A Web-Based Code-Editor

For Use in Programming Courses

Christian Rasmussen
David Åse

Master of Science in Computer Science
Submission date: September 2014
Supervisor: Hallvard Trætteberg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

In this thesis a code-editor was implemented as a part of a bigger web-based system for solving programming assignments in the course TDT4100. The editor was created in order to allow the students of the class to focus solely on writing code, and not on setting up the surrounding framework (installing programming languages and IDEs, setting up projects, etc.).

The editor supports syntax highlighting, error checking, code completion, multiples classes, and running of tests, along with all of the more basic editor functionality such as block indentation, bracket matching, line-numbers, etc. The editor is embedded into each problem contained in an assignments, which allows students to solve basic and intermediate programming challenges directly in their web-browser, without the need for any setup.

The system also utilizes several gamification elements, as described in the thesis' preliminary study, "Gamification of Assignment Systems" (Åse, 2014). Responsive web design principles were used while implementing the system, which allows students to check their ranks and scores from any device (cellphone, tablet, laptop and similar). This was done in order to foster competition between the students, which will in turn increase motivation even further.

The results from the experiments performed indicate that the editor is well suited for use on programming assignments in courses such as TDT4100, TDT4110 and TDT4120, or any other course which has assignments that can be tested programmatically, as the editor has a low response time even for very large programs (64KB). However, the editor is not suited for courses such as TDT4180, or other GUI-programming courses, since the he editor is currently limited to displaying console output and test-results.

Sammendrag

I løpet av denne masteroppgaven ble det implementert en kodeeditor som en del av et større web-basert system for å løse programmeringsøvinger i faget TDT4100. Dette ble gjort for å la studentene få fokusere på å skrive kode, og ikke på rammene rundt (det å installere programmeringsspråk, konfigurere utviklingsverktøy og operativsystem, osv.).

Editoren støtter syntax highlighting, error checking, code completion, flere klasser og kjøring av tester/kode, i tillegg til mer grunnleggende funksjonalitet som blokkinnrykk, bracket matching, linjenummer, osv. Editoren er tilgjengelig i hver oppgave i en øving, hvilket gjør det mulig for studenter å løse grunnleggende og middels avanserte programmeringsøvinger direkte i nettleseren, uten å måtte tenke noe på oppsett.

Systemet benytter seg også av flere spillifiseringselementer, som beskrevet i oppgavens forstudium "Gamification of Assignment Systems" (Åse, 2014). Prinsipper for responsivt design ble brukt under implementasjonen av systemet, noe som gjør det mulig for studenter å holde seg oppdatert fra alle typer enheter (mobil, nettbrett, PC o.l.). Dette ble gjort for å tenne konkurranseinstinktet, som igjen fører til høyre motivasjon.

Testresultatene viser at editoren er godt egnet for bruk på programmeringsøvinger i fag som TDT4100, TDT4110 og TDT4200, samt andre fag med øvinger som kan testes programmatisk, siden editoren har lav responstid selv for store programmer (64KB). Editoren er derimot ikke like godt egnet for øvinger i fag som TDT4180, eller andre GUI-programmeringsfag, siden den på nåværende tidspunkt kun kan vise konsollutskrift og testresultater.

Preface

This is the project report for the subject “TDT4900 – Master Thesis”, written summer 2014. The project was conducted by Christian Rasmussen and David Åse, who are, at the time of writing, studying Computer Science at the Norwegian University of Science and Technology (NTNU). The project was completed under the supervision of Associate Professor Hallvard Trætteberg.

To get the most out of this report, the reader should have a basic understanding of computer science, and be familiar with Eclipse and its ecosystem.

Nomenclature List

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
BPMN	Business Process Modeling Notation
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DAO	Data Access Object
DSL	Domain Specific Language
DTO	Data Transfer Object
ECJ	Eclipse Compiler for Java
EMF	Eclipse Modeling Framework
GOMS	Goals, Operators, Methods, and Selection rules
GUI	Graphical User Interface
HQL	Hibernate Query Language
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IDL	Interface Definition Language
ISO	International Organization for Standardization
JAR	Java Archive
JDK	Java Development Kit
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
JSP	JavaServer Pages
JVM	Java Virtual Machine
MVC	Model-View-Controller
OCL	Object Constraint Language
ORB	Object Request Broker
ORM	Object-Relational Mapping
OSGi	Open Service Gateway initiative
PC	Personal Computer
PNG	Portable Network Graphics
RAM	Random Access Memory
REST	Representational state transfer

RMI	Remote Method Invocation
RPC	Remote Procedure Call
SQL	Structured Query Language
SSE	Server-Sent Events
SSL	Secure Sockets Layer
SUS	System Usability Scale
TA	Teaching Assistant
TCP	Transmission Control Protocol
UML	Unified Modeling Language
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
VM	Virtual Machine
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Table of Contents

1 - Introduction 1

1.1 Background and motivation 1

1.1.1 The pre-study 1

1.1.2 Thesis scope 2

1.2 Goals 3

1.3 Research approach 3

1.4 Thesis structure 4

2 - Background theory 5

2.1 Process efficiency (usability) 5

2.1.1 GOMS 5

2.1.2 BPMN 7

2.2 IDE functionality 8

2.2.1 Syntax highlighting 8

2.2.2 Error checking 8

2.2.3 Code completion 9

2.2.4 IDE architecture 10

2.3 Editor performance 13

2.3.1 Response time 13

2.3.2 Scalability 14

2.4 Responsive design 14

2.5 Gamification 16

2.6 State of the art 16

2.6.1 Educational systems 16

2.6.2 Pure editors 19

2.6.3 Comments and conclusion 23

3 - Re-engineering system processes 25

3.1 The current system processes 25

3.1.1 Publishing an assignment 25
3.1.2 Solving an assignment 25
3.1.3 Approving an assignment 34
3.2 New system processes 34
3.2.1 Publishing an assignment 35
3.2.2 Solving an assignment 35
3.2.3 Approving an assignment 39
3.3 Conclusion 39
4 - Editor architecture 40
4.1 Supported functionality 40
4.2 The evolution of the architecture 41
4.2.1 High-level architecture 41
4.2.2 Making the web-server explicit 43
4.2.3 Adding core functionality 45
4.2.4 Alternative architecture 47
4.2.5 Conclusion 50
4.3 Communication 50
4.3.1 Communication between client and server 50
4.3.2 Internal communication 51
4.3.3 Editor API 53
4.3.4 Conclusion 53
4.4 Possible improvements 54
4.4.1 Re-use Eclipse processes 54
4.4.2 Reduce the footprint of the Eclipse software 56
4.4.3 Scalability 56
4.4.4 Simplify deployment 57
5 - Experiments and results 58
5.1 Experimental plan 58
5.2 Testing editor response time 58

5.2.1 Startup time 59
5.2.2 Running code, running tests, error checking, and code completion..... 60
5.2.3 Conclusion..... 63
5.3 Testing editor scalability..... 64
5.3.1 Startup time 64
5.3.2 Running code, running tests, error checking and code completion..... 65
5.3.3 Conclusion..... 69
5.4 Testing responsive design 70
5.4.1 The “assignment overview” view..... 70
5.4.2 The “leaderboards” view 72
5.4.3 The “my progress” view..... 73
5.4.4 The “problem” view 74
5.4.5 Conclusion..... 75
6 - Evaluation and conclusion 76
6.1 Evaluation..... 76
6.2 Discussion..... 76
6.3 Future work..... 77
6.4 Closing remarks 77
7 - Bibliography..... 78
8 - Appendix A – Constraints, requirements and architecture 82
8.1 Implementation constraints 82
8.2 System requirements..... 82
8.2.1 Data model..... 83
8.2.2 Web-based code-editor 84
8.2.3 Non-functional requirements 84
8.2.4 Additional requirements..... 85
8.2.5 Requirements from the preliminary study that were cut..... 85
8.3 System architecture..... 85
8.3.1 Architectural patterns..... 85

8.3.2 System architecture in ArchiMate 88

9 - Appendix B – Implementation 91

9.1 Data storage 91

9.1.1 Querying the data model 91

9.1.2 Choice of ID 92

9.1.3 The EMF model 95

9.2 Web server 97

9.2.1 Specialized libraries for EMF 97

9.2.2 Generic web frameworks 98

9.2.3 Conclusion 102

9.3 Communication 102

9.3.1 Real-time updating of the web page 103

9.3.2 Communication with the Eclipse application 104

9.4 JavaScript frameworks and libraries 106

9.4.1 Why does this project need JavaScript? 106

9.4.2 Why do we need a frameworks or a library? 108

9.4.3 Determining possible framework candidates 108

9.4.4 Determining possible library solutions 109

9.4.5 Conclusion 110

9.5 JavaScript code-editors 111

9.5.1 Why does this project need a JavaScript code-editor? 111

9.5.2 Determining possible candidates 111

9.5.3 Determining the best candidate 112

9.5.4 Conclusion 113

9.6 CSS-framework 113

9.6.1 Why does this project need a CSS-framework? 113

9.6.2 Determining possible candidates 113

9.6.3 Conclusion 114

9.7 CSS-preprocessor 114

9.7.1 Why does this project need a CSS-preprocessor? 114
9.7.2 Determining possible candidates 115
9.7.3 LESS or SASS? 115
9.7.4 Conclusion 116
9.8 The complete stack 117
9.8.1 Technology in relation to the architecture 118
9.9 Minor technologies 119
9.9.1 LESS Hat 119
9.9.2 jQuery Collapsible 120
9.9.3 Font Awesome 120
9.9.4 Faker.js 120
10 - Appendix C – Miscellaneous 121
10.1 Requirements from the preliminary study 121
10.1.1 Functional requirements 121
10.1.2 Non-functional requirements 121
10.2 Experiment results 122
10.2.1 Startup time scalability 122
10.2.2 Run code scalability 123
10.2.3 Run tests 124
10.2.4 Error checking 125
10.2.5 Code completion 126
10.3 Setup guide 127
10.3.1 Setting up Eclipse 127
10.3.2 Installing Java 8 plugin 127
10.3.3 Setting up JRE8 in eclipse 127
10.3.4 Installing Maven plugin 127
10.3.5 Installing Akka dependencies into Eclipse 127
10.3.6 Generating model code 128
10.3.7 Create a Run configuration 128

10.3.8 Exporting to JAR 128

10.3.9 Set up automatic building 128

10.3.10 Setting up IntelliJ 128

List of Figures

Figure 1.1: Pre-study prototype – “Assignments overview” 1
Figure 1.2: Pre-study prototype - “Solve assignment” 2
Figure 2.1: A complete GOMS model 7
Figure 2.2: BPMN legend subset 7
Figure 2.3: Java syntax highlighting in Eclipse 8
Figure 2.4: Java error checking in Eclipse 9
Figure 2.5: Java code completion in Eclipse 9
Figure 2.6: The architecture of the Eclipse platform 10
Figure 2.7: Layering of the OSGi specification 11
Figure 2.8: Simplified architecture of an IDE 12
Figure 2.9: Eclipse IDE’s hierarchical structure 13
Figure 2.10: Traditional and fluid layout 15
Figure 2.11: Code Hunt’s editor 17
Figure 2.12: CodingBat’s editor 18
Figure 2.13: Learn Java Online’s editor 19
Figure 2.14: Ace code-editor 21
Figure 2.15: CodeMirror editor 21
Figure 2.16: Orion editor 22
Figure 2.17: Eclipse Flux's editor 24
Figure 2.18: Eclipse flux activity diagram 24
Figure 3.1: Solve assignment setup (current, BMPN) 26
Figure 3.2: Solve assignment weekly (current, BPMN) 30
Figure 3.3: The system's editor 35
Figure 3.4: Solve assignment weekly (new, BPMN) 37
Figure 4.1: The supported functionality in the editor 40
Figure 4.2: 2-tier architecture 41
Figure 4.3: The interfaces that the server expose 42
Figure 4.4: The Java interfaces for the communication between the client and the server 43
Figure 4.5: The architecture including the web server 44
Figure 4.6: The interfaces that the web server and the services expose 44
Figure 4.7: The architecture including the data storage and the Eclipse plugin 46
Figure 4.8: The interfaces including the Eclipse Plugin Layer 46
Figure 4.9: An example of how the data flows through the interfaces 47
Figure 4.10: Alternative architecture 48

Figure 4.11: The interfaces for the alternative architecture 49
Figure 4.12: An example of how the data flows through the interfaces (Alternative architecture) 49
Figure 4.13: The architecture including the actors 52
Figure 4.14: Messages sent between client and server (Akka messages) 53
Figure 4.15: Data flow between the web browser and the Eclipse plugin 54
Figure 4.16: A single problem per workspace 55
Figure 4.17: Multiple users per workspace 55
Figure 4.18: Multiple problems per workspace 56
Figure 4.19: Distribution of plugins across multiple nodes 57
Figure 5.1: Editor startup time 60
Figure 5.2: Editor scalability prediction 68
Figure 5.3: “Assignment overview” view (laptops and desktop computers) 71
Figure 5.4:” Assignment overview” view (mobile devices) 71
Figure 5.5: “Leaderboard” view 72
Figure 5.6: “My progress” view 73
Figure 5.7: “Editor” view 74
Figure 5.8: “Editor” view with maximized editor 75
Figure 8.1: Data model requirements 83
Figure 8.2: The Client-Server pattern 86
Figure 8.3: The classic MVC pattern 86
Figure 8.4: Typical web-MVC pattern 87
Figure 8.5: Publish assignment architecture 88
Figure 8.6: Solve assignment architecture 89
Figure 8.7: Approve assignment architecture 90
Figure 9.1: EMF model 95
Figure 9.2: Dynamic instance for validating the model 96
Figure 9.3: Relative popularity between 5 web frameworks 99
Figure 9.4: Relative popularity between 5 more web frameworks 100
Figure 9.5: The relative popularity between the 4 most popular web frameworks 100
Figure 9.6: A typical implementation model of Java RMI 105
Figure 9.7: A client sends a request through its local Object Request Broker (ORB) and to a remote ORB’s servant 105
Figure 9.8: Pure HTML code editor 107
Figure 9.9: A JavaScript-powered editor 107
Figure 9.10: JavaScript MVC-framework popularity 108

Figure 9.11: JavaScript library usage survey 110
Figure 9.12: CSS-framework popularity 114
Figure 9.13: SASS syntax example 116
Figure 9.14: LESS syntax example 116
Figure 9.15: The complete technology stack 117
Figure 9.16: MVC implementation 118
Figure 9.17: Client-Server implementation 119

List of Tables

Table 2.1: JavaScript code-editor comparison 20
Table 2.2: Code-editor feature comparison 23
Table 5.1: David's machine specs 59
Table 5.2: Christian's machine specs 59
Table 5.3: Editor startup time 60
Table 5.4: Running code (performance experiment results) 61
Table 5.5: Running tests (performance experiment results) 61
Table 5.6: Error checking (performance experiment results) 62
Table 5.7: Code completion (performance experiment results)a 62
Table 5.8: Editor performance results summary 63
Table 5.9: Editor performance (goals vs results) 63
Table 5.10: Editor startup scalability 64
Table 5.11: Running code (scalability experiment result) 66
Table 5.12: Running tests (scalability experiment result) 66
Table 5.13: Error checking (scalability experiment result) 67
Table 5.14: Code completion (scalability experiment result) 67
Table 5.15: Editor scalability results summary 67
Table 5.16: Editor scalability prediction 68
Table 9.1: List of Java web frameworks (as of 5/26/2014) 98
Table 9.2: JavaScript framework comparison 109
Table 9.3: JavaScript code-editor comparison 111
Table 9.4: Code-editor feature comparison 112
Table 9.5: LESS and SASS features 115
Table 9.6: The complete technology stack 117
Table 9.7: LESS Hat and generated CSS 119
Table 10.1: Startup time (2 clients) 122
Table 10.2: Startup time (4 clients) 122
Table 10.3: Startup time (8 clients) 122
Table 10.4: Startup time (16 clients, time in seconds) 122
Table 10.5: Run code (2 clients) 123
Table 10.6: Run code (4 clients) 123
Table 10.7: Run code (8 clients) 123
Table 10.8: Run code (16 clients, time in seconds) 123
Table 10.9: Run tests (2 clients) 124

Table 10.10: Run tests (4 clients) 124
Table 10.11: Run tests (8 clients) 124
Table 10.12: Run tests (16 clients, time in seconds) 124
Table 10.13: Error checking (2 clients) 125
Table 10.14: Error checking (4 clients) 125
Table 10.15: Error checking (8 clients) 125
Table 10.16: Error checking (16 clients, time in seconds) 125
Table 10.17: Code completion (2 clients) 126
Table 10.18: Code completion (4 clients) 126
Table 10.19: Code completion (8 clients) 126
Table 10.20: Code completion (16 clients, time in seconds) 126

1 - Introduction

This chapter contains the background and motivation for the thesis, the thesis goals, the research approach for the thesis, and the thesis structure.

1.1 Background and motivation

This thesis builds on a pre-study where the primary motivation was to utilize gamification to create a prototype for a fun and engaging assignment system which could motivate students and increase the overall learning outcome of a programming class (Åse, 2014).

1.1.1 The pre-study

In the pre-study, Åse mentions two primary goals for utilizing gamification in higher level programming courses:

- To increase student motivation
- To increase the learning outcome of a class

By examining case studies and theory, the specific gamification elements which would be most likely to fulfill these goals were identified and a conceptual model for a gamified assignment system was created. The current assignment system of TDT4100 was then fitted into the conceptual model, and the refinements were made where needed. After this, an interactive prototype utilizing these elements was created and a usability test was conducted.

The prototype (Figure 1.1, Figure 1.2) was designed top down in Adobe Fireworks, and the design was completely platform independent. The choice of platform was discussed as a part of the pre-study, but no decision was reached. No technologies for implementation were discussed.

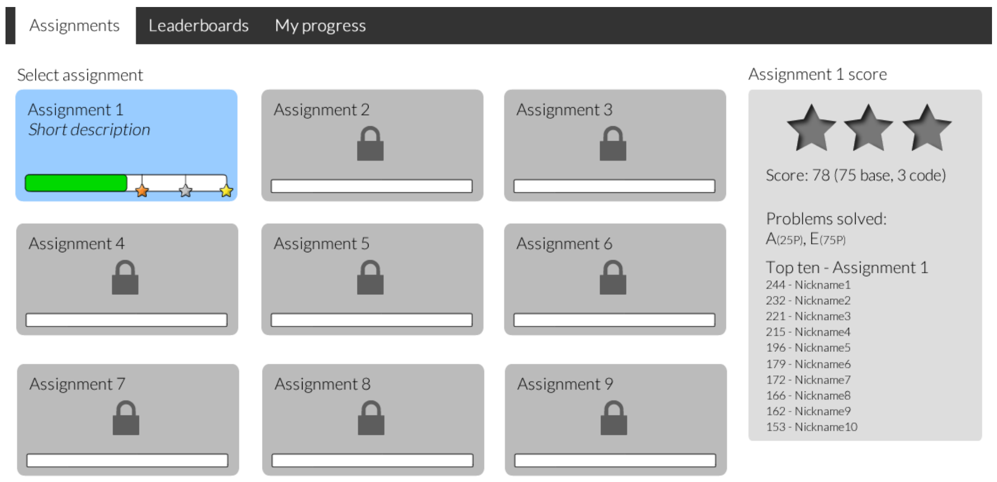
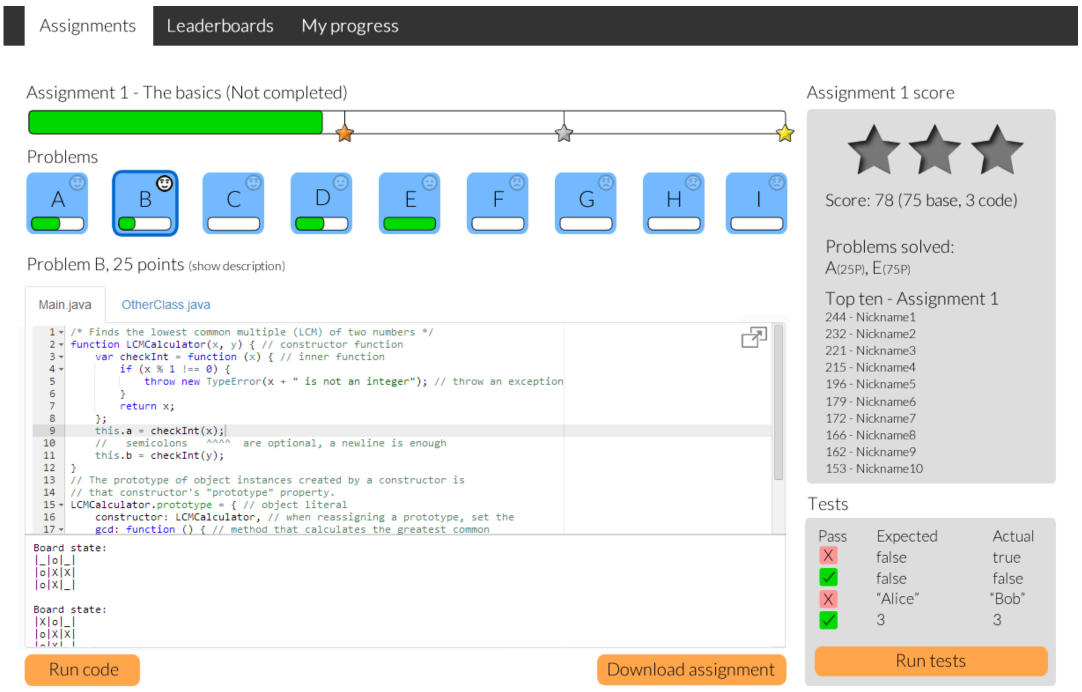


Figure 1.1: Pre-study prototype – "Assignments overview"



```
/* Finds the lowest common multiple (LCM) of two numbers */
function LCMCalculator(x, y) { // constructor function
    var checkInt = function (x) { // inner function
        if (x % 1 !== 0) {
            throw new TypeError(x + " is not an integer"); // throw an exception
        }
        return x;
    };
    this.a = checkInt(x);
    //   semicolons   ^^^^  are optional, a newline is enough
    this.b = checkInt(y);
}
// The prototype of object instances created by a constructor is
// that constructor's "prototype" property.
LCMCalculator.prototype = { // object literal
    constructor: LCMCalculator, // when reassigning a prototype, set the
    gcd: function () { // method that calculates the greatest common
```

```
Board state:
|_|o|_|
|o|X|X|
|o|X|_|

Board state:
|X|o|_|
|o|X|X|
```

Assignment 1 score

Score: 78 (75 base, 3 code)

Problems solved:
A(25P), E(75P)

Top ten - Assignment 1

244	Nickname1
232	Nickname2
221	Nickname3
215	Nickname4
196	Nickname5
179	Nickname6
172	Nickname7
166	Nickname8
162	Nickname9
153	Nickname10

Tests

Pass	Expected	Actual
🟥 X	false	true
✓	false	false
🟥 X	"Alice"	"Bob"
✓	3	3

Figure 1.2: Pre-study prototype - "Solve assignment"

1.1.2 Thesis scope

This thesis aims to implement the prototype described in the pre-study, while also focusing on streamlining the assignment system processes.

The overall design of the prototype from the pre-study was well received (by the test candidates), so only minor changes will be made, but some focus will also be given to implementing the system with a responsive design. This will ensure a good user experience whether the students access the system from their cellphones, tablets, or laptops, and will allow the more competitive students to always remain updated on their leaderboard placements and rankings, further increasing motivation.

We will narrow the scope from the pre-study, and focus primarily on implementing functionality for solving one assignment. We will implement the prototype as a web application, as this will allow the students to focus all of their attention on the code they are writing, and not the usual setup associated with programming in Java (installing JDK, installing an IDE, downloading and configuring JUnit tests, etc.). To ensure minimal setup and configuration we will try to create a web-based code-editor which requires no setup, and which will allow students to write and run Java code directly in their web browser. We aim to support advanced functionality of modern development tools, mainly syntax highlighting, error detection, and code completion.

The data model and the backend services for the editor will be modelled in Eclipse Modeling Framework, as this is a formal requirement from the faculty, but the other technologies will be decided as part of the thesis.

1.2 Goals

- Goal 1: To re-engineer the assignment processes to allow the students to focus on writing code, and not setting up the surrounding framework (Java, IDE, downloading tests, etc.)
- Goal 2: To create a web-based code-editor with good IDE-functionality.
- Goal 3: To determine what kind of assignments in TDT4100, and in general, the finished editor is suited for (size, complexity, etc.).
- Goal 4: To implement the prototype system as described in the pre-study, focusing primarily on one assignment.
    - Goal 4.1: To decide which architectural patterns to use, which technologies to use, etc.
    - Goal 4.2: To utilize responsive design, providing students with a good user experience on cellphones, tablets and computers.

1.3 Research approach

The thesis has multiple goals which require different approaches.

Goal 1 – Re-engineering the assignment processes

To meet the goal pertaining to streamlining the assignment processes (reducing the overhead related to configuring a computer when solving programming assignments, allowing students to focus all of their attention on actual programming), an analytical approach is necessary. We will analyze the tasks in the current assignment system using GOMS (Goals, Operators, Methods, Selection rules) analysis, as described in section 2.1.1. We will also model the system in BPMN, as described in section 2.1.2, to get a good overview of the system and its different environments. We will then create a new system based on the findings from the first modeling session. When the new system is finished, we will analyze and model it using the same notation, making it easy to compare the old and the new system. We will then discuss the comparison.

Goal 2 – Creating a good editor

To meet the goal pertaining to editor quality, we will use both an analytical and empirical approach. To get the graphical design right, we will try to mimic existing IDEs as they are founded on years of experience. To ensure that our technical functionality is good, we will model the problem domain using diagram tools, and validate these models with real data. Based on the resulting models we can implement a solution using suitable design patterns. Finally, we will run some quantitative experiments to ensure that everything works as intended.

Goal 3 – Editor use cases

To meet the goal to determine the optimal use cases for our finished editor, we will analyze the empirical results obtain during the testing of the editor and compare the editors performance to the needs of assignments in the current TDT4100 assignment system, along with the assignments in other courses' assignment systems. This will be a brief discussion and not a thorough analysis.

Goal 4.1 – System implementation

To meet the system implementation goal we will use an analytical approach to determine the most suited technology candidates for our system. We will find the most used technologies by examining online articles, forums and trends. We will then analyze the technologies mentioned and compare them to each other. When we have a good understanding of the potential candidates, we can determine which of the technologies that fits our project better, and include them in our implementation stack.

Goal 4.2 – Responsive design implementation

To meet the goal about responsive design, the principles described in section 2.4 will be utilized, and the system will be continuously tested with an emulation service during the design process. A final test will be performed after the system is finished, and the results (screenshots from all devices) will be included in the report, along with comments about the design.

1.4 Thesis structure

Chapter 1 describes the motivation and background for the thesis. Chapter 2 covers the necessary background theory. Chapter 3 contains an analysis of the current assignment system processes, and introduces a new set of processes. Chapter 4 describes the editor architecture. Chapter 5 covers experiments and results. Chapter 6 concludes the thesis.

Appendix A and B contain the system realization details, such as requirements, architecture, and the technological stack used to implement the system, while Appendix C is reserved for miscellaneous information.

2 - Background theory

This chapter describes the necessary background theory needed to implement the editor and the prototype, along with a state of the art analysis of other services.

2.1 Process efficiency (usability)

The ISO definition of usability is as follows:

Usability - ISO 9241 definition (World Wide Web Consortium, 2014)

The effectiveness, efficiency and satisfaction with which specified users achieve specified goals in particular environments.

- *effectiveness: the accuracy and completeness with which specified users can achieve specified goals in particular environments*
- *efficiency: the resources expended in relation to the accuracy and completeness of goals achieved*
- *satisfaction: the comfort and acceptability of the work system to its users and other people affected by its use*

In the pre-study for this thesis, the effectiveness and satisfaction of the prototype were measured, but efficiency was not considered. This thesis expands on the work done in the pre-study, and one of the goals of this thesis is to create a system which will enable the students to maximize their efficiency, allowing them to learn more programming (goals achieved) per hour spent (resources expended). We believe that a lot of overhead related to computer configuration and assignment setup can be removed by re-engineering the processes in the system.

To achieve this re-engineering, we will use GOMS and BPMN to analyze the tasks and processes in the current system, identifying any time drains. After the analysis is done we will create a new set of processes, and use the same method to analyze them.

2.1.1 GOMS

The GOMS model is centered on four principles: **g**oals, **o**perators, **m**odels, and **s**election rules. The general GOMS concept is defined very simply:

It is useful to analyze the knowledge of how to do a task in terms of goals, operators, methods, and selection rules. (Bonnie E & David E, 1996)

We will explain each category with examples, before we present an example of a GOMS model containing all four concepts.

Goals

Goals refers to what the user wishes to accomplish by using the software. They are usually divided into subgoals. For example, a student using TDT4100's assignment system has a main goal of being allowed to take the final exam. To meet this goal he first has to collect enough points for each individual assignments, and to earn points for the assignments there is yet a number of subgoals he has to meet.

Operators

Operators are defined as the actions that the software enables the user to perform. In most GOMS-models operators are limited to concrete actions, like button events, menu-click events, input events, and so on, but they can also be defined on a more abstract level. We will use a higher level definition in this thesis, such as "navigate to".

Methods

Utilizing system operators to reach a goal or a subgoal is called a method. For example, a process which involves using "It's learning" to navigate to the test code for assignments in the current assignment system for TDT4100 can be classified as a method.

Selection rules

Selection rules become necessary when you have more than one method to reach the same goal. For example, in the current assignment system of TDT4100, you can either download test code by navigating to the correct file from "It's learning" each time, or you could clone the entire repo once by visiting GitHub, where the source code is hosted. Selection rules are personal and based on the user's preferences.

GOMS optimization

GOMS analysis is often used to estimate the time needed to perform a task by breaking it down into the detailed steps a user need to perform to successfully complete it and measuring the time each step takes (usually in milliseconds). This information can then be used to optimize the processes in the system. Due to the high abstraction level we operate on in this thesis, we will only compare the number of steps, not the time each step takes.

Figure 2.1 shows a complete GOMS model which includes all four concepts. You have an initial situation, a selection between two methods (each of which have different operators) and a goal. Not all models need selection rules. Many companies, such as Apple, argue that the fewer methods there that are available to the user, the easier the interface is to understand. We agree.

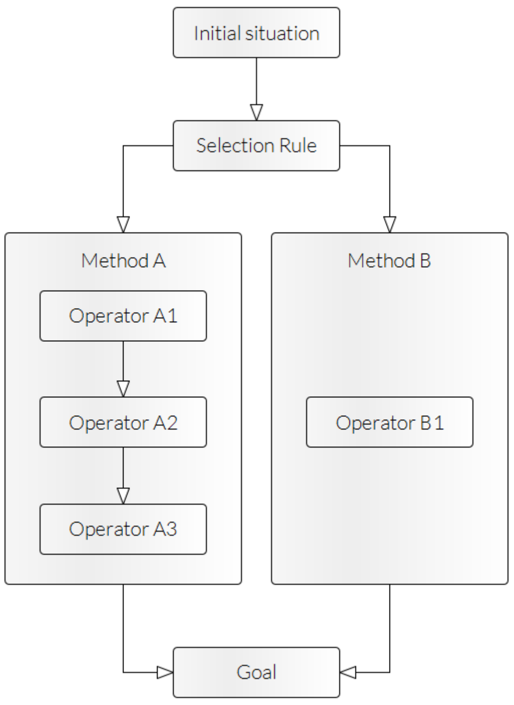


Figure 2.1: A complete GOMS model

2.1.2 BPMN

BPMN (Business Process Model Notation) is used to model business processes. The BPMN legend is very comprehensive, but we will only use BPMN to give our GOMS-tasks context, which means we will only need to use a small subset of the legend. This subset can be seen in Figure 2.2. Swim lanes are typically used to represent entities or roles, but we will use them to represent the different environments the tasks take place in.

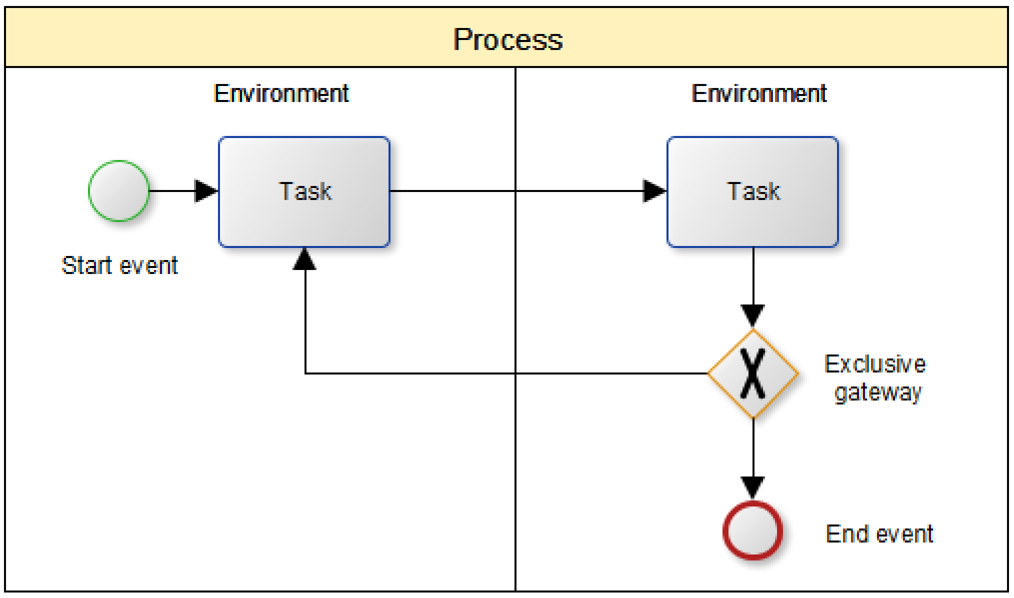


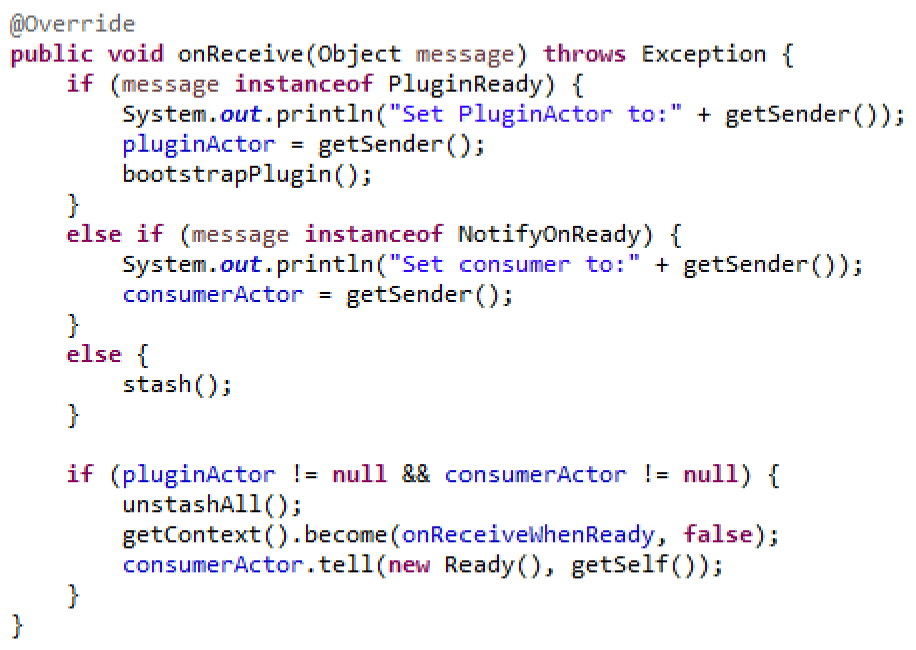
Figure 2.2: BPMN legend subset

2.2 IDE functionality

In this section we will look at typical functionality that IDEs include and how the IDEs are architected to support this functionality.

2.2.1 Syntax highlighting

Syntax highlighting is a very common feature among IDEs and advanced text editors and it is used to improve the readability of code. It works by changing the color of certain key elements of the code, such as string literals and their delimiters, variables, etc. Figure 2.3 shows Java syntax highlighting in the Eclipse IDE.

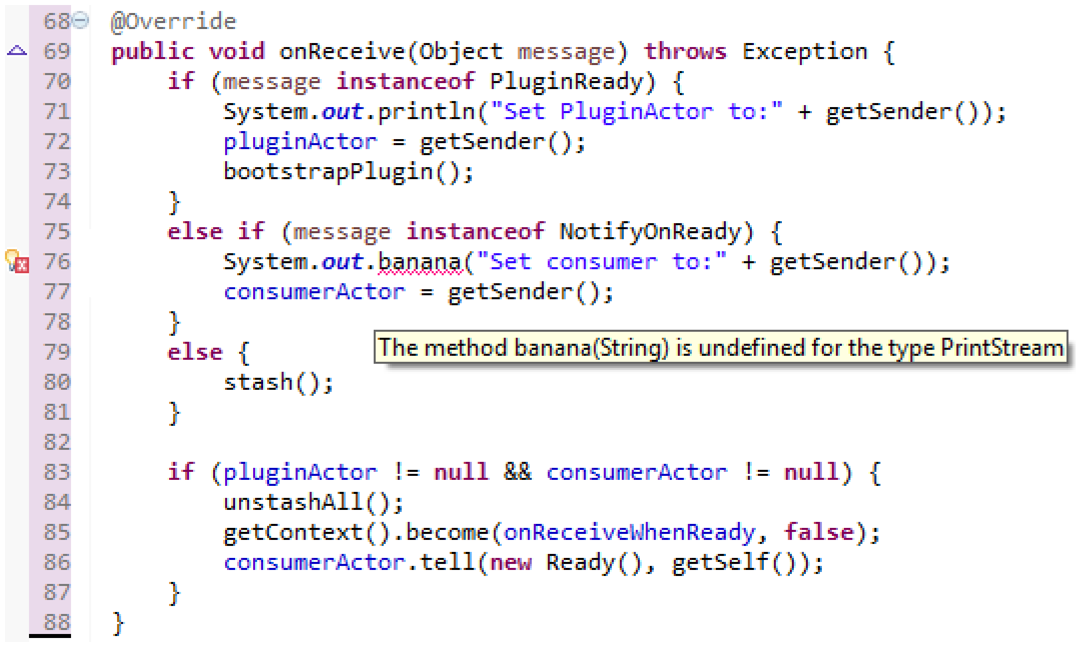


```
@Override
public void onReceive(Object message) throws Exception {
    if (message instanceof PluginReady) {
        System.out.println("Set PluginActor to:" + getSender());
        pluginActor = getSender();
        bootstrapPlugin();
    }
    else if (message instanceof NotifyOnReady) {
        System.out.println("Set consumer to:" + getSender());
        consumerActor = getSender();
    }
    else {
        stash();
    }

    if (pluginActor != null && consumerActor != null) {
        unstashAll();
        getContext().become(onReceiveWhenReady, false);
        consumerActor.tell(new Ready(), getSelf());
    }
}
```

Figure 2.3: Java syntax highlighting in Eclipse

2.2.2 Error checking

Error checking is another common feature among IDEs, but it is much less common in text editors, as many forms of error checking relies on the language compiler or interpreter. By changing one of the `println` lines in Figure 2.3 to `banana`, Eclipse gives us the error checking result that can be seen in Figure 2.4. The whole method gets a red gutter, the line with the mistake is annotated with a light bulb and red square, and the incorrect word is underlined.

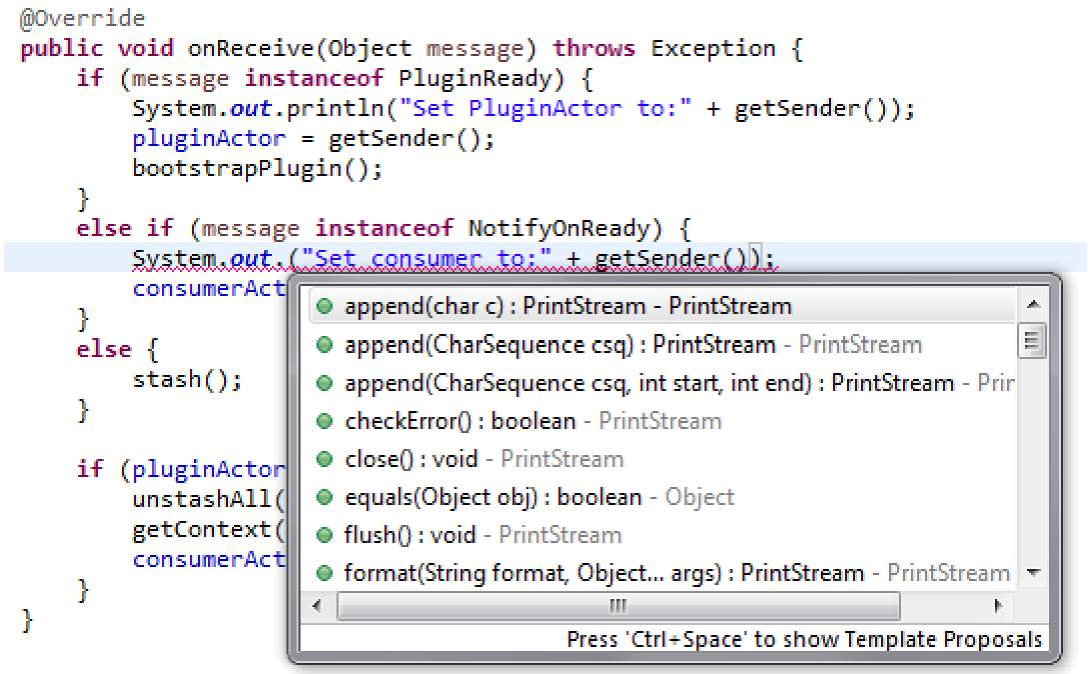


```
@Override
public void onReceive(Object message) throws Exception {
    if (message instanceof PluginReady) {
        System.out.println("Set PluginActor to:" + getSender());
        pluginActor = getSender();
        bootstrapPlugin();
    }
    else if (message instanceof NotifyOnReady) {
        System.out.banana("Set consumer to:" + getSender());
        consumerActor = getSender();
    }
    else {              The method banana(String) is undefined for the type PrintStream
        stash();
    }

    if (pluginActor != null && consumerActor != null) {
        unstashAll();
        getContext().become(onReceiveWhenReady, false);
        consumerActor.tell(new Ready(), getSelf());
    }
}
```

Figure 2.4: Java error checking in Eclipse

2.2.3 Code completion

Code completion is about as common in IDEs and text editors as error checking is. While it is not strictly necessary to rely on the compiler or interpreter to create code completion, it helps provide much better suggestions than a simple dictionary approach would. We will now try to erase the banana from Figure 2.4 and see what Eclipse can come up with (Figure 2.5).



```
@Override
public void onReceive(Object message) throws Exception {
    if (message instanceof PluginReady) {
        System.out.println("Set PluginActor to:" + getSender());
        pluginActor = getSender();
        bootstrapPlugin();
    }
    else if (message instanceof NotifyOnReady) {
        System.out.("Set consumer to:" + getSender());
        consumerAct
    }
    else {
        stash();
    }

    if (pluginActor
        unstashAll(
        getContext(
        consumerAct
    }
}
```

```
append(char c) : PrintStream - PrintStream
append(CharSequence csq) : PrintStream - PrintStream
append(CharSequence csq, int start, int end) : PrintStream - Prir
checkError() : boolean - PrintStream
close() : void - PrintStream
equals(Object obj) : boolean - Object
flush() : void - PrintStream
format(String format, Object... args) : PrintStream - PrintStream
Press 'Ctrl+Space' to show Template Proposals
```

Figure 2.5: Java code completion in Eclipse

Most IDEs sort their code completion list alphabetically, not taking into account the different usage statistics for each method.

2.2.4 IDE architecture

As we are going to create an editor, it is natural to look at how existing IDEs are architected. In this section we will focus on the Eclipse IDE, as this is the IDE of choice of the course responsible, and its architecture is representative for modern IDEs.

Modularity

The architecture of Eclipse is designed to be very modular, shown in Figure 2.6. The modularity is achieved by bundling code into plugins. Even some of the internal subsystems are written as plugins, i.e. the Java Development Tooling (JDT) and the Plugin Developer Environment (PDE). These specific plugins help create new plugins by exposing the internal functionality of Eclipse (The Eclipse Foundation, 2014).

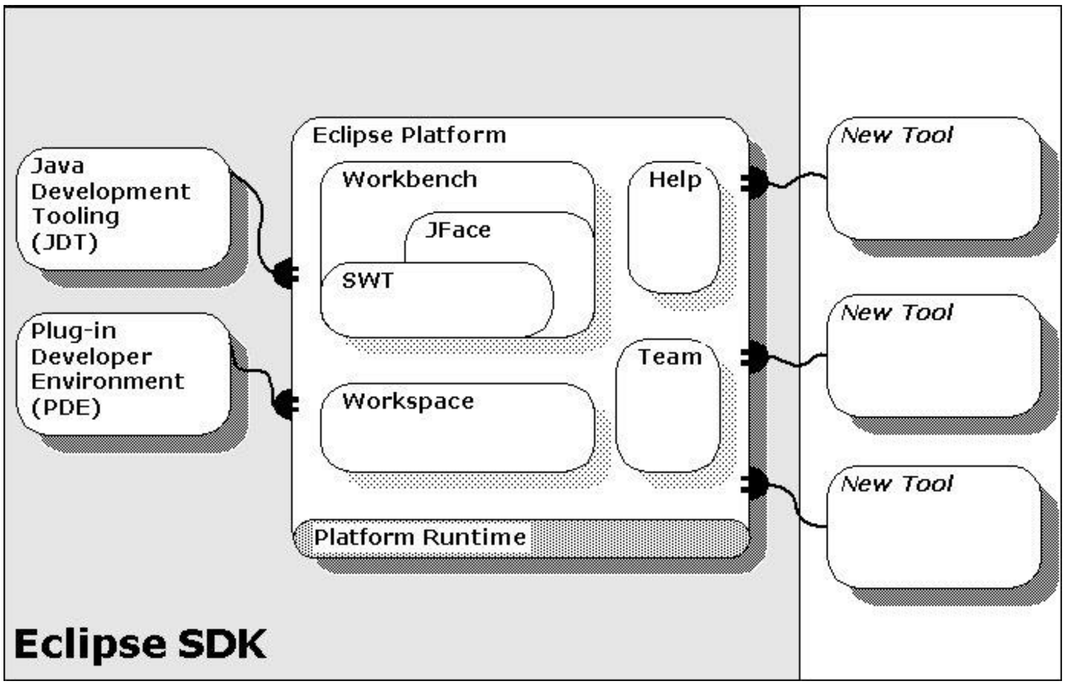


Figure 2.6: The architecture of the Eclipse platform[1]

The plugin support is based on the Open Service Gateway initiative (OSGi) specification. The layering of the OSGi specification is shown in Figure 2.7.

[1] Source: http://help.eclipse.org/juno/topic/org.eclipse.platform.doc.isv/guide/images/sdk-arch.jpg

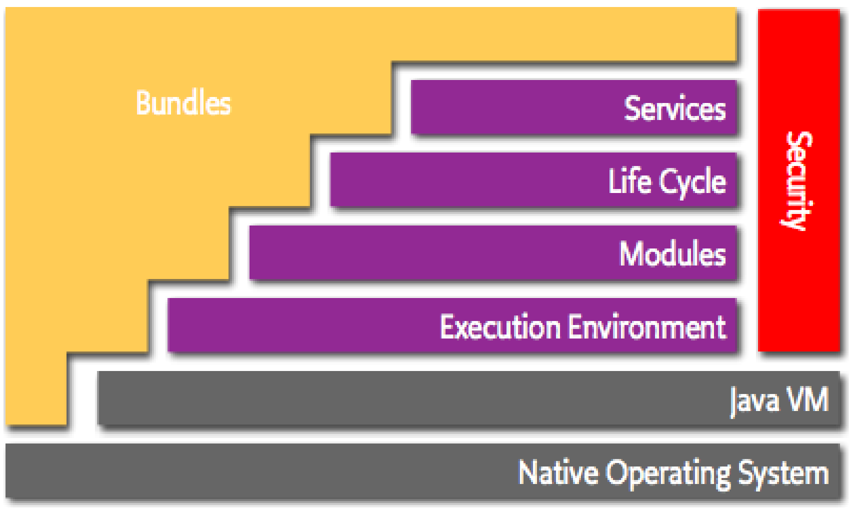


Figure 2.7: Layering of the OSGi specification[2]

The OSGi Alliance has defined these layers as follows (OSGi Alliance, 2014):

- Bundles – Bundles are the OSGi components made by the developers.
- Services – The services layer connects bundles in a dynamic way by offering a publish-find-bind model for plain old Java objects.
- Life-Cycle – The API to install, start, stop, update, and uninstall bundles.
- Modules – The layer that defines how a bundle can import and export code.
- Execution Environment – Defines what methods and classes are available in a specific platform.
- Security – The layer that handles the security aspects.

The OSGi specification makes it is easy to create plugins that depend on each other. A common way to structure the plugins is to create GUI components that mirror some core components. A simplified view of this division between components is shown in Figure 2.8.

[2] Source: http://www.osgi.org/wiki/uploads/About/layering-osgi.png

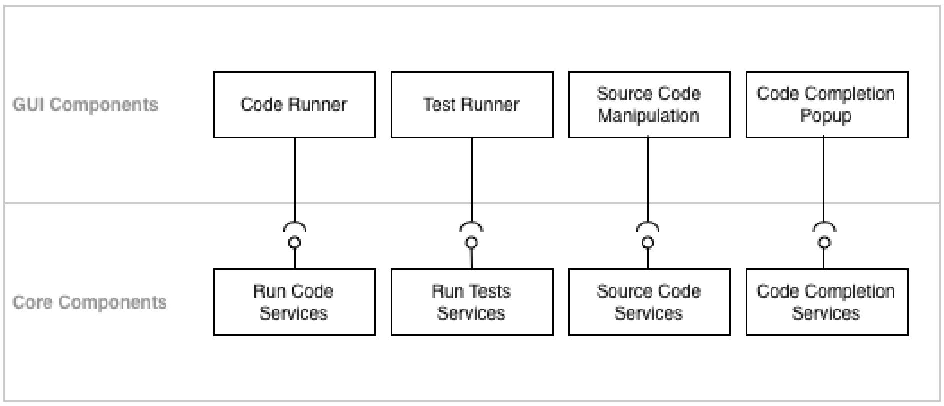


Figure 2.8: Simplified architecture of an IDE

The advantage of this structure is that you can have different GUI components that use the same core components. Or said in another way, it is possible to replace the GUI components with a different set of GUI components, e.g. a web interface.

Code analysis

To support syntax highlighting, error checking and code completion the Eclipse IDE must perform a lexical, syntactic and semantic analysis on the source code files. These types of analytic processes (ordered from the easiest to the hardest) can be described as:

- Lexical analysis – The process of converting a sequence of characters into a sequence of tokens.
- Syntactic analysis – The process of analyzing a sequences of tokens according to the rules of a formal grammar.
- Semantic analysis – The process of adding semantic information to the parse tree and building the symbol table.

Syntax highlighting can be implemented using lexical and syntactic analysis, while error checking and code completion also require semantic analysis. Because semantic analysis requires a full-blown compiler, general-purpose text editors (such as Sublime Text 2[3], Ace.js[4]) usually only include syntax highlighting and not error checking and code completion. To get error checking and code completion you must turn to more powerful IDEs like Eclipse, where the compiler is integrated.

Code analysis is performed during the build process. In the case of the Eclipse IDE, the project is built incrementally whenever a file changes. To avoid introducing errors in other files, the Eclipse

[3] Available at: http://sublimetext.com
[4] Available at: http://ace.c9.io

IDE has a notion of a working copy. A working copy is an open document (a source file in this case) that is not saved to disk. The point of a working copy is that this file will be analyzed separately from the other source files, i.e. if you remove a method in a class that other classes utilize, the other files will not report any errors until the working copy is saved to disk.

The Eclipse IDE organizes the source files in a hierarchical structure, as shown in Figure 2.9:

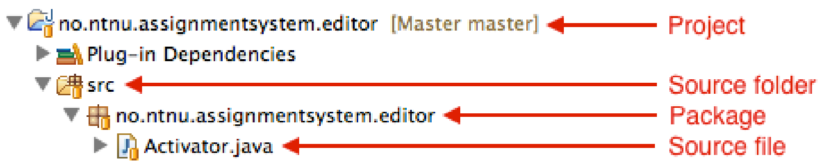

no.ntnu.assignmentsystem.editor [Master master]
Project
Plug-in Dependencies
src
Source folder
no.ntnu.assignmentsystem.editor
Package
Activator.java
Source file

Figure 2.9: Eclipse IDE's hierarchical structure

The Eclipse IDE contains exactly one workspace at a time (not shown in the figure), a workspace may contain multiple projects, a project may contain multiple source folders, a source folder may contain multiple packages, and a package may contain multiple source files.

2.3 Editor performance

Performance is often thought of as the useful amount of work done by a system, compared to the time and resources spent to achieve that work. There are many ways to measure performance, for example, response time, throughput, scalability, availability, etc. In this thesis we will focus primarily on response time and scalability.

2.3.1 Response time

Responsive time is defined as the total amount of time it takes for a computer to respond to a request. Since our system is web-based, this will include the transmission time (the time spent traveling back and forth across the internet) and the time it takes to establish an internet connection, in addition to the performance of our code. We want our editor to behave like a native IDE, so in order to ensure good usability all of these have to be taken into account.

The transmission time is often the limiting bottleneck in modern web-applications (Leighton, 2009), but since our system will be highly localized (the majority of our users will probably be within walking distance of the server), transmission times will most likely not be a problem.

The time associated with establishing connections to send and receive data across the internet, however, could be a big problem. For example, a standard HTML request requires the client and server to establish a TCP connection, which has a three packet handshake. The best case scenario then is one round trip per message sent, which takes about 115ms (Sissel, 2010). Since our editor needs to have a low response time, establishing TCP connections all the time is not an ideal situation, and we will have to look for other approaches, such as SSE or WebSockets, where a channel is kept open until it is closed (Kaazing Corporation, 2014).

2.3.2 Scalability

Scalability has no generally accepted definition (Hill, 1990), but the basic notion is intuitive: how does growth (both in tasks and computing power) affect the system. There are two broad categories for improving performance when scaling a system: scaling out and scaling up (Michael, Moreira, Shiloach, & Wisniewski, 2007). Scaling up (also called vertical scaling) refers to improving the performance of a single node, while scaling out (also called horizontal scaling) refers to distributing the system by adding more nodes. In our system we are most concerned with how the system handles the number of users and the files sizes growing, and how we can ensure that the response time is still kept low even when the system is stressed.

The editor response time and scalability potential will be tested extensively in Chapter 5, and suggestions for improvement will be presented.

2.4 Responsive design

Responsive was born out of necessity with the advent of the smartphone. With so many different smartphone operating systems (iOS, Windows Phone, Android, Symbian, and more), companies could not afford to develop native apps for each different platform. Thus, a new design philosophy was born.

Responsive Web design is the approach that suggests that design and development should respond to the user's behavior and environment based on screen size, platform and orientation. (Knight, 2011)

In his article, "Responsive Web Design: Enriching the User Experience" (Gardner, 2011), Brett Gardner mentions that responsive design has three main elements: fluid layouts, flexible images and media queries. We will take a closer look at each of these three elements.

Fluid layouts

Fluid layouts are layouts that utilize a flexible grid system, which can adapt to different device sizes. This is the main cornerstone of responsive design. As an example, picture a website which has three news article previews in a row on the front page. With a traditional non-fluid layout, this layout will remain the same on a cellphone, rendering three boxes in a row. This will cause every news article to become very small, and the user will have to zoom in to see the articles. With a fluid layout, these boxes will be placed underneath each other in a column, instead of after each other in a row. How this looks is illustrated in Figure 2.10. You can clearly see that the design responds to the device's size.

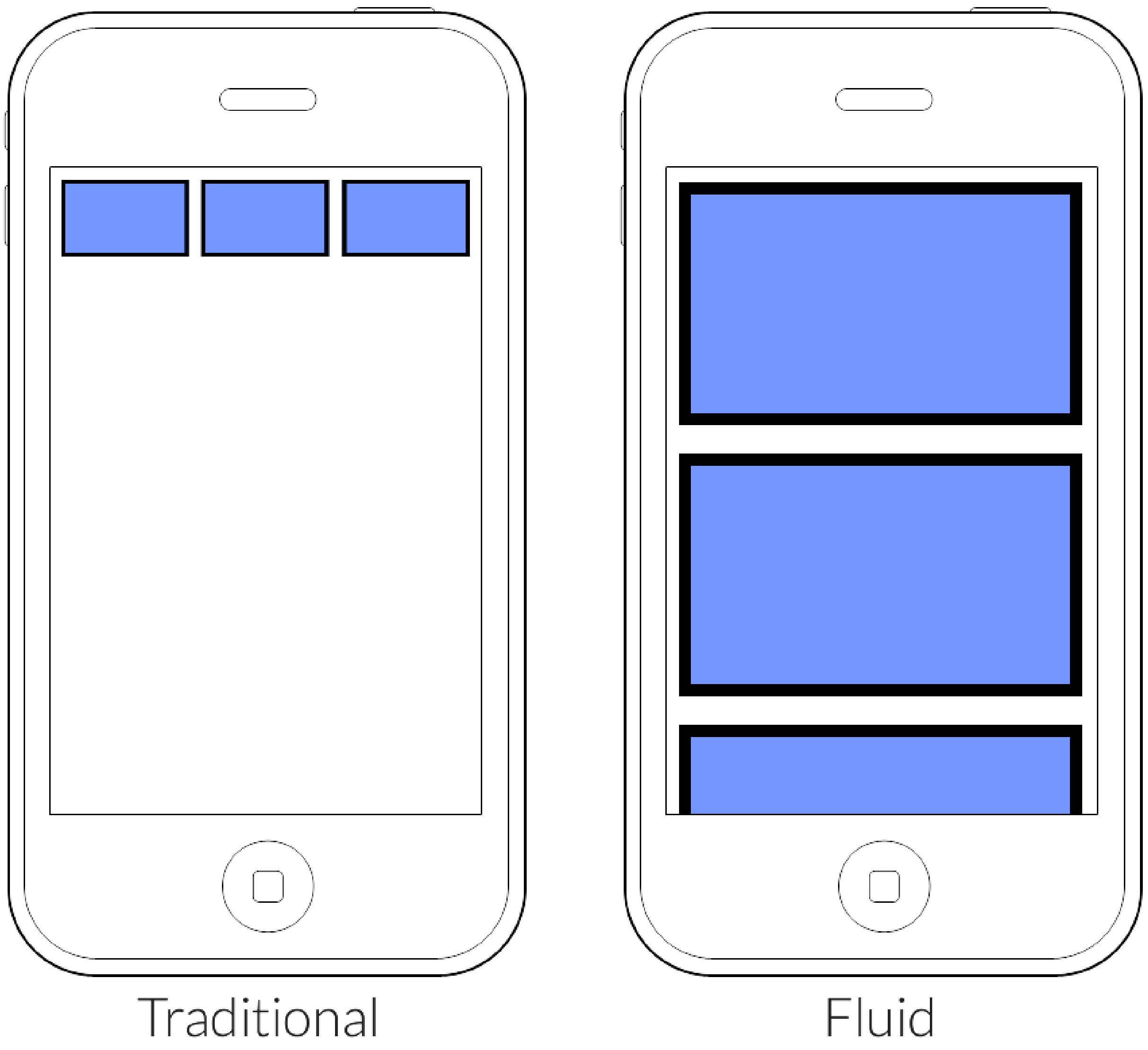


Figure 2.10: Traditional and fluid layout

Flexible images

Flexible images are images that adapt to the size of their container. This is typically achieved by setting the image width to the desired percentage of its container, but in some cases image-cropping is more suitable.

Media queries

Media queries are less general than the two other elements mentioned. With media queries, the designer can write specific behavior for the website based on the dimensions of the devices. A media query typically specifies what should happen below or above a certain limit. For example:

```
@media (max-width: 600px) {
  #left-menu {
    display: none;
  }
}
```

This code specifies that the element that has the ID `#left-menu` should be hidden for all devices with a width of less than 600px. Media queries are typically used to achieve things such as transforming traditional left-menus to dropdown-menus on mobile devices.

Testing responsive design

The only real way to test responsive design is by looking at it on different devices. Since this is not an option most people can afford, different services which emulate devices exist. One example of this is Responsinator[5], a website which emulates the six most popular devices (based on American user data), in both portrait and landscape mode. We will use this service.

2.5 Gamification

The background theory for gamification carries over from the pre-study. The most important theoretical elements were "Self-determination theory" (Ryan & Deci, 2000) and "Fogg's behavior model" (Fogg, 2009). This theory, along with three case studies, were used to determine which game elements would be best suited for an assignment system for programming challenges.

2.6 State of the art

There are a plethora of websites dedicated to teaching programming online, but most of them are very limited in what they offer in terms of editor-capabilities. The most comprehensive site with the best online editor seems to be Microsoft's "Try F#" website[6]. This website is powered by Microsoft's Monaco Editor, which pretty much gives full IDE-functionality, including instant code completion, error detection, imports, and more. Unfortunately, Monaco is a closed source project, and only used by Microsoft for their own programming languages, such as F#, C#, TypeScript, etc.

Since we are looking to implement a Java editor, we focus on what Java alternatives there are available today. We will first look at complete educational systems and, later, pure editors.

2.6.1 Educational systems

Since we are trying to create a system for teaching novice programmers about Java, we will first have a look at the most relevant competitors in this category.

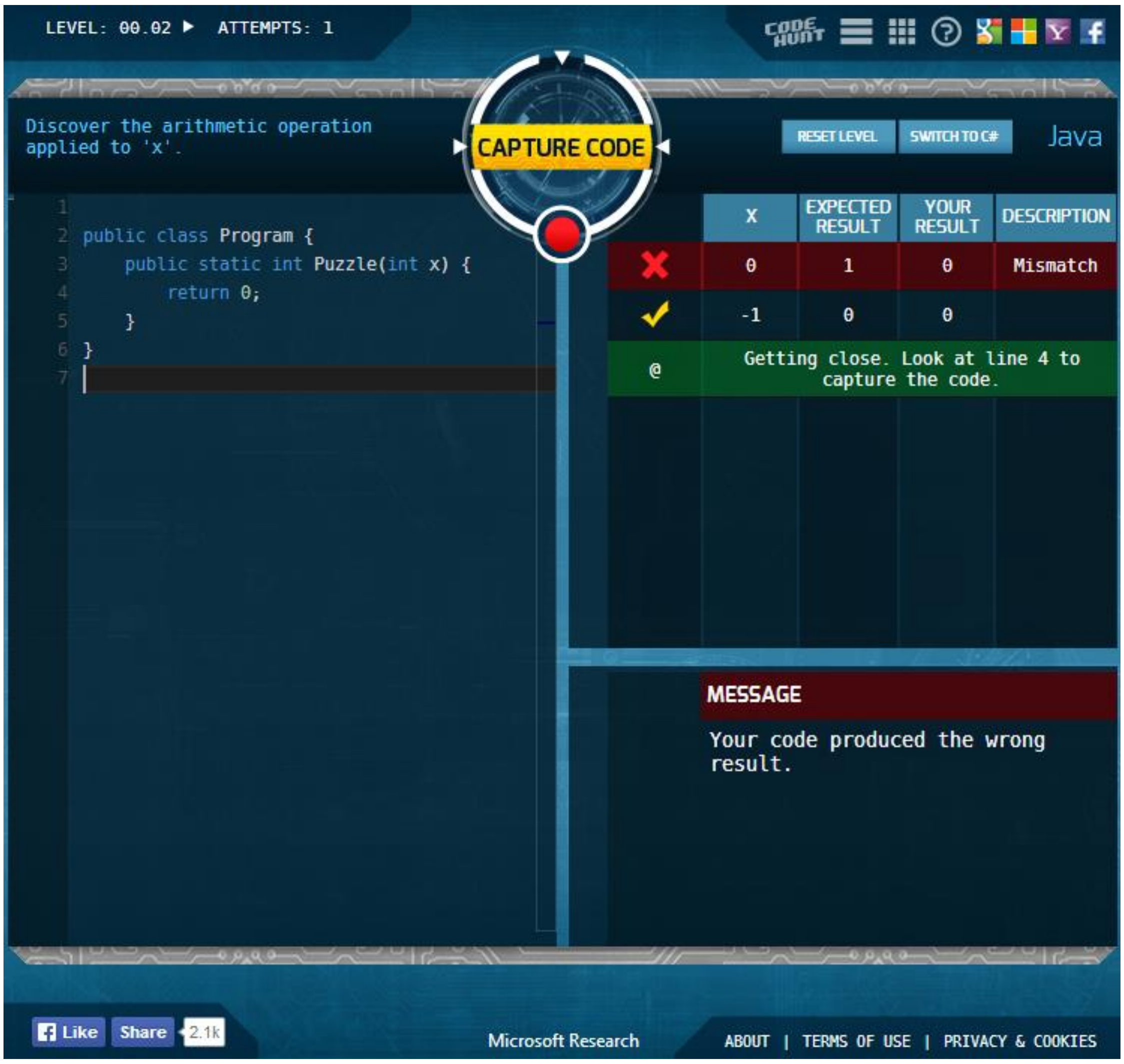
Code Hunt[7]

Code Hunt is another Microsoft project, aimed at teaching (presumably younger) people programming in a fun and visually overwhelming way. It supports both C# and Java.

[5] Available at: http://responsinator.com
[6] Available at: http://tryfsharp.org
[7] Available at: http://codehunt.com

Code Hunt is a game! The player, the code hunter, has to discover missing code fragments. The player wins points for each level won with extra bonus for elegant solutions. As players progresses the sectors, they learn about arithmetic operators, conditional statements, loops, strings, search algorithms and more. Code Hunt is a great tool to build or sharpen your algorithm skills. Starting from simple problems, Code Hunt provides fun for the most skilled coders. (Microsoft Corporation, 2014)

Code Hunt features an editor with syntax highlighting, bracket matching, block indentation, and other basic IDE-functionality, but lacks core features such as code completion and error-detection. The game centers on changing an existing piece of code ("discover missing code fragments") in order to make some tests pass. The tests are always displayed on the right hand side of the application and are run when the user clicks "capture code". Unfortunately, most of the problems seem to be about discovering what mathematical function to return. A screenshot of Code Hunt can be seen in Figure 2.10, where the challenge is to change the "0" on line 4 to "x+1". The game also features loud noises when code runs and tests pass or fail.



```
Discover the arithmetic operation
applied to 'x'.
```

```
1
2 public class Program {
3     public static int Puzzle(int x) {
4         return 0;
5     }
6 }
7
```

	X	EXPECTED RESULT	YOUR RESULT	DESCRIPTION
✗	0	1	0	Mismatch
✓	-1	0	0	
@	Getting close. Look at line 4 to capture the code.			

MESSAGE

Your code produced the wrong result.

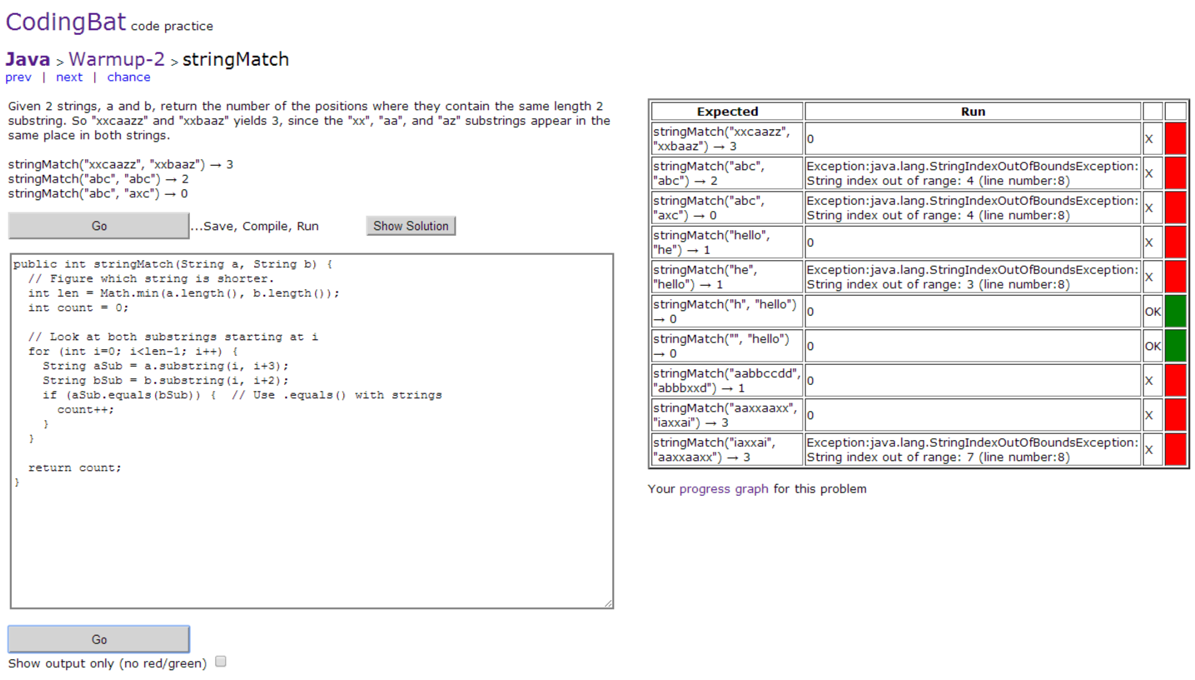
Figure 2.11: Code Hunt's editor

CodingBat[8]

CodingBat is a website created by Nick Parlante, a computer science lecturer at Stanford University. CodingBat aims to teach Java and Python to novice programmers.

CodingBat is a free site of live coding problems to build coding skill in Java, and now in Python (example problem), created by Nick Parlante who is computer science lecturer at Stanford. The coding problems give immediate feedback, so it's an opportunity to practice and solidify understanding of the concepts. (Parlante, 2014)

The website is aimed at beginner programmers learning the very basics of programming. CodingBat features an editor without any form of IDE-functionality at all, not even single line indentation is possible. It uses a simple HTML `<textarea>` element where the user can write code. It does, however, feature unit test results, which are displayed in a table. How this looks can be seen in Figure 2.12.



Given 2 strings, a and b, return the number of the positions where they contain the same length 2 substring. So "xxcaazz" and "xxbaaz" yields 3, since the "xx", "aa", and "az" substrings appear in the same place in both strings.

stringMatch("xxcaazz", "xxbaaz") → 3
stringMatch("abc", "abc") → 2
stringMatch("abc", "axc") → 0

```
public int stringMatch(String a, String b) {
  // Figure which string is shorter.
  int len = Math.min(a.length(), b.length());
  int count = 0;

  // Look at both substrings starting at i
  for (int i=0; i<len-1; i++) {
    String aSub = a.substring(i, i+3);
    String bSub = b.substring(i, i+2);
    if (aSub.equals(bSub)) {  // Use .equals() with strings
      count++;
    }
  }

  return count;
}
```

Expected	Run		
stringMatch("xxcaazz", "xxbaaz") → 3	0	X	🟥
stringMatch("abc", "abc") → 2	Exception:java.lang.StringIndexOutOfBoundsException: String index out of range: 4 (line number:8)	X	🟥
stringMatch("abc", "axc") → 0	Exception:java.lang.StringIndexOutOfBoundsException: String index out of range: 4 (line number:8)	X	🟥
stringMatch("hello", "he") → 1	0	X	🟥
stringMatch("he", "hello") → 1	Exception:java.lang.StringIndexOutOfBoundsException: String index out of range: 3 (line number:8)	X	🟥
stringMatch("h", "hello") → 0	0	OK	🟩
stringMatch("", "hello") → 0	0	OK	🟩
stringMatch("aabbccdd", "abbbxxd") → 1	0	X	🟥
stringMatch("aaxxaaxx", "iaxxai") → 3	0	X	🟥
stringMatch("iaxxai", "aaxxaaxx") → 3	Exception:java.lang.StringIndexOutOfBoundsException: String index out of range: 7 (line number:8)	X	🟥

Your progress graph for this problem

Figure 2.12: CodingBat's editor

Learnjavaonline.org[9]

LearnJavaOnline.org is a free interactive Java tutorial. Our vision is to teach Java in the browser using short and effective exercises. By running real Java code directly from the web browser, students are able

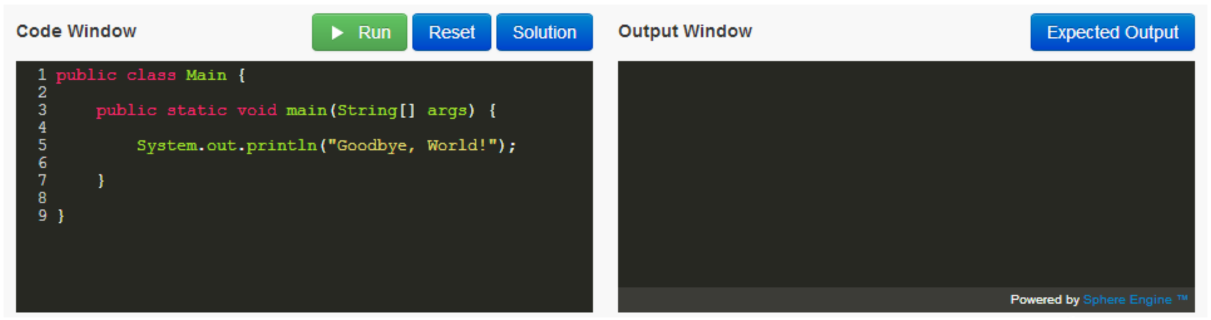
[8] Available at: http://codingbat.com
[9] Available at: http://learnjavaonline.org

to try out Java without installing it. This creates a more efficient learning process, because students focus on the important stuff - learning how to program. (LearnJavaOnline.org, 2014)

Learn Java Online (hereby LJO) is probably the system that is the most similar to ours. It offers different categories, such as "Variables and types" and "Conditionals", and each category page has code examples with explanations, and an integrated editor. LJO's editor features functionality such as syntax highlighting, bracket matching, block indentation, and other basic IDE-functionality, but just like Code Hunt it lacks core features such as code completion and error-detection. The editor can be seen in Figure 2.13. Running code in LJO's editor is incredibly slow, as it took an average of 6.1 seconds per run (ten runs) to execute the following snippet:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, slow!");
  }
}
```

LJO uses a third party service called Sphere Engine™ to run code, so the slowness is probably due to the communication between LJO's and Sphere Engine's servers (Sphere Research Labs, 2014). Needless to say, we will not be using this service.



```
public class Main {

    public static void main(String[] args) {

        System.out.println("Goodbye, World!");

    }

}
```

Figure 2.13: Learn Java Online's editor

2.6.2 Pure editors

When researching the complete educational solutions, it became clear that many systems used an editor plugin (Code Hunt uses Monaco, Learn Java Online uses CodeMirror). Even though none of the complete solutions fit our needs, perhaps there exists an editor which does?

There is an actively maintained Wikipedia entry which has an overview of available JavaScript editors. It features a very handy comparison matrix (Wikimedia Foundation, Inc, 2014). We will use this matrix to determine possible candidates to have a closer look at. We first do a screening based on some key criteria and then take a closer look at the candidates that make it through. The complete list consists of 16 editors, as can be seen in Table 2.1.

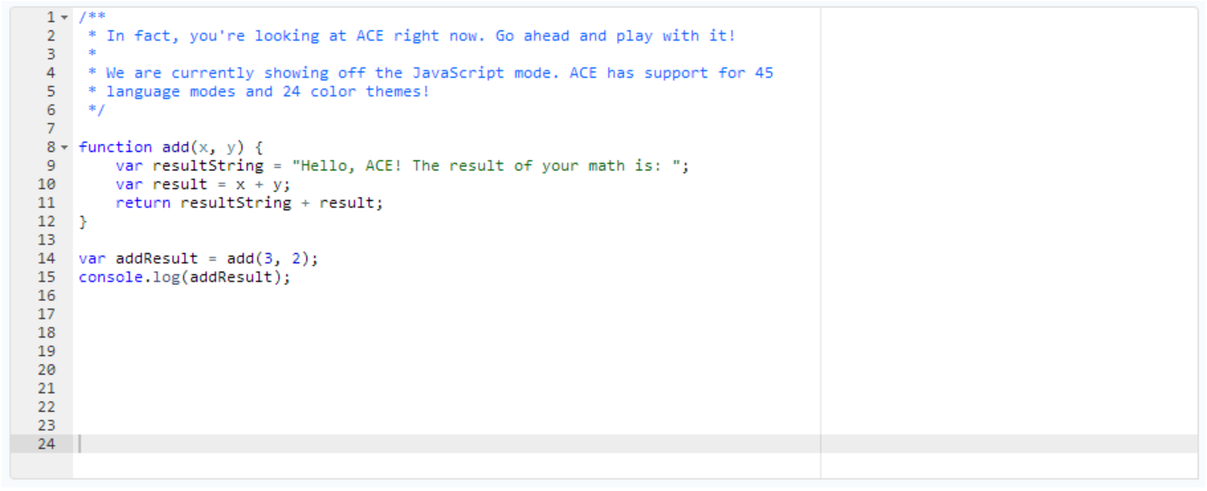
Editor	Cost	License	Open source	Activity
Ace	Free	BSD3	Yes	Yes
CodeMirror	Free	MIT	Yes	Yes
Orion	Free	BSD3/EPL	Yes	Yes
Codenvy	-	"Public Cloud"	No	Yes
Monaco	Used only by Microsoft	-	No	Yes
MDK	Depends on use	"Dual"	No	Yes
Markitup	Free	MIT/GPL	Yes	Some
Micro	Free	GPL	Yes	Some
LDT	Free	MIT/GPL	Yes	Some
Ymacs	Free	BSD	Yes	Some
Codepress	Free	LGPL	Yes	No
CodeTextArea	Free	BSD	Yes	No
EditArea	Free	LGPL	Yes	No
Helene	Free	GPL	Yes	No
9ne	Free	GPL	Yes	No
jsvi	Free	GPL	Yes	No

Table 2.1: JavaScript code-editor comparison

We need an editor which is being actively developed, open source and free. In addition we need it to have a software license which will allow us to use and modify it. This leaves us with just three editors: Ace (Ace, 2014), CodeMirror (CodeMirror, 2014) and Orion (The Eclipse Foundation, 2014).

Ace

Ace is an embeddable code editor written in JavaScript. It matches the features and performance of native editors such as Sublime, Vim and TextMate. It can be easily embedded in any web page and JavaScript application (Ace, 2014)

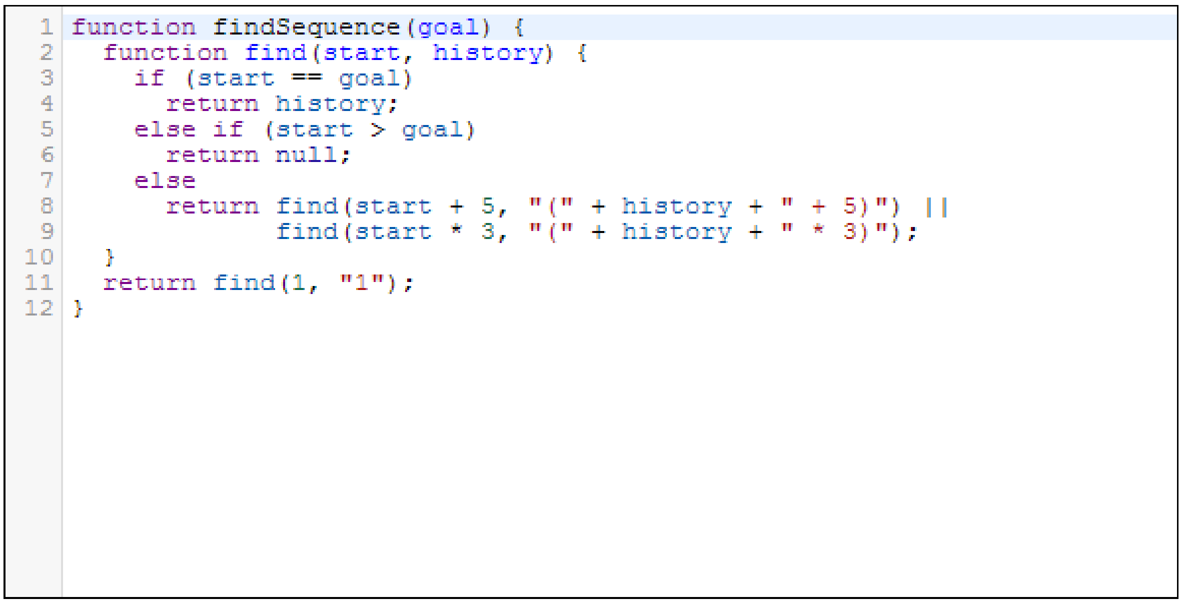


```
/**
 * In fact, you're looking at ACE right now. Go ahead and play with it!
 *
 * We are currently showing off the JavaScript mode. ACE has support for 45
 * language modes and 24 color themes!
 */

function add(x, y) {
    var resultString = "Hello, ACE! The result of your math is: ";
    var result = x + y;
    return resultString + result;
}

var addResult = add(3, 2);
console.log(addResult);
```

Figure 2.14: Ace code-editor

CodeMirror

CodeMirror is a versatile text editor implemented in JavaScript for the browser. It is specialized for editing code, and comes with a number of language modes and addons that implement more advanced editing functionality (CodeMirror, 2014)

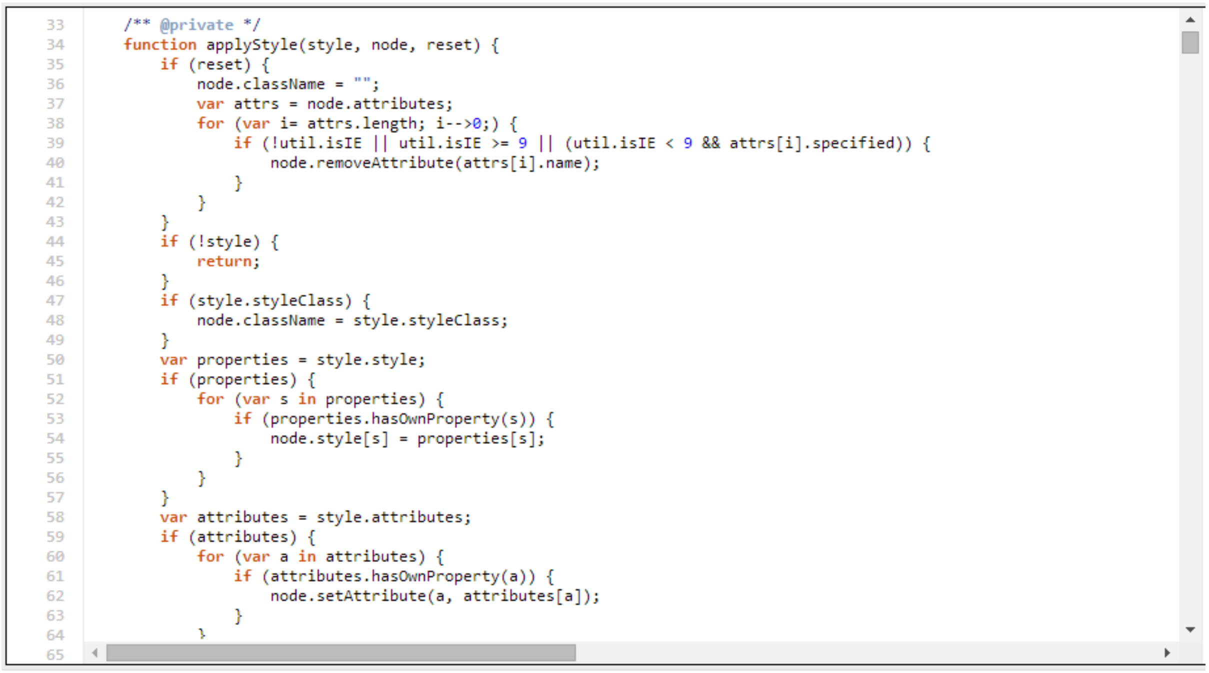


```
function findSequence(goal) {
  function find(start, history) {
    if (start == goal)
      return history;
    else if (start > goal)
      return null;
    else
      return find(start + 5, "(" + history + " + 5)") ||
             find(start * 3, "(" + history + " * 3)");
  }
  return find(1, "1");
}
```

Figure 2.15: CodeMirror editor

Orion

Develop your software where ever you go. It'll be there, in the cloud, right where you left it. Just login from a web browser on desktop, laptop, tablet, at an airport, on a bus or even in your office. Orion incorporates leading edge client technologies together to make an extensible tooling platform (The Eclipse Foundation, 2014)



```
/** @private */
function applyStyle(style, node, reset) {
    if (reset) {
        node.className = "";
        var attrs = node.attributes;
        for (var i= attrs.length; i-->0;) {
            if (!util.isIE || util.isIE >= 9 || (util.isIE < 9 && attrs[i].specified)) {
                node.removeAttribute(attrs[i].name);
            }
        }
    }
    if (!style) {
        return;
    }
    if (style.styleClass) {
        node.className = style.styleClass;
    }
    var properties = style.style;
    if (properties) {
        for (var s in properties) {
            if (properties.hasOwnProperty(s)) {
                node.style[s] = properties[s];
            }
        }
    }
    var attributes = style.attributes;
    if (attributes) {
        for (var a in attributes) {
            if (attributes.hasOwnProperty(a)) {
                node.setAttribute(a, attributes[a]);
            }
        }
```

Figure 2.16: Orion editor

Editor Comparison

As can be seen from the pictures, all of the editors look roughly the same. Any difference is best summed up in a feature comparison matrix, which can be found in Table 2.2. The matrix is inspired from a similar matrix from the previously mentioned Wikipedia article, but we have added key criteria pertaining to the goals of our thesis (the three rows in the beginning of the matrix with No's).

Feature	Ace	CodeMirror	Orion
Java syntax highlight	Yes	Yes	Yes
Java error highlight	No	No	No
Multiple class support	No	No	No
Maximizable	No	No	No
Code completion	Some	Some	Some
Tab support	Yes	Yes	Yes
Indent, new line keeps level	Yes	Yes	Yes
Indent, syntax	Yes	Yes	Yes
Indent, selected block	Yes	Yes	Yes
Bracket matching	Yes	Yes	Yes
Keyboard shortcuts	All common shortcuts and custom key bindings	Fully configurable	Yes
Line numbers	Yes	Yes	Yes
Search & replace	Regex supported	Yes	Yes
Visual styling	Fully theme-able	CSS-based themes	Yes
Undo/Redo	Yes	Yes	Yes
Non US charset support	Yes	Yes	Yes
Code folding	Yes	Yes	Some
Multiple cursors / Block selection	Yes	Yes	No
WebJar available	Yes	Yes	No
Indent guides	Yes	No	No
Code snippets	Yes	Through add-on	Some
Spell checking	Through add-on	No	No

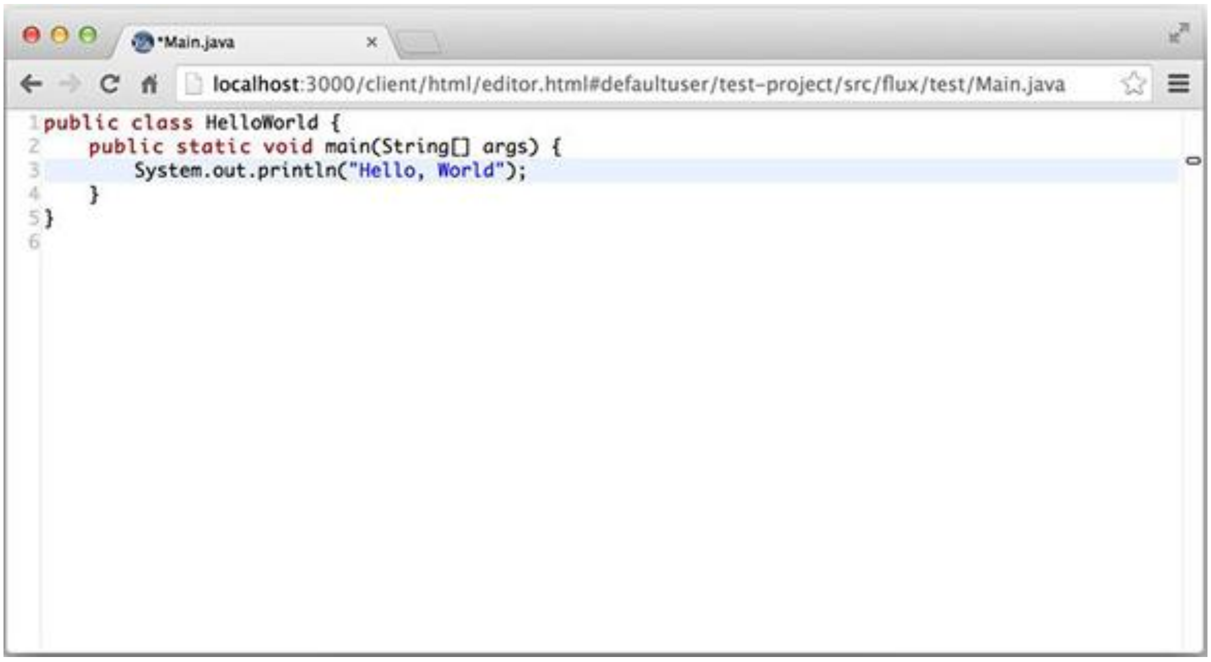
Table 2.2: Code-editor feature comparison

Looking at Table 2.2 the three candidates already support a lot of functionality, but neither of them support all of the goals for our thesis.

2.6.3 Comments and conclusion

Eclipse Flux

Midway through our thesis, our supervisor emailed the Eclipse Foundation about our work, and they responded saying they were working on something similar called Eclipse Flux.


*Main.java
localhost:3000/client/html/editor.html#defaultuser/test-project/src/flux/test/Main.java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

Figure 2.17: Eclipse Flux's editor

Eclipse Flux utilizes the Orion Editor (which we mentioned in section 2.6.2), and is connected to Eclipse via WebSockets (which we mentioned as an alternative in section 2.3.1). This allows for real-time collaboration between eclipse and a web-view of the same eclipse project (you can see the project path in the browser address bar). Unfortunately, flux appears to be dying. There has only been 35 commits in the past six months (see Figure 2.18).

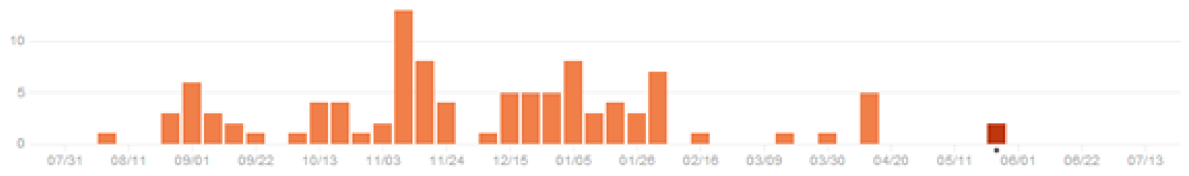


Figure 2.18: Eclipse flux activity diagram[10]

Conclusion

None of the current educational systems or editors support multiple classes, error checking or code completion, but most support syntax highlighting.

While researching online editors, we also found websites claiming to be complete online IDE's (cloud9, Koding, etc.). They all supported multiple classes and syntax highlighting, but neither of them supported error checking or code completion. Most of them used one of the three editors described in section 2.6.2.

[10] Source: https://github.com/spring-projects/flight627/graphs/commit-activity

3 - Re-engineering system processes

This chapter contains a description of the processes required to publish, solve and approve assignments in the old system and their new counterparts. These new processes act as constraints for the editor described in chapter 4. All the processes will be described textually, but the process for solving assignments will also be analyzed with GOMS and modeled with BPMN notation. We will misuse the BPMN syntax slightly, and have each swim lane represent either a physical or virtual environment. We will only analyze and model the assignment solving process because it is the process that has the biggest impact on student efficiency, and it is also the only process which utilizes our editor.

3.1 The current system processes

The current assignment system in TDT4100 has three main groups of actors: the students, the teaching assistants (TAs), and the course instructor. The assignment system consists of ten assignments, and each assignment has one or more problems that the students have to solve.

3.1.1 Publishing an assignment

To publish an assignment in the system, the course instructor starts by creating different problem pages on the course wiki. He does this by writing problem descriptions, uploading test code, and giving each problem a score. When all the problems are done, he collects the various wiki links (URLs) for the problems that make up the assignment and publishes them as an assignment to the "It's Learning" platform. This assignment is a text block with links, and it is up to the student to figure out how it is all connected.

3.1.2 Solving an assignment

First time setup

To configure a computer for Java-development, there are a number of things one has to do. Experienced programmers might think this is trivial, but to a first year student, who has very little experience with development, the task might seem daunting. First the student has to download the correct version of JDK from the Java website. Then he has to install it and set the system environment variables (PATH). Then he has to download the (correct) IDE-version, and, maybe, configure it, depending on which IDE he chose.

The process is different for each of the three major platforms (OS X, Windows, Linux), and a lot of time at the beginning of the semester is spent getting everything up and running.

BPMN model

The model (Figure 3.1) shows the process for first time setup, and all the different tasks the student has to deal with throughout the process. Each swim lane represents an environment. The tasks are broken down into smaller steps in the next section.

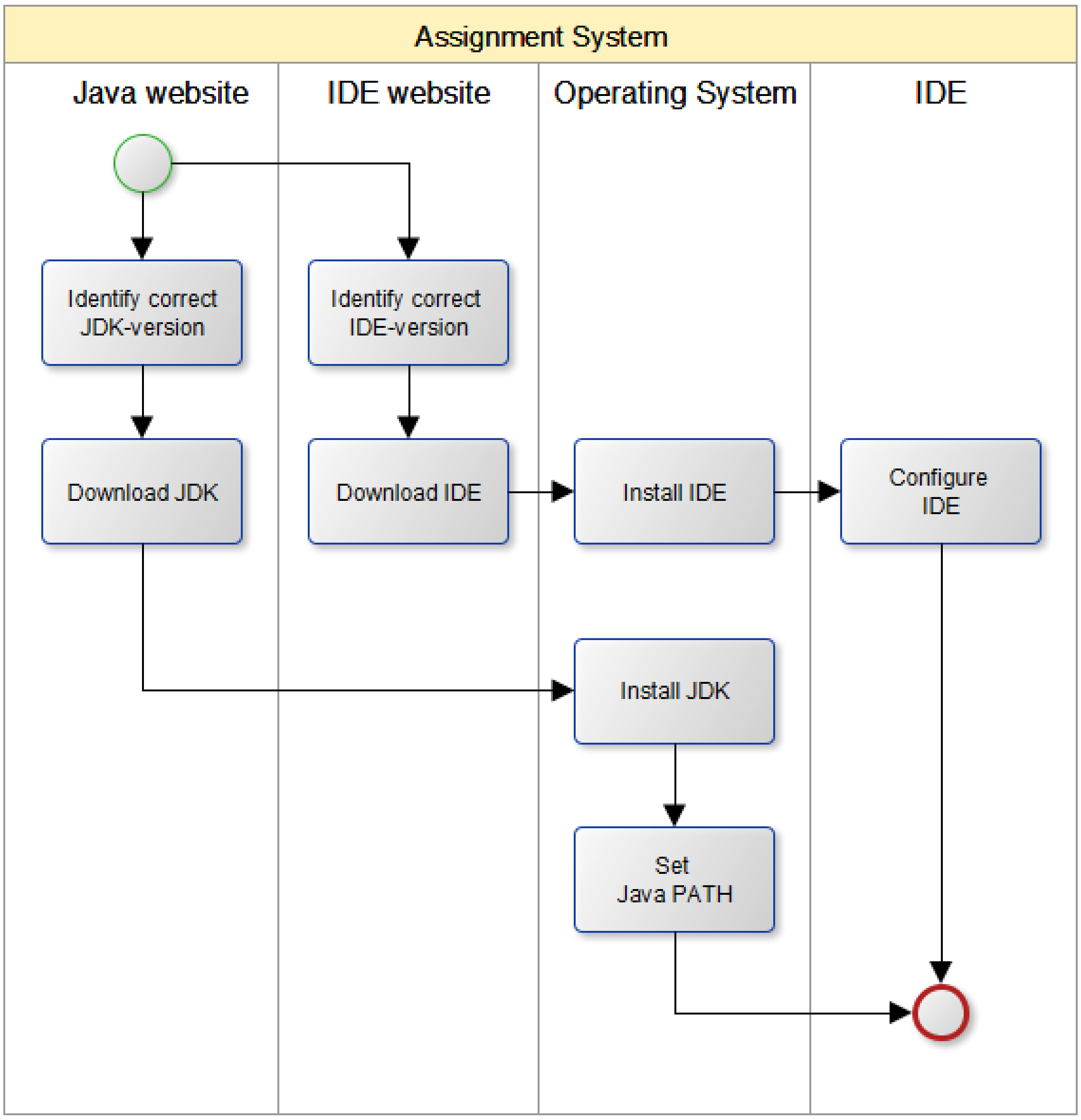


Figure 3.1: Solve assignment setup (current, BMPN)

Task analysis

Goal: Identify and download correct JDK-version

Primary method: Manual Setup
- Open browser
- Navigate to JDK webpage
- Determine correct JDK edition (not always trivial for first year students)
- Locate JDK download section
- Determine correct JDK edition version (not always trivial for first year students)
- Check license agreement
- Click download

Secondary method: Use Ninite[11]
- Open browser
- Navigate to Ninite webpage
- Check "JDK"
- Click download

Selection rules and comments

Ninite is not a well-known application, so most students will perform manual setup by default. Determining the correct JDK-version can be tricky for a new student. When googling "java jdk" or "download jdk" Oracle's download page is the top results. This page has a left menu with three similar options "Java SE", "Java EE" and "Java ME", which can be confusing. When clicking on one of them, you are taken to a product page with different tabs. You then have to find the download tab, which leads you to a list of the operating system specific versions of the JDK edition you selected. You then have to identify the version corresponding to your operating system.

None of this has anything to do with object oriented programming. It is purely operating system specific configuration, so the task should be avoided altogether.

Goal: Identify and download correct IDE

Primary method
- Open browser
- Navigate to appropriate IDE website
- Determine correct IDE version
- Click download

Comments

As with JDK-versions, determining the correct IDE edition and version can be hard for new students. As an example, the Eclipse download section features 13 editions in both 32-bit and 64-bit versions.

[11] https://ninite.com is a service which performs batch-install operations. It can be used to install JDK+IDE, among other things

Goal: Install JDK and set Path

Installing JDK is different for each operating system. Only the Windows approach will be described here, as Windows is still the dominant operating system.

Primary method

- Navigate to and run the downloaded JDK file
- Follow the installation Wizard
- Finish installation
- Navigate to "Environment variables" settings (not always trivial for first year students)
- Locate JDK installation directory
- Add JDK dir to the "Path" variable (not always trivial for first year students)
- Add new system variable "JAVA_HOME"
- Add JDK dir to the "JAVA_HOME" variable (not always trivial for first year students)

Comments

Many newer students have never needed to access the system variables before, thus the process can be confusing, especially on Windows 8 systems where advanced functionality is hidden by default. Teaching Assistants spend a lot of time on these task at the start of each semester, and this is operating system specific knowledge, so the task should be avoided altogether.

Goal: Install and configure IDE

For some IDEs this require that JDK is installed and that the correct path is set. Since this is not the case with Eclipse, we will show how it is done with IntelliJ IDEA.

Primary method

- Open IntelliJ IDEA
- Open the Project Structure dialog (e.g. Ctrl+Shift+Alt+S).
- In the left-hand pane, under Platform Settings, click SDKs.
- To add a new SDK, click add and select the desired SDK type.
- In the dialog that opens, select the SDK home directory and click OK.

Comments

This is tool-specific knowledge and not related to object oriented programming, so the task should be avoided altogether.

Conclusion

As we saw in all the methods, everything related to first time setup is specific to the user's operating system or IDE and has nothing to do with object oriented programming. There are many places for students to make mistakes (determining correct versions and configuring), and whatever knowledge the students obtain doing this is irrelevant to the curriculum and exam. If possible, first time setup should be completely avoided.

Weekly use

To solve an assignment in the system, the students first open “It’s Learning”, where they can find the assignment (some text and a collection of links). They then click the different problem links to visit the wiki where they can download the source code and test code to solve the problems in an IDE of their choice. After the files are downloaded, the students create a new project and import and configure the test files with JUnit.

Each problem has two types of points, “dekningsgradspoeng” and “omfangspoeng”. The students have to choose problems in such a manner that the total score is at least 100 points for both kinds of points. Students have complained that the current score system is confusing and cumbersome to use, as it requires them to manually calculate and keep track of how many points they have earned before they can deliver their assignment.

To deliver the assignment, students seek out a TA who reads their code and then decides how many points to give and approves the assignment.

BPMN model

The model (Figure 3.2) shows the current way to solve assignments on a weekly basis, and the different environments the student has to deal with throughout the process. Each swim lane represents a physical or virtual environment.

The models describes what we think is the optimal way of using the current system and navigating between the different environments, but this is not a process that is presented to the students at any point, they have to figure it out by themselves.

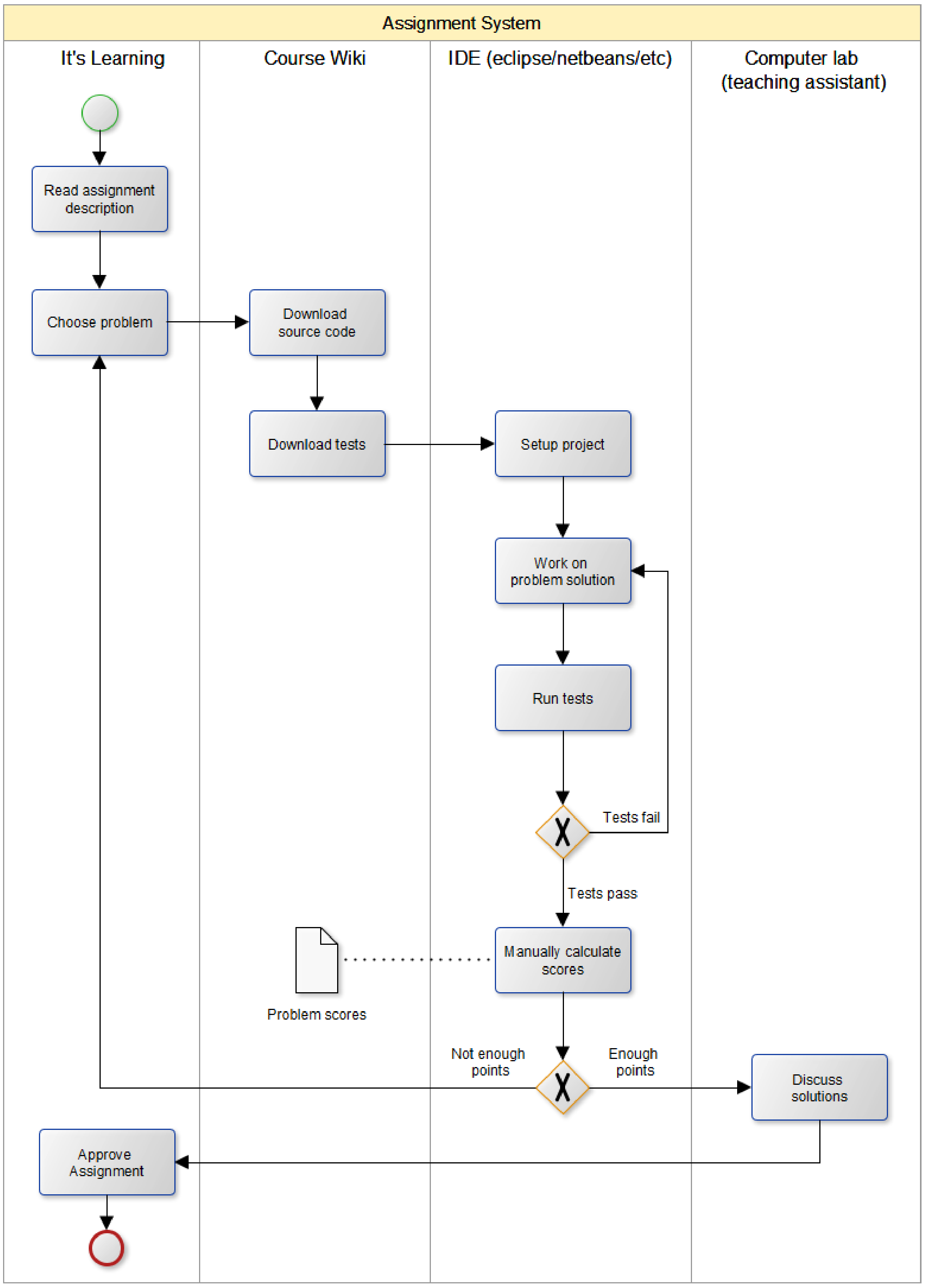


Figure 3.2: Solve assignment weekly (current, BPMN)

Task analysis

Goal: Read assignment description

Primary method
- Open web browser
- Navigate to Innsida
- Click "It's Learning" in the right hand pane
- Log in via Feide
- Find and click "TDT4100" in the course dropdown
- Click the assignment folder in the left hand pane
- Click the correct assignment
- Click assignment description file
- Read the assignment description

Comments

In order to just read the assignment description, student have to deal with both "Innsida" and "It's learning". It should be possible to go to "It's learning" directly, but neither of the authors were able to figure out how this could be done, so it is unlikely that new students will either.

Goal: Choose problem

In order to choose a problem, you first need to know your score. After you have assessed that you need more points, you can use one of the following methods to select a problem.

Primary method: Starting over
- Open web browser
- Navigate to Innsida
- Click "It's Learning" in the right hand pane
- Log in via Feide
- Find and click "TDT4100" in the course dropdown
- Click the assignment folder in the left hand pane
- Click the correct assignment
- Click assignment description file
- Read the assignment description
- Click problem (opens course wiki)
- Read problem description
- Solve current problem, or go back to assignment description to choose another problem

Primary method: Keeping a tab open
- Do not close the tab after you chose the last problem
- Click problem (opens course wiki)
- Read problem description
- Solve current problem, or go back to assignment description to choose another problem

Comments

If the students never close their tabs, then this goal is not too difficult to fulfill. However, since solving problems usually takes a few hours, students are likely to close their tabs. The methods require the students to deal with three different environments.

Goal: Download source code and test code

This task takes place after the students have chosen a problem.

Primary method: Save as
  For each test or source code file:
    - Right click file
    - Select save as
    - Select directory
    - Click save

Secondary method: Copy paste
  For each test or source code file:
    - Click file
    - Select all text and copy
    - Create new file on computer
    - Open file
    - Paste
    - Save file with appropriate extension

Comments

Most students will probably use the first method, but as tablet use is becoming more and more widespread, less and less people are familiar with right clicking. The copy/paste method can also be used to paste directly into the IDE, which can actually make it more effective than the first one.

Goal: Setup project

This task is different for all IDEs. We will describe the process as it is in Eclipse, as this is the recommended IDE for the course. This tasks also requires the students to have obtained any necessary test and source code files.

Primary method
    - Open Eclipse
    - Click the "File" menu item
    - Select "New > Java Project"
    - Write a name
    - Click "Finish"
    - Right click the "src" folder
    - Select "New > Package"
    - Located and open the downloaded test-file
    - Copy package name from test file to the eclipse-wizard
    - Click "Finish"
    - Copy downloaded test-file to the newly created package
    - Install JExercise (or delete JExercise annotations)
    - Right click the newly created package
    - Select "New > Class"
    - Write name (find either at course wiki or in test file)
    - Click "Finish"

Comments

Project setup is biggest hurdle for students to overcome when solving an assignment. Classes and tests have to be placed in correct packages for everything to work, and you either have to install JExercise (which is a task all in itself) or edit the tests files to remove the JExercise annotations. Teaching assistants spend a lot of time on project setup, and while it can be useful to know, it is not a part of the curriculum, and ideally it should be avoided completely when designing the new processes.

Goal: Work on problem solution

```
Primary method
    - Read problem description (at course wiki)
    - Write code (in eclipse)
    - Run tests (in eclipse)
    - Repeat
```

Comments

This is highly individual, of course, but in most cases it probably looks something like described.

Goal: Calculate scores

```
Method 1: Starting over
    - Open web browser
    - Navigate to Innsida
    - Click "It's Learning" in the right hand pane
    - Log in via Feide
    - Find and click "TDT4100" in the course dropdown
    - Click the assignment folder in the left hand pane
    - Click the correct assignment
    - Click assignment description file
        For each solved problem:
            - Click problem (opens course wiki)
            - Check problem scores

Method 2: Writing down
    - Create a local score file
            For each problem
                - When choosing a problem, write down score in local score file
                - If the problem was solved, keep the score
                - If you give up and choose another problem, delete the score
```

Comments

While method 2 is clearly more efficient, it is still needlessly cumbersome and introduces another environment: a score keeping file. Keeping track of what you have completed should not be up to the student. This also becomes a problem when you want to have your assignment approved, and the Teaching Assistant has to double check that everything is okay.

Goal: Discuss assignment and have it approved

Primary method
- Find teaching assistant in computer lab
- Show and discuss your solution with teaching assistant
- Calculate score
- Teaching Assistant then has to:
    - Open web browser
    - Navigate to Innsida
    - Click "It's Learning" in the right hand pane
    - Log in via Feide
    - Find and click "TDT4100" in the course dropdown
    - Click the assignment folder in the left hand pane
    - Click the correct assignment
    - Score the student

Comments

To be fair, the Teaching Assistant probably never closes "It's learning" during his or her two hour approving-session, but it is still needless manual labor.

Conclusion

Some of the tasks in the current system are very complex, often requiring the students to interact with three different environments to complete them. The tasks that are especially complex are "Setup Project" and "Calculate Score", which are also the two least useful tasks in regards to learning outcome and the curriculum. When designing the new system, these two tasks should be ideally be removed altogether. Tasks such as "Select problem" and "Read assignment description" naturally have to be a part of the new system processes, but the amount of environments needed to complete these tasks should be minimized.

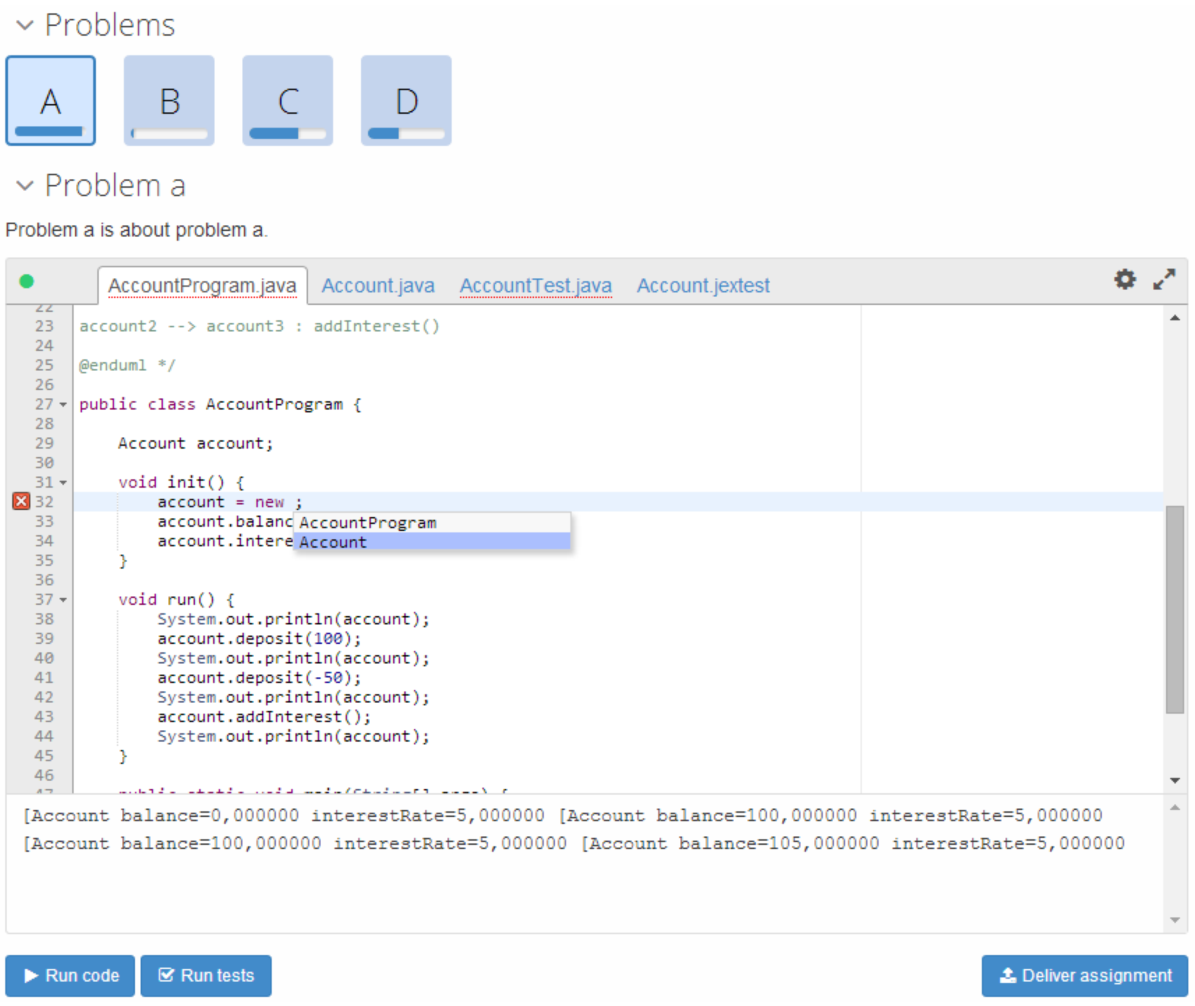
3.1.3 Approving an assignment

To approve an assignment, the TA reads through the student's code manually, while asking questions and offering comments. If the TA decides that the student should pass, he finds the student on "It's learning" and checks an "Approved" box for the assignment in question.

3.2 New system processes

This section describes the processes for publishing, solving and approving assignments in our new assignment system. The main idea behind the new system processes is to maximize student efficiency and create a more seamless and fun experience. To maximize efficiency we move the system to the web, making it platform independent and eliminating first time setup. To make the system fun we utilize gamification and immediate feedback to motivate the students.

A screenshot of the finished editor (Figure 3.3) is included to help the reader understand how the new system will behave in use. Skeleton classes and tests are preloaded into the editor.



Problem a is about problem a.

```
account2 --> account3 : addInterest()

@enduml */

public class AccountProgram {

    Account account;

    void init() {
        account = new ;
        account.balanc AccountProgram
        account.intere Account
    }

    void run() {
        System.out.println(account);
        account.deposit(100);
        System.out.println(account);
        account.deposit(-50);
        System.out.println(account);
        account.addInterest();
        System.out.println(account);
    }

```

```
[Account balance=0,000000 interestRate=5,000000 [Account balance=100,000000 interestRate=5,000000
[Account balance=100,000000 interestRate=5,000000 [Account balance=105,000000 interestRate=5,000000
```

Figure 3.3: The system's editor

3.2.1 Publishing an assignment

The course instructor creates an assignment, adds problems to the assignment, and then adds tests to the problems, all in the same system.

3.2.2 Solving an assignment

First time setup

The system runs Eclipse instances on a webserver and requires no setup on the students' part, except choosing a username and a password (these could also be assigned by the system).

This is a good solution, since everything related to first time setup was either operating system specific or tool (IDE) specific, as we saw in the GOMS analysis and BPMN model of section 3.1.2.

Weekly use

This section details the process for the solving of assignments in the assignment system. When a new assignment is available, the student can open the assignment in the web portal.

From here he can choose which problems he wants to solve, and write his solution directly in the browser-window. The student can test his code by clicking a test-button in the same browser window. If all tests pass, the problem is solved and the student picks a new problem to solve. When the student has enough points, the assignment is automatically approved. The student's score is automatically displayed every time the student runs the tests, and is calculated based on the amounts of tests that pass.

This is a good solution, as it eliminates all the excess environments the student had to deal with before, in addition to greatly simplifying the weekly setup and score calculation tasks, as described in section 3.1.2.

BPMN model

The model (Figure 3.4.) shows the process for weekly use, and the different environments the student has to deal with throughout the process. Each task is broken down into different steps in the next section.

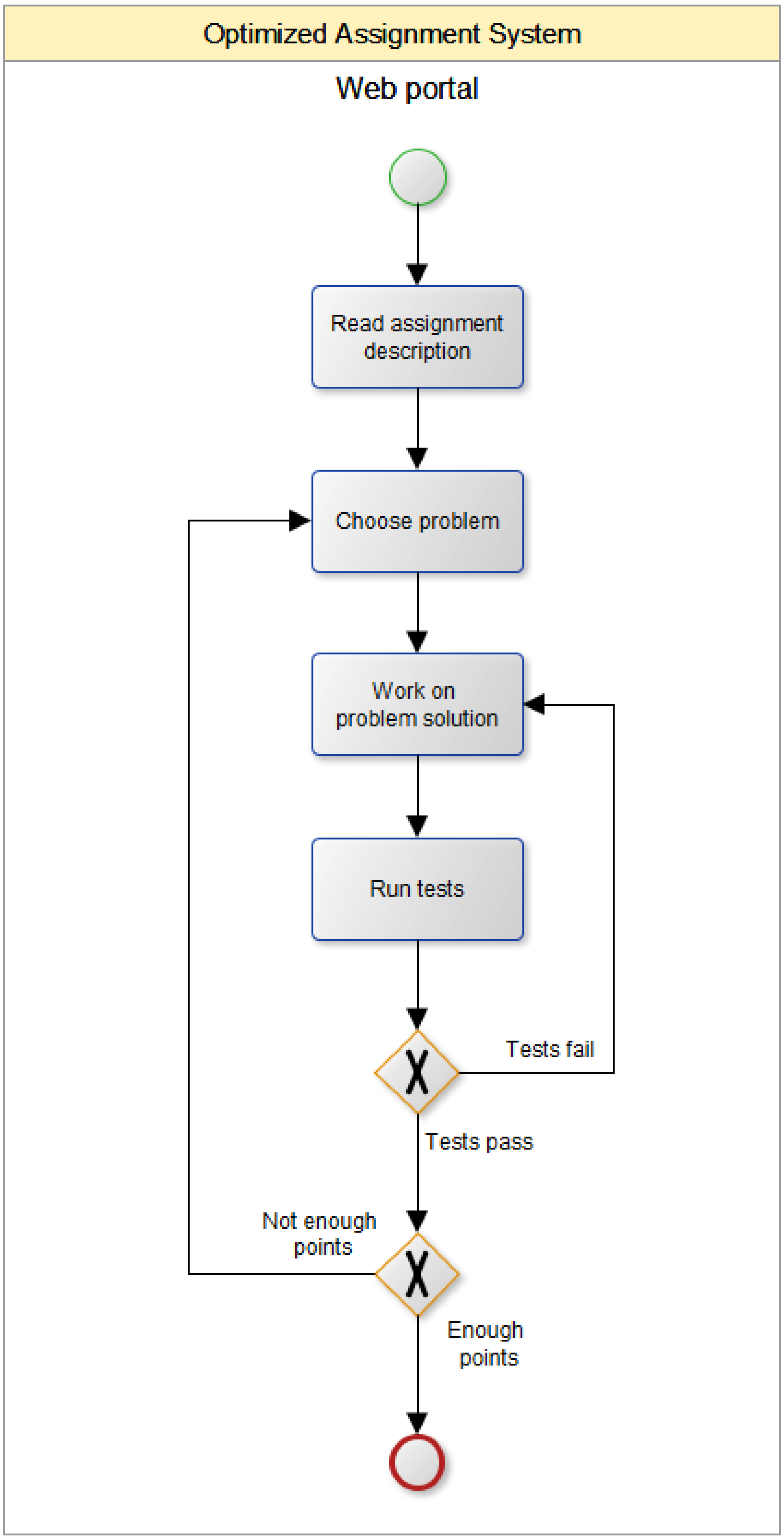


Figure 3.4: Solve assignment weekly (new, BPMN)

Task analysis

Goal: Read assignment description

Method 1

- Open web browser
- Navigate to system webpage
- Select assignment
- Read description

Comments

All assignments are immediately available on the welcome page, and the assignment description is available when you select an assignment.

Goal: Choose problem

- Hover over a problem to see its score
- Compare to score total displayed in right hand column
- Click problem

Comments

The need for manually calculating score has been removed, and all problem score are available from the assignment page (in the old system you had to visit the course wiki and scroll to the bottom of the page to find the problem score).

Goal: Work on problem solution

Primary method

- Read problem description (problem page in system)
- Write code (problem page in system)
- Run tests (problem page in system)
- Repeat

Comments

Since the editor is embedded in the problem page, the user never has to leave the programming environment to check the problem description.

Conclusion

The new system has successfully reduced the number of environments to a minimum (just one), and the new tasks are fewer and simpler than before. Everything related to setup and manual calculations has been removed completely, and the student never has to leave the programming environment. This ensures that the students can focus on actually writing code, and not configuring their computers, their tools or their projects, and that they do not have to navigate around different environments in order to fetch files or calculate scores.

3.2.3 Approving an assignment

If a student is able to collect the required amount of points for an assignment, the assignment is automatically approved by the assignment system, as shown in Figure 3.4. If the student is unable to solve an assignment, the system will set the assignment as not approved. The student can then request help or a manual override from a teaching assistant. The teaching assistant then has to decide if the assignment should be approved or not, and how many points should be awarded.

3.3 Conclusion

The process for solving the first assignment (first time setup + weekly setup) in the current system requires the student to deal with seven or eight different environments: The Java website, the IDE website, the operating system, "It's Learning", the course wiki, the IDE, the computer lab and an optional file for keeping scores.

The student has to go through 18 distinct tasks of varying difficulty. The task called "Setup Project" in Figure 3.2 is fairly complicated task with 16 steps, and requires the student to create a project with the appropriate packages and setting up JUnit to run the tests, something a lot of students ask their teaching assistant to help with.

The 18 tasks consist of a total of 74 steps if the student uses the optimal methods, and 94 if the student uses the sub-optimal methods. This is assuming the first assignment has only one problem to solve, and that the student does everything correctly the first time. For each additional problem, the optimal process increases with 12 steps, and the sub-optimal process increases with 29 steps.

In the new system, the number of environments is reduced to one: the system's website. The student can visit the computer lab if he requires assistance from the teaching assistants, but it is no longer necessary to have an assignment approved in person. The number of tasks needed to solve the first assignment is reduced from 18 to 4, as our system requires literally no setup on the students' part, neither first time nor weekly. The number of steps in the 4 tasks is reduced to just 11 and increases with 7 steps for each additional problem. As we can see, almost every step is associated with selecting and solving problems, and very few steps are wasted elsewhere.

In the pre-study it was discussed which would make for a better solution; a web based system or an IDE-plugin. As one of the goals of this thesis is to increase student efficiency, we decided to go with a web-based version, as the plugin version would still require the computer to be configured for Java development and an IDE-plugin would have to be installed on top of that. A web based system clearly enables the most efficient design, as all setup is taken care of on the server side, and the student can start programming right away.

4 - Editor architecture

This chapter contains the description and evaluation of different architectures which can be used to realize the web-based code-editor for the assignment system.

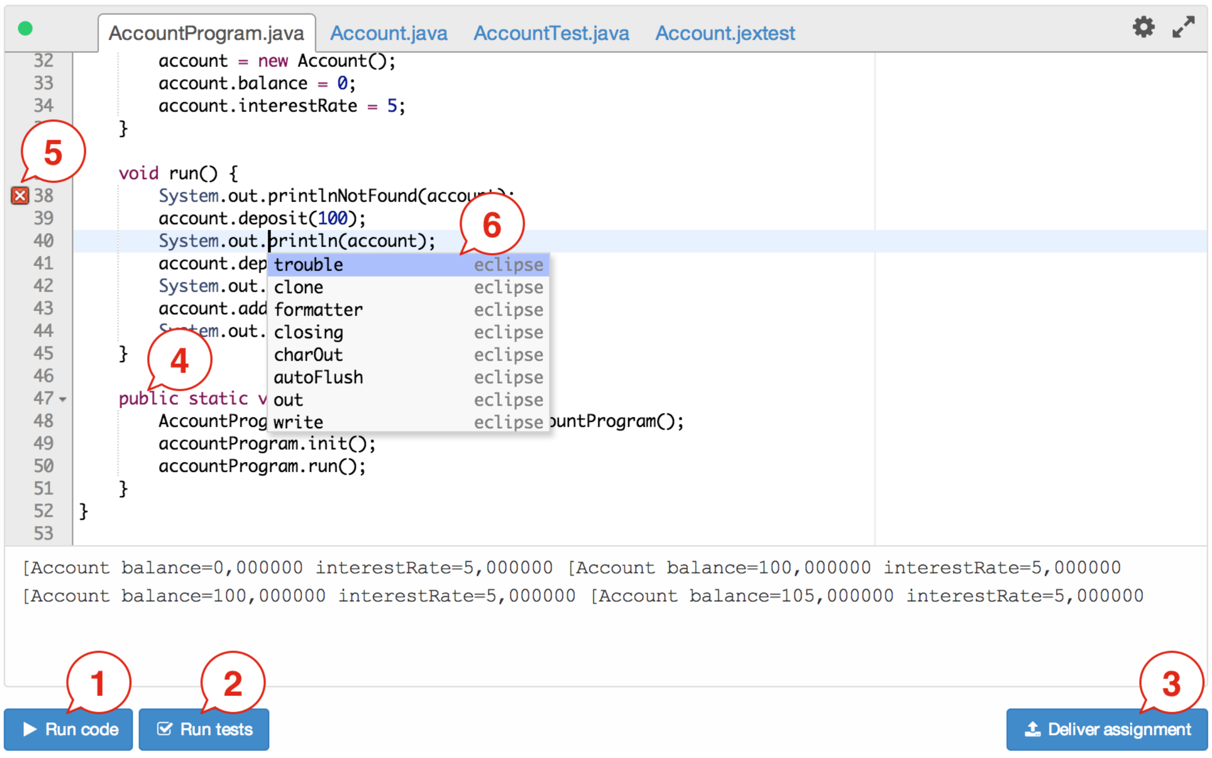
4.1 Supported functionality

The web-based code-editor should support the following functionality:

1. Running code and reporting its output
2. Running tests and reporting the test results
3. Delivering an assignment
4. Syntax highlighting
5. Live error checking
6. Code completion

This list is a summary of the requirements listed in section 8.2.2.

A screenshot of the editor is shown in Figure 4.1 (annotations are displaying the relevant functionality). More in-depth information about the supported functionality can be found in section 2.2.



```
        account = new Account();
        account.balance = 0;
        account.interestRate = 5;
    }

    void run() {
        System.out.printlnNotFound(accou
        account.deposit(100);
        System.out.println(account);
        account.dep
        System.out.
        account.add
        System.out.
    }

    public static v
        AccountProg                        untProgram();
        accountProgram.init();
        accountProgram.run();
    }
}
```

```
[Account balance=0,000000 interestRate=5,000000 [Account balance=100,000000 interestRate=5,000000
[Account balance=100,000000 interestRate=5,000000 [Account balance=105,000000 interestRate=5,000000
```

Figure 4.1: The supported functionality in the editor

4.2 The evolution of the architecture

In section 4.1 we looked at what functionality we want the editor to support. In this section we will look mainly at the functionality for running code, running tests, live error checking and code completion. We are omitting the "Syntax highlighting" functionality because it is solved on the client side, and we are omitting the "Delivering assignments" functionality because it only requires us to set a flag in the database.

To explain the evolution of the architecture we will start at the highest level and delve deeper into the details as we go.

4.2.1 High-level architecture

In section 2.2.4 we looked at how existing IDEs are architected. We will create the architecture of our editor by expanding on this knowledge.

A common way to architect web applications is to use a 2-tier, client/server architecture. This architecture is shown in Figure 4.2.

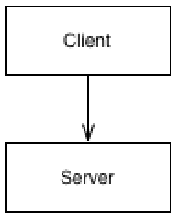


Figure 4.2: 2-tier architecture

The server part of this architecture consist of the business logic for our web application. The server might comprise a database system (in this case the architecture is usually called a 3-tier architecture) or even multiple nodes. The main point is that the web browser (client) is communicating with a central web server (server).

Another way of looking at the relationship between the client and server is looking at how the client uses a set of interfaces that the server exposes, shown in Figure 4.3.

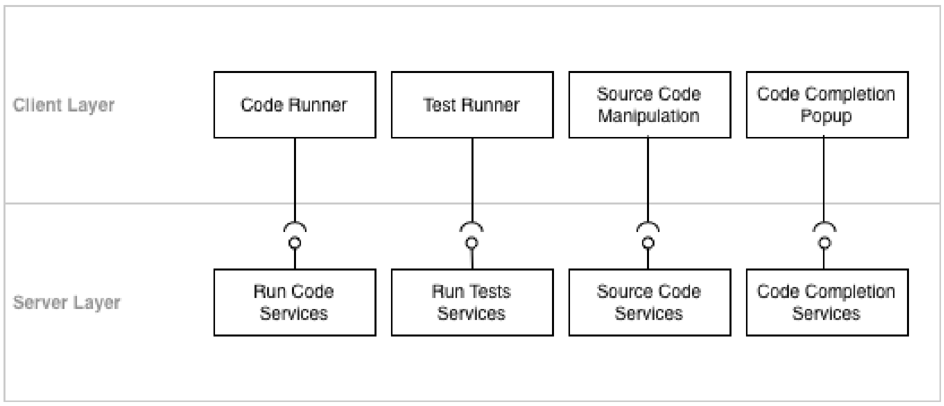


Figure 4.3: The interfaces that the server expose

Figure 4.3 should be familiar, as it is almost identical to Figure 2.8. The reason is simple: We want to reuse the common pattern where GUI components mirror core components. The actual Java interfaces for the server are shown in Figure 4.4. We will not supply the actual Java interfaces for the rest of this chapter, as they will look very similar to this.

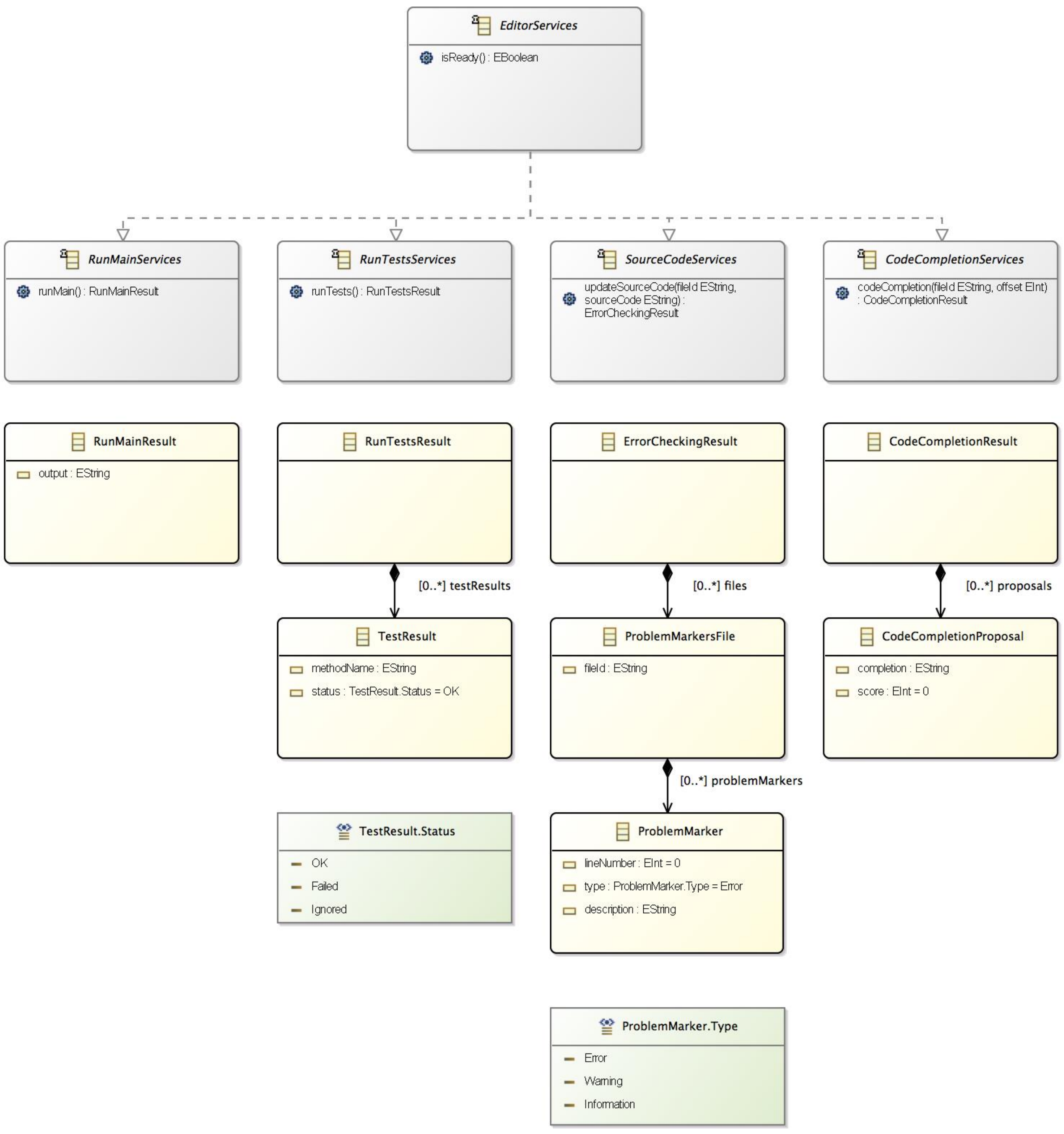


Figure 4.4: The Java interfaces for the communication between the client and the server

4.2.2 Making the web-server explicit

We have already stated that we are going to create the editor using web technologies. This means that we need to have a web server in our architecture. Using Figure 4.2 as the starting point, we can make the web-server explicit as shown in Figure 4.5.

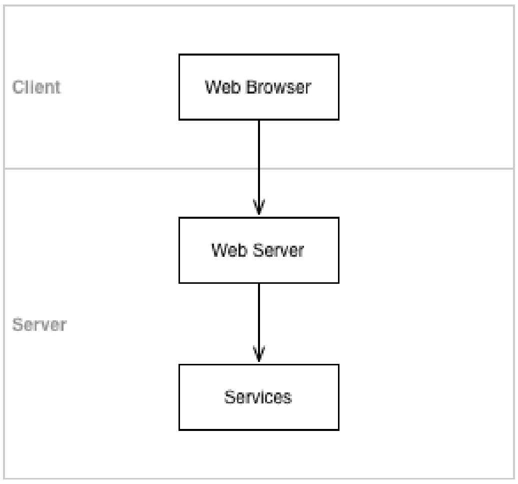


Figure 4.5: The architecture including the web server

This architecture will require a new set of interfaces, shown in Figure 4.6. The way that an interface wraps another interface is known as the adapter design pattern (Data & Object Factory, LLC, 2014).

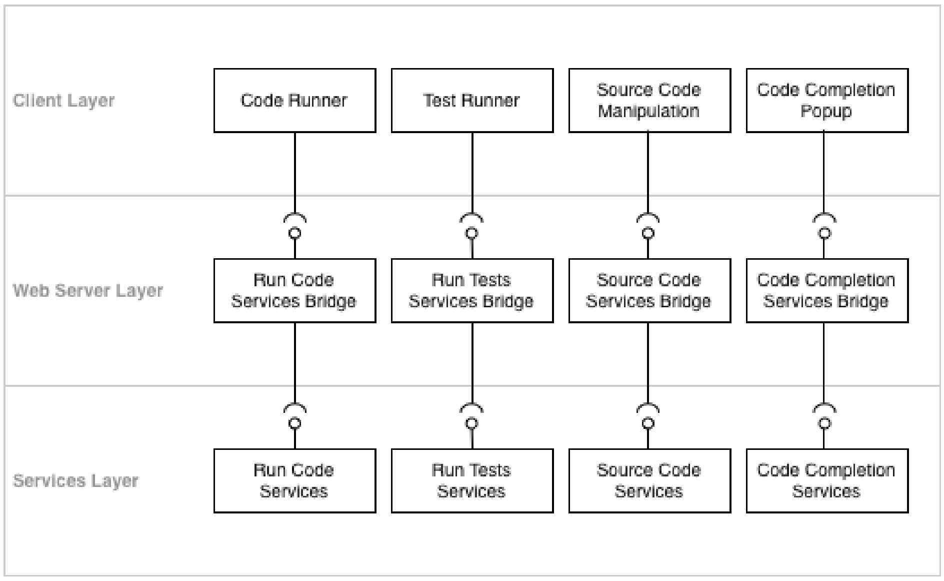


Figure 4.6: The interfaces that the web server and the services expose

In this architecture, the web server layer should be a thin layer on top of the services layer. The actual business logic resides in the services layer. One reason for keeping the web server layer small is to not lock the architecture to a specific web framework (we used Play Framework, as explained in section 9.2).

The web server will handle incoming requests (either by HTTP or through WebSocket) and translate these into the relevant Java method calls in the services layer. The only logic that the web server has to perform is to map a specific request to a corresponding call to the services layer. When calling the services layer, the web server will forward the current client's ID (thus the client must have previously logged in). If the request is a HTTP request, the web server will respond with a rendered HTML template, while if the request is a message through WebSocket, the web server will respond with JSON data.

In the case of the editor, all communication is handled through WebSocket (explained in further details in section 4.3).

4.2.3 Adding core functionality

In order for the editor to be functional, it must at least be able to run and test code. There are multiple ways to add this support:

- Sun JDK's Java-compiler
- Eclipse ECJ (Eclipse Compiler for Java)
- Eclipse JDT (Java Development Tools)
- External services, e.g. Sphere Engine from Sphere Research Labs (Sphere Research Labs, 2014)

Each of these options would suffice to run and test code. However, we want to support all the functionality listed in section 4.1. We could either try to find libraries that give us this functionality or implement it ourselves. It turns out that the Eclipse JDT exposes much of the internal functionality of Eclipse, like live syntax checking and code completion.

Based on the functionality that we want to support and given the functionality available in the Eclipse JDT, it is clear that we should build the system around Eclipse and the Eclipse JDT. To be able to use the Eclipse JDT we must create a plugin for Eclipse. The way a plugin works is that it hooks into a running Eclipse process. Usually, the plugin is activated from the GUI of Eclipse. In our case we need to run an Eclipse instance in the background and expose its functionality to the rest of the system and, ultimately, to the client.

In addition to the core functionality we also need a data storage (implemented as EMF, as explained in section 9.1) for saving and loading data. The expanded architecture is shown in Figure 4.7.

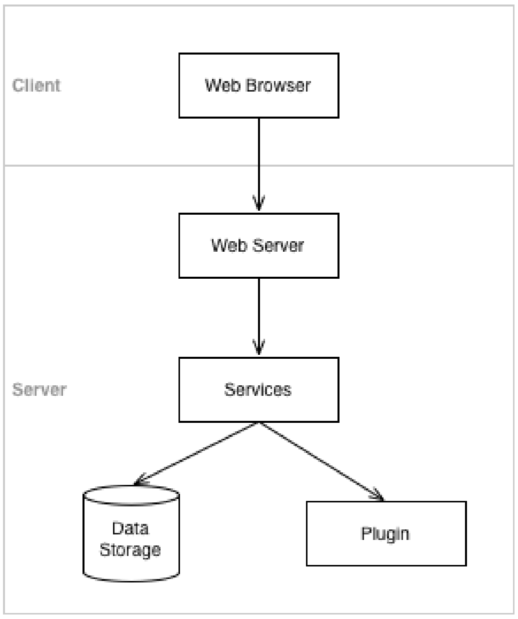


Figure 4.7: The architecture including the data storage and the Eclipse plugin

This architecture will require a new set of interfaces, shown in Figure 4.8.

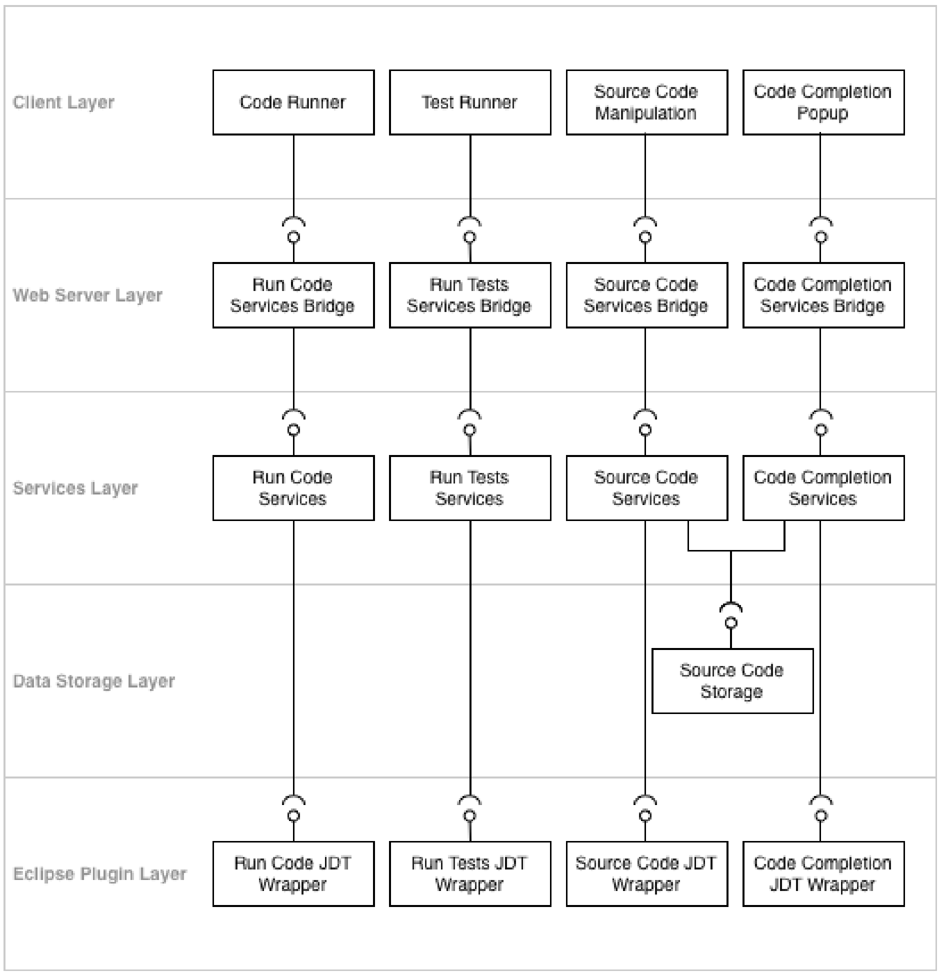


Figure 4.8: The interfaces including the Eclipse Plugin Layer

Figure 4.9 illustrates how the data flows through the interfaces, from the web browser to the Eclipse plugin and back again. In this example, we show what happens when the user modifies the source code in the browser.

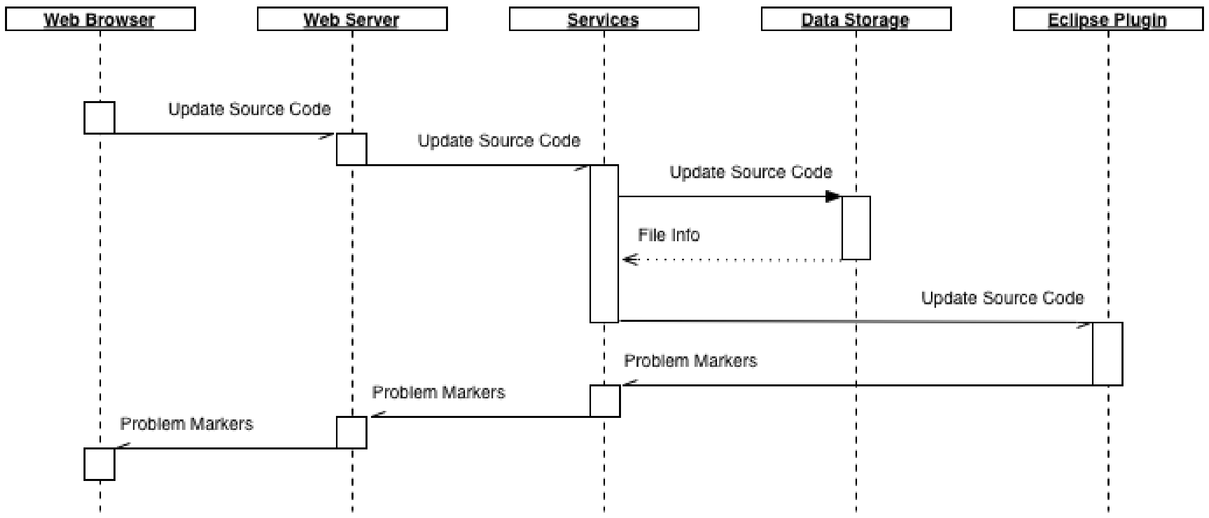


Figure 4.9: An example of how the data flows through the interfaces

The small rectangles (showing the processing time of each layer) emphasize that the processes are run asynchronously. This is important to not freeze the user interface for the user. By allowing the system to work asynchronously, we can potentially move some parts of the system to different nodes. We expect that the Eclipse plugin will become the bottleneck in the system (as compiling and running code are expensive operations). To alleviate this bottleneck we can distribute the load across multiple nodes. How we achieved the asynchronous behavior is described in more details in section 4.3.

Figure 4.9 is also showing how the data from the user is sent to the data storage to be persisted to disk. Even if the user leaves the system, the data will still be available the next time the user enters the system.

4.2.4 Alternative architecture

Another architecture that we considered was to move the plugin layer below the data storage, meaning that the services layer communicates with the plugin layer through the data storage. The alternative architecture is shown in Figure 4.10.

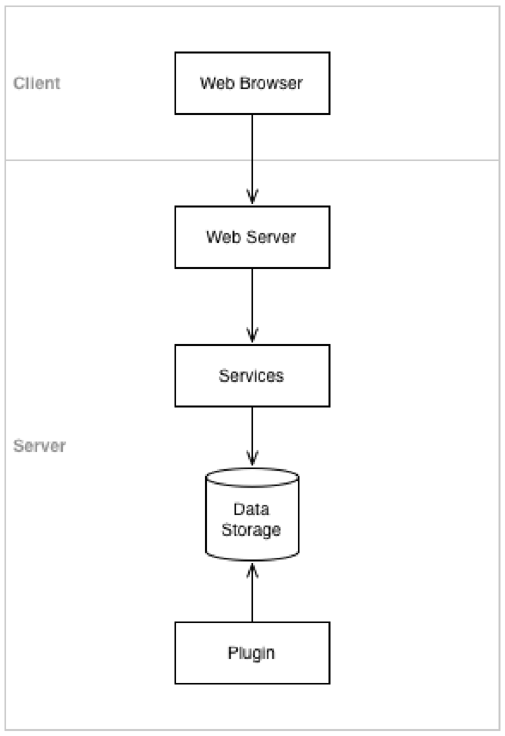


Figure 4.10: Alternative architecture

The interfaces for the alternative architecture are shown in Figure 4.11.

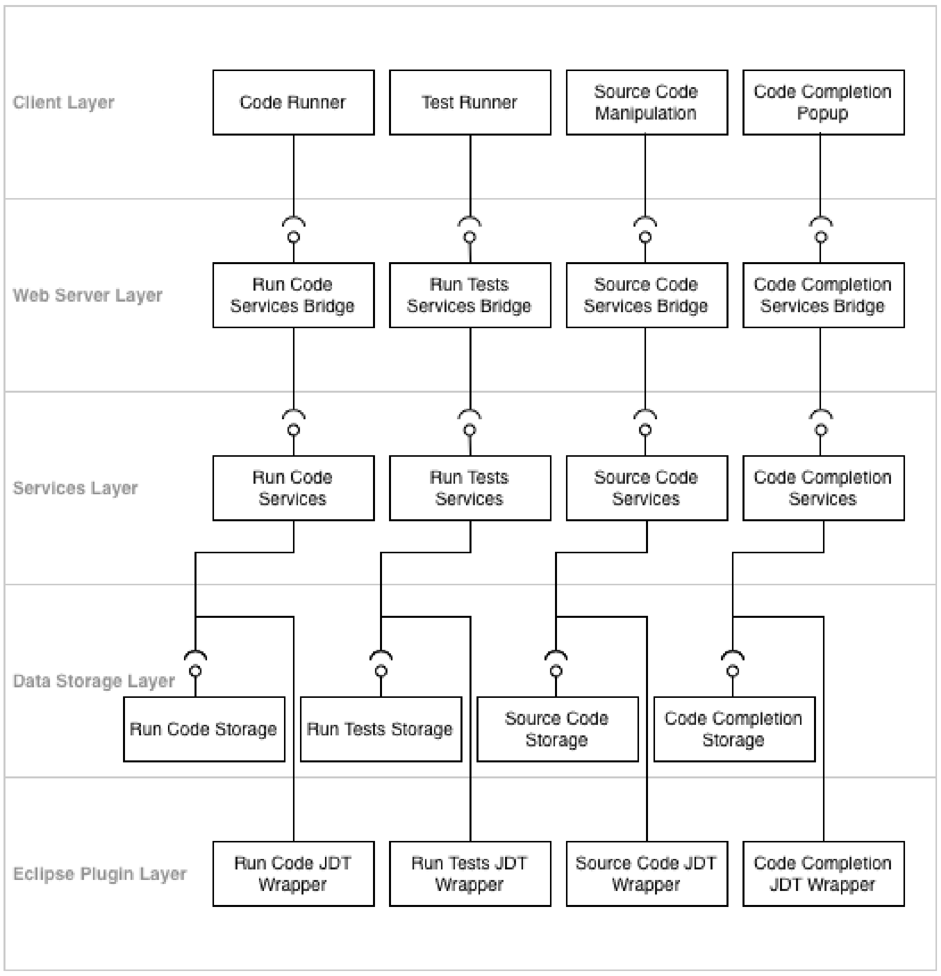


Figure 4.11: The interfaces for the alternative architecture

This architecture would result in another data flow, as shown in Figure 4.12.

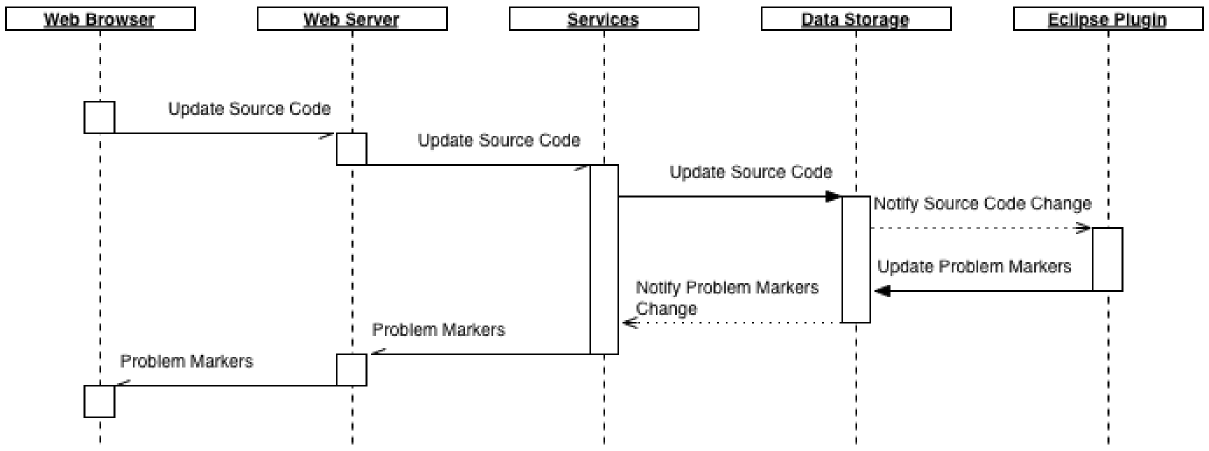


Figure 4.12: An example of how the data flows through the interfaces (Alternative architecture)

This architecture could possibly lead to a more elegant implementation as the services layer would only communicate with the data storage (using a single interface), in contrast to both the data storage and the plugin layer (using two different interfaces).

If this architecture were to be implemented, both the services layer and the plugin layer should observe the data storage for changes and react upon those changes.

A big advantage with this architecture is that the services layer will always have immediate access to the latest data from the plugin layer (because the data is cached in the data storage). In the original approach, the services layer must wait until the plugin layer is ready before it can retrieve the data.

However, this architecture would make the data storage more complex as it must store more data, e.g. data about run code results, problem markers, etc., which is transient in the original approach.

If we consider the scalability of this architecture, we see that data storage could potentially introduce a new bottleneck (in the previous section we stated that we expect the Eclipse plugin to become a bottleneck in the system) because all communication between the services layer and the plugin layer is funneled through the data storage. Depending on how the data storage is implemented, the data storage might save all the data to disk (which require expensive I/O operations) for each request.

4.2.5 Conclusion

Because the alternative architecture (described in section 4.2.4) does not add any significant improvements to our solution (based on the functionality listed in section 4.1) we decided to implement the architecture as shown in Figure 4.7 (from section 4.2.3). Thus, the following sections are based on this architecture.

4.3 Communication

In this section we will look closer at the communication between the different parts of the architecture.

4.3.1 Communication between client and server

As we are using web technologies, we are left with the following options for communication:

- Polling/long-polling
- SSE (Server-Sent Events)
- WebSocket

More information about these technologies are found in section 9.3.1.

Because the client and the server will exchange messages quite frequently (up to 5 messages per second) we chose to communicate using WebSocket. It has the least overhead (it does not need to open a new connection for every message) and it supports bi-directional, asynchronous messaging.

4.3.2 Internal communication

The internal communication must naturally handle the same amount of messages sent between the client and the server (up to 5 messages per second).

Because the Eclipse plugin runs in a separate process, we could not stick to normal Java method invocation. We looked at the following technologies for handling inter-process communication:

- Java RMI
- Jini
- CORBA
- Akka
- Storm

More information about these technologies are found in section 9.3.2.

We elected to use Akka for the internal communication. Akka is a good choice because it is based on the actor model. The actor model makes it easy to communicate across threads, processes and even different nodes (explained in more details in section 9.3.2). Akka also includes a package that is compatible with OSGi, which makes Akka easy to set up with our Eclipse plugin.

As described earlier, our architecture is divided into multiple layers (shown in Figure 4.7). To avoid overlapping responsibilities, we added an actor to each of these layers, shown in Figure 4.13.

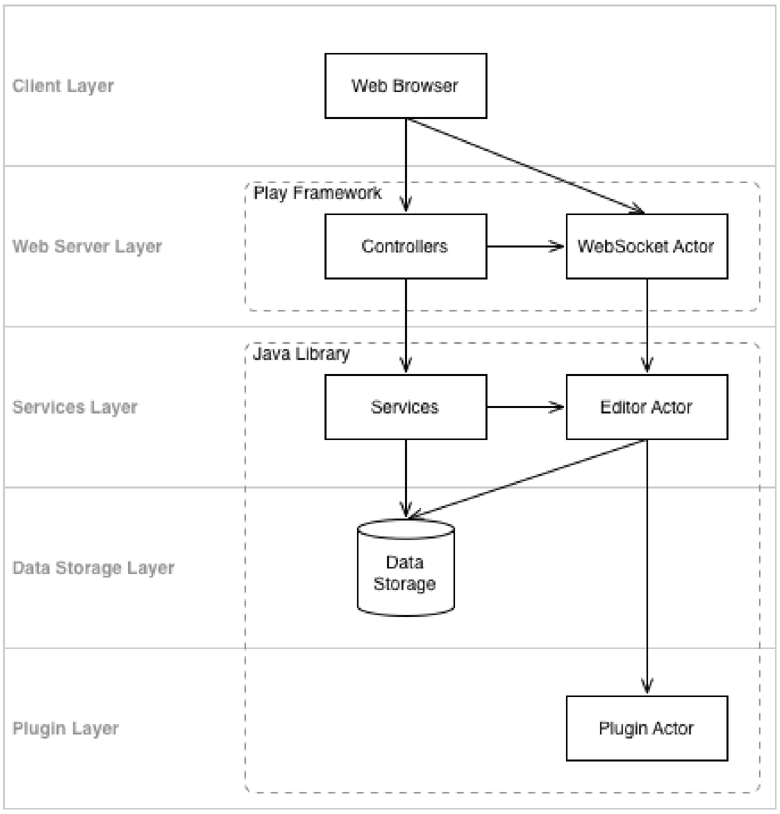


Figure 4.13: The architecture including the actors

The WebSocket Actor is responsible for translating JSON data (received through WebSocket) into Akka messages (and vice-versa) and forwarding them to the Editor Actor. The Editor Actor is responsible for bootstrapping the Eclipse plugin (shown as Plugin Actor). The Plugin Actor is responsible for controlling the Eclipse workspace.

The system will create a new set of actors (WebSocket Actor, Editor Actor, Plugin Actor) for every client. This also means that a new Eclipse instance will be started. We will discuss how this affects the performance of the system and possible improvements in section 4.4.

Because Akka supports “stashing” and “unstashing” of received messages, we set up the Editor Actor to “stash” incoming messages from the client until the Eclipse plugin is ready. What “stashing” means is that an actor can receive messages before it is ready to handle them (maybe because it waits for a specific message). When the actor is ready, it will be able to handle the previously “stashed” messages. We utilize this functionality to allow the client to send source code updates and other actions to the server before the Eclipse process is ready. When the Eclipse process is ready, the messages will be handled and the client will consequently receive the results.

4.3.3 Editor API

In order for the client to know what messages and data to expect from the server, we created an API for the editor. Figure 4.14 shows the messages that the client can send to the server as well as the corresponding replies from the server. These messages are based on the previously defined Java interfaces, shown in Figure 4.4. The methods' parameters denote the values sent from the client to the server, while the methods' return values denote the data that is sent from the server to the client.

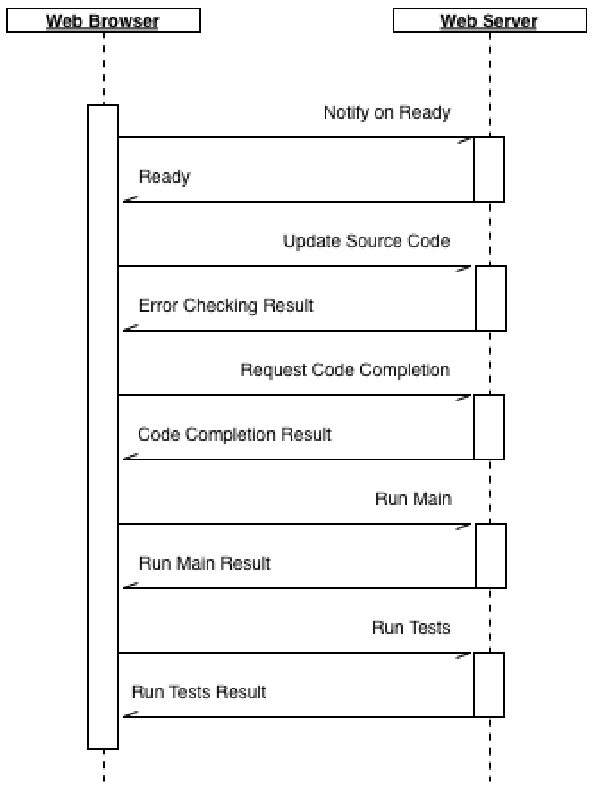


Figure 4.14: Messages sent between client and server (Akka messages)

4.3.4 Conclusion

With the aforementioned setup we have a complete communication channel between the client (web browser) and the Eclipse plugin.

To give an example of how the data flow works, we can look at how the system is initiated. The following sequence describes a similar flow as shown in Figure 4.9:

1. The client requests a WebSocket from the server. When the WebSocket is initialized on the client, it sends a "Notify on ready" message as JSON data.
2. The WebSocket Actor receives the JSON data and translates it to an Akka message and forwards it to the Editor Actor.

3. The Editor Actor starts a new Eclipse plugin in the background, passing its own Akka-address so that the Plugin Actor can inform the Editor Actor when it has started.
4. The Editor Actor receives a “Ready” message from the Plugin Actor and forwards it to the WebSocket Actor.
5. The WebSocket Actor translates the “Ready” message to JSON data and sends it to the client.

From this point the communication is established. Other messages follows the same route. An illustration of the data flow, showing multiple clients, is shown in Figure 4.15.

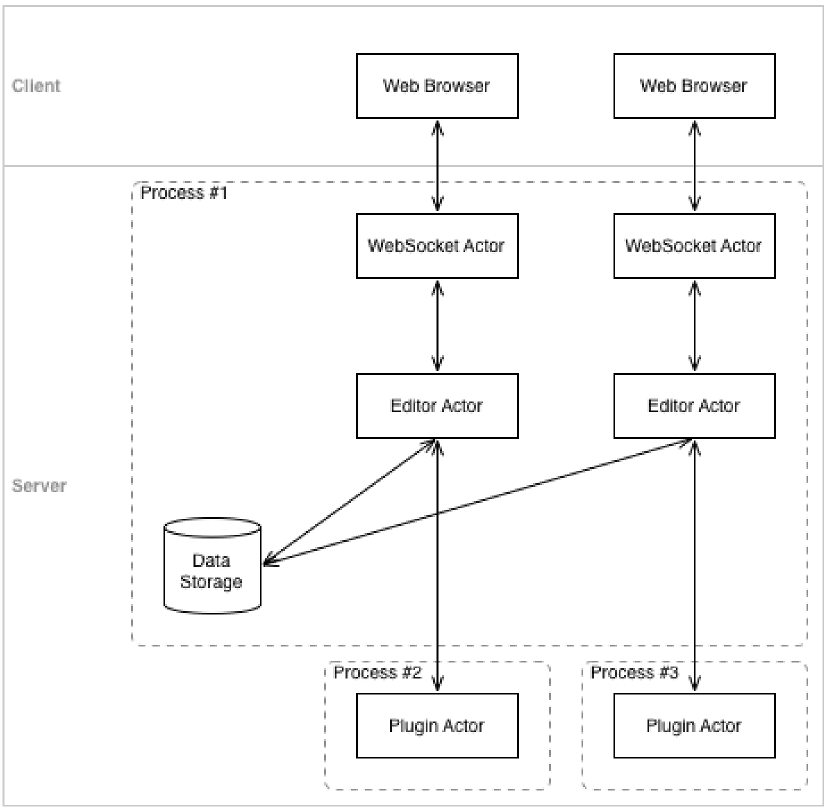


Figure 4.15: Data flow between the web browser and the Eclipse plugin

4.4 Possible improvements

In this section we will look at some possible improvements that we could have made to editor.

4.4.1 Re-use Eclipse processes

Instead of creating new Eclipse processes for each request, we could instead have re-used old Eclipse processes. This would require some changes in the services layer. The services layer must keep track of which Eclipse processes that are currently in use and it must be able to wipe the contents from the previous user.

In section 2.2.4 we saw that the Eclipse IDE has a notion of a workspace, projects, source folders, packages and source files. This is an important distinction when deciding how we should re-use the Eclipse process.

In the current solution we create a new workspace for each user and for each problem, as shown in Figure 4.16. This is the simplest solution because it means that there is no shared state between the users or the problems. However, from a resource allocation-perspective, this solution is not an optimal solution, which is evident in section 5.2.1 and 5.3.1.

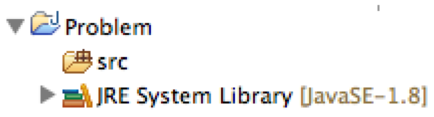


Figure 4.16: A single problem per workspace

We could divide this differently, e.g. share a single workspace for all users, where each user gets a single project and each problem gets a separate package inside of this project, as shown in Figure 4.17. This division is probably too fine-grained and it would limit the system to a single node. Another problem with this division is that changes in one package would trigger a re-build of the whole project (all problems in that project).

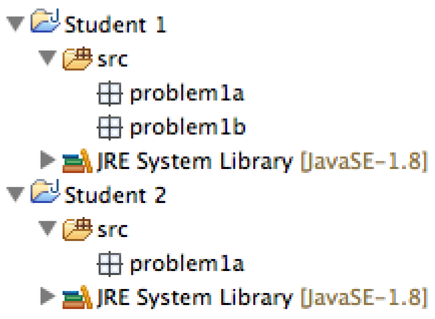

Student 1
src
problem1a
problem1b
JRE System Library [JavaSE-1.8]
Student 2
src
problem1a
JRE System Library [JavaSE-1.8]

Figure 4.17: Multiple users per workspace

To accommodate these problems, a better solution would be to give a single workspace for each user and separate projects for each problem, as shown in Figure 4.18. This division would allow users to be divided on multiple nodes and changes in one project would only trigger a re-build for that exact project.

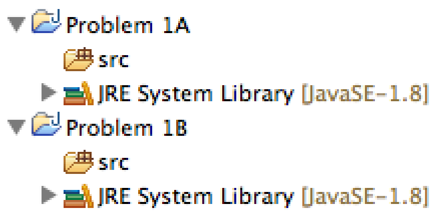

Problem 1A
src
JRE System Library [JavaSE-1.8]
Problem 1B
src
JRE System Library [JavaSE-1.8]

Figure 4.18: Multiple problems per workspace

The big advantage of re-using Eclipse processes is that it would significantly reduce the startup time as well as reduce both CPU and memory consumption.

4.4.2 Reduce the footprint of the Eclipse software

A simpler optimization, but not as effective as re-using Eclipse processes, is to reduce the number of bundled plugins for the installed Eclipse software. Both optimization can be applied simultaneously or individually.

To reduce the footprint of the Eclipse software we could use the Eclipse Target Platform (The Eclipse Foundation, 2014). The Eclipse Target Platform lets us specify the minimum number of plugins that are required by the system. By default, all plugins that are bundled with the Eclipse installation are loaded. The specification is saved in a *target definition file*.

This optimization would reduce the startup time as well as reduce both CPU and memory consumption.

4.4.3 Scalability

The current version of the system is only able to scale up and not scale out (see definitions in section 2.3.2). However, because we have used Akka as the communication interface, the system is well-prepared to distribute the work across multiple nodes. An example of such distribution is shown in Figure 4.19. This figure is very similar to Figure 4.15, where the main difference is that the plugin actors now live on separate nodes instead of in separate processes.

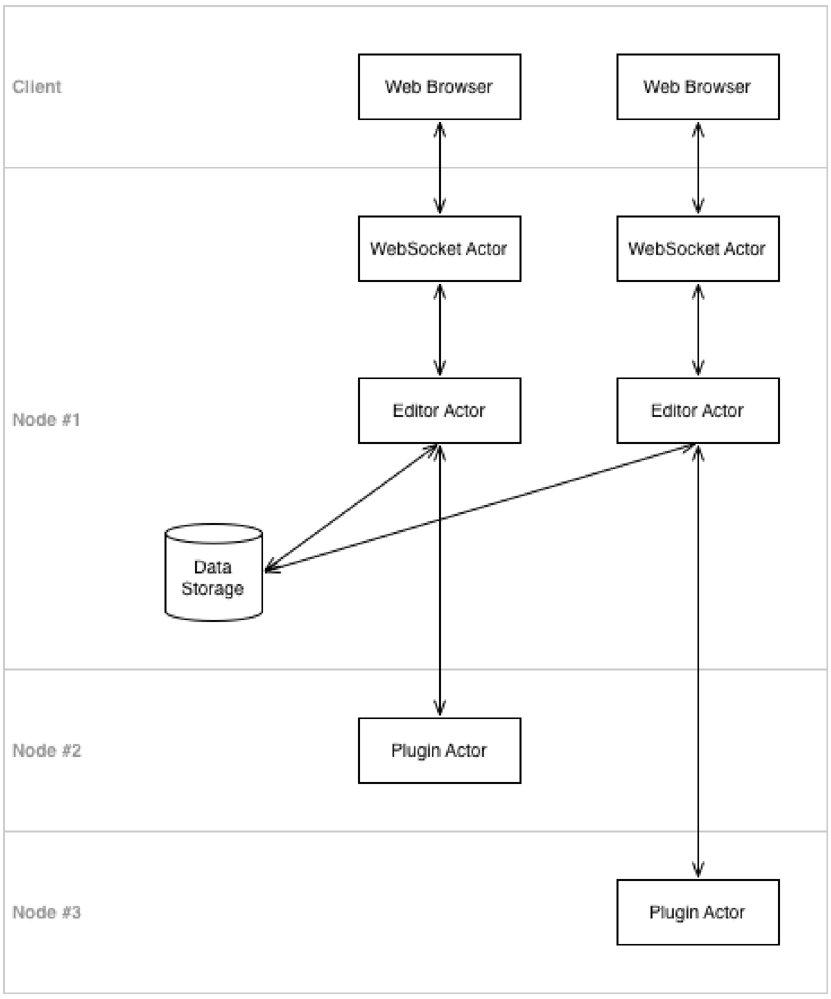


Figure 4.19: Distribution of plugins across multiple nodes

By scaling out, we could (in theory) support a vast amount of users.

4.4.4 Simplify deployment

The current version of the system uses Play Framework as the web server. The Play Framework makes it easy to get started developing web applications (See section 9.2 for more reasons why we chose this web framework).

To simplify the deployment we could have embedded a web server, namely Equinox HTTP service, inside an OSGi plugin (The Eclipse Foundation, 2014). Then the whole system would consist of only OSGi plugins, making the system easier to deploy.

One could take this idea even further by integrating the whole system into a single OSGi plugin. This would, however, limit the scalability of the system.

5 - Experiments and results

This chapter contains the experiments performed on the system to make sure it meets the project goals. We will primarily be testing the goal pertaining to the performance of the web-based code-editor, as the goal pertaining to student efficiency was already proven analytically in Chapter 3, but the goal pertaining to responsive design will also be tested as part of this chapter. The goal pertaining to what kind of assignments the editor is most suited for will be a part of the discussion in section 6.2.

5.1 Experimental plan

We remember from the introduction that the editor goals were:

- To create a web-based code-editor with good IDE-functionality.
- To determine what kind of assignments in TDT4100, and in general, the finished editor is suited for (size, complexity, etc.).

What we mean by "good IDE-functionality" is described in the requirements appendix (section 8.2.2). We mention that we want instantaneous syntax highlighting, one second code completion and one second error checking. We also mention that we want the editor to be ready for use in less than 15 seconds (the measured startup time of Eclipse).

This means that the first goal is primarily about response time, which is easily testable by simulating user interactions and measuring the response time, which is exactly what we will do. We will also test the scalability of the system by simulating multiple users.

The second goal requires a more analytical approach where we have to compare the editor features and the results from the experiments to the kind of assignments available in the course. This is not directly measureable, and will therefore be a part of the discussion in Chapter 6 instead.

We also have a goal about utilizing responsive design when implementing the prototype. This will be tested by simulating running the website on these three types of devices and discussing the results in relation to the theory in section 2.4.

5.2 Testing editor response time

The editor will be tested on two different machines, both belonging to the authors of the thesis. Which machine was used will be written before each experiment. The machine specs can be seen in Table 5.1 and Table 5.2. All measurements in this section are given in milliseconds.

CPU	RAM	Disk	OS
i7-3517U 1.9-2.4GHz	10GB 1600MHz	256GB SSD Read: 540MB/s Write: 520MB/s	Windows 7 PRO 64bit

Table 5.1: David's machine specs

CPU	RAM	Disk	OS
i7-3820QM 2.3GHz	16GB 1600MHz	256GB SSD Read: 400MB/s Write: 400MB/s	Mac OS X 10.9

Table 5.2: Christian's machine specs

In comparison, a typical blade server like the Dell PowerEdge M820 has 16 Xeon cores at 2.2 – 2.9 GHz. It also has a much larger cache, and runs a dedicated operating system, giving it maybe four or five times the power of our best test machine (Christian's laptop).

5.2.1 Startup time

Setup

To notify the users that the editor is ready, we created a readiness-indicator in our editor. When a user loads the webpage, the indicator is red, and when the server side is ready to handle client requests it changes to green. This was achieved by tapping into the WebSocket communication. The startup time of the editor can be easily measured with JavaScript in the same manner (key lines highlighted in red).

```
webSocket.onopen = function() {
	sendMessage("notifyOnReady");
	console.time("measureStartupTime");
};

webSocket.onmessage = function(msgevent) {
	var object = JSON.parse(msgevent.data);
	if (object.type === 'ready') {
		console.timeEnd("measureStartupTime");
		$('#readiness-indicator').addClass('ready');
	}
}
```

To measure the startup time, we will simply start the editor ten times, and calculate the average. The tests will be run from Christian's machine, and David's machine (see Table 5.1) will be used as the server.

Results

As can be seen from Figure 5.1 and Table 5.3, the editor starts in 3.5 seconds on average, with a standard deviation of 178ms. This is a very good result, considering our goal of being ready in below 15 seconds (see 8.2.2). We will find out how well this scales in section 5.3.1.

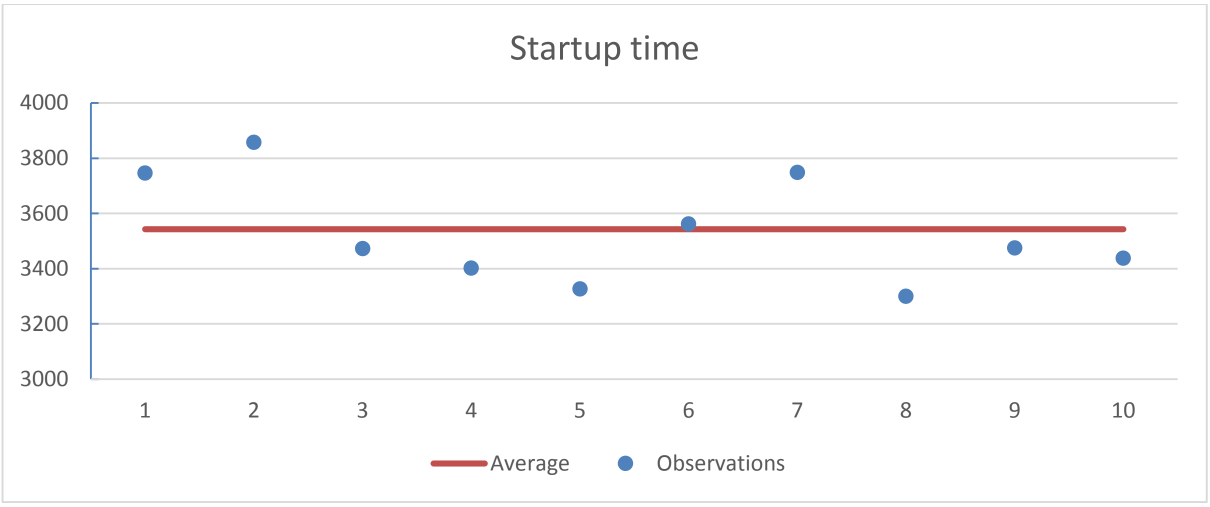


Figure 5.1: Editor startup time

Average startup time	3543ms
Startup time standard deviation	178ms

Table 5.3: Editor startup time

5.2.2 Running code, running tests, error checking, and code completion

Setup

We will use the same approach for measuring the performance of user initiated and real-time events as we did for startup time. We will also be trying out different file sizes to see how much of an impact that will have on the performance. We will use the file sizes 1KB, 16KB, 32KB and 64KB. 64KB allows the students to write programs up to around 3 000 lines of code (assuming an average of 20 bytes per line of code) and is the maximum capacity of WebSocket. We believe this is more than enough for the intended use of the system, where a typical problem solution is most likely to be around a couple of kilobytes.

Due to the amount of tests that have to be run, we will only test each category five times instead of the previous ten. We feel confident in this decision due to the low standard deviation of the first measurements. The reduction makes for 80 measurements instead of 160, and while 160 measurements might seem doable, we are going to be running these multiple times when we are testing the scalability of the editor. Reducing the amount of runs from 10 to 5 reduces the total amount of measurements from around 700 to 350.

Results

Running code

Table 5.4 shows the time in milliseconds it takes from you press the "Run code" button in the client until the result is returned to the client. This operation includes getting the code from the client editor, sending it to the server, compiling the code, running the code, sending the result back to the client, and the client handling the received result.

It is clear from the results that the size of the program does not change the total runtime drastically. When the file size was increased 6300%, the total runtime only increased about 30% on average. This points to most of the time being spent on something other than transferring and compiling/running code. The standard deviation is low meaning that the editor performance is consistent.

Size	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5	AVG	STDEV
1KB	643	608	648	622	611	626	16
16KB	820	642	728	737	723	730	56
32KB	761	799	751	759	692	752	34
64KB	912	850	788	779	769	820	54

Table 5.4: Running code (performance experiment results)

Running tests

Table 5.5 shows the time in milliseconds it takes from you press the "Test code" button in the client until the result is returned to the client. This includes getting the code from the client editor, sending it to the server, compiling it, testing it, sending the result back, and the client handling the received result.

The results are very similar to those of running code, which is to be expected since the operation is very similar too. It is basically running code plus running tests.

Size	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5	AVG	STDEV
1KB	984	917	746	873	942	892	82
16KB	1008	949	944	983	992	975	25
32KB	862	900	925	924	1198	962	120
64KB	1046	933	1004	948	1021	990	43

Table 5.5: Running tests (performance experiment results)

Error checking

Table 5.6 shows the time in milliseconds it takes from an error checking request is sent from the client until the result is returned to the client (these requests occur 300ms after the last editor change-event). This operation includes getting the code from the client editor, sending it to the server, performing the analysis, formatting the analysis result, sending the result back, and the client handling the received result. These measurements are considerably lower than the ones we saw while running and testing code, and it appear that program size has very little impact on the overall runtime.

Size	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5	AVG	STDEV
1KB	297	296	204	246	271	263	35
16KB	325	286	305	336	329	316	18
32KB	281	289	308	270	267	283	15
64KB	386	287	395	205	372	329	73

Table 5.6: Error checking (performance experiment results)

Code completion

Table 5.7 shows the time in milliseconds it takes from a code completion request is sent from the client until the result is returned to the client. This operation includes getting the code from the client editor, getting the current offset (cursor position), sending this information to the server, performing the code completion analysis, formatting the result (removing duplicates), sending the result back, and the client handling the received result.

These measurements are the lowest of all the measurements we gathered, and they also have the highest standard deviation (considering their average runtime). This further suggests that program size has very little impact on the overall runtime, and it also suggest that the variation in runtime is largely dependent of the other processes running on the server.

Size	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5	AVG	STDEV
1KB	120	185	137	205	102	150	39
16KB	178	258	184	106	125	170	53
32KB	108	121	107	213	123	134	40
64KB	219	106	159	122	179	157	40

Table 5.7: Code completion (performance experiment results)a

Results summary

The results summary table (Table 5.8) shows average runtime for each of the four operations and each of the four file sizes. The numbers are collect from the "Average" column in Table 5.4, Table 5.5, Table 5.6 and Table 5.7 All times are in milliseconds.

File size	Run code	Test code	Error checking	Code completion
1KB	626	892	263	150
16KB	730	975	316	170
32KB	752	962	283	134
64KB	820	990	329	157

Table 5.8: Editor performance results summary

From comparing the runtimes of the different operations in Table 5.8 one is able to draw some interesting conclusions. For instance, it appears that no more time than ~150milliseconds is spent on actually transferring the program, as can be seem from the code completion column. Further, there does not appear to be any significant increase in transfer time due to file size (this is further support by the results from the error checking column). This means that the increased runtime for "Run code" and Test code" is due to the runtime of the actual Java program that is being executed on the server. Since the increase in "Run code" and "Test code" runtime is very small in comparison to the increase in file size, it is reasonable that there is a constant cost associated with compiling and running the program which we have little or no control over.

5.2.3 Conclusion

We are very pleased with the results of the response time experiments. As can be seen from Table 5.9, we beat our goals by a fair margin (the column "Actual" contains the worst case average from our tests, disregarding file size). We had no time-goal for running code and running tests, as it depends mostly on the nature of the program being run, and not on the editor.

Feature	Goal	Actual
Startup time	Less than 15 seconds	3.5s
Error checking	Less than 1 second	329ms
Code completion	Less than 1 second	170ms

Table 5.9: Editor performance (goals vs results)

The editor feels just like a native application when running only one instance on the server, and the startup time is much lower than full-fledged IDEs. We will have a look at how multiple users affects the performance next.

5.3 Testing editor scalability

In this section we will examine the impact multiple concurrent users has on our systems response time. Since file size did not impact the performance much, we will use a 16KB file for all tests.

5.3.1 Startup time

Setup

To test the scalability of the startup performance we will double the amount of clients started until we get bored. Each client will be started around the same millisecond, which is the worst case scenario. This will be achieve by using a Chrome plugin called "Reload All Tabs"[12]. This plugin, as the name suggests, allows the user to reload all tabs simultaneously.

Results

Table 5.10 contains the summarized results of the startup time scalability test, and each column represents the average start up time of the individual editors that were started. The full results are available in Table 10.1, Table 10.2, Table 10.3, and Table 10.4. The test was only run one time for 16 clients, due to how long it took to run the tests. Even so, considering how low the standard deviations are for the other groups, it is reasonable to assume that the measurement is approximately representative of the true startup time.

The impact multiple users has on startup time is noticeable and expected. When the server has to start N eclipse instances, the startup time increases by a factor of N-X (the discount factor X is probably due to caching performed by the operating system).

This increase in startup time could be mitigated by starting only one Eclipse instance, and allowing several editors to make use of that instance. The startup time of the each instance could also be shortened by stripping the instance of plugins, but the increased would still be linear.

Size	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5	AVG	STDEV
2	5443	5047	4859	5304	5024	5135	210
4	8927	9138	8853	9193	9277	9077	161
8	17783	17892	17797	17554	17614	17728	125
16	36233	-	-	-	-	36233	-

Table 5.10: Editor startup scalability

[12] https://chrome.google.com/webstore/detail/reload-all-tabs/lpkdnfkjhdkcpimadpdcgapffceacjem

5.3.2 Running code, running tests, error checking and code completion

Setup

As mentioned, since there was very little variation as results of the file size, we will use a 16KB file for all our tests. Due to the wait associated with the editor start-up time, and the low standard deviations we have seen so far, we will only run tests with 16 clients one time. In order to ensure that the client instructions are all execute at the same time, we will utilize a script-utility for Chrome, called chrome-cli[13].

The script code for running code can be seen in the snippet below:

```
#!/bin/bash

code="\$('#run-code-button').click()"

for n in $(chrome-cli list tabs | awk '{print $1}' | sed -e 's/\[//' | sed -e
's/\]//'); do
    chrome-cli execute $code -t $n > /dev/null
done
```

Scripts like these allow us to control button clicks and keyboard events across all browser tabs, which is necessary in order to start all the operations around the same millisecond.

[13] Source: https://github.com/prasmussen/chrome-cli

Results

Running code

What is included in the operation of running code was explained in section 5.2.2, and the average runtime for a 16KB file was found to be 720ms. We expected the server portion of this runtime to increase linearly with the number of active editors, but that there would be a discount due to the fact that the time spent on the client side would be evenly distributed on the clients.

As can be seen from Table 5.11 the results appear to agree with our expectations. The full results of these tests are available in Table 10.5, Table 10.6, Table 10.7, and Table 10.8.

Clients	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5	AVG	STDEV
2	1155	1435	974	958	973	1099	183
4	1855	1800	1876	1754	1805	1818	43
8	3099	2954	2854	2977	2894	2956	84
16	5306	-	-	-	-	5306	-

Table 5.11: Running code (scalability experiment result)

Running tests

Running tests is the same as running code, plus tests. Our findings reflect this fact. Table 5.12 shows the summarized results. The full results of these tests are available in Table 10.9, Table 10.10, and Table 10.11.

Clients	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5	AVG	STDEV
2	1344	1378	1399	1353	1374	1369	19
4	2486	2571	2594	2437	2612	2540	67
8	4777	3390	3315	4676	4538	4139	647
16	6461	-	-	-	-	6461	-

Table 5.12: Running tests (scalability experiment result)

Error checking

What is included in the operation of error checking was explained in section 5.2.2, and the average runtime for a 16KB file was found to be 316ms. The results here (summarized in Table 5.13) are a little surprising, as the number of editors running does not seem to factor in a lot. This indicates that much of the cost associated with error checking is handled on the client side, or at least that the server side effort is minimal. The full results of these tests are available in Table 10.12, Table 10.13, Table 10.14, and Table 10.15.

Clients	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5	AVG	STDEV
2	278	329	284	271	267	286	22
4	287	314	261	285	290	287	17
8	310	452	264	358	355	348	62
16	449	-	-	-	-	449	-

Table 5.13: Error checking (scalability experiment result)

Code completion

What is included in the code completion operation was explained in section 5.2.2, and the average runtime for a 16KB file was found to be 170ms. The results (summarized in Table 5.14) indicate that this operation scales similarly to running code and tests, having a discounted linear increase. The full results of are available in Table 10.17, Table 10.18, Table 10.19 and Table 10.20.

Clients	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5	AVG	STDEV
2	395	387	374	362	376	379	11
4	681	663	710	701	703	691	17
8	1385	1325	1231	1812	1441	1439	199
16	3329	-	-	-	-	3329	-

Table 5.14: Code completion (scalability experiment result)

Results summary

The result table (Table 5.15) shows the average times for each of the four categories and each of the four file sizes. All times are in milliseconds.

Clients	Run code	Test code	Error checking	Code completion
2	1099	1369	286	379
4	1818	2540	287	691
8	2956	4139	348	1439
16*	5306	6461	449	3229

Table 5.15: Editor scalability results summary

Scalability prediction

Based on the previous findings, we performed a regression analysis to predict how the editor would scale if we pushed it beyond 16 clients. The result can be seen in Table 5.16 (predictions in cursive) and Figure 5.2. As we have seen with all our experiments, the increase is linear, although, when the server runs out of RAM, page swapping will most likely lead to huge delays.

Clients	Run Code	Test Code	Error Checking	Code completion	Startup
1	730	975	316	170	3,5s
2	1,1s	1,4s	286	379	5,1s
4	1,8s	2,5s	287	691	9,1s
8	3,0s	4,1s	348	1,4s	17,7s
16	5,3s	6,5s	449	3,2s	36,2s
32	*10,2s*	*12,5s*	*594*	*5,7s*	*62,9s*
64	*19,8s*	*24,3s*	*912*	*11,3s*	*123,1s*
128	*39,1s*	*47,7s*	*1,5s*	*22,4s*	*243,5s*
256	*77,8s*	*94,6s*	*2,8s*	*44,7s*	*484,4s*

Table 5.16: Editor scalability prediction

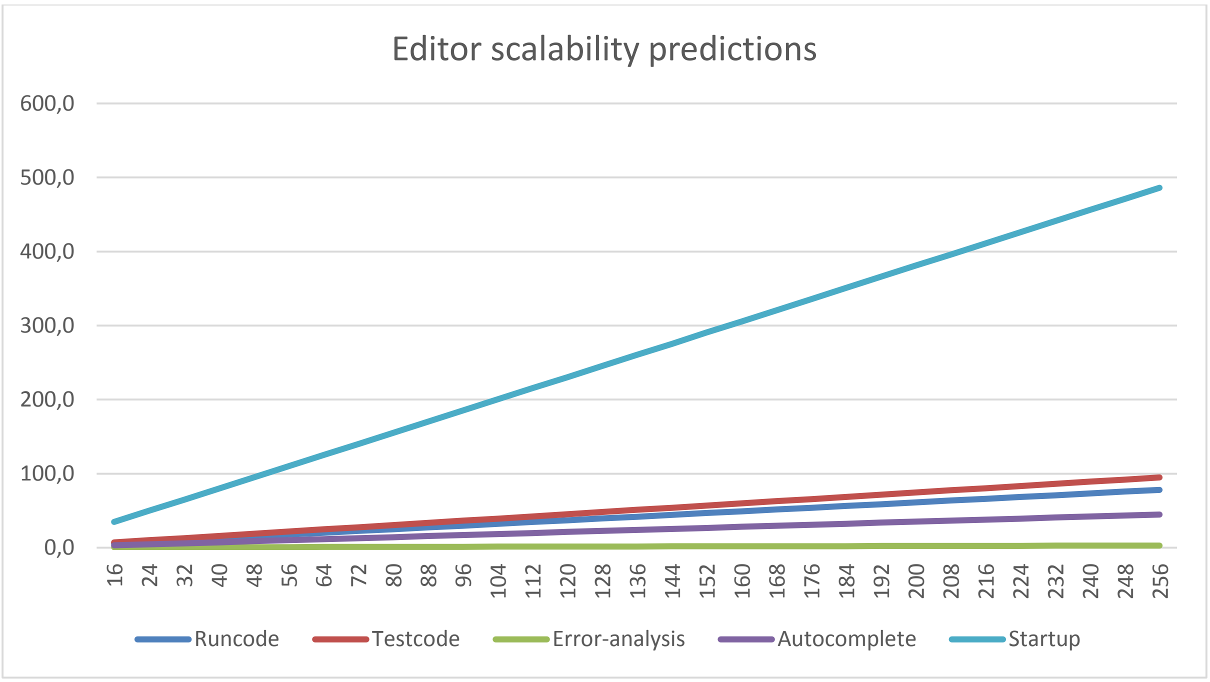


Figure 5.2: Editor scalability prediction

5.3.3 Conclusion

The editor scales more or less as expected. Running two identical Java programs should take about twice as long as running just one, but there is a discount because some of the work is being performed by the clients and the transfer time being distributed. Also the operating system probably caches. Considering how a laptop was able to run 16 Java programs started at the same time and return a result in five seconds, we are not worried about how the system will work in a day to day setting on a server with four or five times the processing power.

One thing that is worrying, however, is the startup time. If, for example, the system is used in a lecture where 200 users open it simultaneously, our predictions indicate that it would take about six minutes for the editor to be ready on each client (if the system is running on David's laptop). Even if we were to strip down the Eclipse instance to reduce the startup time to just one second, it would still take several minutes if 200 users opened it simultaneously. This could be remedied by setting up several servers to run the system, but a change in architecture where multiple users can make use of the same eclipse instance is probably more appropriate.

Another problem with having 200 users is that the RAM-usage for each eclipse instance is about 100MB (not stripped down). This makes for a total of about 20 GBs of memory usage, meaning that we would start to see severe performance problems due to paging. This could be mitigated by adding more RAM or stripping down the Eclipse instance (removing unused plugins), but a change in architecture would again be more appropriate.

5.4 Testing responsive design

To test the responsive design of our website we will use a service called Responsinator[14], which we mentioned in section 2.4. This service emulates iPhone 3-5, old and new Android phones, iPads, and Android tablets, in both landscape and portrait mode. We will use this service to emulate Apple and Android devices, in addition to testing on a laptop. We will test every device in both portrait and landscape mode, but we will not include all tests in the report (it would amount to over 50 screenshots for our four test-pages). The screenshots in this section will also be shrunk in order to prevent report bloat.

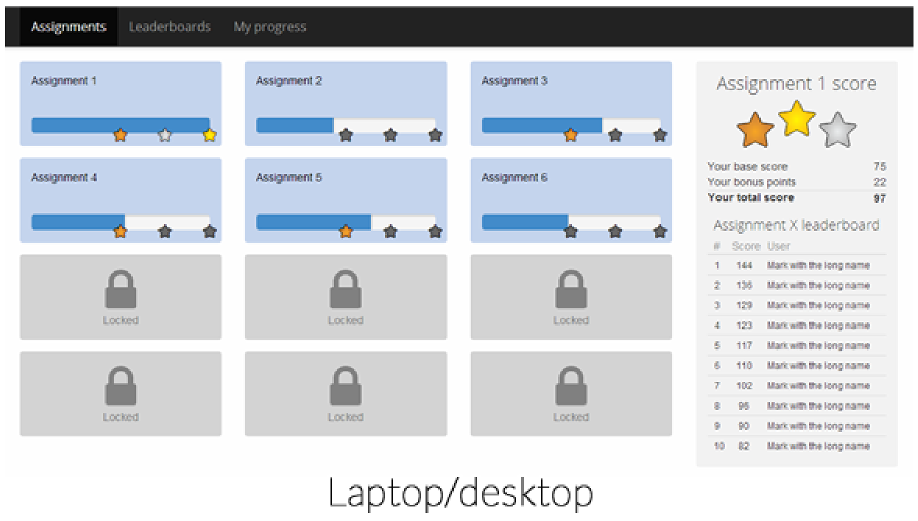
5.4.1 The "assignment overview" view

The assignment overview page is the first page that students see when they open the system. It shows an overview of all the assignments where the students' scores are displayed as progress bars (with star markings). If the user is on a laptop or desktop computer, a leaderboard pane is available on the right hand side (see Figure 5.3). This leaderboard updates to display the correct information when the user hovers over an assignment. On tablets and mobile devices (Figure 5.4), only the assignment grid is shown, as hovering is not possible on touch devices. This design is achieved by utilizing fluid layouts and media queries, as described in section 2.4. Vector graphics are used to ensure that the icons look great on all devices and zoom levels.

A thorough test of the assignment view can be seen in Figure 5.3 (laptops and desktop computers) and Figure 5.4 (mobile devices). As can be seen, there are very few differences between the different mobile phones and tablets.

Going forward, we will only focus on Android devices, as the design is completely platform independent and cutting and arranging the screenshots takes a long time.

[14] Available at: http://www.responsinator.com/



Your base score	75
Your bonus points	22
Your total score	**97**

Assignment X leaderboard

#	Score	User
1	144	Mark with the long name
2	136	Mark with the long name
3	129	Mark with the long name
4	123	Mark with the long name
5	117	Mark with the long name
6	110	Mark with the long name
7	102	Mark with the long name
8	95	Mark with the long name
9	90	Mark with the long name
10	82	Mark with the long name

Figure 5.3: “Assignment overview” view (laptops and desktop computers)

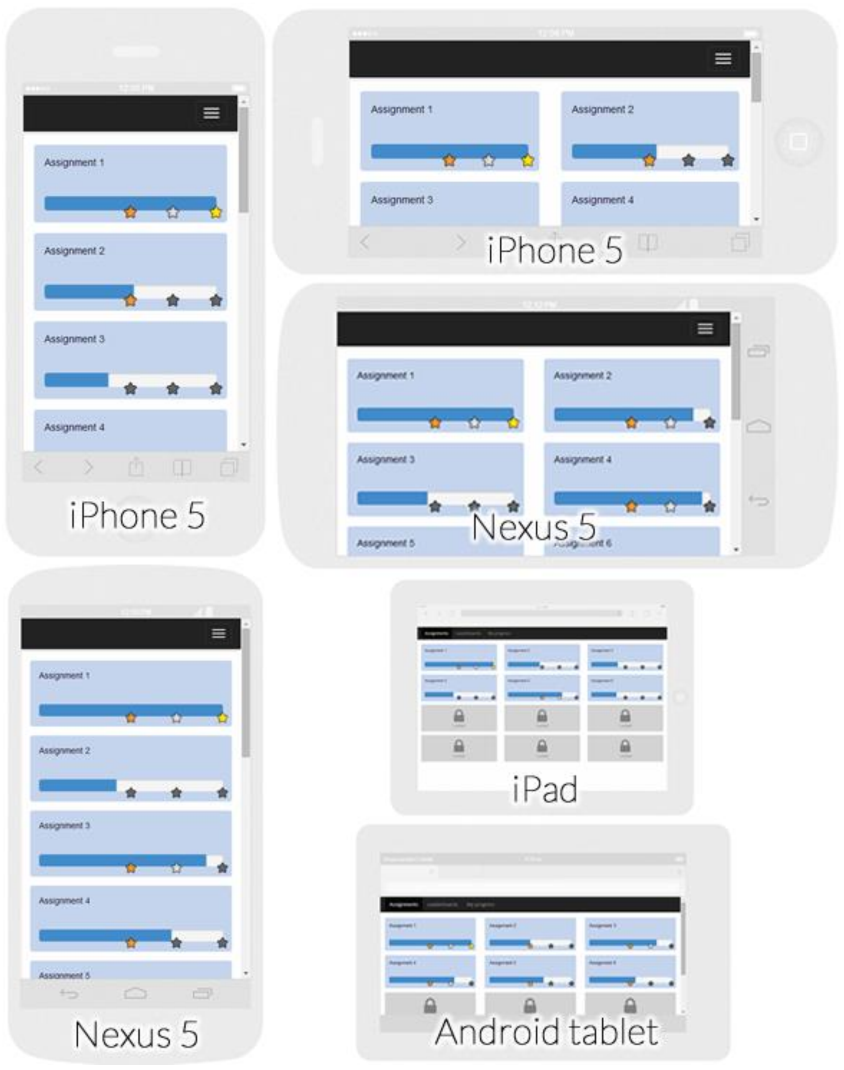
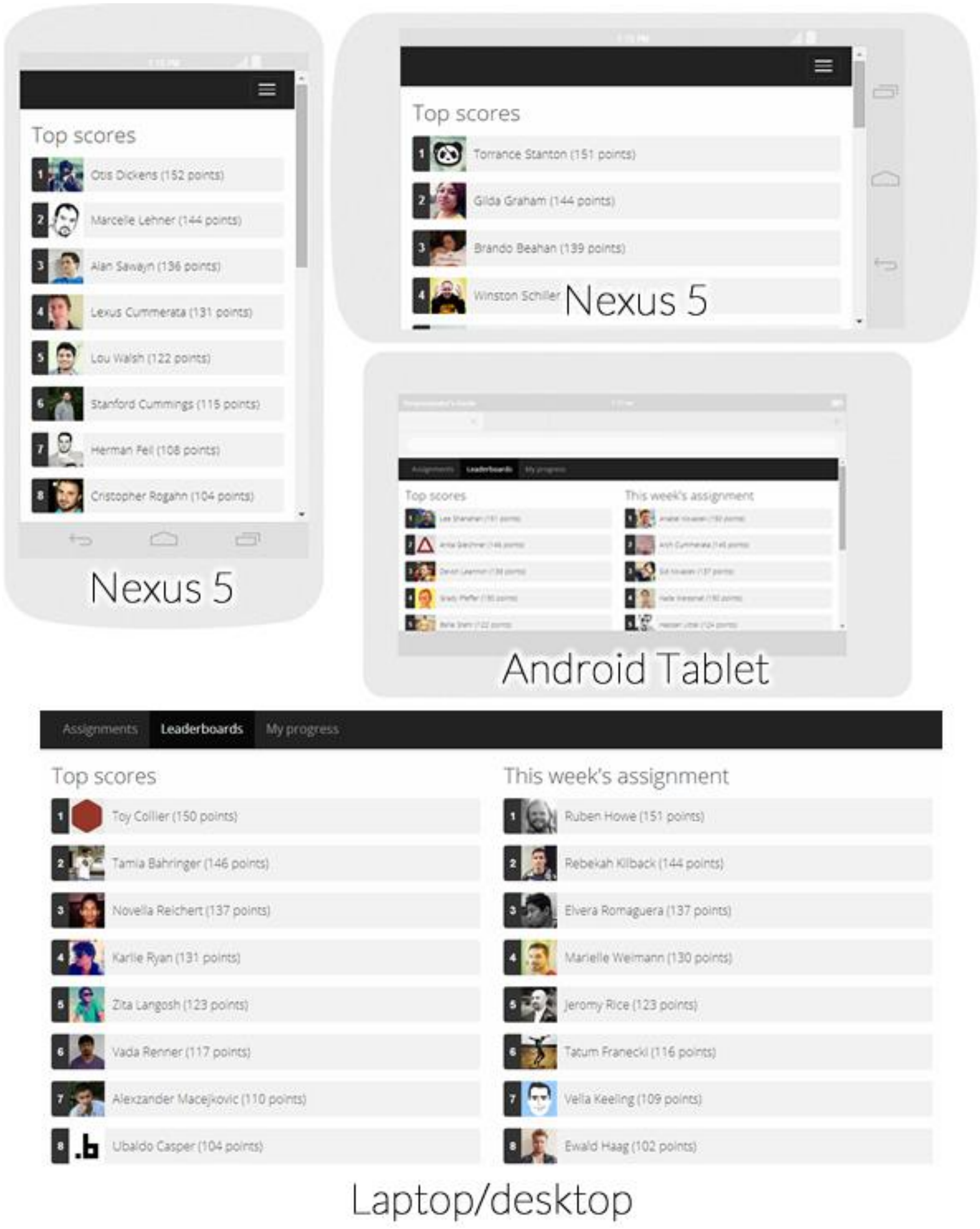


Figure 5.4:” Assignment overview” view (mobile devices)

5.4.2 The “leaderboards” view

The leaderboards page contains two leaderboards: one that shows the overall score, and one that shows the score for the current week. It shows both columns side by side on laptops and tablets in landscape mode, but on smaller devices they are positioned under each other (see Figure 5.5).



Top scores

1 Otis Dickens (152 points)
2 Marcelle Lehner (144 points)
3 Alan Sawayn (136 points)
4 Lexus Cummerata (131 points)
5 Lou Walsh (122 points)
6 Stanford Cummings (115 points)
7 Herman Feil (108 points)
8 Cristopher Rogahn (104 points)

Top scores

1 Torrance Stanton (151 points)
2 Gilda Graham (144 points)
3 Brando Beahan (139 points)
4 Winston Schiller

Top scores

1 Toy Collier (150 points)
2 Tamia Bahringer (146 points)
3 Novella Reichert (137 points)
4 Karlie Ryan (131 points)
5 Zita Langosh (123 points)
6 Vada Renner (117 points)
7 Alexzander Macejkovic (110 points)
8 Ubaldo Casper (104 points)

This week’s assignment

1 Ruben Howe (151 points)
2 Rebekah Kilback (144 points)
3 Elvera Romaguera (137 points)
4 Marielle Weimann (130 points)
5 Jeromy Rice (123 points)
6 Tatum Franecki (116 points)
7 Vella Keeling (109 points)
8 Ewald Haag (102 points)

Figure 5.5: “Leaderboard” view

5.4.3 The “my progress” view

The student progress page contains user information (nickname, score, leaderboard position), along with information about all assignments, problems and achievements. The view has a single column layout on cellphones and portrait mode tablets, and a two row/three column layout on landscape mode tablets, laptops, and desktop computers (see Figure 5.6).

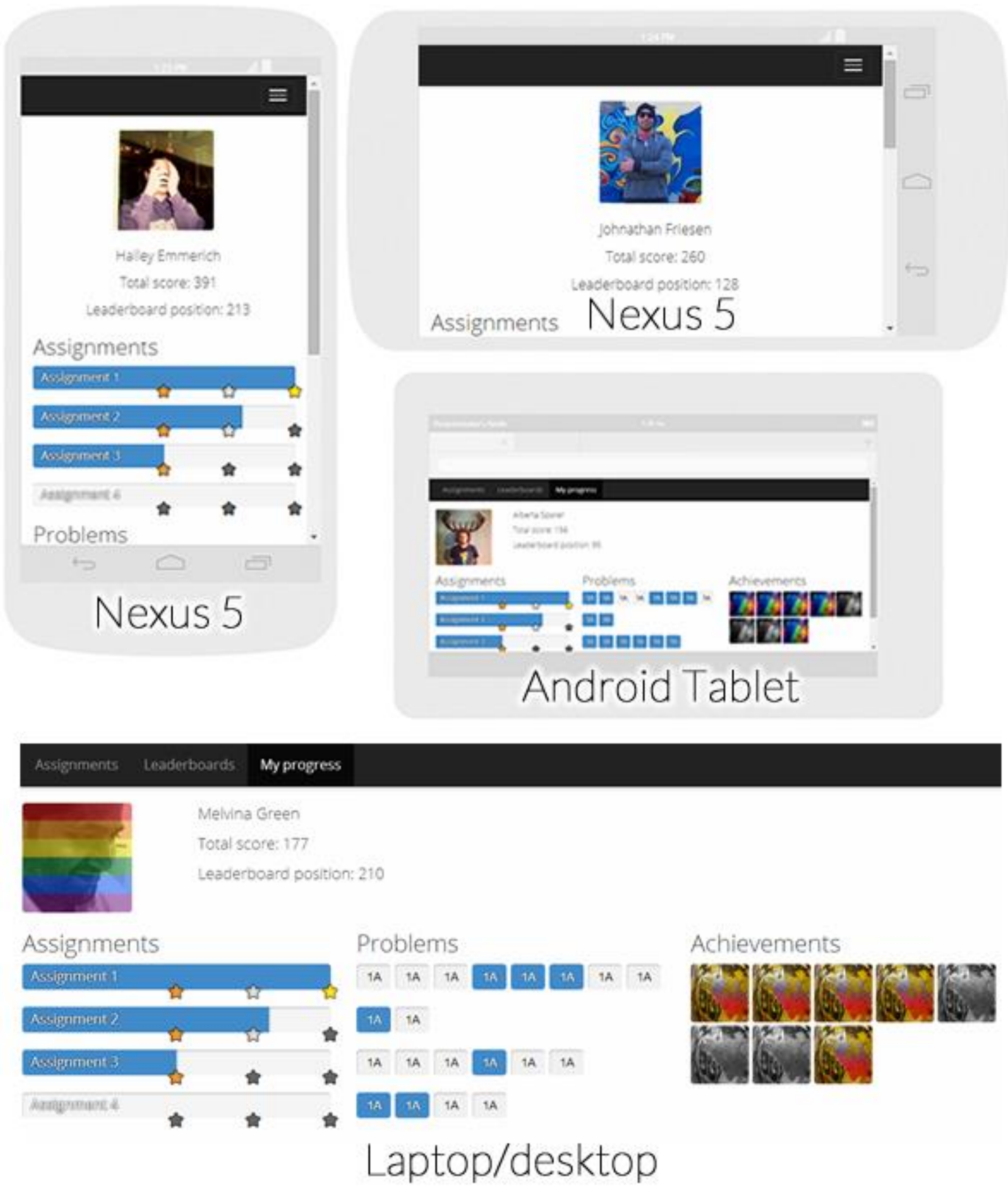
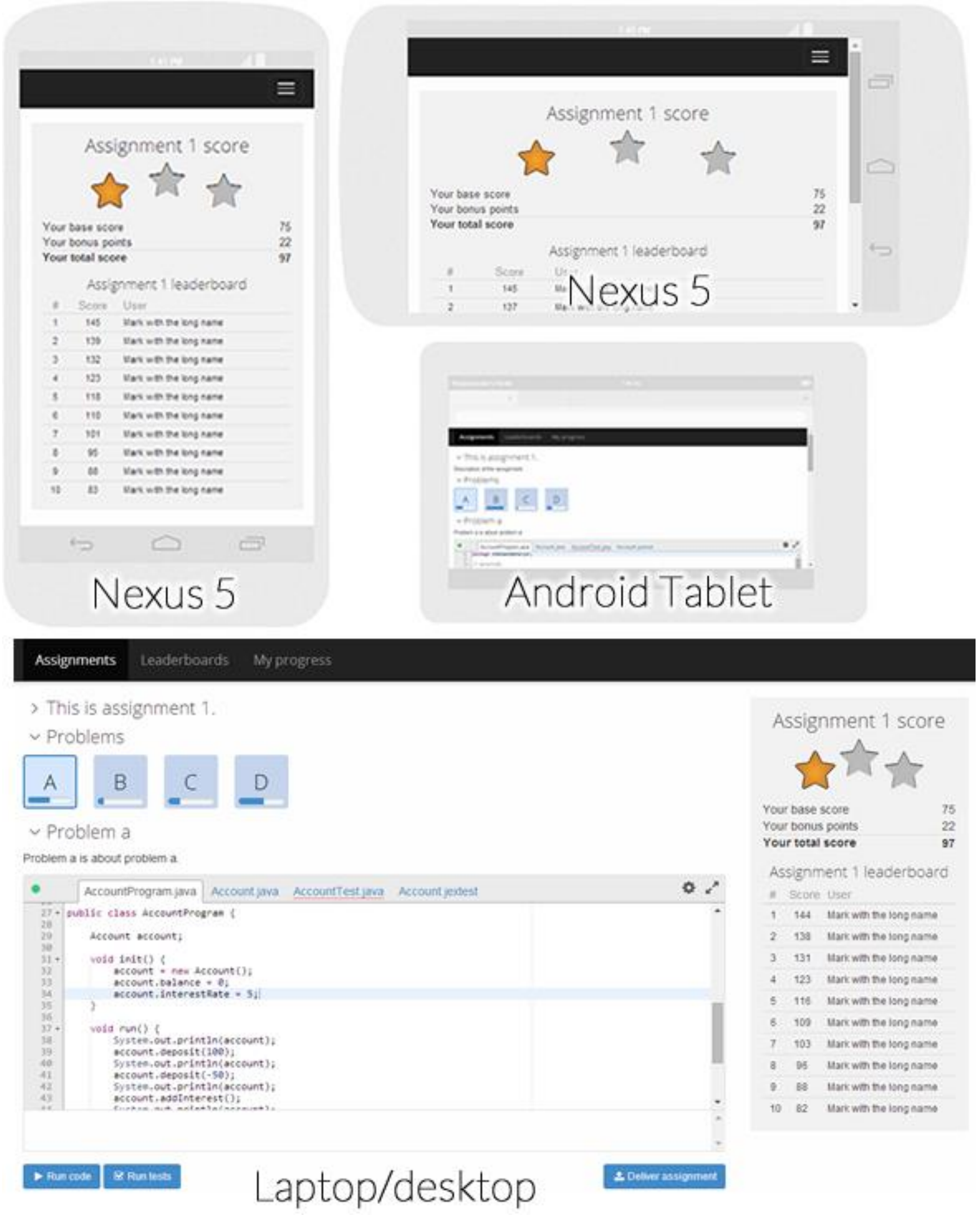


Figure 5.6: “My progress” view

5.4.4 The "problem" view

The "problem" page displays the current assignment description, the current problem description and the code-editor (remember, all problems belong to an assignment). It also displays the leaderboard for the assignment.



Assignment 1 score

Your base score	75
Your bonus points	22
Your total score	**97**

Assignment 1 leaderboard

#	Score	User
1	145	Mark with the long name
2	139	Mark with the long name
3	132	Mark with the long name
4	123	Mark with the long name
5	118	Mark with the long name
6	110	Mark with the long name
7	101	Mark with the long name
8	95	Mark with the long name
9	88	Mark with the long name
10	83	Mark with the long name

> This is assignment 1.

⌄ Problems

⌄ Problem a

Problem a is about problem a.

AccountProgram.java | Account.java | AccountTest.java | Account.jexdest

```
public class AccountProgram {

    Account account;

    void init() {
        account = new Account();
        account.balance = 0;
        account.interestRate = 5;
    }

    void run() {
        System.out.println(account);
        account.deposit(100);
        System.out.println(account);
        account.deposit(-50);
        System.out.println(account);
        account.addInterest();
```

Assignment 1 score

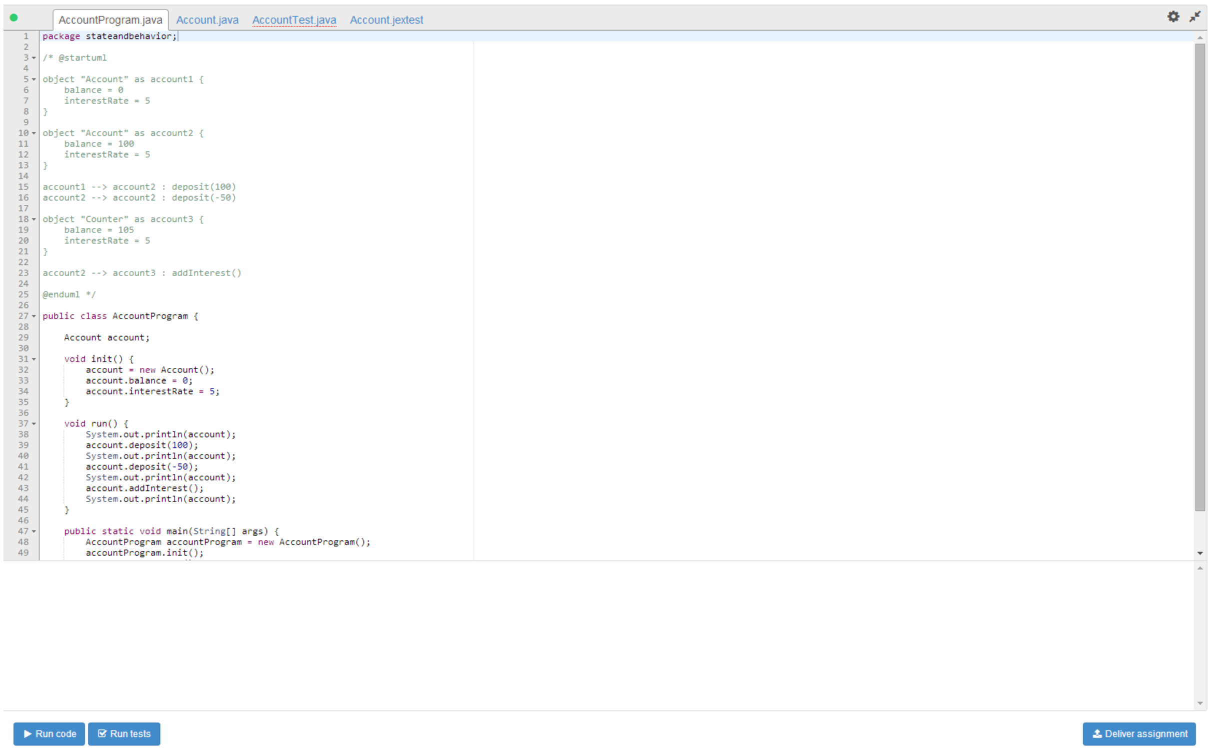
Your base score	75
Your bonus points	22
Your total score	**97**

Assignment 1 leaderboard

#	Score	User
1	144	Mark with the long name
2	138	Mark with the long name
3	131	Mark with the long name
4	123	Mark with the long name
5	116	Mark with the long name
6	109	Mark with the long name
7	103	Mark with the long name
8	96	Mark with the long name
9	88	Mark with the long name
10	82	Mark with the long name

Figure 5.7: "Editor" view

On cellphones and portrait tablets, only the assignment score and leaderboard is displayed. While on landscape mode tablets, laptops, and desktop computers, the full editor is available, and the leaderboard is a side pane (see Figure 5.7). This is done to ensure that students have access to the information they want immediately. When a student is checking a specific assignment using his mobile phone, he is probably interested in seeing the leaderboard, not the editor. The editor can also be maximized, in which case it looks identical on all devices (its width and height is set to 100% of the device). How the editor looks maximized on a laptop can be seen in Figure 5.8.



```java
package stateandbehavior;

/* @startuml

object "Account" as account1 {
    balance = 0
    interestRate = 5
}

object "Account" as account2 {
    balance = 100
    interestRate = 5
}

account1 --> account2 : deposit(100)
account2 --> account2 : deposit(-50)

object "Counter" as account3 {
    balance = 105
    interestRate = 5
}

account2 --> account3 : addInterest()

@enduml */

public class AccountProgram {

    Account account;

    void init() {
        account = new Account();
        account.balance = 0;
        account.interestRate = 5;
    }

    void run() {
        System.out.println(account);
        account.deposit(100);
        System.out.println(account);
        account.deposit(-50);
        System.out.println(account);
        account.addInterest();
        System.out.println(account);
    }

    public static void main(String[] args) {
        AccountProgram accountProgram = new AccountProgram();
        accountProgram.init();
```

Figure 5.8: “Editor” view with maximized editor

5.4.5 Conclusion

As can be seen from the various figures in section 0, the implementation of a responsive design was a success. The design utilizes all of the principles outlined in section 2.4, and all elements are defined in percentages, resulting in a suitable layout on all target devices. No raster image files were used, i.e. all icons and shapes are vector graphics and thus scale perfectly.

Comments

The use of Faker.js (mentioned in section 9.9) was an enormous help in ensuring that the design would fit user generated content. It was used to generate all avatars, user names and achievement images, which changes on every update. This is why the screenshots all have different placeholder information.

6 - Evaluation and conclusion

This chapter contains the evaluation and conclusion of the thesis.

6.1 Evaluation

We had several goals for this thesis, but to sum it up in an informal way: we wanted to create and implement an online platform which would provide students with everything needed to solve simple programming assignments, while simultaneously providing a motivating atmosphere and reducing time spent on configuration, setup, and other non-programming tasks.

In Chapter 3 we analyzed and re-engineered the current assignment system, reducing the number of environments the students' have to deal with from 8 to 1, the number of tasks from 18 to 4, and the number of distinct steps for the tasks from 74-94 (depending on how the students go about performing the task) to just 11. We completely eliminated every task and step related to setup and configuration, so that each of the 11 steps are concerned with actually solving the assignment.

In Chapter 4 we designed the architecture needed to realize the system from Chapter 3, and in Chapter 5 we tested the finished system. The results from the experiments were very positive. The response time for the various functionality (running code, running test, error checking and code completion) were all well below 1 second, even for large files (64KB). These times are very good, especially considering that the system was running on a laptop instead of a proper server. We saw that the editor performance scaled linearly (with a discount rate), which means that the system will scale well if distributed to more nodes. The experiments also showed that the implementation of the responsive design was a success, and that the system works well on all device types.

6.2 Discussion

Another goal of the thesis was to determine what kind of assignment the editor was best suited for. To answer this question, we must first take a look at the findings from our experiments. As we learned in section 5.2, file size had very little impact on the overall runtime of a program written in the editor. The time it took to run code and tests increased about 200ms when the file size increased from 1KB to 64KB, but the response time for code completion and error analysis varied only with a few milliseconds. This indicates that the response time increase was in Java runtime, and that one would most likely see the same runtime increase when using a native IDE. This further means that program size is not an issue when it comes to what kind of assignments the editor is best suited for. One thing the editor currently cannot do, however, is run GUI programs. It can only display console output and tests results.

This means that the editor is best suited for the first assignments of TDT4100, or any assignment that can be tested programmatically. The editor is thus very well suited for solving algorithmic problems in courses such as TDT4120. The editor would also be well suited for the type of assignments found in TDT4110, but then it would have to utilize a Python interpreter instead of a Java compiler. Our design is modular, however, and the web-part of the system would not need to be changed in order to make this work.

6.3 Future work

A few things still remain to be done before the system can be taken into use.

As we saw from the scalability tests, a change in architecture is a good idea in order to reduce the startup time and memory usage of the editor. The simplest way this could be achieve is probably to start some editors when starting the server and let several students make use to the same Eclipse instance. No other architecture changes are considered necessary before the system can be taken into use, as the system performed and scaled surprisingly well.

The system is not secured against malicious code, and basically acts as a big back-door (or front-door, rather) to the server. The editor should be placed in a sandboxed environment before it is taken into use. Functionality for killing processes that are taking too long (infinite loops, etc.) should also be implemented.

The system features simple authentication functionality, but lacks proper password hashing. The system also does not utilize SSL. This has to be implemented before the system is taken into use.

The system as a whole currently relies on placeholder information to demonstrate the design. Before the system is to be taken into use, these placeholders should be removed and replaced with connections to the data model. The model is already integrated though, so this should be easy enough to accomplish.

6.4 Closing remarks

Even if the system is not completely finished we are very pleased with what we have created (especially considering the high complexity of the editor architecture), and we hope someone will take over and finish the remaining work needed to take the system into use.

We believe our editor offers substantial benefits over other web-based Java editors on the market today, and with a slight change in architecture (to enable better utilization of the Eclipse instances) we think the editor could be become a popular tool for running Java code online.

7 - Bibliography

Ace. (2014, 05 10). *Ace*. Retrieved from Ace: http://ace.c9.io/

AngularJS. (2014, 05 03). *AngularJS*. Retrieved from AngularJS: https://angularjs.org/

Backbone.js. (2014, 05 03). *Backbone.js*. Retrieved from Backbone.js: http://backbonejs.org/

Bonnie E, J., & David E, K. (1996). Using GOMS for user interface design and evaluation: which technique? *ACM Transactions on Computer-Human Interaction*, 287-319.

Camilo, A., Javier, C., & Jordi, C. (2014, 05 26). *Introduction*. Retrieved from EMF-REST: http://emf-rest.com/

Catlin, H., Weizenbaum, N., & Eppstein, C. (2014, 04 26). *Sass Basics*. Retrieved from Sass: http://sass-lang.com/guide

CodeMirror. (2014, 05 10). *CodeMirror*. Retrieved from CodeMirror: http://codemirror.net/

Coyier, C. (2012, 06 11). *CSS-Tricks*. Retrieved from CSS-Tricks: http://css-tricks.com/poll-results-popularity-of-css-preprocessors/

Data & Object Factory, LLC. (2014, 07 23). *Adapter* . Retrieved from Data & Object Factory: http://www.dofactory.com/Patterns/PatternAdapter.aspx

Durandal. (2014, 05 03). *Durandal*. Retrieved from Durandal: http://durandaljs.com/

Firebase. (2014, 05 26). *Overview*. Retrieved from Firebase - Build Realtime Apps: https://www.firebase.com/

Fogg, B. (2009). A behavior model for persuasive design. *Persuasive '09 Proceedings of the 4th International Conference on Persuasive Technology*, Article No. 40.

Gardner, B. S. (2011). Responsive Web Design: Enriching the User Experience. *Connectivity and the User Experience*, 13-20.

GoPivotal, Inc. (2014, 05 26). *Home*. Retrieved from Grails: https://grails.org/

Hill, M. D. (1990). What is scalability? *Computer Architecture News*, 18-21.

Kaazing Corporation. (2014, 04 15). *WebSocket.org*. Retrieved from What is WebSocket?: http://www.websocket.org/quantum.html

Knight, K. (2011). Responsive web design: What it is and how to use it. *Smashing Magazine*.

LeadDyno. (2014, 05 26). *intercooler.js*. Retrieved from intercooler.js - Simple, declarativ AJAX using HTML attributes: http://intercoolerjs.org/

LearnJavaOnline.org. (2014, 05 25). *Welcome to LearnJavaOnline.org*. Retrieved from Learn Java: http://www.learnjavaonline.org/

Leighton, T. (2009). Improving Performance on the Internet. *Communications of the ACM*, 44-51.

Meteor Development Group. (2014, 05 26). *Introduction*. Retrieved from Meteor: https://www.meteor.com/

Michael, M., Moreira, J., Shiloach, D., & Wisniewski, R. (2007). Scale-up x Scale-out: A Case Study using Nutch/Lucene. *Parallel and Distributed Processing Symposium*, 1-8.

Microsoft Corporation. (2014, 05 25). *Try F#*. Retrieved from Try F#: http://www.tryfsharp.org/

Object Management Group. (2014, 08 21). *CORBA FAQ*. Retrieved from CORBA Web Site: http://www.omg.org/gettingstarted/corbafaq.htm

Oracle. (2014, 08 18). *Remote Method Invocation Home*. Retrieved from Oracle: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

OSGi Alliance. (2014, 07 28). *The OSGi Architecture*. Retrieved from OSGi Alliance: http://www.osgi.org/Technology/WhatIsOSGi

Parlante, N. (2014, 05 25). *CodingBat*. Retrieved from CodingBat: http://codingbat.com/

Pivotal Software, Inc. (2014, 05 26). *Introduction*. Retrieved from Spring: http://spring.io/

Pusher. (2014, 05 26). *Introduction*. Retrieved from Pusher | HTML5 WebSocket Powered Realtime Messaging Service: http://pusher.com/

Red Hat. (2014, 05 04). *Hibernate ORM*. Retrieved from Hibernate: http://hibernate.org/orm/

Reilly, D. (2014, 08 22). *Java RMI & CORBA*. Retrieved from Java Coffee Break: http://www.javacoffeebreak.com/articles/rmi_corba/

Ryan, R. M., & Deci, E. L. (2000). Self-determination theory and the facilitation of intrinsic motivation, social development, and well-being. *American Psychologist*, 68-78.

Sellier, A. (2014, 04 26). *Language Features*. Retrieved from Less.js: http://lesscss.org/features/

Sissel, J. (2010, 06 04). *SSL-Latency*. Retrieved from Semicomplete: http://www.semicomplete.com/blog/geekery/ssl-latency.html

Sphere Research Labs. (2014, 07 15). *Introduction*. Retrieved from Sphere Research Labs: http://sphere-research.com/

The Apache Software Foundation. (2014, 08 18). *About Apache River*. Retrieved from Apache River: https://river.apache.org/about.html

The Apache Software Foundation. (2014, 08 21). *Storm*. Retrieved from Storm: http://storm-project.net/

The Eclipse Foundation. (2014, 05 04). *Eclipse Modeling Framework Project (EMF)*. Retrieved from Eclipse: https://www.eclipse.org/modeling/emf/

The Eclipse Foundation. (2014, 07 28). *Embedding an HTTP server in Equinox*. Retrieved from Eclipse: http://eclipse.org/equinox/server/http_in_equinox.php

The Eclipse Foundation. (2014, 05 10). *Orion*. Retrieved from Orion: http://www.eclipse.org/orion/

The Eclipse Foundation. (2014, 07 15). *Platform architecture*. Retrieved from Eclipse documentation: http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Farch.htm

The Eclipse Foundation. (2014, 07 28). *Target Platform*. Retrieved from Help - Eclipse Platform: http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.pde.doc.user%2Fconcepts%2Ftarget.htm

The Eclipse Foundation. (2014, 05 04). *Teneo*. Retrieved from Eclipsepedia: https://wiki.eclipse.org/Teneo

The Eclipse Foundation. (2014, 05 26). *Texo*. Retrieved from Eclipsepedia: http://wiki.eclipse.org/Texo

The jQuery Foundation. (2014, 05 05). *jQuery*. Retrieved from jQuery: http://jquery.com/

The World Wide Web Consortium. (2014, 05 05). *The WebSocket API*. Retrieved from W3C: http://www.w3.org/TR/websockets/

The World Wide Web Consortium. (2014, 05 05). *W3C*. Retrieved from Server Sent Events: http://www.w3.org/TR/eventsource/

Typesafe Inc. (2014, 07 24). *Akka*. Retrieved from Akka: http://akka.io/

Typesafe Inc. (2014, 05 26). *Introduction*. Retrieved from Play Framework: http://playframework.com/

Vaadin Ltd. (2014, 05 26). *Home*. Retrieved from Vaadin: https://vaadin.com/

W3Techs. (2014, 05 05). *W3Techs*. Retrieved from W3Techs: http://w3techs.com/technologies/overview/javascript_library/all

Wikimedia Foundation, Inc. (2014, 05 10). *Comparison of JavaScript-based source code editors*. Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Comparison_of_JavaScript-based_source_code_editors

Wikimedia Foundation, Inc. (2014, 05 26). *Comparison of web application frameworks*. Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks#Java

World Wide Web Consortium. (2014, 04 19). *Usability - ISO 9241 definition*. Retrieved from W3C: http://www.w3.org/2002/Talks/0104-usabilityprocess/slide3-0.html

Åse, D. (2014). *Gamification of Assignment Systems for Programming Courses.* NTNU, Trondheim: Unpublished.

8 - Appendix A – Constraints, requirements and architecture

This chapter contains the constraints, requirements and architecture for the system. First we look at the constraints imposed on the system by NTNU, as these will affect the resulting architecture. Then we look at the system requirements for the system as a whole and the web-based code-editor specifically. Finally we describe the system architecture; which architectural patterns we should use, and how we should use them. Each of the three main processes described in section 3.2 will be modelled in ArchiMate, presenting an overview of the system, and the code-editor architecture will be modelled in detail.

8.1 Implementation constraints

Due to concerns relating to operation and further development, NTNU has placed three constraints on our system.

IC1) The system's server side implementation language shall be Java
IC2) The system shall use EMF for data storage
IC3) EMF shall be used to administer the system (no admin web-view shall be created)

The system designed in the preliminary study featured a web-view for administrators, but the course instructor was clear about wanting to use EMF to administer the system. This is partly because he is very familiar with EMF, but also in order to reduce the amount of code that needs to be maintained if the system is taken into use.

8.2 System requirements

This section contains the requirements for the system that is to be implemented during the course of this thesis. The main goal of the thesis is to implement the system designed in the preliminary study as a web-application with a good code-editor with IDE-functionality, therefore a lot of the requirements will carry over from the preliminary study. A complete list of the requirements from the preliminary study can be seen in section 10.1.

We will first describe the data model and which entities and roles we have in the system. We will then look closer at the specific requirements we have for our web-based code-editor. After this we will look at the non-functional requirements for the system (which are for the most part related to gamification and user-friendliness). The requirements that do not fit any of these categories will then be listed in an "Additional requirements" section, and we will round of the section by talking about the requirements from the preliminary study that have been cut.

8.2.1 Data model

The requirements for the data structure were largely decided in the preliminary study, but they will be modelled here to get a better overview.

The system consists of a collection of assignments which again consist of a collection of problems. These assignments are created by the course staff. Students create submissions for the individual problems, and the system creates an evaluation for the student-submitted solutions. A teaching assistant can override the system created evaluation, allowing for manual evaluation. A model of this data structure can be seen in Figure 8.1.

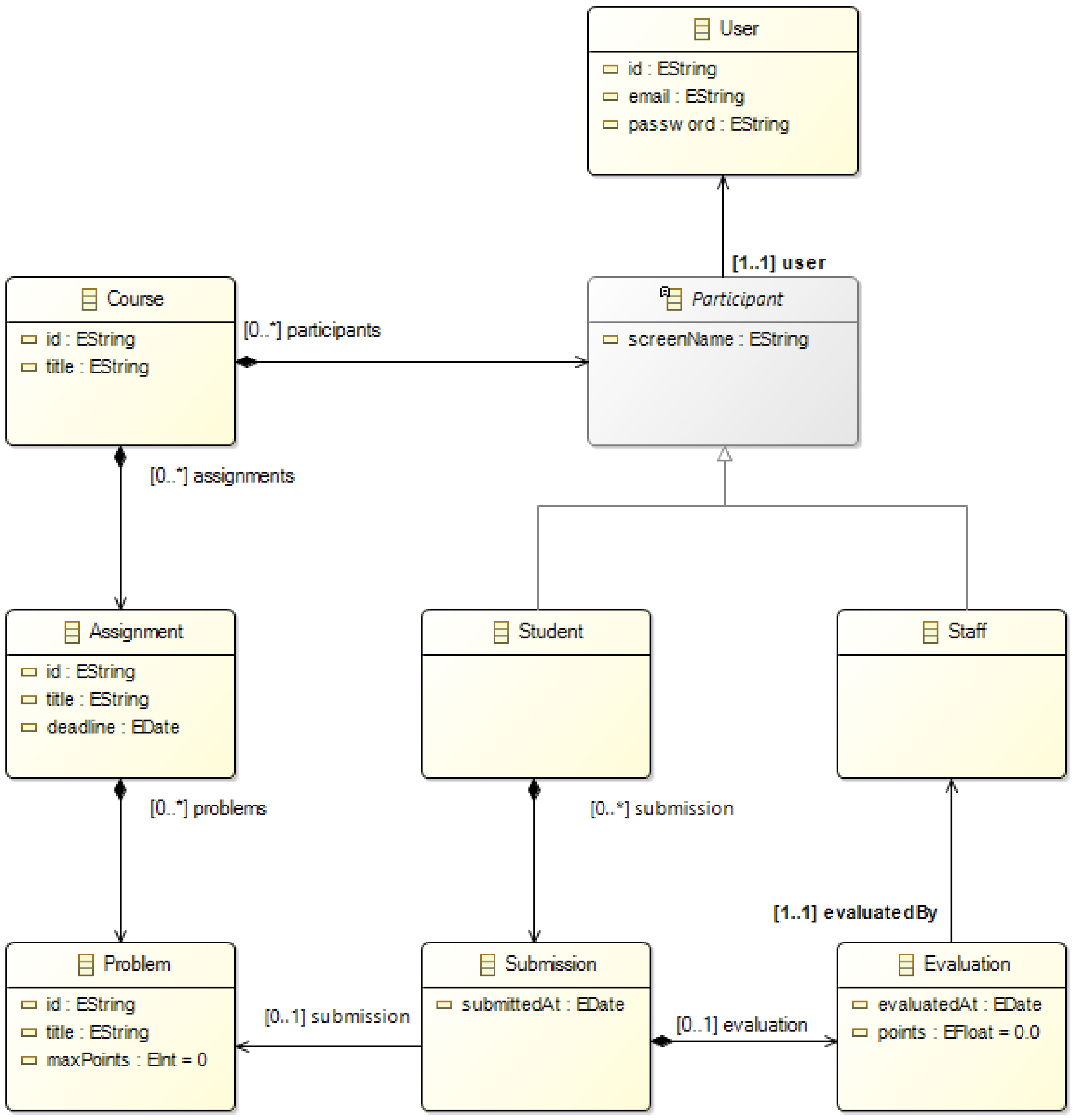


Figure 8.1: Data model requirements

8.2.2 Web-based code-editor

We have mentioned "IDE-functionality" several times throughout this thesis. This section details the specific functionality we want our editor to support. Only major functionality is included in this list, but the editor should also support basic functionality like block-indentation, indentation guides, etc. "Code-editor requirement" is abbreviated to CER.

- CER1) The code-editor shall support Java syntax highlighting
  - CER1.1) Syntax-highlighting shall be instantaneous
- CER2) The code-editor shall support Java error checking
  - CER2.1) Error checking shall take no longer than one second
- CER3) The code-editor shall support Java code completion
  - CER3.1) Code completion shall take no longer than one second
- CER4) The code-editor shall support multiple Java classes
- CER5) The code-editor shall be updated automatically and asynchronously
- CER6) The code-editor shall be maximizable
- CER7) The code-editor shall be ready to use in less than fifteen seconds

The requirements are what we, in discussion with the course responsible, decided to be the most important IDE-features that could realistically be implemented in a web-based code-editor. The last requirement, CER7 was decided based on how long it takes to start eclipse (which is around 10 seconds on a high end gaming PC with an SSD).

8.2.3 Non-functional requirements

The first two requirements are the same as they were in the preliminary study (shown in section 10.1.2), but one more requirement has been added. "Non-functional requirement" is abbreviated to NFR.

- NFR3) The system shall utilize responsive design
  - NFR3.1) The system shall have views that adapt to device size
    - NFR3.1.1) The views shall support laptop, tablet and cellphone sizes

This non-functional requirement (NFR3) was added to further the competition between the most eager students, thereby increasing the students' motivation and hopefully grades. Having a system with a responsive design will allow the students to easily check the leaderboards or track their progress from their cellphone or tablet.

8.2.4 Additional requirements

This section covers additional requirements to the system that is not covered in the other sections. ”Additional requirement” is abbreviated to AR.

AR1) The system shall provide assignment feedback to users

AR1.1) The system shall display unit test results from the code editor

AR2) JUnit shall be used to test student-submitted code

8.2.5 Requirements from the preliminary study that were cut

The requirements concerning quiz functionally have been cut from the original requirements-list, as the main focus of this thesis is to create a solid web-based code-editor for solving Java assignments. Implementing quiz functionality is important to the system, and it should indeed be included in the finished product, but it is technically trivial and we have limited time available, so implementing it will not be a part of this thesis.

The requirement concerning code-quality-analysis has been cut. While the feature is certainly useful, we feel that the limited time available is better spent at getting basic IDE-functionality right, instead of tacking on additional plugins.

No other requirements have been cut.

8.3 System architecture

This section contains the system architecture needed to fulfill the requirements and realize the system. First, we will look at the architectural patterns we need to realize the architecture, then we will model the architecture more in depth using the ArchiMate specification.

8.3.1 Architectural patterns

The system is a web-application where multiple users interact with a centralized server that evaluates, stores, and displays user dependent content. This largely decides the architecture for us: we have to use the Client-Server pattern and some sort of Model-View-* pattern.

The Client-Server pattern

The system we are implementing is a very typical client-server application, leaving no real alternatives to this pattern. Requests are sent over the internet to a server where the user’s code is executed and evaluated. The server then responds back to the clients. An illustration of how this works can be seen in Figure 8.2.

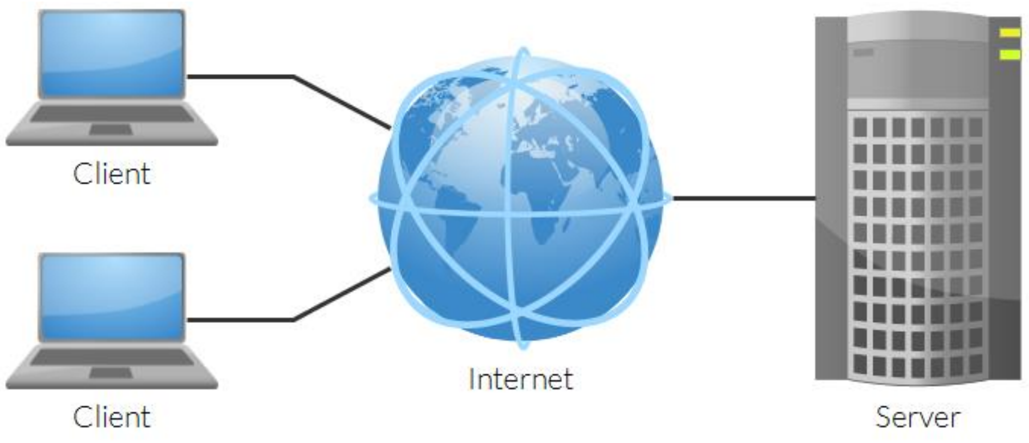


Figure 8.2: The Client-Server pattern

The Model-View-* pattern

Our application would have used the classic MVC pattern had it been a native application, but bringing it to the web complicates things slightly. The basic MVC pattern as used in Java or other languages can be seen in Figure 8.3. However, in a typical web-applications the view is usually a static HTML document, and therefore it cannot observe the model in the same way a Java or other native view would. All communication between the view and the model goes through the web server and the controller. How this works can be seen in Figure 8.4.

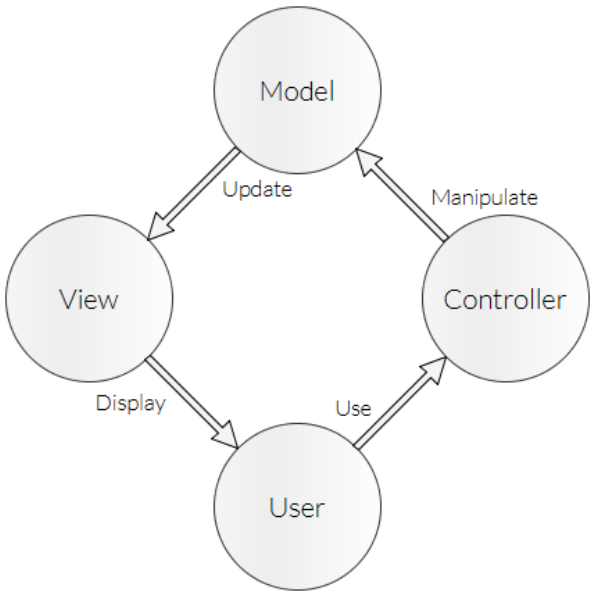


Figure 8.3: The classic MVC pattern

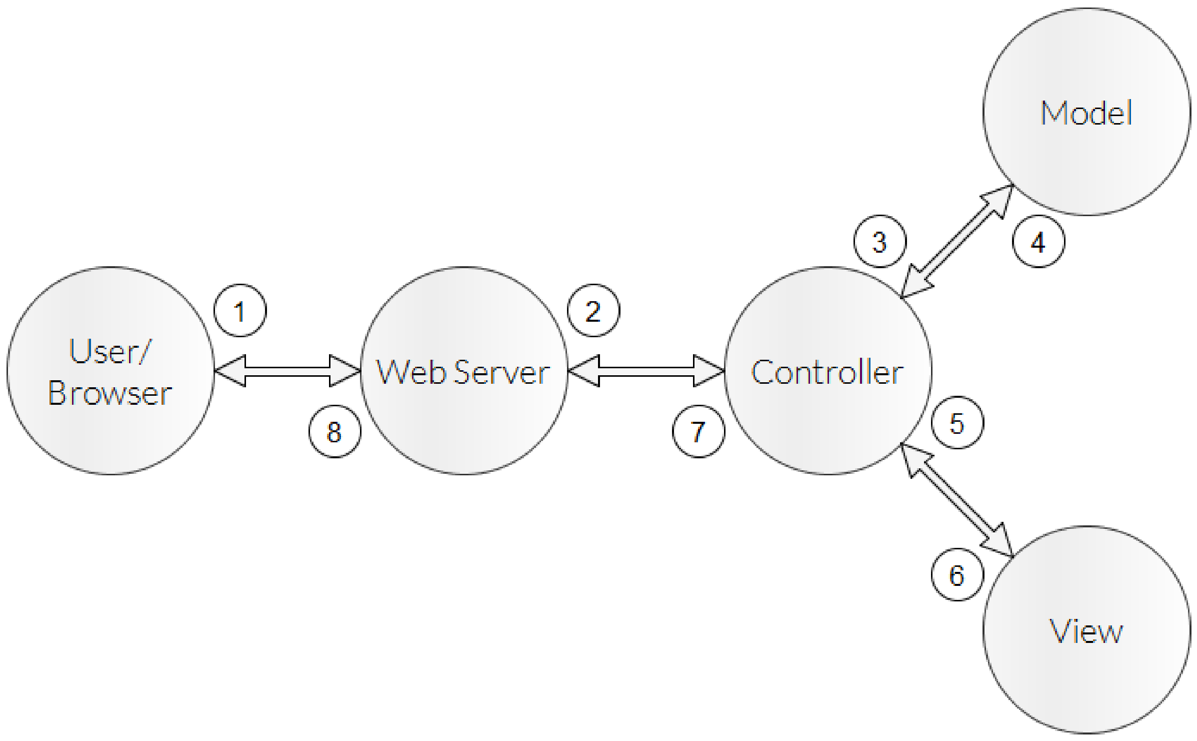


Figure 8.4: Typical web-MVC pattern

The process flow of Figure 8.4 can be broken into eight different steps:

1. The user sends a request to the web server via the browser
2. The web server processes the request and dispatches it to the appropriate controller method, based on which route was used
3. The controller queries or updates the model in order to complete the request
4. The model complies to the request from the controller and returns a result
5. The controller feeds the appropriate information to the view and asks it to render
6. The view renders itself and gives the rendered HTML to the controller
7. The controller assembles the total page's HTML and gives it to the web server
8. The web server returns the page to the browser, which renders it to the user

There are many slight variations of how this pattern works, and which one we will end up using depends largely on which web-server and framework we will choose to build the system with.

8.3.2 System architecture in ArchiMate

This section describes the system architecture using the ArchiMate[15] specification. We will focus on the processes described in section 3.2. We will include the implementation constraints where applicable.

Publish assignment architecture

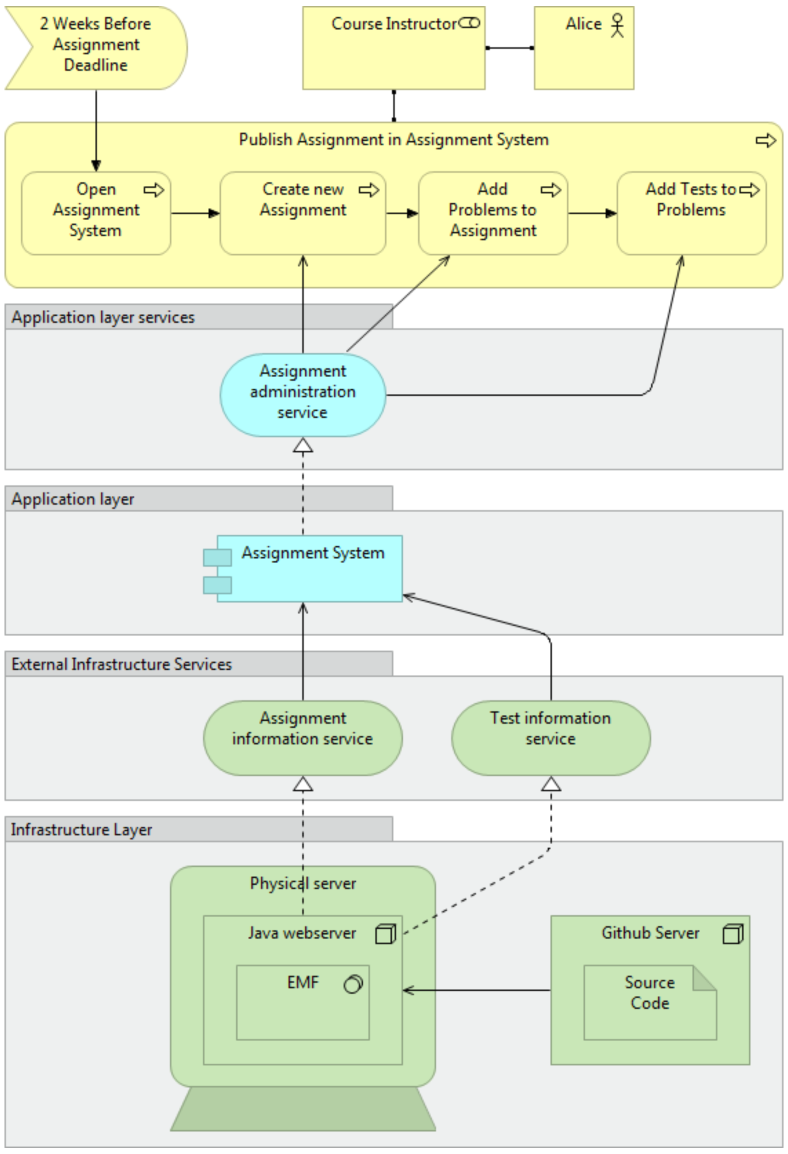
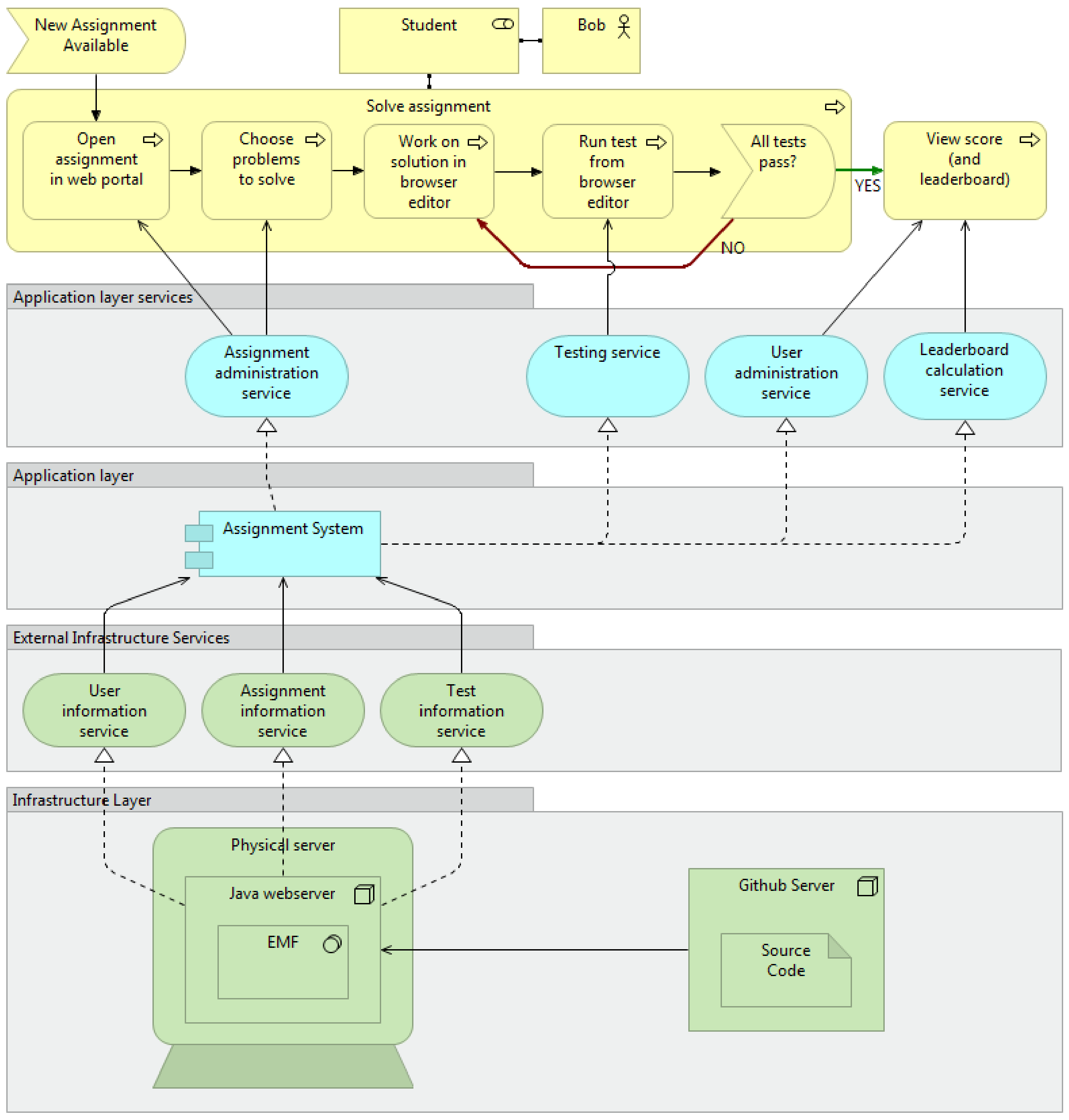

2 Weeks Before Assignment Deadline
Course Instructor
Alice
Publish Assignment in Assignment System
Open Assignment System
Create new Assignment
Add Problems to Assignment
Add Tests to Problems
Application layer services
Assignment administration service
Application layer
Assignment System
External Infrastructure Services
Assignment information service
Test information service
Infrastructure Layer
Physical server
Java webserver
EMF
Github Server
Source Code

Figure 8.5: Publish assignment architecture

Figure 8.5 shows the architecture needed to realize the publishing of assignments. The assignment system uses information services for both assignments and tests, and offers an assignment administration service. The information handling is taken care of by EMF.

[15] Available at: http://www.opengroup.org/subjectareas/enterprise/archimate

Solve assignment architecture



New Assignment Available
Student
Bob
Solve assignment
Open assignment in web portal
Choose problems to solve
Work on solution in browser editor
Run test from browser editor
All tests pass?
YES
NO
View score (and leaderboard)
Application layer services
Assignment administration service
Testing service
User administration service
Leaderboard calculation service
Application layer
Assignment System
External Infrastructure Services
User information service
Assignment information service
Test information service
Infrastructure Layer
Physical server
Java webserver
EMF
Github Server
Source Code

Figure 8.6: Solve assignment architecture

Figure 8.6 shows the architecture needed to realize the solving of assignments. EMF handles information about users, assignments and tests, and the assignment system offers services for administering a user assignment, testing and leaderboards calculations.

Approve assignment architecture

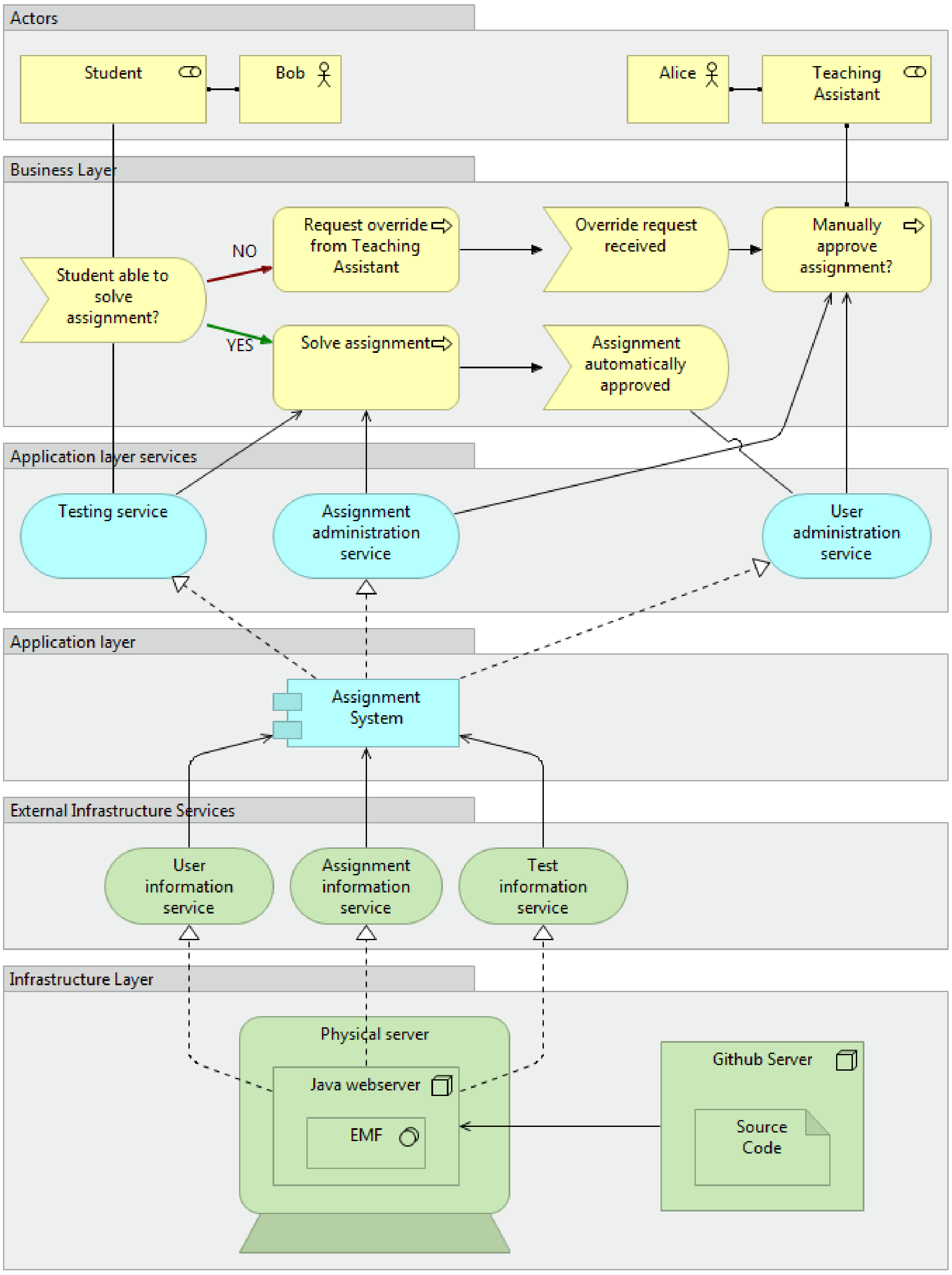

Actors
Student
Bob
Alice
Teaching Assistant
Business Layer
Student able to solve assignment?
NO
YES
Request override from Teaching Assistant
Override request received
Manually approve assignment?
Solve assignment
Assignment automatically approved
Application layer services
Testing service
Assignment administration service
User administration service
Application layer
Assignment System
External Infrastructure Services
User information service
Assignment information service
Test information service
Infrastructure Layer
Physical server
Java webserver
EMF
Github Server
Source Code

Figure 8.7: Approve assignment architecture

Figure 8.7 shows the architecture needed to realize the approving of assignments. It utilizes the same services as the solving process, except it does not need the leaderboard calculation service.

9 - Appendix B – Implementation

This chapter contains the reasoning behind choosing the technology stack to use when implementing our system. We will start at the lowest level of the backend (data storage) and move sequentially to the highest level of the frontend (CSS).

9.1 Data storage

Because of maintainability and operational concerns, the course responsible wanted us to use the Eclipse Modeling Framework (EMF) for data storage. The course instructor has extensive experience with this tool.

EMF is a tool for modeling software (The Eclipse Foundation, 2014). The model can be created using UML diagrams, XML markup or Java code. The software package “Eclipse Modeling Tools” includes graphical tools to create a model. We used these tools to create the model shown in section 0.

EMF does not enforce any specific persistence technology. By default, it can load and save data to XML files. To add a specific persistence technology, we will use Teneo (The Eclipse Foundation, 2014). This technology was recommended to us by the course responsible.

Teneo is built on top of Hibernate and it integrates well with EMF. Hibernate is an object-relational mapping (ORM) library and it supports multiple relational database systems, such as HSQLDB, MySQL (Red Hat, 2014)

To maintain data consistency in the model and to make it easier to retrieve data from the model, we will make a services layer, a data access object (DAO), on top of the model. To prevent this layer from exposing its underlying model, we will instead expose the data in separate data transfer objects (DTO).

9.1.1 Querying the data model

To query the data model we looked at several options:

1. EMF Query
2. EMF OCL Query (EMF Object Constraint Language Query)
3. EMF-IncQuery
4. HQL (Hibernate Query Language)
5. XPath
6. Manual traversal
7. EMF `Resource.getEObject()` method

Because the system will be quite small (storing no more than 10,000 objects), we will not evaluate the performance of each approach. Instead we will focus on which approach that is the most easy-to-use.

EMF Query, EMF OCL Query and EMF-IncQuery are all part of the EMF project umbrella. It would be natural to think that those libraries are the best way to query the data model. Unfortunately, the first two libraries have an unwieldy API while the latter uses a domain specific language (DSL).

HQL (similar to SQL) and XPath are third party libraries that have an easier API than the EMF libraries. However, they are not usable in all circumstances. HQL must be used in conjunction with the Teneo library, which means that we need to have a relational database as the backing store. While we develop the system it is easier to use the XMI-format defined by EMF. XPath, on the other hand, will only work on the XML-document. Thus, we can't change the backing store.

Another option is to manually traverse the object graph by using the getter methods. This is easy if you only need to access the immediate children of an entity, but it gets harder (and slower) if you need to access the children's children or access objects even deeper into the object graph. In addition to traversing the object graph, it is possible to get a reference to a specific object using its ID. To retrieve a specific object you can use the `getEObject()` method which is available on the `Resource` class.

Conclusion

To query the data model we elected to combine manual traversal of the object graph and the EMF `Resource.getEObject()` method. We elected these methods because they complement each other well, i.e. the `Resource.getObject()` method can fetch an object, which will be the starting point, and then we can use manual traversal to get the desired objects. This combination will also work for both for XML-documents and for relational databases.

9.1.2 Choice of ID

To be able to retrieve objects from the data model, we must be able to reference the objects by some kind of ID. All objects in EMF have an inherent ID, which is based on its position in the object graph, e.g.: `//@courses.0/@assignments.0/@problems.0`. However, these IDs quickly become long and unwieldy.

Instead of using the implicit IDs we can model the IDs explicitly. An ID must be unique in a given context. We are defining the contexts as follows:

- Local context – IDs are unique for a given collection of objects

- Entity context – IDs are unique for a given entity type
- Global context – IDs are globally unique

We must evaluate how IDs assigned in these contexts affect the implementation of the system. To evaluate the approaches we will focus on the following criteria:

- What the resulting URL for the system will be (the ID must be a part of the URL to identify what information we are requesting)
- How easy it is to use the services API
- How much work it requires to implement the solution

Local context

Pros:

- Will result in the most user-friendly URLs:
  `/assignments/1/problems/a`
  I.e. the IDs reflect the actual assignment ID and problem ID

Cons:

- Methods need multiple ID-parameters:
  `updateSourceCodeFile(assignmentId, problemId, fileId, sourceCode)`
- Must implement custom logic to query for the given IDs

Entity context

Pros:

- Will result in short URLs:
  `/assignments/3/problems/9`
- Methods in services only need a single ID-parameter:
  `updateSourceCodeFile(fileId, sourceCode)`

Cons:

- Must implement custom logic to handle IDs per entity (must consider thread safety)
- Must implement custom logic to query for the given ID

Global context (using UUID)

Pros:

- Easy to generate ID:
  `EcoreUtil.generateUUID()`
- Methods only need a single ID-parameter:
  `updateSourceCodeFile(fileId, sourceCode)`
- Easy to query database:
  `resource.getEObject(id)`

Cons:

- Will result in long URLs:
  `/assignments/_EU_5wNyCEeOKIfV4FdrdrA/problems/_xkgyENyIEeOuDKXZBf5e3g`

Global context (using incremental ID)

Pros:

- Will result in short URLs:
  `/assignments/32/problems/57`
- Methods only need a single ID-parameter:
  `updateSourceCodeFile(fileId, sourceCode)`
- Easy to query database:
  `resource.getEObject(id)`

Cons:

- Must implement custom logic to handle the current global ID (must consider thread safety)

Conclusion

The two approaches that are based on assigning IDs in a global context are the easiest to implement. The main difference between them are the actual IDs they use; the *UUID*-approach has a 23 character long ID, while the *incremental ID*-approach has a number as the ID (starting from 1). Because we want the URL to be as simple as possible, we will go for the latter solution (using incremental ID).

9.1.3 The EMF model

Figure 9.1 shows the complete EMF model needed to realize the system.

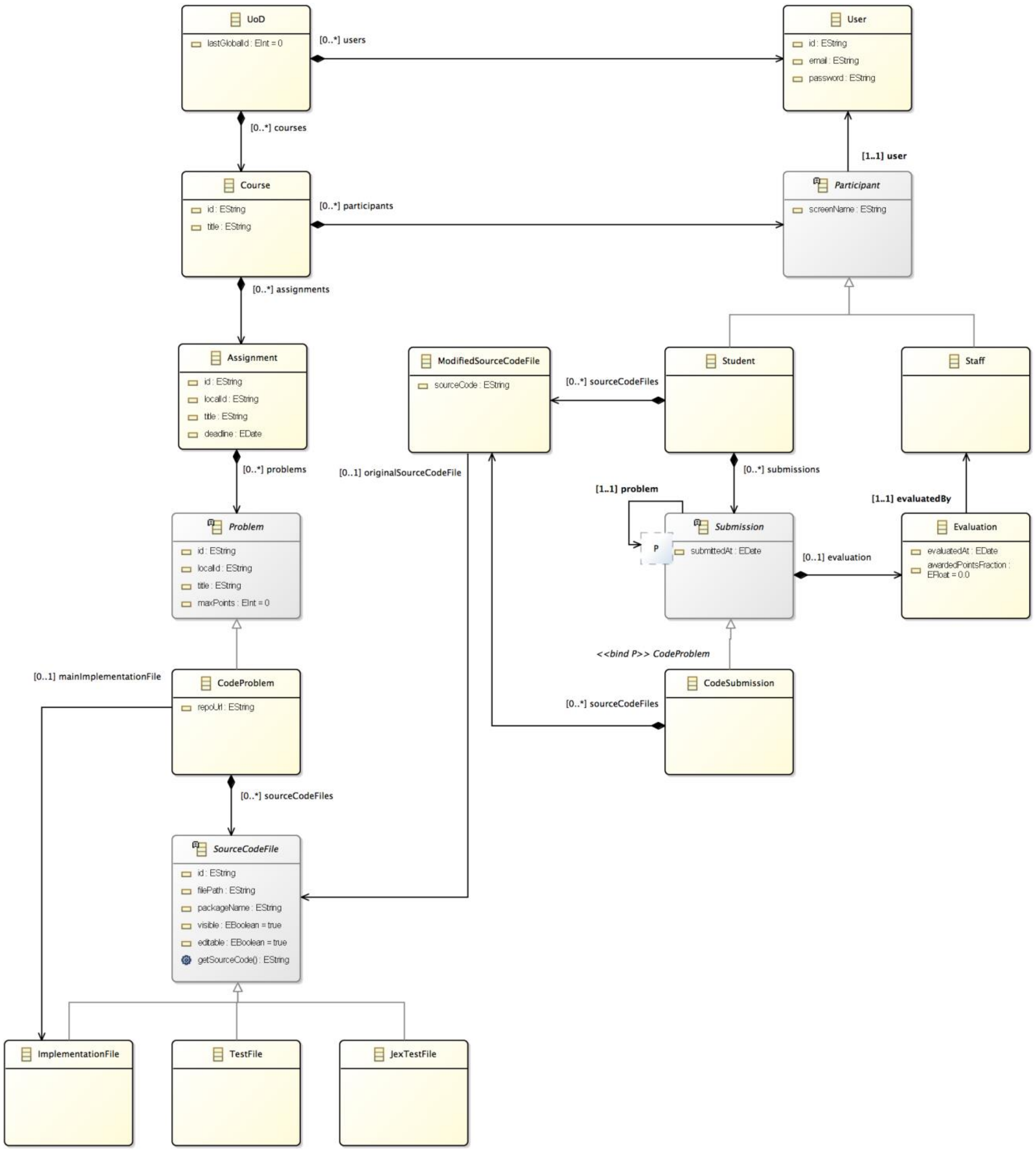


Figure 9.1: EMF model

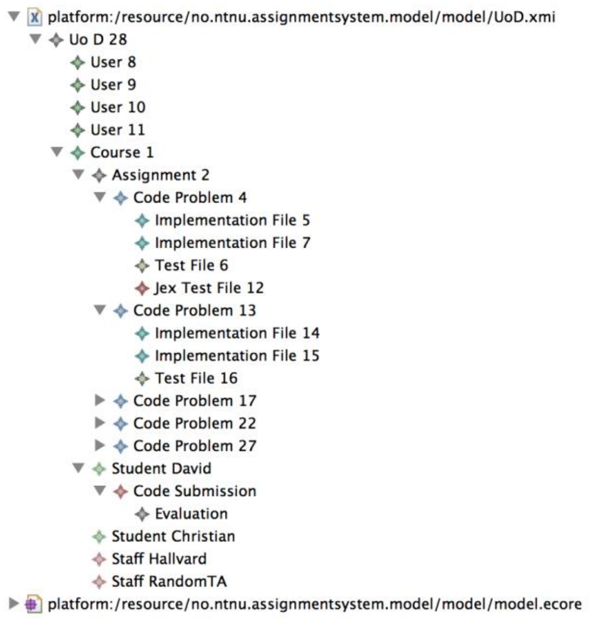
Validating the model

The system has to support the functionality of the prototype proposed in the pre-study. To ensure this, one can create a dynamic instance in EMF, which makes it possible to test different use cases. This section contains a checklist with all the required use cases, along with a screenshot of the finished dynamic instance.

Use cases:

1. The administrator wants to create a staff-member: Hallvard.
2. The administrator wants to create two students: Christian and David.
3. The administrator wants to create a staff-member: Random TA.
4. Hallvard wants to create a course: TDT4100.
5. Hallvard wants to add an assignment to TDT4100: "Assignment 1 – The basics".
6. Hallvard wants to add a code problem to the assignment: "Problem 1 – Hello World".
7. Hallvard wants to add source code to the code problem.
8. David wants to submit his solution to the code problem in "Assignment 1".
9. Hallvard wants to approve David's solution to "Problem 1 – Hello World".

Figure 9.2 shows the dynamic instance after applying the use cases.



```
▼ platform:/resource/no.ntnu.assignmentsystem.model/model/UoD.xmi
  ▼ Uo D 28
      User 8
      User 9
      User 10
      User 11
    ▼ Course 1
      ▼ Assignment 2
        ▼ Code Problem 4
            Implementation File 5
            Implementation File 7
            Test File 6
            Jex Test File 12
        ▼ Code Problem 13
            Implementation File 14
            Implementation File 15
            Test File 16
        ▶ Code Problem 17
        ▶ Code Problem 22
        ▶ Code Problem 27
      ▼ Student David
        ▼ Code Submission
            Evaluation
        Student Christian
        Staff Hallvard
        Staff RandomTA
▶ platform:/resource/no.ntnu.assignmentsystem.model/model/model.ecore
```

Figure 9.2: Dynamic instance for validating the model

Conclusion

The model contains "one of everything" of the data we need to realize the system, and validated without any problems.

9.2 Web server

The main responsibilities of the web server is to supply web content to the users and to process their submissions.

The web server should:

- Serve web content
- Authenticate the user
- Validate the input from the users
- Communicate with the services layer in the data storage

With EMF as the data storage, there are at least two approaches to structure the web server; using specialized libraries to extend EMF, or using a generic web framework as a layer on top of EMF.

9.2.1 Specialized libraries for EMF

A specialized library will be able to expose the data in the data model directly as a web service. The client can then consume the data as necessary. This approach is easy to get up and running, but it has some flaws:

- Because the full data model is exposed, the client is responsible for data consistency
- Clients might be able to read more data than allowed, e.g. user data

This approach is feasible in cases where the EMF data model is consumed by a trustworthy client. Thus, the web services should not be made available directly on the internet.

Example of specialized libraries for EMF are Texo and EMF-REST.

Texo

Texo is a library that can generate a web service based on an EMF model (The Eclipse Foundation, 2014). It also supports generating code from the EMF model to be able to interface with other systems, like Google Web Toolkit and ORM solutions. The web service supports both JSON and XML as data exchange formats.

EMF-REST

EMF-REST is a library that can generate a web service based on an EMF model (Camilo, Javier, & Jordi, 2014). The web service follows the principles of REST (Read Roy T. Fielding's dissertation for more information[16]). EMF-REST also includes a tailor-made JavaScript library that makes it easy to fetch the data from the REST API. The REST API only supports JSON as the data exchange format.

[16] Available at: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

9.2.2 Generic web frameworks

There are a ton of Java-based web frameworks. Wikipedia has an up-to-date overview of many popular web frameworks, shown in Table 9.1 (Wikimedia Foundation, Inc., 2014).

Project	Current stable version	Release date	License
Grails	2.4.0	5/21/2014	Apache
Spring	4.0.5	5/20/2014	Apache 2.0
4WS.Platform	2.1.6	5/7/2014	LGPL
Play!	2.2.3	5/1/2014	Apache 2.0
VRaptor	4.0.0.Final	4/23/2014	Apache 2.0
JavaServer Faces (Mojarra)	2.2.6	3/4/2014	CDDL, GPL 2, Apache 2.0,
Google Web Toolkit	2.6.0	1/30/2014	Apache 2.0
Apache Wicket	6.13.0	1/14/2014	Apache 2.0
Apache Struts 2	2.3.16	12/8/2013	Apache 2.0
Vaadin	7.1.9	12/4/2013	Apache 2.0
Apache OFBiz	12.04.02	7/30/2013	Apache Software License 2.0 (ASL 2.0)
Oracle ADF	12.1.2.0	7/11/2013	Oracle Technology Network Developer License
Apache Tapestry	5.3.7	4/24/2013	Apache
OpenXava	4.7	4/2/2013	LGPL
ZK	6.5.2	3/26/2013	LGPL, ZOL
Wavemaker	6.5.3	3/14/2013	Apache
Jspx-bay	1.2	2/14/2013	Apache 2.0
Eclipse RAP	2	2/11/2013	Eclipse
JVx	1.1	1/23/2013	Apache 2.0
Stripes	1.5.7	5/17/2012	Apache
JBoss Seam	3.1.0 final	1/13/2012	LGPL
ztemplates	2.4.0	9/11/2011	Apache
ItsNat	1.2	5/24/2011	LGPL, proprietary
FormEngine	2.0.1	5/8/2011	Proprietary
Apache Sling	6	4/18/2011	Apache 2.0
AppFuse	2.1	4/4/2011	Apache
Apache Click	2.3.0	3/27/2011	Apache Software License 2.0 (ASL 2.0)
Hamlets	1.7	3/11/2011	BSD
OpenLaszlo	4.9.0	10/21/2010	CPL
WebObjects	5.4.3	9/15/2008	Proprietary
Apache Shale	1.0.4 (Retired)	12/19/2007	Apache
ThinWire	1.2	9/17/2007	GPL
WebWork	2.2.6	7/21/2007	Apache
RIFE	1.6.1	7/14/2007	CDDL, LGPL

Table 9.1: List of Java web frameworks (as of 5/26/2014)

To decide which library to use, we will consider the following factors:

1. Be under active development (latest release should be less than 6 months olds)
2. Use an open source license
3. Popular relative to the others
4. Use Java as the programming language
5. Fulfill our requirements for the web server

When we remove all the web frameworks that have not been updated in the last 6 months, we end up with 10 web frameworks (marked with green background color in Table 9.1). All of these web frameworks use an open source license.

To further reduce the number of choices, we will look at the relative popularity between the web frameworks. Figure 9.3 and Figure 9.4 shows the relative popularity between the web frameworks. Figure 9.5 shows the relative popularity between the most popular web frameworks.

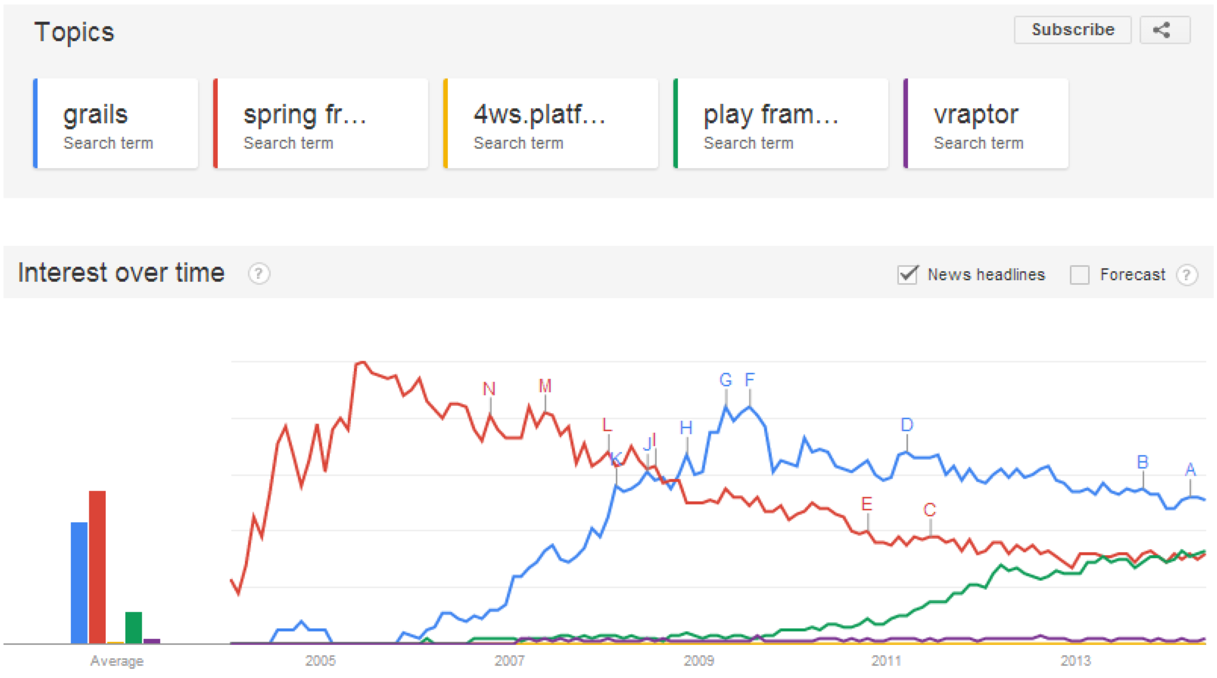


Figure 9.3: Relative popularity between 5 web frameworks

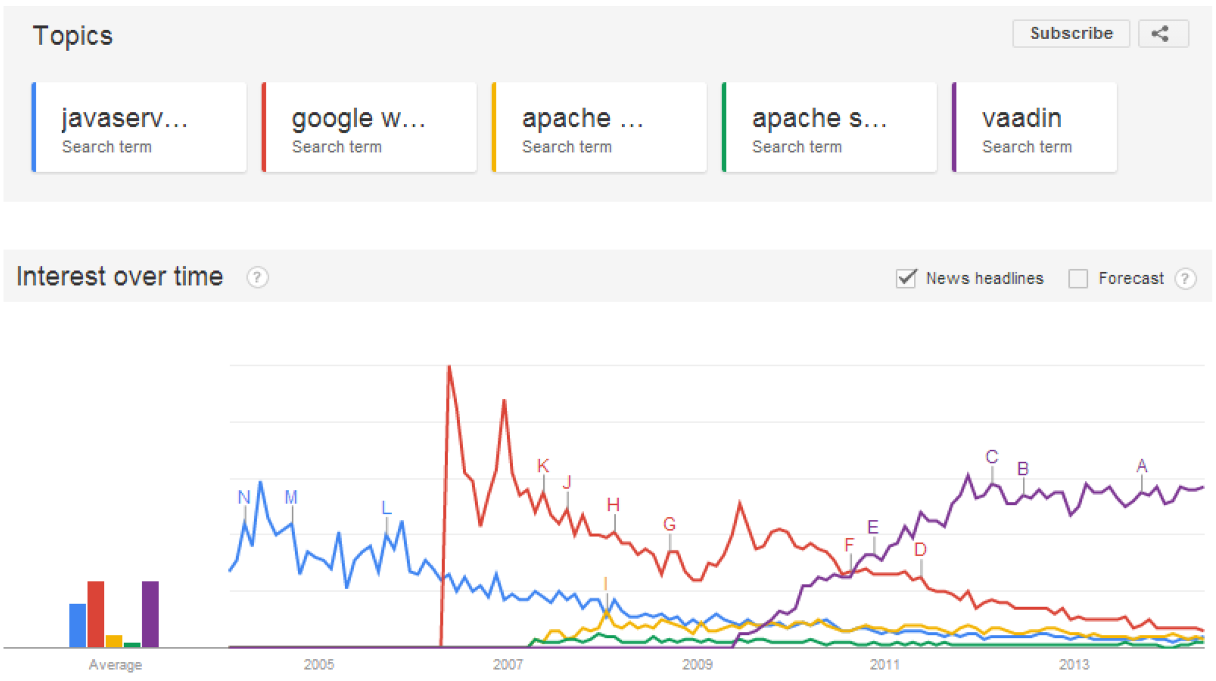


Figure 9.4: Relative popularity between 5 more web frameworks

Figure 9.5: The relative popularity between the 4 most popular web frameworks

Based on the relative popularity between the web frameworks, we will look closer at:

- Grails
- Spring Framework
- Play Framework
- Vaadin

Grails

Grails is the most popular choice of the web frameworks, but it turns out that Grails is based on the Groovy programming language (GoPivotal, Inc., 2014). This means that Grails does not meet the previously listed criteria and that it will not be considered any further.

Spring Framework

The Spring Framework has been around since 2002, thus it should be quite mature. The creators of Spring Framework has the following to say about the framework (Pivotal Software, Inc, 2014):

"Let's build a better Enterprise. Spring helps development teams everywhere build simple, portable, fast and flexible JVM-based systems and applications."

Pros:

- Mature framework
- Integrates well with Eclipse-projects
- Supports JSP (and can use other template engines as well)
- Easy to set up authentication (session based authentication, OpenID-login as well as other authentication providers)
- Integrates well with a vast amount of libraries (database adapters, authentication providers, etc.)

Cons:

- Popularity is declining
- Routes are scattered across multiple classes
- Cumbersome to set up the development environment

Play Framework

The Play Framework is a newer web framework than Spring Framework, but it has quickly become a popular framework. The framework supports both Java and Scala and you can even have both languages in the same project. The creators of Play Framework are calling it (Typesafe Inc, 2014):

"The High Velocity Web Framework for Java and Scala"

Pros:

- Popularity has an upward trend
- Integrates well with Eclipse-projects
- Flexible routing system
- Simple and powerful template system
- Easy to set up session based authentication
- Makes development a breeze
  - Easy to get started (`$ play new`)
  - Recompiles on reload
  - Supports pre-compilation and minification of assets (LESS, CoffeeScript, etc.)
- Supports reactive constructs (promises, etc.)

Cons:

- Uses Scala tool chain and configuration files

Vaadin

Vaadin is a web application framework, meaning it is intended for building large-scale web application in contrast to basic web sites. The creators of Vaadin has the following to say about the framework (Vaadin Ltd., 2014):

"Vaadin is a Java framework for building modern web applications that look great, perform well and make you and your users happy."

Pros:

- Popularity has an upward trend
- Can build websites solely with Java (no need for HTML/CSS/JavaScript)
- Easy to build large web applications

Cons:

- Intended for building web applications
- Hard to customize components
- Must learn concepts that are specific to the framework (data source, etc.)

9.2.3 Conclusion

Common for all the frameworks are big communities and good documentation, due to their popularity.

The Vaadin framework is intended for building large-scale web application and it is quite heavy-weight in that respect. Because the system is just a basic web site, Vaadin is probably not a good fit.

Spring Framework and Play Framework are similar in many respects and both frameworks fulfill the criteria for the web server. Because we (the authors) have prior experience with Play Framework we will choose this framework to implement the web server.

9.3 Communication

In this section we will look at how to update the contents of the web page in real-time and how to communicate with the Eclipse application.

9.3.1 Real-time updating of the web page

To make the system more user-friendly, and to increase the competition between the students, we will implement real-time updating of the web page. The content that will be updated automatically includes the leaderboard as well as live error checking and console output from the editor.

Polling

It may be dubious to call polling a real-time technology. However, what we mean by real-time is simply that the user does not need to update the web page manually. In this respect, polling might be suitable as long as we do not require quick updates.

The polling mechanism is the easiest approach to achieve real-time updating of content. It does not require any special code on the server. The obvious disadvantage is that the server will incur a higher load than necessary.

Polling can be implemented using JavaScript and by sending regular AJAX calls to the server. There also exist JavaScript libraries to make it even easier to implement polling. One such library is intercooler.js, which will let you specify the source URL and polling interval inside the HTML markup (LeadDyno, 2014).

Long-polling (Comet)

Long-polling is similar to the polling mechanism as both mechanisms will open new connections to the server intermittently. The difference is that long-polling will try to keep the connection to the server alive as long as possible. To keep the connection alive longer the server can send a "ping"-signal (usually a whitespace). Whenever the connection is closed, the client side JavaScript will re-open a new connection.

Server Sent Events (SSE)

SSE is a specification to allow the server to send push notifications (events) to the client (The World Wide Web Consortium, 2014). These notifications are sent over a normal HTTP connection. To open an SSE connection, the browser must send an HTTP request that includes the following header: `Accept: text/event-stream`.

SSE only supports one-way communication (simplex) which means that only the server can communicate with the client and not the other way around. It does however support automatic reconnection.

SSE has a very easy JavaScript API and it is trivial to implement SSE on the server.

WebSocket

Similar to SSE, the WebSocket protocol also maintain an open connection to the server (The World Wide Web Consortium, 2014). A big advantage with WebSocket is that it supports bi-directional communication. The protocol is a more low-level protocol than SSE, as it only exposes two data streams and it is up to the developer to define a communication protocol between the server and the client.

There are numerous libraries that seek to make it easier to use WebSocket. One such library is Pusher (Pusher, 2014). There are also complete web frameworks, e.g. Meteor.js (Meteor Development Group, 2014), and even database systems, e.g. Firebase (Firebase, 2014), that heavily rely on WebSocket.

Conclusion

As noted earlier, we want to support real-time updating of the leaderboard as well as live error checking and console output from the editor.

The leaderboards are not something that needs to be updated very often, but we want to provide real-time updating as convenience. We will implement this feature using polling because that is the simplest solution and it ought to be sufficiently fast.

The editor will require bi-directional communication with the server (to send updated source code and to retrieve console output). In practice, this leaves us with a single choice: WebSocket.

9.3.2 Communication with the Eclipse application

As the Eclipse application runs in a separate process we need to perform inter-process communication. We will look at the following options:

- Java RMI (Java Remote Method Invocation)
- Jini
- CORBA (Common Object Request Broker Architecture)
- Akka
- Storm

Java RMI (Java Remote Method Invocation)

Java RMI is an object oriented equivalent of Remote Procedure Calls (RPC). Java RMI was developed by Sun and it was released with the Java 2 SDK (Oracle, 2014). A typical implementation model of Java RMI is shown in Figure 9.6.

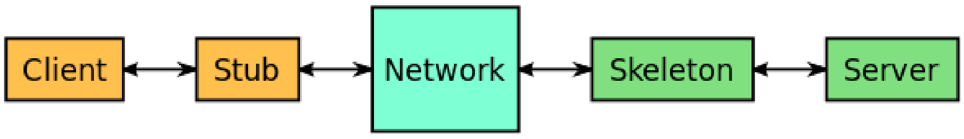


Figure 9.6: A typical implementation model of Java RMI[17]

The Java RMI is easy to use because it hides the complexity behind a proxy object, called a stub. The stub's methods are called using normal method invocation. The Java RMI has support for sending executable code to remote systems. This feature is very powerful, but it also represents a potential security threat. A drawback with Java RMI is that it only supports communication with other systems running in a JVM.

Jini

Jini is a more advanced version of the Java RMI. Jini was originally developed by Sun, but the project was later transferred to Apache and renamed to Apache River (The Apache Software Foundation, 2014).

CORBA (Common Object Request Broker Architecture)

CORBA is a language-agnostic alternative to Java RMI. CORBA was developed by the Object Management Group (Object Management Group, 2014). An example of how a request is sent from the client to the server (called servant in the figure) is shown in Figure 9.7.

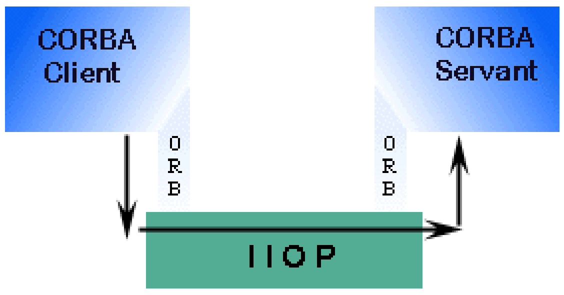


Figure 9.7: A client sends a request through its local Object Request Broker (ORB) and to a remote ORB's servant[18]

The services are described by interfaces, written in the Interface Definition Language (IDL). The IDL has mappings to many common programming languages. CORBA is not as powerful as Java RMI because it does not allow executable code to be sent to remote systems. This does, however, make CORBA safer to use (Reilly, 2014).

[17] Source: http://upload.wikimedia.org/wikipedia/commons/thumb/b/ba/RMI-Stubs-Skeletons.svg/600px-RMI-Stubs-Skeletons.svg.png
[18] Source: http://www.javacoffeebreak.com/articles/rmi_corba/corba.gif

Akka

Akka was developed by Typesafe Inc, and they describes Akka as (Typesafe Inc, 2014):

Akka is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on the JVM.

Akka is based on the actor model. An actor is usually just a separate thread, but it can also be another process or even another node. Messages are sent asynchronously to and from actors as objects, instead of normal method invocation. These objects can be serialized and thus sent over the network.

Akka includes a package that integrates well with OSGi.

Storm

Storm was originally developed by BackType, but is now maintained by The Apache Software Foundation (ASF). ASF describes Storm as (The Apache Software Foundation, 2014):

Apache Storm is a free and open source distributed realtime computation system. Storm makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing.

While it is possible to use Storm for inter-process communication, it is mainly used for processing of big data across multiple nodes.

Conclusion

All of the alternatives that we looked at could work in our case. However, we decided to go with Akka because it is included in the Play Framework (which we chose as the web server framework in section 9.2) and it can easily be used in conjunction with OSGi.

9.4 JavaScript frameworks and libraries

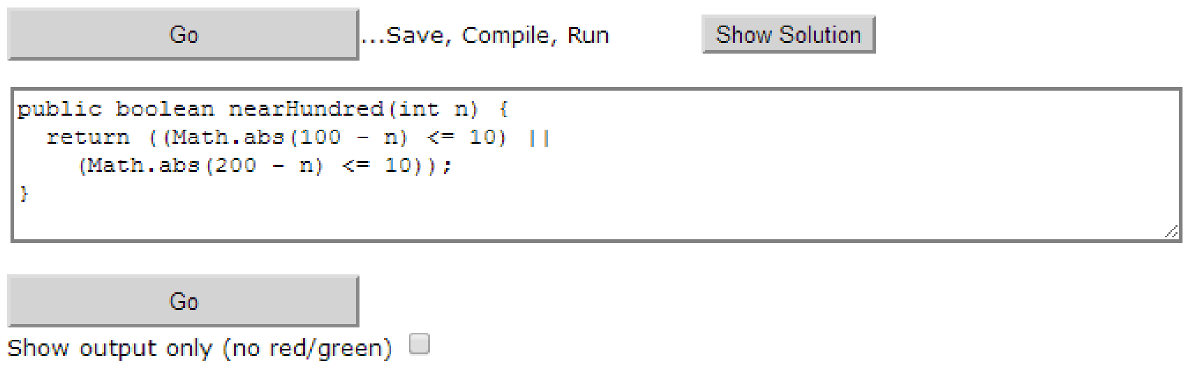
This section contains an overview and evaluation of different JavaScript frameworks and libraries. Only the biggest and most popular libraries will be considered.

9.4.1 Why does this project need JavaScript?

First things first. Why does this project even need JavaScript? The main focus of this project is to create a web-based code editor in which students of TDT4100 can solve their weekly java programming assignments. A standard webpage utilizing only HTML is static and stateless, and requires the user to explicitly perform requests and then wait for the server to respond. With JavaScript, requests can be sent asynchronously in the background, without the user knowing about it, and the webpage can be updated accordingly. Since JavaScript is executed on the client side, it can also provide features like syntax highlighting on the fly.

Let us look at the difference between a pure HTML editor and a JavaScript-powered editor.

Pure HTML editor

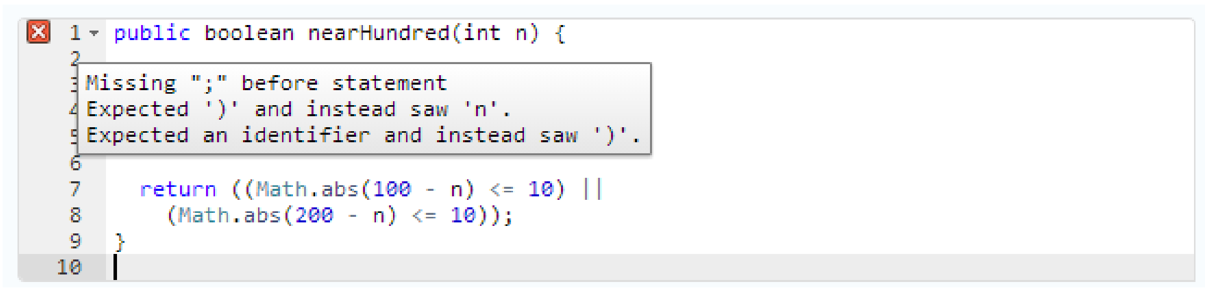


```
public boolean nearHundred(int n) {
  return ((Math.abs(100 - n) <= 10) ||
    (Math.abs(200 - n) <= 10));
}
```

Figure 9.8: Pure HTML code editor[19]

The pure HTML editor (Figure 9.8) is nothing more than a text area in which the user can input code. When the user clicks “Go”, the code is sent to the server, compiled, run, and the result is sent back to the user.

JavaScript powered editor



```
public boolean nearHundred(int n) {

Missing ";" before statement
Expected ')' and instead saw 'n'.
Expected an identifier and instead saw ')'.

  return ((Math.abs(100 - n) <= 10) ||
    (Math.abs(200 - n) <= 10));
}
```

Figure 9.9: A JavaScript-powered editor[20]

The JavaScript editor (Figure 9.9) much more closely resembles the editor from an IDE. It has features such as syntax highlighting, error detection, line numbering, and line highlighting. The use of asynchronous requests would also make it possible to send the code from the editor to the server, analyze it, and then send it back to the client. This allows for updating the editor (with, for example, code analysis feedback) without the user performing any explicit requests.

[19] Available at: http://codingbat.com
[20] Available at: http://ace.c9.io

9.4.2 Why do we need a frameworks or a library?

The problem of making asynchronous requests to servers and handling the responses (effectively making web-pages behave more like ordinary desktop applications) is a complex and difficult problem with no one right solution. Still, the problem has been solved in various ways by different frameworks and libraries. The use of asynchronous requests is but a means to an end in this project, as our challenge is creating a good web-based editor. We need frameworks and libraries to streamline the development process, allowing us to focus on the thing we want to create.

9.4.3 Determining possible framework candidates

We want our web based code editor to mimic a standard MVC application, so looking for the top JavaScript MVC frameworks is a good start. There are a ton to choose from, but from reading various website tutorials and guides, the most popular ones seem to be: Angular, Backbone, Durandal, Ember, and TodoMVC. To determine which ones were the most popular, we ran these through google trends. The result can be seen in Figure 9.10.

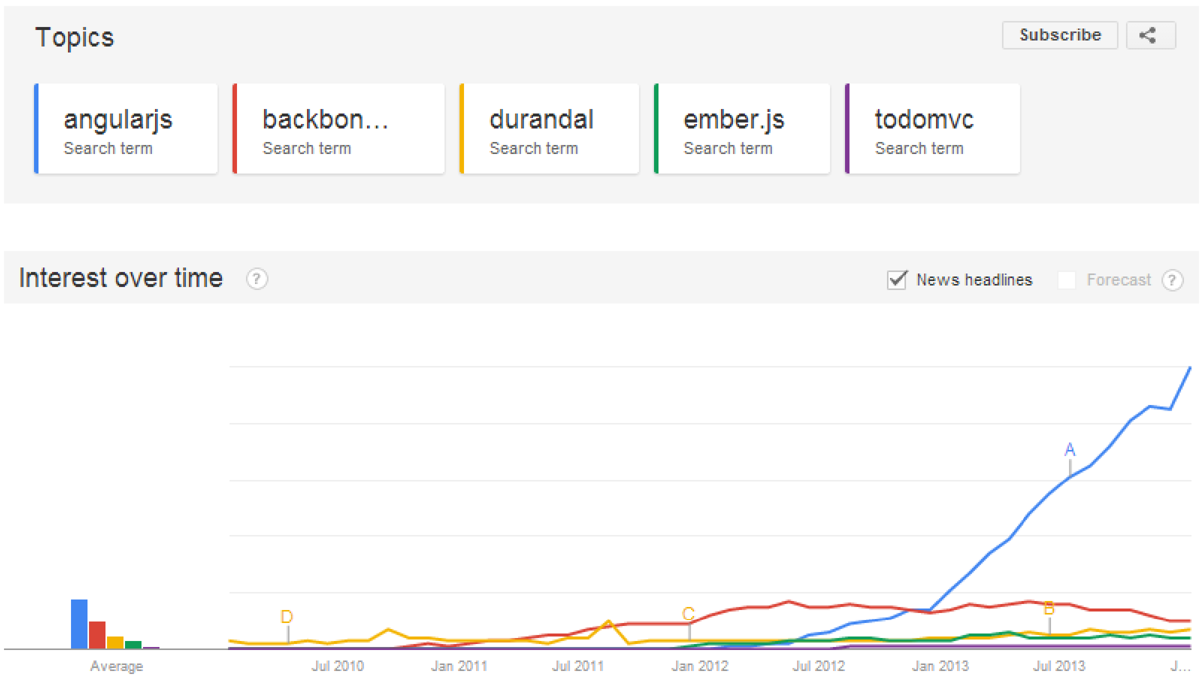


Figure 9.10: JavaScript MVC-framework popularity

As can be seen from the graph in Figure 9.10, Durandal and Backbone have been around for a long time, but Angular surpassed them both shortly after it was released. Ember and TodoMVC have seen increasing popularity since 2012, but are nowhere near the cumulative popularity of the other frameworks. We will take a closer look at Angular, Backbone and Durandal.

Feature	Angular	Backbone	Durandal
Observables	✓	✓	✓
Routing	✓	✓	✓
View bindings	✓	✗	✓
Two-way bindings	✓	✗	✗
Partial views	✓	✗	✓

Table 9.2: JavaScript framework comparison

Upon further investigation it turns out that Backbone (as the name implies) is not a full-fledged solution for mimicking a MVC architecture, but only offers the backbone for the pattern. You need another framework on top of it in order to support views and bindings fully.

It also turns out that all of these frameworks are very extensive and have steep learning curves, with different websites estimating ramp-up times of up to a few weeks. We primarily need to handle asynchronous calls to our webserver when dealing with the code editor, so a full-fledged framework might be overkill, especially considering the limited time frame of this project and the fact that Play Framework (which we decided on in section 0) already provides routing and partial view support. While frameworks definitely have a purpose, we will look for a more lightweight solution. Information about the Angular, Backbone and Durandal were collected from their respective websites: (AngularJS, 2014), (Backbone.js, 2014), and (Durandal, 2014).

9.4.4 Determining possible library solutions

Determining the candidates for libraries is much easier than it was for frameworks, thanks to a continuous survey performed by w3techs.com. The survey (Figure 9.11) is created by examining the *top ten million* websites in the world, and clearly shows that the jQuery library dominates with its 97.3% market share (W3Techs, 2014).

What is jQuery?

jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript. (The jQuery Foundation, 2014)

jQuery does exactly what we need and has an enormous ecosystem built around it, making the choice of a JavaScript library exceptionally easy.

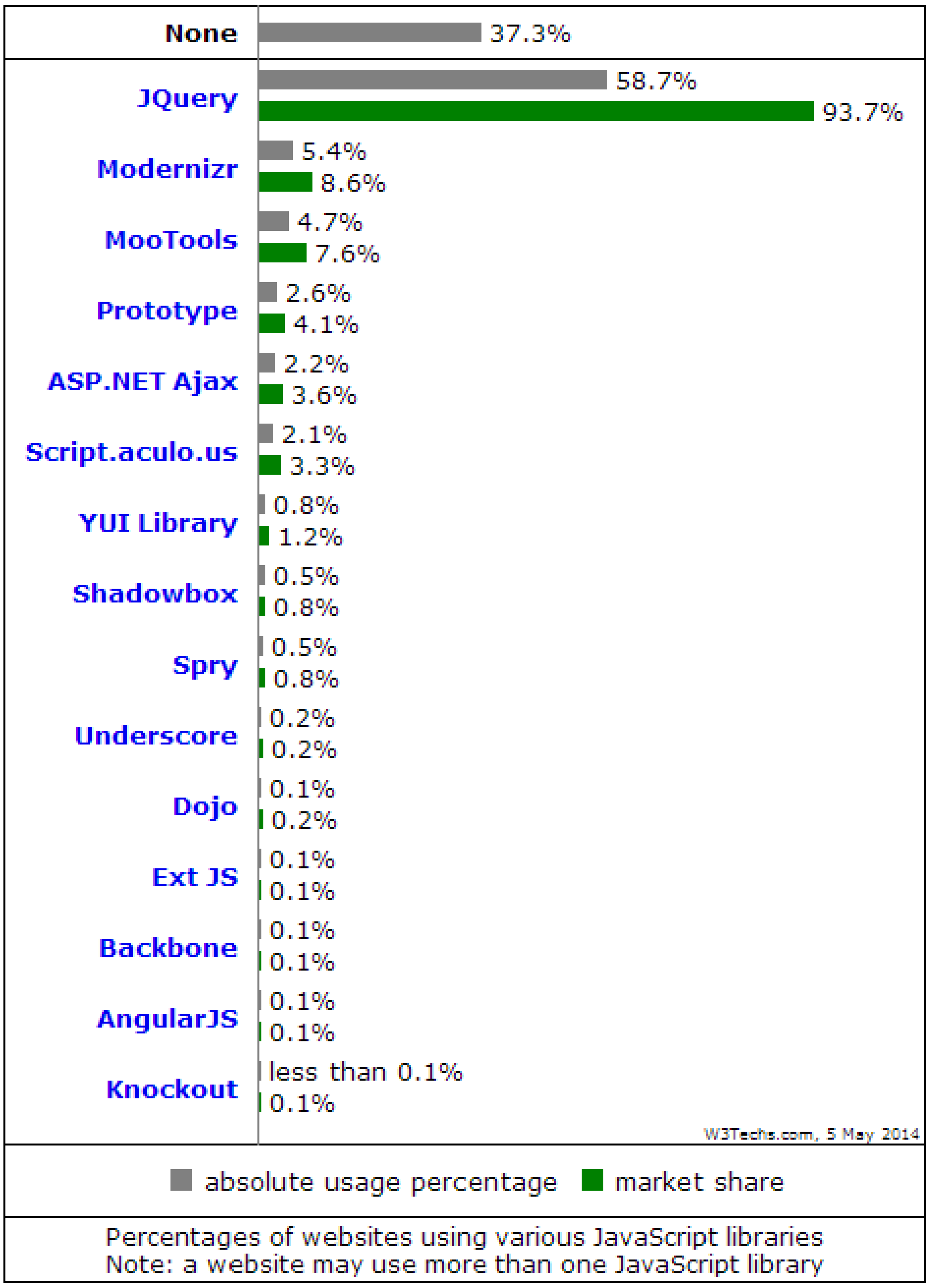


Figure 9.11: JavaScript library usage survey

9.4.5 Conclusion

We definitely need JavaScript to realize the online code-editor, but a full-fledged framework was considered to be counterproductive due to the steep learning curve involved. The jQuery library, however, does everything we need and more, and has a thousands of plugins available, making it a perfect candidate for this project.

9.5 JavaScript code-editors

This section contains an overview and evaluation of different JavaScript code-editors that are available today.

9.5.1 Why does this project need a JavaScript code-editor?

Creating functionality for solving Java-problems is the subject of this thesis, and the online editor is the most technically complicated part of this. Luckily we do not have to start from scratch, as many web-based code-editors have already been created. One of them will surely serve as a good starting point.

9.5.2 Determining possible candidates

There is an actively maintained Wikipedia entry for this, with a very handy comparison matrix (Wikimedia Foundation, Inc, 2014). We will use this matrix to determine possible candidates. We first do a screening based on some key criteria, and have a closer look at the candidates that make it through. The complete list consists of 16 editors, as can be seen in Table 9.3.

Editor	**Cost**	**License**	**Open source**	**Activity**
Ace	Free	BSD3	Yes	Yes
CodeMirror	Free	MIT	Yes	Yes
Orion	Free	BSD3/EPL	Yes	Yes
Codenvy	-	"Public Cloud"	No	Yes
Monaco	Used only by Microsoft	-	No	Yes
MDK	Depends on use	"Dual"	No	Yes
Markitup	Free	MIT/GPL	Yes	Some
Micro	Free	GPL	Yes	Some
LDT	Free	MIT/GPL	Yes	Some
Ymacs	Free	BSD	Yes	Some
Codepress	Free	LGPL	Yes	No
CodeTextArea	Free	BSD	Yes	No
EditArea	Free	LGPL	Yes	No
Helene	Free	GPL	Yes	No
9ne	Free	GPL	Yes	No
jsvi	Free	GPL	Yes	No

Table 9.3: JavaScript code-editor comparison

We want to use an editor which is being actively developed, open source and free. In addition we need it to have a software license which will allow us to use and modify it. This leaves us with just three editors: Ace (Ace, 2014), CodeMirror (CodeMirror, 2014) and Orion (The Eclipse Foundation, 2014).

9.5.3 Determining the best candidate

The article in which we found the comparison matrix also has a feature comparison matrix. We will use this matrix as a starting point to determine which of the three candidates is most suited for our project. Some rows have been changed to better fit our thesis, for example "Syntax Highlighting" was changed to "Java Syntax Highlighting", as Java is the language that is to be used with our editor. Some new rows based on the requirements for our thesis have also been added.

Feature	**Ace**	**CodeMirror**	**Orion**
Java syntax highlight	Yes	Yes	Yes
Java error highlight	No	No	No
Multiple class support	No	No	No
Maximizable	No	No	No
Code completion	Some	Some	Some
Tab support	Yes	Yes	Yes
Indent, new line keeps level	Yes	Yes	Yes
Indent, syntax	Yes	Yes	Yes
Indent, selected block	Yes	Yes	Yes
Bracket matching	Yes	Yes	Yes
Keyboard shortcuts	All common shortcuts and custom key bindings	Fully configurable	Yes
Line numbers	Yes	Yes	Yes
Search & replace	Regex supported	Yes	Yes
Visual styling	Fully theme-able	CSS-based themes	Yes
Undo/Redo	Yes	Yes	Yes
Non US charset support	Yes	Yes	Yes
Code folding	Yes	Yes	Some
Multiple cursors / Block selection	Yes	Yes	No
WebJar available	Yes	Yes	No
Indent guides	Yes	No	No
Code snippets	Yes	Through add-on	Some
Spell checking	Through add-on	No	No

Table 9.4: Code-editor feature comparison

As can be seen from Table 9.4, the three candidates already support a lot of functionality, but neither of them support all of the must-have requirements for our thesis. Out of all of the editors, Ace is the one that supports the most features. Especially block selection and indent guides are very nice features to have that the other editors do not support. As they are equal in all other respects, we feel confident moving forward with Ace as our code-editor.

9.5.4 Conclusion

Many web-based code-editors already exist, but none of them support the requirements we specified in 8.2.2, except the requirement about Java syntax highlighting. The Ace editor provides the best starting point as we move forward trying to implement these missing features.

9.6 CSS-framework

This sections contains a brief overview and evaluation of different CSS-frameworks. Only the biggest and most popular frameworks will be considered.

9.6.1 Why does this project need a CSS-framework?

A CSS-framework is a set of styling-rules used to speed up development and reduce browser incompatibilities. The main focus of this project is to create a good online editor, so as little time as possible should be spent on boilerplate HTML/CSS, such as a grid-system, a menu bar, etc.

9.6.2 Determining possible candidates

To determine what the most popular CSS-frameworks were, several CSS centric websites were visited. The frameworks mentioned the most frequently were: “Twitter Bootstrap”, “Zurb Foundation”, “Skeleton” and “YAML”. We then ran these through google trends to see which ones were the most popular. The results can be seen in Figure 9.12.

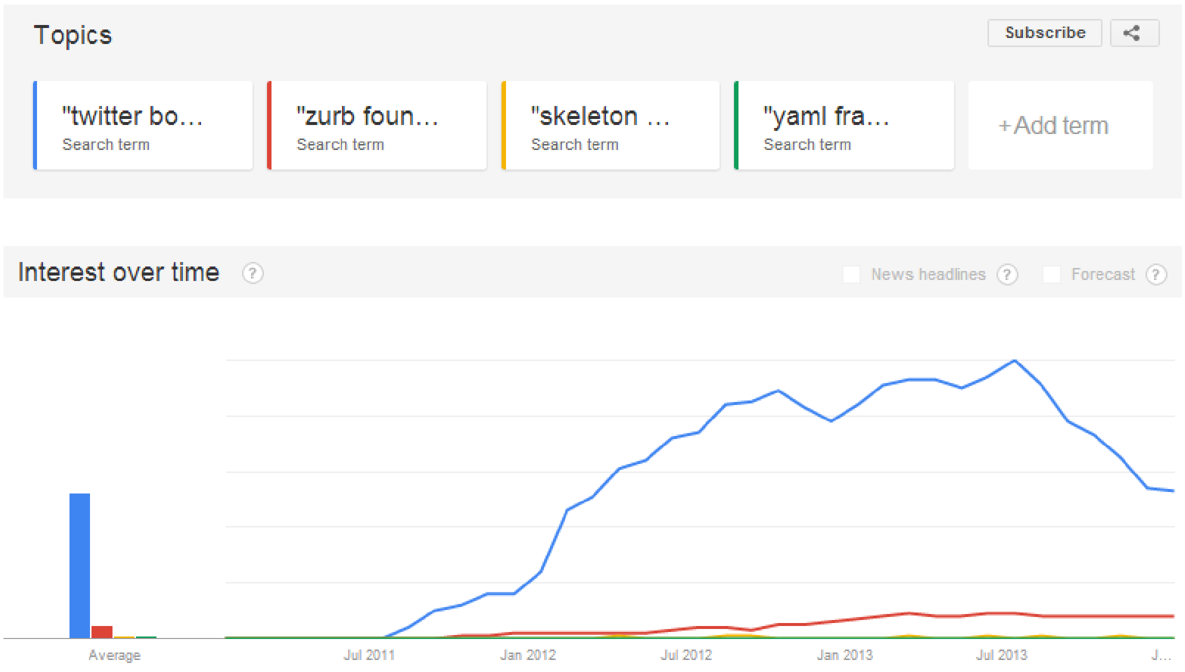


Figure 9.12: CSS-framework popularity

9.6.3 Conclusion

Twitter Bootstrap is by far the most popular CSS-framework, which means it will have the largest online community and the most tutorials available. Since this project is not about design, no further research into CSS-frameworks is really needed. We will go with the most popular one. Twitter Bootstrap also offers a range of JavaScript plugins which are built with jQuery, which we decided to use in section 9.4. This can probably further speed up development.

9.7 CSS-preprocessor

This section contains a brief overview and evaluation of different CSS-preprocessors.

9.7.1 Why does this project need a CSS-preprocessor?

A CSS-preprocessor is an extension of the CSS language, which adds more programming-like features, such as variables, functions, etc. The main focus of this project is to create a good online editor, so as little time as possible should be spent on getting the webpage styling to behave as intended. CSS-preprocessors allow for more rapid prototyping, as changing just a few variables can change every styling rule throughout your page. Preprocessors also allow you to break down CSS code into more manageable pieces and stitch them together into a single minified CSS file which is served to the client. This can reduce the load time for the webpage significantly.

9.7.2 Determining possible candidates

According to a survey performed by CSS-Tricks, there are only two real contenders in this category. Of the ~13 000 responses they gathered, about 50% had a preferred CSS-preprocessor. Among these responses LESS was preferred by 51%, while SASS was preferred by 41%. Eight percent preferred other preprocessors (Coyier, 2012).

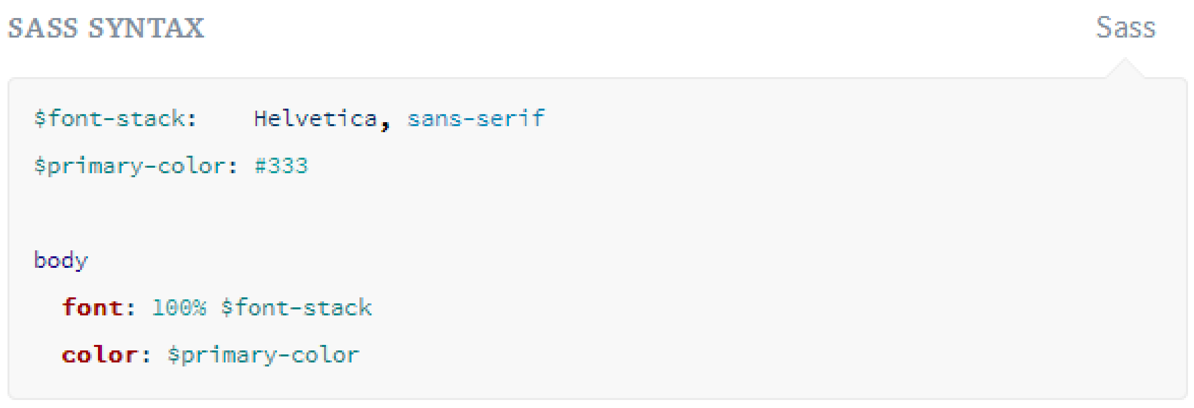
9.7.3 LESS or SASS?

LESS features	SASS features
Variables	Variables
Extend	Inheritance
Mixins	Mixins
Import Directives	Import
Import Options	Nesting
Parametric Mixins	Partials
Mixins as Functions	Operators
Passing Rule sets to Mixins	
Mixin Guards	
Loops	
Merge	
Parent Selectors	

Table 9.5: LESS and SASS features

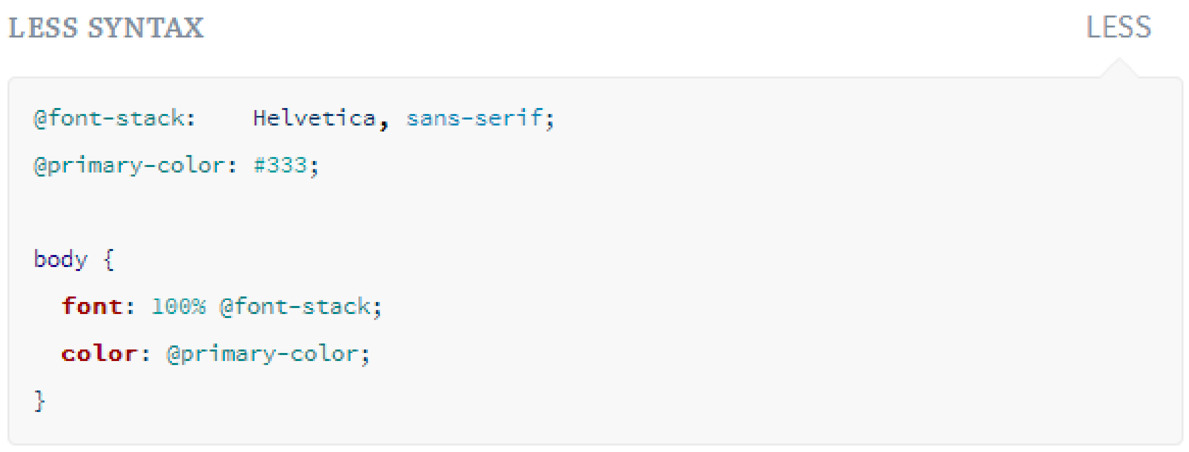
LESS and SASS both offer much of the same functionality (Sellier, 2014), (Catlin, Weizenbaum, & Eppstein, 2014), although they may label each feature differently (for example "Extend" vs "Inheritance"). At first glance LESS looks much more feature rich than SASS, but that is only because the LESS team chose to include five different kinds of mixins and two different kinds of Imports in their feature overview. SASS also supports these features, but does not list them explicitly in the overview.

LESS uses the special character "@", while SASS uses the special character "$". SASS is whitespace/indentation dependent, but most people usually use an extension of the language called SCSS (**Sass**y CSS), which re-introduces semi colons and the curly bracket scope, making the syntax more like regular CSS and LESS.



```
$font-stack:    Helvetica, sans-serif
$primary-color: #333

body
  font: 100% $font-stack
  color: $primary-color
```

Figure 9.13: SASS syntax example



```
@font-stack:    Helvetica, sans-serif;
@primary-color: #333;

body {
  font: 100% @font-stack;
  color: @primary-color;
}
```

Figure 9.14: LESS syntax example

9.7.4 Conclusion

LESS and SASS offer more or less the same features, the syntax being biggest difference. This difference can be seen in Figure 9.13 and Figure 9.14. However, with the SCSS extension the only real difference in syntax is which special character is used: “@” or “$”.

Since we already decided to use Play Framework (which has native support for LESS) in section 0, and Twitter Bootstrap (which is written in LESS) in section 9.6, LESS becomes the obvious choice for our project.

9.8 The complete stack

This section contains a tabular summary (Table 9.6) of all the technology choices made in the in the previous sections, along with a technology stack diagram (Figure 9.15).

Data Storage	EMF
Web server	Play Framework
JavaScript-library	jQuery
Code-editor	Ace
CSS-framework	Twitter Bootstrap
CSS-preprocessor	LESS

Table 9.6: The complete technology stack

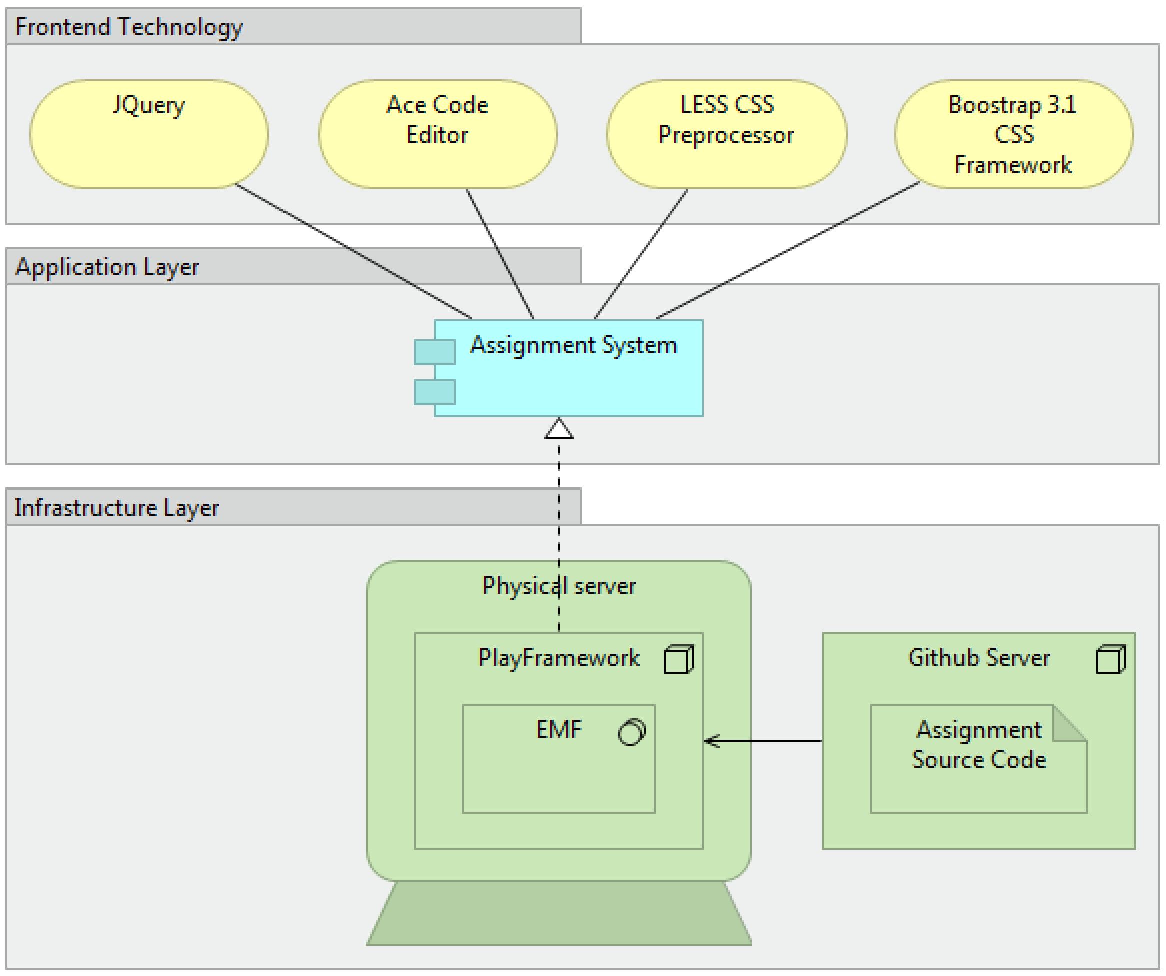


Figure 9.15: The complete technology stack

9.8.1 Technology in relation to the architecture

The diagrams in Figure 9.16 and Figure 9.17 show how the architecture described in section 8.3 is implemented.

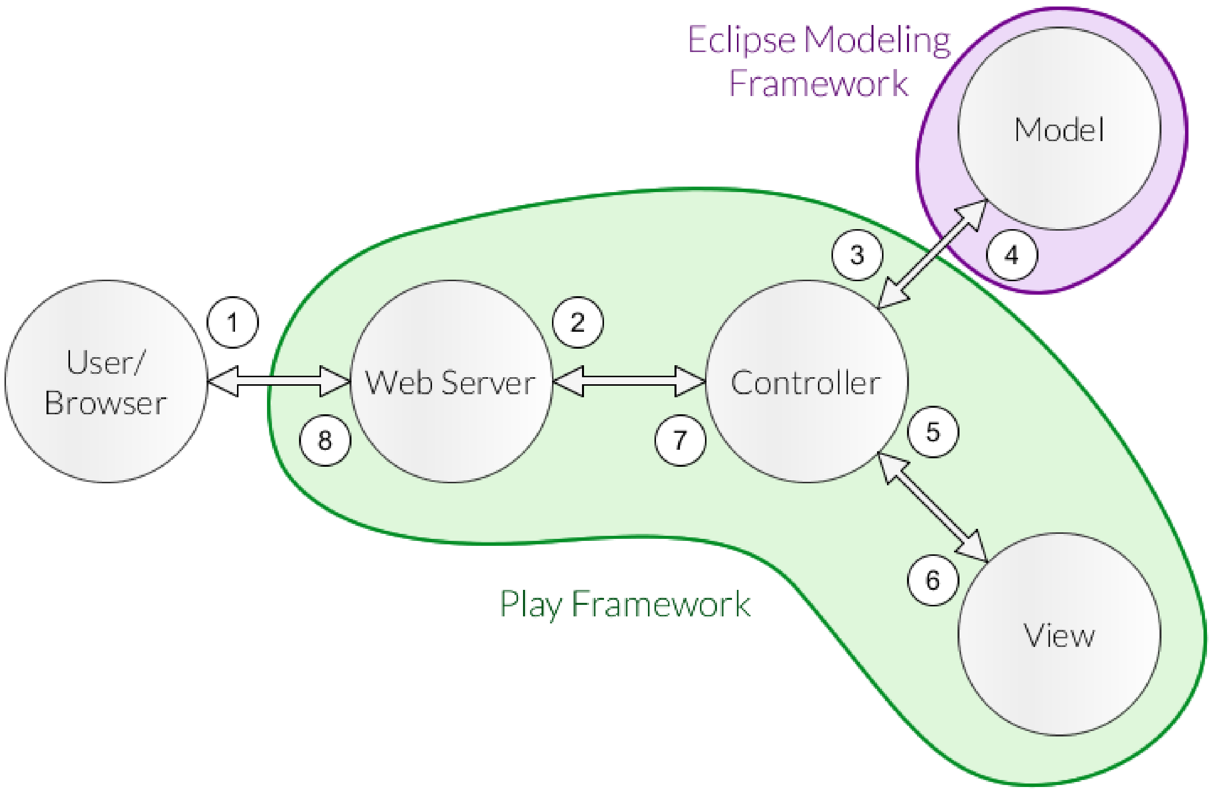


Figure 9.16: MVC implementation

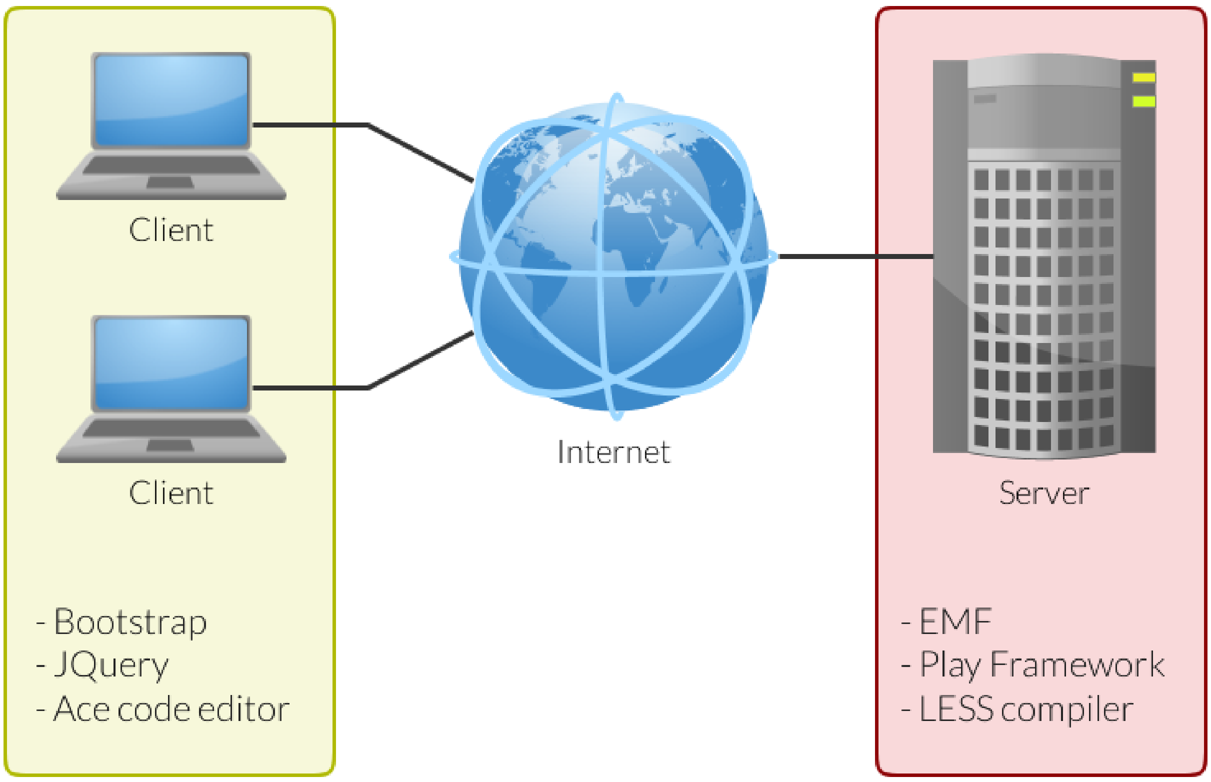


Figure 9.17: Client-Server implementation

9.9 Minor technologies

This section covers minor libraries and plugins included in the project. No extensive reasoning will be given for choosing each plugin, but a short description of the plugin and what it is used for will be included in each section.

9.9.1 LESS Hat

LESS hat is a library for LESS. It provides 86 mixins to help speed up development and reduce cross-platform independencies and bugs. Table 9.7 shows a LESS Hat function and the resulting CSS. One line of code turns into seven, due to all the browser specific prefixes.

LESS Hat	Generated CSS

`.flex-direction(row);`	`-webkit-box-direction: normal;`
	`-moz-box-direction: normal;`
	`-webkit-box-orient: horizontal;`
	`-moz-box-orient: horizontal;`
	`-webkit-flex-direction: row;`
	`-ms-flex-direction: row;`
	`flex-direction: row;`

Table 9.7: LESS Hat and generated CSS

9.9.2 jQuery Collapsible

jQuery Collapsible is a simple jQuery plugin which allows every header in a `<div>` element to be minimized. It is used primarily on the “problems” page to give the students the ability to hide the assignment description and the problem description (giving the editor more space).

9.9.3 Font Awesome

Font Awesome is a font consisting of vector icons. There are no PNG, JPG or SVG files in the implementation of the system, every icon is handled by the Font Awesome font. This reduces load times and keeps the markup neat, in addition to scaling perfectly.

9.9.4 Faker.js

Faker.js is a JavaScript plugin which provides the website with fake content (fake names, fake avatars, etc.). Faker.js will not be a part of the finished system, but we used it to give us a better feeling of how the system will look when it is populated with real users. This allows us to notice design flaws before actually taking the system into use.

10 - Appendix C – Miscellaneous

10.1 Requirements from the preliminary study

This section contains the requirements for the system as described in the preliminary study. The conceptual model and the prototype created in the pre-study was proven to fulfill all of these requirements, meaning that our system also will fulfill them if implemented as it was designed.

10.1.1 Functional requirements

FR1) The system shall support at least ten assignments

FR2) Every assignment shall support at least six problems

FR3) Every problem shall have a score of between 0 and 100 points

FR4) Every problem shall support one or more tests

FR5) The system shall allow for manual assignment approval

FR6) The system shall provide automatic assignment approval

  FR6.1) The system shall use unit tests to test code

  FR6.2) The system shall use quiz functionality for summative assessment

FR7) The system shall provide feedback to users

  FR7.1) The system shall display unit test results

  FR7.2) The system shall use code analysis software and display results

  FR7.3) The system shall offer functionality for explaining quiz answers

FR8) The system shall let users track their progress

10.1.2 Non-functional requirements

NRF1) The system shall motivate the users

  NFR1.1) The system shall use gamification as found effective in case studies

  NFR1.2) The system shall use gamification that is effective in theory

NFR2) The system shall be easy to use

  NFR2.1) The system shall have a SUS score above average (above 68 points)

  NFR2.2) The system shall have a First Click score above 90%.

10.2 Experiment results

10.2.1 Startup time scalability

All measurements are in milliseconds, unless stated otherwise.

Client	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5
#1	5443	4935	4865	5311	4747
#2	5443	5158	4853	5297	5301
AVG	5443	5047	4859	5304	5024

Table 10.1: Startup time (2 clients)

Client	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5
#1	7900	8744	8252	9056	9123
#2	9127	9378	9188	8995	9447
#3	9298	9079	9033	9404	8966
#4	9381	9349	8937	9316	9573
AVG	8927	9138	8853	9193	9277

Table 10.2: Startup time (4 clients)

Client	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5
#1	15015	14194	14501	13909	15240
#2	18258	18101	17775	16998	17723
#3	16968	18364	18600	16373	18076
#4	17863	18898	18198	18769	18639
#5	18605	18595	18880	18942	18019
#6	19192	19311	18621	18707	17296
#7	18122	16999	17023	18229	17377
#8	18243	18674	18780	18504	18538
AVG	17783	17892	17797	17554	17614

Table 10.3: Startup time (8 clients)

C	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14	#15	#16	AVG
M	36	36	39	36	37	39	39	36	35	35	34	37	35	35	34	36	36

Table 10.4: Startup time (16 clients, time in seconds)

10.2.2 Run code scalability

All measurements are in milliseconds, unless stated otherwise.

Client	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5
#1	1132	1311	1032	939	973
#2	1178	1559	916	976	973
AVG	1155	1435	974	958	973

Table 10.5: Run code (2 clients)

Client	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5
#1	1846	1892	1922	1829	1824
#2	1822	1801	1979	1820	1847
#3	1797	1760	1776	1741	1828
#4	1953	1745	1826	1624	1719
AVG	1855	1800	1876	1754	1805

Table 10.6: Run code (4 clients)

Client	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5
#1	3231	2953	2885	3062	2791
#2	3171	3147	3024	2987	3016
#3	3224	3011	3005	3063	3131
#4	3286	3121	3005	3213	2929
#5	3136	3001	2885	2977	2962
#6	3065	2874	2785	2934	2957
#7	2943	2909	2739	2811	2720
#8	2734	2617	2507	2771	2649
AVG	3099	2954	2854	2977	2894

Table 10.7: Run code (8 clients)

C	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14	#15	#16	AVG
M	4,7	5,1	5,5	5,5	5,1	5,6	5,5	5,8	5,4	5,5	5,6	5,3	5,2	5,1	4,9	4,9	5,3

Table 10.8: Run code (16 clients, time in seconds)

10.2.3 Run tests

All measurements are in milliseconds, unless stated otherwise.

Client	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5
#1	1348	1368	1396	1366	1387
#2	1340	1387	1401	1339	1360
AVG	1344	1378	1399	1353	1374

Table 10.9: Run tests (2 clients)

Client	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5
#1	2411	2765	2454	2360	2472
#2	2541	2589	2804	2598	2724
#3	2465	2526	2569	2455	2659
#4	2526	2403	2548	2335	2591
AVG	2486	2571	2594	2437	2612

Table 10.10: Run tests (4 clients)

Client	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5
#1	4588	3396	3086	3915	3906
#2	5158	3446	3505	4850	4707
#3	4914	3503	3457	4954	4688
#4	4725	3563	3509	4915	4927
#5	4629	3341	3385	4943	4512
#6	4874	3430	3303	4608	4676
#7	4704	3198	3190	4911	4587
#8	4625	3241	3082	4310	4298
AVG	4777	3390	3315	4676	4538

Table 10.11: Run tests (8 clients)

C	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14	#15	#16	AVG
M	6,0	6,8	6,4	6,7	6,6	6,6	6,9	6,4	6,7	6,8	6,4	6,1	6,4	5,9	5,7	7,0	6,5

Table 10.12: Run tests (16 clients, time in seconds)

10.2.4 Error checking

All measurements are in milliseconds, unless stated otherwise.

Client	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5
#1	267	339	287	260	259
#2	289	319	280	282	275
AVG	278	329	284	271	267

Table 10.13: Error checking (2 clients)

Client	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5
#1	266	303	251	261	236
#2	271	314	264	287	302
#3	302	359	254	298	352
#4	307	281	273	294	270
AVG	287	314	261	285	290

Table 10.14: Error checking (4 clients)

Client	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5
#1	272	259	248	266	254
#2	252	254	262	344	328
#3	304	317	243	388	304
#4	289	413	240	393	379
#5	306	438	299	397	227
#6	371	596	316	348	533
#7	345	290	251	362	493
#8	337	1046	255	363	321
AVG	310	452	264	358	355

Table 10.15: Error checking (8 clients)

C	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14	#15	#16	AVG
M	0,4	0,4	0,3	0,3	0,3	0,5	0,8	0,8	0,5	0,5	0,5	0,2	0,5	0,5	0,3	0,4	0,4

Table 10.16: Error checking (16 clients, time in seconds)

10.2.5 Code completion

All measurements are in milliseconds, unless stated otherwise.

Client	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5
#1	387	387	397	384	364
#2	403	387	350	340	387
AVG	395	387	374	362	376

Table 10.17: Code completion (2 clients)

Client	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5
#1	662	588	606	709	661
#2	714	666	725	649	749
#3	711	707	772	732	672
#4	638	690	735	713	728
AVG	681	663	710	701	703

Table 10.18: Code completion (4 clients)

Client	Measure 1	Measure 2	Measure 3	Measure 4	Measure 5
#1	1207	972	1014	1479	1478
#2	1267	1140	1380	1422	1465
#3	1416	1299	1272	1973	1413
#4	1367	1512	1148	1737	1411
#5	1436	1382	1199	1870	1538
#6	1397	1450	1365	2079	1432
#7	1534	1410	1214	1991	1423
#8	1452	1435	1258	1944	1367
AVG	1385	1325	1231	1812	1441

Table 10.19: Code completion (8 clients)

C	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14	#15	#16	AVG
M	2,3	2,2	4,9	2,6	2,5	2,6	2,7	4,2	4,1	4,0	3,8	3,6	3,5	3,3	3,0	2,6	3,2

Table 10.20: Code completion (16 clients, time in seconds)

10.3 Setup guide

10.3.1 Setting up Eclipse

1. Start Eclipse
2. Set workspace to the root of project (the folder containing README.md)
3. Open "File" > "Import..."
4. Select "General" > "Existing Projects into Workspace"
5. Press "Next"
6. In the "Select root directory" field, browse to "/no.ntnu.assignmentsystem.model"
7. Make sure the project is checked
8. Click "Finish"

10.3.2 Installing Java 8 plugin

1. Go to "Help" in the menu bar
2. Open "Eclipse Marketplace"
3. Search for `java 8 kepler`
4. Install "Java 8 support for Eclipse Kepler SR2"
5. Complete the wizard

10.3.3 Setting up JRE8 in eclipse

1. Go to "Window" > "Preferences" > "Java" > "Installed JREs"
2. If jre8 is not in the list, click "Add"
3. Choose "Standard VM"
4. Set "JRE Home" to your jre8 path.
5. Click "Finish"

10.3.4 Installing Maven plugin

1. Go to "Help" in the menu bar
2. Open "Eclipse Marketplace"
3. Search for `maven 1.4`
4. Install "Maven Integration for Eclipse (Juno or newer) 1.4"
5. Complete the wizard.

10.3.5 Installing Akka dependencies into Eclipse

1. Open terminal and navigate to "/setup" folder
2. Run `mvn p2:site` (https://github.com/reficio/p2-maven-plugin)
3. Open Eclipse
4. Go to "Help" > "Install new Software..." in menu bar
5. Click "Add..."
6. Click "Local"
7. Navigate to "/setup/target/repository" and click "Open"
8. Click "OK"
9. Check "Maven osgi-bundles" in table view
10. Click "Next >"
11. Click "Finish"

10.3.6 Generating model code

1. Navigate to "model/model.genmodel"
2. Right-click on "Model"
3. Click on "Generate Model Code"

10.3.7 Create a Run configuration

1. Open "Run" > "Run Configurations" in menu bar
2. Right-click "Java Application" and select "New"
3. Set name to `Main`
4. Set project to "no.ntnu.assignmentsystem.model"
5. Set main class to `Main`
6. Click "Run" and confirm that it compiles

10.3.8 Exporting to JAR

1. Right-click the project and select "Export"
2. Select "Java" > "Runnable JAR file"
3. Click "Next"
4. Set launch configuration to "Main"
5. Set export destination to "AssignmentModel/lib"
6. Set library handling to "Copy required libraries into a sub-folder next to the generated JAR"
7. Check "Save an ANT script"
8. Click "Finish"
9. Move the JAR-files from sub-folder to "AssignmentModel/lib"

10.3.9 Set up automatic building

1. Right-click project and select "Properties"
2. Go to "Builders"
3. Click "New..."
4. Select "Ant Builder"
5. Click "OK"
6. Set name to `Model Builder`
7. Set buildfile to the generated ANT-file
8. Set base directory to the root folder (folder containing README.md)
9. Click "OK"

10.3.10 Setting up IntelliJ

Generate IDEA-files (run `activator idea` in directory)

1. Start IntelliJ
2. Select "Open Project"
3. Navigate to "/AssignmentSystem"