ForYouShipment
Project Report
Group 6

Dušana Milinković (s194741),
Emma Andreea Chirlomez (s194780),
Paulo Lima (s194729), Dario Cannistra (s194830),
Renjue Sun(s181294) and Diana Podoroghin (s194768)

02160 Agile Object-oriented Software Development
Danmarks Tekniske Universitet
May 2, 2021

Contents

1 Introduction

This project aims to represent the development of *ForYouShipment*, a Remote Container Management System. The main goal of the system is to be a point of contact between a logistic company and its clients. We decided to implement this software as a Web Application which is currently deployed on the cloud, running in a *Heroku* instance.

Our team implemented all the mandatory and 4 of the suggested optional tasks: $O1$, $O2$, $O4$, and $O5$.

This report describes the process of the project's development based on 2 very important principles: Agile Development and Object-Orientation.

2 A First Agile Approach

2.1 User Stories

User Stories are informal explanations of what a user needs the software to do. They are the base that we are required to follow in order to implement appropriate functionalities. In our project, we have two types of users - a logistic company and its clients. Here are the User Stories from both of them:

As a logistic company:

- **M1:** I want to be able to add new clients so that they can log in to the system.
- **M1:** I want to be able to search for clients based on simple criteria so that I can easily get all the info about the specific client.
- **M2:** I want to be able to update the journey information so that I can guarantee my clients perfectly valid journey information at any time.
- **M3:** I want to be able to edit the internal status of the container on every journey so that I can keep the information of the container up to date.
- **M3:** I want to be able to mark containers as "on a journey" or "free" so that I can reuse the same container multiple times.
- **M4:** I want to have access to a web page so that I can do all the required actions concerning my clients, their journeys, and the containers corresponding to them.
- **O1:** I want the system to store all the data of the specific container (all of its journeys and its internal status) so that I can access all information concerning one specific container and have a clear overview of it.
- **O2:** I want to have access to all the containers and their information through all their journeys so that I can have a complete overview of all containers.
- **O4:** I would like my clients' profiles, containers' and journeys' information to be saved in a system so that I don't lose all the data when I close the application (all data should remain intact).
- **O5:** I want to have all the ports marked on the World map so that I can get a nice visualization of all the places I have my containers.

As a client:

- **M1:** I want to have a unique username and a valid password, so I can use it for login.
- **M1:** I want to have a profile so that all my activities are saved and I can access them whenever I want.
- **M1:** I want to be able to update my profile so that my information is always valid.
- **M2:** I want to be able to register containers for my journeys so that they can be shipped.

- **M2:** I want to be able to track my journeys so that I always know where they are.
- **M2:** I want each journey to have an ID so that I can share it with my customers and that they can be identified.
- **M2:** I want to be able to find my journeys based on some simple criteria so that I can have a clear overview of the complete journey's info.
- **M3:** I want to be able to see the internal status of the container on my journey so that I can constantly keep the track of its internal status.
- **M4:** I want to be able to have access to a website so that I can create and get information on my journeys and see and change my personal information through it.
- **M4:** I want to access the home page of the logistic website so that I can see my package's information at any time by entering the corresponding ID.
- **O1:** I want to see the current and the previous container's internal status during my journey so that I can have a clear overview of all the measurements over the time of the journey.
- **O2:** I want my container's info to be visible to someone else only if I give permission so that I can provide the access to my customers, but still, have the information hidden for everybody else.
- **O2:** I want to have insight into all of my containers and the information corresponding to them so that I can have a clear picture of all my containers.
- **O5:** I want to see the location of my container on the World Map at any point so that I can easily follow its path to the destination.

2.2 UML class diagram

After finishing the User Stories and understanding the needs of each user, we started developing a first drawing of the *UML class diagram*. Notice that the spring framework forced us to implement our code on the base of the MVC pattern, which will be discussed in the following sections.

The core of our program depends on the following classes:

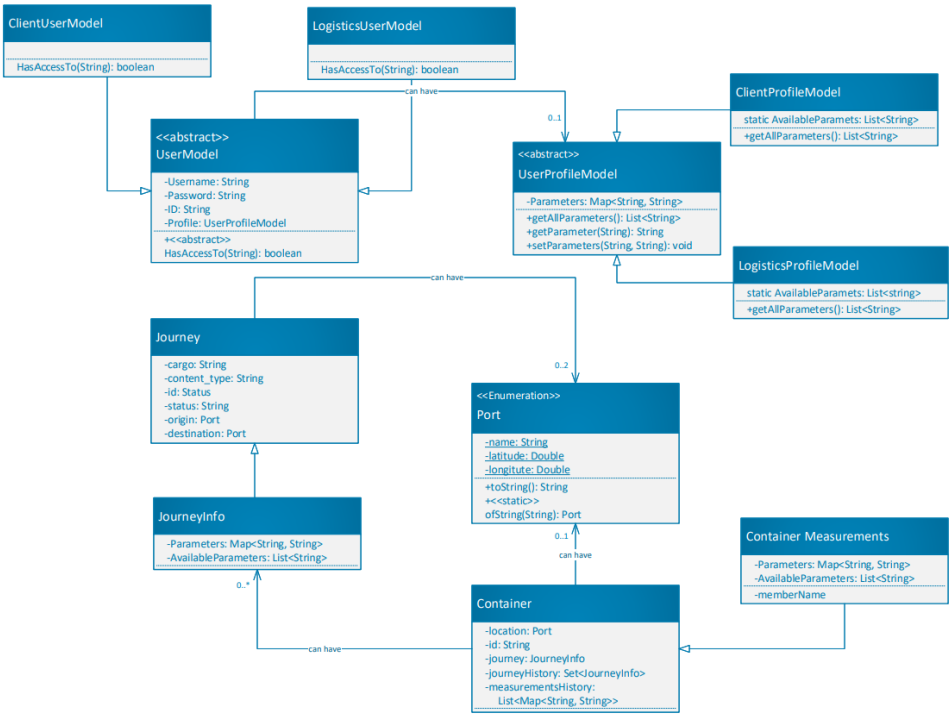

Figure 1: UML for the Model classes

As you can see from Figure 2 each *UserModel* can have one *UserProfileModel*. A UserModel is a class containing information about the login credentials of both client and logistic company. UserProfileModel is used for storing relevant information about clients (such as first name, last name, email, etc..). These classes can be accessed both by the client (which is allowed to see and edit just his/her information), and also by the logistic company (which is allowed to register new clients).

After a client is registered, he/she can request a new journey. When a journey is approved a container is linked to it, as you can always see from the figure above. In addition, Container measurements is extending the container class, this class is used by the logistic user to inform their clients about the latest status and measurements updates of their respective container.

This is a very brief description of our UML diagram, but we will elaborate more during the presentation, where we will talk about the mandatory and optional tasks in a more detailed way, since it is easier to explain.

To see the complete UML with all the dependencies you can follow this link.

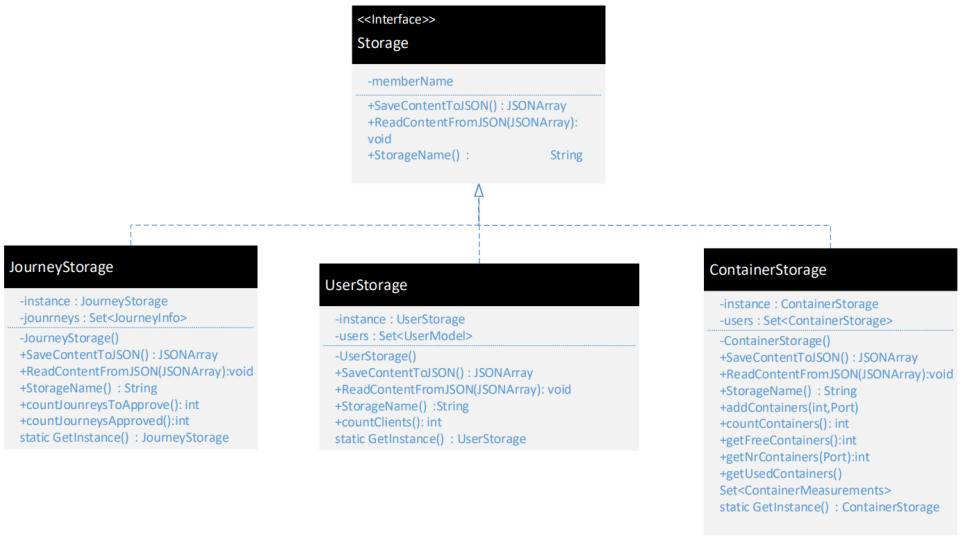

Figure 2: UML for the Storage classes

3 Agile implementation of the code

3.1 BDD - Behaviour Driven Development

BDD is known as the abbreviation of 'Behavior Driven Development'. Particularly, before starting developing, an agile team first thinks about multiple feasible scenarios that can fulfill each user story. BDD makes developers shift from thinking in "test" to thinking in "behavior".[3] BDD is an extension of TDD, which additionally helped our group to understand each other's work better and to acquire a better project overview.

Cucumber tests are a popular tool for BDD development. We started the project by writing down the user stories and all of the required **Cucumber** tests in **Gherkin**, but didn't end up using them during the project, due to the conflicts between **JUnit**, **Jacoco** and **Spring Boot Tests**.

Our **Cucumber** tests cover all functionalities for $M1$.

After implementing **M1** using *BDD*, we switched to a *TDD* approach, as *Spring Boot* integrates better with *JUnit* than *Jacoco*.

3.2 TDD - Test Driven Development

TDD, known as the abbreviation of 'Test Driven Development', which pushes the developer to only implement what is minimally needed to implement a feature, so it can help us shape our design to actual or real use and avoids any gold plating in our implementation [1].

TDD is an iterative development process. Each iteration starts with a set of tests written for a new piece of feature/functionality. These tests are supposed to fail at the start of iteration as there is no code implemented for the tests. In the next phase of the iteration, application code is written with an intention to pass all tests. The iteration will continue until all tests are passed by implementing and refactoring of the code.[5] Once the iteration ends, a feature is completed.

After implementing M1, we decided to follow the *TDD* concept, as the *Spring Boot* framework, *Maven* and *JUnit* work nicely together. **JUnit** is an excellent Java framework for unit testing, which we used for all of our unit tests.[4] We have a coverage of 100% of business-logic related classes, and 99% on other classes like controllers.

Our **JUnit** tests expand in over 120 unit tests, most of which require complex web requests, sessions and server mocking.

4 Software Architecture and Design

4.1 Environment and Workflow

We used a wide variety of tools, named in the following:

- We used *Git* as a control software, where we stored out repository. We also used *Github* for our *Continuous Integration* workflow, and our *Github Project* boards.
- As *IDE*, we used both *Visual Studio Code* and *InteliJ*. The main advantage of *VSCode* is the speed and flexibility, while *InteliJ* has great code auto-generation tools.
- As a build tool we used *Apache Maven*, which being compatible cross-platform and cross-IDE, made the debug process easier and allowed us to use the *CI Github Workflow*.
- We also used the *Spring MVC* framework, which offered both flexibility and powerful tools to our project.
- We tested our project on Windows, *Linux* and *Mac OS*, on all the main Internet browsers (*Chrome*, *Safari*, *Firefox* and *Safari*).

4.2 Model-View-Controller Architecture

As our project has a large number of features and moving parts, we decided to use the *Model-View-Controller* Architecture.

Figure 3 illustrates the primary flow of request handling in our project, which respects the MVC data flow.

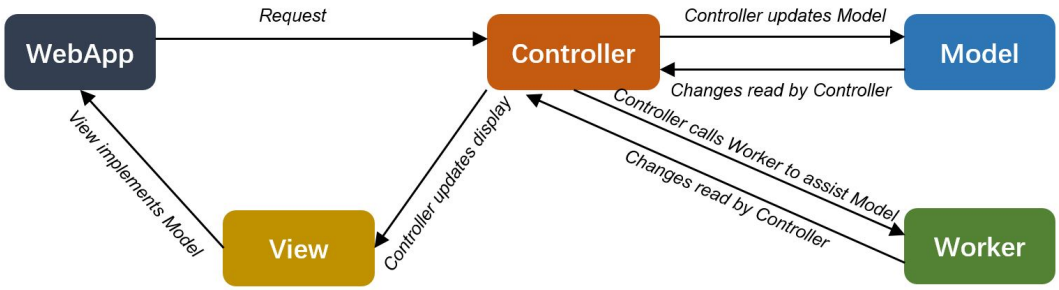

Figure 3: Spring MVC

We divided our entities into 7 main categories, which corresponds to the packages used in Java :

- **Models** - Entities in charge of storing entries, like a user or a journey. Note that this entities are not allowed to perform any actions besides changing their state.

- **Storages** - Entities in charge of storing collections of Models. Storages also have the ability to be saved to a persistent memory.
- **Controllers** - Controller classes are the bridge between the front-end and the back-end, forwarding requests to workers and responding with the appropriate HTML pages.
- **Workers** - Hidden classes implementing the business logic; they are all based on static methods.
- **View** - Files written in the *JSP* language, rendering *HTML* pages.
- **Constants** - Static classes and enumerations storing global constants.
- **Persistence** - Set of classes able to serialize and retrieve *Storages* to and from memory.

Note that this modular approach made the code easier to understand. Large classes or methods were refactored into smaller ones when possible, and meaningful documentation made the code easier to follow. The usage of the described MVC architecture gave each component a different meaning, which together with meaningful variables, classes and methods naming and the single responsibility principle made any bugs easy to pinpoint and fix.

4.3 Persistence

We implemented a persistence layer for our application. Custom persistence classes serialize the storages into JSON files, which are stored on the hard drive by a back-up service, which is running on a separate thread and saves the data at regular intervals.

4.4 Filter Design Pattern

Our application is designed to be used by regular customers, for which we needed a powerful, flexible and easy-to-use search system. We decided to filter classes for either search among a list of clients, journeys or containers using a Generic Interface. Filter pattern is a efficient pattern because it allows to combine multiple criteria through logical operations.

4.5 Singleton and Static Classes

Our software architecture requires many stateless classes, such as workers and classes with a single instance such as storages. We used static and singleton classes wherever it made sense, to keep the readability of the code.

4.6 Factory Design Pattern

Due to the polymorphic nature of our models (for instance, we have a basic *User* abstract class, inherited by both clients and administrators), we found the need for factories. We implemented a factory for each polymorphic type, which was able to construct an object based on its JSON representation.

4.7 Object Orientation

Throughout the development of the software we stuck to the *OOP* principles, such as:

- *Encapsulation* - We made sure to use proper access modifiers to prevent any undesirable modification of the data.
- *Data Aggregation / Composition* - We properly used aggregation and composition for member fields, based on their utilisation, to both prevent corruption of data and improve the application's performance.
- *Polymorphism and Inheritance* - We used abstract classes (such as *UserModel* and *UserProfile*) to avoid code repetition, interfaces (such as *Storage*), to infer meaning to particular classes and regular inheritance (one example of inheritance would be all the controllers, inheriting a *BaseController* class implementing authentication mechanisms.

4.8 SOLID Principles

When it comes to writing quality software, three important questions arise. How do we build software quality? What is the difference between high-quality and low-quality code? How to start with the implementation of such a solution? The answer is given by the SOLID principles, the backbone of object-oriented design.

Single Responsibility Principle

"A class should only have one reason to change."

The single responsibility principle is mainly handled by the MVC architecture. We made sure to not break the principle, to keep the code organized and easy to understand.

Open Closed Principle

"Software entities should be open for extension, but closed for modification."

While our code is flexible, easy to adapt and can easily integrate new features, each implemented class has a particular objective, independent of the rest of the code. While we did some refactoring which modified already implemented classes, we didn't have to alter classes when adding new features. For example, when we tackled the M3 features we just exending the already existed Container class without the need to modify it.

Liskov Substitution Principle

"Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."

It should be possible to replace an object with an instance of its subclass, while not changing the behavior of the program. That is, a class should be replaceable by any of its sub classes without any errors. The child class should have ability of doing all jobs as its parent class.

We applied this principle by applying polymorphic types, such as *UserModel* abstract class.

Interface Segregation Principle

"Make fine grained interfaces that are client specific."

The interface should have only as many functions as the class that implements it would use. Otherwise, the class will have unnecessary functions that won't be used. Therefore, one should analyze the interface and the classes that would implement it before creating the interface itself. This is how the readability of the code is improved.

We used the Storage Interface, in such a way that they can be serialized to memory, by the Persistence Workers.

Dependency Inversion Principle

"Depend on abstractions, not on concretions."

High-level code shouldn't depend on low-level code, but both should depend on abstractions. Abstraction only says what a class should do, not how. We used abstraction to bring important portions of the code to a high-level implementation, where features such as a user's particular profile, authentication or a journey's history was abstracted away.

4.9 Refactoring

As with any application, we had to refactor the code multiple times though-out the development process. One of the main goals of refactoring was to obtain a code following the SOLID principles:

- We split long sections of code to have a better readability. The main target of doing so was to increment the quality of the refactored code focusing on Single-Responsibility principle and the Open Close one.
- As we added more and more optional features to our project, we had to rethink our UML diagrams, and as such, refactor our code to reflect them.
- We were recommended to split the controllers into two different entities: Shallow controllers and facades. However, we reverted back to the original controller design, as the code lost readability.

5 Features

The management system was constructed in several phases. The fundamental one is to design the business objects (Container, Client and Port) as well as to determine the amount of information carried by them, where the client name will be able to delegate responsibilities to other objects.

The following step was to create an abstract structure of the future web application. We agreed on splitting the users in two different categories of logistic company and its clients along with distant features accessible for them. For instance, the logistic company can: add new clients or search through them and update containers' location along with their internal conditions. If we are looking from clients' side of view, he is able to update his own information, register new journeys and to visualize the existing data.

Along with mandatory features we have implemented the following additional ones:

- After login, a client is able to search through the database finding specific journeys by entering their origin, destination, content type, company name as well as applicable details related to the container status.
- For storing different business objects in the database, a persistence layer was created.
- The client is also able to visualize his journeys waiting for approval and the active ones having been approved by the logistic company. At the end of the trip, there is displayed destination and the last updated date for the cargo.
- The progress of the internal changes (pressure,temperature,humidity) are updated by the logistic company. The changing history is stored and can be seen.
- A user-friendly website capable for the login and possible operations and checking for both the logistic company user and the client user.

By looking at the implemented features, one can see that the *ForYouShimpent* Web Application implements the optional features $O1$, $O2$, $O4$ and $O5$ in addition to the mandatory tasks ($M1$ to $M4$).

6 Project management

6.1 Work Distribution

We decided to divide the workload equally between all members - we were switching roles regularly so that all of us participated in all parts. This way we ensured that all of us understand everything we did as a group. Our typical working week was split as follow:

- Every Wednesday we were having a meeting right after the lecture where we were brainstorming about the work that had to be done.
- We were giving ourselves some time for analyzing and studying the new material. This way, when we were meeting the next time, we had some ideas for possible implementations and improvements.

- During the weekend, we would have a full-day of coding. First half an hour we used for determining what are we going to work on during the day. We would split into three groups and start coding. Every pair would switch roles every so often. After every two hours, we would reorganize pairs. This didn't work as perfectly as we wanted, but it was something we were highly aiming for anyway.

6.2 Pair programming

Taking in consideration our current global situation we didn't manage to meet in person. We met twice or more per week on Teams. Working in pairs, we get used to each others approaches and learned a lot from each others. Every half an hour the other one would share its screen. While one is sharing the screen, the other has the control and both cursors are displayed on the screen. This Teams' option helped as a lot and saved us a lot of time. Every 2 to 3 hours we would reorganize the pairs. In order to switch group for one problem, a previous group had to write nice comments and explanations of all that they did while they were coding. At the beginning, it was hard to get used to this approach but after some time we became very comfortable with it.

6.3 Version control

Version control software is used to keep track of, manage, and secure changes to file [2]. As all of our group members have already had some experience with Git, we agreed to use Git for our version control and host a repository on GitHub.

To implement our stories, we were first opening a new issue on GitHub, this issue was then assigned to some group members and was moved into a project. Each project was representing a single program functionality, where we were going to add our issue/user stories.

User stories were all assigned to a *'To do'* section at the beginning. As the project went on, we moved them to *'In progress'* and *'Done'* timely. Enhancement and bug handling are also important during the development. When we encountered bugs needed fixing and thought of doing enhancement to improve software quality, we followed the same tracking process as we did with user stories. The progress tracking is described in Figure 4. We actually had more functionalities but to keep it simple, we only took two (M1 and M2) as examples.

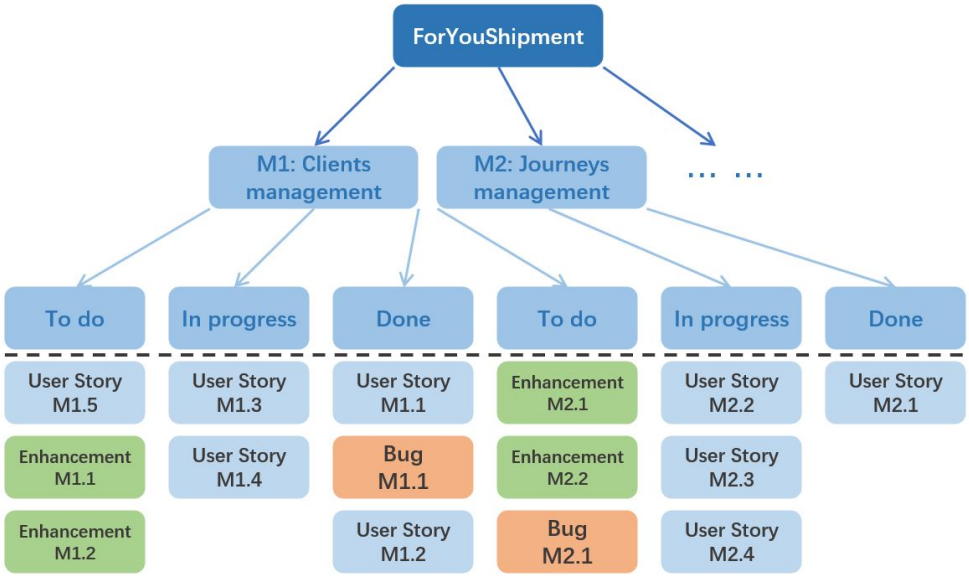

Figure 4: Work breakdown and progress tracking

At team meetings, we would decide to assign different tasks under 'To do' to group members and then moved them to 'In progress'. For each issue under 'In progress' we had one or more divided branches for group members to work on.

7 User manual

Our web application provides access to two different kinds of users: logistic company user and clients of the logistic company. Link to the user manual

8 Links

- Github Repo: Github - ForYouShipment
- Software Releases: Github Releases
- Deployed WebServer: http://foryoushipment.ml/
- Jira Boards: Github Projects
- Source Code: WebApp - Github
- Unit Tests: Unit Tests - Junit
- Cucumber Tests: Test definition Test implementation
- Screen Cast on YouTube: ForYouShipment Demo

9 Conclusion and considerations

During the work that we have done on this project we learnt a lot about agile methodology as well as programming. We realized how powerful GitHub can be and how useful is to practice continuous integration when working in group. Sometimes we found it difficult to implement features and work together without meeting in person but the challenges made us more motivated. In addition, we also encountered some difficulties with unit tests. As mentioned before, it was very complicated to test the web application, but after some time spent on it we managed to get a good coverage overall. We really understood the importance of agile methodology because it can save us a lot of time, especially when working in a big group. We will for sure treasure all the gained knowledge to make our future projects in this field as much optimized as possible. To sum it up, we managed to work productively and delivered a functional web application following TDD at the end.

10 Appendix

References

[1] Nolan Godfrey. *Agile Swift*, chapter Test Driven Development. Apress, 2016.

[2] Davis Adam L. *Modern Programming Made Easy*, chapter Version Control. Apress, 2020.

[3] Virender Singh. Behavior driven development, . URL https://www.toolsqa.com/cucumber/behavior-driven-development/.

[4] Virender Singh. Junit introduction, . URL https://www.toolsqa.com/java/junit-framework/junit-introduction/.

[5] Virender Singh. Test driven development (tdd), . URL https://www.toolsqa.com/cucumber/test-driven-development-tdd/.