



CSE2115 Software Engineering Methods

Assignment 1 - Task 2

AUTHORS

Bogdan-Andrei Bancuta - 5507340
Alexandru Gabriel Cojocaru - 5520967
Khalit Gulamov - 5491541
Alexandru-Nicolae Ojica - 5477484
Wing-Yan Joyce Sung - 5011825
Efe Unluyurt - 5452481

23 December 2022

1 Factory design pattern

The factory design pattern is a creational design pattern. It defines an interface for creating objects, but allows subclasses to alter the type of objects that will be created. It provides a way to delegate the instantiating of objects to subclasses while hiding the implementation details of object creation from the client.

This pattern was implemented twice in our project.

1.1 The Producer factory

In Apache Kafka, a producer is a type of program used to send messages to a Kafka broker. It determines which message should be sent to which partition within a topic. In Spring Boot, the `KafkaTemplate` is a high-level abstraction that simplifies sending messages to Kafka topics from within a Spring application. It provides a range of convenient methods for sending messages and is part of the *Spring for Apache Kafka* project, which offers a comprehensive set of APIs for working with Kafka in a Spring application. The `KafkaTemplate` acts as a wrapper around a Kafka producer, providing a more convenient and streamlined way to send messages to Kafka topics. In addition, it reduces the boilerplate code, making it easier to integrate Kafka into a Spring application.

In Spring Boot, the `ProducerFactory` is a factory bean used to create `Producer` instances in the Apache Kafka API. It can be configured with settings such as the bootstrap servers for the Kafka cluster, the serializer for serializing keys and values, and other settings related to producing messages. Using a `ProducerFactory` allows the developers to easily create and configure `Producer` instances in their Spring Boot application. It is a key component of the Kafka support provided by Spring Boot.

These are a few reasons behind the design choice to use a `ProducerFactory` alongside `KafkaTemplate` instead of instantiating a `Producer` every time it was needed:

1. **Convenience:** The `KafkaTemplate` provides a convenient way to send messages to Kafka without dealing with the Kafka API's low-level details. By using a `ProducerFactory` to create the `Producer` instances that the `KafkaTemplate` uses, you can easily configure the `Producers` with the appropriate settings and let the `KafkaTemplate` handle the details of sending the messages.
2. **Performance:** The `KafkaTemplate` is optimized for performance, and it uses a `ProducerFactory` to create `Producer` instances that are also optimized for performance. This can help improve the efficiency of the application when sending messages to Kafka.
3. **Resource management:** The `KafkaTemplate` manages the `Producer` instances' life cycle, ensuring that they are properly closed and resources are released when they are no longer needed. This can help reduce the risk of resource leaks in the application.
4. **Error handling:** The `KafkaTemplate` handles errors that may occur when sending messages to Kafka, including retrying failed sends and logging errors. This can help simplify error handling in the application and make it more robust.

1.1.1 Implementation of the Producer factory

To create a Producer instance, the `ProducerFactory` first creates a `Map` of properties that configure the Producer. Once the properties have been configured, the `ProducerFactory` uses the `KafkaProducer` class to create a new Producer instance every time it is needed, passing in the properties as an argument. The Producer instance is then returned to the caller, ready to be used to send messages to a Kafka topic.

In our application we define 2 beans: `producerFactory()` and `kafkaTemplate()`. By defining the `producerFactory` as a bean we can autowire and use it anywhere in the code (fig. 1).

Then the `kafkaTemplate` bean uses the `producerFactory` bean to configure the Producer (fig. 2).

```
/**
 * Configures the kafka producer with values from application.properties
 *
 * @return properties of the kafka producer
 */
4 usages  ▲ Alexandru Ojica
public Map<String, Object> producerConfig() {
    HashMap<String, Object> props = new HashMap<>();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    if (useAuthentication) {
        props.put("saslm.mechanism", kafkaSaslMechanism);
        props.put("saslm.jaas.config", kafkaJaasConfig);
        props.put("security.protocol", kafkaSecurityProtocol);
    }
    return props;
}

2 usages  ▲ Alexandru Ojica
@Bean
public ProducerFactory<String, String> producerFactory() {
    return new DefaultKafkaProducerFactory<>(producerConfig());
}
```

Figure 1: The `producerFactory` bean

```
1 usage  ▲ Alexandru Ojica
@Bean
public KafkaTemplate<String, String> kafkaTemplate(ProducerFactory<String, String> producerFactory) {
    return new KafkaTemplate<>(producerFactory);
}
```

Figure 2: The `kafkaTemplate` bean

This can be seen graphically represented in the class diagram of the KafkaProducerConfig (fig. 3).

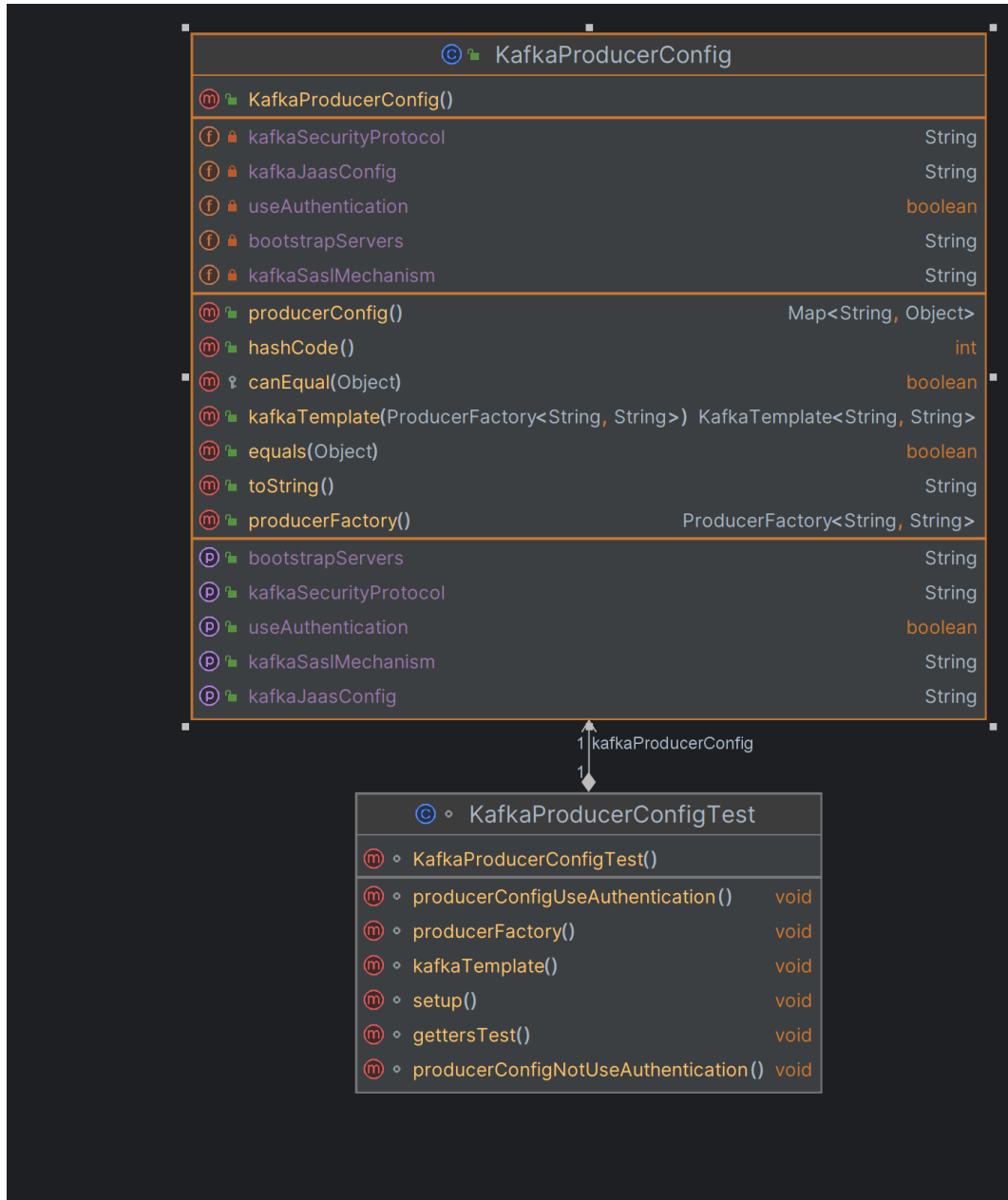


Figure 3: Class diagram of KafkaProducerConfig

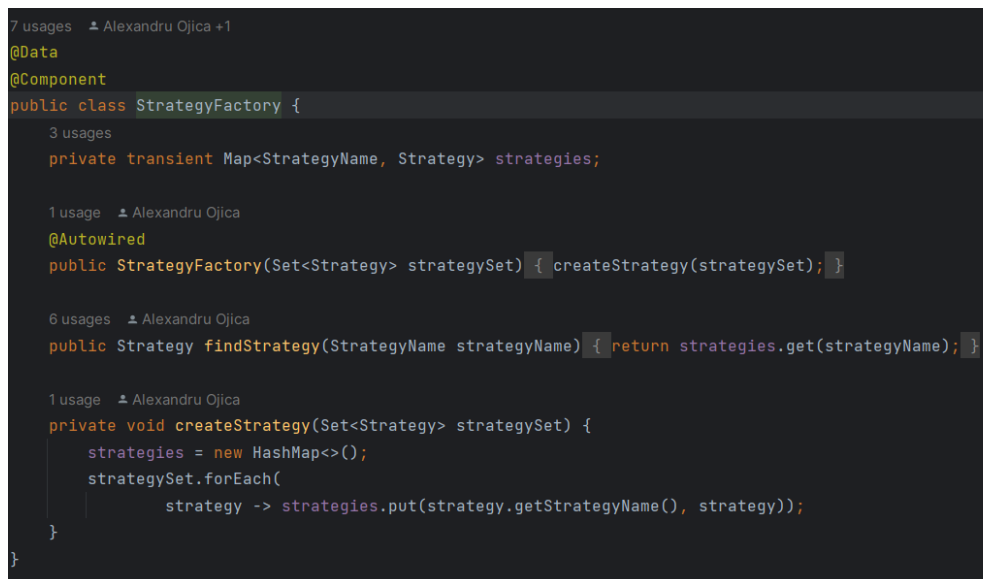
1.2 The strategy factory

1.2.1 Necessity of the strategy factory

We decided to implement the factory design pattern for creating strategies, since as the application grows in terms of active users, more methods, and thus strategies, to send the users notifications will be needed. To this end, we implemented the StrategyFactory, which takes all types of notification strategies defined and builds one of them depending on the needs of the user.

1.2.2 Implementation of the strategy factory

We implemented the StrategyFactory class (fig. 4) and annotated it with @Component so Spring knows how to initialize it. The constructor is Autowired, and by doing this, Spring will find the set of all classes that implement the strategy interface and inject them. To keep track of strategies, each of them corresponds to an enum. The findStrategy method returns the strategy that corresponds to the enum received. The createStrategy method is used in the constructor to initialize the map of Strategies. This is used inside the NotifyUserService depending on the response of another microservice (fig 5).

A screenshot of an IDE showing the code for the StrategyFactory class. The code is written in Java and includes annotations like @Data, @Component, and @Autowired. It features a private transient Map for strategies, a constructor that takes a Set of Strategy objects and calls createStrategy, a findStrategy method that returns a Strategy object based on its name, and a private createStrategy method that initializes the map by iterating over the provided strategies and putting them into the map.

```
7 usages  ▲ Alexandru Ojica +1
@Data
@Component
public class StrategyFactory {
    3 usages
    private transient Map<StrategyName, Strategy> strategies;

    1 usage  ▲ Alexandru Ojica
    @Autowired
    public StrategyFactory(Set<Strategy> strategySet) { createStrategy(strategySet); }

    6 usages  ▲ Alexandru Ojica
    public Strategy findStrategy(StrategyName strategyName) { return strategies.get(strategyName); }

    1 usage  ▲ Alexandru Ojica
    private void createStrategy(Set<Strategy> strategySet) {
        strategies = new HashMap<>();
        strategySet.forEach(
            strategy -> strategies.put(strategy.getStrategyName(), strategy));
    }
}
```

Figure 4: The StrategyFactory class

This can be seen graphically represented in the class diagram of the StrategyFactory (fig. 6).

```

//sending the request
try {
    ResponseEntity<String> response = restTemplate.exchange(uri, HttpMethod.GET, requestHttp, String.class);
    System.out.println(response);
    strategy =
        strategyFactory.findStrategy(StrategyName.EMAIL);
    notification = new Notification(request, response.getBody());
    setVariables(notification);
    strategy.notifyUser(notification);
} catch (RestClientException e) {
    if (e.getMessage().contains("404")) {
        System.out.println(e.getMessage());
        strategy =
            strategyFactory.findStrategy(StrategyName.KAFKA);
        notification = new Notification(request, request.getUsername(), useKafka: true);
        setVariables(notification);
        strategy.notifyUser(notification);
    } else {
        throw e;
    }
}

```

Figure 5: The StrategyFactory used inside NotifyUserService

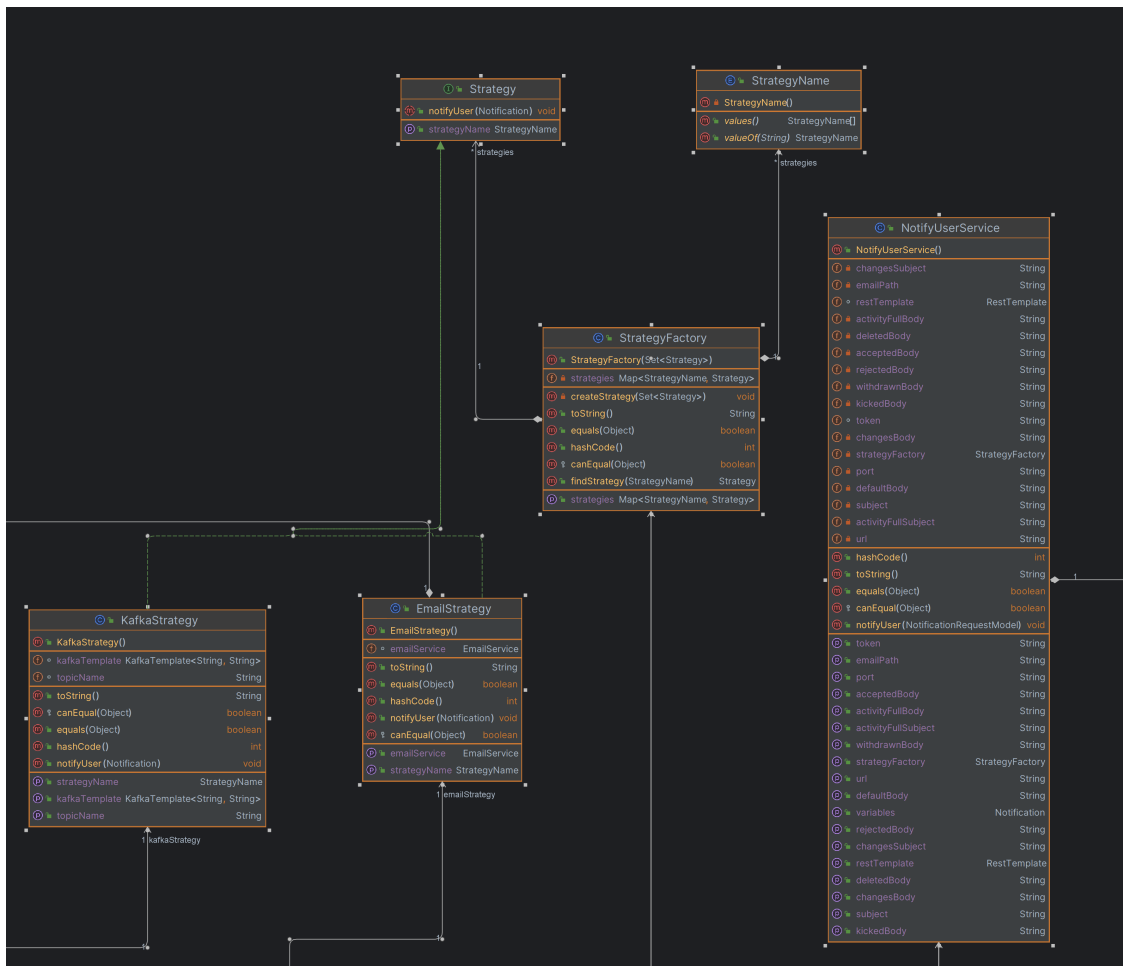


Figure 6: Class diagram of StrategyFactory

2 Builder design pattern

2.1 Necessity

Having different types of activities was one of the main issues for implementing our application. There are the types training, competition and this should be able to be expanded to even more types in future developments. To facilitate this, trainings and competitions need to be grouped to store their information in a database. However, the code should also be kept modular for future developments of the application, in which more activity types might be added. To enable both of these features, we decided to use the builder design pattern to implement the activity structure and its' subtypes.

Builder is a creational design pattern, its' purpose is to enable you to construct complex objects step by step. This is facilitated by the ability to use the same construction code to produce different types and representations of a certain object. The creation of this object is organized into a set of steps by the builder, which when executed in the right order yield the type of object that is required. In the construction of the object it isn't required to call of the steps, but only the ones that are necessary to create a particular object.

To assist our builder we used a director. The director class often accompanies the builder class and enables a clean codebase. A director object leads the product through all the steps of its construction, while the builder provides the implementation for the construction steps. The main advantage gained by using the director is that the details of the object's construction process is abstracted away. This way the client code can create the object, without any concerns about underlying implementation details of the object.

2.2 Implementation of the builder design pattern

The builder design pattern is implemented in our application on the activity class and is used to build two kind of activities: training and competition. A training is a basic activity and a competition is an activity which shares the same properties as training, with a couple of additional requirements. The two classes are instantiated using a builder interface (fig. 7) and then two builder classes which implement this interface (fig. 8). This interface enables us to create a prototype of the functions that should be contained in every activity. Then using the two builder classes TrainingBuilder (fig. 8a) and CompetitionBuilder (fig. 8b) we build instances of activities with additional requirements. An abstract class called Activity ties the two type of activities together and enables us to have a shared repository for the two different types of activities. However, to actually instantiate an object of either the two types of activities, we need the appropriate builder class to build the object.

```

8  /**
9   * Builder interface that sets prototype for the builder classes.
10  *
11  */
12  public interface Builder {
13
14      void setId(UUID id);
15
16      void setName(String name);
17
18      void setType(String type);
19
20      void setOwner(String owner);
21
22      void setStart(Date start);
23
24      void setLocation(String location);
25
26      void setPositions(List<Position> positions);
27
28      void setApplicants(List<String> applicants);
29
30      void setBoatType(String boatType);
31
32      Activity build();
33  }

```

Figure 7: The builder interface

```

1  package rowing.activity.domain;
2
3  import ...
4
5  @Data
6  public class TrainingBuilder implements Builder {
7      private UUID id;
8      private String owner;
9      private String name;
10     private List<Position> positions;
11     private String type;
12     private Date start;
13
14     private String location;
15     private List<String> applicants;
16     private String boatType;
17
18     public Training build() {
19         return new Training(id, owner, name, type, start, location, positions, applicants, boatType);
20     }
21 }

```

(a) The training builder class

```

1  package rowing.activity.domain;
2
3  import ...
4
5  @Data
6  public class CompetitionBuilder implements Builder {
7      private UUID id;
8      private String owner;
9      private String name;
10     private String type;
11     private Date start;
12     private String location;
13     private Gender gender;
14     private String organisation;
15     private List<Position> positions;
16     private List<String> applicants;
17     private String boatType;
18
19     public Competition build() {
20         return new Competition(id, owner, name, type, start, location,
21                                gender, organisation, positions, applicants, boatType);
22     }
23 }

```

(b) The competition builder class

Figure 8: Builder classes

To facilitate this we implemented 4 functions in the director class (fig. 9). Two of the functions are used for setting the parameters of the activity object using raw lists of attributes, while the other two functions are used to set the parameters using data from data transfer objects (DTOs). DTOs are used in our application to transfer necessary information about activities between microservices without giving the complete information about the activity. With these functions in the director we can then easily construct entities from their matching DTOs for easy storage and operations.

Our implementation can be seen in the class diagram (fig. 10).

```

13  /**
14   * Director that will set the values for building instances of activity objects.
15   *
16   */
17  public class Director {
18      /**
19       * Constructor for the competition out of raw attributes.
20       *
21       * @param builder that will build
22       * @param id of the activity
23       * @param name of the activity
24       * @param owner of the activity
25       * @param type training in our case
26       * @param start time the competition starts
27       * @param positions list of positions to be filled
28       * @param applicants list of applicants names
29       */
30      @ public void constructTraining(TrainingBuilder builder, UUID id, String name, String owner, String type,
31                                     Date start, String location,
32                                     List<Position> positions, List<String> applicants, String boatType) {
33          builder.setId(id);
34          builder.setName(name);
35          builder.setOwner(owner);
36          builder.setType(type);
37          builder.setStart(start);
38          builder.setLocation(location);
39          builder.setPositions(positions);
40          builder.setApplicants(applicants);
41          builder.setBoatType(boatType);
42      }

```

Figure 9: The director class and a setter for training parameters

2.3 Advantages

The builder design pattern is extremely useful in our scenario as it allows step-by-step construction of objects which can be changed into different types upon construction. This is exactly which is required for users making training or competition activities. This is enabled, while allowing the reuse of the same construction code and isolating the construction code from the logic of the product. One thing to keep in mind is that these advantages do come at the cost of somewhat increased complexity by marginally increasing the code, as the pattern requires multiple new classes.

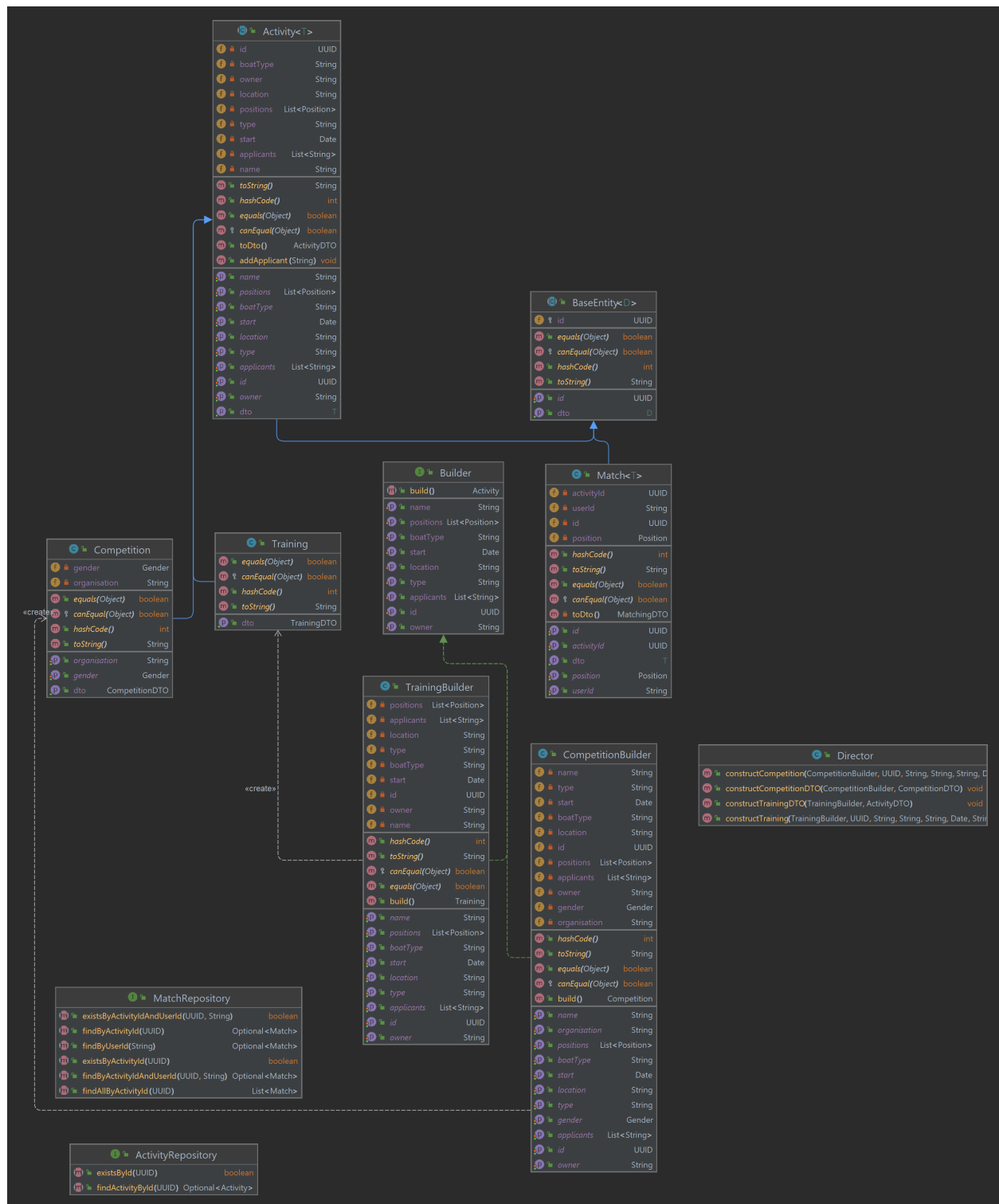


Figure 10: The class diagram implementing the builder design pattern

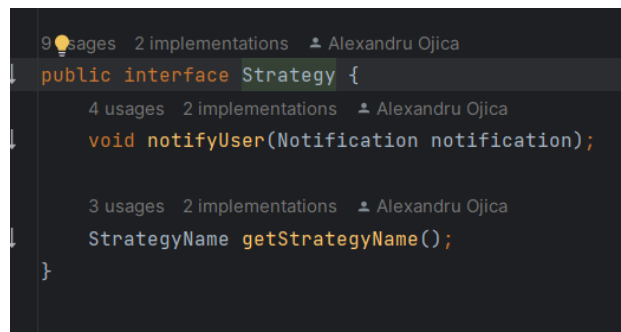
3 Strategy design pattern

The strategy pattern is a behavioral design pattern that enables an algorithm's behavior to be selected at runtime. The strategy pattern defines a family of algorithms, encapsulates each, and makes the algorithms interchangeable. The strategy pattern creates objects representing various strategies and executes a specific behavior depending on the strategy selected at runtime. This pattern involves implementing a behavior interface familiar to all strategies and implementing concrete strategies which are interchangeable in the context where the behavior interface is used. In the strategy pattern, the context is decoupled from the specific implementation of the algorithm. This allows the context to be flexible and easily adaptable to different strategies, as it does not need to know the details of the specific strategy. Instead, the context relies on the behavior interface defined by the strategy, and the concrete strategies can be changed or swapped out as needed without affecting the context.

3.1 Necessity of the strategy pattern

Our application needs to send notifications to users. We first thought of sending email messages but that becomes impossible when the user doesn't have an email address. The solution to this problem was to send a kafka message to the users without an email address. Since both options require a complex algorithm, we decided to create a strategy for each of them. Additionally, this approach adds modularity to our microservice, making it easy to add more strategies in the future.

Some of the actual implementation in code can be seen in figure 11, 12, 13 and 14.

A screenshot of an IDE showing a Java code snippet for a Strategy interface. The code is highlighted in a dark theme. At the top, there are statistics: '9 usages 2 implementations' and the author 'Alexandru Ojica'. The code defines a public interface Strategy with two methods: notifyUser(Notification notification) and getStrategyName(). The second method has its own statistics: '3 usages 2 implementations' and the same author.

```
9 usages 2 implementations Alexandru Ojica
public interface Strategy {
    4 usages 2 implementations Alexandru Ojica
    void notifyUser(Notification notification);

    3 usages 2 implementations Alexandru Ojica
    StrategyName getStrategyName();
}
```

Figure 11: Code snippet of Strategy interface

```

7 usages  ▲ Alexandru Ojica
@Data
@Component
public class KafkaStrategy implements Strategy {

    1 usage
    @Autowired
    KafkaTemplate<String, String> kafkaTemplate;

    1 usage
    @Value("notification")
    String topicName;

    3 usages  ▲ Alexandru Ojica
    @Override
    public StrategyName getStrategyName() { return StrategyName.KAFKA; }

    4 usages  ▲ Alexandru Ojica
    @Override
    public void notifyUser(Notification notification) {
        String message = notification.retrieveSubject() + "\n" + notification.retrieveBody();
        JSONObject json = new JSONObject();
        json.put(notification.getUsername(), message);
        kafkaTemplate.send(topicName, json.toString());
        System.out.println(message);
    }
}

```

Figure 12: Code snippet of KafkaStrategy

```

7 usages  ▲ Alexandru Ojica +1
@Data
@Component
public class EmailStrategy implements Strategy {

    1 usage
    @Autowired
    transient EmailService emailService;

    3 usages  ▲ Alexandru Ojica
    @Override
    public StrategyName getStrategyName() { return StrategyName.EMAIL; }

    4 usages  ▲ Alexandru Ojica
    @Override
    public void notifyUser(Notification notification) { emailService.sendEmail(notification); }
}

```

Figure 13: Code snippet of the EmailStrategy

```

▲ Alexandru Ojica
public enum StrategyName {

    6 usages
    EMAIL,

    6 usages
    KAFKA

}

```

Figure 14: Code snippet of StrategyName Enum