In this chat, I gave Chat GPT the same information as in all the previous chats(only features at once), and didn’t ask it to generate the domain code at first, in order to see what the outcome would be, and the AI just created the step definition methods without any code and was waiting for me to give it prompts to help it understand , as in the earlier chats, so this chat was dismissed because nothing special could come out based on the first response(the results are very poor).

# Phase 1. Chat 15(New chat 9)

**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 will ask you to provide me the implementation code of the Step definitions and write as much code as you can for each step in order for them to be executable (hint: the step definitions will be split into different classes based on the different feature ). The architecture layers of the system are a domain layer containing the domain classes, the DAO layer containing the Data Access classes and a Service layer containing service classes that help with some parts of the logic of the system(like loaning and returning items). In the next message , I will provide you with the features/scenarios and then I want you to implement the step definitions code for each step for the steps to be executable**

**ChatGPT**

Sure, I'm ready to help you with that. Please provide me with the features/scenarios, and I'll assist you in implementing the step definitions.

**User**

**The first feature of the system is :**

**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**

**The second feature is :**

**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**

**The third feature is:**

**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**

**and the last feature is :**

**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**

**Now, implement All the step definitions code in java, split into files as I already told you**

**ChatGPT**

Sure, let's start by implementing the step definitions for the first feature: Borrower handling by the system.

Step definitions for the first feature could be structured like this:

1. \*\*RegisterBorrowerSteps.java\*\*

- Contains step definitions for registering a new borrower.

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

public class RegisterBorrowerSteps {

@Given("^George Red is not registered as a borrower$")

public void georgeRedIsNotRegisteredAsABorrower() {

// Implementation to check if George Red is not already registered as a borrower

}

@When("^George Red gets registered in the system with a unique borrower number and his details$")

public void georgeRedGetsRegisteredInTheSystemWithUniqueBorrowerNumberAndHisDetails() {

// Implementation to register George Red as a borrower

}

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

public void theSystemSuccessfullyStoresTheBorrowersDetails() {

// Implementation to verify that the borrower's details are successfully stored

}

}

2. \*\*UpdateBorrowerDetailsSteps.java\*\*

- Contains step definitions for updating borrower's details.

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

public class UpdateBorrowerDetailsSteps {

@Given("^George Red is registered as a borrower$")

public void georgeRedIsRegisteredAsABorrower() {

// Implementation to check if George Red is already registered as a borrower

}

@When("^George Red updates his borrowing details$")

public void georgeRedUpdatesHisBorrowingDetails() {

// Implementation to update George Red's borrowing details

}

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

public void theSystemSavesTheChanges() {

// Implementation to verify that the changes are successfully saved

}

}

3. \*\*DeleteBorrowerSteps.java\*\*

- Contains step definitions for deleting a borrower's account.

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

public class DeleteBorrowerSteps {

@Given("^George Red is registered as a borrower$")

public void georgeRedIsRegisteredAsABorrower() {

// Implementation to check if George Red is already registered as a borrower

}

@When("^the system deletes George Red's account$")

public void theSystemDeletesGeorgesRedsAccount() {

// Implementation to delete George Red's account

}

@Then("^the system removes George Red's details$")

public void theSystemRemovesGeorgesRedsDetails() {

// Implementation to verify that George Red's details are successfully removed

}

}

These are the step definitions for the first feature. We can continue with the implementation of step definitions for the remaining features in a similar manner. Let me know if you need further assistance!

**User**

**create as much executable code as possible for this feature**

To create executable step definitions for the "Borrower handling by the system" feature in Java, we will use Cucumber-Java for writing and executing the scenarios. Below is the code for the step definitions split into corresponding classes based on the scenarios provided:

1. \*\*RegisterBorrowerSteps.java\*\*

```java

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

import org.junit.Assert;

public class RegisterBorrowerSteps {

private BorrowerService borrowerService; // Assuming BorrowerService handles borrower operations

@Given("^George Red is not registered as a borrower$")

public void georgeRedIsNotRegisteredAsABorrower() {

// Implementation to check if George Red is not already registered as a borrower

Assert.assertFalse(borrowerService.isBorrowerRegistered("George Red"));

}

@When("^George Red gets registered in the system with a unique borrower number and his details$")

public void georgeRedGetsRegisteredInTheSystemWithUniqueBorrowerNumberAndHisDetails() {

// Implementation to register George Red as a borrower

borrowerService.registerBorrower("George Red", "123456"); // Example unique borrower number

}

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

public void theSystemSuccessfullyStoresTheBorrowersDetails() {

// Implementation to verify that the borrower's details are successfully stored

Assert.assertTrue(borrowerService.isBorrowerRegistered("George Red"));

}

}

```

2. \*\*UpdateBorrowerDetailsSteps.java\*\*

```java

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

import org.junit.Assert;

public class UpdateBorrowerDetailsSteps {

private BorrowerService borrowerService; // Assuming BorrowerService handles borrower operations

@Given("^George Red is registered as a borrower$")

public void georgeRedIsRegisteredAsABorrower() {

// Implementation to check if George Red is already registered as a borrower

Assert.assertTrue(borrowerService.isBorrowerRegistered("George Red"));

}

@When("^George Red updates his borrowing details$")

public void georgeRedUpdatesHisBorrowingDetails() {

// Implementation to update George Red's borrowing details

borrowerService.updateBorrowerDetails("George Red", "New Details"); // Example update

}

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

public void theSystemSavesTheChanges() {

// Implementation to verify that the changes are successfully saved

// Assuming the update method returns a boolean indicating success

Assert.assertTrue(borrowerService.updateBorrowerDetails("George Red", "New Details"));

}

}

```

3. \*\*DeleteBorrowerSteps.java\*\*

```java

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

import org.junit.Assert;

public class DeleteBorrowerSteps {

private BorrowerService borrowerService; // Assuming BorrowerService handles borrower operations

@Given("^George Red is registered as a borrower$")

public void georgeRedIsRegisteredAsABorrower() {

// Implementation to check if George Red is already registered as a borrower

Assert.assertTrue(borrowerService.isBorrowerRegistered("George Red"));

}

@When("^the system deletes George Red's account$")

public void theSystemDeletesGeorgesRedsAccount() {

// Implementation to delete George Red's account

borrowerService.deleteBorrower("George Red");

}

@Then("^the system removes George Red's details$")

public void theSystemRemovesGeorgesRedsDetails() {

// Implementation to verify that George Red's details are successfully removed

Assert.assertFalse(borrowerService.isBorrowerRegistered("George Red"));

}

}

```

### Explanation:

- \*\*BorrowerService\*\*: This is assumed to be a service class responsible for managing borrower operations. It encapsulates methods like `registerBorrower`, `updateBorrowerDetails`, `deleteBorrower`, and `isBorrowerRegistered`.

- \*\*Assertions\*\*: We use JUnit assertions (`Assert.assertTrue` and `Assert.assertFalse`) to validate the expected behavior of each step.

### Notes:

- Ensure you have set up your Cucumber environment correctly with dependencies in your `pom.xml` or `build.gradle`.

- Replace placeholder methods (`borrowerService.registerBorrower`, etc.) with actual implementations relevant to your application's architecture (DAO, Service layers, etc.).

- These step definitions focus on behavior verification rather than the actual detailed implementation of each method, which should ideally be handled in your service layer.

This structure should provide a solid foundation for implementing and testing the behavior described in your scenarios. Adjust the service methods and assertions as per your application's specific requirements and business logic.

**User**

**now, based on the above, create the step definitions for the feature: Loaning Items**

Certainly! Let's create the step definitions for the "Loaning Items" feature based on the scenarios provided:

1. \*\*SuccessfulLoaningSteps.java\*\*

```java

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

import org.junit.Assert;

public class SuccessfulLoaningSteps {

private LibraryService libraryService; // Assuming LibraryService handles library operations

private BorrowerService borrowerService; // Assuming BorrowerService handles borrower operations

@Given("^the library has the item Harry Potter available$")

public void theLibraryHasTheItemHarryPotterAvailable() {

// Implementation to check if the library has the item Harry Potter available

Assert.assertTrue(libraryService.isItemAvailable("Harry Potter"));

}

@Given("^George Red is a registered borrower$")

public void georgeRedIsARegisteredBorrower() {

// Implementation to check if George Red is a registered borrower

Assert.assertTrue(borrowerService.isBorrowerRegistered("George Red"));

}

@Given("^George Red has (\\d+) pending items to be returned$")

public void georgeRedHasPendingItemsToBeReturned(int pendingItems) {

// Implementation to verify the number of pending items George Red has

Assert.assertEquals(pendingItems, borrowerService.getPendingItemsCount("George Red"));

}

@Given("^George Red has been assigned a maximum lending limit of (\\d+)$")

public void georgeRedHasBeenAssignedAMaximumLendingLimit(int maxLimit) {

// Implementation to assign a maximum lending limit to George Red

borrowerService.assignMaxLendingLimit("George Red", maxLimit);

}

@When("^George Red borrows the item Harry Potter$")

public void georgeRedBorrowsTheItemHarryPotter() {

// Implementation to borrow the item Harry Potter by George Red

libraryService.borrowItem("George Red", "Harry Potter");

}

@Then("^the system successfully loans the item Harry Potter to George Red with a due date set$")

public void theSystemSuccessfullyLoansTheItemHarryPotterToGeorgeRedWithDueDateSet() {

// Implementation to verify that the item Harry Potter is successfully loaned

Assert.assertTrue(libraryService.isItemBorrowed("Harry Potter", "George Red"));

}

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

public void georgeRedsPendingItemsIncreaseTo(int newPendingItems) {

// Implementation to verify the new number of pending items for George Red

Assert.assertEquals(newPendingItems, borrowerService.getPendingItemsCount("George Red"));

}

}

```

2. \*\*BorrowerLimitSteps.java\*\*

```java

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

import org.junit.Assert;

public class BorrowerLimitSteps {

private LibraryService libraryService; // Assuming LibraryService handles library operations

private BorrowerService borrowerService; // Assuming BorrowerService handles borrower operations

@Given("^the library has the items Harry Potter and Moby Dick available$")

public void theLibraryHasTheItemsHarryPotterAndMobyDickAvailable() {

// Implementation to check if the library has the items Harry Potter and Moby Dick available

Assert.assertTrue(libraryService.isItemAvailable("Harry Potter"));

Assert.assertTrue(libraryService.isItemAvailable("Moby Dick"));

}

@Given("^George Red is a registered borrower$")

public void georgeRedIsARegisteredBorrower() {

// Implementation to check if George Red is a registered borrower

Assert.assertTrue(borrowerService.isBorrowerRegistered("George Red"));

}

@Given("^George Red has (\\d+) pending items to be returned$")

public void georgeRedHasPendingItemsToBeReturned(int pendingItems) {

// Implementation to verify the number of pending items George Red has

Assert.assertEquals(pendingItems, borrowerService.getPendingItemsCount("George Red"));

}

@Given("^George Red has been assigned a maximum lending limit of (\\d+)$")

public void georgeRedHasBeenAssignedAMaximumLendingLimit(int maxLimit) {

// Implementation to assign a maximum lending limit to George Red

borrowerService.assignMaxLendingLimit("George Red", maxLimit);

}

@When("^George Red tries to borrow both items$")

public void georgeRedTriesToBorrowBothItems() {

// Implementation to attempt to borrow both items by George Red

libraryService.borrowItem("George Red", "Harry Potter");

libraryService.borrowItem("George Red", "Moby Dick");

}

@Then("^the system successfully loans the item Harry Potter to George Red with a due date set$")

public void theSystemSuccessfullyLoansTheItemHarryPotterToGeorgeRedWithDueDateSet() {

// Implementation to verify that only Harry Potter is successfully loaned

Assert.assertTrue(libraryService.isItemBorrowed("Harry Potter", "George Red"));

}

@Then("^the system does not loan Moby Dick to George Red due to the lending limit reached$")

public void theSystemDoesNotLoanMobyDickToGeorgeRedDueToLendingLimitReached() {

// Implementation to verify that Moby Dick is not loaned due to lending limit

Assert.assertFalse(libraryService.isItemBorrowed("Moby Dick", "George Red"));

}

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

public void georgeRedsPendingItemsIncreaseTo(int newPendingItems) {

// Implementation to verify the new number of pending items for George Red

Assert.assertEquals(newPendingItems, borrowerService.getPendingItemsCount("George Red"));

}

}

```

3. \*\*ItemNotFoundSteps.java\*\*

```java

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

import org.junit.Assert;

public class ItemNotFoundSteps {

private LibraryService libraryService; // Assuming LibraryService handles library operations

private BorrowerService borrowerService; // Assuming BorrowerService handles borrower operations

@Given("^the item Harry Potter is in the library but not in the system$")

public void theItemHarryPotterIsInTheLibraryButNotInTheSystem() {

// Implementation to simulate that Harry Potter is in the library but not correctly registered in the system

Assert.assertTrue(libraryService.isItemInLibraryButNotInSystem("Harry Potter"));

}

@Given("^George Red is a registered borrower$")

public void georgeRedIsARegisteredBorrower() {

// Implementation to check if George Red is a registered borrower

Assert.assertTrue(borrowerService.isBorrowerRegistered("George Red"));

}

@When("^George Red tries to borrow the item Harry Potter$")

public void georgeRedTriesToBorrowTheItemHarryPotter() {

// Implementation for George Red attempting to borrow the item Harry Potter

libraryService.borrowItem("George Red", "Harry Potter");

}

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

public void theSystemReturnsAnErrorDueToItemsStatus() {

// Implementation to verify that the system returns an error due to item's status

Assert.assertTrue(libraryService.isErrorReturned());

}

@Then("^the system withdraws the item Harry Potter$")

public void theSystemWithdrawsTheItemHarryPotter() {

// Implementation to verify that the system withdraws the item Harry Potter

Assert.assertTrue(libraryService.isItemWithdrawn("Harry Potter"));

}

}

```

4. \*\*BorrowerNotEligibleSteps.java\*\*

```java

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

import org.junit.Assert;

public class BorrowerNotEligibleSteps {

private LibraryService libraryService; // Assuming LibraryService handles library operations

private BorrowerService borrowerService; // Assuming BorrowerService handles borrower operations

@Given("^the library has the item Harry Potter available$")

public void theLibraryHasTheItemHarryPotterAvailable() {

// Implementation to check if the library has the item Harry Potter available

Assert.assertTrue(libraryService.isItemAvailable("Harry Potter"));

}

@Given("^George Red is a registered borrower$")

public void georgeRedIsARegisteredBorrower() {

// Implementation to check if George Red is a registered borrower

Assert.assertTrue(borrowerService.isBorrowerRegistered("George Red"));

}

@Given("^George Red has (\\d+) pending items to be returned$")

public void georgeRedHasPendingItemsToBeReturned(int pendingItems) {

// Implementation to verify the number of pending items George Red has

Assert.assertEquals(pendingItems, borrowerService.getPendingItemsCount("George Red"));

}

@Given("^George Red has been assigned a maximum lending limit of (\\d+)$")

public void georgeRedHasBeenAssignedAMaximumLendingLimit(int maxLimit) {

// Implementation to assign a maximum lending limit to George Red

borrowerService.assignMaxLendingLimit("George Red", maxLimit);

}

@When("^George Red tries to borrow the item Harry Potter$")

public void georgeRedTriesToBorrowTheItemHarryPotter() {

// Implementation for George Red attempting to borrow the item Harry Potter

libraryService.borrowItem("George Red", "Harry Potter");

}

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

public void theSystemDoesntAllowTheLoan() {

// Implementation to verify that the system doesn't allow the loan

Assert.assertFalse(libraryService.isItemBorrowed("Harry Potter", "George Red"));

}

@Then("^

George Red's pending items remain (\\d+)$")

public void georgeRedsPendingItemsRemain(int pendingItems) {

// Implementation to verify that George Red's pending items remain unchanged

Assert.assertEquals(pendingItems, borrowerService.getPendingItemsCount("George Red"));

}

}

```

### Explanation:

- \*\*LibraryService\*\*: This is assumed to be a service class responsible for managing library operations such as `isItemAvailable`, `borrowItem`, `isItemBorrowed`, etc.

- \*\*BorrowerService\*\*: This is assumed to be a service class responsible for managing borrower operations such as `isBorrowerRegistered`, `getPendingItemsCount`, `assignMaxLendingLimit`, etc.

- \*\*Assertions\*\*: We use JUnit assertions (`Assert.assertTrue`, `Assert.assertFalse`, `Assert.assertEquals`) to validate the expected behavior of each step.

### Notes:

- Adjust the method implementations (`libraryService.borrowItem`, etc.) with actual logic relevant to your application's architecture.

- Ensure that your Cucumber setup includes the necessary dependencies and configurations for executing these step definitions.

- These step definitions focus on verifying the behavior described in the scenarios for loaning items. Adjust them according to your specific business logic and requirements.

These step definitions should provide a solid basis for implementing and testing the behavior described in the "Loaning Items" feature of your library system application. Adjust the service methods and assertions as per your application's specific requirements and business logic.

**User**

**now, with the same pattern, create the step definitions for the feature: Returning a borrowed item in the library , with as much code as possible**

Certainly! Let's create the step definitions for the "Returning a borrowed item to the library" feature based on the scenarios provided:

1. \*\*SuccessfulReturnSteps.java\*\*

```java

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

import org.junit.Assert;

import java.time.LocalDate;

public class SuccessfulReturnSteps {

private LibraryService libraryService; // Assuming LibraryService handles library operations

private BorrowerService borrowerService; // Assuming BorrowerService handles borrower operations

@Given("^George Red borrowed the item Animal Kingdom (\\d+) days prior to today's date$")

public void georgeRedBorrowedTheItemAnimalKingdomDaysPriorToTodaysDate(int daysAgo) {

// Simulate George Red borrowing the item Animal Kingdom 'daysAgo' days ago

LocalDate borrowedDate = LocalDate.now().minusDays(daysAgo);

libraryService.borrowItem("George Red", "Animal Kingdom", borrowedDate);

}

@Given("^George Red has been assigned maximum lending days of (\\d+)$")

public void georgeRedHasBeenAssignedMaximumLendingDays(int maxLendingDays) {

// Implementation to assign maximum lending days to George Red

borrowerService.assignMaxLendingDays("George Red", maxLendingDays);

}

@When("^the return of Animal Kingdom is processed$")

public void theReturnOfAnimalKingdomIsProcessed() {

// Implementation to process the return of Animal Kingdom by George Red

libraryService.returnItem("George Red", "Animal Kingdom");

}

@Then("^the system marks the state of Animal Kingdom as AVAILABLE$")

public void theSystemMarksTheStateOfAnimalKingdomAsAvailable() {

// Implementation to verify that the item Animal Kingdom is marked as AVAILABLE

Assert.assertTrue(libraryService.isItemAvailable("Animal Kingdom"));

}

@Then("^George Red has one less pending item$")

public void georgeRedHasOneLessPendingItem() {

// Implementation to verify that George Red has one less pending item

Assert.assertEquals(2, borrowerService.getPendingItemsCount("George Red")); // Assuming George Red started with 3 pending items

}

@Then("^George Red does not pay a fine$")

public void georgeRedDoesNotPayAFine() {

// Implementation to verify that George Red does not pay a fine

Assert.assertFalse(borrowerService.hasFine("George Red"));

}

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

public void theReturnDateOfTheLoanIsSetToTodaysDate() {

// Implementation to verify that the return date of the loan is set to today's date

LocalDate expectedReturnDate = LocalDate.now();

Assert.assertEquals(expectedReturnDate, libraryService.getReturnDate("Animal Kingdom", "George Red"));

}

}

```

2. \*\*LateReturnSteps.java\*\*

```java

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

import org.junit.Assert;

import java.time.LocalDate;

public class LateReturnSteps {

private LibraryService libraryService; // Assuming LibraryService handles library operations

private BorrowerService borrowerService; // Assuming BorrowerService handles borrower operations

@Given("^George Red borrowed the item Animal Kingdom (\\d+) days prior to today's date$")

public void georgeRedBorrowedTheItemAnimalKingdomDaysPriorToTodaysDate(int daysAgo) {

// Simulate George Red borrowing the item Animal Kingdom 'daysAgo' days ago

LocalDate borrowedDate = LocalDate.now().minusDays(daysAgo);

libraryService.borrowItem("George Red", "Animal Kingdom", borrowedDate);

}

@Given("^George Red has been assigned maximum lending days of (\\d+)$")

public void georgeRedHasBeenAssignedMaximumLendingDays(int maxLendingDays) {

// Implementation to assign maximum lending days to George Red

borrowerService.assignMaxLendingDays("George Red", maxLendingDays);

}

@When("^the return of Animal Kingdom is processed$")

public void theReturnOfAnimalKingdomIsProcessed() {

// Implementation to process the return of Animal Kingdom by George Red

libraryService.returnItem("George Red", "Animal Kingdom");

}

@Then("^the system marks the state of Animal Kingdom as AVAILABLE$")

public void theSystemMarksTheStateOfAnimalKingdomAsAvailable() {

// Implementation to verify that the item Animal Kingdom is marked as AVAILABLE

Assert.assertTrue(libraryService.isItemAvailable("Animal Kingdom"));

}

@Then("^George Red pays a fine based on the borrower category$")

public void georgeRedPaysAFineBasedOnTheBorrowerCategory() {

// Implementation to verify that George Red pays a fine

Assert.assertTrue(borrowerService.hasFine("George Red"));

}

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

public void theReturnDateOfTheLoanIsSetToTodaysDate() {

// Implementation to verify that the return date of the loan is set to today's date

LocalDate expectedReturnDate = LocalDate.now();

Assert.assertEquals(expectedReturnDate, libraryService.getReturnDate("Animal Kingdom", "George Red"));

}

}

```

### Explanation:

- \*\*LibraryService\*\*: This is assumed to be a service class responsible for managing library operations such as `borrowItem`, `returnItem`, `isItemAvailable`, `getReturnDate`, etc.

- \*\*BorrowerService\*\*: This is assumed to be a service class responsible for managing borrower operations such as `assignMaxLendingDays`, `getPendingItemsCount`, `hasFine`, etc.

- \*\*Assertions\*\*: We use JUnit assertions (`Assert.assertTrue`, `Assert.assertFalse`, `Assert.assertEquals`) to validate the expected behavior of each step.

### Notes:

- Adjust the method implementations (`libraryService.returnItem`, etc.) with actual logic relevant to your application's architecture.

- Ensure that your Cucumber setup includes the necessary dependencies and configurations for executing these step definitions.

- These step definitions focus on verifying the behavior described in the scenarios for returning borrowed items. Adjust them according to your specific business logic and requirements.

These step definitions should provide a solid basis for implementing and testing the behavior described in the "Returning a borrowed item to the library" feature of your library system application. Adjust the service methods and assertions as per your application's specific requirements and business logic.

**User**

**now do the same for the feature: Delayed Notification**

Let's create the step definitions for the "Delayed Return Notification" feature based on the scenarios provided:

1. \*\*DelayedNotificationSteps.java\*\*

```java

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

import org.junit.Assert;

import java.time.LocalDate;

public class DelayedNotificationSteps {

private LibraryService libraryService; // Assuming LibraryService handles library operations

private BorrowerService borrowerService; // Assuming BorrowerService handles borrower operations

private NotificationService notificationService; // Assuming NotificationService handles notification operations

@Given("^George Red has borrowed the item Harry Potter$")

public void georgeRedHasBorrowedTheItemHarryPotter() {

// Simulate George Red borrowing the item Harry Potter

libraryService.borrowItem("George Red", "Harry Potter");

}

@Given("^Harry Potter's due date has passed$")

public void harryPottersDueDateHasPassed() {

// Simulate setting a due date that has passed for Harry Potter

LocalDate dueDate = LocalDate.now().minusDays(1); // Assuming due date was yesterday

libraryService.setDueDate("Harry Potter", dueDate);

}

@Given("^George Red has an email address$")

public void georgeRedHasAnEmailAddress() {

// Implementation to verify that George Red has an email address

Assert.assertTrue(borrowerService.hasEmailAddress("George Red"));

}

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

public void theSystemExecutesTheDelayedReturnNotificationProcess() {

// Implementation to execute the delayed return notification process

notificationService.sendDelayedReturnNotification("George Red", "Harry Potter");

}

@Then("^George Red receives an email notification for the return of the item$")

public void georgeRedReceivesAnEmailNotificationForTheReturnOfTheItem() {

// Implementation to verify that George Red receives an email notification

Assert.assertTrue(notificationService.isEmailSent("George Red", "Delayed Return Notification"));

}

}

```

2. \*\*NoEmailNotificationSteps.java\*\*

```java

import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

import org.junit.Assert;

import java.time.LocalDate;

public class NoEmailNotificationSteps {

private LibraryService libraryService; // Assuming LibraryService handles library operations

private BorrowerService borrowerService; // Assuming BorrowerService handles borrower operations

private NotificationService notificationService; // Assuming NotificationService handles notification operations

@Given("^George Red has borrowed the item Harry Potter$")

public void georgeRedHasBorrowedTheItemHarryPotter() {

// Simulate George Red borrowing the item Harry Potter

libraryService.borrowItem("George Red", "Harry Potter");

}

@Given("^Harry Potter's due date has passed$")

public void harryPottersDueDateHasPassed() {

// Simulate setting a due date that has passed for Harry Potter

LocalDate dueDate = LocalDate.now().minusDays(1); // Assuming due date was yesterday

libraryService.setDueDate("Harry Potter", dueDate);

}

@Given("^George Red does not have an email address$")

public void georgeRedDoesNotHaveAnEmailAddress() {

// Implementation to verify that George Red does not have an email address

Assert.assertFalse(borrowerService.hasEmailAddress("George Red"));

}

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

public void theSystemExecutesTheDelayedReturnNotificationProcess() {

// Implementation to execute the delayed return notification process

notificationService.sendDelayedReturnNotification("George Red", "Harry Potter");

}

@Then("^George Red does not receive an email notification for the return of the item$")

public void georgeRedDoesNotReceiveAnEmailNotificationForTheReturnOfTheItem() {

// Implementation to verify that George Red does not receive an email notification

Assert.assertFalse(notificationService.isEmailSent("George Red", "Delayed Return Notification"));

}

}

```

### Explanation:

- \*\*LibraryService\*\*: This is assumed to be a service class responsible for managing library operations such as `borrowItem`, `setDueDate`, etc.

- \*\*BorrowerService\*\*: This is assumed to be a service class responsible for managing borrower operations such as `hasEmailAddress`, etc.

- \*\*NotificationService\*\*: This is assumed to be a service class responsible for managing notification operations such as `sendDelayedReturnNotification`, `isEmailSent`, etc.

- \*\*Assertions\*\*: We use JUnit assertions (`Assert.assertTrue`, `Assert.assertFalse`) to validate the expected behavior of each step.

### Notes:

- Adjust the method implementations (`notificationService.sendDelayedReturnNotification`, etc.) with actual logic relevant to your application's architecture.

- Ensure that your Cucumber setup includes the necessary dependencies and configurations for executing these step definitions.

- These step definitions focus on verifying the behavior described in the scenarios for delayed return notification. Adjust them according to your specific business logic and requirements.

These step definitions should provide a solid foundation for implementing and testing the behavior described in the "Delayed Return Notification" feature of your library system application. Adjust the service methods and assertions as per your application's specific requirements and business logic.