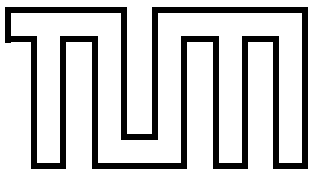


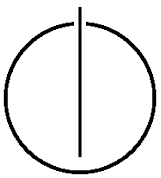
DEPARTMENT OF INFORMATICS

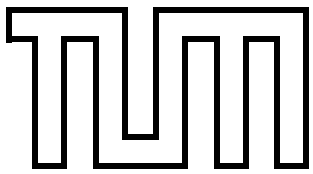
TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Development of an IDE Plugin for ArTEMiS

Alexander Ungar





DEPARTMENT OF INFORMATICS

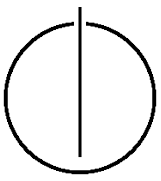
TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Development of an IDE Plugin for ArTEMiS

Entwicklung eines IDE Plugins für ArTEMiS

Author:	Alexander Ungar
Supervisor:	Prof. Bernd Brügge, Ph.D.
Advisor:	Dr. Stephan Krusche
Date:	17.02.2020



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

Munich, 17.02.2020 Alexander Ungar

Acknowledgements

I would like to use this opportunity to express my gratitude to everyone, who has supported me during the last six months.

First of all, I would like to thank my advisor Dr. Stephan Krusche. His feedback and insights not only shaped this thesis, but also me and my views as a software developer. Despite being very busy and involved in many projects, he always finds the time to show an interest in the work of his students.

Beyond that, I would like to thank the whole Artemis developer team and especially Thilo Behnke. I was always looking forward to our meetings and working together on the platform. This experience would not have been the same without the people I met.

I also want to thank my friends, in particular Denis, Samih and Sarah. You have given me the motivation and strength to push through this tough time.

Lastly, I want to thank my parents, who always had my back and never doubted me, not only during the last few months, but during my entire time at university.

Abstract

Artemis is an automated individual feedback system for interactive learning, which allows for the creation of programming exercises. Students, as well as instructors can use an integrated online code editor to work on tasks involving the exercise participation or administration. However, the editor is missing advanced functionalities like code completion, live compilation, or debugging. This can be overcome by using an IDE, which commonly offers more sophisticated features. The alternative IDE setup introduces a media disruption as some actions require interacting with the *Artemis* client. Instructors face additional difficulties since the administration of exercises involves building source code from a combination of multiple repositories.

In this thesis, we combine both approaches to solve these limitations and develop the IDE plugin *Orion*, which leads to an integrated user experience. *Orion* unifies *Artemis* with a development environment by incorporating the programming exercise administration and participation processes into the IDE. Students are able to review *Artemis* results and feedback directly in the IDE, while instructors are provided with the toolset to edit programming exercises in one single IDE project. The plugin hides complex details of the exercise workflow and lowers the entry barrier.

We developed a first prototype of *Orion* for IntelliJ and sent it to beta users, who provided feedback for the formative improvements. Following the release of the plugin, students and instructors in the course *Introductions to Software Engineering* in the next semester can use *Orion* to work on *Artemis* programming exercises without media disruptions. In the future, *Orion* can be extended to integrate the code review process of teaching assistants. Hints could be created and displayed directly in the IDE. Team based exercises in *Orion* would allow students to collaboratively solve tasks inside multiple IDEs.

Zusammenfassung

Die interaktive Lernplattform *Artemis* ermöglich das Erstellen von Programmieraufgaben, für welche Nutzer automatisiertes Feedback erhalten können. Sowohl Studenten, als auch Kursleiter nutzen den integrierten online Code Editor um Aufgaben zu lösen und zu verwalten. Dem Editor selbst fehlen jedoch Funktionen wie die automatische Code Vervollständigung, Live Kompilierung, oder Debugging. Eine IDE hat diese Nachteile nicht, da sie typischerweise über ausgefeiltere Features verfügt. Dieses alternative Setup bringt jedoch einen Medienbruch mit sich, da manche Schritte Interaktionen mit der *Artemis* Webanwendung zwingend erfordern. Kursleiter stehen vor weiteren Schwierigkeiten, da die Administration von Aufgaben die Kombination von Code aus mehreren Repositories erfordert.

Mit der Entwicklung des IDE Plugins *Orion* lösen wir diese Limitierungen durch das Kombinieren beider Ansätze und schaffen somit eine einheitliche Nutzererfahrung. *Orion* vereint *Artemis* mit einer Entwicklungsumgebung indem Administrations- und Teilnahmeprozesse von Programmieraufgaben in einer IDE eingebunden werden. Studenten wird die Analyse von Feedback und Ergebnissen direkt in der IDE ermöglicht, während Kursleiter das Toolset erhalten, durch welches sie Aufgaben in einem einzelnen IDE Projekt bearbeiten können. Das Plugin vereinfacht komplexe Abläufe und sinkt die Einstiegshürde für Anfänger.

Wir entwickelten einen ersten Prototypen von *Orion* für IntelliJ und schickten ihn an Nutzer, deren Feedback für iterative Verbesserungen genutzt wurde. Nach der Veröffentlichung des Plugins können Studenten und Leiter des Kurses *Einführung in die Softwaretechnik* im nächsten Semester *Orion* nutzen um an Programmieraufgaben in *Artemis* ohne Medienbrüche teilzunehmen. Zukünftig kann *Orion* um die Integration des Reviewprozesses von Tutoren erweitert werden. Hinweise könnten direkt in der IDE erstellt und angezeigt werden. Teambasierte Aufgaben in *Orion* würden Studenten die kollaborative Arbeit über mehrere IDEs ermöglichen.

Contents

1 Introduction **3**
- 1.1 Problem . . . 4
- 1.2 Motivation . . . 6
- 1.3 Objectives . . . 7
- 1.4 Outline . . . 8

2 Background **9**
- 2.1 Integrated Development Environments . . . 9
- 2.2 Dependencies of Programming Exercises . . . 10

3 Related Work **13**
- 3.1 Test My Code . . . 13
- 3.2 JetBrains Edu Tools . . . 15
- 3.3 Coding Tools of the openHPI Platform . . . 16

4 Requirements Analysis **19**
- 4.1 Current System . . . 19
- 4.2 Proposed System . . . 20
  - 4.2.1 Functional Requirements . . . 21
  - 4.2.2 Nonfunctional Requirements . . . 23
- 4.3 System Models . . . 24
  - 4.3.1 Scenarios . . . 24
  - 4.3.2 Use Case Model . . . 26
  - 4.3.3 Analysis Object Model . . . 29
  - 4.3.4 Dynamic Model . . . 31

5 System Design **35**
5.1 Overview . . . 35
5.2 Design Goals . . . 37
5.3 Subsystem Decomposition . . . 38
5.3.1 Connector Components . . . 38
5.3.2 Build Components . . . 40
5.3.3 Exercise Components . . . 40
5.4 Hardware Software Mapping . . . 42

6 Object Design **45**
6.1 Support for the IntelliJ IDE . . . 45
6.2 Connecting Orion to Artemis . . . 47
6.3 Connecting Artemis to Orion . . . 48
6.4 Exercise services . . . 50
6.5 Remote Build Result Processing . . . 52

7 Summary **55**
7.1 Status . . . 55
7.2 Conclusion . . . 56
7.3 Future Work . . . 58

GUI Graphical User Interface

UI User Interface

CI Continuous Integration

CIS Continuous Integration System

VCS Version Controls System

DVCS Distributed Version Control System

SCCS Source Code Control System

IDE Integrated Development Environment

UI User Interface

IoC Inversion of Control

POJO Plain Old Java Object

MOOC Massive Open Online Course

TMC Test My Code

Chapter 1

Introduction

University courses face multiple challenges when teaching lectures at modern institutions. Especially with the increasing number of enrolled students[1], delivering the same level of competence and knowledge to all participants equally becomes a significant issue [MK10].

Solution approaches for the these problems are diverse. Evidence suggests that implementing modern approaches such as providing students with the possibility of an interactive learning experience, helps coping with the growing lack of interest and lowered participation rate in courses [KvFA17].

Artemis, an open source[2], automatic assessment management system for interactive learning [KS18] is a concrete example of how modern learning methods can both improve the acceptance rate of students and ease the workload on tutors and instructors. Students can interact with the platform by solving different types of exercises and receiving (partially automated) feedback. Meanwhile, instructors are provided with an environment, which enables them to create, release and assess exercises in a more compact and less complex way.

Specifically interesting in the context of this thesis is the concept of programming exercises, which was incorporated in the first version of the platform and a leading motivation behind the initial development [MK16]. Assessing programming exercises in courses with hundreds, or even thousands of participants is primarily limited by the number of instructors and teaching assistants. Therefore, automating this process using a combination of continuous integration (CI) and version control services (VCS) allows courses to scale more flexible while still being accessible to students on a beginner level.

[1] `https://de.statista.com/statistik/daten/studie/221/umfrage/anzahl-der-studenten-an-deutschen-hochschulen/`

[2] `https://github.com/ls1intum/Artemis`

This is achieved by making the underlying processes transparent to the user, who only has to checkout the assignment's code, work on it by e.g. using an integrated development environment (IDE) and then submit it back to the remote repository.

1.1 Problem

Currently, users have to always use the *Artemis* web client in order to interact with the platform. While this might be a good approach for text, quiz [SK18a] or even modeling exercises [SK18b, WK19], other components of the application can be further improved. One example would be the integrated online code editor [MK17, Beh19], which doesn't have the extensive toolkit and flexibility of a classic IDE like IntelliJ[3]. IDEs normally offer a range of helpful features such as debugging frameworks, automatic code completion, or live compilation, which have become a de facto standard in software development.

After starting a programming exercise, *Artemis* creates a participation for the student, which references a remote repository. This repository can be manually downloaded and imported into an IDE in order to modify it and solve the exercise. This involves performing multiple VCS operations (see figure 1.1), generally with the help of an external VCS client like Sourcetree[4]. Especially novices to software engineering have never worked with a VCS before and are therefore required to learn a range of new technologies in addition to an already difficult programming language. This approach has additional disadvantages, because it does not fit into the otherwise interlocking workflow of *Artemis*. Figure 1.2 illustrates the fragmentation of use cases between the IDE, *Artemis* and VCS. Interactions with the *Artemis* client are mandatory since this is the only way to *start* an exercise and *review* test results. VCS operations are necessary to modify the source code of a user's personal repository and *push* the changes back to *Artemis*. While these limitations can be circumvented by using the online code editor, the extended capabilities of an IDE are an important factor when solving more complex exercises. Hence, dealing with the resulting fragmentations and trade-offs is unavoidable in some cases.

Furthermore, the missing connections mean that *Artemis* cannot forward test results to the IDE, thus requiring users to switch back to the web client if they want to analyze feedback. Students should be able to start, solve and

[3] `https://www.jetbrains.com/idea/`
[4] `https://www.sourcetreeapp.com/`

submit a programming exercise without having to switch between different systems, i.e. without any media disruptions.

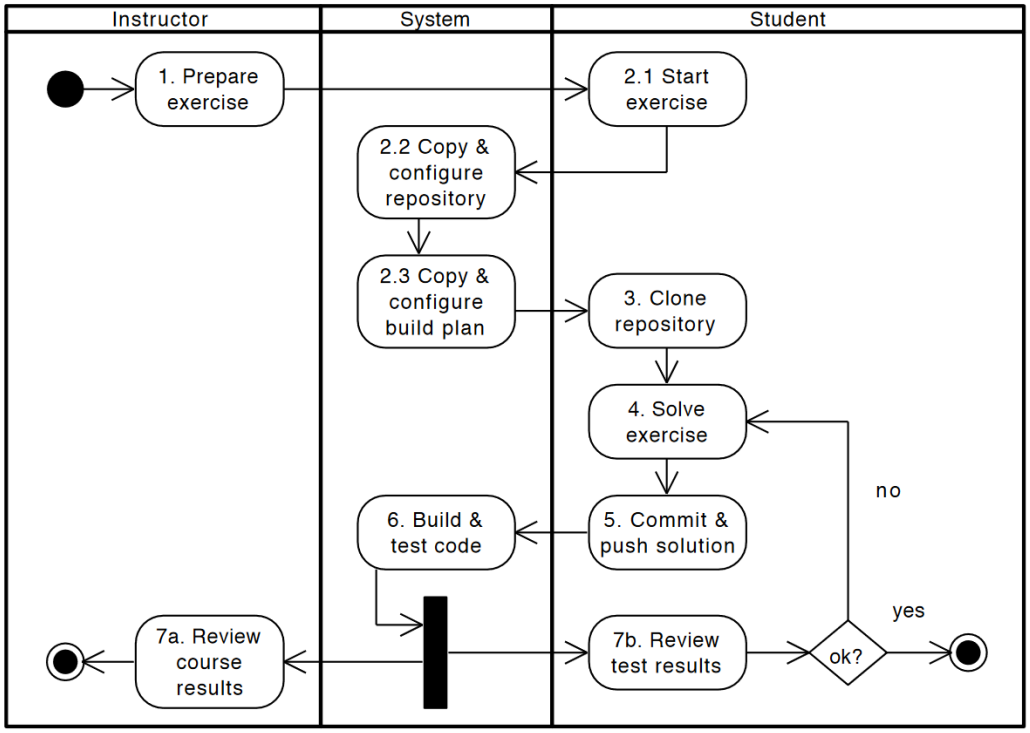


Figure 1.1: Activity diagram of the automated assessment process of Artemis [KS18]. Students have to perform various VCS operations (3, 5) and interactions with the IDE (4) and the *Artemis* client (2.1, 7b) in order to download, solve and submit an exercise.

Course instructors are affected by the same restrictions as they also have to modify source code and test it by submitting it to the *Artemis* servers. The administration of programming exercises requires the interaction with multiple repositories, because every exercise includes three *base participations*:

1. The *template participation* references the *template repository*, which contains the code every student receives when starting a new exercise.

2. The *solution participation* references the *solution repository*, which contains the sample solution achieving a full score.

3. The *test repository* contains all tests, based on which a participation receives a score and gets graded.

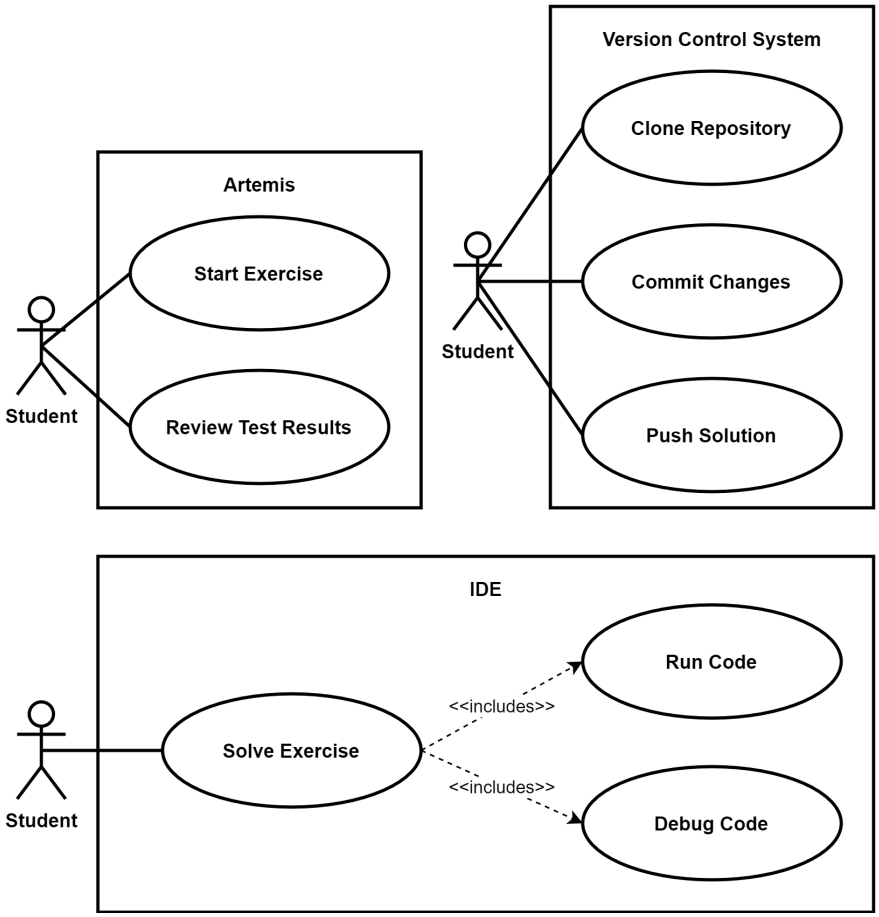


Figure 1.2: Use case diagrams of a student solving a programming exercise. The whole process is split up into three separate diagrams as it requires interacting with multiple clients and systems.

Instructors build and test these base participations locally before uploading the exercise to *Artemis*. This involves manual copy operations, that combine the tests with any of the base repositories. This is a time consuming and error-prone process, as any mistake can result in local build failures. Using standard IDE build tools is not an option, because these are processes unique to *Artemis*.

1.2 Motivation

Users of the system do not have a way of interacting with *Artemis* directly while still using their preferred IDE. With the presented approach, we aim to unify the process of solving and editing programming exercises and inter-

acting with the system. The main benefit therein lies in improving the user experience for both students and instructors:

Eliminate media disruptions Participants don't have to switch between a browser running the *Artemis* client and their IDE. The fragmented workflow is especially problematic for inexperienced students, who have to get familiar with multiple applications and user interfaces (UIs): VCS, *Artemis* and the IDE. This can have an adverse effect on the learning process since students might be overwhelmed with the amount of new input and solving the actual programming exercise becomes unnecessarily complex.

Exercise administration in one IDE project Instructors who administer exercises, deal with an even more fragmented workflow since every programming exercise relates to at least three base repositories. Editing these repositories simultaneously could be improved by migrating this process into an IDE and allowing instructors to modify, build and test an exercise using a single IDE project.

Summarized, the integration of *Artemis* itself into an IDE could simplify interactions with the system and lower the entry barrier for inexperienced users of the platform. The IDE should be incorporated into *Artemis'* system architecture and connected to all relevant subsystems.

1.3 Objectives

Based on the previous sections, we can derive the following three main objectives for this thesis:

Unify Artemis with a modern IDE *Artemis* is currently not integrated into an IDE at all. Users should be able to interact with the system from within the IDE without any media disruptions. We want to connect *Artemis* to the IDE and allow both systems to communicate with each other. A new IDE plugin should provide a bidirectional connection and enable an interactive and responsive learning experience.

Simplify VCS interactions Addressing the necessity to learn a complex VCS like Git by hiding complex VCS operations behind simplified actions lowers the entry barrier. Users should be able to interact with the platform without having to execute an overwhelming number of VCS commands. Instead, a simplified user interface would introduce them to the VCS and enhance the overall programming exercise workflow without requiring any prior knowledge.

Enhance build and test capabilities We want to build upon the work of previous authors [Beh19, MK17] on the code-editor regarding its display of test results by porting this functionality into an IDE. The plugin should be able to provide the same level of detail when reporting build results to the user, which enables a more responsive workflow and allows quicker reactions to negative feedback. Above that, introduce the possibility to run builds locally. As a result, instructors should be able to edit and debug programming exercises using a workflow, which requires fewer manual interactions and is less susceptible to errors.

1.4 Outline

Chapter 2 introduces technical background information relevant tot his thesis. *Chapter 3* analyzes functional and nonfunctional requirements and based on these, provides visualizations of the proposed system from different perspectives. *Chapter 4* decomposes the system into smaller units mapping the from the requirements derived designs onto subsystems. *Chapter 5* refines solution specific objects and introduces interfaces and operations used to implement the concrete system. *Chapter 6* concludes this thesis by reflecting on open and completed goals and gives an outlook on future work.

Chapter 2

Background

In this chapter, we elaborate on the technical details, which are relevant to the in this thesis presented designs and implementations. We first explain the basic concept of an integrated development environment (IDE) and how IntelliJ allows plugins to extend its offered functionalities. We then introduce continuous integration and version control systems, which are essential for programming exercises in *Artemis*.

2.1 Integrated Development Environments

Integrated Development Environments stand in contrast to regular text editors, which are limited in their capabilities and mostly only offer syntax highlighting as an additional feature, if any. While there is no binding definition, IDEs can be described as programs, which offer an extended toolkit for editing and executing source code during software development. In addition to the features of a regular editor, an IDE commonly bundles tools related to the compilation, debugging, testing and building of complex software projects. However, this list of tools is neither exhaustive, nor required for every IDE. As every programming language requires a special set of complementary software in order to work with it in a productive manner, IDEs can be just as flexible in their integrated features, depending on which languages they support.

IntelliJ is an IDE developed by JetBrains[1]. The core application (titled Community Edition) is released under the Apache 2.0 license[2] and its code open sourced [O'M02] and available on GitHub[3]. IntelliJ heavily relies on

[1] `https://www.jetbrains.com/idea/`
[2] `https://www.apache.org/licenses/LICENSE-2.0`
[3] `https://github.com/JetBrains/intellij-community`

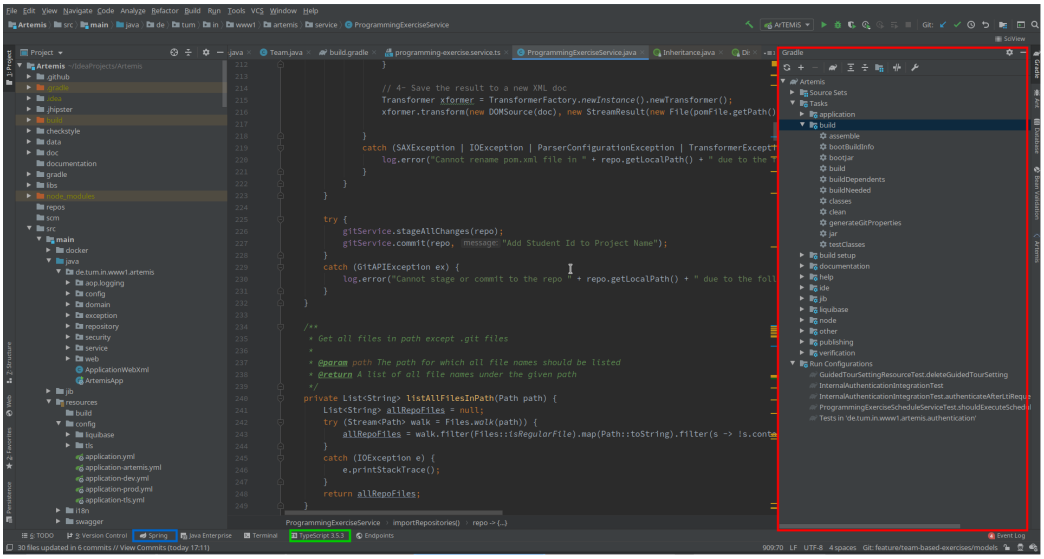


Figure 2.1: Screenshot of an opened software project in IntelliJ. Highlighted are components of different plugins: The Gradle plugin on the right side, the Java Spring plugin and the Typescript plugin on the lower side. The overall color theme got changed using the Darcula theme plugin[5]

the usage of plugins developed by either JetBrains, or any interested external developer. Together with the core application, these plugins can be installed in order to customize the installed IDE (e.g. by adding visual improvements like a different color theme), or extend it to add completely new functionalities. As an example, users can add support for a new programming language by installing the Python plugin[4]. One is also able to connect third-party platforms such as *Artemis* to the IDE by developing a dedicated plugin. This thesis follows the same approach by implementing a new IntelliJ plugin called *Orion*. Figure 2.1 shows a screenshot of an opened project in IntelliJ. The highlighted components are not part of the vanilla (unmodified) installation and got added by installing the specified plugins.

2.2 Dependencies of Programming Exercises

Programming exercises in *Artemis* depend on external systems, so that the history of code changes by users can be tracked and persisted. Furthermore, the test feedback process also requires continuously building changed code

[4] `https://github.com/JetBrains/intellij-community/tree/master/python`
[5] `https://github.com/vecheslav/darcula-darker`

and forwarding the results to the platform. Therefore, this sections covers the basics of version control (VC) and continuous integration (CI).

Version Control System The Source Code Control System (SCCS) from 1975 [Roc75] was the first major Version Control System (VCS). The core idea was the same as today in a modern VCS, which is to store the original file and all deltas (i.e. changes made to the file) which applied sequentially lead up to the most recent *version* of the file. Meanwhile, repositories can be defined as the grouping of multiple files, often related to one single software project. *Artemis* uses the distributed VCS (DVCS) *Git*[6], which has the advantage of being decentralized, meaning that every client holds the complete history of all files in a repository and does not have to rely on a central remote. As a result, every user can work on his personal repository, even offline and synchronize it at a later point in time with *Artemis* and any connected other remote. A simplified visualization of a DVCS can be found in figure 2.2.

Continuous Integration Continuous Integration describes the process of deploying the current version of a software project, often to a productive or test server, several times a day [Boo90, FF06]. In order to facilitate this continuous flow, the CI system (CIS) observers a VCS and reacts to every change by merging the update into the local repository on the CIS and then starts a new deploy. This is often accompanied by automated tests, which ensure that the changes wouldn't break any relevant components. *Artemis* uses a CIS in conjunction with participations in programming exercises. For every submission to a participation, the CIS runs tests on the submitted code and reports the test results to *Artemis* (see figure 2.3).

[6] `https://git-scm.com/`

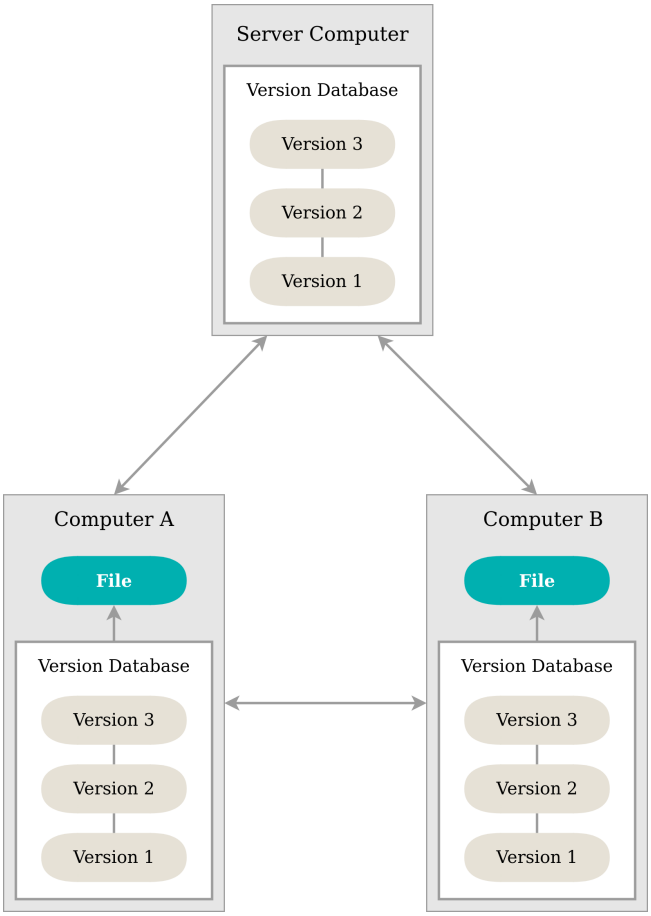


Figure 2.2: Distributed version control [CS14]. Every computer holds a copy of the version database. Repositories can be synced between computers.

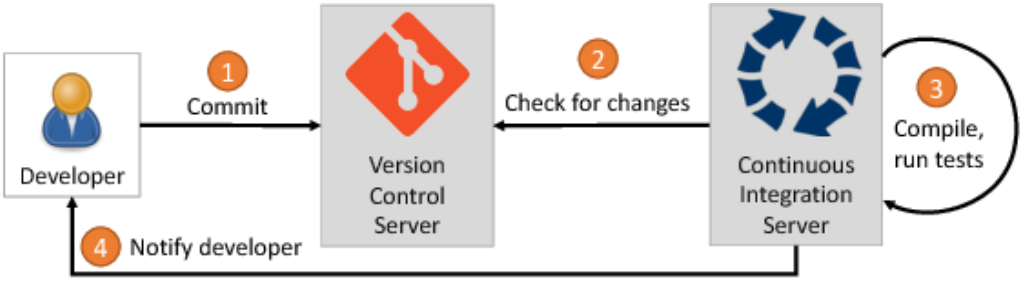


Figure 2.3: Typical CI + VCS setup in Artemis [KS18]. Users commit their changes with a new submission (1), which the observing CI server fetches (2) and builds (3). The test results get then forwarded to the user, so he can adapt his code, or finish his participation (4).

Chapter 3

Related Work

There already exists a multitude of mostly web-based code editors and programming exercise feedback and teaching tools. With regards to *Artemis*, the web client offers its own online code editor including interactive test result feedback and syntax highlighting [MK17, Beh19]. On the other side, plugins for IDEs that bridge the gap between an e-learning platform and a sophisticated development environment are just starting to emerge. In this chapter, we analyze two IDE plugins, that aim to integrate the submission and feedback processes of MOOCs into IDEs such as IntelliJ and NetBeans[1]. We further compare *Orion* to an online code editor, draw parallels to *Artemis* and elaborate on the fundamental differences to *Orion*.

3.1 Test My Code

Test My Code (TMC) is a suite of tools developed at the University of Helsinki [VVLP13]. The plugin[2] integrates into the ecosystem of the bigger TMC platform, which includes a server application[3] and user-mode linux images for building and testing the submissions[4] (see figure 3.2). Similar to Artemis, the platform allows instructors to create and update programming exercises, while providing students with immediate feedback for every submission. Additionally, students can request code reviews, which will get answered by instructors (see figure 3.1) and displayed in the IDE, if the plugin is installed. The plugin is available for NetBeans and IntelliJ on an open-source basis. One distinctive feature is the ability of TMC to gather

[1] `https://netbeans.org/`
[2] `https://github.com/testmycode/tmc-netbeans`
[3] `https://github.com/testmycode/tmc-server`
[4] `https://github.com/testmycode/tmc-sandbox`

user data, e.g. by tracking keystrokes of students[5].

In comparison to TMC, *Artemis* splits every programming exercise up into three repositories (template, test, solution), while TMC follows the approach of having one codebase including the solution to the exercise: Code, that the student should not receive is marked with specific comments and gets removed by the TMC server before sending it to the student's machine [PLVV13]. Including all tests in the student's repository also introduce the disadvantage that a participant can reverse engineer the solution by analysing the tests and matching the expected results. *Orion* explicitly runs all tests for students on the remote CIS and only displays the results without revealing information about the implementation of the actual test. Furthermore, *Orion* also offers a simplified UI for automated download, submit and build workflows, which TMC does not support without performing manual steps. Finally, *Orion* integrates the *Artemis* client into the IDE in contrast to the separate web application of TMC.

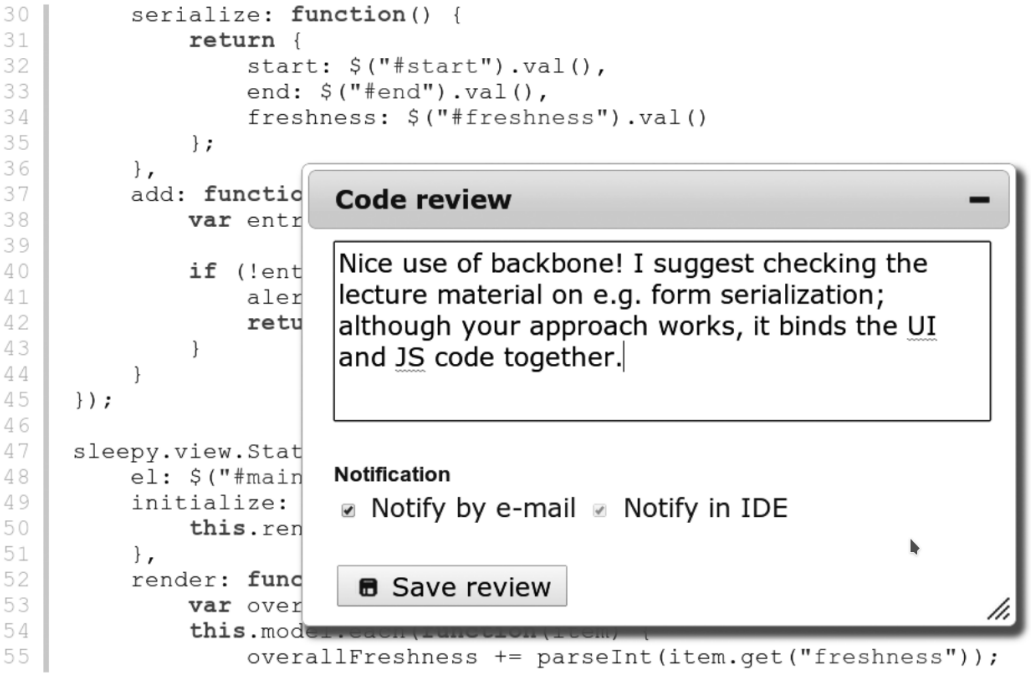


Figure 3.1: Code review of a submission in TMC [VVLP13]

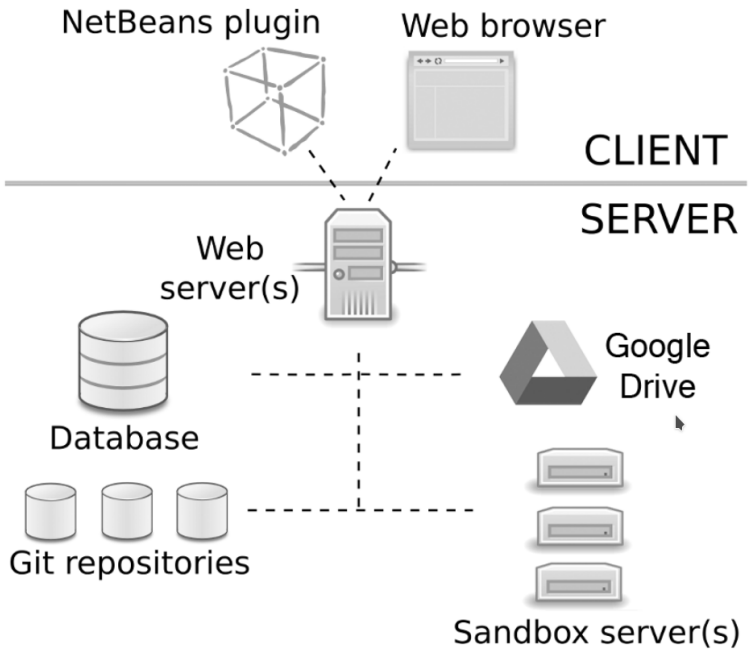


Figure 3.2: Architecture of TMC [VVLP13]. The separation of the plugin and the web application result in a media disruption. Web servers represent the central communication nodes for all system components.

3.2 JetBrains Edu Tools

JetBrains offers the inhouse developed Edu Tools plugin[7,8] for creating complete courses with programming exercise and sharing them privately or publicly with other students or instructors. The plugin is only available for IntelliJ. Compared to TMC, or *Artemis*, the focus of managing and sharing Edu Tools courses lies within the IDE itself rather than having a secondary management platform, such as a dedicated web application. However, it is possible to upload and integrate courses into online learning systems such as Stepik[9], which is natively supported by the plugin. A similarity to *Orion* is the additional tool window (see figure 3.3), which displays styled instructions for the opened task. The Edu Tools also display test results immediately since the tests themselves run locally in the environment of the IDE. Solutions can also be made visible to the student depending on the configuration.

[5] `https://testmycode.github.io/`
[7] `https://github.com/JetBrains/educational-plugin`
[8] `https://plugins.jetbrains.com/plugin/10081-edutools/`
[9] `https://stepik.org/catalog?language=en`

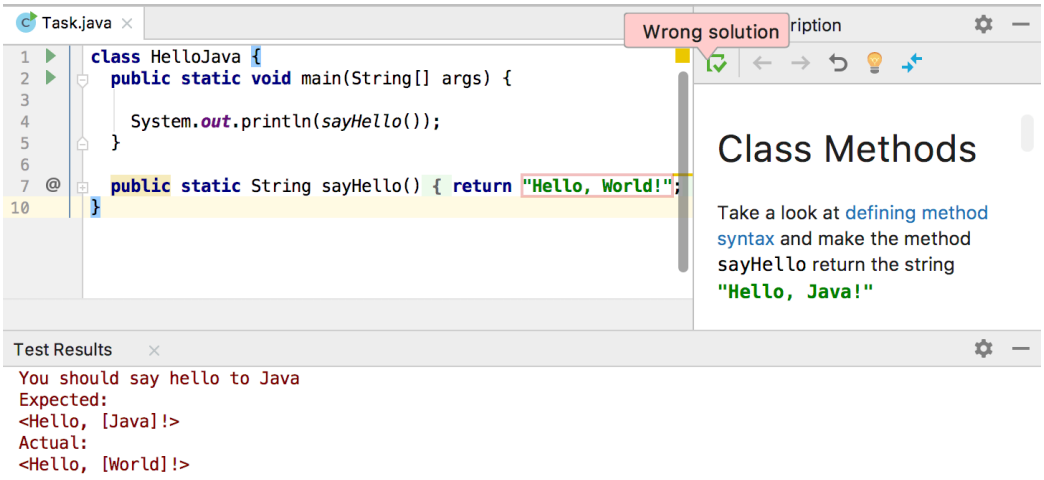


Figure 3.3: Feedback for a failed submission using the Edu Tools plugin[6]. Tests are run locally and results are immediately availabe to the student. The plugin also offers styled instructions for every task, which get displayed inside the IDE in a separate tool window.

In contrast to *Orion* we can summarize the following differences: Test implementations are available to the student and interactions with connected e-learning platforms from within the IDE are limited compared to using a web client. Just as it is the case with TMC, multiple repositories, or a VCS in general are not supported, which *Artemis* uses to offer more modular and customizable exercises. Both plugins diverge when it comes to evaluating the performance of a student: Contrary to the Edu Tools plugin, *Orion* is aimed at courses which rely on students not knowing about test and solution implementations for their grading process.

3.3 Coding Tools of the openHPI Platform

Web-based code editors provide basic code editing functionalities in browsers. openHPI's *CodeOcean* platform is a standalone application, which can be connected to external e-learning systems [SKT+16]. It is part of the on programming exercises focused interactive coding tools developed at the Hasso-Plattner-Institut. In addition to the online editor, the stack includes the video conferencing program *CodePilot* and the exercise sharing portal *CodeHarbor* [STM17b], which instructors can use to both share and import created tasks [STM17a]. Because programming exercises take a long time to create, the goal of *CodeHarbor* is to distribute the workload so that useful

exercises can be reused in different courses. *Artemis* also allows imports, although this is limited to users who are also registered as instructors in the original course.

Furthermore, users of the openHPI stack can request direct help from tutors via the videoconferencing functionality of *CodePilot* [TWS17]. Alternatively, tasks can also be solved together with other students using pair programming as users can see and talk to each other while using the online editor.

CodeOcean can be combined with the aforementioned tools and connected to MOOCs and other e-learning platforms. The *CodeOcean* editor can communicate with external systems via the learning tools interoperability interface (LTI)[10], the same interface *Artemis* also uses for its external connections. *CodeOcean* combines a Ruby on Rails based web application with the OS-level virtualization technology Docker (see figure 3.5). In contrast to Artemis, the hereby used workflow does not involve a combination of CI and VCS, but rather code execution and testing using Docker containers[11]. Users can solve simple tasks using such an editor, but are limited as soon as more complex implementations are queried. The involvement of an IDE as enabled by *Orion* is not possible at all since there is no VCS, which would allow students to download a repository. Consequently, debugging and code completion are also not supported. Figure 3.4 shows the *CodeOcean* editor in a browser. The edited source code can be either run or tested, which provides the user with the output of his submission, or a score based on the results.

[10] `http://www.imsglobal.org/activity/learning-tools-interoperability`
[11] `https://www.docker.com/`

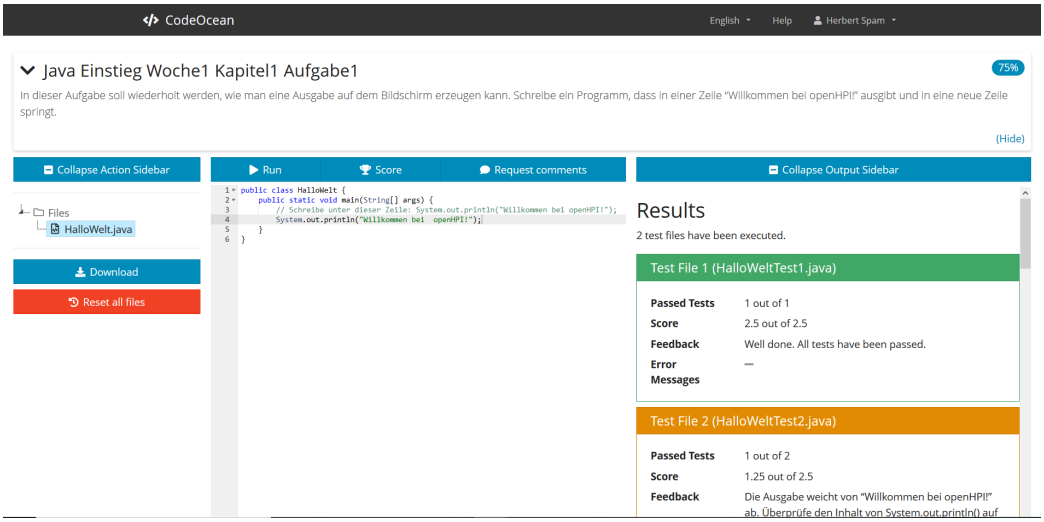


Figure 3.4: CodeOcean online editor[12]. The user partially solved the task and received a grade of 75%. Changes to the code can be either run, which just displays the output, or tested resulting in a score.

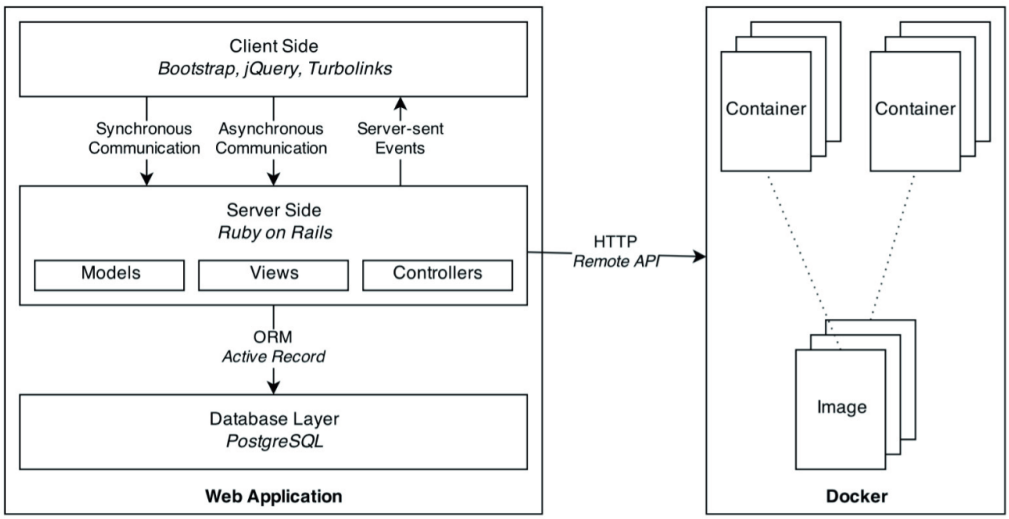


Figure 3.5: High level architecture of CodeOcean [SKT+16]. A Ruby on Rails web application is combined with Docker to build an interactive online code editor. Code runs in containers and results get displayed in the web client.

Chapter 4

Requirements Analysis

This chapter focuses on the requirements and scenarios that form the basis of this thesis. The following sections adhere to the standards specified by Brügge and Dutoi [BD09], specifically the *Requirements Analysis Document Template*. The first section gives an overview over the current and proposed system, followed by a list of functional and nonfunctional requirements. Section 4.3 finally visualizes all changes and new components on an application domain level.

4.1 Current System

In order to facilitate programming exercises, Artemis has multiple external dependencies, that provide the required functionalities for storing, building and testing source code. The current system in figure 4.1 illustrates these components, which consist of the *Version Control* and the *Continuous Integration* systems. Both are connected to the *Artemis Server*, which implements an interface to the *Artemis Client*. The client is needed to start an exercise and analyze any feedback for a submission. Code submissions in the form of a push to the VCS trigger a build on the CIS. The CIS reports all results to the *Artemis Server*, which links them to the submission and forwards the feedback to the client.

As a consequence, this setup induces media disruptions, because users have to switch between the IDE, Artemis and often an additional VCS GUI: The association between the IDE and the VCS cannot be seen as given since it depends on the concrete IDE implementation and is only optional. There is no way to circumvent switches between different clients in the current system, if the user prefers an IDE over the in the *Artemis Client* integrated code editor, because he wants to work with a more sophisticated toolset.

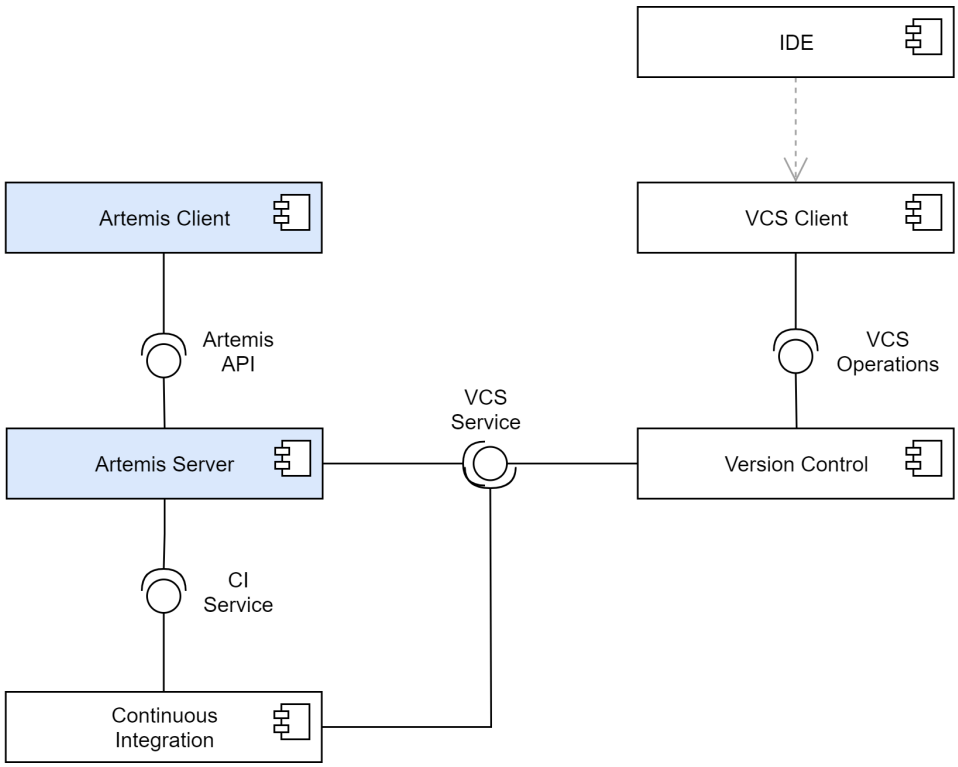


Figure 4.1: Current programming exercise system of *Artemis*, depicted as a high-level component diagram, adapted from [Beh19]. *Artemis* VCS client components are not directly connected and have to communicate via the *Artemis* server. The by *Artemis* provided components are colored in blue.

4.2 Proposed System

The proposed solution connects the *Artemis Client* with the development environment using a plugin, *Orion*, which can be installed in the *IDE*. *Orion* provides new interfaces, that form a bidirectional relation between *Artemis* and the *IDE*. External interactions with programming exercises can then be performed without any media disruptions.

From a high-level perspective, there exist two in *Orion* integrated connectors, which link all components (see figure 4.2): The *Artemis Connector* handles interactions with the existing client, while the *VCS Connector* contains an adapter to all necessary interfaces of a version control system. An alternative to the *Artemis Connector* would be to create a dedicated *Artemis IDE Client*. This would result in duplicated implementations of the same functionalities, because we want to offer the same feature range as the

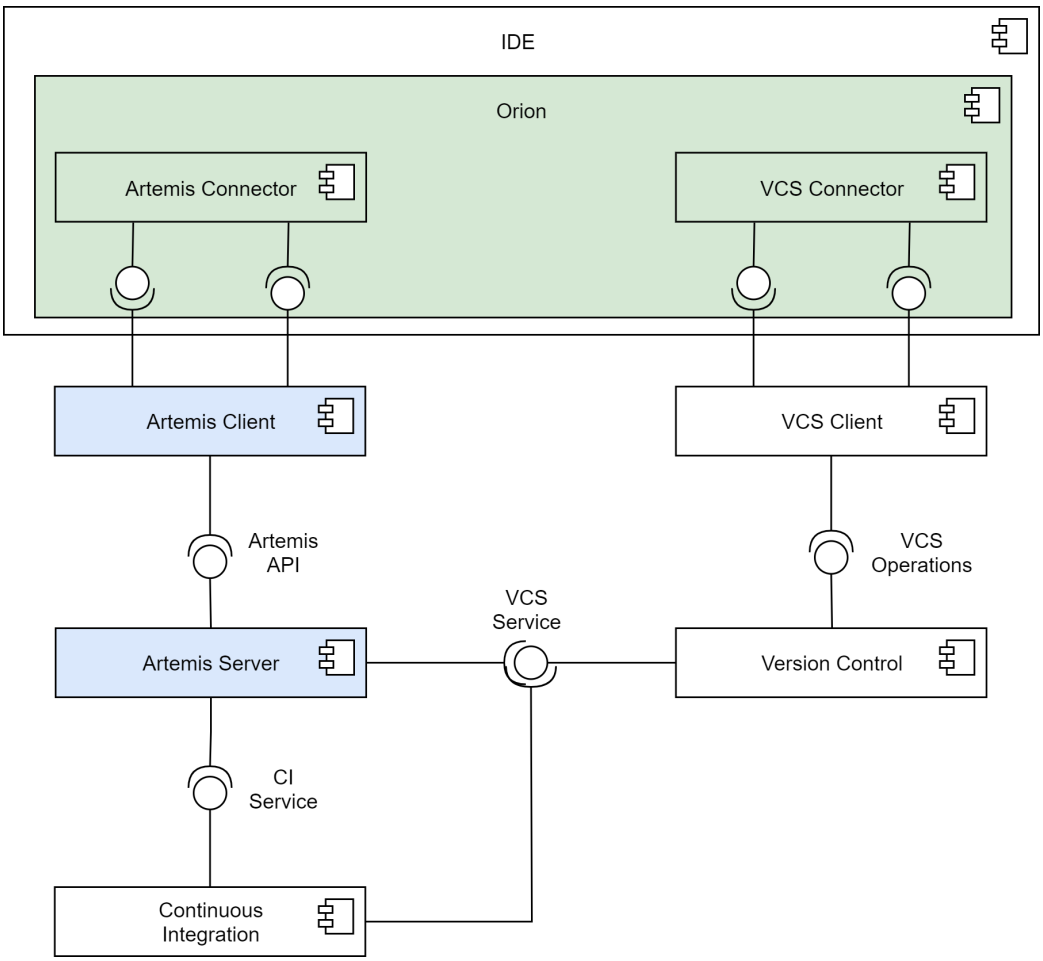


Figure 4.2: Component diagram of the proposed system of *Artemis* including the new *Orion* IDE plugin. Media disruptions are eliminated by connecting the *Artemis* client with the IDE and VCS. The by *Artemis* provided components are colored in blue, new components are colored in green.

already existing client does. Deciding to use connectors has the additional advantage of a more lightweight implementation, which is easier to maintain and less likely to introduce errors that have already been discovered and solved in the regular client.

4.2.1 Functional Requirements

The following sections describes the functional requirements of *Orion*. We list what concrete expectations a user might have with regards to his interactions

with the system. In order to provide a more comprehensive list, we group the requirements into three sections related to

- The *VCS Functionality*
- The *Build and Test Functionality*
- The *Artemis IDE Project Functionality*

VCS Functionality

FR1.1 **Download participation**: Students should be able to download the repository that is related to their participation.

FR1.2 **Download base repositories**: Instructors should be able to download the test, template and solution repositories of an exercise.

FR1.3 **Download student's submissions**: Teaching assistants should be able to download the repository related to a student's submission.

FR1.4 **Edit exercise in one project**: Instructors can modify the test, template and solution repositories of an exercise in one single project.

FR1.5 **Resolve conflicts**: If there are any merge conflicts while updating a repository, users should have to opportunity to let *Orion* resolve them automatically.

FR1.6 **Submit changes**: Users should be able to save all current changes to a repository and upload them to the remote *Artemis* VCS.

Build and Test Functionality

FR2.1 **Test base participations locally**: Instructors should be allowed to execute tests for the solution and template participation locally.

FR2.2 **Build participations remotely**: Users should be able to trigger the build for any imported participation from within the IDE.

FR2.3 **Analyze test results**: For any task of an exercise, the last reported result should be displayed next to it, so that users can analyze potential errors of their submission.

FR2.4 **Display build results faster**: Instructors should be able to see preliminary test results while waiting for the remote build to finish execution.

Artemis IDE Project Functionality

FR3.1 **Generate Artemis IDE project for exercise**: Instructors should be able to generate an *Artemis* IDE project for any of their exercises.

FR3.2 **Generate Artemis IDE project for participations**: Students should be able to generate an *Artemis* IDE project for any of their exercise participations.

FR3.3 **Customize storage location**: Users should be able to freely choose the storage location for generated *Artemis* IDE projects

FR3.4 **Move Artemis projects**: Users should be able to move an already downloaded *Artemis* IDE project to a different location.

4.2.2 Nonfunctional Requirements

The next list follows the FURPS+ model [BD09] and provides an overview over all nonfunctional requirements. We omit the *functional* category as it has already been covered in the previous section.

NFRU.1 **Usability**: The amount of clicks, that are necessary to download a repository should be lowered from the current maximum of 5 to 2.

NFRU.2 **Usability**: The amount of clicks, that are necessary to submit changes to the remote repository should be lowered from the current maximum of 7 to 2.

NFRU.3 **Usability**: The by *Orion* added UI elements should be consistent with the existing style and user experience of the IDE.

NFRU.4 **Usability**: The system should offer all already existing interactions related to programming exercises in the IDE.

NFRU.5 **Usability**: Users should not need more than 30 seconds to find the UI elements used for starting the import and submit processes.

NFRP.1 **Performance**: *Orion* should not block interactions with the IDE, that are not directly related to programming exercises.

NFRP.2 **Performance**: If an action, that is already available in the current *Artemis* client, is performed from within the IDE using *Orion*, the total number of requests sent to the *Artemis* server should not be higher compared to the current *Artemis* client approach.

NFRP.3 **Performance**: The delay between incoming test results and their display in the IDE should be lower than $\frac{1}{2}$ second.

NFRS.1 **Supportability**: *Orion* should be written in a Java interoperable programming language as to not introduce new language fragmentations for future maintenance tasks regarding the *Artemis* platform.

NFRS.2 **Supportability**: A developer, who added a new feature to the *Artemis* client (except the online code editor), should be able to integrate the same feature into *Orion* within one business day.

NFRS.3 **Supportability**: The plugin should support English and German localization.

NFRS.4 **Supportability**: *Orion* should support Windows 10, macOS 10.15 Catalina and any Linux distribution with the KDE, Gnome or Unity desktop environment.

4.3 System Models

The next sections visualize different perspectives on the system. Use case diagrams give an insight into the the relationship between actors and the implemented system, followed by an abstracted view in the form of an analysis object model.

4.3.1 Scenarios

The next two subsections describe exemplary scenarios related to the previously defined requirements. They offer additional insight as to how *Orion* is used for the programming exercise creation and participation. The first scenario focuses on the role of a student, who imports a new exercise into the IDE. The second outlines the administrative view of an instructor, which is centered around editing an exercise by updating and testing the source code for all three base repositories.

Scenario 1: Working on an Exercise and Solving it

This scenario describes how *Orion* can be used for importing the repository of a participation into the IDE, solving all tasks and submitting the solution. The actor is Denis, a student participating in a programming course at the university. The entry condition requires that there already exists a released programming exercise, which is accessible to all students in the related course.

Denis takes part in this course, therefore he is eligible to participate in the exercise.

Denis opens his IDE and the by *Orion* integrated window containing an all available interactions with *Artemis*. As the currently opened project is a previously imported exercise from *Artemis*, *Orion* displays the details of the related exercise showing Denis his score, the exercise instructions and the test results for his last submission. Because he wants to start the newly released exercise for the course, he navigates to the course overview and finally to the latest programming exercise. Denis starts his participation with the click on a button, which then causes a loading animation signaling him that his participation is being prepared. Afterwards, a new interaction becomes visible allowing Denis to import his personal repository into an *Artemis* project inside the IDE. After using the import option, *Orion* signals to Denis that all relevant files are being downloaded. *Orion generates* an IDE project at the end of this process, which Denis opens to interact with the exercise.

Denis can now read through the instructions and starts implementing his solution. He has to write a program based on two sorting algorithms, Bubble and Merge Sort. There is one task Denis has to solve, which in turn contains multiple subtasks:

1. **Task**: Implement both sorting algorithms
    (a) **Subtask**: Implement the Merge Sort algorithm
    (b) **Subtask**: Implement the Bubble Sort algorithm

Denis implements both sorting algorithms and debugs his program locally using the in the IDE integrated tools. He then wants to verify and save his progress, so he uses the *submit* functionality offered by Orion. He gets a confirmation that all his changes have been saved and uploaded to the server, followed by an automatically opened window informing him that his code is currently being tested. After the test run, the results show that there are still errors in the Bubble Sort code, so Denis *analyzes* the provided feedback and tries to fix the problems. After *submitting* a second time, the results show no error, so the exercise has been successfully completed.

Scenario 2: Editing an exercise

This scenario describes how an instructor can use *Orion* to edit all repositories of an exercise simultaneously using one opened instance of his IDE. The actor is Jane, an instructor for the course "Basics of Programming". For a

valid entry condition, there has to already be a created exercise, so that any instructor can edit the template, solution and test repositories.

Jane opens her IDE and the *Artemis Project* tool window inside it, in which *Orion* displays an overview over all courses. Jane navigates to the course administration and selects a newly created programming exercise, which needs to be prepared for her students. She clicks on the *edit* button, which prompts her to select a storage location for the new *Artemis Project*. After confirming the import, a pop-up informs her that all three base repositories are being downloaded. After the creation of the new project is done, Jane opens the imported exercise and has the option of editing files in one of three submodules, one for each base repository: The test, the solution, or the template repository. She selects the *test* repository and implements all tests for the tasks in the exercise instructions. After a click on the *submit* button, *Orion* informs her, that all changes have been successfully uploaded to the server. Jane shifts her focus to the *solution* submodule and implements the reference solution for all future submissions. She verifies her implementation by clicking on the *local test* button, which builds and runs the code on her local machine. The result window informs Jane, that there were no errors during execution, so she decides to *submit* the code to the remote repository. After an upload confirmation message, the test result window displays an animation, showing that the submitted code is being built on the server. Preliminary results allow Jane to already check whether her submission contains any serious errors. After waiting for a couple of seconds, the visualized remote build failed, which apparently was caused by a wrong build file configuration. After she fixes the error, both builds (local and remote) complete without any problems and the exercise is set up for the release date.

4.3.2 Use Case Model

After establishing the requirements, we will now detail use case models of *Orion*. The use cases can be split up into two models related to importing and working on an exercise. The actors in these cases include:

Student Participates in courses and exercises. In the displayed diagrams, students participate in programming exercises.

Instructors Creates and edits exercises. Has full access to all materials and exercises related to the course in which this role was assigned.

Start or resume programming exercise Repositories of programming exercise participations can be imported into the IDE creating a new *Artemis*

IDE project. Figure 4.3 illustrates how a student interacts with *Orion* by either starting or resuming a previously imported exercise. If a new exercise is started, the participation can be imported into the IDE. This entails the generation of a new *Artemis IDE Project*, which holds the downloaded repository. Because *Orion* simplifies all VCS specific operations, the student only has to *open* the project and can commence his work.

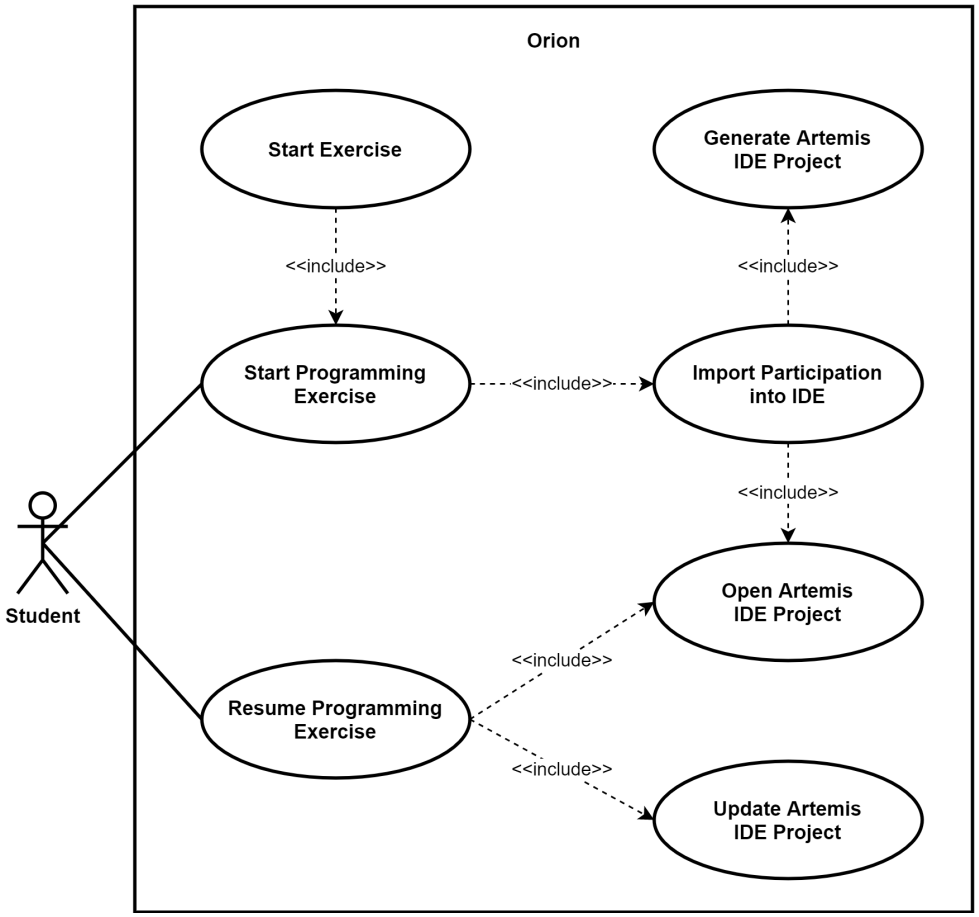


Figure 4.3: Use case diagram of starting/resuming a programming exercise participation in *Orion*. The student imports the participation by generating an IDE project and opening it.

Analogously in order to *resume* an exercise, an *update* is necessary which synchronizes the IDE project with the remote repository. Again, *opening* an existing project suffices in order to be connected to *Artemis* and modify the imported source code. On a second note, instructors can also import exercises, albeit this involves the download of multiple repositories. Because

the scenario in this case is almost identical to the use cases depicted in figure 4.3, we omit the model for this variant.

Submit changes to a programming exercise Both an instructor, or a student can *submit* their local changes to the student participation, or the whole exercise respectively. A student is limited to his own participation repository, while an instructor is allowed to work on the before mentioned base repositories of an exercise (see figure 4.4). We therefore introduce a common *submit* use case, which can be specialized in the form of a *participation submit* and an *exercise submit*.

The general submission process was already available in the online code editor. The plugin now ports this functionality into the IDE and handles any complex VCS operation related to saving the changes or uploading them to the remote repository. The user has to *update the repository* and *save his changes*, so that *Orion* can perform the submit using the most recent version of the user's source code.

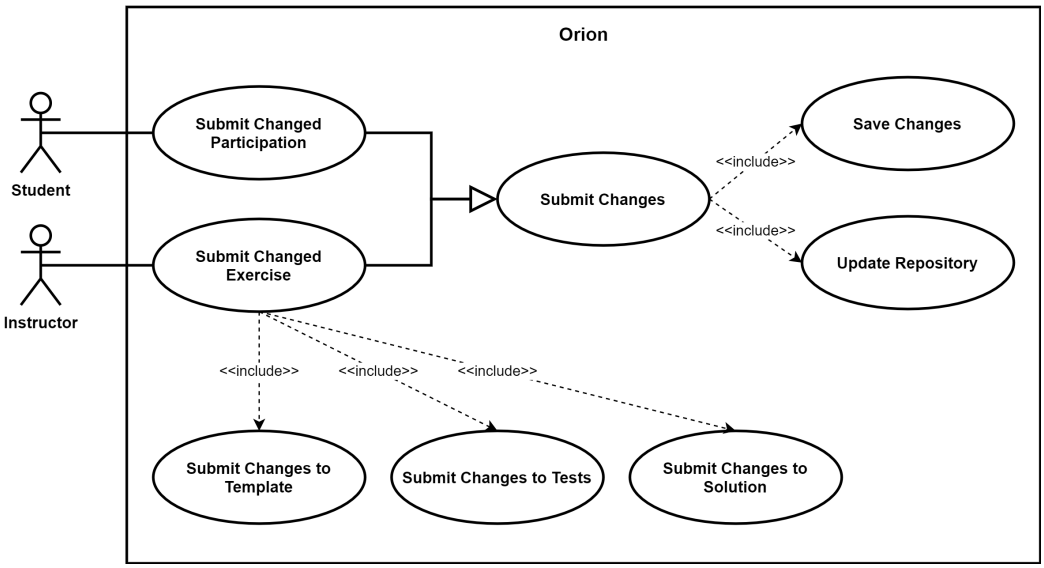


Figure 4.4: Use case diagram of the submission of changes to an exercise or participation. Instructors can edit an exercise by submitting changes to all three base repositories. Students only need to submit one.

4.3.3 Analysis Object Model

The following section explains how the updated use cases and analyzed requirements lead to the next iteration of the analysis object model. Additionally, we introduce new classes based on the architecture of *Orion* and show how a connection to the existing *Artemis* components is realised. Figure 4.5 provides a visual representation of the described model.

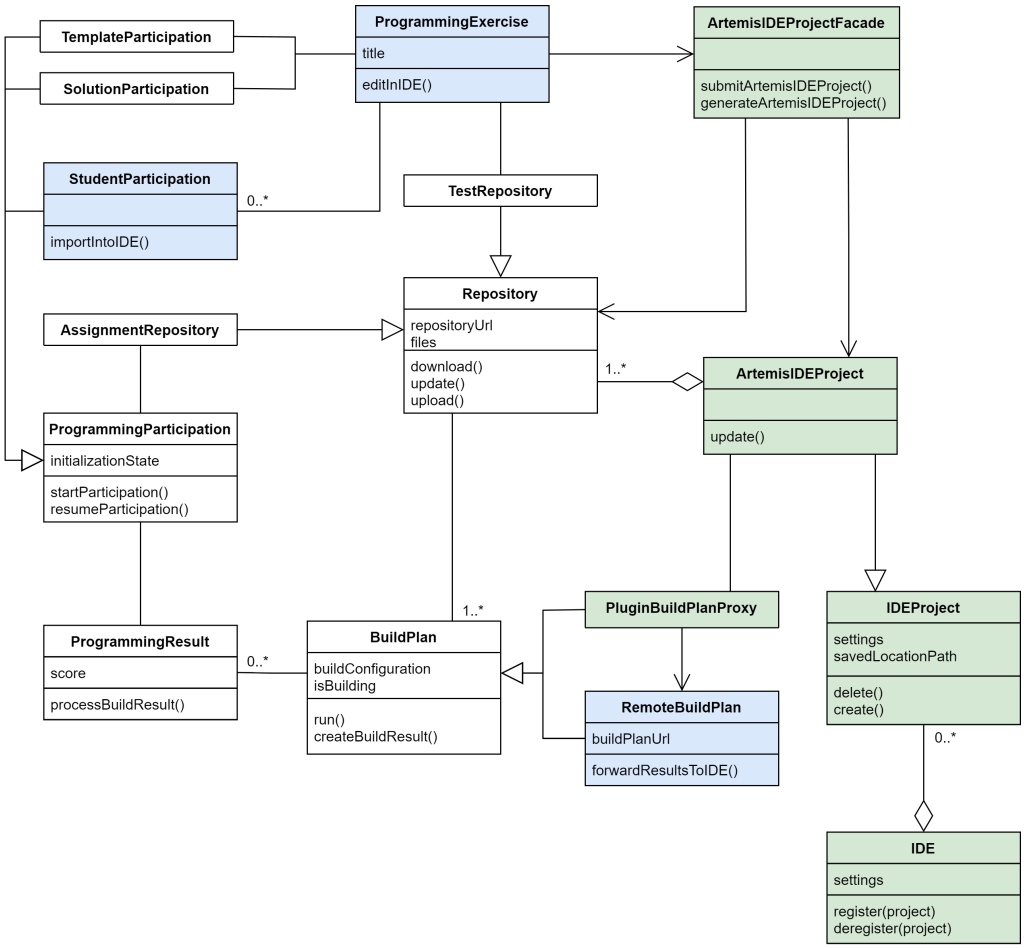


Figure 4.5: Class diagram of the analysis object model of *Orion*, adapted from [Beh19]. The new facade connects the IDE with the existing programming exercise components of *Artemis* and simplifies complex operations. Updated classes are colored in blue, new classes in green.

IDE The `IDE` represents the core component in which the developed plugin is embedded. An `IDE` can define different settings, which influence all containing entities. IDEs can register projects, so that the user may open and

work on them. While this is fully optional, an `IDEProject` does not necessarily have to be associated to a specific IDE instance, they are independent. Projects can be deregistered and moved to different environments, in which an `IDE` can register them again.

IDEProject As just described, projects can be seen as self containing units, identitified amongst other attributes by the saved *location*. Users can work on the source code of an exercise, or participation via `ArtemisIDE Projects`, as they wrap `Repositories` with the purpose of providing a single point of interaction. Because *editing* an exercise involves working with three base `Participations`, we allow projects to reference more than one `Repository`. There is no explicit upper limit, so that future updates of programming exercises which might require additional `Repositories`, are taken into consideration.

ArtemisIDEProjectFacade VCS operations mostly relate to the download and submission of changes to a `Repositoriy`. As we want to simplify these operations and allow them to be performed from within the `IDE`, i.e. via an `ArtemisIDEProject`, there has to be a connection between existing exercise components and the new *Artemis* project type. A facacde [GHJV95] fulfills both requirements. It serves as a link between the `ProgrammingExercise` and the `IDE`. Moreover, it provides an interface, which allows users to *generate* new projects and *submit* the locally downloaded `Repositories` while decreasing the number of necessary steps by bundling complex details under one method.

Build Plans For the `ArtemisPlugin` to be able to perform builds and display test results, the introduction of a `PluginBuildPlanProxy` is necessary. This proxy [GHJV95] offers the same interface as a regular `BuildPlan`, but produces a stand-in result while the underlying remote plan finishes execution. A simultaneously running local build provides the stand-in, so that *Orion* can display feedback quicker than the current implementation. Meanwhile, an update with the actual remote result as soon as it arrives ensures that this feedback is consistent with the one shown in the regular *Artemis Client*. Students are currently not able to perform builds locally since a `StudentParticipation` does not have access to the test repository. Hence, only instructors could benefit from the improved performance for now. Until a future update introduces hidden local tests to students, the proxy can just be used as a passthrough in these cases and switch to incorporating a local student's build as soon as the feature is released.

ProgrammingExercise and StudentParticipations For users to be able to download the in an `ArtemisIDEProject` contained repositories, the

`ProgrammingExercise` and `StudentParticipation` classes have to provide the necessary import operations. Based on the different use cases of instructors and students, there exist two different options:

1. Because an exercise including all base repositories can get edited by an instructor, the `ProgrammingExercise` itself has to offer an *editInIDE* functionality.
2. In the latter case of just solving the exercise while working on a `StudentParticipation`, there only needs to be a reference to the `AssignmentRepository` and the related `BuildPlan`. Consequently, using the *importIntoIDE* operation on the participation suffices.

4.3.4 Dynamic Model

We conclude this chapter with a visualization of the process of editing an exercise as an *instructor*. While *students* only have to work with one repository at a time, the general workflow stays the same when working on a participation with the exception of the local build process. Incidentally, the presented diagram focuses on one single repository out of all three base repositories for the same reason. The related model is depicted in figure 4.6

Edit and Submit Changes to an Exercise

The *edit* of an exercise in an IDE initiates the *download* of all relevant repositories related to the exercise template, solution and tests. In the meantime, the newly *generated Artemis IDE Project* contains additional information about the settings of the project itself (e.g. related to the programming language) and the imported exercise. This project can be opened in order to *edit the containing repositories*. The instructor can verify that his changes did not break the actual exercise and produce the expected result by *running a local build* as a first step. This is faster than a direct run of the remote plan since the submission and build result forwarding steps can be omitted. If this local result already produces errors, adaptions have to be made and the instructor *edits the repository again*.

If there is no reason to stay in this refinement process based on the local build results, then all changes can be *submitted* and persisted in the remote repository. Such a submit results in two parallel activities:

- While the remote build is running, *Orion* uses the *Build Plan Proxy* in order to display an intermediary local result until it receives the actual test results from the remote plan. These results can contain

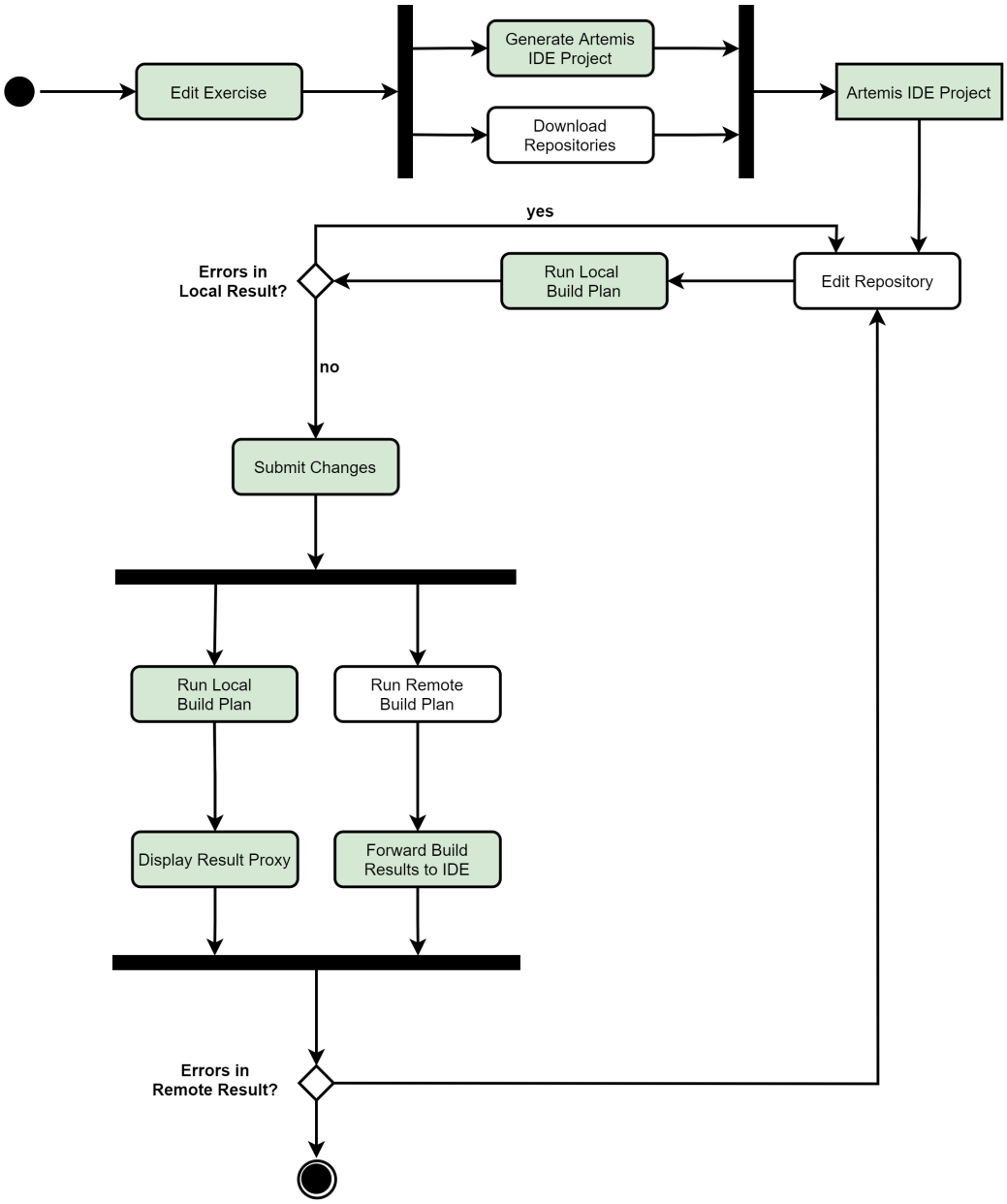


Figure 4.6: Activity diagram of an instructor editing a programming exercise. Over two edit and test loops, the user modifies the exercise repositories until the local and remote builds produce positive results. New activities are colored in green.

new errors, which necessitate another edit of the repository. This can happen, because the plan on the CIS does not have to be identical to the one used by the local plugin. Both can run the submitted code, but the environments in which these builds are executed might be different and have an influence on the final result.

- The run of the remote build is an already existing process in the current system. It produces all test results, that are relevant for grading the participants. However, in the case of interacting with a programming exercise using *Orion*, these results are *forwarded to the IDE*, so that the user can immediately react to any negative feedback.

The final test analysis can be performed after all results got reported to *Orion*. Based on these tests, the instructor can decide whether to conclude the process, or edit the repository again.

Chapter 5

System Design

Based on the *System Design Document Template* by Brügge and Dutoit [BD09], this chapter maps the findings of the analysis in chapter 4 to the solution domain of the implemented system. First, we will give an overview over the high-level architecture of the plugin and the considered design goals and then provide visualizations of the decomposition of all relevant subsystems.

5.1 Overview

Because the plugin provides an alternative, but not completely new way of interacting with programming exercises, it can make use of the existing systems displayed in the analysis object model from the previous chapter (figure 4.5). Moreover, *Artemis* developers should be able to perform basic maintenance tasks (as described in the nonfunctional requirements), so choosing to *connect* the plugin to the *Artemis Client* minimizes the needed time to familiarize with *Orion*. *Artemis* developers already have experiences with the client, which lowers the entry barrier when they have to work with the new system. Instead of re-implementing solutions for already solved problems, we thereby use the client to bootstrap the base features of *Orion*. These involve common workflows such as starting and resuming programming exercises, or reading through instructions.

All implementations of *Artemis* can be interfaced using the *Artemis Connector*, which in turn routes calls originating from the client to the appropriate subsystem. As this is a *bidirectional* relationship, the connector can report various internal system states to the client, so that it can react to any actions within the IDE, e.g. by displaying a status message to the user. Summarized, the connector provides a two-way adapter [GHJV95] to both

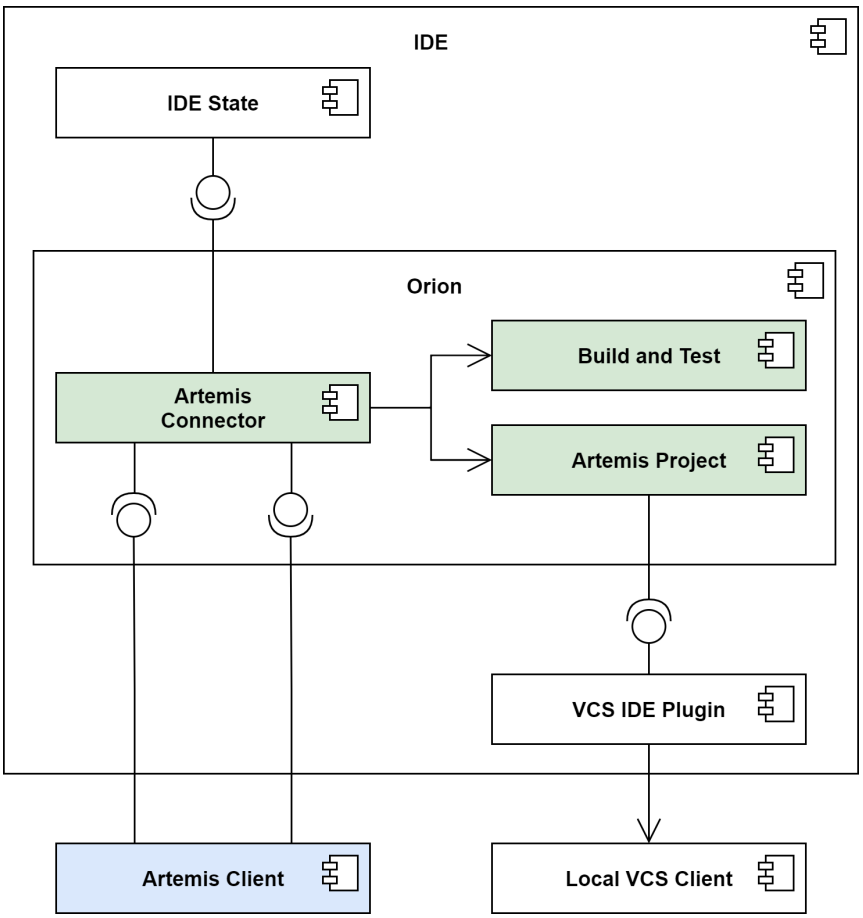


Figure 5.1: Component diagram of the *Orion* system design. Depicted are the most important components of the plugin and their connections to external modules. Updated components are colored in blue, new components in green.

the *Artemis* client and *Orion*. Additionally, it realizes a facade, that simplifies all interactions with the VCS and IDE projects and makes them more accessible to users of the platform.

As can be seen in figure 5.1, besides the connector, *Orion* is split up into two other subsystems:

Build The *Build* component handles incoming test results from the CIS and structures them according to the tasks in the instructions of the exercise. It displays these feedbacks to the user and enables him to *analyze* potential errors in the submission. Alternatively, if a user triggers a local build, this component creates all necessary configurations and executes the plan.

Artemis IDE Project The project component manages all imported exercises by tracking and updating the generated IDE projects, which in turn contain the downloaded repositories. The interface to the *VCS IDE Plugin* allows *Orion* to simplify complex version control systems such as Git for inexperienced users. Nevertheless, if required by a new feature, the full set of VCS operations is still available and only needs to be added in the internal adapter.

5.2 Design Goals

In the following, we prioritize the from the nonfunctional requirements derived design goals. They aid in the decision making process when implementing the system, as potential trade-offs can be weighed based on this prioritization. Therefore, the following list ranks them based on their importance from highest to lowest:

1. Usability *Orion* focuses on making it as easy as possible for new users to work with the *Artemis* platform. Additionally, the usability of the system compared to traditional approaches influences the decision of instructors whether to use *Artemis* in future courses, or recommend it to colleagues. Therefore, it is necessary to provide the users with an **intuitive** and **simple** UI (NFRU.3, NFRU.5), which introduces an improved workflow requiring fewer user interactions (NFRU.1, NFRU.2).

By the plugin hidden complex details of a VCS are especially important to students on a beginner level. Lowering the total amount of necessary interactions with a VCS makes the system more approachable and lowers the entry barrier (NFRU.1, NFRU.2). Users should be able to expect the same functionalities that the current system offers with regards to programming exercises, so the same interactions (start exercise, view instructions, etc.) as already available in the *Artemis Client* should also be available when working with *Orion* (NFRU.4).

2. Performance The existing workflow for programming exercises using the online code-editor can be seen as a benchmark that *Orion* should be able to match (NFRP.2). Because the plugin should be a preferred alternative to existing approaches, performance cannot be noticeably worse. In order to not negatively affect the user experience due to disproportionately high idle times caused by a bad performance, the plugin has to add a low overhead to the regular usage of the IDE. This involves the execution time of often performed actions such as downloading, or submitting the code for a participation (NFRP.2), or analyzing build results (NFRP.3).

Moreover, if there is a waiting period due to an unpreventable time consuming operation (e.g. a build), unrelated UI elements should not get blocked. Users should be allowed to still work on the exercise, while time intensive processes finish execution in the background (NFRP.1).

2. Supportability As *Artemis* is a project which steadily evolves, *Orion* has to be extendable if there is an update related to programming exercises. New features in this area might have an impact on the plugin and require adaptations to the implementation. Ideally, the developer, who is responsible for an update to *Artemis*, should also be able to implement any needed changes to *Orion* (NFRS.1, NFRS.2).

From a users perspective, just as it is the case with the *Artemis* client, a plugin has to support the most commonly spoken languages with regards to its user base. As of the writing of this thesis, the client is available in English and German localizations, so *Orion* should also offer translations in these cases (NFRS.3). Lastly, as IDEs can be installed on a variety of operating systems, *Orion* should match all by the IDE supported environments (NFRS.4).

5.3 Subsystem Decomposition

The in figure 5.1 depicted system can be further decomposed. Hence, this sections provides an analysis of the previously mentioned three subsystems of *Orion* and illustrates how the plugin interfaces off-the-shelf components [BD09]. We explain how existing systems and *IDE Plugins* are used to combine VCS, *IDE Project* and *Artemis* functionalities.

5.3.1 Connector Components

Because the before mentioned supportability goals require the reuse of parts of the existing *Artemis* platform in *Orion*, we have to provide a connection between those two components. This results in two connectors (see figure 5.2), which link the in the IDE installed plugin (and thereby the IDE itself) with *Artemis*. As already mentioned, these components are divided between a so called *Orion Connector* and a *Client Connector*, a result of two different possible directions of communication:

1. The client can call specific service interfaces on the *Orion Connector* related to the different interactions with programming exercises. We distinguish between the *Exercise Service*, which facilitates the import and management of exercises, the *VCS Service*, which exposes the

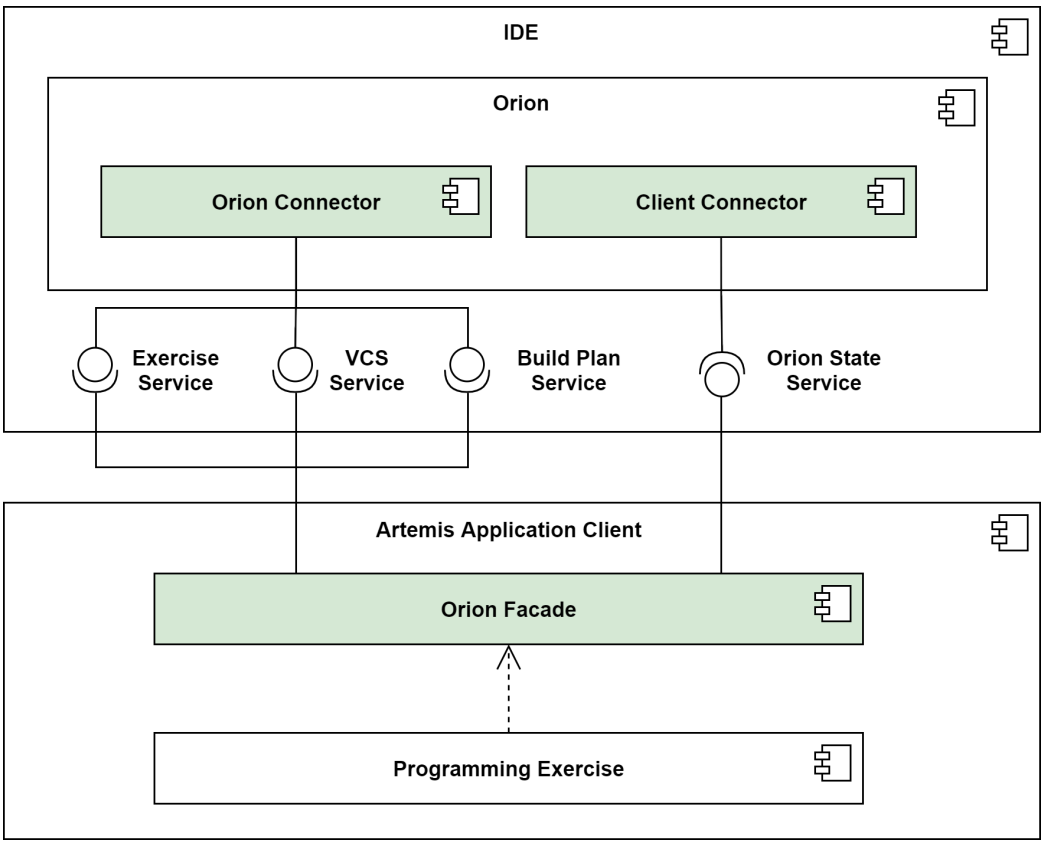


Figure 5.2: Component diagram of the connector subsystem between *Artemis* and *Orion*. The bidirectional relation allows for responses to invocations on both sides of the connection. Details of the connections between the systems are hidden behind a facade. New components are colored in green

update and submit functionalities of all downloaded repositories and finally the *Build Plan Service*, which wraps the running of builds and reporting of executed tests.

2. The *Client Connector* informs the client about any changes to the internal state of the plugin. The state is defined by the opened exercise and reflects whether there currently is an ongoing build, submit, or update. Any incoming state changes get then propagated via the *Orion State Service* to the client, where a visual representation of the ongoing process is displayed to the user.

In the client a new *Orion Facade* hides all details related to the connection to the plugin. With the goal of low coupling between the major systems,

the existing *Programming Exercise* component only needs to interact with this facade if it has to communicate with the IDE. Furthermore, all service interfaces between client and plugin get moved to the IDE in order to minimize additions to the client. *Orion* should be an itself closed system, which contains all necessary implementations for connecting to *Artemis*. The resulting high cohesion within the plugin allows the client to only require generic interfaces in the facade.

5.3.2 Build Components

Depending on whether the user wants to build the exercise locally, or run it on the CIS, we need to provide different run configurations in the IDE. A run configuration bundles all settings related to executing the implemented code and the rules for parsing the output of the execution. Figure 5.3 visualizes the two different types of configurations and shows how they are connected to the remaining system components. The first steps of editing an exercise as an instructor always include running the code locally. *Orion* uses a *Local Run Configuration*, which is already connected to the *Test Result Interpreter* of the IDE. The interpreter collects all results and displays them to the user.

Because the interpreter can only process messages, that adhere to an IDE standardized format, the plugin has to interject a *Test Result Translator* that can parse the from the *Remote Build* component received results. The in figure 4.5 introduced proxy is placed between the translator and the interpreter in order to provide intermediate local results, which can be produced by the associated local configuration.

5.3.3 Exercise Components

The central *Exercise* component (see figure 5.4) creates, tracks and updates all programming exercises. If the user decides to import a new exercise, the *Artemis IDE Project Creator* is responsible for generating all for the IDE relevant files that constitute an *Artemis* project. The non-existing connection to the *Artemis IDE Project* is intentional, because the creator should not concern itself with the underlying relations between project files and actual exercise. For that purpose, we introduce the *Exercise Registry*. The creator is therefore stateless and does not reference any created project.

The registry keeps track of all imported exercises, while also being able to register a project that has been moved or copied. This is based on the aggregation between projects and the IDE from the analysis object model, which states that projects can exist independently. Thus, the *Artemis IDE*

Project is lowly coupled to *Orion* and does not rely on references to any plugin internals, which would bind it to the a specific IDE instance.

Lastly, the repository update, submit and download functionalities are built upon an existing *VCS Plugin* for the used IDE. In order to properly connect such a plugin to the *Exercise* component of Orion, there is a *VCS Adapter*, which offers all for these processes necessary interfaces and translates and forwards all invocations to the underlying *VCS Plugin*.

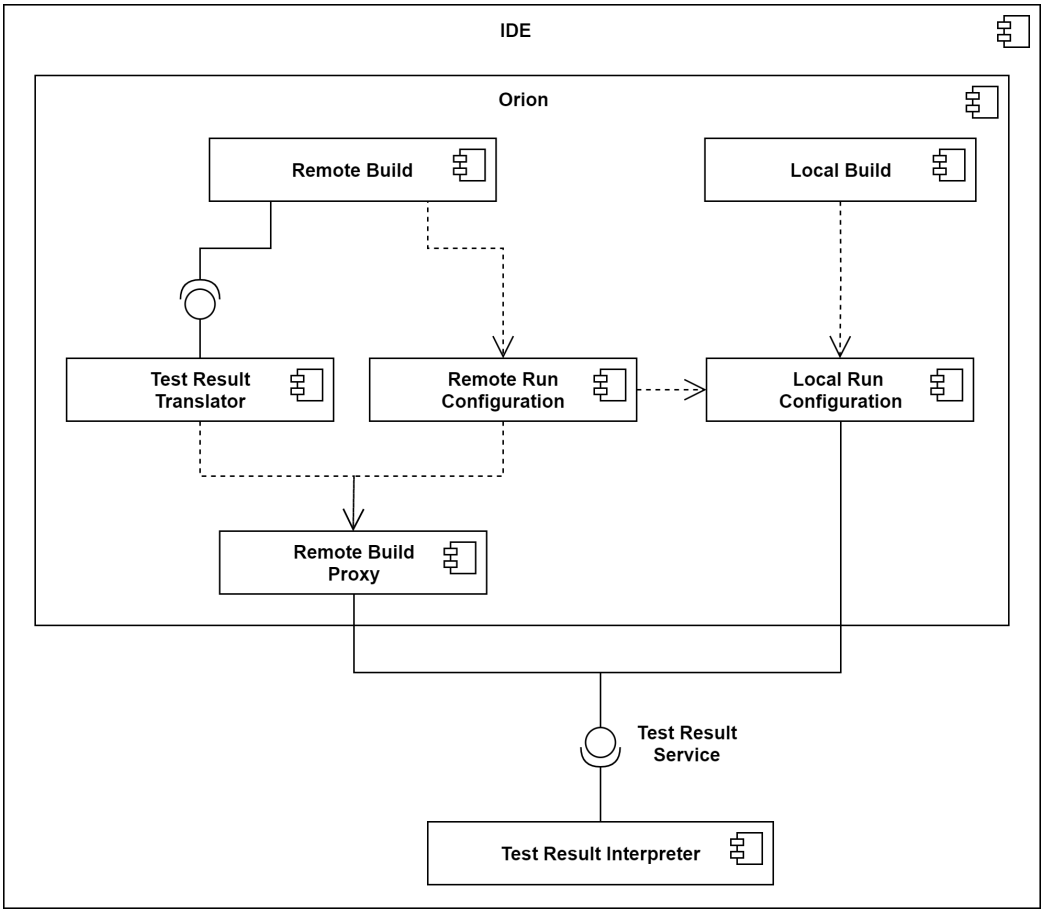


Figure 5.3: Component diagram of the build plan systems in *Orion* and the reporting of test results. *Orion* distinguishes between remote and local build plans. In the remote variant, the results pass through a translator, which parses the feedback into a standardized IDE format. The proxy displays an intermediate local result until the remote build finishes execution.

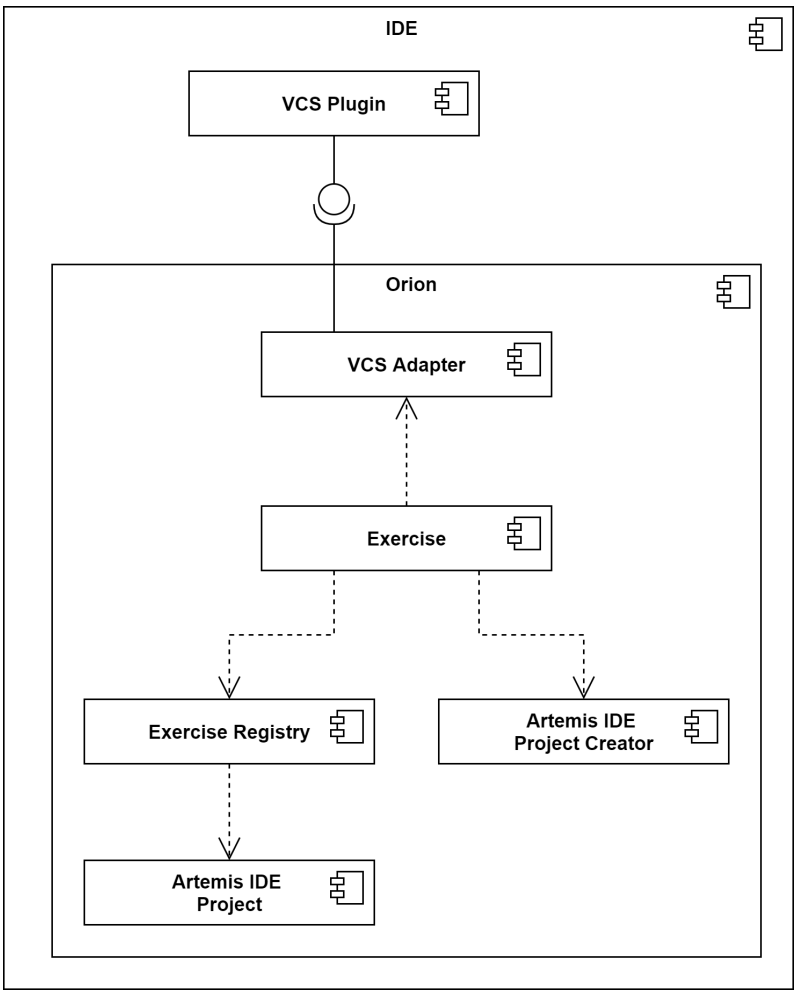


Figure 5.4: Component diagram of the exercise subsystem of *Orion*. VCS related functionalities are enabled by providing an adapter to the existing VCS plugin of the IDE. Imported projects get tracked by a dedicated registry.

5.4 Hardware Software Mapping

With the introduction of *Orion* as an alternative access point to the *Artemis Client*, the by Montag described hardware software mapping has to be updated [MK17]. Visualized in figure 5.5, the student's machine now doesn't contain three separate components (Artemis client, IDE and VCS client) anymore. Assuming *Orion* is installed, the IDE is now fully integrated into the *Artemis* deployment, thereby closing the gap between the Artemis and VCS clients. The current version of *Orion* has been implemented for IntelliJ,

but the general system architecture including the analysis object model and subsystem decomposition can be applied for all IDEs. *Orion* is designed with a potential port to different IDE instances in mind, so all models that we have shown so far are independent from any concrete instantiation.

Apart from the newly connected components, the base deployment of the system is identical with the old design: VCS and CIS interact with the *Artemis Application Server* by providing and building the source code related to programming exercises. With regards to the CIS, these builds can be performed directly on the CI server, or by a number of external build agents. Agents allow an improved scalability in the case of high loads. The IDE can fetch the source code directly using the installed VCS client and receive processed and formatted test results via the *Artemis Application Server*.

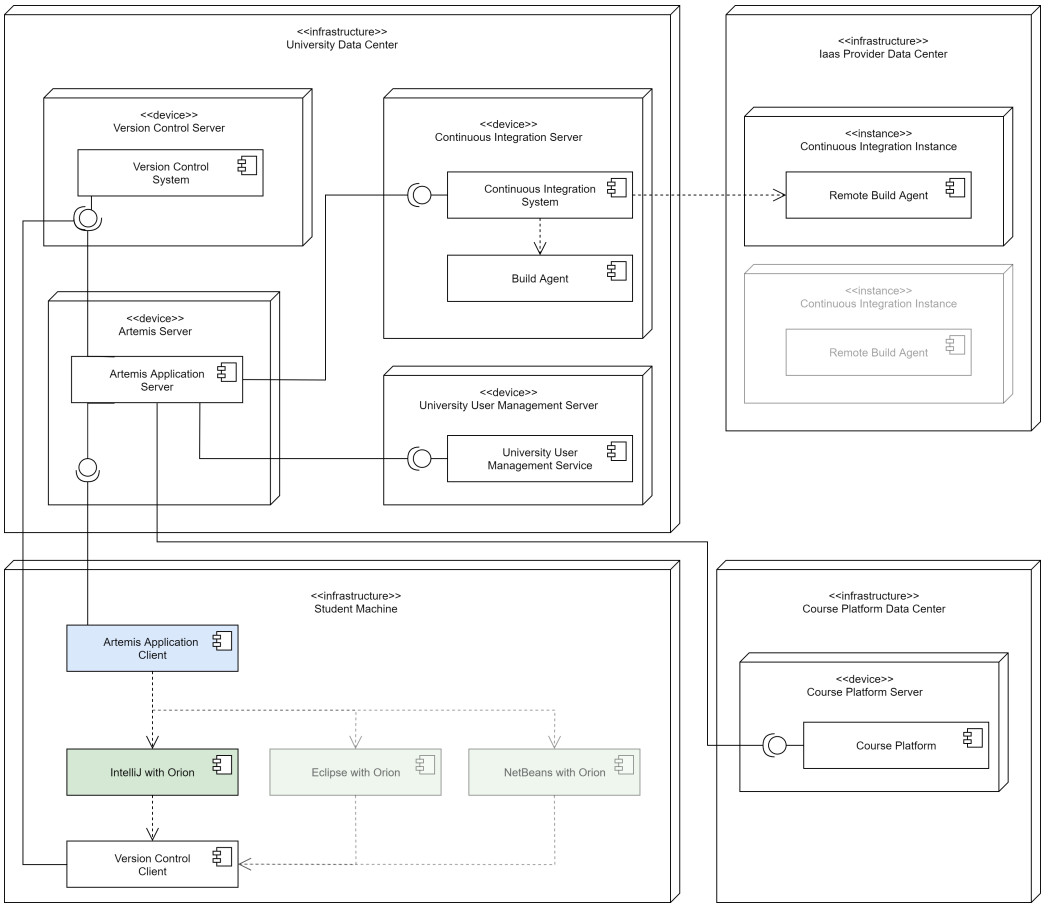


Figure 5.5: Hardware software mapping of *Artemis* and *Orion*, adapted from [MK17]. *Orion* connects the previously unrelated *Artemis* and version control clients. Updated *Artemis* components are colored in blue, new components in green.

Chapter 6

Object Design

This chapter introduces refinements related to the subsystems defined in chapter 5. As described in [BD09], we provide specifications of the concrete implementation of objects from a solution domain perspective including the offered operations, their visibilities and signatures. Throughout this process we use design patterns [GHJV95] to efficiently solve challenges related to the implementation. In order to understand the design decisions, we have to first explain some of the underlying technologies used by *Artemis* and *Orion* in the following sections.

6.1 Support for the IntelliJ IDE

Orion is written in a mix of Java[1] and Kotlin[2] and currently supports the installation in the IntelliJ IDE. The reasoning behind this mix is that the main developer of IntelliJ - JetBrains - is currently migrating the codebase from Java to Kotlin. Consequently, in order to be consistent with the latest conventions, all IntelliJ specific plugin subsystems are also implemented in the same language. However, allowing to port the plugin to other IDEs is still an option since the IDE independent core functionalities are based on Java and define generic interfaces that can then be implemented using IDE specific solutions. The following section explains why IntelliJ was chosen to be the first supported IDE and what implications IntelliJ's architecture has on the current system.

Choosing IntelliJ The choice to support IntelliJ is based on the following three main factors:

[1] `https://www.java.com/`
[2] `https://kotlinlang.org/`

1. The market share of IntelliJ has been steadily growing over the course of the past years. Surveys show, that users increasingly switch from the main competitor of IntelliJ - Eclipse - to JetBrain's IDE[3,4,5]. As programming exercises should teach students knowledge relevant to current and future software development, it is desirable to also reflect this in the choice of the officially supported IDE. Therefore, *Orion* should be available for IDEs that are actively used in productive environments.
2. Just like IntelliJ's core, *Orion* is developed and released under an open source license. Including closed-source dependencies would introduce difficulties when maintaining the plugin, or developing new features. *Orion* has to be integrated into the supported IDE, so developers have to be able to get an insight into the codebase of the embedding platform and reuse existing implementations without risking licensing conflicts.
3. The *Artemis* server is written in Java, the client in TypeScript. Fragmenting the platform by introducing another programming language raises the entry barrier for new developers and requires existing maintainers to learn an additional language if they are not familiar with it. It is therefore desirable to develop *Orion* for an IDE, which allows plugins to be based on the Java Development Kit (JDK). IntelliJ supports both Java and Kotlin, which is interoperable with Java.

Architectural principles of IntelliJ IntelliJ provides a layered architecture [BMR+96] divided into three hierarchical levels:

1. **Application Layer** All services related to the IDE application instance. Operations influence the settings and systems of the whole IDE.
2. **Project Layer** All services related to an IDE project. Operations only influence the files in the scope of the project's root directory
3. **Module Layer** All services related to an IDE project module. Projects can have multiple modules under the root directory. Modules depend on the parent project, but can be independent from each other.

This layered architecture is combined with an inversion of control (IoC) principle [Mat99], which allows developers to define a range of services and the level they should operate on. The IDE then injects the needed dependencies during runtime. *Orion* mostly uses project services: Every class, that holds

[3] `https://www.jrebel.com/blog/java-tools-map`
[4] `https://www.baeldung.com/java-ides-2016`
[5] `https://www.jrebel.com/blog/java-trends-and-historical-data`

a `project` reference as a member variable can therefore be assumed to be a project service.

6.2 Connecting Orion to Artemis

The *Artemis Client* is implemented using the Angular[6] framework, which is based on the typesafe extension of the JavaScript language TypeScript[7]. Relevant for the understanding of the connection between the IDE and the *Artemis* client is the property of TypeScript to be fully backwards compatible to JavaScript. Any JavaScript native operation can be used for communicating with the *Artemis* client. Consequently, any in the IDE displayed web browser can be used to establish a connection between the web client and the IDE itself. *Orion* uses the *JavaFX WebView*[8], which contains interfaces for executing JavaScript code on the displayed web page from a Java application and vice versa. Based on this framework, the plugin can set instances of Java objects on the `window` object in the web view[9] and initiate a bidirectional communication between *Artemis* and *Orion* via this reference. The disadvantage of this setup is the interfacing with non-typesafe JavaScript objects: All interactions only support primitive datatypes (Integers, Floats, Strings, Enumerations) and cannot directly serialize JSON[10] strings into plain old Java objects (POJOs).

Additionally, the headers of JavaScript functions are not known to the Java plugin, so any call to the client is performed by handing the `WebEngine` of JavaFx a script in form of a plain String. Figure 6.1 illustrates how *Orion* circumvents this limitations by introducing enumerations for all relevant JavaScript invocations. Every function is represented by an enum value, which holds the *method name* as a String and all *argument types*. Because the explicit function name is typed only once, i.e. when the developer adds the function to the enum, other objects can never accidentally pass the wrong names or parameters since all values are statically typed when compiling the code. Services can then call the *executeJSFunction* on the `ArtemisClientConnector` and just pass the desired enum value, which also hides the underlying implementation of the connection to the client. The connector places these calls in a *dispatch queue* and waits for the client to

[6] `https://angular.io/`

[7] `https://www.typescriptlang.org/`

[8] `https://docs.oracle.com/javase/8/javafx/api/javafx/scene/web/WebView.html`

[9] `https://www.w3schools.com/jsref/obj_window.asp`

[10] `https://www.json.org/`

finish initialization until all scripts get executed.

Lastly, the connector *listens* to all state changes of *Orion* (e.g. ongoing submissions, builds, etc.) by subscribing to the relevant message topic on the `MessageBus`. This follows a basic publish/subscribe pattern [Jac09] and waits for publishers in *Orion* to emit new messages about new events.

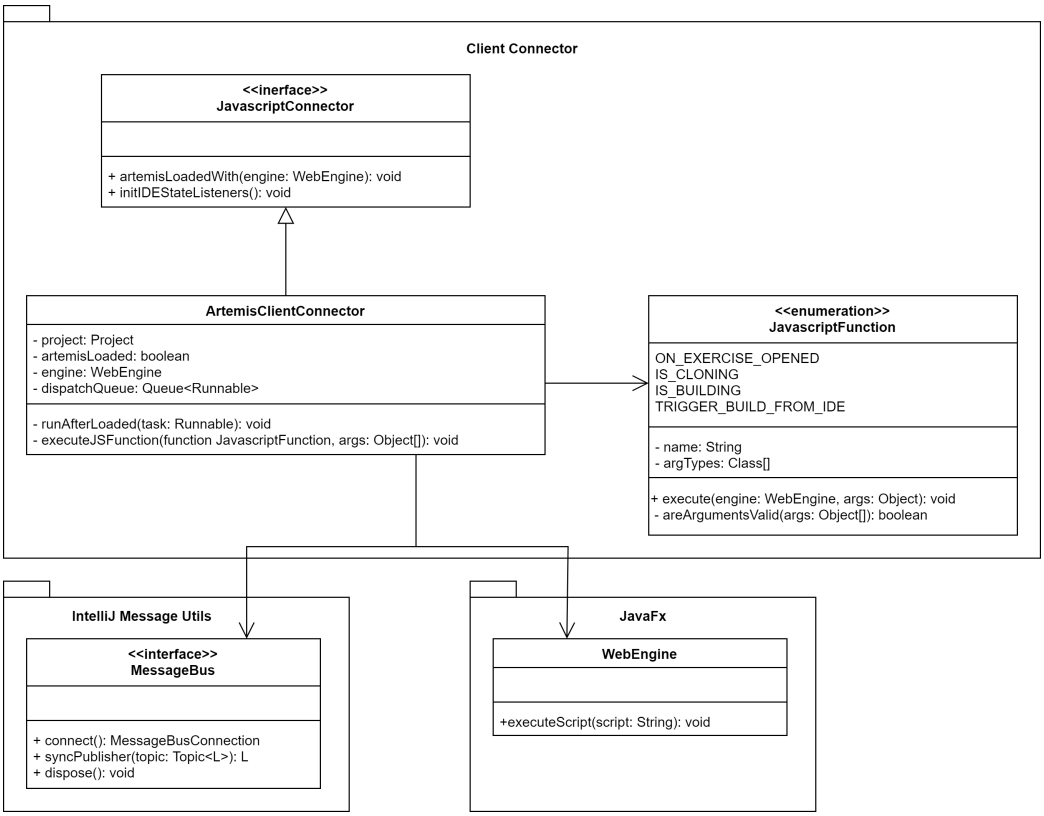


Figure 6.1: Diagram of the *Orion* classes relevant for the connection to the *Artemis* client. JavaScript functions are statically mapped using an enumeration. The connector forwards important plugin state changes by subscribing to the relevant topics on a message bus.

6.3 Connecting Artemis to Orion

The connector subsystem explained in the previous chapter describes how linking the *Artemis* client to *Orion* requires multiple connector services. Therefore, the design in figure 6.2 depicts the three main connector classes related to tests, builds and the handling of exercises. All of these provide a facade [GHJV95] to the linked client by binding multiple operations within

the plugin to a simplified interface. This is especially important, because of the non-typesafe way of setting up the interface on the JavaScript *window* object: We want to keep the number of method invocations that have to serialize the provided primitive arguments into a POJO to a minimum and hide this fact completely from the internal components of *Orion*.

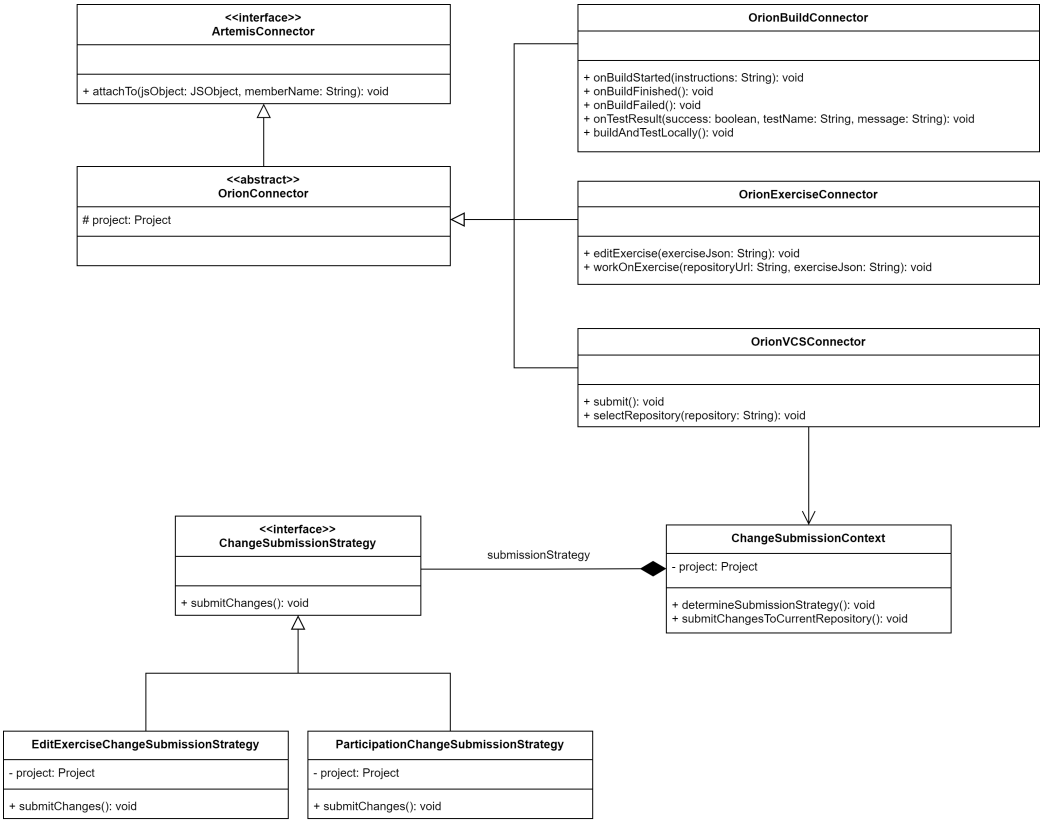


Figure 6.2: Class diagram of the connectors from the *Artemis* client to *Orion*. Connectors are split up based on the related programming exercise functionality. Submissions can be performed using different strategies: Edit exercise vs. participate in exercise.

Low coupling between connectors The classification of connectors into three areas of responsibility avoids the creation of one big controller class. This lowers the probability of creating a Blob antipattern [BMMM98], because future additions are less likely to be implemented in one single class. As programming exercises might get extended with new features (e.g. new build steps, repositories), all connecting interfaces have to be both extendable but also maintainable and should therefore not get overloaded with unrelated operations. This would only result in unnecessary high coupling.

As a result, the `ArtemisConnector` interface specifies only the `attachement` method, which gets implemented by the abstract `OrionConnector`. This abstract parent class attaches itself to the `window` object of the opened *Artemis* client website. All extending implementations then only have to focus on the actual handling of incoming requests from the client related to the workflow of exercises. This is based on the reasoning that this workflow is not related to the technical implementation of a connection between Java and JavaScript code.

We can categorize subprocesses of this workflow into the exercise import feature (`ExerciseConnector`), the handling of builds (`BuildConnector`) and the final parsing of incoming test results (`TestResultConnector`). This has the advantage of creating a separation of concerns within the component, thereby keeping the different connector implementations use case specific enough.

Supporting Different Submission Types Since *Orion* supports importing an exercise as an instructor as well as a student, the submission of changed code cannot be represented by the same process. The related connector and called interface however are identical, because the triggering action for all users is a generalized `submit`. A strategy pattern [GHJV95], which focuses on the underlying submission operations allows the plugin to determine the submission algorithm during runtime based on the selected repository and whether the user is a student, or an instructor. This also keeps the context open to new strategies e.g. for teaching assistants.

6.4 Exercise services

The `ExerciseService` and all associated classes manage the handling of the programming exercise in relation to the related *Artemis IDE Project*. This includes the import of new exercises, opening of already imported ones and the update of the wrapped repositories. The available operations and their behavior rely on the `state` of the project in the IntelliJ. A `state` is a persistent set of properties, scoped on a single project, or the whole IntelliJ installation. The state gets loaded during the start of the development environment, while a project state is only fetched for opened projects. At the end of the relevant lifecycle, both get stored on the local filesystem. Figure 6.3 gives an overview over all for the exercise management relevant components.

Exercise Registry The `ExerciseRegistry` interface is the entry point for all operations, which require information about the state of the in the IDE

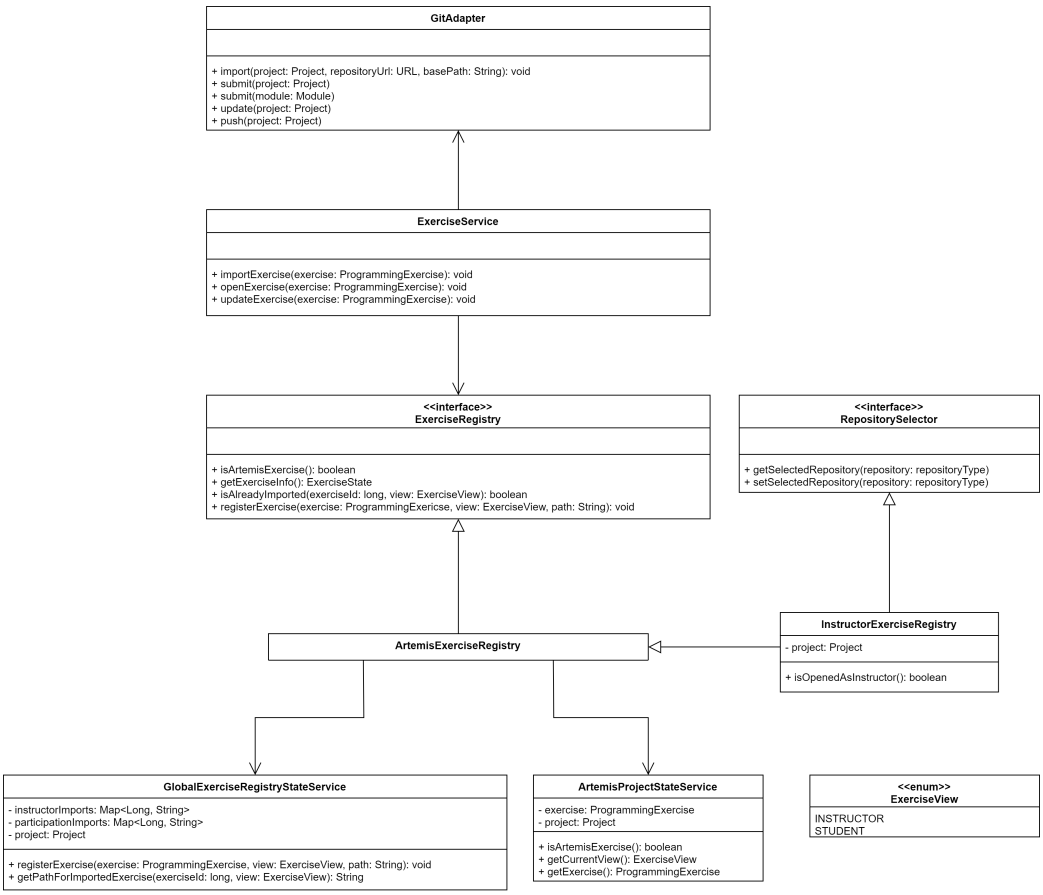


Figure 6.3: Class diagram of the *Orion* IDE project object design. The plugin uses the adapter pattern in order to provide an interface to the Git plugin of IntelliJ. *Orion* tracks imports by keeping an application wide registry with references to all downloaded exercises.

imported exercises. Because, there is a general set of methods that should be available to all users, independent of their role, there exists one single `ArtemisExerciseRegistry` that provides an implementation of the interface. This class is open to extensions by use cases that require additional operations. Editing an exercise as an instructor includes dealing with multiple repositories in a single project. Hence, instructors have to be able to select the repository they are currently working on. The `InstructorExerciseRegistry` is therefore introduced to provide support for an `ExerciseRegistry` in combination with a `RepositorySelector`.

Exercise Registry States Because *Artemis* IDE projects can be moved between IDE installations, or just be persisted and opened on a filesystem

with a simple code-editor, the `ArtemisProjectStateService` holds all for a single imported exercise relevant properties. Properties are relevant, if they are needed for identifying the exercise, fetching the selected repository (for instructor imports) or deciding whether the exercise was imported because of a participation (student) or in order to edit it (instructor). All of this information is stored in a project scoped state within the project configuration, so the IDE can identify and re-import it, e.g. if it has been moved or the IDE got re-installed.

On the application layer, the `GlobalExerciseRegistryStateService` tracks the paths of all imported exercises using a map of

$$exerciseId \rightarrow pathOnLocalFilesystem$$

This way, the IDE specific implementation regarding the tracking of imports is clearly separated from the project itself, further ensuring the low coupling between *Orion* and generated projects.

6.5 Remote Build Result Processing

The in the previous chapters described proxy has to be placed between a console responsible for displaying build results and any process, which might report remotely executed tests. Therefore, *Orion* adds a new type of build setup to the IDE, that can seamlessly switch between intermediary local results and the structured presentation of remote build feedback. A diagram containing all relevant classes can be found in figure 6.4.

Remote build configuration The `RemoteBuildService` acts as an entry point for all processes related to remote build executions. It triggers the build process in the IDE, which uses an internal `DefaultProgramRunner` to *execute* a new process using the `RemoteBuildCommandLineState`. This custom state encapsulates the run configuration and any spawned process and manages the executed build. *CommandLineStates* in general are IntelliJ native classes, that bundle processes with consoles and produce execution results to be handled by the calling IDE components. Consoles can display any form of information passed to them, though they might specialize in different types of information such as test results (like the `SMTRunnerConsoleView`). This way, a custom run configuration can use existing consoles (for displaying the run output to the user), while attaching them to customized processes, or vice versa. The `RemoteBuildCommandLineState` of *Orion* supplies a standard test console with the by the `TestResultTranslator` parsed remote build results.

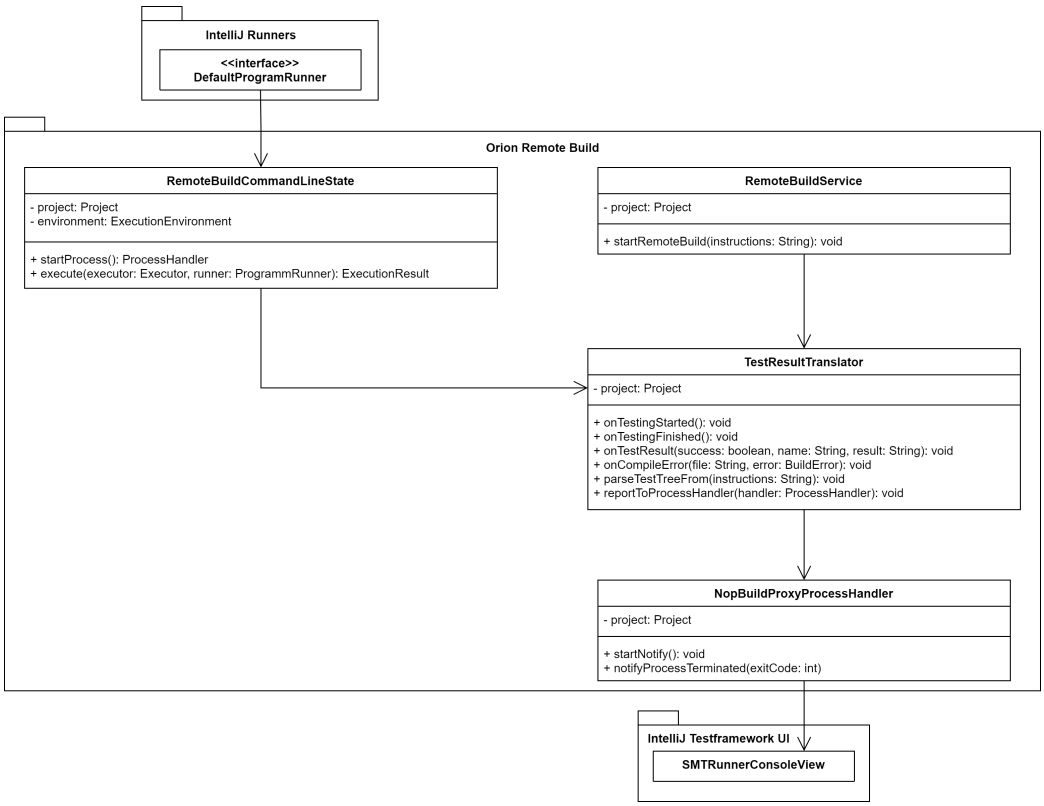


Figure 6.4: Class diagram related to the reporting of test results created by remote builds. A stand-in proxy will allow future updates to display preliminary local results until a fitting remote result has been reported to the IDE.

Test result translation Because form the *Artemis Client* incoming test results do not conform to the by IntelliJ standardized format, we introduce an intermediary translator. Furthermore, these results have to be remapped to the instructions of the programming exercise, so that the user can more easily understand the root of potential errors. Therefore, the result translator is also able to create a tree of tasks and subtasks out of the exercise instructions, which reference specific test cases. As the instructions contain identifiers of these tests [MK17], the translator can then inject the concrete results into this test tree and forward a structured report to the IDE. Figure 6.5 depicts how exercise instructions can form a tree-like structure when they are grouped based on the unique task and test names.

Proxying build results The proxy pattern [GHJV95] allows *Orion* to display a preliminary local result, so that users do not have to wait for remote builds to finish execution until they can react to potential errors. Due to

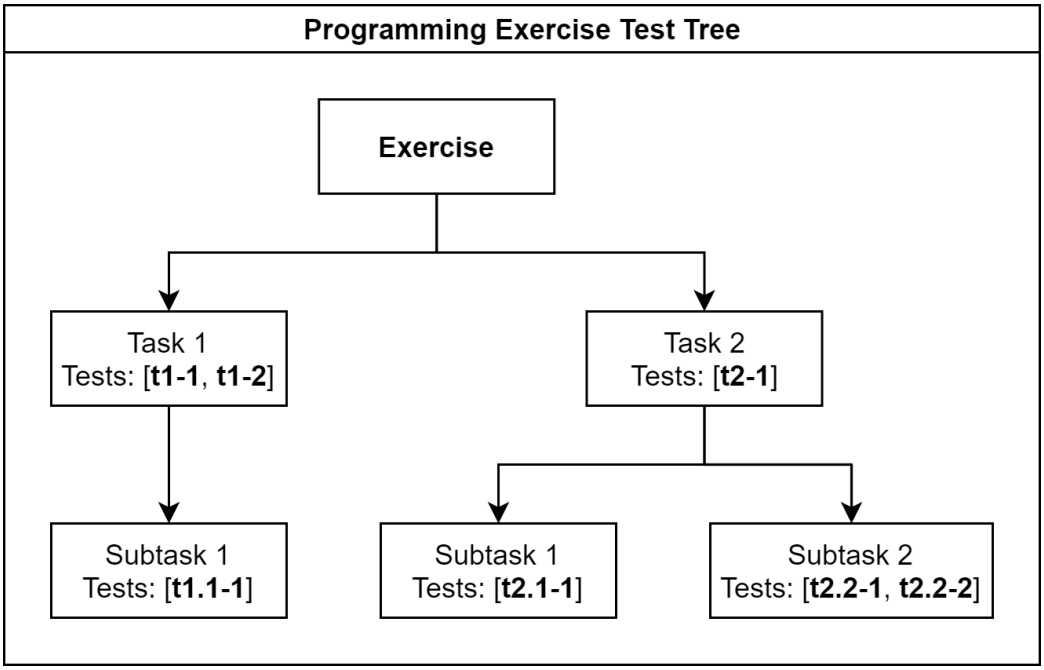


Figure 6.5: Example of an exercise divided into tasks and tests. Every task can relate to multiple tests. Instructions can always be represented by a tree, from which structured test results can be built.

time constraints, a full proxy is not yet implemented. The existing class only offers passthroughs of remote results: The `NopBuildProxyProcessHandler` represents an empty **n**o **o**peration **p**rocess, which keeps reporting a running build to the `SMTRunnerConsoleView` of the IDE. This creates the impression of a local test execution to the user, who expects this type of behavior from regular local test runs and is therefore presented with a familiar UI. As soon as the actual results arrive, the proxy just behaves as any normal process handler and forwards the translated values to the console view. A full proxy will be implemented in future iterations of the plugin.

Chapter 7

Summary

This final chapter summarizes the work and the conclusions of this thesis. We evaluate the current status of *Orion* and list all realized and open goals. This is concluded with a recap of all solved problems and an outlook on future work.

7.1 Status

In this section we list reports regarding all functional requirements related to their completion state. We categorize these goals in three statuses and provide explanations if some have not been fully implemented, yet. We define the following three categories:

- ● **Fully implemented**: The goal is has been fully realized. No additional work is necessary.
- ◐ **Partially implemented**: The goal has only been fulfilled to a certain degree. Additional work is necessary.
- ○ **Not implemented**: Implementation for this goal has not been started, yet. This goal is moved to future work.

We integrated *Artemis* and its client into a modern IDE **eliminating any media disruption** caused by the fragmentation of the UI into an IDE, VCS client and *Artemis* client. This could be achieved by implementing the IDE plugin *Orion* and connecting it to interfaces relevant to *Artemis* and VCS processes. The developed plugin can be installed by students and instructors and used for the exercise participation and administration. We decoupled the generated *Artemis* IDE projects from *Orion*, so that users can freely choose the storage location and move already downloaded projects without breaking the connection to the IDE (see table 7.1).

Functional Requirement	Status
FR3.1 Generate Artemis IDE project for exercise	●
FR3.2 Generate Artemis IDE project for participations	●
FR3.3 Customize storage location	●
FR3.4 Move Artemis projects	●

Table 7.1: Status of functional requirements related to the Artemis IDE project functionality

We further created new local run configurations for *Artemis* IDE projects, which can be used by instructors to test their changes to an exercise repository on their local machine. Moreover, all users can directly trigger builds on the *Artemis* CIS from within the IDE for all imported exercises. We **improved the responsiveness and performance** of the feedback and refinement process of the programming exercise workflow, as build results get immediately displayed in the IDE after they have been reported to the client. Users are able to react more quickly to potential errors in their submitted code as a result (see table 7.3). Dynamically switching out local with remote results is currently not fully implemented. The proxy and all necessary associations exist, but are restricted to a passtrough mode. As a consequence, only remote or local results can be displayed. The integration of simultaneously running a local build during remote execution has to be realised in a future update.

With regards to the version control functionalities, we introduced an adapter to the IDE's VCS plugin and hide complex VCS operations behind the user interface of *Orion*. Thus, we **lowered the entry barrier** for inexperienced users and provided them with the toolset to easily perform all for the programming exercise workflow necessary VCS interactions. However, we were not able to realise an automatic conflict resolution process as repositories can end up in multiple conflict states. Resolving these while still giving the user some control over the chosen approach requires more time and will therefore be moved to future work. The download of student's submissions is also not possible, but there already exist the required interfaces and methods in the VCS adapter. The remaining teaching assistant services and client connections have to be released in future versions of *Orion*.

7.2 Conclusion

In this thesis we developed the IDE plugin *Orion*. We connected the VCS and *Artemis* clients with IntelliJ to overcome media disruptions when work-

Functional Requirement	Status
FR2.1 Download participation	●
FR2.2 Download base repositories	●
FR2.3 Download student's submissions	◐
FR2.4 Edit exercise in one project	●
FR2.5 Resolve conflicts	○
FR2.6 Submit changes	●

Table 7.2: Status of functional requirements related to the VCS functionality

Functional Requirement	Status
FR2.1 Test base participations locally	●
FR2.2 Build participations remotely	●
FR2.3 Analyze test results	●
FR2.4 Display build results faster	◐

Table 7.3: Status of functional requirements related to building and testing

ing with programming exercises in *Artemis*. Because of *Orion*, exercise administration and participation processes can be performed within an IDE project. Users no longer have to interact with multiple UIs, but only need one central application to use *Artemis*. Furthermore, we hide complex VCS operations behind the plugin, which lowers the entry barrier, especially for programming beginners.

We allow students to import their participations and submit code changes directly through the IDE. Test results are displayed in a structured way according to the tasks of the exercise, so users can analyze potential errors more easily. Instructors are provided with the possibility to import all base repositories into a single IDE project, allowing them to edit exercises more efficiently. Moreover, manual copy processes are no longer needed in order to test template and solution repositories locally. We enabled this simplification by introducing the local build feature with *Orion*.

We developed a first prototype, which was tested by a group of beta users. We used the feedback from this release for our formative development process and adapted the requirements accordingly. Summarized, the current version of *Orion* allows users to interact with *Artemis* using an IDE with a comprehensible UI, which makes the whole platform more approachable and does not require switches between different systems and GUIs.

7.3 Future Work

There still are open goals regarding the development of *Orion* and new features, which would potentially improve the plugin and the user experience in several ways. Accordingly, the following section describes relevant future work.

Support for teaching assistants Submissions of users sometimes have to be graded manually by teaching assistants after the automatic testing process. This can have various reasons, e.g. if the solution to an exercise might be partially correct and an automatic test would not be able to reflect this. Teaching assistants should therefore be able to download a submission, so that they are able to run additional tests and determine a grade based on their findings from within an IDE. *Orion* already implements all necessary interfaces and adapters to the VCS plugin of IntelliJ in order to clone any remote repository into a new project. New UI elements in the *Artemis* client and the addition of a `TEACHING_ASSISTANT` type to the available exercise views would complement the missing feature.

Code reviews The current model in *Artemis* does not allow manual results for programming submissions to directly link to concrete code passages. However, it can be beneficial to the learning process, if code reviews are applied to help the student better understand specific errors or bad practices. Peer reviews have been found to have a significant impact on student's learning [WLF+12]. *Orion* could integrate reviews into the IDE by allowing instructors and teaching assistants to directly add comments to parts of the submitted code in the IDE's editor. Students would then be able to see markings on these passages and read the related comments.

Plugins for code review services of external VCS and review platforms like GitLab or Upsource by JetBrains are already available for IntelliJ[1,2]. While it is possible to integrate these external workflows into *Orion*, the result would be a dependency on a concrete service provider and IDE implementation. An independent solution would be to either commit hidden files including review comments and code section information to the submitted repository, or create new code review database models on the *Artemis* server. Reviews could then also be integrated into the online code-editor, which would add the benefit of an interoperability of reviews between the editor and IDE.

Inline hints Hints have already been integrated into *Artemis* in previous work [Beh19]. Currently, these can be assigned to individual tasks in the

[1] `https://plugins.jetbrains.com/plugin/7223-code-review-for-intellij-idea`
[2] `https://www.jetbrains.com/help/upsource/codereview-ide-plugin.html`

exercise instructions, but not to a specific line in the template code. Using Orion, instructors could assign hints directly in the IDE's editor to relevant source code sections. If tests related to a code passage referenced by a hint fail, *Orion* would display the hint to the user. The implementation of this feature would be similar to the independent approach for code reviews, meaning an update for the database models of hints, or hidden files in the repository with all required information.

Team based exercises The grouping of students in teams, who work on exercises together, has positive influences on the learning effect in various aspects [KSB+10, Las09]. *Orion* could integrate these benefits by having multiple students edit a repository together. Any changes to the source code would be visible in the editor of the IDE and different colors could indicate which participant is currently working on which line, or file. Current approaches range from visualizing changes to simultaneously edited files[3,4] to live transmissions of the cursor movements and edits for every single line of code[5]. The live transmission is based on the external provider *Floobits*[6], but the code of the plugin is released under an open source license and could be analyzed and used as a basis for an independent solution for *Artemis* and other learning platforms.

Automatic conflict resolution Asynchronous changes in multiple repository instances can result in different conflict states. If users then try to synchronize their local repository with a remote version containing changes to the same code sections, version control systems can no longer merge the changes without manual intervention. This is one of the most complicated and error-prone processes when dealing with a VCS and requires users to be aware of the outcome of their chosen resolution approach. *Orion* should simplify these approaches by only offering two basic options: *Overwrite all remote changes with the local ones*, or *Overwrite the local changes with the remote ones*. Additionally, the plugin could offer a third option for experienced users, who still want to resolve all conflicts manually. The adapted Git plugin of IntelliJ already offers resolution options. *Orion* could use these by intercepting conflict events when pulling and merging a repository, or alternatively just delegating any conflict states to the IntelliJ plugin in the first place.

[3] `https://devpost.com/software/can-ttouchthis`
[4] `https://gitlab.com/Fantailed/cant-touch-this`
[5] `https://github.com/Floobits/floobits-intellij/`
[6] `https://floobits.com/`

List of Figures

1.1 Automated assessment process of Artemis 5
1.2 Use case diagrams of a student solving a programming exercise 6

2.1 Screenshot of an opened project in IntelliJ with plugins 10
2.2 Distributed version control 12
2.3 Typical CI + VCS setup in Artemis 12

3.1 Code review of a submission in TMC 14
3.2 Architecture of TMC 15
3.3 Feedback for a failed submission using the Edu Tools plugin . 16
3.4 CodeOcean online editor 18
3.5 High level architecture of CodeOcean 18

4.1 Current system components of Artemis programming exercises 20
4.2 Proposed system of Orion 21
4.3 Import/resume exercise use cases 27
4.4 Exercise submission use cases 28
4.5 Analysis object model of Orion 29
4.6 Activity diagram of an an instructor editing an exercise 32

5.1 Overview of the Orion system design 36
5.2 Connector subsystem between Artemis and Orion 39
5.3 Build plan systems in Orion 41
5.4 Exercise subsystem of Orion 42
5.5 Hardware software mapping of Artemis and Orion 43

6.1 Orion connector to Artemis client 48
6.2 Connectors to Orion 49
6.3 Orion IDE project object design 51
6.4 Classes related to the reporting of remote test results 53
6.5 Example of an exercise test tree 54

List of Tables

7.1 Status of functional requirements related to the Artemis IDE project functionality 56
7.2 Status of functional requirements related to the VCS functionality 57
7.3 Status of functional requirements related to building and testing 57

Bibliography

[BD09] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall Press, USA, 3rd edition, 2009.

[Beh19] Jan-Thilo Behnke. Extenstion of programming exercises in artemis. Master's thesis, Technical University of Munich, 2019.

[BMMM98] William H. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., USA, 1st edition, 1998.

[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.

[Boo90] Grady Booch. *Object Oriented Design with Applications*. Benjamin-Cummings Publishing Co., Inc., USA, 1990.

[CS14] Scott Chacon and Ben Straub. *Pro Git*. Apress, USA, 2nd edition, 2014.

[FF06] Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 122:14, 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[Jac09] Hans-Arno Jacobsen. *Publish/Subscribe*, pages 2208–2211. Springer US, Boston, MA, 2009.

[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*, pages 284–289. ACM, 2018.

[KSB+10] Paul Koles, Adrienne Stolfi, Nicole Borges, Stuart Nelson, and Dean Parmelee. The impact of team-based learning on medical students' academic performance. *Academic medicine : journal of the Association of American Medical Colleges*, 85:1739–45, 09 2010.

[KvFA17] Stephan Krusche, Nadine von Frankenberg, and Sami Afifi. Experiences of a software engineering course based on interactive learning. In *SEUH*, pages 32–40, 2017.

[Las09] Patricia Lasserre. Adaptation of team-based learning on a first term programming class. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '09, page 186–190, New York, NY, USA, 2009. Association for Computing Machinery.

[Mat99] Michael Mattsson. Object-oriented frameworks. 04 1999.

[MK10] Catherine Mulryan-Kyne. Teaching large classes at college and university level: Challenges and opportunities. *Teaching in Higher Education*, 15(2):175–185, 2010.

[MK16] Dominik Münch and Stephan Krusche. Conducting interactive programming exercises in large lectures. Master's thesis, Technical University of Munich, 2016.

[MK17] Josias Montag and Stephan Krusche. Conducting interactive programming exercises in online courses. Master's thesis, Technical University of Munich, 2017.

[O'M02] Siobhan Clare O'Mahony. *The Emergence of a New Commercial Actor: Community Managed Software Projects*. PhD thesis, Stanford University, Stanford, CA, USA, 2002. AAI3048587.

[PLVV13] Martin Pärtel, Matti Luukkainen, Arto Vihavainen, and Thomas Vikberg. Test my code. *International Journal of Technology Enhanced Learning 2*, 5(3-4):271–283, 2013.

[Roc75] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, Dec 1975.

[SK18a] Valentin Schlattinger and Stephan Krusche. Extending artemis: Interactive live quizzes in the classroom. Master's thesis, Technical University of Munich, 2018.

[SK18b] Marius Schulz and Stephan Krusche. Assessment of solutions to modeling exercises in education. Master's thesis, Technical University of Munich, 2018.

[SKT+16] T. Staubitz, H. Klement, R. Teusner, J. Renz, and C. Meinel. Codeocean - a versatile platform for practical programming excercises in online environments. In *2016 IEEE Global Engineering Education Conference (EDUCON)*, pages 314–323, April 2016.

[STM17a] T. Staubitz, R. Teusner, and C. Meinel. Towards a repository for open auto-gradable programming exercises. In *2017 IEEE 6th International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*, pages 66–73, Dec 2017.

[STM17b] Thomas Staubitz, Ralf Teusner, and Christoph Meinel. openhpi's coding tool family: Codeocean, codeharbor, codepilot. In *ABP*, 2017.

[TWS17] R. Teusner, N. Wittstruck, and T. Staubitz. Video conferencing as a peephole to mooc participants: Understanding struggling students and uncovering content defects. In *2017 IEEE 6th International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*, pages 100–107, Dec 2017.

[VVLP13] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel. Scaffolding students' learning using test my code. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, page 117–122, New York, NY, USA, 2013. Association for Computing Machinery.

[WK19] Julian Willand and Stephan Krusche. Refactoring and extending the uml modeling editor apollon. Master's thesis, Technical University of Munich, 2019.

[WLF+12] Yanqing Wang, Hang Li, Yuqiang Feng, Yu Jiang, and Ying Liu. Assessment of programming language learning based on peer code review model: Implementation and experience report. *Computers & Education*, 59(2):412–422, 2012.