Σε αυτή τη συνομιλία, παρείχα στο LLM όλα τα features του συστήματος ένα προς ένα, μαζί με τα ονόματα των κλάσεων domain που πρέπει να χρησιμοποιήσει, την αρχιτεκτονική και γενικές πληροφορίες για το σύστημα. Η διαφορά αυτή τη φορά ήταν ότι, μετά από κάθε feature που δόθηκε, ζήτησα από το LLM να δημιουργήσει τον κώδικα για τις κλάσεις domain, DAOs και services. Το LLM κατάλαβε πολύ καλά αυτό το βήμα και δεν το ξέχασε, δημιουργώντας κάθε feature με την ίδια διαδικασία. Ωστόσο, χρειάστηκε κάποια επανάληψη και υπενθύμιση για να ολοκληρώσει όλα τα step definitions για κάθε feature. Συνολικά, χρησιμοποίησε αρκετά καλά την αρχιτεκτονική και μάντεψε πολλές ιδιότητες και σχέσεις μεταξύ των κλάσεων. Παρά την καλή χρήση της αρχιτεκτονικής, οι step definitions δεν ήταν όλες εξαιρετικές. Μερικές ήταν πολύ απλοϊκές, ενώ κάποιες άλλες ήταν αποδεκτές.

Phase 2 Chat 4

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 the name of the domain classes I want you to use and with these features/scenarios one at a time and then I will ask you to provide me the implementation code of the Step definitions for each feature, in java. I want your architecture to have Domain classes, Data Access Objects for saving/deleting entities and Service classes. Do you have any questions?

ChatGPT

That sounds like a great approach to building and testing your library system application! Using Gherkin scenarios along with domain-driven design will help structure your application effectively. To ensure I can provide the most accurate and helpful Java implementations for your step definitions, it would be useful to know:

1. **Domain Classes**: Could you provide the details or fields of the domain classes you have in mind?

2. **Data Access Objects (DAOs)**: Do you have specific requirements or methods that should be included in the DAOs? For example, basic CRUD operations or something more specific?

3. **Service Classes**: What specific responsibilities or methods should these classes handle? Typically, they would involve business logic that interacts with the DAOs.

4. **Gherkin Scenarios**: Please provide the specific Gherkin scenarios for which you need step definitions implemented.

5. **Environment Setup**: What libraries or frameworks are you using for your testing and dependency injection (e.g., JUnit, Spring, Cucumber)?

With this information, I can create step definitions that integrate seamlessly with your existing architecture and testing framework.

User

The Domain classes you will need are : Book, Borrower, BorrowerCategory, Item, ItemState, Loan, Person. In the next message, I will provide you with the first feature of the system

ChatGPT

Thanks for providing the names of the domain classes! As we move forward with the implementation, it would be helpful to know more about the properties and relationships between these classes. For instance:

- **Book**: What attributes does it have? (e.g., title, author, ISBN)

- **Borrower**: Does it extend the Person class? What specific attributes are associated with a Borrower?

- **BorrowerCategory**: What kind of categories are there? (e.g., student, faculty, external)

- **Item**: How does it relate to Book? Does it represent a physical copy of a book?

- **ItemState**: What are the possible states? (e.g., available, checked out, under repair)

- **Loan**: What information does this class contain? (e.g., due date, loan date)

- **Person**: What are the basic attributes of a Person? (e.g., name, contact information)

Understanding the attributes and relationships will help ensure that the DAOs and service classes are properly implemented, and that the step definitions align well with the underlying domain model.

Feel free to provide the first feature when you're ready, and any additional details about these domain classes!

User

The first feature 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

Create the domain/daos/services code first

ChatGPT

To implement the borrower handling system effectively, let's first establish the domain models, Data Access Objects (DAOs), and service classes based on the scenarios and domain classes you provided. Here’s a simplified approach to each component:

Domain Models

We'll define basic Java classes for each domain model:

Person

```java
public class Person {
    private Long id;
    private String name;
    private String contactInfo; // assuming contact info is a string

    // Constructor, getters and setters
}
```

Borrower

```java
public class Borrower extends Person {
    private String borrowerNumber;
    private BorrowerCategory category;

    // Constructor, getters and setters
}
```

BorrowerCategory

```java
public enum BorrowerCategory {
    STUDENT, FACULTY, EXTERNAL;
}
```

Item

```java
public class Item {
    private Long id;
    private Book book;
    private ItemState state;

    // Constructor, getters and setters
}
```

ItemState

```java
public enum ItemState {
    AVAILABLE, CHECKED_OUT, UNDER_REPAIR;
}
```

Loan

```java
public class Loan {
    private Long id;
    private Borrower borrower;
    private Item item;
    private LocalDate dueDate;
    private LocalDate loanDate;

    // Constructor, getters and setters
}
```

Data Access Objects (DAOs)

These interfaces will interact with the database to manage the persistence of data.

BorrowerDAO

```java
public interface BorrowerDAO {
    void addBorrower(Borrower borrower);
    void updateBorrower(Borrower borrower);
    void deleteBorrower(Borrower borrower);
    Borrower findBorrowerByBorrowerNumber(String borrowerNumber);
    List<Loan> findLoansByBorrower(Borrower borrower);
}
```

Service Classes

These classes will contain the business logic.

BorrowerService

```java
public class BorrowerService {
    private BorrowerDAO borrowerDAO;
    public BorrowerService(BorrowerDAO borrowerDAO) {
        this.borrowerDAO = borrowerDAO;
    }

    public void registerBorrower(Borrower borrower) throws Exception {
        if (borrowerDAO.findBorrowerByBorrowerNumber(borrower.getBorrowerNumber()) != null)
{
            throw new Exception("Borrower already exists.");
        }
        borrowerDAO.addBorrower(borrower);
    }

    public void updateBorrower(Borrower borrower) throws Exception {
        if (borrowerDAO.findBorrowerByBorrowerNumber(borrower.getBorrowerNumber()) ==
null) {
            throw new Exception("Borrower does not exist.");
        }
        borrowerDAO.updateBorrower(borrower);
    }

    public void deleteBorrower(String borrowerNumber) throws Exception {
        Borrower borrower = borrowerDAO.findBorrowerByBorrowerNumber(borrowerNumber);

        if (borrower == null) {

            throw new Exception("Borrower does not exist.");

        }

        if (!borrowerDAO.findLoansByBorrower(borrower).isEmpty()) {

            throw new Exception("Borrower has pending items.");

        }

        borrowerDAO.deleteBorrower(borrower);

    }

}
```

This architecture defines the primary components of your system: domain models for representing entities, DAOs for database interactions, and services for encapsulating business logic. Once you confirm or adjust the attributes and relationships, we can then proceed to implement the step definitions for the Gherkin scenarios you've outlined.

User

Provide me with all the step definitions of this first feature

ChatGPT

To create the step definitions for the Gherkin scenarios related to borrower management in your library system, we will need to integrate these steps with Cucumber, a popular tool for running automated tests written in a Behavior-Driven Development (BDD) style. Below are the step definitions in Java that correspond to the scenarios you provided. These steps assume the use of Cucumber-Java and JUnit.

Step Definitions for Borrower Handling

First, make sure you have Cucumber set up with your project. You'll need to add dependencies for Cucumber-Java and Cucumber-JUnit in your `pom.xml` if you're using Maven, or in your `build.gradle` if you're using Gradle.

Here's the implementation of the step definitions:

```java
import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import static org.junit.Assert.*;

public class BorrowerSteps {

    private BorrowerService borrowerService;
    private Borrower borrower;
    private String operationResult;
    public BorrowerSteps() {
        BorrowerDAO borrowerDAO = new BorrowerDAOImpl(); // Assuming you have an implementation
        borrowerService = new BorrowerService(borrowerDAO);
    }

    @Given("^{String} is not registered as a borrower$")
    public void georgeRedIsNotRegisteredAsABorrower(String name) {
        assertNull(borrowerService.findBorrowerByName(name));
    }

    @When("^{String} gets registered in the system with a unique borrower number and his details$")
    public void georgeRedGetsRegisteredInTheSystemWithAUniqueBorrowerNumberAndHisDetails(String name) {
        try {
            borrower = new Borrower();
            borrower.setName(name);
            borrower.setBorrowerNumber("BR123456");
            borrower.setCategory(BorrowerCategory.STUDENT);
            borrowerService.registerBorrower(borrower);
            operationResult = "Success";
        } catch (Exception e) {
            operationResult = e.getMessage();
        }
    }

    @Then("^the system successfully stores the borrower's details$")
    public void theSystemSuccessfullyStoresTheBorrowersDetails() {
        assertEquals("Success", operationResult);
    }

    @Given("^{String} is registered as a borrower$")
    public void georgeRedIsRegisteredAsABorrower(String name) {
        borrower = new Borrower();
        borrower.setName(name);
        borrower.setBorrowerNumber("BR123456");
        borrower.setCategory(BorrowerCategory.STUDENT);
        borrowerService.registerBorrower(borrower);
    }

    @When("^the system attempts to register {string} with a unique borrower number and his details$")
    public void
theSystemAttemptsToRegisterGeorgeRedWithAUniqueBorrowerNumberAndHisDetails(String
name) {
        try {
            borrower = new Borrower();
            borrower.setName(name);
            borrower.setBorrowerNumber("BR123456"); // Should be unique but using the same for
simulation
            borrower.setCategory(BorrowerCategory.STUDENT);
            borrowerService.registerBorrower(borrower);
            operationResult = "Success";
        } catch (Exception e) {
            operationResult = e.getMessage();
        }
    }

    @Then("^the system informs that the user already exists$")
    public void theSystemInformsThatTheUserAlreadyExists() {
        assertEquals("Borrower already exists.", operationResult);
    }

    @When("^{String} updates his borrowing details$")
```

```
public void georgeRedUpdatesHisBorrowingDetails(String name) {
    try {
        borrower.setContactInfo("New Contact Info");
        borrowerService.updateBorrower(borrower);
        operationResult = "Updated";
    } catch (Exception e) {
        operationResult = e.getMessage();
    }
}

@Then("^the system saves the changes$")
public void theSystemSavesTheChanges() {
    assertEquals("Updated", operationResult);
}

@When("^{String} tries to update his borrowing details$")
public void georgeRedTriesToUpdateHisBorrowingDetails(String name) {
    try {
        borrower = null; // Simulating non-existence
        borrowerService.updateBorrower(borrower);
        operationResult = "Updated";
    } catch (Exception e) {
        operationResult = e.getMessage();
    }
}

@Then("^the system displays an error message indicating that {string} does not exist$")
public void theSystemDisplaysAnErrorMessageIndicatingThatGeorgeRedDoesNotExist(String name) {
    assertEquals("Borrower does not exist.", operationResult);
}

@When("^the system deletes {string}'s account$")
public void theSystemDeletesGeorgeRedsAccount(String name) {
    try {
        borrowerService.deleteBorrower(borrower.getBorrowerNumber());
        operationResult = "Deleted";
    } catch (Exception e) {
        operationResult = e.getMessage();
    }
}

@Then("^the system removes {string}'s details$")
public void theSystemRemovesGeorgeRedsDetails(String name) {
    assertEquals("Deleted", operationResult);
}


@Then("^the system informs about the pending items$")
public void theSystemInformsAboutThePendingItems() {
    assertEquals("Borrower has pending items.", operationResult);
}
}
```

Key Points

- This setup assumes that the `BorrowerService` can throw exceptions when things go wrong, which are then caught in the step definitions to set the `operationResult` which is checked in the assertions.

- Replace `"BorrowerDAOImpl"` with your actual DAO implementation.

- This is a basic implementation to get you started. Depending on your system's complexity and requirements, you might need to refine error handling and the management of borrower registration, updates, and deletions.

With these step definitions, you should be able to run Cucumber tests that validate the borrower management feature of your library system.

User

You are missing some step definitions, create them

ChatGPT

Certainly! Let’s add and refine the missing step definitions to complete the implementation for the scenarios you've described, especially covering scenarios that handle error conditions and those regarding deletion when a borrower has pending loans.

Additional Step Definitions

Here are the step definitions that address scenarios involving errors and pending loan items, which were not covered or need further elaboration:

```java
import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import static org.junit.Assert.*;

public class BorrowerSteps {

    private BorrowerService borrowerService;
    private Borrower borrower;
    private String operationResult;
    private List<Loan> pendingLoans;

    public BorrowerSteps() {
        BorrowerDAO borrowerDAO = new BorrowerDAOImpl(); // Assuming you have an implementation
        borrowerService = new BorrowerService(borrowerDAO);
    }

    // Definitions already provided earlier should be assumed to be present.

    @Given("^{String} has pending items$")
    public void georgeRedHasPendingItems(String name) {
        // Simulating pending loan items
        pendingLoans = new ArrayList<>();
        Loan loan = new Loan();
        loan.setBorrower(borrower);
        Item item = new Item();
        item.setState(ItemState.CHECKED_OUT);
        loan.setItem(item);
        pendingLoans.add(loan);
        when(borrowerDAO.findLoansByBorrower(borrower)).thenReturn(pendingLoans);
    }
    @When("^the system attempts to delete {string}'s account$")
    public void theSystemAttemptsToDeleteGeorgeRedsAccount(String name) {
        try {
            borrowerService.deleteBorrower(borrower.getBorrowerNumber());
            operationResult = "Deleted";
        } catch (Exception e) {
            operationResult = e.getMessage();
        }
    }

    @Then("^the system does not remove {string}'s details$")
    public void theSystemDoesNotRemoveGeorgeRedsDetails(String name) {
        assertEquals("Borrower has pending items.", operationResult);
    }

    @Then("^the system informs that the borrower does not exist$")
    public void theSystemInformsThatTheBorrowerDoesNotExist() {
        assertEquals("Borrower does not exist.", operationResult);
    }
}
```

Explanation of New Step Definitions

1. **Handling Pending Items**:

   - The `georgeRedHasPendingItems` step simulates a scenario where the borrower has pending loans that need to be returned before the account can be deleted. This is done by setting up a list of loans that are assumed to be pending.

   - The deletion attempt in this context will then be handled by the `borrowerService` which should ideally check for pending items before proceeding with the deletion.

2. **Improved Error Handling**:

   - The step definition for trying to delete an account when the borrower isn't registered (`theSystemAttemptsToDeleteGeorgeRedsAccount` and `theSystemInformsThatTheBorrowerDoesNotExist`) ensures that your system handles such cases gracefully by checking if the borrower exists in the system before attempting any operations.

These steps are designed to interact with a mock or actual implementation of `BorrowerDAO` which needs to be capable of handling database interactions such as finding and deleting borrowers, and checking for pending loans. Make sure to use proper mocking libraries like Mockito for unit testing these steps to simulate database interactions.

This setup now covers all the scenarios provided, ensuring comprehensive testing for your library management system's borrower handling features. If you need further assistance or adjustments based on your specific architecture or business rules, feel free to ask!

User

The second feature is :

Feature: Delayed Return Notification

User Story

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

Do the same as above

ChatGPT

For the feature concerning the delayed return notification system, we will need to create a service that can check for overdue items and send email notifications accordingly. First, let's define the domain models, services, and step definitions needed to automate this feature using Cucumber.

Service Class for Notifications

Before jumping into the step definitions, let's define a service that will handle the logic for detecting overdue loans and sending notifications:

```java
public class NotificationService {
    private LoanService loanService;
    private EmailService emailService;

    public NotificationService(LoanService loanService, EmailService emailService) {
        this.loanService = loanService;
        this.emailService = emailService;
    }

    public void notifyOverdueItems() {
        List<Loan> overdueLoans = loanService.findOverdueLoans();
        for (Loan loan : overdueLoans) {
            if (loan.getBorrower().getContactInfo() != null && !loan.getBorrower().getContactInfo().isEmpty()) {
                emailService.sendEmail(loan.getBorrower().getContactInfo(),
                                       "Overdue Notification",
                                       "Dear " + loan.getBorrower().getName() + ", " +
                                       "The item '" + loan.getItem().getBook().getTitle() + "' is overdue. Please return it as soon as possible.");
            }
        }
    }
}
```

Email Service

Here is a simple email service interface and its implementation to handle sending emails:

```java
public interface EmailService {
    void sendEmail(String to, String subject, String content);
}

public class MockEmailService implements EmailService {
    @Override
    public void sendEmail(String to, String subject, String content) {
        System.out.println("Sending email to: " + to);
        System.out.println("Subject: " + subject);
        System.out.println("Content: " + content);
    }
}
```

Step Definitions for Delayed Return Notification

Let's now create the step definitions that will simulate the process of notifying borrowers about overdue items.

```java
import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import static org.junit.Assert.*;


public class NotificationSteps {


    private NotificationService notificationService;
    private Borrower borrower;
    private Item item;
    private Loan loan;
    private boolean emailSent;


    public NotificationSteps() {
        LoanService loanService = new MockLoanService(); // Assume this is an implementation
        EmailService emailService = new MockEmailService();
        notificationService = new NotificationService(loanService, emailService);
    }


    @Given("^{String} has borrowed the item {string}$")
    public void georgeRedHasBorrowedTheItem(String borrowerName, String bookTitle) {
        borrower = new Borrower();
        borrower.setName(borrowerName);
        borrower.setContactInfo("george.red@example.com");
        item = new Item();
        item.setBook(new Book(bookTitle));

        loan = new Loan();
        loan.setBorrower(borrower);
        loan.setItem(item);
        loan.setDueDate(LocalDate.now().minusDays(1)); // Due date has passed
    }

    @Given("^{String}'s due date has passed$")
    public void harryPottersDueDateHasPassed(String bookTitle) {
        // This is implicitly set in the previous step.
    }

    @Given("^{String} has an email address$")
    public void georgeRedHasAnEmailAddress(String name) {
        borrower.setContactInfo("george.red@example.com");
    }

    @Given("^{String} does not have an email address$")
    public void georgeRedDoesNotHaveAnEmailAddress(String name) {
        borrower.setContactInfo(null);
    }

    @When("^the system executes the delayed return notification process$")
    public void theSystemExecutesTheDelayedReturnNotificationProcess() {
        notificationService.notifyOverdueItems();
        emailSent = MockEmailService.wasEmailSentTo(borrower.getContactInfo()); // Assume we can track this
    }

    @Then("^{String} receives an email notification for the return of the item$")
    public void georgeRedReceivesAnEmailNotificationForTheReturnOfTheItem(String name) {
        assertTrue(emailSent);
    }

    @Then("^{String} does not receive an email notification for the return of the item$")
    public void georgeRedDoesNotReceiveAnEmailNotificationForTheReturnOfTheItem(String name) {
        assertFalse(emailSent);
    }
}
```

Explanation

1. **Service and Domain Setup**: Before creating the steps, you need to have the `LoanService` and `EmailService` properly defined and mocked to simulate the checking of overdue books and sending emails.

2. **Step Definitions**: The steps simulate the condition where an item is borrowed and overdue. Depending on the borrower’s contact information, the system will or will not send a notification.

3. **Testing Strategy**: These steps assume that the `NotificationService` uses both the `LoanService` to retrieve overdue loans and the `EmailService` to send emails, which are mocked for the purpose of testing.

This setup should allow you to automate the testing of your notification system based on the scenarios provided. Adjust the mocking and service logic to fit your exact application architecture and requirements.

User

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

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

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

Do the same for this one

ChatGPT

To implement the step definitions for the "Loaning items" feature described in your scenarios, we need to handle various conditions related to loan limits, item availability, and borrower eligibility. Below, we'll define step definitions based on the provided scenarios.

Prerequisites for Implementation

For these scenarios, you will require a service layer to manage loans (`LoanService`), check item states (`ItemService`), and validate borrower status (`BorrowerService`). We will assume basic methods for these services are available, such as checking item availability, updating loan records, and verifying borrower limits.

Step Definitions for Loaning Items

Here's how you could set up the step definitions for the "Loaning items" feature:

```java
import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

import static org.junit.Assert.*;
```

```
public class LoanSteps {


    private LoanService loanService;

    private ItemService itemService;

    private BorrowerService borrowerService;

    private Item itemHarryPotter, itemMobyDick;

    private Borrower borrower;

    private boolean loanResult;

    private String loanError;


    public LoanSteps() {

        loanService = new LoanService();

        itemService = new ItemService();

        borrowerService = new BorrowerService();

    }


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

    public void theLibraryHasTheItemAvailable(String itemName) {

        Item item = new Item();

        item.setTitle(itemName);

        item.setState(ItemState.AVAILABLE);

        if (itemName.equals("Harry Potter")) {
```

```
        itemHarryPotter = item;

    } else {

        itemMobyDick = item;

    }

}


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

public void georgeRedIsARegisteredBorrower() {

    borrower = new Borrower();

    borrower.setName("George Red");

    borrower.setContactInfo("george.red@example.com");

}


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

public void georgeRedHasPendingItemsToBeReturned(int pendingItems) {

    borrower.setPendingItems(pendingItems);

}


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

public void georgeRedHasBeenAssignedAMaximumLendingLimitOf(int limit) {

    borrower.setLendingLimit(limit);

}
```

```
    @When("^George Red borrows the item (Harry Potter|Moby Dick)$")
    public void georgeRedBorrowsTheItem(String itemName) {
        Item item = itemName.equals("Harry Potter") ? itemHarryPotter : itemMobyDick;
        loanResult = loanService.loanItemToBorrower(item, borrower);
    }

    @When("^George Red tries to borrow both items$")
    public void georgeRedTriesToBorrowBothItems() {
        loanResult = loanService.loanItemToBorrower(itemHarryPotter, borrower);
        if (loanResult) {
            loanResult = loanService.loanItemToBorrower(itemMobyDick, borrower);
        }
    }

    @Then("^the system successfully loans the item (Harry Potter|Moby Dick) to George Red with a due date set$")
    public void theSystemSuccessfullyLoansTheItemToGeorgeRedWithADueDateSet(String itemName) {
        assertTrue(loanResult);
        assertNotNull(borrowerService.getLoanDueDate(borrower, itemName));
    }
    @Then("^the system does not loan Moby Dick to George Red due to the lending limit reached$")
    public void theSystemDoesNotLoanMobyDickToGeorgeRedDueToTheLendingLimitReached() {
        assertFalse(loanResult);
    }

    @Then("^George Red's pending items increase to (\\d+)$")
    public void georgeRedsPendingItemsIncreaseTo(int expectedItems) {
        assertEquals(expectedItems, borrower.getPendingItems());
    }

    @Then("^the system returns an error due to the item's status$")
    public void theSystemReturnsAnErrorDueToTheItemsStatus() {
        assertEquals("Item not available", loanError);
    }

    @Then("^the system doesn't allow the loan$")
    public void theSystemDoesntAllowTheLoan() {
        assertFalse(loanResult);
    }
    @Then("^the system withdraws the item (Harry Potter|Moby Dick)$")
    public void theSystemWithdrawsTheItem(String itemName) {
        Item item = itemName.equals("Harry Potter") ? itemHarryPotter : itemMobyDick;
        assertEquals(ItemState.WITHDRAWN, itemService.getItemState(item));
    }
}
```

Explanation

1. **Multiple Scenarios**: Each step definition handles different conditions such as item availability, borrower eligibility, and lending limits.

2. **Method Stubs and Mocking**: This example assumes certain methods like `loanItemToBorrower`, `getItemState`, and `getLoanDueDate` are part of the services. You need to implement these methods in your service classes.

3. **Conditional Logic in Steps**: The steps consider conditions under which an item can be loaned or not, based on borrower limits and item availability.

You'll need to adapt this template to fit the actual implementation of your item, loan, and borrower services. Ensure you properly mock or implement these services for unit testing to simulate the real interactions as closely as possible.

User

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

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

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

Do the same for this as above

ChatGPT

For the feature regarding the return of borrowed items to the library, we need to handle the return process with conditions such as on-time returns, late returns, fines, and updates to the item's state and borrower's record. Below, we'll define step definitions based on the provided scenarios.

Prerequisites for Implementation

For these scenarios, you will require service layers to manage loans (`LoanService`), check and update item states (`ItemService`), and handle borrower records and fines (`BorrowerService`). We will assume these services include methods to process returns, calculate fines, and update loan and item statuses.

Step Definitions for Returning Items

Here's how you could set up the step definitions for the "Returning a borrowed item" feature:

~~~
```java
import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import static org.junit.Assert.*;

public class ReturnSteps {

    private LoanService loanService;
    private ItemService itemService;
    private BorrowerService borrowerService;
    private Item item;
    private Borrower borrower;
    private Loan loan;

    public ReturnSteps() {
        loanService = new LoanService();
        itemService = new ItemService();
        borrowerService = new BorrowerService();
    }
~~~

```
    @Given("^George Red borrowed the item (\\w+) (\\d+) days prior to today's date$")
    public void georgeRedBorrowedTheItemDaysPriorToTodaysDate(String itemName, int daysAgo) {
        borrower = new Borrower();
        borrower.setName("George Red");
        borrower.setBorrowerCategory(BorrowerCategory.STUDENT); // Assuming category for fine calculation

        item = new Item();
        item.setTitle(itemName);
        item.setState(ItemState.CHECKED_OUT);

        loan = new Loan();
        loan.setItem(item);
        loan.setBorrower(borrower);
        loan.setLoanDate(LocalDate.now().minusDays(daysAgo));
        loanService.addLoan(loan);
    }

    @Given("^George Red has been assigned maximum lending days of (\\d+)$")
    public void georgeRedHasBeenAssignedMaximumLendingDaysOf(int maxDays) {
        borrower.setMaxLendingDays(maxDays);
```

```
}


@When("^the return of (\\w+) is processed$")

public void theReturnOfIsProcessed(String itemName) {

    loanService.processReturn(loan);

}


@Then("^the system marks the state of (\\w+) as AVAILABLE$")

public void theSystemMarksTheStateOfAsAvailable(String itemName) {

    assertEquals(ItemState.AVAILABLE, itemService.getItemState(item));

}


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

public void georgeRedHasOneLessPendingItem() {

    int expectedPendingItems = borrower.getPendingItems() - 1;

    assertEquals(expectedPendingItems, borrowerService.getPendingItems(borrower));

}


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

public void georgeRedDoesNotPayAFine() {

    assertEquals(0, borrowerService.getFineAmount(borrower));

}
```

```
    @Then("^George Red pays a fine based on the borrower category$")
    public void georgeRedPaysAFineBasedOnTheBorrowerCategory() {
        assertTrue(borrowerService.getFineAmount(borrower) > 0);
    }

    @Then("^the return date of the loan is set to today's date$")
    public void theReturnDateOfTheLoanIsSetToTodaysDate() {
        assertEquals(LocalDate.now(), loan.getReturnDate());
    }
}
```

Explanation

1. **Setup and Mocking**: These step definitions assume the presence of `loanService`, `itemService`, and `borrowerService` with methods capable of managing the complex conditions related to the return of items. The steps simulate the borrowing and return process, including handling late returns and fines.

2. **Methods and Calculations**: The example assumes methods like `addLoan`, `processReturn`, `getItemState`, `getPendingItems`, and `getFineAmount` are implemented in the respective service classes. These methods need to handle the logic as described in the scenarios, such as calculating fines based on the number of days overdue and borrower category.

3. **Handling Return Logic**: The steps handle both on-time and late returns by checking the item's state after processing the return, updating the borrower's pending items, and setting or not setting a fine as appropriate.

This template should enable you to automate testing for item returns in your library system. You might need to adjust the service method implementations and mocking strategy based on your application's architecture and business rules.