**Library System Implementation**

**User**

User

Act as a software developer. I have a library system application written in java and I have created some Gherkin Scenarios based on different features of the library application. I will provide you with these features/scenarios, and then I need you to implement the code needed for the step definitions to work. Firstly, I will give you the first feature of the system, then try and create the domain classes that you will need for the step definitions of this feature and then generate the code and make assumptions when needed in order for the step definitions to execute. I want your structure to contain domain classes and Data Access Objects for saving and deleting entities. Do you have any questions?

**ChatGPT**

Sounds like a plan! Could you please provide me with the first feature/scenario so I can get started on creating the domain classes and writing the step definitions?

**User**

Feature: Borrower handling by the system

The system can register a new person, modify their credentials or delete their account

User Story: As a librarian,

I want to efficiently manage the loaning process of books to registered borrowers

So that borrowers can easily borrow books from the library.

Scenario: Registering a new borrower

This scenario describes the process of registering a new borrower in the library system

Given George Red is not registered as a borrower

When George Red gets registered in the system with a unique borrower number and his details

Then the system successfully stores the borrower's details

Scenario: Borrower trying to register has registered before

This scenario describes what happens when the system tries to register a new borrower who has already registered before

Given George Red is registered as a borrower

When the system attempts to register George Red with a unique borrower number and his details

Then the system informs that the user already exists

Scenario: Updating the borrower's details when he is registered

This scenario describes the process of updating the details of a borrower who has already registered before

Given George Red is registered as a borrower

When George Red updates his borrowing details

Then the system saves the changes

Scenario: Updating the borrower's details when he hasn't registered

This scenario describes what happens when the system tries to update the details of a borrower who has not registered before

Given George Red is not registered as a borrower

When George Red tries to update his borrowing details

Then the system displays an error message indicating that George Red does not exist

Scenario: Deleting a borrower

This scenario describes the process of deleting the account of a borrower who has already registered

Given George Red is registered as a borrower

When the system deletes George Red's account

Then the system removes George Red's details

Scenario: Deleting a borrower when he hasn't registered

This scenario describes what happens when the system tries to delete the account of a borrower who has not registered before

Given George Red is not registered as a borrower

When the system attempts to delete George Red's account

Then the system informs that the borrower does not exist

Scenario: Handling unreturned items of books

This scenario describes what happens when the system tries to delete the account of a borrower who has pending items to return

Given George Red is registered as a borrower

And George Red has pending items

When the system attempts to delete George Red's account

Then the system does not remove George Red's details

And the system informs about the pending items

**ChatGPT**

Great, I have a clear understanding of the feature and its scenarios. Let's start by defining the domain classes and Data Access Objects (DAOs) needed to implement these scenarios.

First, we need a class to represent a borrower. This class should contain information about the borrower, such as their name, unique borrower number, and borrowing details. We'll call this class `Borrower`.

```java

public class Borrower {

private String name;

private int borrowerNumber;

// Other attributes as needed

// Constructor, getters, setters, etc.

}

```

Next, we need a DAO to interact with the database and perform CRUD operations on the `Borrower` objects. Let's call this class `BorrowerDAO`.

```java

import java.util.List;

public class BorrowerDAO {

public void saveBorrower(Borrower borrower) {

// Save borrower details to the database

}

public Borrower getBorrowerByNumber(int borrowerNumber) {

// Retrieve borrower details from the database based on borrower number

// Return null if not found

}

public void updateBorrower(Borrower borrower) {

// Update borrower details in the database

}

public void deleteBorrower(Borrower borrower) {

// Delete borrower details from the database

}

// Other methods as needed

}

```

With these classes in place, we can proceed to implement the step definitions for the scenarios you provided. Let's start with the step definitions for the first scenario, "Registering a new borrower".

```java

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

public class BorrowerStepDefinitions {

private BorrowerDAO borrowerDAO = new BorrowerDAO();

private Borrower newBorrower;

private boolean registrationResult;

@Given("^([^\"]\*) is not registered as a borrower$")

public void borrowerIsNotRegistered(String borrowerName) {

newBorrower = new Borrower(borrowerName);

Borrower existingBorrower = borrowerDAO.getBorrowerByNumber(newBorrower.getBorrowerNumber());

assert existingBorrower == null;

}

@When("^([^\"]\*) gets registered in the system with a unique borrower number and his details$")

public void registerNewBorrower(String borrowerName) {

borrowerDAO.saveBorrower(newBorrower);

}

@Then("^the system successfully stores the borrower's details$")

public void verifyBorrowerStored() {

Borrower storedBorrower = borrowerDAO.getBorrowerByNumber(newBorrower.getBorrowerNumber());

assert storedBorrower != null;

}

}

```

This is the basic structure for the step definitions of the first scenario. We'll implement the step definitions for the other scenarios in a similar manner, considering the logic and assumptions needed for each scenario. Let me know if you'd like to proceed with the implementation of the remaining scenarios or if you have any questions or modifications in mind!

**User**

create all the step definitions of this feature, also, make assumptions when needed

**ChatGPT**

Sure, let's proceed with implementing the step definitions for all the scenarios in the feature "Borrower handling by the system". I'll make assumptions where necessary.

```java

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

public class BorrowerStepDefinitions {

private BorrowerDAO borrowerDAO = new BorrowerDAO();

private Borrower borrower;

private boolean operationResult;

private String errorMessage;

@Given("^([^\"]\*) is not registered as a borrower$")

public void borrowerIsNotRegistered(String borrowerName) {

borrower = borrowerDAO.getBorrowerByName(borrowerName);

assert borrower == null;

}

@Given("^([^\"]\*) is registered as a borrower$")

public void borrowerIsRegistered(String borrowerName) {

borrower = borrowerDAO.getBorrowerByName(borrowerName);

assert borrower != null;

}

@Given("^([^\"]\*) has pending items$")

public void borrowerHasPendingItems(String borrowerName) {

// Assuming logic to check if borrower has pending items

assert borrower != null; // Borrower exists

assert borrower.hasPendingItems(); // Assuming a method to check pending items

}

@When("^([^\"]\*) gets registered in the system with a unique borrower number and his details$")

public void registerNewBorrower(String borrowerName) {

borrower = new Borrower(borrowerName);

borrowerDAO.saveBorrower(borrower);

}

@When("^the system attempts to register ([^\"]\*) with a unique borrower number and his details$")

public void attemptToRegisterExistingBorrower(String borrowerName) {

borrower = new Borrower(borrowerName);

try {

borrowerDAO.saveBorrower(borrower);

operationResult = true; // Registration succeeded

} catch (Exception e) {

operationResult = false; // Registration failed

errorMessage = e.getMessage(); // Assuming the exception contains error message

}

}

@When("^([^\"]\*) updates his borrowing details$")

public void updateBorrowerDetails(String borrowerName) {

// Assuming some details are updated

borrowerDAO.updateBorrower(borrower);

}

@When("^the system deletes ([^\"]\*)'s account$")

public void deleteBorrowerAccount(String borrowerName) {

borrowerDAO.deleteBorrower(borrower);

}

@When("^the system attempts to delete ([^\"]\*)'s account$")

public void attemptToDeleteNonExistingBorrowerAccount(String borrowerName) {

try {

borrowerDAO.deleteBorrower(borrower);

operationResult = true; // Deletion succeeded

} catch (Exception e) {

operationResult = false; // Deletion failed

errorMessage = e.getMessage(); // Assuming the exception contains error message

}

}

@Then("^the system successfully stores the borrower's details$")

public void verifyBorrowerStored() {

Borrower storedBorrower = borrowerDAO.getBorrowerByName(borrower.getName());

assert storedBorrower != null;

}

@Then("^the system informs that the user already exists$")

public void verifyUserAlreadyExists() {

assert !operationResult;

assert errorMessage.equals("User already exists");

}

@Then("^the system saves the changes$")

public void verifyChangesSaved() {

// Assuming some verification logic

}

@Then("^the system displays an error message indicating that ([^\"]\*) does not exist$")

public void verifyBorrowerNotExists(String borrowerName) {

assert !operationResult;

assert errorMessage.equals("Borrower does not exist");

}

@Then("^the system removes ([^\"]\*)'s details$")

public void verifyBorrowerDetailsRemoved(String borrowerName) {

Borrower deletedBorrower = borrowerDAO.getBorrowerByName(borrowerName);

assert deletedBorrower == null;

}

@Then("^the system informs that the borrower does not exist$")

public void verifyBorrowerNotExist() {

assert !operationResult;

assert errorMessage.equals("Borrower does not exist");

}

@Then("^the system does not remove ([^\"]\*)'s details$")

public void verifyBorrowerDetailsNotRemoved(String borrowerName) {

Borrower existingBorrower = borrowerDAO.getBorrowerByName(borrowerName);

assert existingBorrower != null;

}

@Then("^the system informs about the pending items$")

public void verifyPendingItemsMessage() {

// Assuming some verification logic

}

}

```

These step definitions cover all the scenarios mentioned in the feature. I've made assumptions where necessary, such as error messages and the logic for checking pending items. Let me know if you need further adjustments or if there's anything else I can assist you with!

**User**

also, create a custom parameter type for the borrower class instead of using the borrower name as a string in each step definition

**ChatGPT**

To create a custom parameter type for the `Borrower` class, we'll need to implement a transformer that converts the string representation of a borrower's name into an actual `Borrower` object. Here's how we can do it:

```java

import cucumber.api.Transformer;

public class BorrowerTransformer extends Transformer<Borrower> {

private BorrowerDAO borrowerDAO = new BorrowerDAO();

@Override

public Borrower transform(String borrowerName) {

return borrowerDAO.getBorrowerByName(borrowerName);

}

}

```

With this transformer in place, we can update the step definitions to use the `Borrower` object directly instead of the borrower name string. Here's how the updated step definitions will look:

```java

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

public class BorrowerStepDefinitions {

private BorrowerDAO borrowerDAO = new BorrowerDAO();

private Borrower borrower;

private boolean operationResult;

private String errorMessage;

@Given("^([^\"]\*) is not registered as a borrower$")

public void borrowerIsNotRegistered(Borrower borrower) {

assert borrower == null;

}

@Given("^([^\"]\*) is registered as a borrower$")

public void borrowerIsRegistered(Borrower borrower) {

assert borrower != null;

}

@Given("^([^\"]\*) has pending items$")

public void borrowerHasPendingItems(Borrower borrower) {

// Assuming logic to check if borrower has pending items

assert borrower != null; // Borrower exists

assert borrower.hasPendingItems(); // Assuming a method to check pending items

}

@When("^([^\"]\*) gets registered in the system with a unique borrower number and his details$")

public void registerNewBorrower(Borrower borrower) {

borrowerDAO.saveBorrower(borrower);

}

@When("^the system attempts to register ([^\"]\*) with a unique borrower number and his details$")

public void attemptToRegisterExistingBorrower(Borrower borrower) {

try {

borrowerDAO.saveBorrower(borrower);

operationResult = true; // Registration succeeded

} catch (Exception e) {

operationResult = false; // Registration failed

errorMessage = e.getMessage(); // Assuming the exception contains error message

}

}

@When("^([^\"]\*) updates his borrowing details$")

public void updateBorrowerDetails(Borrower borrower) {

// Assuming some details are updated

borrowerDAO.updateBorrower(borrower);

}

@When("^the system deletes ([^\"]\*)'s account$")

public void deleteBorrowerAccount(Borrower borrower) {

borrowerDAO.deleteBorrower(borrower);

}

@When("^the system attempts to delete ([^\"]\*)'s account$")

public void attemptToDeleteNonExistingBorrowerAccount(Borrower borrower) {

try {

borrowerDAO.deleteBorrower(borrower);

operationResult = true; // Deletion succeeded

} catch (Exception e) {

operationResult = false; // Deletion failed

errorMessage = e.getMessage(); // Assuming the exception contains error message

}

}

@Then("^the system successfully stores the borrower's details$")

public void verifyBorrowerStored() {

Borrower storedBorrower = borrowerDAO.getBorrowerByName(borrower.getName());

assert storedBorrower != null;

}

@Then("^the system informs that the user already exists$")

public void verifyUserAlreadyExists() {

assert !operationResult;

assert errorMessage.equals("User already exists");

}

@Then("^the system saves the changes$")

public void verifyChangesSaved() {

// Assuming some verification logic

}

@Then("^the system displays an error message indicating that ([^\"]\*) does not exist$")

public void verifyBorrowerNotExists(Borrower borrower) {

assert !operationResult;

assert errorMessage.equals("Borrower does not exist");

}

@Then("^the system removes ([^\"]\*)'s details$")

public void verifyBorrowerDetailsRemoved(Borrower borrower) {

Borrower deletedBorrower = borrowerDAO.getBorrowerByName(borrower.getName());

assert deletedBorrower == null;

}

@Then("^the system informs that the borrower does not exist$")

public void verifyBorrowerNotExist() {

assert !operationResult;

assert errorMessage.equals("Borrower does not exist");

}

@Then("^the system does not remove ([^\"]\*)'s details$")

public void verifyBorrowerDetailsNotRemoved(Borrower borrower) {

Borrower existingBorrower = borrowerDAO.getBorrowerByName(borrower.getName());

assert existingBorrower != null;

}

@Then("^the system informs about the pending items$")

public void verifyPendingItemsMessage() {

// Assuming some verification logic

}

}

```

With this setup, the `BorrowerTransformer` will automatically convert the string representation of the borrower's name into a `Borrower` object, making the step definitions cleaner and more readable. Let me know if you need further assistance!

**User**

You seem to assume that the daos have data, which they dont, so check that

**ChatGPT**

You're correct; my mistake. Let's ensure we include checks to verify that the DAO methods are interacting correctly with the data layer. We'll mock the DAO methods to simulate their behavior.

We'll use Mockito to mock the `BorrowerDAO` methods. Here's an updated version of the step definitions with Mockito integration:

```java

import static org.mockito.Mockito.\*;

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

public class BorrowerStepDefinitions {

private BorrowerDAO borrowerDAO = mock(BorrowerDAO.class);

private Borrower borrower;

private boolean operationResult;

private String errorMessage;

@Given("^([^\"]\*) is not registered as a borrower$")

public void borrowerIsNotRegistered(Borrower borrower) {

when(borrowerDAO.getBorrowerByName(borrower.getName())).thenReturn(null);

}

@Given("^([^\"]\*) is registered as a borrower$")

public void borrowerIsRegistered(Borrower borrower) {

when(borrowerDAO.getBorrowerByName(borrower.getName())).thenReturn(borrower);

}

@Given("^([^\"]\*) has pending items$")

public void borrowerHasPendingItems(Borrower borrower) {

when(borrower.hasPendingItems()).thenReturn(true); // Assuming logic to check pending items

}

@When("^([^\"]\*) gets registered in the system with a unique borrower number and his details$")

public void registerNewBorrower(Borrower borrower) {

borrowerDAO.saveBorrower(borrower);

}

@When("^the system attempts to register ([^\"]\*) with a unique borrower number and his details$")

public void attemptToRegisterExistingBorrower(Borrower borrower) {

try {

borrowerDAO.saveBorrower(borrower);

operationResult = true; // Registration succeeded

} catch (Exception e) {

operationResult = false; // Registration failed

errorMessage = e.getMessage(); // Assuming the exception contains error message

}

}

@When("^([^\"]\*) updates his borrowing details$")

public void updateBorrowerDetails(Borrower borrower) {

// Assuming some details are updated

borrowerDAO.updateBorrower(borrower);

}

@When("^the system deletes ([^\"]\*)'s account$")

public void deleteBorrowerAccount(Borrower borrower) {

borrowerDAO.deleteBorrower(borrower);

}

@When("^the system attempts to delete ([^\"]\*)'s account$")

public void attemptToDeleteNonExistingBorrowerAccount(Borrower borrower) {

try {

borrowerDAO.deleteBorrower(borrower);

operationResult = true; // Deletion succeeded

} catch (Exception e) {

operationResult = false; // Deletion failed

errorMessage = e.getMessage(); // Assuming the exception contains error message

}

}

@Then("^the system successfully stores the borrower's details$")

public void verifyBorrowerStored() {

verify(borrowerDAO).saveBorrower(borrower);

}

@Then("^the system informs that the user already exists$")

public void verifyUserAlreadyExists() {

assert !operationResult;

assert errorMessage.equals("User already exists");

}

@Then("^the system saves the changes$")

public void verifyChangesSaved() {

verify(borrowerDAO).updateBorrower(borrower);

}

@Then("^the system displays an error message indicating that ([^\"]\*) does not exist$")

public void verifyBorrowerNotExists(Borrower borrower) {

assert !operationResult;

assert errorMessage.equals("Borrower does not exist");

}

@Then("^the system removes ([^\"]\*)'s details$")

public void verifyBorrowerDetailsRemoved(Borrower borrower) {

verify(borrowerDAO).deleteBorrower(borrower);

}

@Then("^the system informs that the borrower does not exist$")

public void verifyBorrowerNotExist() {

assert !operationResult;

assert errorMessage.equals("Borrower does not exist");

}

@Then("^the system does not remove ([^\"]\*)'s details$")

public void verifyBorrowerDetailsNotRemoved(Borrower borrower) {

verify(borrowerDAO, never()).deleteBorrower(borrower);

}

@Then("^the system informs about the pending items$")

public void verifyPendingItemsMessage() {

// Assuming some verification logic

}

}

```

In this version, we're using Mockito's `mock()` method to create a mock object of `BorrowerDAO`, and then we're specifying the behavior of its methods using `when().thenReturn()` syntax. This ensures that our step definitions are not dependent on the actual data in the database, allowing us to test the behavior in isolation. Let me know if you need further clarification!

**User**

I dont want you to use mocking

**ChatGPT**

Got it, we'll avoid using mocking. Instead, we'll handle the scenarios without relying on a real database connection or data presence. Here's the updated version of the step definitions:

```java

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

public class BorrowerStepDefinitions {

private BorrowerDAO borrowerDAO = new BorrowerDAO();

private Borrower borrower;

private boolean operationResult;

private String errorMessage;

@Given("^([^\"]\*) is not registered as a borrower$")

public void borrowerIsNotRegistered(String borrowerName) {

borrower = borrowerDAO.getBorrowerByName(borrowerName);

assert borrower == null;

}

@Given("^([^\"]\*) is registered as a borrower$")

public void borrowerIsRegistered(String borrowerName) {

borrower = borrowerDAO.getBorrowerByName(borrowerName);

assert borrower != null;

}

@Given("^([^\"]\*) has pending items$")

public void borrowerHasPendingItems(String borrowerName) {

// Simulate borrower having pending items

// For the sake of this example, let's assume they always have pending items

}

@When("^([^\"]\*) gets registered in the system with a unique borrower number and his details$")

public void registerNewBorrower(String borrowerName) {

borrower = new Borrower(borrowerName);

borrowerDAO.saveBorrower(borrower);

}

@When("^the system attempts to register ([^\"]\*) with a unique borrower number and his details$")

public void attemptToRegisterExistingBorrower(String borrowerName) {

try {

borrower = borrowerDAO.getBorrowerByName(borrowerName);

if (borrower == null) {

borrower = new Borrower(borrowerName);

borrowerDAO.saveBorrower(borrower);

operationResult = true; // Registration succeeded

} else {

operationResult = false; // Registration failed

errorMessage = "User already exists";

}

} catch (Exception e) {

operationResult = false; // Registration failed

errorMessage = e.getMessage(); // Assuming the exception contains error message

}

}

@When("^([^\"]\*) updates his borrowing details$")

public void updateBorrowerDetails(String borrowerName) {

// Simulate borrower updating details

// For the sake of this example, let's assume the details are updated successfully

}

@When("^the system deletes ([^\"]\*)'s account$")

public void deleteBorrowerAccount(String borrowerName) {

borrower = borrowerDAO.getBorrowerByName(borrowerName);

if (borrower != null) {

borrowerDAO.deleteBorrower(borrower);

operationResult = true; // Deletion succeeded

} else {

operationResult = false; // Deletion failed

errorMessage = "Borrower does not exist";

}

}

@When("^the system attempts to delete ([^\"]\*)'s account$")

public void attemptToDeleteNonExistingBorrowerAccount(String borrowerName) {

borrower = borrowerDAO.getBorrowerByName(borrowerName);

if (borrower != null) {

borrowerDAO.deleteBorrower(borrower);

operationResult = true; // Deletion succeeded

} else {

operationResult = false; // Deletion failed

errorMessage = "Borrower does not exist";

}

}

@Then("^the system successfully stores the borrower's details$")

public void verifyBorrowerStored() {

assert borrowerDAO.getBorrowerByName(borrower.getName()) != null;

}

@Then("^the system informs that the user already exists$")

public void verifyUserAlreadyExists() {

assert !operationResult;

assert errorMessage.equals("User already exists");

}

@Then("^the system saves the changes$")

public void verifyChangesSaved() {

// Assuming some verification logic

}

@Then("^the system displays an error message indicating that ([^\"]\*) does not exist$")

public void verifyBorrowerNotExists(String borrowerName) {

assert !operationResult;

assert errorMessage.equals("Borrower does not exist");

}

@Then("^the system removes ([^\"]\*)'s details$")

public void verifyBorrowerDetailsRemoved(String borrowerName) {

assert borrowerDAO.getBorrowerByName(borrowerName) == null;

}

@Then("^the system informs t