Handout Software Engineering

EiCompany

EiBoard

Eileen Fahrner
Matteo Staar
Julian Stadler
Niklas Geppert
Marius Schad

Agenda

- Elevator PITCH (Project goal/vision)
- Architecture decisions and reasoning behind
- Applied design patterns and/or development platforms, libraries
- summary of your tech stack
- Quality assurance, e.g., test coverage
- CI/CD setup
- **Live demo**
- Project management facts and lessons learned
- Numbers and Data

Project goal/vision

Fully functional calendar that can be accessed via Browser.

Comfortable integration of Rapla.

Architecture decisions and reasoning behind

Logical View:
To display (university) tasks directly to the user. In addition, own tasks can be added and timetable can be seen.

Process View
The ***LoginSequenceDiagram*** basically shows that the user sees the login page and enters his login data there. This data is sent to the backend, which then compares it with the help of a database and sends back a result.

Development View:
The subsystem ***"Account"*** is responsible for the modules ***"Login"*** and ***"Registration"***. If the login is successful, the subsystem ***"Dashboard"*** is used. This contains the modules ***"Calendar", "Tasklist", "Navigation"*** and ***"Impressumbar"***. If the ***"Calendar"*** module is accessed, the ***"Rapla"*** subsystem is called, which returns the requested timetable entries.

Physical View:
The software can be used in the ***browser*** and on mobile ***Android*** devices. This was realised through the Flutter development kit. The frontend communicates with the backend, which was realised with Spring Boot. The entire backend runs on a Jakarta server and can also access the Rapla content via a Rest API. Whole app runs on a contabo server.

MVC using the MVC pattern for ***Frontend and Backend*** enables a clean software architecture with separate ***model, view and controller***.

Frontend: In the Frontend no MVC Tool is needed, because the MVC Pattern is integrated into ***Flutter*** development. View: activities in the app

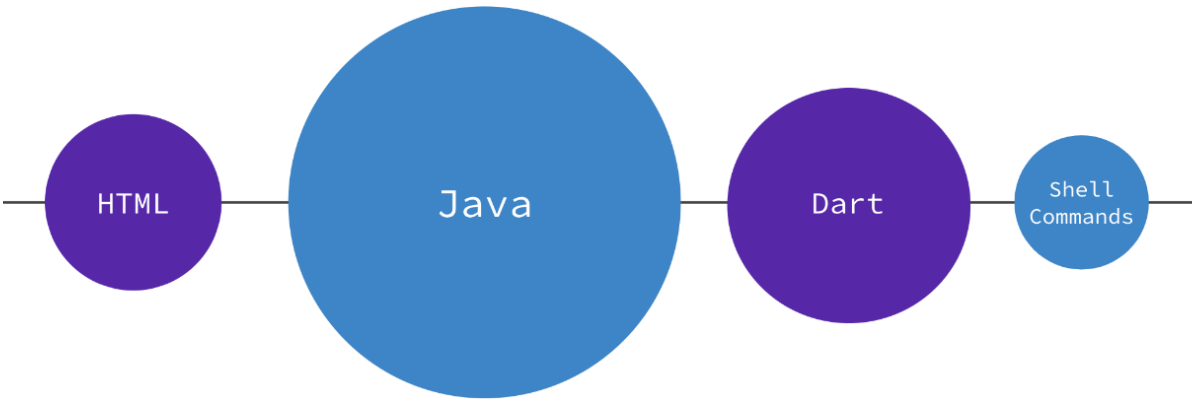
Backend: Is written in Java. As MVC tool we are using ***Spring Boot***. The Server offers multiple REST APIs which are accessed by our frontend. Model: domain specific classes Controller: RestController.

Applied design patterns and/or development platforms, libraries

- SRP: The Single Responsibility Principle.
    - One module does one thing properly (but not necessarily to one, and only one, actor.)
- OCP: The Open-Closed Principle
    - Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
    - Using Controllers, interfaces etc.
- ISP: The Interface Segregation Principle
    - No code should be forced to depend on methods it does not use
    - Using Controllers, interfaces etc.

- Eclipse (IDE)
- IntelliJ (IDE)
- Flutter/Dart (Frontend)
- Spring Boot/Java (Backend)
- Tomcat (Server Backend)
- H2 Database

summary of your tech stack



Frontend Tools
Eclipse, IntelliJ, Dart (Flutter), Cypress (End to End testing), Sellenium

Backend Tools
Eclipse, IntelliJ, mySQL (Database), Tomcat, Spring, Swagger

Other Tools
Jenkins (CI/CD), Contabo (Server), KeyCloak (Login-Security), Docker, H2 Database, Paypal (to pay for the server)

Documentation Tools
GIT (Storage), Google Docs, Discord (Communication), Microsoft Office, YouTrack, different Tools to create Diagrams

Quality assurance, e.g., test coverage

Frontend:

For **the Unit** tests in the Dart frontend we use the testing library **"flutter_test"**, which comes with the Flutter framework.

For the **UI tests** we use **Selenium**. Some of the UI things are tested **automated** but not every single user flow. After each commit we are clicking ourselves throughout the entire application though, which still covers UI testing, but manual.

Not every single component is tested, as most of the logic of the application comes from the backend.

Backend:

For the **Unit** and **Integration** test in the Java backend we use the testing frameworks **JUnit** and **Mockito**.

For test coverage we use "**EclEmma**" (Eclipse plugin), and our developers working in IntelliJ IDEA use the built-in "**IntelliJ IDEA Code Coverage**".

5.1.1 Unit Testing

Unit testing ensures, that the tested sourcecode works as expected. Therefore small parts of the sourcecode will be tested independently.

	Description
Technique Objective	Ensure that each unit of code (functions, methods, classes) works as intended
Technique	Implement test methods using JUnit Framework, Mockito library (Backend) and flutter_test library (Frontend)
Oracles	Test results are logged in CI/CD tool (Jenkins), and compared against expected output to determine if the tests passed or failed
Required Tools	JUnit 5 and Mockito Dependencies in Backend, and flutter_test library in Frontend. CI/CD Pipeline with test stages
Success Criteria	All tests pass. Coverage is above 10% (Frontend) / 60% (Backend)
Special Considerations	-

5.1.2 UI Testing

UI testing evaluates the application's performance from a user's point of view, with the aim of verifying that the user interface operates as intended.

	Description
Technique Objective	Test application automated from the perspective of the user through UI Test
Technique	Writing test cases using Selenium WebDriver API and Java to simulate user interactions with the application's UI (mainly use cases).
Oracles	Expect that the UI elements are displayed and behave as expected during test execution. Test results are logged in CI/CD tool (Jenkins), and compared against expected output to determine if the tests passed or failed.
Required Tools	Selenium WebDriver API and added dependencies for Java Maven project. CI/CD Pipeline with test stages
Success Criteria	All UI tests pass.
Special Considerations	-

5.1.3 Integration Testing (API Testing)

API Testing is part of integration testing. Integration tests test multiple modules of an application together. The main goal of API testing is to ensure, that the provided APIs of the Backend behave as expected.

	Description
Technique Objective	Ensure that the implemented API functions correctly and returns expected results
Technique	Implement test methods using JUnit Framework and REST-assured library to perform HTTP requests and verify responses
Oracles	Test results are logged in CI/CD tool (Jenkins), and compared against expected output to determine if the tests passed or failed
Required Tools	JUnit 5 and REST-assured library dependencies in Backend
Success Criteria	All tests pass. Coverage is above 60%
Special Considerations	APIs must be in a testable state, e.g., mock objects are used for dependencies that may not be available during tests

CI/CD setup

Backend **Jenkins** pipeline that automatically does these after each push and merge event:

- check out the latest code (Checkout stage)
- build the code with the help of Maven (Building stage)
- run all tests of the backend with "mvnw test (Testing stage)
- deploy the new code (Deploy stage)

These steps should be defined with the help of a Jenkinsfile, which is located in our Git repository.

Live demo

http://eiboard.de/

Project management facts and lessons learned

Continue ⏩
What helped us move forward:

- study/subject
- interest in different tool
- creativity
- Regular team discussions
- Challenge of the lecture
- Being responsible for our own product
- sharing accounts (email etc.)

Stop ⛔
What held us back:

- Jenkins
- overengineering of tasks
- Overload of new tools
- Security
- time used for other lectures
- Lectures in the afternoon/
- rooms to heated
- exams of other lectures

Invent 🎨
How could we do things differently:

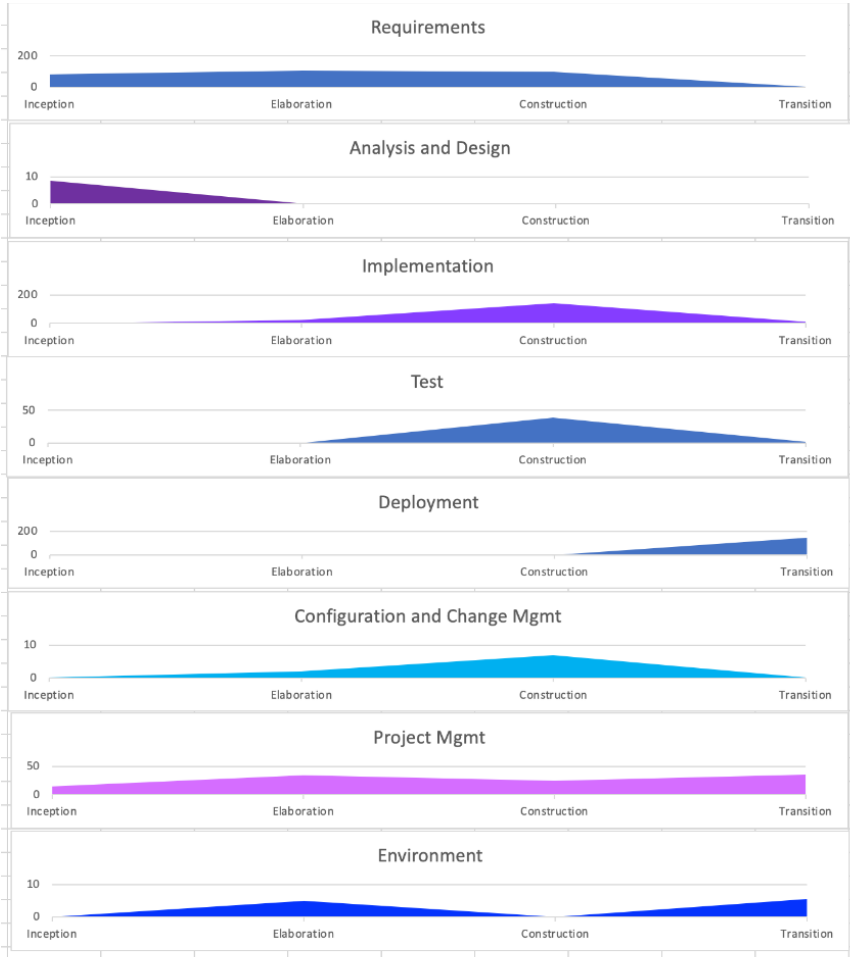
- Keep up with set time management
- Stopping to overengineering tasks
- better plan for merger of frontend and backend
- Setting a weekly working appointment (in cooled room or in the morning)
- Better communication between teams

Risks That Happened:

1. **Server crashes**
   Backup of data
2. **No connection between frontend and backend**
   regular tests
3. **merge conflicts**
   code refactoring (Jenkins)

Numbers and Data

RUP Diagram:



*data in hours

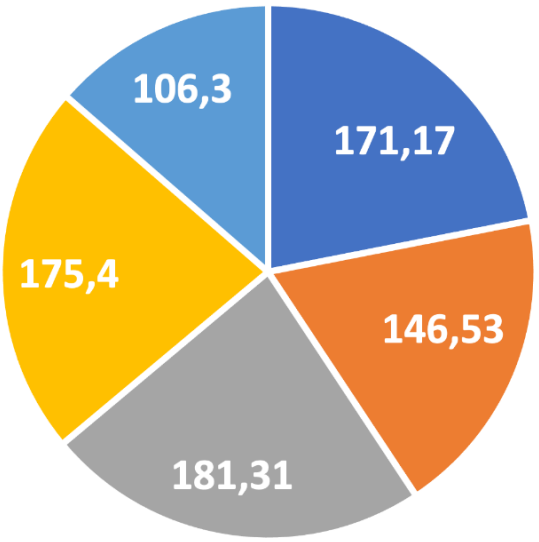
		Transition	Inception	Elaboration	Construction	Kein rup phase
Gesamt	**27w 2d 5h 4m**	**4w 3d 6h 50m**	**2w 3d 1h 7m**	**4w 1d 3h 24m**	**7w 3d 5h 59m**	**8w 3h 44m**
Test	1w 21m	1h 59m			4d 6h 22m	
Requirements	7w 5h 13m		2w 1h 30m	2w 3d 1h 18m	2w 2d 2h 25m	
Project Management	2w 2d 4h 19m	3d 3h 27m	1d 7h	4d 1h 54m	2d 7h 58m	
Implementation	4w 2d 2h 24m	1d 2h 58m		3d 1h 12m	3w 2d 6h 14m	
Environment	1d 2h 38m	5h 38m		5h		
Deployment	3w 3d 48m	3w 3d 48m				
Config & Change Mgmt	1d 1h			2h	7h	
Analysis & Design	1d 37m		1d 37m			
Kein rup workflow	8w 3h 44m					8w 3h 44m

* one day equals 8 hours. One week equals 5 days

Time Per Person:

- Eileen
- Matteo
- Marius
- Julian
- Niklas

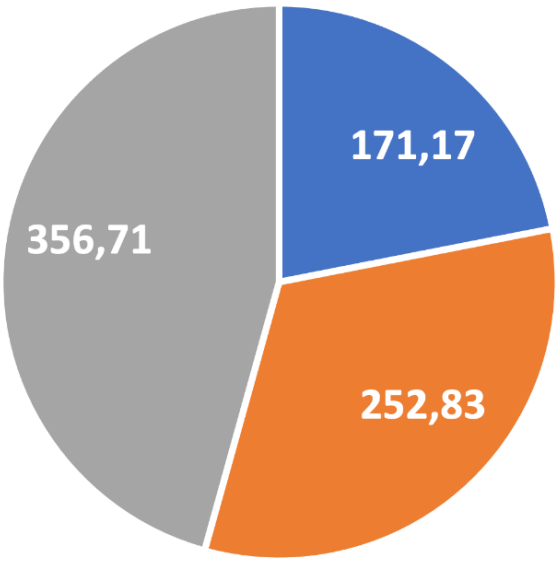
*data in hours



Frontend vs. Backend:

- Frontend
- Backend
- Mix

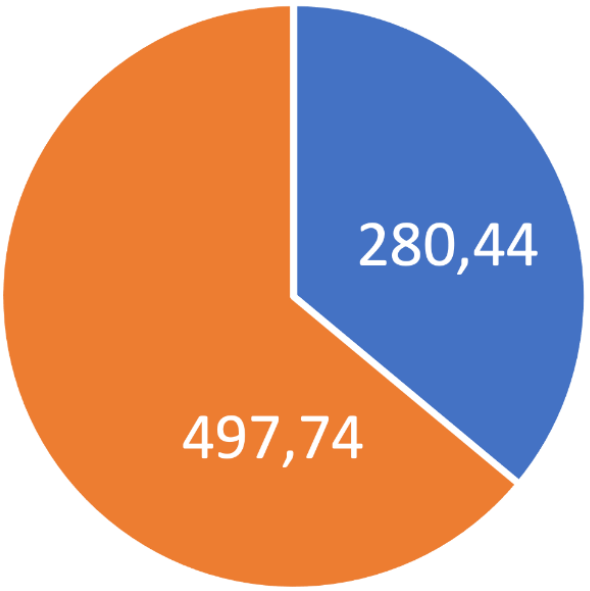
*data in hours



3rd vs 4th semester:

- 4th semester
- 3rd semester

*data in hours



Major contributions

Eileen: frontend and design
Matteo: backend and server
Marius: project management and server
Julian: backend and server
Niklas: frontend