Delft University of Technology

Software Engineering Methods: Assignment 1.2

CSE2115 Group 31b

Alexandra Darie	5511100
Elena Dumitrescu	5527236
Robert Vadastreanu	5508096
Sander Vermeulen	5482127
Xingyu Han	5343755
Zoya van Meel	5114470

December 23, 2022

1 Introduction

For this assignment, our task is to implement at least two design patterns that were covered during the lectures. We have applied three design patterns in total, one for each core domain. In the following sections, we will motivate our choices and explain the exact implementation of the design patterns. Moreover, we have provided class diagrams for a better overview of the patterns' structures.

2 User Microservice - Facade

The User microservice was designed using the **facade pattern**. The UML class diagram is shown in Figure 1. The facade pattern serves to improve the usability and readability of the User microservice. End-users interact with the UserController (the facade) and the UserController calls on the appropriate Services for actual functionality. There are three Services implemented in the UserController: the NotificationService, the UserService, and the OwnerService. The endpoint that a user accesses, will influence which service is called by the facade. The operations in each service are hidden from the facade and can easily be changed and swapped out if needed. In Figure 1, it can be seen that the functionality of each Service can be complex, while only having a single connection with the facade. Each Service is structured by implementing the correct Service interface and using the appropriate Model classes. These classes are representations of different types of user input.

The NotificationService handles functionality related to pulling incoming notifications from the Scheduler microservice. If the communication with the Scheduler ever changes, the UserController will remain unaffected and only the NotificationService is changed. This same principle can be applied to all other Services. The UserService handles interaction for everything related to users. This means the handling of personal information and applying to events. The OwnerService handles application interaction from Owners. This relates to creating events and approving applicants to events they have created.

3 Scheduler Microservice - Chain of Responsibility

The second design pattern implemented in our application is the **chain of responsibility**. The UML class diagram is shown in Figure 2. Our **choice** to use this design pattern is motivated by the significant improvement in terms of scalability and extensibility that our code benefits from. This design pattern is applied in the context of filtering events: when a user makes a request to join a competition or training, the events need to be filtered based on the person's preferences and capabilities before being selected as a potential match. The chain of responsibility aims to simplify a potential addition to the filtering measures. As an example, if, in the future, competitions need more requirements besides gender, organization, and professional status, this can be easily modified considering our current implementation.

In order to **implement** this design pattern, we made use of multiple classes. Firstly, we use an interface called Validator. This interface ensures that all our validators will implement the handle() and setNext() methods, which represent vital functionality for building the chain. Secondly, we use the abstract class BaseValidator that implements our interface. The BaseValidator class implements the setNext() method, as well as another method called checkNext(), which moves the filtering to the next part of the chain via next.handle(). Lastly, we implemented the actual chain parts using validators that extend the BaseValidator. The three current validators, AvailabilityValidator, CertificateValidator, and CompetitionValidator represent the most important part of the filtering and contain the actual functionality of selecting the proper events. The chain is instantiated in the Event Service, after the controller calls the utility method filterEvents(). In the future, if needed, extending the chain with another validator for more complex filtering is very easy.

4 Event Microservice - Builder

For the event microservice, we have decided to implement the **builder design pattern**. The UML class diagram can be seen in Figure 3. We have chosen to do this because the event class is quite a large and complex object, while with the builder design pattern we were able to separate the construction of the object so that it was unambiguous and more legible for anyone who reads the source code. It is also the case that we have two different types of events, a training and a competition, and with the builder pattern, we were able to allow the different representations of events to be created, while not having to duplicate any code and simply using the respective instance of the builder. Finally, our Event class will act as the director, with the "parseModel" function doing the conversion from raw data to an Event object.

We have made the builder in three main parts: First of all, we have an interface called "Builder", which simply defines all of the methods for the builder. We then decided to make an abstract "EventBuilder", which implements all methods which are equal for both types of events (training and competitions). Finally, we have added two classes, each for their respective type, called "TrainingEventBuilder" and "CompetitionEventBuilder". These then implement the methods that do *not* share any functionality with each other. In practice, this means that our abstract EventBuilder does most of the heavy lifting, with the Training- and CompetitionEventBuilder only implementing one method which differs between the two. This is also reflected in the class diagram.

A Figures

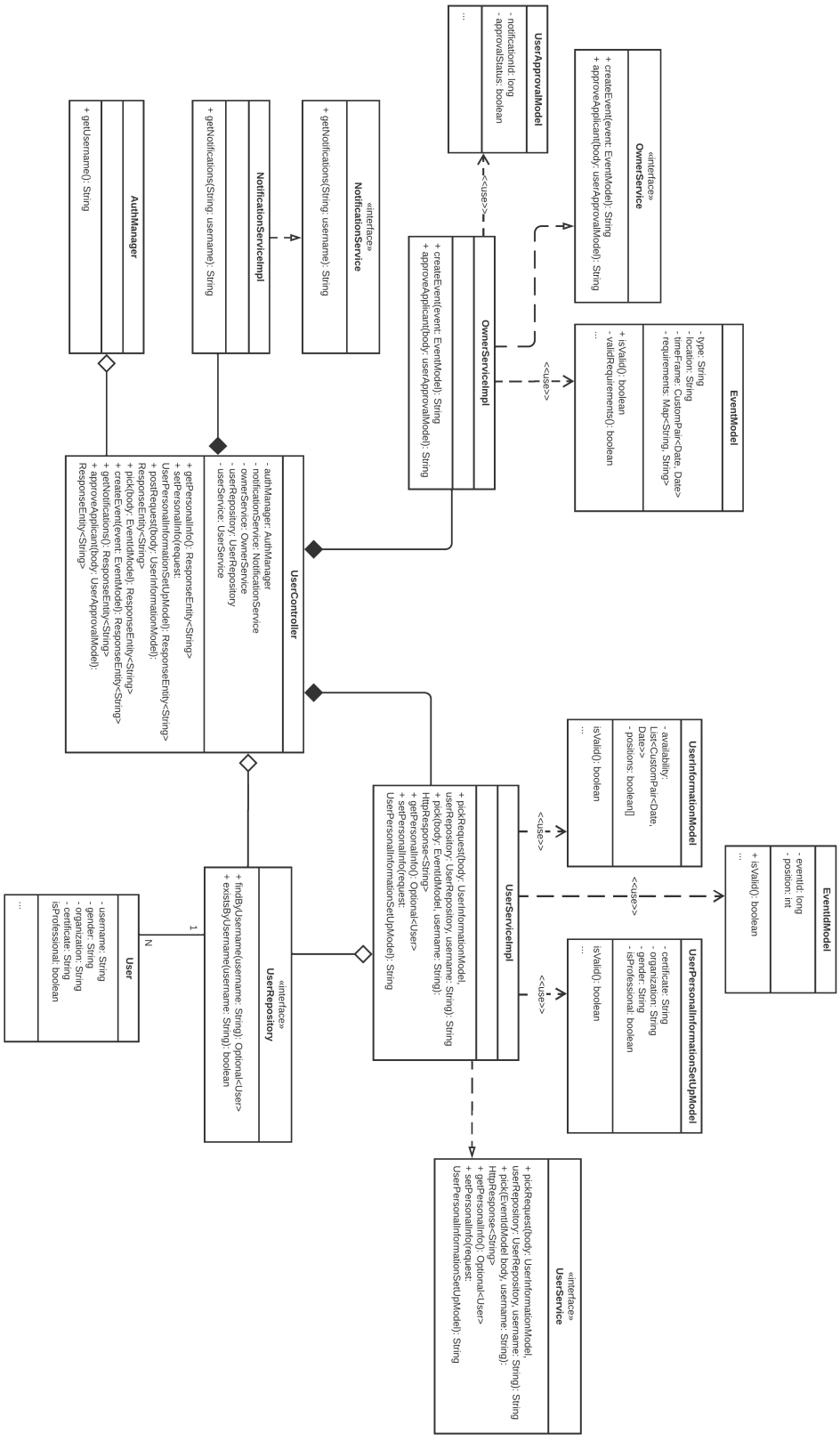

Figure 1: UML class diagram for the User microservice. Several classes include ellipses (...) in their methods. These are to indicate getter, setter, equals, hashcode, and toString methods.

Figure 2: UML class diagram for the Scheduler microservice.

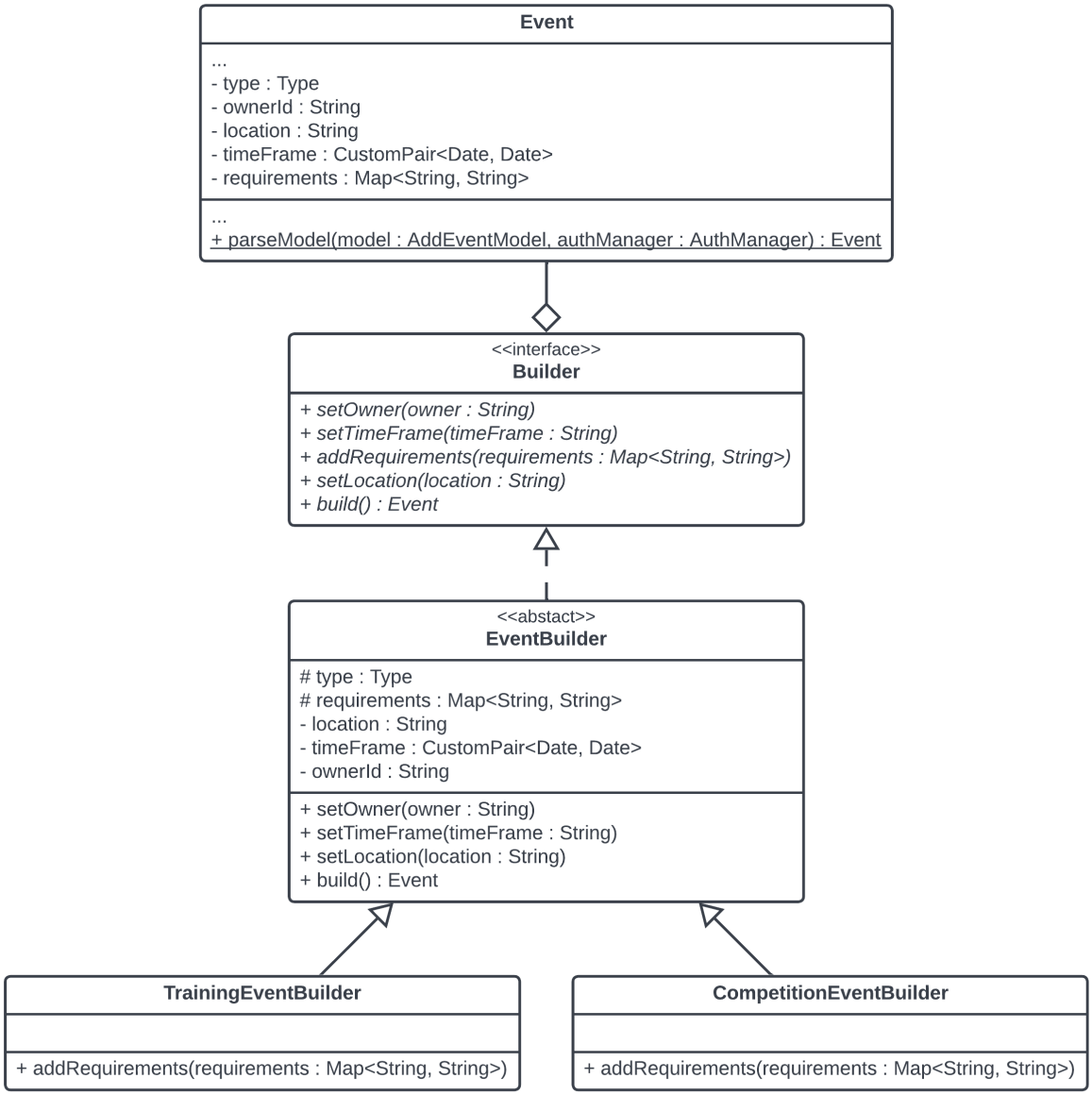

Figure 3: UML class diagram for the Event microservice. For the event class, some insignificant fields and methods (including getters/setters) have been left out for the sake of clarity.