Στη συνομιλία αυτή, παρέθεσα στο LLM όλα τα features του συστήματος ένα προς ένα, μαζί με τα ονόματα, τις ιδιότητες και τις μεθόδους των κλάσεων τομέα, καθώς και την αρχιτεκτονική και τις γενικές γνώσεις του συστήματος. Το LLM κατανόησε εύκολα την εργασία και χρειάστηκε πολύ λίγα μηνύματα για την παραγωγή των αποτελεσμάτων. Παρά την καλή κατανόηση, τα αποτελέσματα ήταν πιο αδύναμα σε σύγκριση με άλλες συνομιλίες.

# Phase 4 Chat 3

### 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 on at a time and the code of all the domain classes with their attributes and methods that you need, 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

Great! It sounds like you have a well-structured approach to implementing your library system application using behavior-driven development with Gherkin scenarios. I'll help you with the step definitions for your features in Java.

To get started, I'll need a bit more information:

1. **Gherkin Scenarios**: Please provide the specific Gherkin scenarios you've written for the features of the library system.

2. **Domain Classes**: Details about your domain classes, including attributes and methods, will be essential to understand how to interact with the underlying data.

3. **Architecture Details**: A brief overview of the architecture, specifically how the Data Access Objects (DAOs) and Service classes are set up, would be helpful. This includes any specific frameworks or libraries you are using (like Spring, Hibernate, etc.).

4. **Testing Frameworks**: Let me know which frameworks you're using for testing (e.g., Cucumber-JVM, JUnit).

Once I have this information, I can assist you more effectively by writing Java code for the step definitions that match your scenarios and fit into your architectural design.

**User**

**The domain classes are:**

**1)class "Book" with the attributes:**

**private ISBN isbn,**

**private String title,**

**private String publication,**

**private int publicationyear,**

**private Publisher publisher,**

**private Set<Item> items = new HashSet<Item>(),**

**private Set<Author> authors = new HashSet<Author>(),**

**and with the methods:**

**Setters and Getters for every attribute and a constructor,**

**public Set<Item> getItems(),**

**public void addItem(Item item),**

**public void removeItem(Item item),**

**Set<Item> friendItems(),**

**public void addAuthor(Author author),**

**public void removeAuthor(Author author),**

**Set<Author> friendAuthors()**

**2)class "Borrower" with the attributes:**

**private int borrowerNo,**

**private TelephoneNumber telephone,**

**private EmailAddress emailAddress,**

**private Address address,**

**private BorrowerCategory category,**

**private Person person = new Person(),**

**private Set<Loan> loans = new HashSet<Loan>(),**

**and with the methods:**

**Setters and Getters for every attribute and a constructor,**

**public Set<Loan> getLoans(),**

**Set<Loan> friendLoans(),**

**public int countPendingItems(),**

**public boolean canBorrow(),**

**public LocalDate getLoanDue(LocalDate loanDate),**

**public Money getDailyFine()**

**3)class "BorrowerCategory" with the attributes:**

**private String description,**

**private int maxLendingItems,**

**private int maxLendingDays,**

**private Money dailyFine,**

**and with the methods:**

**Setters and Getters for every attribute and a constructor,**

**public LocalDate getLoanDue(LocalDate loanDate),**

**public boolean canBorrow(int pendingItems)**

**4)class "Item" with the attributes:**

**private int itemNumber = 0,**

**private Book book,**

**private ItemState state = ItemState.NEW**

**and with the methods:**

**Setters and Getters for every attribute and a constructor,**

**public Loan borrow(Borrower borrower),**

**public void available(),**

**public void withdraw(),**

**public void lost(),**

**5)enum class "ItemState" with the Enum values :**

**NEW,AVAILABLE,LOANED,LOST,WITHDRAWN**

**6)class "Loan" with the attributes:**

**private LocalDate loanDate = SystemDate.now(),**

**private LocalDate returnDate,**

**private Borrower borrower,**

**private Item item**

**and with the methods:**

**Setters and Getters for every attribute and a constructor,**

**public LocalDate getDue(),**

**public boolean isPending(),**

**public void returnItem(),**

**public boolean isOverdue(),**

**public long daysToDue(),**

**public long getOverdue(),**

**public Money getFine(),**

**7)class "Person" with the attributes:**

**private String firstName,**

**private String lastName,**

**and with the methods:**

**Setters and Getters for every attribute and a constructor**

**In the next message, I will provide you with the first feature of the system**

ChatGPT

Thanks for providing the detailed descriptions of your domain classes. This information will be crucial for implementing the step definitions for your Gherkin scenarios effectively, as it allows me to understand the relationships and functionalities within your library system.

Before proceeding to the first feature you'll share, here are a few things we need to consider:

1. **Data Access Objects (DAO)**: We might need DAO classes for accessing and manipulating the data related to each entity (like `Book`, `Borrower`, etc.). This could include methods like `saveBook`, `findBookByISBN`, `deleteBook`, etc. If you're using an ORM like Hibernate, this would typically involve using repositories or DAO implementations.

2. **Service Layer**: The service classes will handle the business logic of the application. They would use the DAOs to interact with the database and provide services to the higher layers (like your step definitions). Examples could include `BookService`, `BorrowerService`, etc., with methods like `checkOutBook`, `returnBook`, `registerBorrower`, etc.

3. **Testing and Framework Setup**: Ensuring that we have the testing setup with Cucumber-JVM integrated properly with your project. This would include the Cucumber setup in your `pom.xml` or `build.gradle` file and the configuration classes.

4. **Step Definitions Context**: We'll need to set up a context or use dependency injection (e.g., with Spring) to share state between steps, like references to specific books, borrowers, or loans created or manipulated during the scenario.

Please provide the first feature and any specific scenario you want to start with, and we can begin crafting the step definitions accordingly. If there are any specific technical or architectural preferences (like database handling strategies or transaction management), let me know!

**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**

ChatGPT

Thanks for sharing the detailed scenarios for the "Borrower handling by the system" feature. To implement these scenarios as step definitions in Java using Cucumber, you'll need to ensure your testing environment is set up properly with Cucumber and possibly Spring for dependency injection.

Given your architecture, which includes domain classes, DAOs, and services, let's plan out the step definitions. We'll integrate these steps with a service layer that handles business logic, while DAOs will interact with the database for CRUD operations.

Here's a proposed implementation outline for the step definitions corresponding to the first scenario: "Registering a new borrower".

### Example Java Step Definitions for "Registering a new borrower"

#### 1. Setup

Firstly, ensure you have Cucumber set up in your Java project, and configure your `pom.xml` or `build.gradle` accordingly with dependencies like Cucumber-Java, Cucumber-Spring, JUnit, etc.

#### 2. Implement Step Definitions

We'll create a `BorrowerSteps` class to encapsulate the step definitions related to borrower scenarios.

```java
package steps;

import cucumber.api.java.en.Given;
import cucumber.api.java.en.When;
import cucumber.api.java.en.Then;
import org.springframework.beans.factory.annotation.Autowired;
import services.BorrowerService;
import domain.Borrower;
import domain.Person;
import exception.BorrowerAlreadyExistsException;
```

```
public class BorrowerSteps {

    @Autowired
    private BorrowerService borrowerService;

    private Borrower currentBorrower;

    @Given("^George Red is not registered as a borrower$")
    public void george_red_is_not_registered_as_a_borrower() {
        // Ensure George Red is not in the system
        currentBorrower = borrowerService.findBorrowerByName("George", "Red");
        assert currentBorrower == null;
    }

    @When("^George Red gets registered in the system with a unique borrower number and his details$")
    public void george_red_gets_registered_in_the_system() {
        // Create new Borrower details
        Person person = new Person("George", "Red");
        currentBorrower = new Borrower();
        currentBorrower.setPerson(person);
        currentBorrower = borrowerService.registerBorrower(currentBorrower);
    }


    @Then("^the system successfully stores the borrower's details$")
    public void system_successfully_stores_the_borrowers_details() {
        // Check if the borrower details are stored correctly
        assert currentBorrower != null && currentBorrower.getPerson().getFirstName().equals("George");
    }
}
```

### Explanation

- **@Autowired**: This uses Spring's dependency injection to inject the `BorrowerService` which will handle all borrower-related business logic.

- **Step Definitions**: These directly map to the scenarios you've defined in Gherkin. We're assuming operations like `findBorrowerByName` and `registerBorrower` are defined within the `BorrowerService`.

### Next Steps

- Implement DAOs for accessing borrower data from the database.

- Create the actual implementations for methods like `findBorrowerByName` and `registerBorrower` in the `BorrowerService`.

- Repeat a similar pattern for other scenarios, handling each specific case (like handling already registered borrowers, updates, deletions, etc.).

If you need further details on any of these aspects or additional scenarios, let me know! We can also work through the implementations for error handling and edge cases based on your scenarios.

**User**

**Give me all the step definitions of this feature**

ChatGPT

Sure, let's craft step definitions for each of the scenarios you've described in your feature. Below are Java step definitions for handling borrower registration, updates, deletions, and error scenarios.

### Java Step Definitions for "Borrower handling by the system"

#### 1. Dependencies and Setup

Make sure your environment is ready with Cucumber, Spring (for dependency injection), and your DAO and Service layers set up as described previously.

#### 2. Step Definitions Class

We will use a class to manage all step definitions related to borrowers.

```java
```

```java
package steps;


import cucumber.api.java.en.Given;

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

import org.springframework.beans.factory.annotation.Autowired;

import services.BorrowerService;

import domain.Borrower;

import domain.Person;

import exception.BorrowerAlreadyExistsException;


public class BorrowerManagementSteps {


    @Autowired

    private BorrowerService borrowerService;


    private Borrower currentBorrower;

    private String operationResult;


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

    public void george_red_is_not_registered_as_a_borrower() {

        currentBorrower = borrowerService.findBorrowerByName("George", "Red");
        assert currentBorrower == null;
    }


    @Given("^George Red is registered as a borrower$")
    public void george_red_is_registered_as_a_borrower() {
        currentBorrower = borrowerService.findOrCreateBorrower("George", "Red");
    }


    @When("^George Red gets registered in the system with a unique borrower number and his
details$")
    public void george_red_gets_registered() {
        try {
            Person person = new Person("George", "Red");
            currentBorrower = new Borrower();
            currentBorrower.setPerson(person);
            borrowerService.registerBorrower(currentBorrower);
            operationResult = "Registered";
        } catch (BorrowerAlreadyExistsException e) {
            operationResult = "Exists";
        }
    }
    @When("^the system attempts to register George Red with a unique borrower number and his
details$")
    public void attempt_registration() {
        george_red_gets_registered(); // Reuse the registration method
    }

    @When("^George Red updates his borrowing details$")
    public void george_red_updates_details() {
        if (currentBorrower != null) {
            currentBorrower.getPerson().setLastName("UpdatedLast");
            borrowerService.updateBorrower(currentBorrower);
            operationResult = "Updated";
        } else {
            operationResult = "NotFound";
        }
    }

    @When("^George Red tries to update his borrowing details$")
    public void george_red_tries_to_update_details() {
        george_red_updates_details(); // Reuse the update details method
    }
```

```
@When("^the system deletes George Red's account$")
public void delete_account() {
    if (currentBorrower != null) {
        borrowerService.deleteBorrower(currentBorrower);
        operationResult = "Deleted";
    } else {
        operationResult = "NotFound";
    }
}

@When("^the system attempts to delete George Red's account$")
public void attempt_delete_account() {
    delete_account(); // Reuse delete account method
}

@Then("^the system successfully stores the borrower's details$")
public void check_stored_details() {
    assert "Registered".equals(operationResult);
}

@Then("^the system informs that the user already exists$")
public void user_already_exists() {
    assert "Exists".equals(operationResult);

}


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

public void save_changes() {

    assert "Updated".equals(operationResult);

}


@Then("^the system displays an error message indicating that George Red does not exist$")

public void display_error_message() {

    assert "NotFound".equals(operationResult);

}


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

public void remove_details() {

    assert "Deleted".equals(operationResult);

}


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

public void borrower_does_not_exist() {

    assert "NotFound".equals(operationResult);

}
```

```
    @Given("^George Red has pending items$")
    public void george_red_has_pending_items() {
        // Assume pending items check is a method in service
        if (!borrowerService.checkForPendingItems(currentBorrower)) {
            throw new RuntimeException("Borrower has no pending items");
        }
    }

    @Then("^the system does not remove George Red's details$")
    public void does_not_remove_details() {
        assert "Deleted".equals(operationResult) == false;
    }

    @Then("^the system informs about the pending items$")
    public void inform_pending_items() {
        assert borrowerService.checkForPendingItems(currentBorrower);
    }
}
```

### Explanation

- **Spring Integration**: Using `@Autowired` to inject the `BorrowerService` which is responsible for the business logic.

- **Reuse of Methods**: Methods like `george_red_gets_registered()` and `delete_account()` are reused for similar steps in different scenarios to ensure DRY (Don't Repeat Yourself) principles.

- **Operation Result Handling**: We use a `operationResult` string to manage the outcome of operations, making assertions based on this result in the `Then` steps.

This setup assumes that you have exception handling like `BorrowerAlreadyExistsException` and methods such as `findOrCreateBorrower`, `registerBorrower`, `updateBorrower`, `deleteBorrower`, and `checkForPendingItems` defined in your service layer. Adjust these assumptions as necessary to fit your application's architecture and requirements.

**User**

**Give me the code of the borrower Service you are using**

ChatGPT

Let's define a `BorrowerService` class in Java that corresponds with the operations we referenced in the step definitions for your library system. This service will handle business logic for managing borrowers, including registering, updating, deleting, and checking for pending items.

### BorrowerService Java Class

This service class will make use of a DAO (Data Access Object) for borrowers to interact with the database. We'll assume that you have a simple DAO implementation and that your system can throw a `BorrowerAlreadyExistsException` when attempting to register a borrower who already exists.

```java
package services;

import dao.BorrowerDao;
import domain.Borrower;
import domain.Person;
import exception.BorrowerAlreadyExistsException;
import java.util.Optional;

public class BorrowerService {

    private final BorrowerDao borrowerDao;

    public BorrowerService(BorrowerDao borrowerDao) {
        this.borrowerDao = borrowerDao;
    }

    public Borrower findBorrowerByName(String firstName, String lastName) {
        return borrowerDao.findByName(firstName, lastName).orElse(null);
    }
```

```
  }

  public Borrower findOrCreateBorrower(String firstName, String lastName) {
    Optional<Borrower> existingBorrower = borrowerDao.findByName(firstName, lastName);
    return existingBorrower.orElseGet(() -> new Borrower(new Person(firstName, lastName)));
  }

  public Borrower registerBorrower(Borrower borrower) throws
BorrowerAlreadyExistsException {
    Optional<Borrower> existingBorrower =
borrowerDao.findByBorrowerNo(borrower.getBorrowerNo());
    if (existingBorrower.isPresent()) {
      throw new BorrowerAlreadyExistsException("Borrower with number: " +
borrower.getBorrowerNo() + " already exists.");
    }
    return borrowerDao.save(borrower);
  }

  public Borrower updateBorrower(Borrower borrower) {
    if (!borrowerDao.exists(borrower.getBorrowerNo())) {
      throw new RuntimeException("Borrower not found");
    }
    return borrowerDao.update(borrower);
  }
```

```
    public void deleteBorrower(Borrower borrower) {
        if (!borrowerDao.exists(borrower.getBorrowerNo())) {
            throw new RuntimeException("Borrower not found");
        }
        if (checkForPendingItems(borrower)) {
            throw new RuntimeException("Borrower has pending items and cannot be deleted");
        }
        borrowerDao.delete(borrower.getBorrowerNo());
    }

    public boolean checkForPendingItems(Borrower borrower) {
        return borrower.getLoans().stream()
                      .anyMatch(loan -> loan.isPending());
    }
}
```

### DAO Interface Example

Below is a simple interface for the `BorrowerDao` that your service would depend on.

```java
```

```
package dao;


import domain.Borrower;
import java.util.Optional;


public interface BorrowerDao {
    Optional<Borrower> findByName(String firstName, String lastName);
    Optional<Borrower> findByBorrowerNo(int borrowerNo);
    Borrower save(Borrower borrower);
    Borrower update(Borrower borrower);
    void delete(int borrowerNo);
    boolean exists(int borrowerNo);
}
```

### Explanation

- **findBorrowerByName and findOrCreateBorrower**: These methods handle finding borrowers by name, optionally creating a new one if not found.

- **registerBorrower**: Checks for an existing borrower and throws an exception if one is found, otherwise saves the new borrower.

- **updateBorrower and deleteBorrower**: Perform update and delete operations, including checks to ensure the borrower exists and in the case of delete, that they have no pending loans.

- **checkForPendingItems**: Checks if the borrower has any loans that are still pending, which would prevent account deletion.

This service class and DAO interface need to be tailored to fit your specific database implementation, error handling policies, and overall application architecture. Adjust the methods, exceptions, and integration points as necessary to match your system's requirements.

**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**

ChatGPT

To implement the step definitions for the "Delayed Return Notification" feature in your library system, we will need a mechanism to simulate email notifications. For this example, we'll assume there is a service, possibly named `NotificationService`, that handles the sending of emails. We also assume that the loan information includes an `Item` object, a due date, and a borrower's contact information, which is linked to a `Borrower` object.

### Step Definitions for Delayed Return Notification

#### NotificationService and LoanService Mock-up

First, here's a basic mock-up of the `NotificationService` that could be used:

```java
```

```
package services;

public class NotificationService {
    public void sendEmail(String emailAddress, String message) {
        System.out.println("Email sent to: " + emailAddress + " with message: " + message);
    }
}
```

Let's define the `LoanService` to manage loan-specific operations:

```java
package services;

import domain.Loan;

public class LoanService {
    public boolean isLoanOverdue(Loan loan) {
        return loan.getDue().isBefore(LocalDate.now());
    }
}
```

#### Step Definitions Class

Now, let's define the Cucumber step definitions for the given scenarios.

```java
package steps;

import cucumber.api.java.en.Given;
import cucumber.api.java.en.When;
import cucumber.api.java.en.Then;
import org.springframework.beans.factory.annotation.Autowired;
import services.LoanService;
import services.NotificationService;
import domain.Borrower;
import domain.Item;
import domain.Loan;

public class NotificationSteps {

    @Autowired
    private LoanService loanService;
    @Autowired
```

```
private NotificationService notificationService;

private Borrower borrower;
private Loan loan;
private boolean emailSent = false;

@Given("^George Red has borrowed the item Harry Potter$")
public void george_red_has_borrowed_item() {
    borrower = new Borrower(); // Mock borrower
    borrower.setEmailAddress("george.red@example.com");

    Item item = new Item(); // Mock item
    item.setTitle("Harry Potter");

    loan = new Loan(); // Mock loan
    loan.setBorrower(borrower);
    loan.setItem(item);
}

@Given("^Harry Potter's due date has passed$")
public void item_due_date_has_passed() {
    loan.setDue(LocalDate.now().minusDays(2)); // Setting due date 2 days ago
}

@Given("^George Red has an email address$")
public void george_red_has_an_email_address() {
    borrower.setEmailAddress("george.red@example.com");
}

@Given("^George Red does not have an email address$")
public void george_red_does_not_have_an_email_address() {
    borrower.setEmailAddress(null);
}

@When("^the system executes the delayed return notification process$")
public void execute_delayed_return_notification_process() {
    if (loanService.isLoanOverdue(loan) && borrower.getEmailAddress() != null) {
        notificationService.sendEmail(borrower.getEmailAddress(), "Your loan for '" + loan.getItem().getTitle() + "' is overdue. Please return it as soon as possible.");
        emailSent = true;
    }
}

@Then("^George Red receives an email notification for the return of the item$")
public void george_red_receives_an_email_notification() {
    assert emailSent;
}

@Then("^George Red does not receive an email notification for the return of the item$")
public void george_red_does_not_receive_an_email_notification() {
    assert !emailSent;
}
}
```

### Explanation

- **NotificationService**: This service is responsible for sending emails. In a real system, you would replace the print statement with actual email sending logic, potentially using an SMTP server or an email API like SendGrid.

- **LoanService**: Contains business logic to determine if a loan is overdue.

- **Step Definitions**: These map directly to the scenarios provided. We check if the loan is overdue and if the borrower has an email address, then we send an email and set a flag `emailSent` to verify in assertions.

This setup requires the `Borrower`, `Loan`, and `Item` classes to be defined appropriately with methods to get and set properties as used here. Adjust these details according to your domain model specifics and infrastructure.

**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: 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

For the "Loaning items" feature, you have described scenarios covering successful and unsuccessful loaning based on different conditions like item availability, borrower eligibility, and lending limits. Let's implement the step definitions for these scenarios. We will use a hypothetical `LoanService` that integrates with your existing `Item` and `Borrower` entities to handle these operations.

### Step Definitions for "Loaning Items" Feature

Here, we'll create the necessary step definitions in Java using Cucumber to execute these scenarios.

#### Mockup Services and Entities

Assuming we have `ItemService` and `BorrowerService` along with the main `LoanService`:

```java
```

```
package services;


import domain.Borrower;

import domain.Item;

import domain.Loan;

import java.time.LocalDate;


public class LoanService {

    public Loan loanItem(Borrower borrower, Item item) {

        if (!item.isAvailable() || !borrower.canBorrow() || borrower.getLoans().size() >=
borrower.getCategory().getMaxLendingItems()) {

            throw new RuntimeException("Cannot loan item due to constraints");

        }

        Loan loan = new
Loan(LocalDate.now().plusDays(borrower.getCategory().getMaxLendingDays()), borrower, item);

        item.setLoaned();

        borrower.addLoan(loan);

        return loan;

    }


    public boolean canBorrowMoreItems(Borrower borrower) {

        return borrower.getLoans().size() < borrower.getCategory().getMaxLendingItems();
    }
}
```

### Cucumber Step Definitions

Here's how we might define the step definitions for these scenarios in Java:

```java
package steps;

import cucumber.api.java.en.Given;
import cucumber.api.java.en.When;
import cucumber.api.java.en.Then;
import org.springframework.beans.factory.annotation.Autowired;
import domain.Borrower;
import domain.Item;
import domain.Loan;
import services.ItemService;
import services.LoanService;

public class LoanSteps {
}



@Autowired

private LoanService loanService;

@Autowired

private ItemService itemService;



private Borrower borrower;

private Item itemHarryPotter, itemMobyDick;

private Loan loan;

private boolean loanSuccessful;



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

public void the_library_has_harry_potter_available() {

   itemHarryPotter = itemService.findItem("Harry Potter");

   itemHarryPotter.setAvailable(true);

}



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

public void george_red_is_a_registered_borrower() {

   borrower = new Borrower();

   borrower.register("George", "Red");

}
```

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

public void george_red_has_pending_items_to_be_returned(int pendingItems) {

    borrower.setPendingItems(pendingItems);

}


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

public void george_red_has_been_assigned_a_maximum_lending_limit(int limit) {

    borrower.setLendingLimit(limit);

}


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

public void george_red_borrows_harry_potter() {

    try {

        loan = loanService.loanItem(borrower, itemHarryPotter);

        loanSuccessful = true;

    } catch (Exception e) {

        loanSuccessful = false;

    }

}
```

```
    @Then("^the system successfully loans the item Harry Potter to George Red with a due date set$")
    public void system_successfully_loans_harry_potter() {
        assert loanSuccessful && loan.getDueDate() != null;
    }

    @Then("^George Red's pending items increase to (\\d+)$")
    public void george_reds_pending_items_increase(int expectedPending) {
        assert borrower.getPendingItems() == expectedPending;
    }

    @Then("^the system returns an error due to the item's status$")
    public void system_returns_error_due_to_item_status() {
        assert !loanSuccessful;
    }

    @Then("^the system withdraws the item Harry Potter$")
    public void system_withdraws_item_harry_potter() {
        assert !itemHarryPotter.isAvailable();
    }

    @Then("^the system doesn't allow the loan$")
    public void system_does_not_allow_the_loan() {
        assert !loanSuccessful;
    }

    @Then("^George Red's pending items remain (\\d+)$")
    public void george_reds_pending_items_remain(int expectedPending) {
        assert borrower.getPendingItems() == expectedPending;
    }
}
```

### Explanation

- **Service Methods**: The `loanItem` method in `LoanService` encapsulates logic to check availability, borrower eligibility, and item conditions before creating a `Loan`.

- **Step Definitions**: Each step definition matches the Gherkin scenario you provided, handling different conditions like item availability, borrower registration, and lending limits.

Ensure that your services (`ItemService`, `BorrowerService`, and `LoanService`) and domain classes (`Borrower`, `Item`, `Loan`) are properly implemented to support the logic in these step definitions. Adjust the method calls and property checks according to your actual application's API and data models.

**User**

**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**

ChatGPT

For the "Returning a borrowed item to the library" feature, let's write step definitions that handle the scenarios of both successful and late returns. The system needs to update the item's availability, adjust the borrower's pending items, possibly apply fines, and record the return date.

### Step Definitions for Returning Items Feature

We will need a `ReturnService` that handles the logic of returning items and computing fines if necessary. Here's a basic implementation outline for this service:

#### ReturnService Mock-up

```java
package services;

import domain.Item;
import domain.Loan;
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;
```

```
public class ReturnService {
    public void returnItem(Loan loan) {
        Item item = loan.getItem();
        item.setState(ItemState.AVAILABLE);
        loan.setReturnDate(LocalDate.now());
        long daysLate = ChronoUnit.DAYS.between(loan.getDueDate(), LocalDate.now());
        if (daysLate > 0) {
            double fine = daysLate * loan.getBorrower().getCategory().getDailyFine().amount();
            loan.getBorrower().payFine(fine);
        }
    }
}
```

### Cucumber Step Definitions

Now, let's define the Cucumber step definitions for these scenarios in Java:

```java
package steps;
```

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

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;

import org.springframework.beans.factory.annotation.Autowired;

import domain.Borrower;

import domain.Item;

import domain.Loan;

import services.ReturnService;


public class ReturnSteps {


    @Autowired

    private ReturnService returnService;


    private Borrower borrower;

    private Loan loan;

    private Item item;


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

    public void george_red_borrowed_item_days_prior(int daysAgo) {

        borrower = new Borrower();

        item = new Item();
        item.setTitle("Animal Kingdom");

        loan = new Loan();

        loan.setItem(item);

        loan.setBorrower(borrower);

        loan.setLoanDate(LocalDate.now().minusDays(daysAgo));

        borrower.addLoan(loan);

    }


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

    public void george_red_assigned_max_lending_days(int maxDays) {

        borrower.getCategory().setMaxLendingDays(maxDays);

    }


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

    public void return_of_animal_kingdom_processed() {

        returnService.returnItem(loan);

    }


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

    public void item_marked_as_available() {

        assert item.getState() == ItemState.AVAILABLE;

    }
```

```
    @Then("^George Red has one less pending item$")
    public void george_red_has_one_less_pending_item() {
        assert borrower.getPendingItems() == 0; // Assuming this was the only item
    }

    @Then("^George Red does not pay a fine$")
    public void george_red_does_not_pay_a_fine() {
        assert borrower.getTotalFinesPaid() == 0;
    }

    @Then("^George Red pays a fine based on the borrower category$")
    public void george_red_pays_fine() {
        assert borrower.getTotalFinesPaid() > 0;
    }

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

### Explanation

- **ReturnService**: This service handles the logic to mark the item as available again, calculate fines for late returns, and set the return date of the loan.

- **Step Definitions**: These methods correspond to each step in your scenarios. They simulate borrowing and returning the items, adjusting the borrower's pending items count, and verifying fines.

This setup assumes:

- `Borrower`, `Item`, and `Loan` classes have appropriate fields and methods to support operations shown.

- `ItemState` enum and other dependencies are properly defined to support state changes and fine calculations.

This example also simplifies complex operations such as database transactions and real-world error handling for brevity. Adjust the code and logic to match the specific requirements and capabilities of your system.