*In this chat, as in chat 4, 5, 9, 10, I tried giving the same information as always(I gave the features one at a time, which seems to help the AI stay focused on the specific feature it was given, and based on that, try and guess the code for the other features), but changed the first prompt a little bit to see if there would be any major changes(I tried giving a more detailed explanation of the process I want the AI to follow). The AI provided good results, guessed some domain classes at first that were correct , with the DAOs, and started implementing code from the start( Which is the theme in every chat where I instructed it to provide the domain code first, whereas in other chats it was struggling to create the srtep definitions code without more explanation etc). Overall, the results were good for all the features, it guessed correctly some classes and it required very little prompt explanations in order for it to understand what it needs to do.*

# Chat 8

**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`.

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`.

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".

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.

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:

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:

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:

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:

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 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) {

assert borrowerDAO.getBorrowerByName(borrowerName) != null;

}

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

public void verifyPendingItemsMessage() {

// Assuming some verification logic

}

}

In this version, we're simulating the behavior of the data layer without relying on a real database connection. We're assuming certain behaviors for the DAO methods to mimic database interactions. Let me know if you need further adjustments or explanations!

**User**

**Now Do the same you did above for this feature :**

**Feature: Loaning items**

**The library application allows for the librarian to loan an item of a book to a borrower based**

**on some conditions of the system**

**User story: As a library member**

**I want to be able to borrow items**

**So that I can study them at home**

**Scenario: Successful loaning of an item**

**This scenario describes the successful process of loaning an item to a borrower that is entitled to borrow**

**Given the library has the item Harry Potter available**

**And George Red is a registered borrower**

**And George Red has 2 pending items to be returned**

**And George Red has been assigned a maximum lending limit of 5**

**When George Red borrows the item Harry Potter**

**Then the system successfully loans the item Harry Potter to George Red with a due date set**

**And George Red's pending items increase to 3**

**Scenario: Borrower can borrow only one item due to his lending limit**

**This scenario describes the successful process of loaning only one but not two items to a borrower that is entitled to borrow only one item**

**Given the library has the items Harry Potter and Moby Dick available**

**And George Red is a registered borrower**

**And George Red has 2 pending items to be returned**

**And George Red has been assigned a maximum lending limit of 3**

**When George Red tries to borrow both items**

**Then the system successfully loans the item Harry Potter to George Red with a due date set**

**And the system does not loan Moby Dick to George Red due to the lending limit reached**

**And George Red's pending items increase to 3**

**Scenario:Item not found**

**This scenario describes the edge case where the library system cannot find the item, so the loan isn't happening**

**Given the item Harry Potter is in the library but not in the system**

**And George Red is a registered borrower**

**When George Red tries to borrow the item Harry Potter**

**Then the system returns an error due to the item's status**

**And the system withdraws the item Harry Potter**

**Scenario:The borrower is not eligible to borrow**

**This scenario describes the unsuccessful process of loaning an item to a borrower that has reached his max lending limit**

**Given the library has the item Harry Potter available**

**And George Red is a registered borrower**

**And George Red has 3 pending items to be returned**

**And George Red has been assigned a maximum lending limit of 3**

**When George Red tries to borrow the item Harry Potter**

**Then the system doesn't allow the loan**

**And George Red's pending items remain 3**

**ChatGPT**

Sure, let's create the step definitions for the scenarios in the "Loaning items" feature. We'll follow a similar approach as before, avoiding mocking and handling scenarios without relying on a real database connection.

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

public class LoaningStepDefinitions {

private Library library = new Library();

private Borrower borrower;

private Item item;

private int initialPendingItems;

private boolean loanResult;

private String errorMessage;

@Given("^the library has the item ([^\"]\*) available$")

public void libraryHasItemAvailable(String itemName) {

item = new Item(itemName);

library.addItem(item);

}

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

public void borrowerIsRegistered(String borrowerName) {

borrower = new Borrower(borrowerName);

}

@Given("^([^\"]\*) has (\\d+) pending items to be returned$")

public void borrowerHasPendingItems(String borrowerName, int pendingItems) {

// Simulate borrower having pending items

borrower.setPendingItems(pendingItems);

initialPendingItems = pendingItems;

}

@Given("^([^\"]\*) has been assigned a maximum lending limit of (\\d+)$")

public void borrowerHasLendingLimit(String borrowerName, int lendingLimit) {

borrower.setLendingLimit(lendingLimit);

}

@When("^([^\"]\*) borrows the item ([^\"]\*)$")

public void borrowerBorrowsItem(String borrowerName, String itemName) {

try {

loanResult = library.loanItem(item, borrower);

} catch (Exception e) {

loanResult = false;

errorMessage = e.getMessage();

}

}

@Then("^the system successfully loans the item ([^\"]\*) to ([^\"]\*) with a due date set$")

public void verifySuccessfulLoan(String itemName, String borrowerName) {

assert loanResult;

assert borrower.hasItem(item);

// Assuming due date is set successfully

}

@Then("^the system does not loan ([^\"]\*) to ([^\"]\*) due to the lending limit reached$")

public void verifyLoanNotHappening(String itemName, String borrowerName) {

assert !loanResult;

assert errorMessage.equals("Maximum lending limit reached");

}

@Then("^the system returns an error due to the item's status$")

public void verifyItemNotFound() {

assert !loanResult;

assert errorMessage.equals("Item not found in the system");

}

@Then("^the system doesn't allow the loan$")

public void verifyLoanNotAllowed() {

assert !loanResult;

assert errorMessage.equals("Borrower not eligible to borrow");

}

@Then("^the system withdraws the item ([^\"]\*)$")

public void verifyItemWithdrawn(String itemName) {

assert !library.hasItem(item);

}

@Then("^George Red's pending items increase to (\\d+)$")

public void verifyPendingItemsIncrease(int newPendingItems) {

assert borrower.getPendingItems() == initialPendingItems + newPendingItems;

}

}

These step definitions cover all the scenarios mentioned in the "Loaning items" feature. We're simulating the behavior of the library and borrower objects without relying on an actual database. Let me know if you need further assistance or adjustments!

**User**

**use data access objects**

**ChatGPT**

Certainly! Let's integrate Data Access Objects (DAOs) to interact with the data layer. We'll implement the DAOs to simulate database interactions without relying on an actual database connection.

Here's how we can update the step definitions:

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

public class LoaningStepDefinitions {

private LibraryDAO libraryDAO = new LibraryDAO();

private BorrowerDAO borrowerDAO = new BorrowerDAO();

private Library library;

private Borrower borrower;

private Item item;

private int initialPendingItems;

private boolean loanResult;

private String errorMessage;

@Given("^the library has the item ([^\"]\*) available$")

public void libraryHasItemAvailable(String itemName) {

item = new Item(itemName);

libraryDAO.addItem(item);

}

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

public void borrowerIsRegistered(String borrowerName) {

borrower = new Borrower(borrowerName);

borrowerDAO.saveBorrower(borrower);

}

@Given("^([^\"]\*) has (\\d+) pending items to be returned$")

public void borrowerHasPendingItems(String borrowerName, int pendingItems) {

borrower = borrowerDAO.getBorrowerByName(borrowerName);

borrower.setPendingItems(pendingItems);

borrowerDAO.updateBorrower(borrower);

initialPendingItems = pendingItems;

}

@Given("^([^\"]\*) has been assigned a maximum lending limit of (\\d+)$")

public void borrowerHasLendingLimit(String borrowerName, int lendingLimit) {

borrower = borrowerDAO.getBorrowerByName(borrowerName);

borrower.setLendingLimit(lendingLimit);

borrowerDAO.updateBorrower(borrower);

}

@When("^([^\"]\*) borrows the item ([^\"]\*)$")

public void borrowerBorrowsItem(String borrowerName, String itemName) {

borrower = borrowerDAO.getBorrowerByName(borrowerName);

item = libraryDAO.getItemByName(itemName);

try {

loanResult = libraryDAO.loanItem(item, borrower);

} catch (Exception e) {

loanResult = false;

errorMessage = e.getMessage();

}

}

@Then("^the system successfully loans the item ([^\"]\*) to ([^\"]\*) with a due date set$")

public void verifySuccessfulLoan(String itemName, String borrowerName) {

assert loanResult;

borrower = borrowerDAO.getBorrowerByName(borrowerName);

assert borrower.hasItem(item);

// Assuming due date is set successfully

}

@Then("^the system does not loan ([^\"]\*) to ([^\"]\*) due to the lending limit reached$")

public void verifyLoanNotHappening(String itemName, String borrowerName) {

assert !loanResult;

assert errorMessage.equals("Maximum lending limit reached");

}

@Then("^the system returns an error due to the item's status$")

public void verifyItemNotFound() {

assert !loanResult;

assert errorMessage.equals("Item not found in the system");

}

@Then("^the system doesn't allow the loan$")

public void verifyLoanNotAllowed() {

assert !loanResult;

assert errorMessage.equals("Borrower not eligible to borrow");

}

@Then("^the system withdraws the item ([^\"]\*)$")

public void verifyItemWithdrawn(String itemName) {

item = libraryDAO.getItemByName(itemName);

assert item != null;

assert !item.isAvailable();

}

@Then("^George Red's pending items increase to (\\d+)$")

public void verifyPendingItemsIncrease(int newPendingItems) {

borrower = borrowerDAO.getBorrowerByName("George Red");

assert borrower != null;

assert borrower.getPendingItems() == initialPendingItems + newPendingItems;

}

}

These step definitions interact with the data layer through DAOs to simulate database interactions. We're assuming the behavior of the DAO methods to mimic database operations without relying on an actual database connection. Let me know if you need further assistance or have any questions!

**User**

**Now , based on the above, do the same thing for this feature: Feature: Returning a borrowed item to the library**

**The borrower returns the book copy in the library and then the system needs to accept it based on some conditions**

**User story: As a librarian**

**I want to accurately record the return of one or more books**

**So that books are returned on time, and late returns are fined**

**Scenario:Successful return of an item**

**This scenario describes the successful process of returning an item by a borrower**

**Given George Red borrowed the item Animal Kingdom 5 days prior to today's date**

**And George Red has been assigned maximum lending days of 10**

**When the return of Animal Kingdom is processed**

**Then the system marks the state of Animal Kingdom as AVAILABLE**

**And George Red has one less pending item**

**And George Red does not pay a fine**

**And the return date of the loan is set to today's date**

**Scenario:Late return of an item from a borrower**

**This scenario describes what happens when a borrower returns a loaned item late and has to pay a fine**

**Given George Red borrowed the item Animal Kingdom 10 days prior to today's date**

**And George Red has been assigned maximum lending days of 7**

**When the return of Animal Kingdom is processed**

**Then the system marks the state of Animal Kingdom as AVAILABLE**

**And George Red pays a fine based on the borrower category**

**And the return date of the loan is set to today's date**

**ChatGPT**

Sure, let's create the step definitions for the scenarios in the "Returning a borrowed item to the library" feature. We'll follow a similar approach as before, using Data Access Objects (DAOs) to interact with the data layer without relying on an actual database connection.

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

public class ReturningStepDefinitions {

private LibraryDAO libraryDAO = new LibraryDAO();

private BorrowerDAO borrowerDAO = new BorrowerDAO();

private Library library;

private Borrower borrower;

private Item item;

private int initialPendingItems;

private boolean returnResult;

private boolean lateReturn;

private double fineAmount;

@Given("^([^\"]\*) borrowed the item ([^\"]\*) (\\d+) days prior to today's date$")

public void borrowerBorrowedItem(String borrowerName, String itemName, int daysPrior) {

borrower = borrowerDAO.getBorrowerByName(borrowerName);

item = libraryDAO.getItemByName(itemName);

// Simulating borrower borrowing item daysPrior days ago

// This method can be implemented based on the actual logic in the system

}

@Given("^([^\"]\*) has been assigned maximum lending days of (\\d+)$")

public void borrowerHasMaximumLendingDays(String borrowerName, int maxLendingDays) {

borrower = borrowerDAO.getBorrowerByName(borrowerName);

borrower.setMaxLendingDays(maxLendingDays);

borrowerDAO.updateBorrower(borrower);

}

@When("^the return of ([^\"]\*) is processed$")

public void returnProcessed(String itemName) {

item = libraryDAO.getItemByName(itemName);

borrower = borrowerDAO.getBorrowerByName("George Red"); // Assuming fixed borrower for simplicity

// Calculate if the return is late based on the current date and return date of the loan

lateReturn = // Logic to check if return is late

// Process the return and calculate fine if applicable

returnResult = libraryDAO.processReturn(item, borrower, lateReturn);

if (lateReturn) {

fineAmount = // Calculate fine amount based on borrower category

}

}

@Then("^the system marks the state of ([^\"]\*) as AVAILABLE$")

public void verifyItemMarkedAsAvailable(String itemName) {

item = libraryDAO.getItemByName(itemName);

assert item.isAvailable();

}

@Then("^([^\"]\*) has one less pending item$")

public void verifyPendingItemsDecrease(String borrowerName) {

borrower = borrowerDAO.getBorrowerByName(borrowerName);

assert borrower.getPendingItems() == initialPendingItems - 1;

}

@Then("^([^\"]\*) does not pay a fine$")

public void verifyNoFine(String borrowerName) {

assert !lateReturn;

assert fineAmount == 0.0;

}

@Then("^([^\"]\*) pays a fine based on the borrower category$")

public void verifyFinePaid(String borrowerName) {

assert lateReturn;

assert fineAmount > 0.0;

}

@Then("^the return date of the loan is set to today's date$")

public void verifyReturnDateSetToToday() {

// Assuming verification logic

}

}

These step definitions interact with the data layer through DAOs to simulate database interactions. We're assuming the behavior of the DAO methods to mimic database operations without relying on an actual database connection. Let me know if you need further assistance or have any questions!

**User**

**now, do the same for this feature: Feature: Delayed Return Notification**

**User Story:As a Library Manager**

**I want the loan system to automatically notify via email those who have delayed the return of an item**

**So that borrowers are informed when they have delayed the return of an item**

**Scenario: Notifying the borrower via email**

**This scenario handles the case where an item hasn't been returned on time and the system must notify the borrower via email**

**Given George Red has borrowed the item Harry Potter**

**And Harry Potter's due date has passed**

**And George Red has an email address**

**When the system executes the delayed return notification process**

**Then George Red receives an email notification for the return of the item**

**Scenario: Borrower does not have an email address**

**This scenario handles the case where an item hasn't been returned on time and the system must notify the user via email but he doesn't have one**

**Given George Red has borrowed the item Harry Potter**

**And Harry Potter's due date has passed**

**And George Red does not have an email address**

**When the system executes the delayed return notification process**

**Then George Red does not receive an email notification for the return of the item**

**ChatGPT**

Let's create the step definitions for the scenarios in the "Delayed Return Notification" feature. We'll again use Data Access Objects (DAOs) to interact with the data layer without relying on an actual database connection.

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

public class DelayedReturnNotificationStepDefinitions {

private LibraryDAO libraryDAO = new LibraryDAO();

private BorrowerDAO borrowerDAO = new BorrowerDAO();

private NotificationService notificationService = new NotificationService();

private Library library;

private Borrower borrower;

private Item item;

private boolean emailSent;

@Given("^([^\"]\*) has borrowed the item ([^\"]\*)$")

public void borrowerHasBorrowedItem(String borrowerName, String itemName) {

borrower = borrowerDAO.getBorrowerByName(borrowerName);

item = libraryDAO.getItemByName(itemName);

libraryDAO.loanItem(item, borrower); // Assuming item is successfully loaned to borrower

}

@Given("^([^\"]\*)'s due date has passed$")

public void dueDateHasPassed(String borrowerName) {

// Assuming due date has passed for simplicity

}

@Given("^([^\"]\*) has an email address$")

public void borrowerHasEmailAddress(String borrowerName) {

borrower.setEmail("george.red@example.com"); // Assuming borrower has an email address

}

@Given("^([^\"]\*) does not have an email address$")

public void borrowerDoesNotHaveEmailAddress(String borrowerName) {

borrower.setEmail(null); // Assuming borrower does not have an email address

}

@When("^the system executes the delayed return notification process$")

public void executeDelayedReturnNotificationProcess() {

emailSent = notificationService.sendDelayedReturnNotification(borrower);

}

@Then("^([^\"]\*) receives an email notification for the return of the item$")

public void verifyEmailNotificationSent(String borrowerName) {

assert emailSent;

// Assuming verification logic for email notification

}

@Then("^([^\"]\*) does not receive an email notification for the return of the item$")

public void verifyNoEmailNotificationSent(String borrowerName) {

assert !emailSent;

// Assuming verification logic for absence of email notification

}

}

These step definitions interact with the data layer through DAOs to simulate database interactions. We're also simulating email notifications through a hypothetical `NotificationService`. Let me know if you need further assistance or have any questions!