliJ, edit it, and execute the automatic tests locally. Highlighting the student's changes proved to be more complicated than expected due to the limitations by IntelliJ and had to be postponed. While the code is functional to also handle complaints and more feedback requests, the GUI has not yet been adapted to enable it.

The status of the requirements for the manual assessment is listed in Table 6.2. We enabled reviewers to view the problem statement, assessment

	Requirement	Status
FR1.1	Start Assessment of Exercise	●
FR1.2	Assess Submission	●
FR1.3	View Code in IDE	●
FR1.4	Highlight Changes	○
FR1.5	Edit and Execute Student's Code	●
FR1.6	Support Feedback Requests and Complaints	◐

Table 6.1: Status of the functional requirements regarding the download and execution of student submissions

	Requirement	Status
FR2.1	View Problem Statement and Assessment Instructions	●
FR2.2	View Result	●
FR2.3	View Feedback	●
FR2.4	Add, Edit, and Delete Feedback	◐
FR2.5	Save Assessment	●
FR2.6	Cancel Assessment	●
FR2.7	Use Structured Grading Instructions	○

Table 6.2: Status of the functional requirements regarding the manual assessment in Orion

instructions, result, and feedback in IntelliJ. Inline feedback is shown inline. Reviewers are able to add, edit, and delete feedback comments in their IDE; however, feedback can only be referencing a line or be general. We did not manage to implement feedback referencing a file; this is also not yet supported by Artemis. The assessment can be both saved and cancelled, similar to the previous assessment workflow. We also did not manage to implement support for Structured Grading Instructions (SGIs) in the given time.

We list the status of the nonfunctional requirements in Table 6.3. By integrating the assessment process into the IDE, we managed to reduce the amount of interactions required to execute the students' code without raising the amount of interactions required to perform the assessment. The GUI of the new components, most notably the inline assessment comments, looks similar to the current GUI, with the current GUI shown in Figure 3.2 and the new GUI displayed in Figure 3.11. We reused components provided by IntelliJ to ensure a consistent style. This, however, clashes with Artemis's style. Most notably, Artemis uses a bright background with dark text, whereas IntelliJ also offers styles with dark background and bright text. Future work is required to also offer such a style for Artemis with a different color scheme,

	Requirement	Status
NFR1.1	Usage Complexity (Start Assessment)	●
NFR1.2	Usage Complexity (Add Feedback Comment)	●
NFR1.3	GUI Consistency with Artemis	●
NFR1.4	GUI Consistency with IntelliJ	◐
NFR2.1	Download Large Repositories	●
NFR2.2	Illegal Input	●
NFR3.1	Communication Artemis Client and Server	●
NFR3.2	Load Time	●
NFR3.3	Save Time	●

Table 6.3: Status of the nonfunctional requirements

which then needs to be integrated in Orion to ensure the colors fit in all available color modes.

We fulfilled the required thresholds for reliability with the implementation being able to handle the required repository size of 10 megabytes. Raising that limit would require the download process to be rewritten to split the data into smaller packages that are sent separately. By reusing the already implemented components from Artemis, we also ensured feedback from Orion is validated the same way, guaranteeing only valid feedback is stored. The given performance requirements are also met with the operations being implemented sufficiently lightweight to not cause notable additional delay.

6.2 Conclusion

By integrating manual assessment into Orion, we enabled reviewers to edit and execute the students' code immediately without requiring further download or configuration. This allows reviewers to try out potential fixes for the code and to receive an automatic test result for it at no additional effort. This helps to accelerate the assessment process for faulty submissions as well as to improve the feedback quality by increasing the reviewer's understanding of the submission.

We also integrated the ability to create, edit, and delete feedback comments into Orion, enabling reviewers to perform manual assessment without using any other program. We thereby eliminated the media disruption caused by the previous assessment process.

We introduced a generic extension point directive into the Artemis client to decouple Orion from Artemis while reusing the shared code. This improves the maintainability of the code by eliminating code duplication as well as the extensibility by removing the coupling to Orion, allowing other, unrelated systems to reuse the same extension points without further changes.

6.3 Future Work

With this thesis, Orion now supports the three mayor tasks of the programming exercise workflow: creating exercises, solving exercises and assessing exercises. Nonetheless, more features could further improve the user experience. Several suggestions are discussed in this section.

Support SGIs: As described in Section 6.1, we did not manage to implement SGIs in this thesis. Supporting them would, however, be very helpful, since they enable reviewers to immediately use certain feedback comments without having to type them. This both accelerates the assessment process and unifies the feedback by encouraging the reviewers to use the same feedback for all submissions. Support for SGIs should work as described in FR2.7. The grading instructions should be displayed in Artemis in Orion's browser and it should be possible to drag-and-drop them into feedback comments in Orion.

Refactor Orion Into Its Own Micro Frontend: While the refactoring described in Section 5.1 does improve the decoupling of Orion's and Artemis's subsystems in the Artemis client, it is not optimal. After the refactoring, the Angular router [Fre20, chapter 25–27][1] has to decide which component should be displayed. Currently, it just queries the Orion Connector Service. While the coupling with Orion has been removed from the components, it is now present in the router, which, due to its decentralized nature with every module defining its own routes, still creates a high coupling between Orion and many different routes.

In addition, Orion's code is still part of the same modules as Artemis, causing it to be sent to any client, even if they use a regular browser. While the components don't get displayed, they are still transferred, costing performance. Further refactoring Orion into its own modules and its own micro frontend would improve this by centralizing the detection of Orion users. Lazily loading Orion only for users connecting via Orion would ensure no unnecessary data from Orion is sent to users connecting via a regular browser as well as completely decouple Orion in the client.

[1] see also `https://angular.io/guide/router`

Enable Other Programming Languages in Orion: Only Java[2] is currently fully supported by Orion as programming language for programming exercises. Artemis, however, at the time of writing of this thesis, offers eight languages[3]. IntelliJ supports three of them: Java, Python[4], and Kotlin[5]. Additionally, Jetbrain's CLion[6], which is compatible with Orion, offers support for C and Swift[7]. Ensuring compatibility of Orion with these languages would enable non-Java courses to also use Orion. In order to support a new language, the configuration of exercises needs to be extended to select the correct environment and organize the files according to the needs of the specific language.

[2] `https://www.java.com`

[3] `https://github.com/ls1intum/Artemis/blob/091997fd53bfec901840242cddd8635e984b1b95/src/main/java/de/tum/in/www1/artemis/domain/enumeration/ProgrammingLanguage.java`

[4] `https://plugins.jetbrains.com/plugin/631-python`

[5] `https://plugins.jetbrains.com/plugin/6954-kotlin`

[6] `https://www.jetbrains.com/clion`

[7] `https://plugins.jetbrains.com/plugin/8240-swift`

Appendix A

Statistics of the TUM Department of Informatics

The following table lists the number of students and research assistants at the department of informatics at TUM according to their website[1].

year	number of students	number of research assistants	number of students per research assistant
2013	3555	395	9.0
2014	3815	407	9.4
2015	4240	392	10.8
2016	4744	397	11.9
2017	5399	397	13.6
2018	5986	408	14.7
2019	6458	454	14.2
2020	7444	480	15.5

Table A.1: Number of students and research assistants at the TUM Department of Informatics from 2013 to 2020

[1] https://www.in.tum.de/en/the-department/profile-of-the-department/facts-figures

List of Figures

List of Tables

Bibliography

[BBL16] Selim Buyrukoglu, Firat Batmaz, and Russell Lock. Increasing the similarity of programming code structures to accelerate the marking process in a new semi-automated assessment approach. In *2016 11th International Conference on Computer Science Education (ICCSE)*, pages 371–376, 2016.

[BD09] Bernd Bruegge and Allen H Dutoit. *Object Oriented Software Engineering Using UML, Patterns, and Java.* Prentice Hall, 2009.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – Volume 1: A System of Patterns.* Wiley Publishing, 1996.

[BRJ05] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley object technology series. Addison-Wesley, 2005.

[EKN+11] Emma Enstrom, Gunnar Kreitz, Fredrik Niemela, Pehr Soderman, and Viggo Kann. Five years with kattis – using an automated assessment system in teaching. In *Proceedings of the 2011 Frontiers in Education Conference*, FIE '11, pages T3J-1–T3J-6, USA, 2011. IEEE Computer Society.

[Elh20] Hanya Elhashemy. Structured grading criteria for the assessment of exercises in artemis. Bachelor's thesis, Technical University of Munich, April 2020.

[Fre20] Adam Freeman. *Pro Angular 9: Build Powerful and Dynamic Web Apps.* Apress, fourth edition, June 2020.

[GHJV07] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley professional computing series. Addison-Wesley, Boston, 2007.

[Gra92] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement.* Prentice-Hall, Inc., USA, 1992.

[GSS21] Florian Glombik, Johannes Stöhr, and Niclas Schümann. Improving the tutor experience in artemis. Bachelor's thesis, Technical University of Munich, August 2021.

[IS15] David Insa and Josep Silva. Semi-automatic assessment of unrestrained java code: A library, a dsl, and a workbench to assess exams and exercises. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, pages 39–44, New York, NY, USA, 2015. Association for Computing Machinery.

[KS18] Stephan Krusche and Andreas Seitz. Artemis: An automatic assessment management system for interactive learning. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 284–289, New York, NY, USA, 2018. Association for Computing Machinery.

[lCY21] Francisco Javier De las Casas Young. Manual assessment of programming exercises in artemis. Master's thesis, Technical University of Munich, January 2021.

[Mon17] Josias Montag. Conducting interactive programming exercises in online courses. Master's thesis, Technical University of Munich, April 2017.

[Mü16] Dominik Münch. Conducting interactive programming exercises in large lectures. Master's thesis, Technical University of Munich, November 2016.

[Pie13] Vreda Pieterse. Automated assessment of programming assignments. In *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research*, CSERC '13, pages 45–56, Heerlen, NLD, 2013. Open Universiteit, Heerlen.

[Ung20] Alexander Ungar. Development of an ide plugin for artemis. Master's thesis, Technical University of Munich, February 2020.

[ZKF11] Daniel M. Zimmerman, Joseph R. Kiniry, and Fintan Fairmichael. Toward instant gradeification. In *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE T)*, pages 406–410, 2011.