TED UNIVERSITY

CMPE 492

Senior Project

Gesture Guide: Virtual Assistant for the Hearing-Impaired

Testing Plan

Submission Date: 04.05.2024

Team Members:

- Berke Lahna
- Canberk Aydemir
- Ceyhun Toker
- Mehmet Fatih Ülker

Advisor: Venera Adanova

Jurors: Gökçe Nur Yılmaz and Eren Ulu

Web Site: https://berkelahna.github.io/Pages

TABLE OF CONTENTS

1. Introduction

1.1 Overview

The Gesture Guide application is the virtual assistant specialized for the hearing impaired which also provides sign language interpretation. Until now we have worked primarily on the software planning and modelling aspect of the project working with deliverables such as high and low-level design documents. But now the time has come for us to start testing both the system as a whole and each individual sub-component. This document is dedicated completely to planning the testing phase of the Gesture Guide application. Throughout the document for every software component, different testing methodologies will be proposed, and steps of tests will be analyzed deeply.

1.2 Testing Documentation Guidelines

Test Case ID:	The Id of the test case		
Initial Conditions:	What set of conditions should be met in order to conduct the test		
Description:	Short description of the test		
Status:	Status of the test		
Tester:	Who is responsible of the test		
Due Date:	Due Date	**Test Date:**	Proposed Testing Interval
Variation	**Action Taken**	**Expected Result**	
Variation of the unit test	What action should be performed	What is the expected output/behavior.	

1.3 Definitions, Acronyms, and Abbreviations

Abbreviations

- UML: Unified Modelling Language
- ISO: International Organization for Standardization
- ID: Identification Number
- OOP: Object Oriented Programming
- ANN: Artificial Neural Networks
- CNN: Convolutional Neural Networks
- CPU: Central Processing Unit

For detailed definitions please refer to the glossary section located at the end of this document.

2. Scope

The Gesture Guide application started as a machine learning project that would wrap multiple machine learning models together. But as the project has progressed, we have emphasized more on the user-oriented virtual assistant aspect of the Gesture Guide project as the project evolved into a full-scale software application that offers so much more than machine learning models. If we look at the big picture, gesture guide is a software application that encapsulates an user interface(front-end), back-end (persistence operations layer) and machine learning algorithms. In order to test the gesture guide application as a whole it is required to test all three of these software components which require different methodologies of testing and different test cases.

The user interface/front-end is the user's gateway to the application and is the layer that interacts mostly with the back-end services. Whenever users interact with components of the user interface, front-end services handle the required actions the application should execute, whether it is making an HTTP request or a simple animation. The user interface of the Gesture Guide will handle tasks such as helping users to register, login, access past assistant queries, and open virtual assistant/sign language interpreter while cooperating with back-end and the machine learning models. In order to make sure front-end services are working as expected, testing of these functionality is required.

On the other hand, back-end of the application is focused on the state of the application which constructs the business logic that is required through the application. For the case of Gesture Guide there are 2 persistent entities which are user and virtual assistant queries. They both have dedicated databases; the back-end communicates and manipulates these databases by the usage of object relational mapping(ORM) which maps entities of the database to plain objects. Using these mapped objects, the gesture guide application handles object specific tasks such as finding a user entity given a user id. Without the back-end services the gesture guide application cannot handle user interactions which makes it one of the backbones of the application. For gesture guide we have proposed a repository-service-controller scheme and service layer is the layer that establishes the business logic. Therefore, it is crucial to test the service layer functions to test the back-end.

Finally, the machine learning models are very important as they are one of the biggest features of gesture guide providing logic for the virtual assistant. There are vast number of testing metrics for machine learning applications and more specifically classification problems. We will calculate the validation and test set metrics, but we will also be trying to see how good these models work on the practice by trying to recreate the test data ourselves.

3. Testing Strategy

After analyzing the scope of testing, it will be suitable to talk about possible testing methodologies that will be used throughout the testing phase. There are different types of methodologies that would be suitable for each one of the software components to be tested. Since the methodology and applications of back-end components and front-end services are completely different to each other. This section will focus on preferred testing methodologies both universal to the project and specific to unique components of the application

Our approach will be generally manual rather than automated due to time constraints. Automated testing has its own advantages but requires more time to establish, so the testing approach we have chosen to apply to throughout the application will be manual. Meaning that a person out of our group will explicitly check the behavior of a component by simulating the required conditions.

Now let us address which kind of test will be used for each of the components. For the back-end we will be writing explicit unit tests that will test out service layer functions with various different conditions and boundaries. For front-end we will be also testing units with different conditions, but we will not be writing explicit code because of the time constraints and the fact that there are still features that we must add to the user interface. For machine learning components we will be analyzing the validation scores as well as many other classification and YOLO metrics such as MAP-50, class loss, box loss, f1 and ROC AUC. But we will also be testing the models using synthetic data which we will provide. If there is enough time after testing has finished for each component integration tests can also be conducted.

While we are testing the application at the same time we will continue to develop the application. Meaning that any defect we come across, will be tried to solve depending on the importance of the defect. Eventually we will be using this feedback loop of test and defect to develop our application further. Until the delivery of the project, we will be following this cycle. Each defect will be assigned both an importance and effort, so that the development team can work on the most crucial defect first. This way even if there are defects left unhandled for the demonstration, they will be relatively less crucial for the application.

4. Testing Environment

For clarification of how the test cases will be conducted, the testing environment should be discussed thoroughly as well. As we have set clear there are 3 main components that require testing and validation. These parts being front-end services (user interface), back-end services and the machine learning models. And each one of these components requires different testing environments.

The back-end services will be tested using Java's most famous testing library Junit. Junit is specialized for writing unit test in Java applications. Since the back-end of the system is implemented in Java, it is not a hard decision to use Junit for explicitly writing different unit test. For mocking different software components, we will also be using Mockito which is a great complementary service to what Junit already brings into the table. So, the software environment for unit testing the back-end services will be handled using both Junit and Mockito libraries.

The front-end services will be tested using the front-end server itself. Rather than explicitly coding unit tests we will be testing out the functionality by testing each unit and recording the outcomes. The latter sections will be detailing more on this process of unit testing without writing the code for it.

For machine learning models, we will be testing the models inside python applications. Since the models are originally trained and migrated from python. We will be testing these models straight from the python application and applying validation test with our synthetic data obtain necessary performance metrics.

5. Test Cases

5.1 Front-End/User Interface Test Cases

As we have mentioned earlier the front-end is the layer that lets users interact and manipulate the system. Only through front-end services users can completely utilize the application for their benefits. In this context the gesture guide application has 5 main front-end services which are register, log in, view past assistant queries, use sing language/gesture recognition model and get assistant response. All of these functionalities should be tested unit by unit. Since we still are refining the user interface, we will be unit testing without explicit code. We will be focusing more on the specific sequences taken for a test case and the behavior we have obtained from that sequence.

Test case 1: User Registration via the user Interface(front-end)

Test Case ID:	1		
Initial Conditions:	The application as a whole should be running (front-end, back-end and the database servers).		
Description:	Registration through the user interface is tested		
Status:	Not Started		
Tester:	Ceyhun		
Due Date:	June 1, 2024	**Test Date:**	05.05.2024-01.06.2024

Variation	**Action Taken**	**Expected Result**
1.1	User does not fill the registration form and still submits the form.	The form informs user that required fields cannot be empty.
1.2	User only fills some of the fields on the form.	In this case as well the form should indicate that the user should fill all of the fields.
1.3	User completely fills the form and submits the form.	Registration should be completed successfully, and the user information should be sent to the back-end server.

Registration operation starts with the front-end. A form is displayed for users to fill, and then the information on this form is sent to the back-end server so that a new entity can be created on the database.

Test case 2: User Log In via the User Interface(front-end)

Test Case ID:	2		
Initial Conditions:	For log in to be successful the user should have been registered to the system.		
Description:	Log in functionality through the user interface is tested.		
Status:	Not Started		
Tester:	Ceyhun		
Due Date:	June 1, 2024	**Test Date:**	05.05.2024-01.06.2024

Variation	Action Taken	Expected Result
2.1	User does not fill the log in form and still submits the form.	The form informs user that required fields cannot be empty.
2.2	User only fills password or email fields on the form.	The form should indicate that the user should fill all of the fields.
2.3	User completely fills the form with invalid credentials and submits the form.	The HTTP request is sent to the back-end. Back-end cannot find the entity. The user is informed stating the credentials are invalid
2.4	User completely fills the log in form with correct credentials.	The back-end server approves the credentials, the user is granted access to their profiles.

These set of unit test are the starting point of the logic for enabling users to be able to log in to the system to access their specialized version of the application. The front-end displays a form to the user which the user fills in. Then this information is validated by the back-end server, if the email password combination is correct the user is granted a cookie that enables them to access different routes that are special to each user such as the virtual assistant route. If the combination is invalid, then the application asks the user to either re-enter their credentials or register to the system.

Test case 3: Viewing Past Assistant Queries via the User Interface(front-end)

Test Case ID:	3		
Initial Conditions:	The user should be both registered and logged in to the system.		
Description:	The functionality of viewing past assistant queries is tested.		
Status:	Not Started		
Tester:	Ceyhun		
Due Date:	June 1, 2024	**Test Date:**	05.05.2024-01.06.2024

Variation	Action Taken	Expected Result
3.1	User presses a button to view past assistant queries and has at least made 1 assistant query in the past.	The past assistant queries are fetched using the back end and are displayed inside a table.
3.2	User presses a button to view past assistant queries but has never used the virtual assistant.	A message is displayed stating that there are no past assistant usages to display.

These unit test emphasize the functional requirement “Viewing past assistant usages”. The assistant queries are stored on the database with an affiliated user field. Whenever user presses a button to view their assistant queries of the past, the back-end returns a list of assistant queries.

Test case 4: Accessing Virtual Assistant/Sign Language Interpreter on the User Interface(front-end)

Test Case ID:	4		
Initial Conditions:	The tester should be logged in to the system.		
Description:	Accessing and using the virtual assistant/sign language interpreter is tested.		
Status:	Not Started		
Tester:	Berke		
Due Date:	June 1, 2024	**Test Date:**	05.05.2024-01.06.2024

Variation	**Action Taken**	**Expected Result**
4.1	User presses the virtual assistant/sign language interpreter button. The user does not permit webcam usage.	The user cannot access the virtual assistant/sign language interpreter. User is informed that the webcam usage should be enabled.
4.2	User presses the virtual assistant/sign language interpreter button. The user permits webcam usage.	The webcam starts recording, each of the frames are redirected to the machine learning models.

The front-end services also enable users to use the machine learning models on-the-go. The models are integrated into the front-end and are able to predict frames that are retrieved from the webcam of users. The test cases 4.1 and 4.2 test if users can access these models and utilize their webcams to start utilizing the models.

Test case 5: Getting Response from Virtual Assistant on the User Interface(front-end)

Test Case ID:	5		
Initial Conditions:	The user should be logged in and have started using virtual assistant through their webcam.		
Description:	Using and getting a response from the virtual assistant is tested.		
Status:	Not Started		
Tester:	Berke		
Due Date:	June 1, 2024	**Test Date:**	05.05.2024-01.06.2024

Variation	Action Taken	Expected Result
5.1	User tries all of the allowed gestures.	For each of the gestures the unique response to that gesture is created.
5.2	User tries a not allowed gesture.	Nothing happens.

The front-end services are also responsible for creating the functionality of the virtual assistant to create responses to given hand gestures. These unit test aim to test the response based on the gesture.

5.2 Back-End Test Cases

The back-end components are the bridge between the users of the gesture guide application and the persistent data that is stored in the database of the project. The back-end services allow the application to establish central data access, privacy and security and is one of the backbones of every software application. The business logic surrounding the users of gesture guide is completely built on top of the back-end components. The specified back-end components encapsulate the whole user sub-system and some parts of the virtual assistant system. We have chosen to specifically test the service layer functions because they are the most important layer in creating both ORM and business logic surrounding users. The service layer classes and functions have been discussed in the low-level design but it is recommended that before proceeding in this chapter see Appendix A and Appendix B in the appendix section.

For testing the back-end components, the most suitable testing methodology is unit testing. Unit testing can be defined as testing the smallest components of code such as methods. While we are applying unit tests, we write various test cases testing out corner cases and expected workflow for every method. Each method should have 3 or more dedicated unit tests that are simulated with different inputs, parameters and instance variables which aim to produce an unexpected output. The same idea of unit testing will be applied to Gesture Guide back-end components using JUnit.

Test case 6: Creating Users via UserService Class' createUser method

Test Case ID:	6		
Initial Conditions:	There are no specific initial conditions for this test case. As long as the back-end server is intact.		
Description:	Creation of a new user entities within the back-end server is tested.		
Status:	Not Started		
Tester:	Canberk		
Due Date:	June 1, 2024	**Test Date:**	05.05.2024-01.06.2024

Variation	Action Taken	Expected Result
6.1	Create a user with all instance variables set as null. Call the createUser method with user.	An exception should be thrown stating that the required fields cannot be null.
6.2	Create a user model with only name and surname set. Call the createUser method with user.	Failure in model creation should occur, stating that the required fields are not set.
6.3	Create a user model with a name surname and password set. Call the createUser method with user.	Failure in model creation should occur, stating that the required fields are not set.
6.4	Create a user model with name surname, password and email set. Call the createUser method with user.	Failure in model creation should occur, stating that the required fields are not set.
6.5	Create a user model with all of the required instance variables set. Call the createUser method with user.	The user model should be created successfully, and a HTTP 200 response should be returned

The User model implemented in the back-end has 4 required fields being name-surname, password, email and phone number. Even if one of these fields is not set, the user instantiation should fail. If all of these fields are set, then the user model should be created and stored in the database. This test case has been split into 5 different conditions and 5 different unit tests, all of these unit test try to create different users with different fields set.

Test case 7: Fetching all of the users via UserService Class' getAllUsers method

Test Case ID:	7		
Initial Conditions:	For variation 1 we assume that there are a couple of user entities registered to the system. For variation 2 we assume there are no users registered.		
Description:	Getting all of the user models in a list is tested.		
Status:	Not Started		
Tester:	Canberk		
Due Date:	June 1, 2024	**Test Date:**	05.05.2024-01.06.2024

Variation	Action Taken	Expected Result
7.1	Call the getAllUsers method. (Assuming there are registered users)	A list of all registered users should be returned back.
7.2	Call the getAllUsers method.	An empty list is returned.

These set of unit test, test the getAllUsers method which is used to return all the user models that are stored in the database.

Test case 8: Fetching a specific user with id via UserService Class' getUser method

Test Case ID:	8		
Initial Conditions:	For variation 1 we assume that the user id parameter is valid. For variation 2 we assume there the user id is nonexistent.		
Description:	Getting a specific user model with a specified id is tested		
Status:	Not Started		
Tester:	Canberk		
Due Date:	June 1, 2024	**Test Date:**	05.05.2024-01.06.2024

Variation	**Action Taken**	**Expected Result**
8.1	Call the getUser method with an existing id.	The user with the specified id should be returned.
8.2	Call the getUser method with a non-existing id.	An empty user model should be returned.

UserService method getUser takes parameter user id and then returns the user with the matching user id. Unit test 8.1 and 8.2 both try to simulate the function with a user id for a user that exists and a user id with a user that does not exist to see how the function responds in these situations.

Test case 9: Deletion of users via UserService Class' deleteUser method

Test Case ID:	9		
Initial Conditions:	We should use mocking tools for simulating the behavior of the ORM tools.		
Description:	Testing of the deletion of user entities.		
Status:	Not Started		
Tester:	Canberk		
Due Date:	June 1, 2024	**Test Date:**	05.05.2024-01.06.2024

Variation	Action Taken	Expected Result
9.1	Call the deleteUser method with an existing user id.	The user and assistant queries of this user should be deleted.
9.2	Call the deleteUser method with a not existing user id.	The deletion should fail indicating that no user is found with the specified ID.

Test case 10: Update of Virtual Assistant Queries via UserService Class' updateAssistantQueries method.

Test Case ID:	10		
Initial Conditions:	The specified user entities should have an initialized list of assistantQueries which is handled automatically during instantiation.		
Description:	Testing the functionality of updating virtual assistant queries of users.		
Status:	Not Started		
Tester:	Canberk		
Due Date:	June 1, 2024	**Test Date:**	05.05.2024-01.06.2024

Variation	**Action Taken**	**Expected Result**
10.1	Call the updateAssistantQueries method with actual assistant query entities.	The user and its assistant queries are updated. (Assistant Queries are a list which is an instance field of user entities)
10.2	Call the updateAssistantQueries method with an empty list of assistant query entities.	The user and the assistant queries should remain unchanged.

Test case 11: Fetching assistant history of a specific user via UserService Class' getAssistant Queries method

Test Case ID:	11		
Initial Conditions:	We should use mocking tools for simulating the behavior of the ORM tools.		
Description:	Testing of the functionality to fetch user's virtual assistant queries of the past.		
Status:	Not Started		
Tester:	Canberk		
Due Date:	June 1, 2024	**Test Date:**	05.05.2024-01.06.2024

Variation	Action Taken	Expected Result
11.1	Call the getAssistantHistory method with an existing user id.	A list of user assistant queries should be returned.
11.2	Call the getAssistantHistory method with a not existing user id.	An empty list should be returned.
11.3	Call the getAssistantHistory method with an existing user with no past assistant query.	An empty list should be returned.
11.4	Call the getAssistantHistory method with an existing user with past assistant query.	A list of past assistant history should be returned.

Test case 12: Creating Assistant Query Entities via AssistantQueryService Class' createAssistantQuery method

Test Case ID:	12		
Initial Conditions:	There are no specific initial conditions for this test case. The ORM tools such as repository classes should be mocked.		
Description:	Creation of a new assistant query within the back-end server is tested.		
Status:	Not Started		
Tester:	Canberk		
Due Date:	June 1, 2024	**Test Date:**	05.05.2024-31.05.2024

Variation	Action Taken	Expected Result
12.1	Create an assistant query entity with all instance variables set as null. Call the createAssistantQuery method with the assistant query.	The model is not saved on the database and an error message stating AssistantQuery cannot have empty action group and user is sent.
12.2	Create an assistant query entity with only the action group set. Call the createAssistantQuery method with the assistant query.	The model is saved on the database, but there won't be any users related to the assistant query entity. The user field of this assistant query entity should be explicitly set later.
12.3	Create an assistant query entity with all of the required fields set. Call the createAssistantQuery method with the assistant query.	The model is saved on the database and the user field of this assistant query entity is set successfully as well

The AssistantQuery model implemented in the back-end has 1 required field being action type. If this field is not set than instantiation should fail. If this field is set, then the assistant query model should be created and stored in the database. This test case has been split into 3 different conditions and 3 different unit tests, all of these unit test try to create different assistant queries with different fields conditions.

Test case 13: Fetching all of the assistant query entities via AssistantQueryService Class' getAllAssistantQuery method.

Test Case ID:	13		
Initial Conditions:	For variation 1 we assume that there are a couple of assistant queries registered to the system. For variation 2 we assume there are no assistant query registered.		
Description:	Getting all of the assistant query models in a list is tested.		
Status:	Not Started		
Tester:	Canberk		
Due Date:	June 1, 2024	**Test Date:**	05.05.2024-01.06.2024

Variation	Action Taken	Expected Result
13.1	Call the getAllAssistantQuery method. (Assuming there are registered assistant queries)	A list of all registered assistant queries should be returned back.
13.2	Call the getAllAssistantQuery method (No assistant query entity exists).	An empty list of assistant queries is returned back.

These set of unit test, test the getAllAssistantQuery method which is used to return all of the assistant query models that are stored in the database.

Test case 14: Fetching a specific assistant query entity with specific id via AssistantQueryService Class's getAssistantQuery method.

Test Case ID:	14		
Initial Conditions:	For variation 1 we assume that the assistant query id parameter is valid. For variation 2 we assume there the assistant query id is nonexistent.		
Description:	Getting a specific assistant query model with a specified id is tested		
Status:	Not Started		
Tester:	Canberk		
Due Date:	June 1, 2024	**Test Date:**	05.05.2024-01.06.2024

Variation	Action Taken	Expected Result
14.1	Call the getAssistantQuery method with an existing id.	The assistant query with the specified id should be returned.
14.2	Call the getAssistantQuery method with a non-existing id.	An empty assistant query object should be returned.

AssistantQueryService method getAssistantQuery takes parameter assistant query id and then returns the assistant with the matching assistant query id. Unit test 14.1 and 14.2 both try to simulate the function with an assistant query id for a assistant query that exists and a assistant query id with a assistant query that does not exist to see how the function responds in these situations.

Test case 15: Deletion of assistant query entities via AssistantQueryService Class' deleteAssistantQuery method

Test Case ID:	15		
Initial Conditions:	We should use mocking tools for simulating the behavior of the ORM tools.		
Description:	Testing the functionality of deletion of assistant query entities.		
Status:	Not Started		
Tester:	Canberk		
Due Date:	June 1, 2024	**Test Date:**	05.05.2024-01.06.2024

Variation	Action Taken	Expected Result
15.1	Call the deleteAssistantQuery method with an existing user id.	The assistant queries with the specified entity should be deleted.
15.2	Call the deleteAssistantQuery method with a not existing user id.	The deletion should fail indicating that assistant query is not found with the specified ID.

With these unit test we aim to test the deletion of assistant query entities with a specific id. We simulate bot the case that the id exists, and it does not exist.

Test case 16: Getting the user related to AssistantQuery via AssistantQueryService Class' getUser method.

Test Case ID:	16		
Initial Conditions:	The specified assistant query should be mapped to a user entity.		
Description:	Testing the functionality of getting user related to a specific virtual assistant query.		
Status:	Not Started		
Tester:	Canberk		
Due Date:	June 1, 2024	**Test Date:**	05.05.2024-01.06.2024

Variation	Action Taken	Expected Result
16.1	Call the getUser method with using the id of an assistant query.	The user entity in relation to the specified assistant query is returned.
16.2	Call the getUser method with an assistant query entity that is not coupled to a user entity.	An empty user entity is send back indicating the assistant query Is not coupled.

5.3 Machine Learning Model Tests

The sign language interpretation and gesture recognition are tasks that are handled by the machine learning algorithms. Which means that we also need to evaluate the performance of the machine learning models in order to provide best service to our users. The machine learning models we use are models that are specialized in object detection such as YOLO and ResNet. These models are use multiple layers of CNNs to handle both segmentation and classification. The primary metrics these models try to minimize are MAP-50(mean average precision), box loss (How accurate is the rectangle box) and class loss. We can easily obtain these and many more metrics (recall precision curve, ROC AUC, ...) after model training and validation phase and have the required statistic for the models. But in order to simulate the experience of the users we should be testing the models with our synthetic data that are obtained through webcam and test these data with the models to estimate how the model will work under normal conditions. Tests in this section will propose creation of synthetic data to test the performance of models rather than proposing unit tests.

Test Case 17: Sign Language Model Testing

For testing the sign language model, we can choose to create static data by capturing multiple photographs of each class and then test the model with this input. Another option is to use OpenCV to retrieve webcam feed and make predictions non-stop which would simulate the behavior the user would come across using the application. After enough observations we can come up with our confusion matrix and assess the classification on the more practical aspect.

Test Case 18: Sign Language Model Testing

For testing the hand gesture model, we can apply basically the same methodology we have proposed for the sign language model. Either to create static data by capturing multiple photographs of each class or use live feed for continuous prediction. Eventually we would have substantial amounts of data to observe how the model will work from the perspective of the users.

6. Testing Schedule

Test Case Id	Methodology	Importance	Tester	Testing Date
6	**Unit Test Via Junit**	**Moderate**	**Canberk**	**06/05-12/05**
7	**Unit Test Via Junit**	**Moderate**	**Canberk**	**06/05-12/05**
8	**Unit Test Via Junit**	**Moderate**	**Canberk**	**06/05-12/05**
9	**Unit Test Via Junit**	**Low**	**Canberk**	**06/05-12/05**
10	**Unit Test Via Junit**	**Moderate**	**Canberk**	**06/05-12/05**
11	**Unit Test Via Junit**	**Moderate**	**Canberk**	**06/05-12/05**
1	**Unit Test without Explicit Code**	**Crucial**	**Ceyhun**	**13/05-19/05**
2	**Unit Test without Explicit Code**	**Crucial**	**Ceyhun**	**13/05-19/05**
3	**Unit Test without Explicit Code**	**Crucial**	**Ceyhun**	**13/05-19/05**
4	**Unit Test without Explicit Code**	**Moderate**	**Berke**	**13/05-19/05**
5	**Unit Test without Explicit Code**	**Moderate**	**Berke**	**13/05-19/05**
17	**Validation Testing**	**Crucial**	**Fatih**	**19/05-31/05**
18	**Validation Testing**	**Crucial**	**Fatih**	**19/05-31/05**
12	**Unit Test Via Junit**	**Moderate**	**Canberk**	**27/05-31/05**
13	**Unit Test Via Junit**	**Moderate**	**Canberk**	**27/05-31/05**
14	**Unit Test Via Junit**	**Moderate**	**Canberk**	**27/05-31/05**
15	**Unit Test Via Junit**	**Low**	**Canberk**	**27/05-31/05**
16	**Unit Test Via Junit**	**Moderate**	**Canberk**	**27/05-31/05**

7. Risk Assessment

Now that we have set every aspect of testing phase, we can consider possible risks that we can come across. These possible risks and problems can occur in topics such as integration of each component being tested, lack of integration testing and possible problems that can occur through reporting and handling of the defects.

- **Integration of Each Components**: Since each of the components that are being tested is tested independently, after testing each component we may need to consider testing the behavior of the system as a whole. But since the front-end server is a component that encapsulates other 2 components, testing the front-end becomes more crucial than other components. The risks of failure in integration can be solved by focusing more thoroughly on the front-end tests.
- **Lack Of Integration Tests**: Even though we do not have explicit integration testing, the front-end service can be thought of as an integrated service that uses both the machine learning models and back-end services. Therefore, testing the front-end service can replace the integration tests that cannot be conducted due to time constraints.
- **Reporting and Handling Defects**: After applying the test cases, the tester should report the defects that may have occurred during testing. These defects should be expressed to the development team soon as possible with a projected effort so that the developers can handle this error based on both effort and the importance of the defect. Therefore, while reporting defects a very appropriate effort and importance should be declared by the testers.

8. Appendix

- **Authentication**: In a software system, validating if a user exists in the system.
- **Authorization**: In a software system, granting authority to users based on their roles.
- **Classification**: In machine learning, the process of determining which class a sample belongs to.
- **Credentials**: Personal information that users use to register/log in to a system. Credentials are a crucial part of the authentication process.
- **Model (Machine Learning)**: An algorithm that is used to predict on regression or classification tasks.
- **Unit**: The smallest possible software component.
- **Recall**: A model classification metric that focuses on number of false negative predictions (Type 2 error also knowns as Beta).
- **Precision**: A classification metric that focuses on false positive prediction (Type 1 error also known as Alpha).

UserService Class	
public User createUser(User newUser);	The method that creates and save a user entity to the database.
public List<User> getAllUsers();	The method that returns all of the user entities from the database.
public User getUser(int userId);	The method to find a user by the user id field of the user.
public boolean deleteUser(int userId)	The method to delete the user with the specified user id.
boolean updateAssistantQueries(int id, List<AssistantQuery> assistantQueryList)	Update the list of assistant queries for a user with a new list of assistant queries.
List<AssistantQuery> getAssistantQueries(int id);	Get the assistant queries of the user with the given user id.

Appendix A: The UserService Class Methods

AssistantQueryService Class	
public AssistantQueryDto createAssistantQuery(AssistantQuery assistantQuery);	Create new assistant query entity with the given assistant query.
public List<AssistantQueryDto> getAllQueries();	Return the list of all the assistant query entities.
public AssistantQueryDto getQuery(int id);	The method for returning a specific assistant query with the given id.
boolean deleteAssistantQuery(int id);	Delete a specific assistant query with the specified assistant query id.
public User getUser (int id);	Get the user affiliated with the assistant query.

Appendix B: The AssistantQueryService Class Methods

9. References

- Sommerville, I. (2016). Software Engineering, 10th edition. Pearson Education Limited. ISBN 10: 1-292-09613-6
- Martin, R.C. (2002). UML for Java Programmers. 1'st edition. Prentice Hall
- Bruegge, B et al. (2004). Object-Oriented Software Engineering, Using UML, Patterns, and Java, 2nd Edition. Pearson
- Crispin, L., & Gregory, J. (2009). Agile Testing: A Practical Guide for Testers and Agile Teams. Addison-Wesley Professional.