Qupids

Speed Dating

presented by

Group 6: Ade Aiho, Heta Hartzell, Mika Laakkonen, Jonne Roponen

Introduction

The final half of Qupids Speed Dating application development happened during sprints 5-8 and focused on improving and completing functionality for deployment. This included design and implementation of multilingual support for both UI and database.

The applications quality and stability was verified with static code analysis using tools such as SpotBugs, Checkstyle, PMD and SonarQube. Code improvements and overall application reliability improved after the code analysis results were studied and acted upon.

Acceptance testing and beta testing were conducted based on user stories and their acceptance criteria. The test results verified that the application was working as intended in all usage scenarios.

Finally the project documentation was completed. Finalized document includes architecture diagrams, detailed feature descriptions, testing summaries, and installation instructions. Everything needed for future and continued deployment.

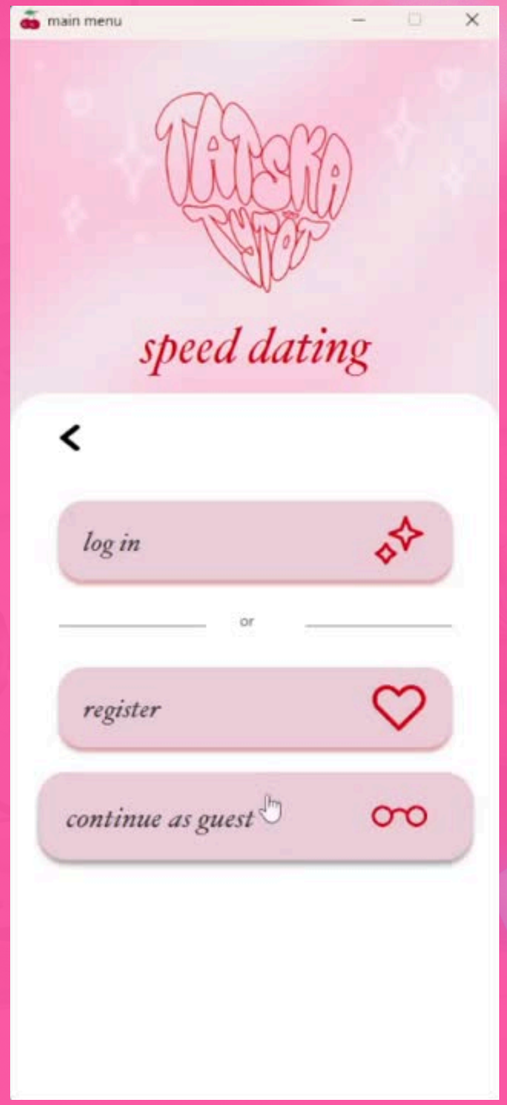
Project Vision & Goals

The vision was to create an intuitive, multilingual speed dating application that enables both guest and registered users to find compatible matches based on shared interests, while offering a visually pleasing and seamless user experience.

Goals

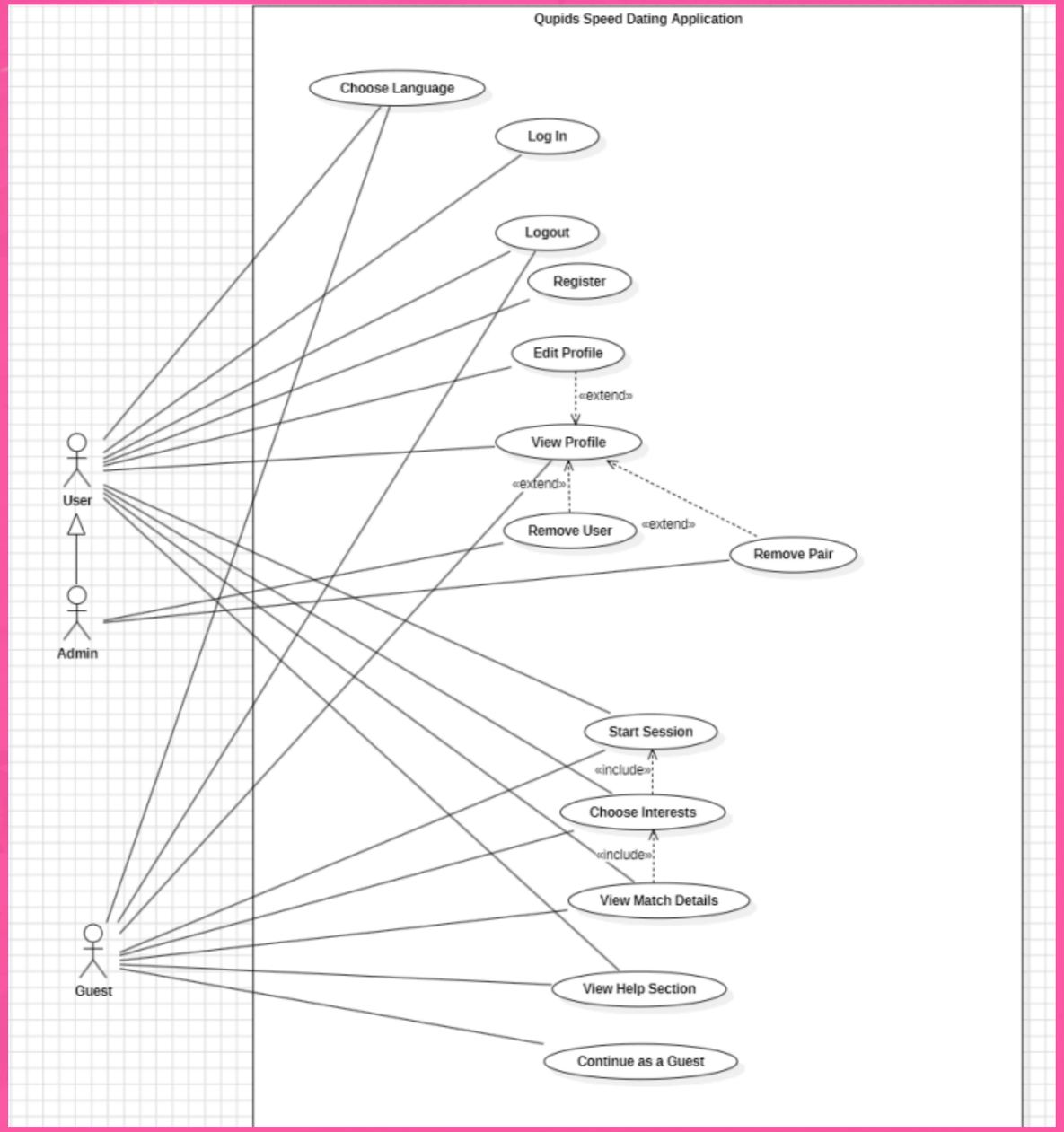
- Design and implement a fully functional speed dating application using JavaFX.
- Support multiple user types (guest, user, admin) with distinct functionalities.
- Enable interest-based matchmaking using a compatibility scoring system.
- Provide multilingual support (Finnish, English, Japanese, Chinese) for greater accessibility.
- Ensure clean code quality, responsive UI, and maintainable architecture using modern development tools and agile practices.

Product Demo



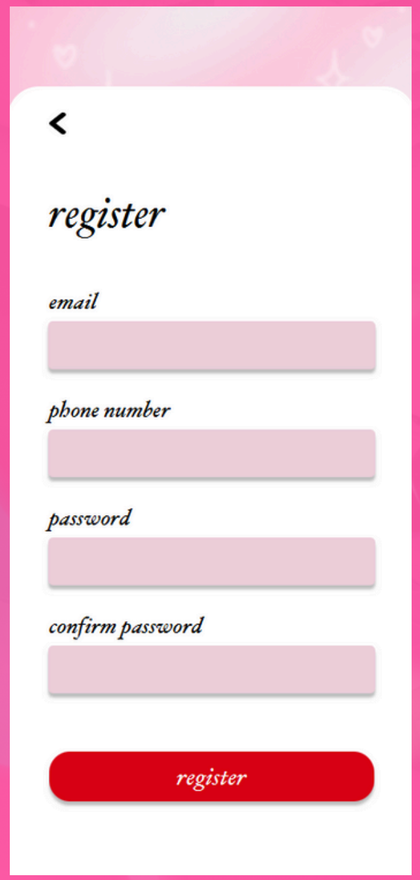
Use-Case Diagram

- **Three actors:**
  - User
  - Guest
  - Admin (inherits user's use-cases)
- **Multiple use-cases:**
  - Login features
  - Session
  - Profile
  - Others



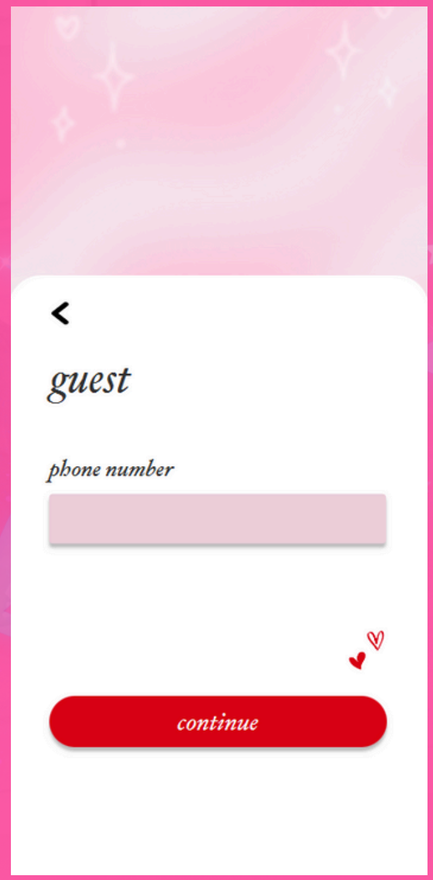
Application Features - Login

Registration



Login

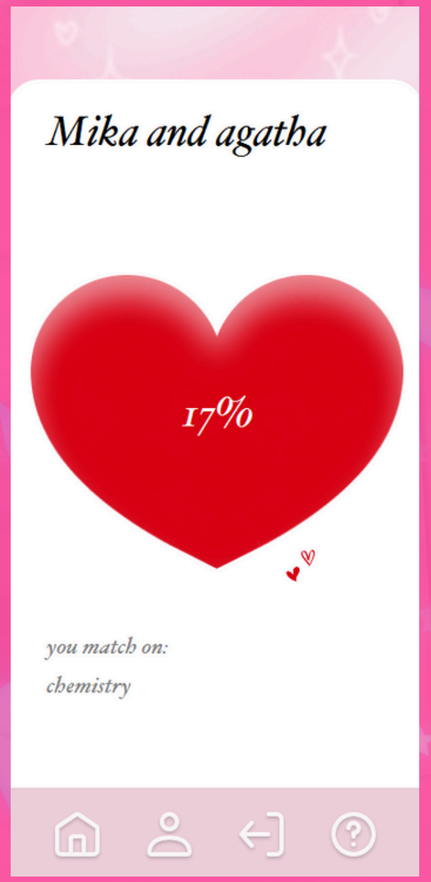
Guest Login



Application Features - Session

Session

Viewing Details



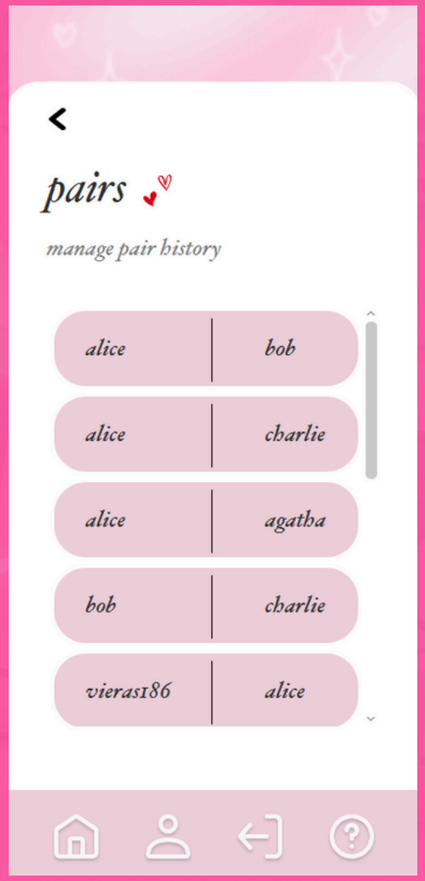
Application Features - Profile

View Profile

Edit Profile

Application Features - Admin

Manage Pairs



Manage Users

Application Features - Other

Change Language

View Help

Software Architecture

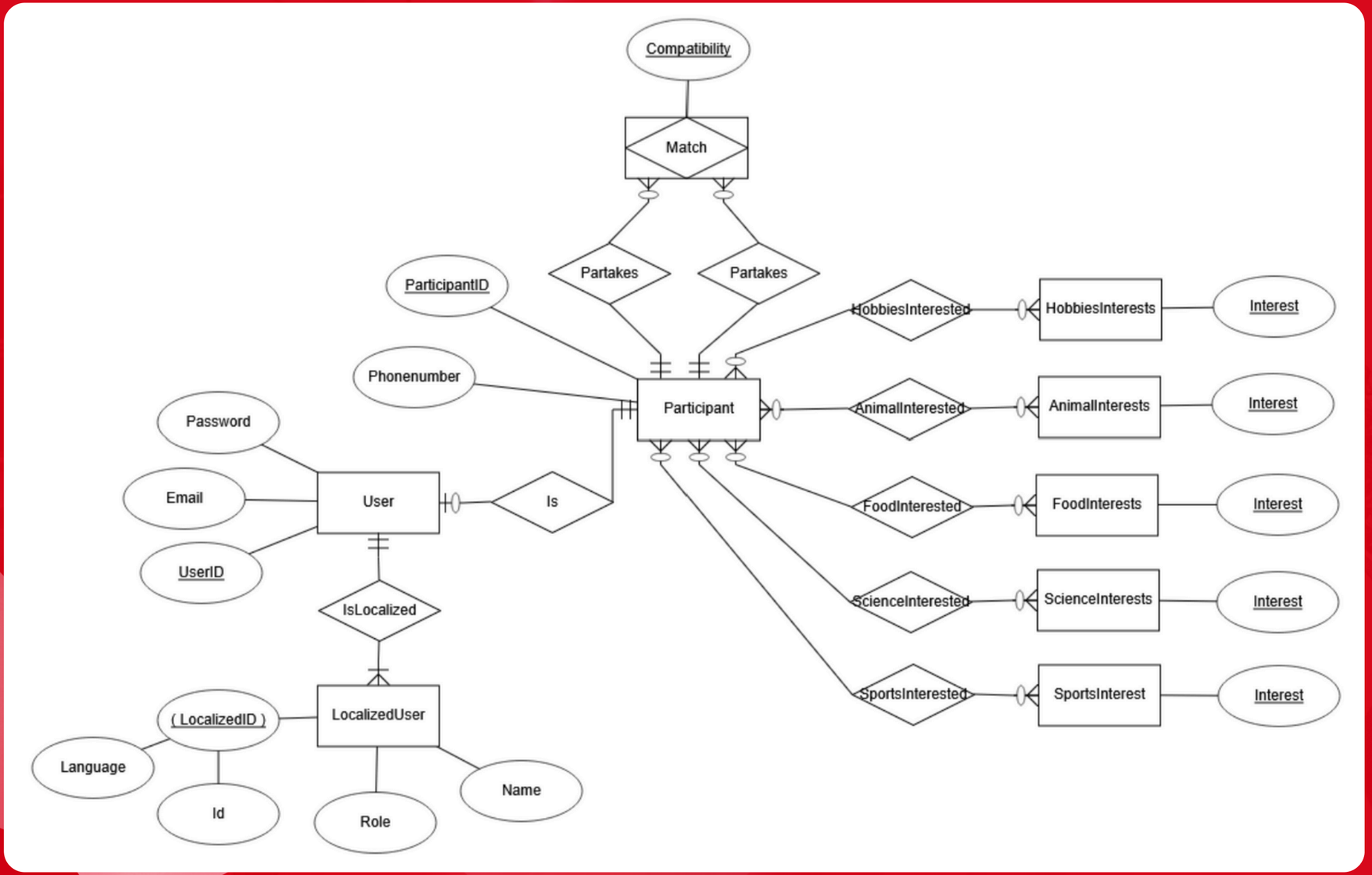
Layered Architecture

Separates logic across different layers to ensure modularity, scalability and maintainability. Keeps the system organized.

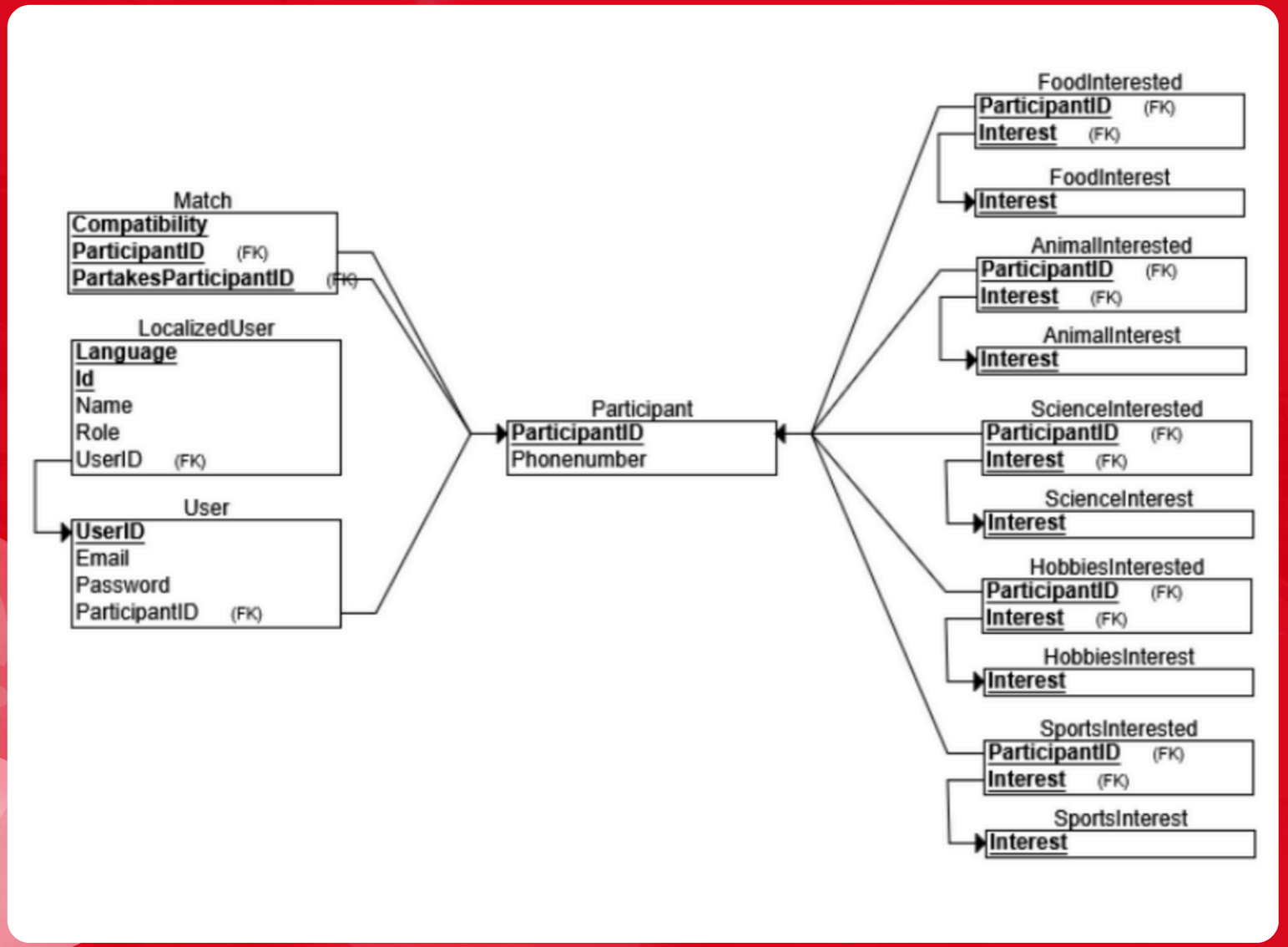
High-level overview of the layers:

- **UI Layer:** Handles user interactions thorugh a JavaFX UI.
- **Controller Layer:** Manages user input and delegates actions to the model and service layers.
- **Model Layer:** Contains data structures and core business logic.
- **Service Layer:** Acts as a bridge between the controllers/models and the DAO layer.
- **DAO layer:** Manages database access.
- **Database layer:** MariaDB database to store application data.
- **Context layer:** Stores session data and language preferences.

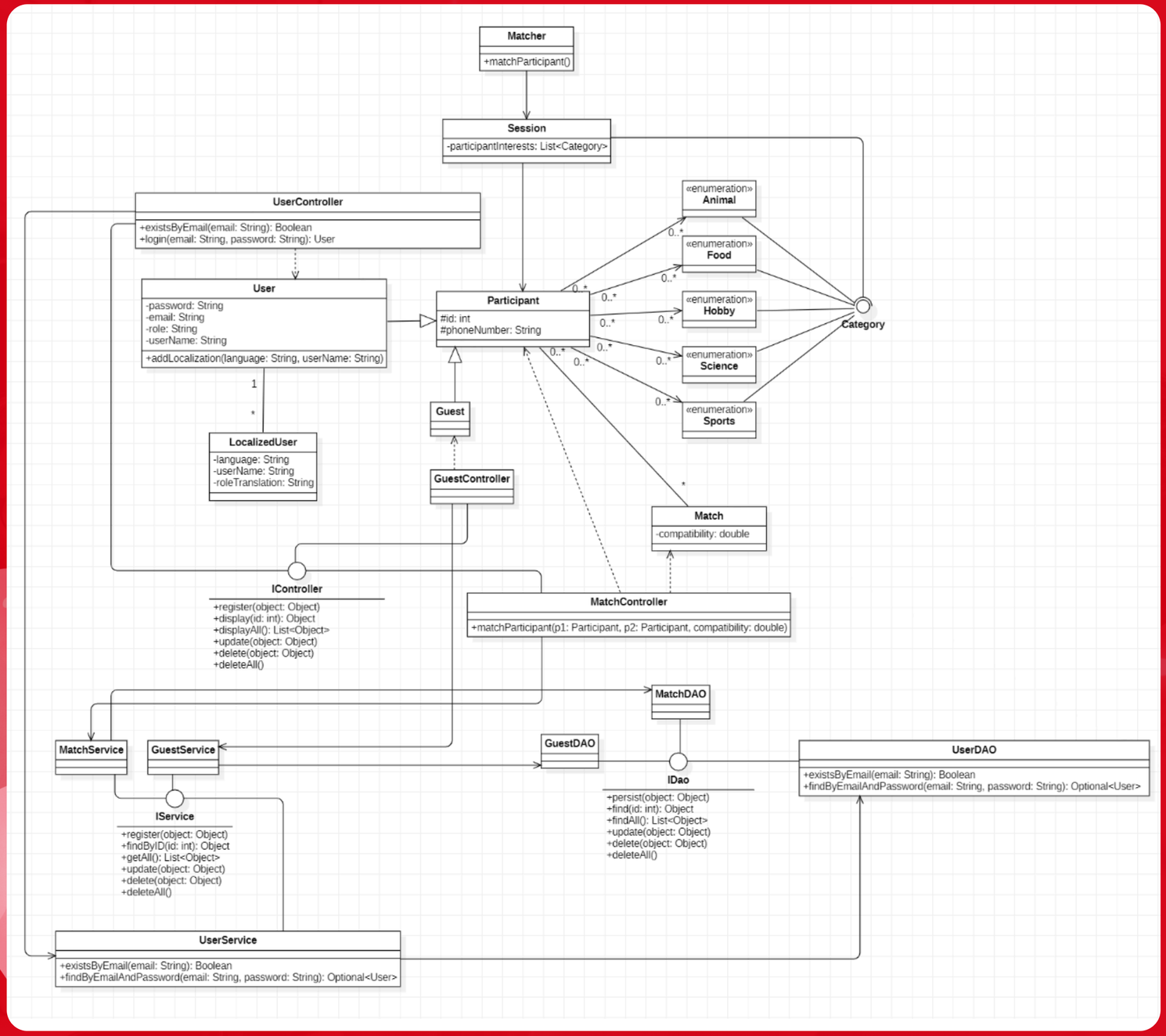
ER-Diagram


Compatibility
Match
Partakes
Partakes
ParticipantID
Phonenumber
Participant
HobbiesInterested
HobbiesInterests
Interest
AnimalInterested
AnimalInterests
Interest
FoodInterested
FoodInterests
Interest
ScienceInterested
ScienceInterests
Interest
SportsInterested
SportsInterest
Interest
Password
Email
UserID
User
Is
IsLocalized
(LocalizedID)
LocalizedUser
Language
Id
Role
Name

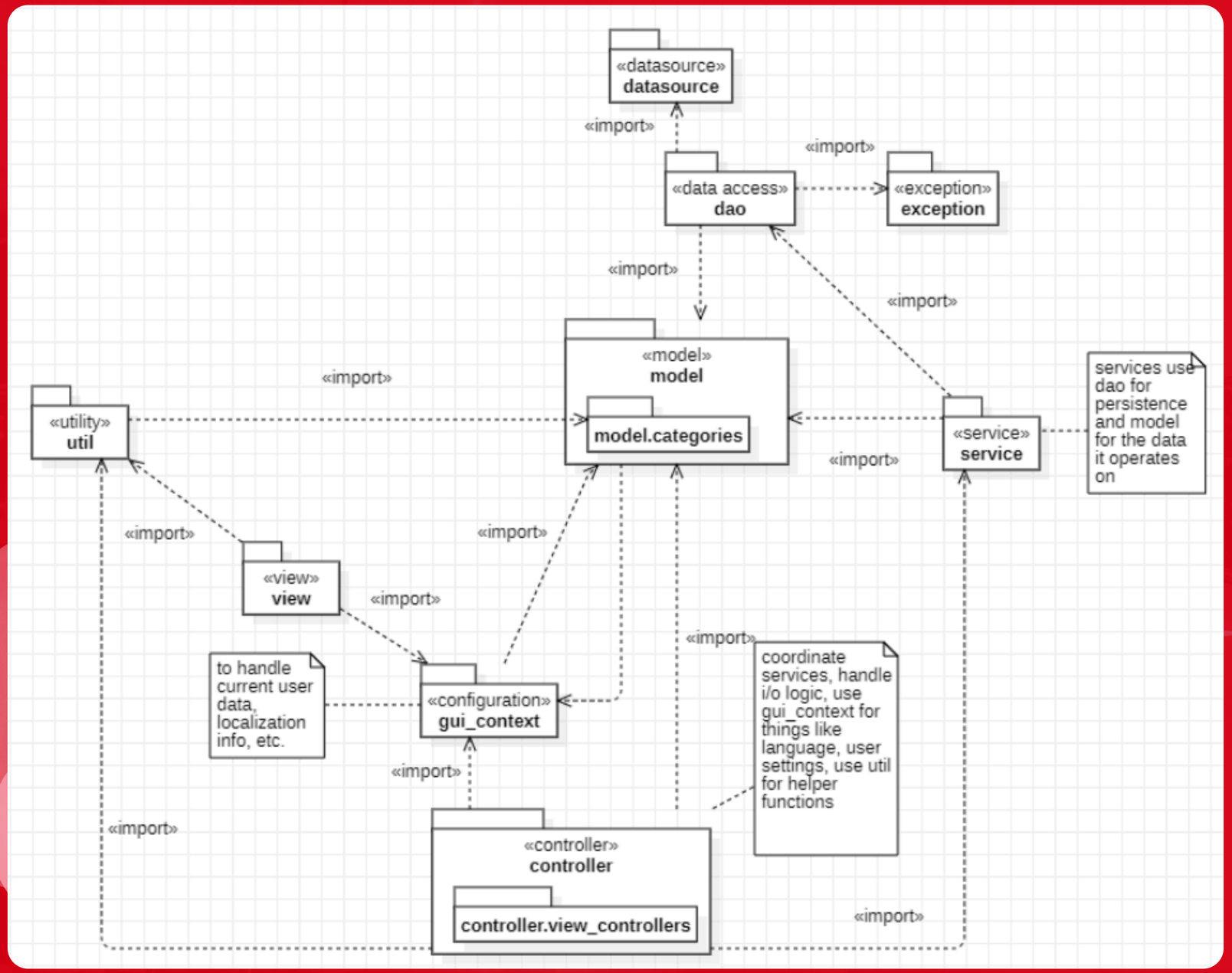
Relational Database Diagram



Class Diagram



Package Diagram



Technologies Used

Testing & Code Quality

- JUnit 5 (Unit testing)
- Mockito (Testing framework)
- JaCoCo (Code coverage)
- SpotBugs, Checkstyle, PMD, SonarQube (Static analysis)

DevOps / CI-CD

- Docker (Containerization)
- Jenkins (CI/CD)

Backend & Persistence

- Java 21
- JPA (ORM)
- Hibernate (JPA implementation)
- MariaDB (Database)

Build Tool

- Maven (Build tool)

Version Control & Collaboration

- GitHub (Version control)
- Trello (Task tracking)

Frontend

- JavaFX 23.0.2 (UI framework)
- TestFX (UI testing)
- VcXsrv (X server for Docker GUI)

How the Technologies Were Used

This project uses Java 21 and JavaFX to create the localized user interface, with Maven for handling dependencies. Hibernate and Jakarta Persistence API manage multilingual data storage in MariaDB. JUnit 5 and Mockito are used for functional and acceptance testing, while TestFX verifies UI behavior across languages. Docker is used to containerize the app for deployment, and Jenkins automates builds and code analysis. VcXsrv enables GUI display when running the application in Docker on Windows.

Localization

User Interface Localization

Supported Languages:

- English (en-US)
- Finnish (fi-FI)
- Japanese (ja-JP)
- Simplified Chinese (zh-CN)

Implementation Details

- JavaFX UI text is managed through ResourceBundle
- Each language has its own .properties file e.g Messages_fi_FI.properties
- Users choose a language at first launch or in profile settings

Dynamic Language Changing

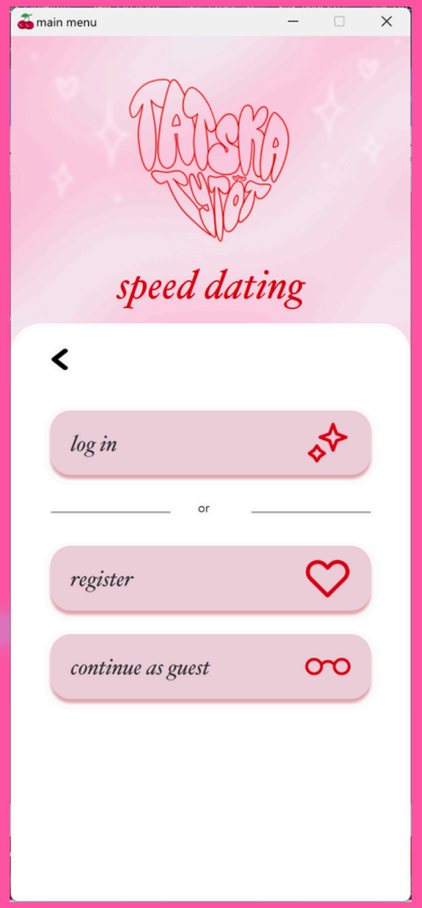
- Language can be changed dynamically at runtime
- No restart required as the UI updates instantly

Benefits

- Inclusive user experience
- Simple to expand with new languages in future

Localization

fi-FI



en-US

ja-JP

zh-CN

Localization

Database Localization

LocalizedUser Table

- Stores translations for user-specific fields:
  - userName
  - roleTranslation
- Linked to the User entity via a foreign key.
- Ensures one translation per language per user using a unique constraint on (user_id, language).

Multilingual Data Support

- Names, emails, and interests can include international characters e.g.张伟@例子.cn
- Relevant for both users and guests

Technical Setup

- All relevant fields use utf8mb4 encoding
- Collation: utf8mb4_general_ci

Benefits

- Prevents character corruption
- Enables global use and testing with localized data

Scalability

- Database is prepared for potential i18n features like localized content storage

Quality Assurance

Usability Evaluation (Nielsen's Heuristics)

- 13 issues identified
- Examples: lack of back navigation, inconsistent UI elements, session loss, unclear language
- Severity ranged from 0 (minor) to 4 (critical)
- Fixes proposed: persistent selections, clearer labels, better navigation, error prevention

User Acceptance Testing

- 10 scenarios tested by all group members
- Includes: registration, matching, profile editing, admin actions, error handling
- All tests passed
- Testers were internal, so future testing with real users is recommended for broader insights

Programmatic Testing

- JUnit tests were used to verify that core features like login, registration, and matchmaking functioned correctly throughout development. Using JUnit 5, Mockito, and TestFX, tests ensured the stability of backend logic and parts of the UI. They helped catch bugs, supported safe code refactoring, and reduced the risk of regressions. Overall, unit testing was a key part of our quality assurance strategy.

Statistical Code Review Summary

- **Tools Used:** Checkstyle, PMD, SpotBugs
- **Total Issues Detected:** ~565

Checkstyle: 328 warnings,
PMD: 122 issues,
SpotBugs: 115 bugs

- Code Complexity: 2 methods exceeded cyclomatic complexity
- Long Methods: 7 methods exceeded 30 lines
- Duplicate/Dead Code: Some unused methods and parameters
- Logging Issues: 70+ System.out.println calls used instead of proper logging
- Mutability Risks: 70 SpotBugs issues from exposing/storing mutable object references
- Thread Safety: Lazy initialization of static fields without synchronization
- Internationalization Flaws: Improper use of toUpperCase() / toLowerCase() with locale-sensitive data
- Performance Flags: Many FXML methods incorrectly flagged as unused

Lessons Learned

Localization

- Creating resource bundles made UI translations manageable.
- Supporting multilingual data in the database required careful design e.g. LocalizedUser table.
- Locale switching without restarting the app added UX complexity.

Code Quality & Refactoring

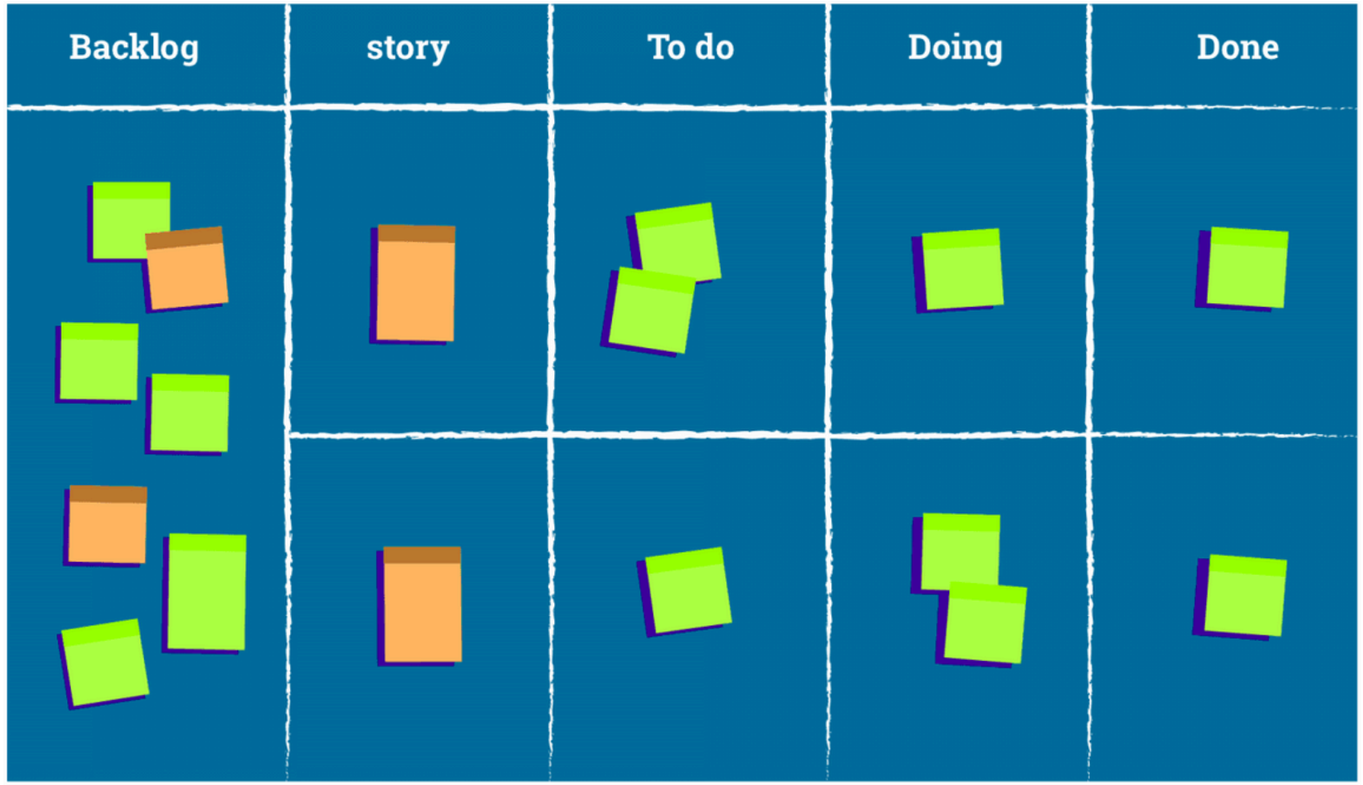
- SpotBugs, Checkstyle, and SonarQube helped catch unused code and maintain structure.
- Refactoring improved readability and reduced technical debt

Finalization & Documentation

- Creating UML/ER diagrams clarified our architecture.
- Writing final documentation early saved time before submission.

Acceptance Testing

- Defining clear criteria helped validate core functionality.
- Simulating user flow revealed usability improvements.



- **Trello**

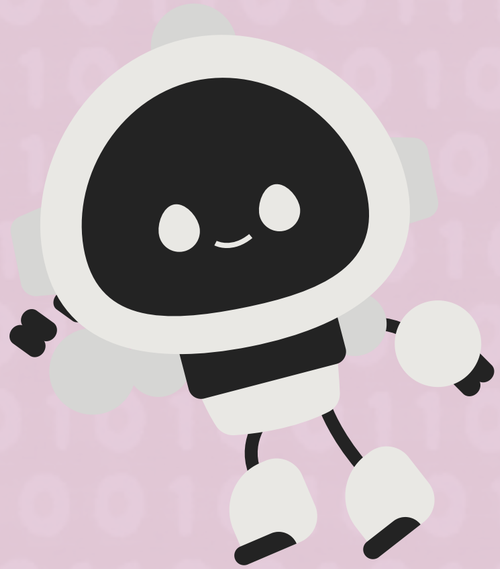
https://trello.com/b/yD4DRsqo/product-backlog

- **GitHub**

https://github.com/Goliathuzzzz/OTP_Group6

Relevant Links

Applied AI Tools



- **Trello**
  Link to product backlog.
- **Github**
  Link to Github repo.
- **AI tools**
  GitHub Copilot, ChatGPT, Claude

- **AI tools**
  Were used to write user stories and acceptance criteria based on sprint requirements.

  They were also used to generate boilerplate code, fix bugs, and address issues found through statistical code analysis.

  Also during sprint planning, AI was used to assign tasks based on the project's user stories and to set deadlines.

Our Team

ADE AIHO

DEVELOPER

HETA HARTZELL

DEVELOPER

MIKA LAAKKONEN

DEVELOPER

JONNE ROPONEN

DEVELOPER

Thank you!