How Finite Models Can Be Useful for Testing

Finite models, particularly Finite State Machines (FSMs), are highly useful in software testing because they provide a structured way to model and analyze the behavior of a system with a finite number of states and well-defined transitions. By using FSMs, testers can systematically verify that the system correctly responds to various inputs and transitions between states as expected.

1. Ensuring Complete State Coverage

FSMs help ensure that all possible states of a system are considered during testing. By defining states explicitly, testers can check whether each state and its corresponding transitions are correctly implemented.

2. Detecting Unexpected Behaviors

Many software failures occur due to incorrect state transitions or missing conditions. By formally defining state transitions, FSM-based testing can identify situations where the system might behave incorrectly or enter an undefined state.

3. Automating Test Case Generation

Since FSMs provide a clear model of how the system transitions between states, they enable automated test case generation. Tools can be used to derive test cases that cover all possible transitions, reducing the likelihood of missing important test scenarios.

4. Model-Based Testing (MBT)

FSMs are widely used in model-based testing, where the system is tested against a predefined model. This approach helps in verifying that implementations match their specifications and detecting discrepancies early in the development cycle.

5. Testing Real-World Applications

Many real-world systems naturally fit FSMs, such as login/logout mechanisms, traffic light controllers, vending machines, and network protocols. Using FSMs for testing these systems ensures that all valid and invalid state transitions are considered, leading to more robust software.

A Feature or Component that lends itself well to being described by FSM

I'll analyze the user registration feature from UserServiceTest.java in detail, as it's an excellent example of a component that can be modeled using a finite state machine.

User Registration Feature Analysis (20%)

1. Component Selection Justification:
   (1) The user registration feature in UserServiceTest.java is ideal for finite state machine modeling because:
      ① It has clearly defined input parameters (userAccount, userPassword, checkPassword, planetCode)
      ② It follows a strict validation sequence
      ③ It has well-defined state transitions
      ④ Each state has binary outcomes (pass/fail)
      ⑤ The process is deterministic and sequential

2. Component Structure:

long userRegister(String userAccount, String userPassword, String checkPassword, String planetCode)

3. Key Characteristics:

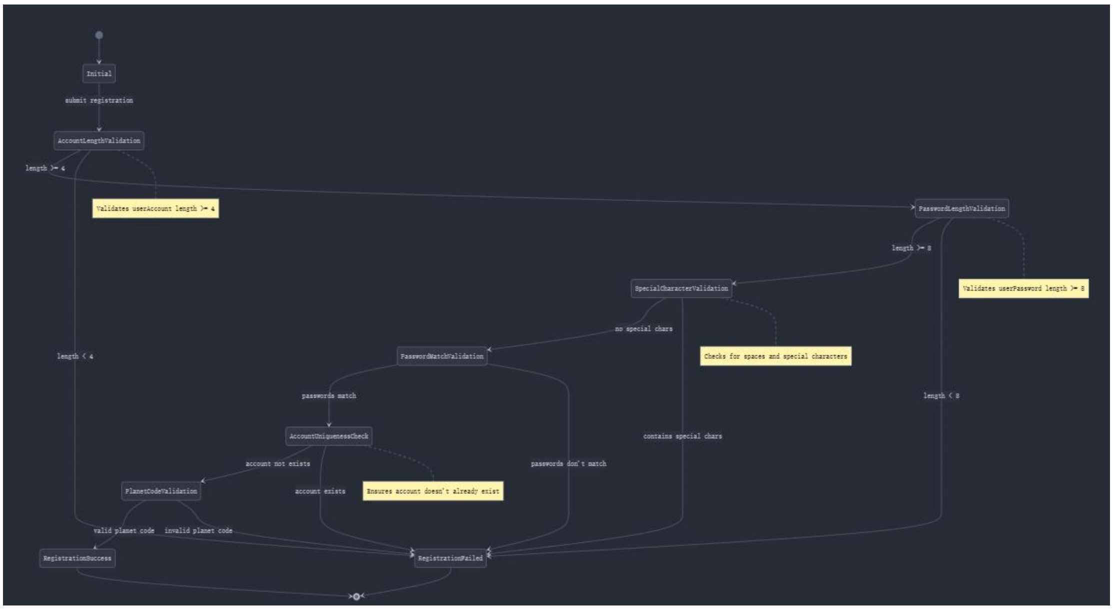
a) Input Validation States:
   (1) Account length validation
   (2) Password length validation
   (3) Special character validation
   (4) Password matching validation
   (5) Account uniqueness validation

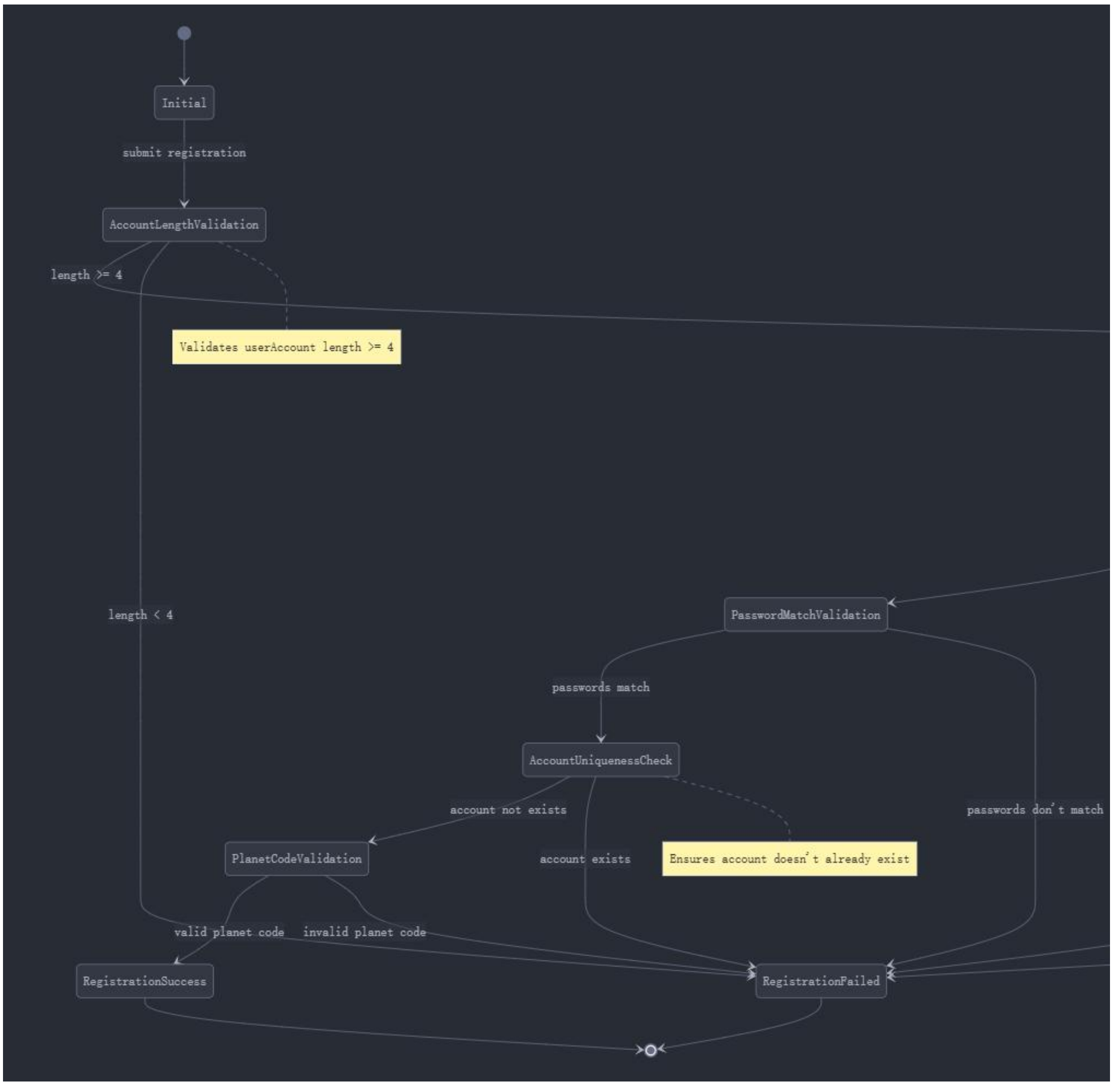
b) Transition Triggers:
   (1) Input parameter values
   (2) Database query results
   (3) Validation results

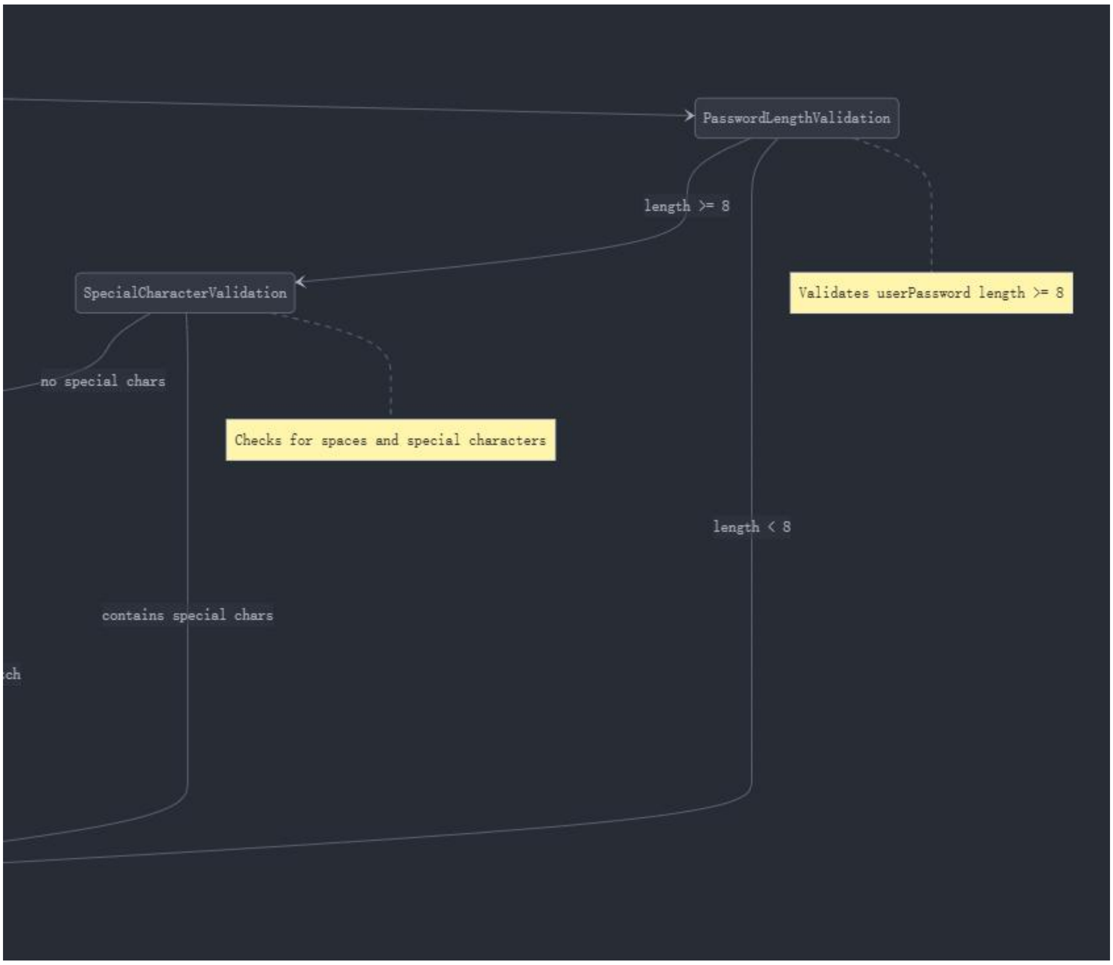
c) Output States:
   (1) Success (-1)
   (2) Various failure states (represented by different return values)

4. State Dependencies:
   (1) Each validation state depends on the success of previous validations
   (2) States are sequential and hierarchical
   (3) Failure at any state leads to immediate termination
   (4) Success requires passing all validation states

State Machine Diagram






PasswordLengthValidation
length >= 8
Validates userPassword length >= 8
SpecialCharacterValidation
no special chars
Checks for spaces and special characters
length < 8
contains special chars
ch

How it works

Validation States

1. **Account Length Validation**
    1. Input: userAccount
    2. Validation: length >= 4
    3. Transition conditions:
        1. Success: Move to Password Length Validation
        2. Failure: Move to Registration Failed
2. **Password Length Validation**

    1. Input: userPassword
    2. Validation: length >= 8
    3. Transition conditions:
        1. Success: Move to Special Character Validation
        2. Failure: Move to Registration Failed

3. **Special Character Validation**

    1. Input: userAccount
    2. Validation: No spaces or special characters
    3. Transition conditions:
        1. Success: Move to Password Match Validation
        2. Failure: Move to Registration Failed

4. **Password Match Validation**

    1. Input: userPassword, checkPassword
    2. Validation: Passwords must match exactly
    3. Transition conditions:
        1. Success: Move to Account Uniqueness Check
        2. Failure: Move to Registration Failed

5. **Account Uniqueness Check**

    1. Input: userAccount
    2. Validation: Account must not exist in database
    3. Transition conditions:
        1. Success: Move to Planet Code Validation
        2. Failure: Move to Registration Failed

6. **Planet Code Validation**

    1. Input: planetCode
    2. Validation: Valid planet code format
    3. Transition conditions:
        1. Success: Move to Registration Success
        2. Failure: Move to Registration Failed

Terminal States

1. **Registration Success**

    1. Final success state
    2. Returns user ID
    3. Stores user data in database

2. **Registration Failed**

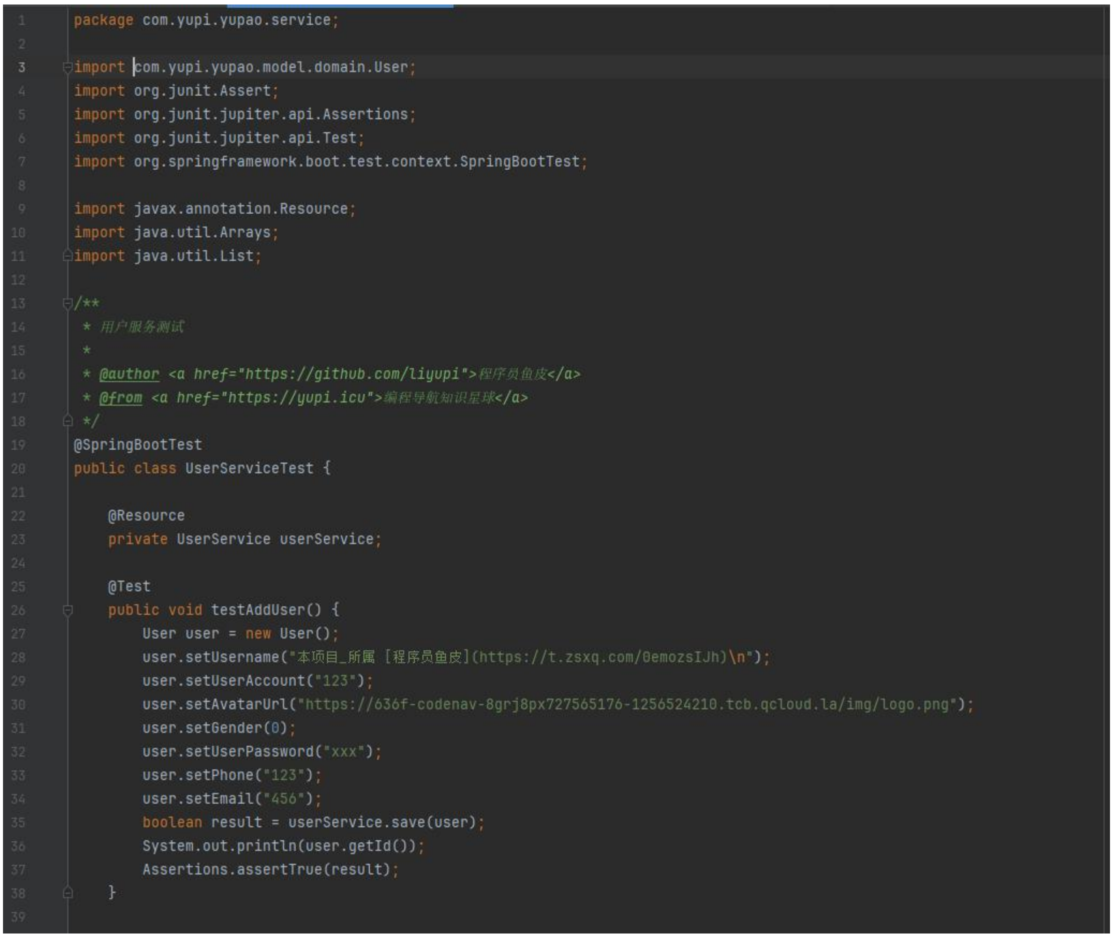
    1. Final failure state
    2. Returns -1
    3. Indicates validation failure

3. State Transitions

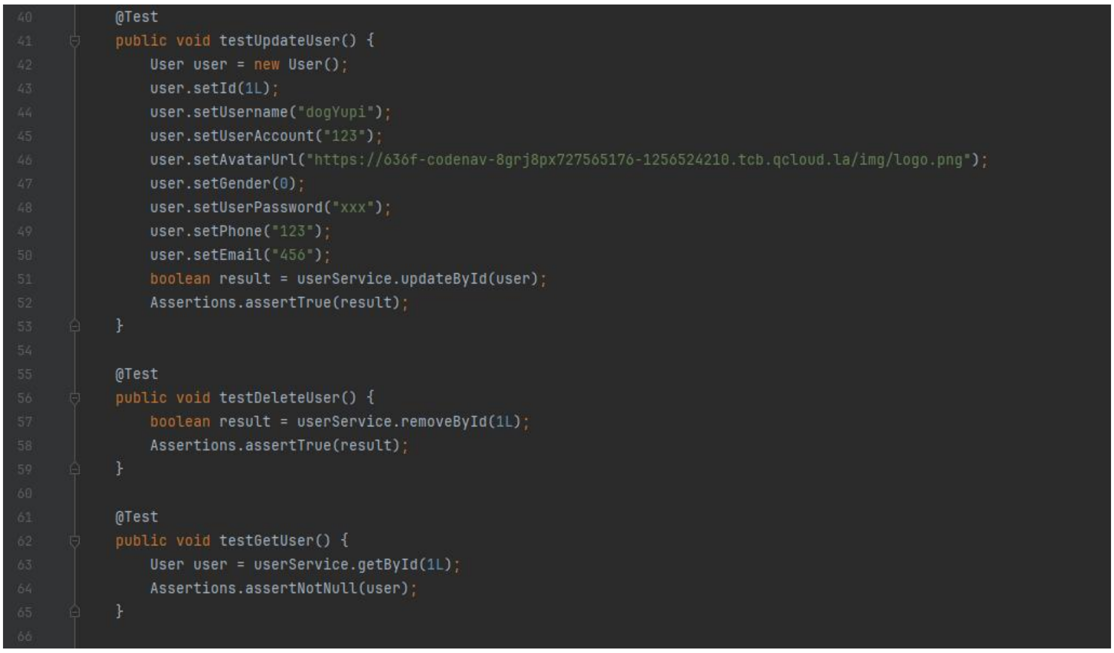
Success Path

Initial → AccountLengthValidation → PasswordLengthValidation → SpecialCharacterValidation → PasswordMatchValidation → AccountUniquenessCheck → PlanetCodeValidation → RegistrationSuccess

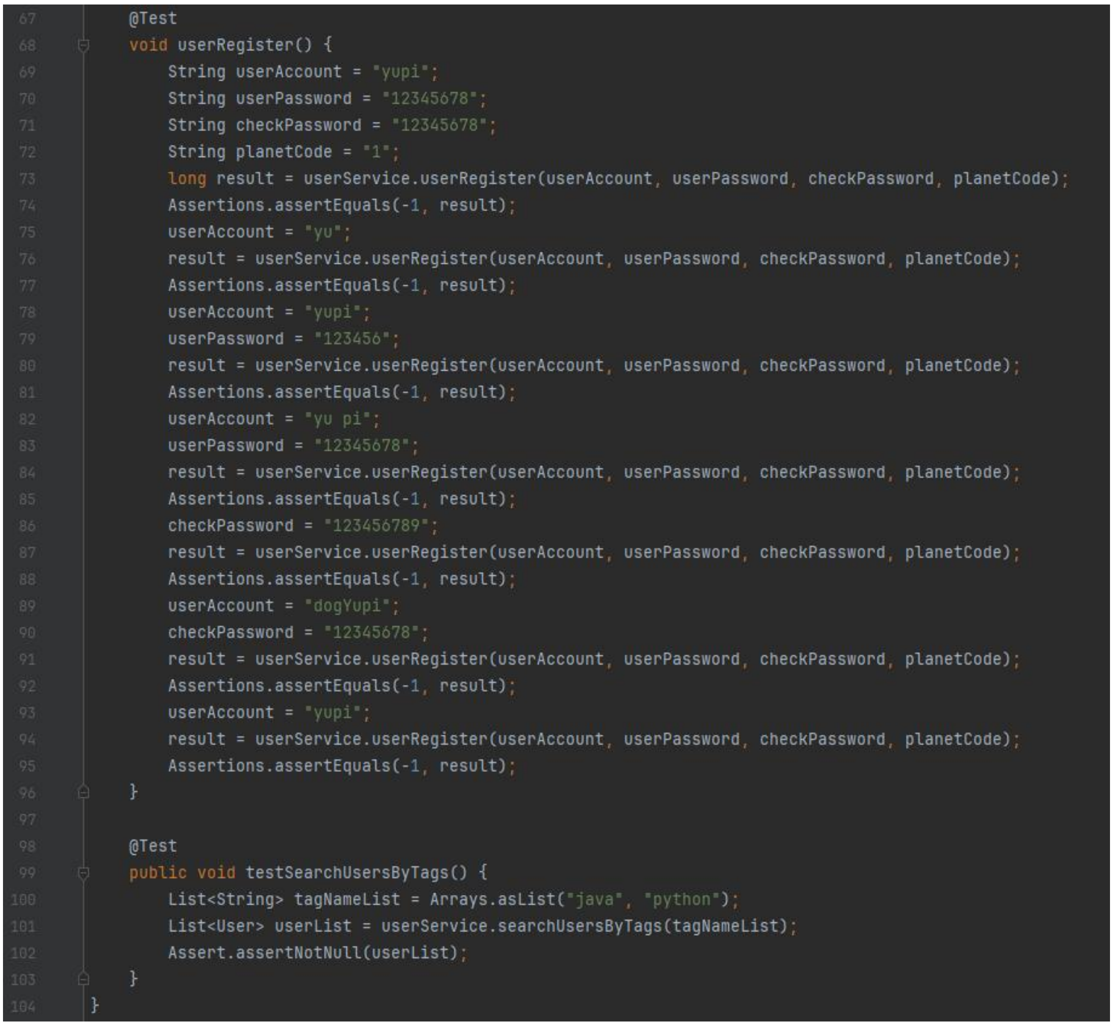
Testcases:



```java
package com.yupi.yupao.service;

import com.yupi.yupao.model.domain.User;
import org.junit.Assert;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

import javax.annotation.Resource;
import java.util.Arrays;
import java.util.List;

/**
 * 用户服务测试
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
@SpringBootTest
public class UserServiceTest {

    @Resource
    private UserService userService;

    @Test
    public void testAddUser() {
        User user = new User();
        user.setUsername("本项目_所属 [程序员鱼皮](https://t.zsxq.com/0emozsIJh)\n");
        user.setUserAccount("123");
        user.setAvatarUrl("https://636f-codenav-8grj8px727565176-1256524210.tcb.qcloud.la/img/logo.png");
        user.setGender(0);
        user.setUserPassword("xxx");
        user.setPhone("123");
        user.setEmail("456");
        boolean result = userService.save(user);
        System.out.println(user.getId());
        Assertions.assertTrue(result);
    }

```



```java
    @Test
    public void testUpdateUser() {
        User user = new User();
        user.setId(1L);
        user.setUsername("dogYupi");
        user.setUserAccount("123");
        user.setAvatarUrl("https://636f-codenav-8grj8px727565176-1256524210.tcb.qcloud.la/img/logo.png");
        user.setGender(0);
        user.setUserPassword("xxx");
        user.setPhone("123");
        user.setEmail("456");
        boolean result = userService.updateById(user);
        Assertions.assertTrue(result);
    }

    @Test
    public void testDeleteUser() {
        boolean result = userService.removeById(1L);
        Assertions.assertTrue(result);
    }

    @Test
    public void testGetUser() {
        User user = userService.getById(1L);
        Assertions.assertNotNull(user);
    }

```



```java
    @Test
    void userRegister() {
        String userAccount = "yupi";
        String userPassword = "12345678";
        String checkPassword = "12345678";
        String planetCode = "1";
        long result = userService.userRegister(userAccount, userPassword, checkPassword, planetCode);
        Assertions.assertEquals(-1, result);
        userAccount = "yu";
        result = userService.userRegister(userAccount, userPassword, checkPassword, planetCode);
        Assertions.assertEquals(-1, result);
        userAccount = "yupi";
        userPassword = "123456";
        result = userService.userRegister(userAccount, userPassword, checkPassword, planetCode);
        Assertions.assertEquals(-1, result);
        userAccount = "yu pi";
        userPassword = "12345678";
        result = userService.userRegister(userAccount, userPassword, checkPassword, planetCode);
        Assertions.assertEquals(-1, result);
        checkPassword = "123456789";
        result = userService.userRegister(userAccount, userPassword, checkPassword, planetCode);
        Assertions.assertEquals(-1, result);
        userAccount = "dogYupi";
        checkPassword = "12345678";
        result = userService.userRegister(userAccount, userPassword, checkPassword, planetCode);
        Assertions.assertEquals(-1, result);
        userAccount = "yupi";
        result = userService.userRegister(userAccount, userPassword, checkPassword, planetCode);
        Assertions.assertEquals(-1, result);
    }

    @Test
    public void testSearchUsersByTags() {
        List<String> tagNameList = Arrays.asList("java", "python");
        List<User> userList = userService.searchUsersByTags(tagNameList);
        Assert.assertNotNull(userList);
    }
}
```

Document:

Overview

This document outlines the test cases for the UserServiceTest class, which is responsible for testing the functionality of the UserService in a Spring Boot application. The UserService handles user-related operations such as adding, updating, deleting, and retrieving users, as well as user registration and searching users by tags.

Test Environment

Framework: JUnit 5

Spring Boot Version: As per project configuration

Dependencies: Spring Boot Test, JUnit Jupiter

Test Cases

1. testAddUser

Objective: Verify that a new user can be successfully added to the database.

Steps:

Create a new User object.

Set the user properties: username, userAccount, avatarUrl, gender, userPassword, phone, and email.

Call the save method of UserService to add the user.

Print the user ID to the console.

Assert that the result of the save method is true.

Expected Result: The user is successfully added to the database, and the save method returns true.

2. testUpdateUser

Objective: Verify that an existing user's information can be successfully updated.

Steps:

Create a new User object.

Set the user properties, including the id of the user to be updated.

Call the updateById method of UserService to update the user.

Assert that the result of the updateById method is true.

Expected Result: The user's information is successfully updated in the database, and the updateById method returns true.

3. testDeleteUser
Objective: Verify that a user can be successfully deleted from the database.

Steps:

Call the removeById method of UserService with the ID of the user to be deleted.

Assert that the result of the removeById method is true.

Expected Result: The user is successfully deleted from the database, and the removeById method returns true.

4. testGetUser
Objective: Verify that a user can be successfully retrieved from the database by their ID.

Steps:

Call the getById method of UserService with the ID of the user to be retrieved.

Assert that the returned User object is not null.

Expected Result: The user is successfully retrieved from the database, and the returned User object is not null.

5. userRegister
Objective: Verify the user registration process with various scenarios.

Steps:

Test with a valid userAccount, userPassword, checkPassword, and planetCode.

Assert that the result is -1 (indicating a failure, as per the test logic).

Test with a short userAccount.

Assert that the result is -1.

Test with a short userPassword.

Assert that the result is -1.

Test with a userAccount containing spaces.

Assert that the result is -1.

Test with a checkPassword that does not match userPassword.

Assert that the result is -1.

Test with a different userAccount.

Assert that the result is -1.

Test with the original userAccount again.

Assert that the result is -1.

Expected Result: The registration process correctly handles invalid inputs and returns -1 for each scenario.

6. ***testSearchUsersByTags***
Objective: Verify that users can be searched based on a list of tags.

Steps:

Create a list of tags (java, python).

Call the searchUsersByTags method of UserService with the tag list.

Assert that the returned list of users is not null.

Expected Result: The method returns a list of users who match the specified tags, and the list is not null.

Conclusion
The UserServiceTest class comprehensively tests the core functionalities of the UserService. By executing these test cases, we ensure that the user management operations work as expected and handle various edge cases appropriately. The use of assertions and the Spring Boot testing framework facilitates the identification and resolution of issues in the service logic.