*Luis Martins & George Carp*

National College of Ireland

Object Oriented Software Engineering Project

Part B

****

Table of Contents

[1. Using an appropriate object-oriented language (e.g., Java), fully develop the classes required to implement one of the use cases described in part 1 of the assessment. The use case must contain at least one alternate flow. 4](#_Toc151022881)

[2. Fully test the classes developed for Q1 above, naming and justifying the test methodology followed. Describe the tests carried out, detailing how they will ensure that the classes are free from errors and detail the results of the tests. A minimum of 3 tests must be created and run. 5](#_Toc151022882)

[3. Provide detailed artefacts of the agile methodology followed, such as user stories, backlogs and burndown charts. 6](#_Toc151022883)

# Using an appropriate object-oriented language (e.g., Java), fully develop the classes required to implement one of the use cases described in part 1 of the assessment. The use case must contain at least one alternate flow.

import java.util.regex.\*;

import java.util.Arrays;

public class AuthenticationService {

// Public methods

public boolean login(String username, String password) {

if (verifyPassword(username, password)) {

return true;

} else {

displayWrongDetails();

return false;

}

}

public void createAccount(String username, String password) {

// There should be a database with user accounts, but since there’s not, let’s pretend there is and only one account for now exists with username ‘admin’

String[] usernames = {“admin”};

// If value of username is present in usernames, throw exception.

if(Arrays.asList(usernames).contains(username)) {

throw new IllegalArgumentException(”username already registered.”);

}

// Check if the password contains at least one capital letter, one number, and one symbol

String regex = "^(?=.\*[A-Z])(?=.\*[0-9])(?=.\*[!@#$%^&\*()-\_+=]).+$";

Pattern pattern = Pattern.compile(regex);

Matcher matcher = pattern.matcher(password);

boolean isPasswordComplex = matcher.find();

if (isPasswordComplex) {

// Create account

System.out.println("Account created successfully for username: " + username);

} else {

// Display an error message

System.out.println("Error: Password does not meet requirements. It should contain at least one capital letter, one number, and one symbol.");

}

}

// Private methods

private boolean verifyPassword(String username, String password) {

// Private methods

return true;

}

private void displayWrongDetails() {

System.out.println("Error: Incorrect username or password!");

}

public static void main(String[] args) {

AuthenticationService authService = new AuthenticationService();

// Example usage:

authService.createAccount("user123", "Passw0rd!"); // Should succeed

authService.login("user123", "Passw0rd!"); // Should succeed

authService.createAccount("admin", "password123"); // Should fail due duplicate username.

authService.login("user123", "Passw0rd!"); // Should fail due to incorrect details.

}

}

# 2. Fully test the classes developed for Q1 above, naming and justifying the test methodology followed. Describe the tests carried out, detailing how they will ensure that the classes are free from errors and detail the results of the tests. A minimum of 3 tests must be created and run.

As a team we decided to use Test-Driven Development (TDD) as out test methodology during the development of the AuthenticationService class for the “AccessApp” use case.

TDD is a development approach where tests are written before code implementation.

It involves writing a failing test, then writing the minimum code required to pass it, and finally, refactoring the code.

This approach was chosen because, by starting with tests, we catch potential issues early. And also, because one team-member was writing the implementation code, while another was writing the tests. Making us benefit from a clear separation of tasks — the test writer looks at the external behaviour and requirements, while the implementer focuses on meeting those requirements. Working on different tasks simultaneously, considerably accelerated the development process.

The specific testing framework we used to conduct the tests was JUnit, simply because it is the most popular unit testing framework for Java.

* Test Cases:

1. Login
   1. - Login with valid credentials
   * Check if the login method correctly returns **true** with valid credentials.
   * Creates a new instance of “AuthenticationService”, adds a user account, and then attempts to log in using it.
   * Call JUnit’s assertTrue method to check if the result is **true**.

@Test  
public void testLoginWithValidCredentials() {  
 AuthenticationService authService = new AuthenticationService();  
 authService.createAccount("testUser", "password123");  
  
 boolean result = authService.login("testUser", "password123");  
  
 *assertTrue*(result);  
}

**Result**: The test passes, indicating that the login functionality for valid credentials is working as expected.

* 1. - Login with invalid credentials
* Ensure that the login method returns **false** when invalid credentials are provided.
* It creates a new instance of “AuthenticationService”, adds a user account, and then attempts to log in with incorrect credentials.
* Call JUnit’s assertFalse method to check if the result is **false**.

@Test  
public void testLoginWithInvalidCredentials() {  
 AuthenticationService authService = new AuthenticationService();  
 authService.createAccount("testUser", "password123");  
  
 boolean result = authService.login("testUser", "wrongPassword");  
  
 *assertFalse*(result);  
}

**Result:** The test passes, indicating that the login functionality for invalid credentials is working as expected.

1. Create account
   1. – Create an account successfully

* Ensure that the “createAccount” method successfully creates a new user account.
* Creates a new instance of “AuthenticationService” and calls createAccount to add a new user.
* Tries to login with the newly created account, and store the result in a variable called result.
* Call JUnit’s assertTrue method to check if the result is true, indicating that the account was created successfully.

@Test  
public void testCreateAccount() {  
 AuthenticationService authService = new AuthenticationService();  
  
 authService.createAccount("newUser", "newPassword");  
  
 boolean result = authService.login("newUser", "newPassword");  
   
 *assertTrue*(result);  
}

**Result**: The test passes, indicating that the account creation functionality is working as expected.

* 1. – Create an account with an existing username.
* This test ensures that the createAccount method fails to create an account if the username already exists.
* It creates a new AuthenticationService instance, adds a user account, and then attempts to create another account with the same username.
* “assertThrows” method from Junit is used to confirm that calling createAccount with an existing username will result in an IllegalArgumentException being thrown. (As specified in the createAccount method).

@Test  
public void testCreateAccountWithExistingUsername() {  
 AuthenticationService authService = new AuthenticationService();  
 authService.createAccount("existingUser", "password123");  
   
 *assertThrows*(IllegalArgumentException.class, () -> authService.createAccount("existingUser", "newPassword"));  
}

**Result**: The test passes, indicating that the class correctly prevents creating an account with an existing username.

The other methods, “displayWrongDetails” and “verifyPassword()” were not tested, since they are private and are used internally by other methods.

# 3. Provide detailed artefacts of the agile methodology followed, such as user stories, backlogs and burndown charts.