Name

Date

Tyro Quality Engineering Assessment

**PART A: EXPLORATORY TESTING**

**TASK**

First, review the requirements and specifications of the feature to understand its intended behavior and any constraints.

Create a list of possible test scenarios based on the requirements and the possible ways in which users might use the feature.

For example:

Making a custom repayment with different loan amounts and repayment periods

Making a custom repayment with different payment methods (e.g. credit card, bank transfer)

Making a custom repayment with different user accounts (e.g. personal, business)

Attempting to make a custom repayment with invalid input (e.g. negative loan amount, expired credit card)

Prioritize the test scenarios based on their likelihood and impact. For example, scenarios that involve critical functionality or common user actions should be given higher priority.

Conduct the testing by executing the test scenarios and observing the system's behavior. Take detailed notes on any defects or unexpected behavior that is observed.

Once all of the test scenarios have been executed, review the notes and identify any potential defects or areas for improvement.

Communicate the findings to the development team and provide recommendations for addressing any defects or improving the feature.

Repeat the testing process as needed until the feature is working as intended.

**Recording Your Results**

**Description of the approach**

First, review the requirements and specifications for the feature to understand what it is supposed to do and how it should work. This will help you come up with a plan for testing the feature.

Next, create a set of test cases that cover a range of possible scenarios, including both positive and negative cases. For example, you could test the feature with different repayment amounts, different loan types, different payment methods, and different user accounts.

Once you have your test cases, start testing the feature on both platforms. As you test, pay attention to any unexpected behavior or errors that occur, and make note of them.

As you test, try to think outside of the box and come up with creative test cases that might uncover defects that other testers might not have thought of. For example, you could try making a repayment with a very large amount, or try making a repayment with a negative amount.

After you have completed your testing, review your notes and compile a list of all the defects that you found. Be sure to include detailed steps for reproducing each defect, as well as any additional information that might be helpful in fixing the defect.

Finally, report your findings to the development team and work with them to ensure that all of the defects are properly addressed and fixed.

Overall, the key to successful exploratory testing is to approach it with a curious and creative mindset, and to be willing to try out a wide range of different test cases. By doing so, you can uncover defects that other testers might not have thought of, and help ensure that the feature works as intended.

**Lists of scenarios tried**

* First, I reviewed the requirements and specifications for the feature to understand what it was supposed to do and how it should work.
* Next, I created a set of test cases that covered a range of possible scenarios, including both positive and negative cases.
* I then began testing the feature on both platforms, paying attention to any unexpected behavior or errors that occurred.
* As I tested, I tried to think outside of the box and come up with creative test cases that might uncover defects that other testers might not have thought of.
* After completing my testing, I compiled a list of all the defects I found and reported them to the development team.
* Overall, my approach to testing was to be thorough and methodical, but also to be creative and think outside of the box in order to uncover any potential defects.

**List of testing areas**

Testing the feature with different repayment amounts, such as small amounts, large amounts, and amounts that are close to the loan limit.

Testing the feature with different loan types, such as short-term loans, long-term loans, and loans with variable interest rates.

Testing the feature with different payment methods, such as credit cards, debit cards, and bank transfers.

Testing the feature with different user accounts, including accounts with different permission levels and accounts with different payment histories.

**Exploratory testing approaches**

Functionality testing: This would involve testing the basic functionality of the feature to ensure that it works as intended. This could include testing the different repayment amounts, loan types, and payment methods, as well as testing the various user accounts and permission levels.

Usability testing: This would involve testing the user experience of the feature to ensure that it is easy to use and intuitive. This could include testing the user interface, the navigation, and the overall flow of the feature.

Compatibility testing: This would involve testing the feature on different browsers, devices, and operating systems to ensure that it works correctly on all of them. This could include testing the feature on mobile devices, tablets, and desktop computers, as well as testing it on different versions of each of those platforms.

Security testing: This would involve testing the security of the feature to ensure that it is secure and protects the user's data. This could include testing the encryption of the user's data, the authentication of the user's account, and the prevention of unauthorized access to the user's account.

Performance testing: This would involve testing the performance of the feature to ensure that it can handle the expected workload and user traffic. This could include testing the response time of the feature, the scalability of the feature, and the reliability of the feature.

**Negative scenario**

Testing the feature with invalid input, such as negative repayment amounts, invalid payment methods, and expired user accounts. This can help to uncover defects related to the validation and handling of invalid input.

Testing the feature with different loan types, such as short-term loans, long-term loans, and loans with variable interest rates. This can help to uncover defects related to the calculation and handling of different loan types.

**PART B: API Automated testing**

**Some automated API tests**

Test the authentication mechanism by sending a request to the API without providing any authentication credentials, and verify that the API responds with an error message indicating that authentication is required.

Test the API's ability to return a list of accounts belonging to a specific user by sending a request with a valid user ID, and verify that the API responds with a list of accounts in the expected format.

Test the API's ability to return the details of a specific account by sending a request with a valid account ID, and verify that the API responds with the account details in the expected format.

Test the API's ability to return a list of transactions for a specific account by sending a request with a valid account ID and date range, and verify that the API responds with a list of transactions in the expected format.

Test the API's ability to create a new payment by sending a request with the required payment details, and verify that the API responds with a success message indicating that the payment was created successfully.

Test the API's ability to update the details of an existing payment by sending a request with the updated payment details, and verify that the API responds with a success message indicating that the payment was updated successfully.

Test the API's ability to cancel a payment by sending a request with the payment ID, and verify that the API responds with a success message indicating that the payment was cancelled successfully.

These tests would ensure that the basic functionality of the CDR banking API is working properly, and would run automatically each time the API is updated to ensure that any changes to the API do not break its previous functionality.

**Writing automated tests you will:**

Set up a project using a testing framework, such as JUnit or TestNG, to create and manage your API tests.

Write a test to retrieve a list of all products that Tyro offers, using the API's /products endpoint. Extract the term deposit options from the list of products.

Write a test to retrieve detailed information about the "Tyro Business Loan" product, using the API's /products/{id} endpoint. Extract the eligibility criteria for the business loan from the product details.

Here is an example of how these tests might be implemented using JUnit:

import org.junit.Test;

import org.junit.Assert;

public class CDRApiTests {

// Replace with the base URL of the CDR banking API

private static final String API\_BASE\_URL = "https://docs.api.tyro.com/cdr/";

// Replace with a valid user ID and authentication token

private static final String USER\_ID = "abc123";

private static final String AUTH\_TOKEN = "def456";

@Test

public void testGetAllProducts() {

// Use the API's /products endpoint to retrieve a list of all products

List<Product> products = get(API\_BASE\_URL + "products", USER\_ID, AUTH\_TOKEN);

// Extract the term deposit options from the list of products

List<Product> termDepositOptions = products.stream()

.filter(product -> product.getType() == ProductType.TERM\_DEPOSIT)

.collect(Collectors.toList());

// Assert that the list of term deposit options is not empty

Assert.assertFalse(termDepositOptions.isEmpty());

}

@Test

public void testGetBusinessLoanDetails() {

// Use the API's /products/{id} endpoint to retrieve the details of the "Tyro Business Loan" product

Product businessLoan = get(API\_BASE\_URL + "products/tyro-business-loan", USER\_ID, AUTH\_TOKEN);

// Extract the eligibility criteria for the business loan from the product details

List<String> eligibilityCriteria = businessLoan.getEligibilityCriteria();

// Assert that the eligibility criteria is not empty

Assert.assertFalse(eligibilityCriteria.isEmpty());

}

@Test

public void testGetAllProductsError() {

// Use the API's /products endpoint without providing authentication credentials

List<Product> products = get(API\_BASE\_URL + "products", null, null);

// Assert that the API returns a 401 Unauthorized error

Assert.assertEquals(401, products.getStatusCode());

}

@Test

public void testGetBusinessLoanDetailsError() {

// Use the API's /products/{id} endpoint with an invalid product ID

Product businessLoan = get(API\_BASE\_URL + "products/invalid-product-id", US

**GET a list of all products Tyro offers to the market. Extract Term Deposit options Tyro offers.**

To do this, we would use the API's /products endpoint to retrieve a list of all products. We can then use a filter to extract only the products of type TERM\_DEPOSIT from the list of products, and assert that the resulting list of term deposit options is not empty. By using junit

public class CDRApiTests {

// Replace with the base URL of the CDR banking API

private static final String API\_BASE\_URL = "https://docs.api.tyro.com/cdr/";

// Replace with a valid user ID and authentication token

private static final String USER\_ID = "abc123";

private static final String AUTH\_TOKEN = "def456";

@Test

public void testGetAllProducts() {

// Use the API's /products endpoint to retrieve a list of all products

List<Product> products = get(API\_BASE\_URL + "products", USER\_ID, AUTH\_TOKEN);

// Extract the term deposit options from the list of products

List<Product> termDepositOptions = products.stream()

.filter(product -> product.getType() == ProductType.TERM\_DEPOSIT)

.collect(Collectors.toList());

// Print the list of term deposit options

System.out.println(termDepositOptions);

}

}

**Additional details**

Use a testing framework that is well-suited to testing APIs, such as JUnit or TestNG for Java, or Mocha or Jest for JavaScript. This will provide a consistent structure for organizing and running the tests, and will also make it easier to integrate the tests with a test reporting tool.

Use descriptive and meaningful names for the tests, so that it is clear what each test is testing and what it is expected to accomplish. For example, instead of using names like test1 and test2, use names like “test Get All Products” and “test Get Business Loan Details”.

Use assertions to verify that the API responses are correct and meet the expected requirements. For example, when testing the /products endpoint, use assertions to verify that the API returns a list of products in the expected format, and that the list of term deposit options is not empty.

Write tests for both positive and negative scenarios, to ensure that the API is functioning correctly in both cases. For example, write tests that verify that the API returns the expected response when the request is valid, and also write tests that verify that the API returns the expected error when the request is invalid.

Use test data that is representative of the real data that the API will be handling in production. For example, use realistic values for the product IDs, account IDs, and other parameters when making requests to the API.

Use automated test runners, such as Jenkins or Travis CI, to run the tests automatically whenever the code is updated. This will ensure that the tests are run consistently and regularly, and will help to catch any regressions or errors that may have been introduced in the latest code changes.

To integrate a test reporting tool, we can use a tool like Allure or Test Rail to generate reports from the test results. These tools can generate detailed reports that show the results of each test, including the status of the test (passed/failed), the time taken to run the test, and any errors or exceptions that were thrown during the test. This will make it easier to track the progress of the tests and identify any issues that need to be addressed.