Σε αυτήν τη συνομιλία, παρείχα στο LLM τα features του συστήματος ένα προς ένα, αντί να τα δώσω όλα μαζί, καθώς και την αρχιτεκτονική. Αυτή η προσέγγιση είχε ως αποτέλεσμα το LLM να θυμάται καλύτερα τις εντολές, καθώς κάθε feature δινόταν ξεχωριστά. Ωστόσο, το LLM δεν είχε γνώση των μεταγενέστερων features στις πρώτες υλοποιήσεις των step definitions. Συνολικά, το LLM κατάλαβε αρκετά καλά την εργασία, αλλά χρειάστηκε και πάλι επαναλαμβανόμενες εντολές για να δημιουργήσει τα πλήρη step definitions για τα features, όπως συνέβη και σε όλες τις άλλες περιπτώσεις.

# Phase 1 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 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

Great, setting up a library system application with Gherkin scenarios sounds like a structured way to approach behavior-driven development (BDD). Using Java along with domain classes, data access objects (DAOs), and service classes is a robust architectural choice. Here's what I'd like to confirm or suggest before we proceed:

1. **Test Framework**: Are you using Cucumber for the Gherkin scenarios in Java? This will help determine the exact syntax and annotations for the step definitions.

2. **Build Tool & Dependencies**: Are you using Maven or Gradle as your build tool? Knowing this can help set up dependencies correctly, especially for libraries like Cucumber, JUnit, and any database connectivity libraries you might be using.

3. **Database Interaction**: What type of database are you planning to use (e.g., MySQL, PostgreSQL, H2)? Additionally, will you be using an ORM (like Hibernate) or just plain JDBC for database interactions?

4. **Domain Model Details**: Can you share more about the domain model? Knowing the entities (like Book, User, Loan, etc.) and their relationships will help in writing more precise DAOs and service classes.

5. **Versioning and Other Tools**: Which version of Java are you using? Are there any specific libraries or frameworks (like Spring Boot, JPA, etc.) that you would like to integrate?

6. **Scope of Each Feature**: As you provide Gherkin scenarios, please specify if you want the step definitions to handle the integration fully (like interacting with the database) or if mock/fake implementations (e.g., using in-memory data structures) are acceptable for the sake of demonstration.

Feel free to share the first feature or scenario whenever you're ready, and I'll help you with the step definitions and any necessary architectural setup!

**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

Based on the scenarios you've provided, we'll start by creating the step definitions for these Gherkin scenarios in a Java project using Cucumber. To make this comprehensive, let's sketch out a basic structure for the domain model, the data access objects (DAO), and the services that we'll be using.

### Domain Model

For this feature, our main entity is a `Borrower`:

```java

public class Borrower {
    private Long id;
    private String name;
    private boolean hasPendingItems;

    // Constructor, getters and setters
}
```

### Data Access Object (DAO)

We'll create an interface for our DAO to handle borrower-related operations:

```java
public interface BorrowerDAO {
    void save(Borrower borrower);
    void update(Borrower borrower);
    void delete(Long borrowerId);
    Borrower findById(Long borrowerId);
    boolean exists(Long borrowerId);
    boolean hasPendingItems(Long borrowerId);
}
```

### Service Layer

We'll encapsulate business logic in a service layer:

```java
public class BorrowerService {
    private BorrowerDAO borrowerDAO;

    public BorrowerService(BorrowerDAO borrowerDAO) {
        this.borrowerDAO = borrowerDAO;
    }

    public void registerBorrower(Borrower borrower) throws Exception {
        if (borrowerDAO.exists(borrower.getId())) {
            throw new Exception("Borrower already exists");
        }
        borrowerDAO.save(borrower);
    }

    public void updateBorrower(Borrower borrower) throws Exception {
        if (!borrowerDAO.exists(borrower.getId())) {
            throw new Exception("Borrower does not exist");
        }
        borrowerDAO.update(borrower);
    }

    public void deleteBorrower(Long borrowerId) throws Exception {
        if (!borrowerDAO.exists(borrowerId)) {
            throw new Exception("Borrower does not exist");
        }
        if (borrowerDAO.hasPendingItems(borrowerId)) {
            throw new Exception("Borrower has pending items");
        }
        borrowerDAO.delete(borrowerId);
    }
}
```

### Step Definitions

Now, let's define the step definitions for Cucumber based on your scenarios:

```java
```

```
}


public class BorrowerSteps {

    private BorrowerService borrowerService;

    private BorrowerDAO borrowerDAO; // This would be mocked in tests

    private Map<String, Borrower> borrowers;

    private Exception lastException;



    public BorrowerSteps() {

        borrowerDAO = new InMemoryBorrowerDAO(); // Or mock

        borrowerService = new BorrowerService(borrowerDAO);

        borrowers = new HashMap<>();

    }



    @Given("{string} is not registered as a borrower")

    public void georgeRedIsNotRegisteredAsABorrower(String name) {

        // Assure that borrower is not in the system

        Long id = name.hashCode().longValue(); // Simplified ID generation

        if (borrowerDAO.exists(id)) {

            borrowerDAO.delete(id);

        }

        borrowers.put(name, new Borrower(id, name, false));

    }
    @When("{string} gets registered in the system with a unique borrower number and his details")
    public                                                                                         void
georgeRedGetsRegisteredInTheSystemWithAUniqueBorrowerNumberAndHisDetails(String
name) {
        try {
            borrowerService.registerBorrower(borrowers.get(name));
        } catch (Exception e) {
            lastException = e;
        }
    }

    @Then("the system successfully stores the borrower's details")
    public void theSystemSuccessfullyStoresTheBorrowersDetails() {
        assert lastException == null;
    }

    @Then("the system informs that the user already exists")
    public void theSystemInformsThatTheUserAlreadyExists() {
        assertNotNull(lastException);
        assertEquals("Borrower already exists", lastException.getMessage());
    }
    // Additional steps based on other scenarios
}
```

### Additional Implementation Notes:

- **DAO Mocking**: In actual implementation, you should mock the `BorrowerDAO` during testing to isolate tests from the database.

- **Error Handling**: Exceptions are used to handle errors like duplicate registrations or updates on non-existent borrowers.

You can expand on these examples to cover all scenarios. Adjust the implementation based on your project requirements, such as the use of a real database or specific frameworks like Spring Boot.

**User**

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

ChatGPT

Certainly! Below are the complete step definitions for all the scenarios described in your feature "Borrower handling by the system." These step definitions assume the use of Cucumber with Java and utilize a service layer with exception handling for business logic enforcement. The DAO is assumed to be either mocked or implemented with an in-memory representation for testing purposes.

### Complete Step Definitions for Borrower Handling Feature

```java
import org.junit.Assert;
import java.util.HashMap;
import java.util.Map;

public class BorrowerSteps {
    private BorrowerService borrowerService;
    private Map<String, Borrower> borrowers;
    private String lastOperationMessage;
    private Exception lastException;

    public BorrowerSteps() {
        BorrowerDAO borrowerDAO = new InMemoryBorrowerDAO(); // This should be mocked in an actual test setup
        borrowerService = new BorrowerService(borrowerDAO);
        borrowers = new HashMap<>();
    }

    @Given("{string} is not registered as a borrower")
    public void givenBorrowerIsNotRegistered(String name) {
        Long id = name.hashCode().longValue(); // Simplified ID generation for example purposes
        borrowers.put(name, new Borrower(id, name, false));
    borrowerService.getBorrowerDAO().delete(id); // Ensure the borrower is not in the system
}

@Given("{string} is registered as a borrower")
public void givenBorrowerIsRegistered(String name) {
    Long id = name.hashCode().longValue();
    Borrower borrower = new Borrower(id, name, false);
    borrowerService.getBorrowerDAO().save(borrower); // Ensure the borrower is in the system
    borrowers.put(name, borrower);
}

@Given("{string} has pending items")
public void givenBorrowerHasPendingItems(String name) {
    Borrower borrower = borrowers.get(name);
    borrower.setHasPendingItems(true);
    borrowerService.getBorrowerDAO().update(borrower);
}

@When("{string} gets registered in the system with a unique borrower number and his details")
public void whenBorrowerGetsRegistered(String name) {
    try {
        borrowerService.registerBorrower(borrowers.get(name));
        lastOperationMessage = "Registered successfully";
    } catch (Exception e) {
        lastException = e;
    }
}

@When("{string} updates his borrowing details")
public void whenBorrowerUpdatesDetails(String name) {
    try {
        borrowerService.updateBorrower(borrowers.get(name));
        lastOperationMessage = "Update successful";
    } catch (Exception e) {
        lastException = e;
    }
}

@When("{string} tries to update his borrowing details")
public void whenBorrowerTriesToUpdateDetails(String name) {
    whenBorrowerUpdatesDetails(name);
}
```

```
    @When("the system attempts to register {string} with a unique borrower number and his details")
    public void whenSystemAttemptsToRegister(String name) {
        whenBorrowerGetsRegistered(name);
    }

    @When("the system deletes {string}'s account")
    public void whenSystemDeletesAccount(String name) {
        try {
            borrowerService.deleteBorrower(borrowers.get(name).getId());
            lastOperationMessage = "Deletion successful";
        } catch (Exception e) {
            lastException = e;
        }
    }

    @When("the system attempts to delete {string}'s account")
    public void whenSystemAttemptsToDeleteAccount(String name) {
        whenSystemDeletesAccount(name);
    }

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

```
public void thenSystemStoresDetails() {

   Assert.assertNull(lastException);

   Assert.assertEquals("Registered successfully", lastOperationMessage);

}


@Then("the system saves the changes")

public void thenSystemSavesChanges() {

   Assert.assertNull(lastException);

   Assert.assertEquals("Update successful", lastOperationMessage);

}


@Then("the system displays an error message indicating that {string} does not exist")

public void thenSystemDisplaysErrorBorrowerDoesNotExist(String name) {

   Assert.assertNotNull(lastException);

   Assert.assertEquals("Borrower does not exist", lastException.getMessage());

}


@Then("the system removes {string}'s details")

public void thenSystemRemovesDetails(String name) {

   Assert.assertNull(lastException);

   Assert.assertEquals("Deletion successful", lastOperationMessage);

}
```

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

public void thenSystemInformsBorrowerDoesNotExist() {

   Assert.assertNotNull(lastException);

   Assert.assertEquals("Borrower does not exist", lastException.getMessage());

}


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

public void thenSystemInformsUserAlreadyExists() {

   Assert.assertNotNull(lastException);

   Assert.assertEquals("Borrower already exists", lastException.getMessage());

}


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

public void thenSystemInformsPendingItems() {

   Assert.assertNotNull(lastException);

   Assert.assertEquals("Borrower has pending items", lastException.getMessage());

}


@Then("the system does not remove {string}'s details")

public void thenSystemDoesNotRemoveDetails(String name) {

   Assert.assertNotNull(lastException);
```

```
        Assert.assertEquals("Borrower has pending items", lastException.getMessage());
    }
}
```

### Explanation:

- **Setup and Initialization**: The `BorrowerSteps` class sets up necessary services and a mock DAO.

- **Handling Scenarios**: Each method handles a different part of the scenarios you described. The use of `try-catch` blocks allows us to capture and handle exceptions thrown by the service layer, which are then checked in the `Then` steps to confirm appropriate behavior.

- **Messaging and Exceptions**: Messages and exceptions are stored to verify that the system responds as expected under various conditions.

This set of step definitions assumes basic implementation details and may need to be adapted based on your specific technologies and frameworks (like Spring).

**User**

**Give me the methods of the borrower class**

ChatGPT

Below is a simple Java class for a `Borrower`, based on the needs outlined in your feature scenarios. This class includes basic properties, a constructor, and getter/setter methods, which are typical for a Java entity model, especially if you are planning to use it with an ORM framework like Hibernate for database operations.

### Borrower Java Class

```java
public class Borrower {
    private Long id;
    private String name;
    private boolean hasPendingItems;

    // Constructor with parameters
    public Borrower(Long id, String name, boolean hasPendingItems) {
        this.id = id;
        this.name = name;
        this.hasPendingItems = hasPendingItems;
    }

    // Default constructor
    public Borrower() {
    }

    // Getter and setter for id
    public Long getId() {
        return id;
```

```
}


public void setId(Long id) {

    this.id = id;

}


// Getter and setter for name

public String getName() {

    return name;

}


public void setName(String name) {

    this.name = name;

}


// Getter and setter for hasPendingItems

public boolean hasPendingItems() {

    return hasPendingItems;

}


public void setHasPendingItems(boolean hasPendingItems) {

    this.hasPendingItems = hasPendingItems;
    }

    // toString method for easier debugging and logging
    @Override
    public String toString() {
        return "Borrower{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", hasPendingItems=" + hasPendingItems +
                '}';
    }
}
```

### Explanation of the `Borrower` Class:

1. **Fields**:
   - `id`: A unique identifier for the borrower. This is typically used as a primary key when storing data in a database.
   - `name`: The name of the borrower.
   - `hasPendingItems`: A boolean flag indicating whether the borrower has any items that have not been returned to the library.

2. **Constructors**:

- A parameterized constructor to allow easy creation of a `Borrower` instance with all fields initialized.

- A no-argument constructor, which is often required for ORM frameworks and also for serialization purposes.

3. **Getters and Setters**:

- These methods provide the ability to retrieve and update the properties of a `Borrower`. They are standard practice in Java to ensure encapsulation and are particularly useful when interacting with frameworks that rely on them, like Java Persistence API (JPA).

4. **toString Method**:

- Overrides the `Object.toString()` method to provide a string representation of the borrower, which is very useful for logging and debugging purposes.

This class can be further modified or extended based on specific requirements, such as adding validation logic within the setters or implementing interfaces for serialization or other functionalities. If you are using a database and an ORM framework, you might also want to annotate this class with JPA annotations to define how it maps to your database schema.

**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

For the feature "Delayed Return Notification", we'll start by outlining the necessary step definitions in Java using Cucumber, similar to the previous setup. We'll assume some form of service that manages notifications and another for managing loans, in addition to a simple domain model for the borrower and the items they borrow.

### Domain Model and Services

Let's expand on our `Borrower` class to include an email address and outline a basic `LoanItem` class:

```java
public class Borrower {
    private Long id;
    private String name;
    private String emailAddress; // Added email address field
    private boolean hasPendingItems;

    // Constructors, getters and setters for all fields, including email
}

public class LoanItem {
    private String itemName;

    private LocalDate dueDate;



    // Constructors, getters and setters

}
```

Next, we need a notification service:

```java
public class NotificationService {

    public void sendEmail(String emailAddress, String message) {

        if (emailAddress != null && !emailAddress.isEmpty()) {

            // Logic to send email (this is a mockup)

            System.out.println("Sending email to " + emailAddress + ": " + message);

        }

    }

}
```

### Step Definitions for Delayed Return Notification Feature

```java
import org.junit.Assert;


public class NotificationSteps {
    private Map<String, Borrower> borrowers;
    private Map<String, LoanItem> loanItems;
    private NotificationService notificationService;
    private String lastNotificationMessage;


    public NotificationSteps() {
        borrowers = new HashMap<>();
        loanItems = new HashMap<>();
        notificationService = new NotificationService();
    }


    @Given("{string} has borrowed the item {string}")
    public void givenBorrowerHasBorrowedItem(String borrowerName, String itemName) {
        borrowers.putIfAbsent(borrowerName, new Borrower(borrowerName.hashCode().longValue(), borrowerName, null, false));
        loanItems.put(itemName, new LoanItem(itemName, LocalDate.now().minusDays(1))); // Assume it's overdue
    }
    @Given("{string}'s due date has passed")
    public void givenItemDueDateHasPassed(String itemName) {
        // This is effectively set during the item setup
    }

    @Given("{string} has an email address")
    public void givenBorrowerHasEmailAddress(String borrowerName) {
        Borrower borrower = borrowers.get(borrowerName);
        borrower.setEmailAddress(borrowerName.toLowerCase().replace(" ", "") +
"@example.com");
    }

    @Given("{string} does not have an email address")
    public void givenBorrowerDoesNotHaveEmailAddress(String borrowerName) {
        Borrower borrower = borrowers.get(borrowerName);
        borrower.setEmailAddress(null);
    }

    @When("the system executes the delayed return notification process")
    public void whenSystemExecutesNotificationProcess() {
        borrowers.forEach((name, borrower) -> {
            if (borrower.getEmailAddress() != null && !borrower.getEmailAddress().isEmpty()) {
                String message = "Please return the overdue item.";
                notificationService.sendEmail(borrower.getEmailAddress(), message);
                lastNotificationMessage = "Email sent to " + name;
            } else {
                lastNotificationMessage = "No email sent to " + name;
            }
        });
    }

    @Then("{string} receives an email notification for the return of the item")
    public void thenBorrowerReceivesEmail(String borrowerName) {
        Assert.assertTrue(lastNotificationMessage.contains("Email sent to " + borrowerName));
    }

    @Then("{string} does not receive an email notification for the return of the item")
    public void thenBorrowerDoesNotReceiveEmail(String borrowerName) {
        Assert.assertTrue(lastNotificationMessage.contains("No email sent to " + borrowerName));
    }
}
```

### Explanation:

- **Domain Setup**: The `Borrower` and `LoanItem` classes are basic constructs that facilitate this scenario.

- **Notification Logic**: The `NotificationService` is simplistic and would ideally be replaced or extended to handle real email delivery mechanisms.

- **Testing Flow**: The steps simulate conditions where an overdue item should trigger an email notification, or its absence, based on whether the borrower has an email address.

This setup uses placeholders and simplifications suitable for testing scenarios. For actual implementations, you'd need a robust email service and possibly integrate with a library or service that supports email sending functionalities.

**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**

ChatGPT

To handle the feature "Loaning items" in a library system, we'll need to define several domain classes and establish step definitions for managing loan processes under various scenarios. This involves interaction with entities like `LoanItem` and `Borrower`, and the introduction of loan management logic in service classes.

### Domain Model and Services

I'll expand the `Borrower` and introduce a `LoanItem` class for handling item loans. Additionally, a `LoanService` class will manage the logic related to loaning items based on conditions such as item availability and borrower eligibility.

### Domain Classes

**Borrower.java**

```java
public class Borrower {
    private Long id;
    private String name;
    private int pendingItems;
    private int maxLendingLimit;
    private String emailAddress; // Optional, for notification purposes.

    // Constructors, getters, and setters
}
```

**LoanItem.java**

```java
public class LoanItem {
    private String title;
    private boolean isAvailable;

    // Constructors, getters, and setters
}
```

### Service Layer

**LoanService.java**

```java
import java.time.LocalDate;

public class LoanService {
}

private Map<String, LoanItem> inventory;
private Map<String, Borrower> borrowers;

public LoanService(Map<String, LoanItem> inventory, Map<String, Borrower> borrowers) {
    this.inventory = inventory;
    this.borrowers = borrowers;
}

public boolean loanItem(String itemName, String borrowerName) throws Exception {
    LoanItem item = inventory.get(itemName);
    Borrower borrower = borrowers.get(borrowerName);
    if (item == null || !item.isAvailable()) {
        throw new Exception("Item not available");
    }
    if (borrower.getPendingItems() >= borrower.getMaxLendingLimit()) {
        throw new Exception("Lending limit reached");
    }
    borrower.setPendingItems(borrower.getPendingItems() + 1);
    item.setAvailable(false);
    return true;
}
```

```
    public void returnItem(String itemName, String borrowerName) {
        LoanItem item = inventory.get(itemName);
        Borrower borrower = borrowers.get(borrowerName);
        if (item != null && !item.isAvailable()) {
            item.setAvailable(true);
            borrower.setPendingItems(borrower.getPendingItems() - 1);
        }
    }
}
```

### Step Definitions

Below are the step definitions based on the scenarios provided:

```java
import org.junit.Assert;

public class LoanSteps {
    private Map<String, LoanItem> inventory = new HashMap<>();
    private Map<String, Borrower> borrowers = new HashMap<>();
    private LoanService loanService = new LoanService(inventory, borrowers);
    private String lastOperationResult;


    @Given("the library has the item {string} available")
    public void theLibraryHasTheItemAvailable(String itemName) {
        inventory.put(itemName, new LoanItem(itemName, true));
    }


    @Given("{string} is a registered borrower")
    public void georgeRedIsARegisteredBorrower(String borrowerName) {
        borrowers.put(borrowerName, new Borrower(borrowerName.hashCode().longValue(),
borrowerName, 0, 5, null));
    }


    @Given("{string} has {int} pending items to be returned")
    public void georgeRedHasPendingItemsToBeReturned(String borrowerName, int
pendingItems) {
        Borrower borrower = borrowers.get(borrowerName);
        borrower.setPendingItems(pendingItems);
    }


    @Given("{string} has been assigned a maximum lending limit of {int}")
```

```java
    public void georgeRedHasBeenAssignedAMaximumLendingLimitOf(String borrowerName, int maxLimit) {

        Borrower borrower = borrowers.get(borrowerName);

        borrower.setMaxLendingLimit(maxLimit);

    }


    @When("{string} borrows the item {string}")

    public void georgeRedBorrowsTheItem(String borrowerName, String itemName) {

        try {

            loanService.loanItem(itemName, borrowerName);

            lastOperationResult = "Loan successful";

        } catch (Exception e) {

            lastOperationResult = e.getMessage();

        }

    }


    @Then("the system successfully loans the item {string} to {string} with a due date set")

    public void theSystemSuccessfullyLoansTheItemWithADueDateSet(String itemName, String borrowerName) {

        Assert.assertEquals("Loan successful", lastOperationResult);

    }
```

```
  @Then("{string}'s pending items increase to {int}")
  public void georgeRedsPendingItemsIncreaseTo(String borrowerName, int expectedItems) {
    Assert.assertEquals(expectedItems, borrowers.get(borrowerName).getPendingItems());
  }

  @Then("the system does not loan {string} to {string} due to the lending limit reached")
  public void theSystemDoesNotLoanItemDueToLendingLimit(String itemName, String
borrowerName) {
    Assert.assertTrue(lastOperationResult.contains("Lending limit reached") ||
!inventory.get(itemName).isAvailable());
  }

  @Then("the system returns an error due to the item's status")
  public void theSystemReturnsAnErrorDueToTheItemsStatus() {
    Assert.assertTrue(lastOperationResult.contains("Item not available"));
  }

  @Then("the system doesn't allow the loan")
  public void theSystemDoesntAllowTheLoan() {
    Assert.assertTrue(lastOperationResult.contains("Lending limit reached"));
  }
```

```
    @Then("the system withdraws the item {string}")
    public void theSystemWithdrawsTheItem(String itemName) {
        Assert.assertFalse(inventory.get(itemName).isAvailable());
    }
}
```

### Summary

These step definitions and the accompanying service and domain classes provide the framework for handling different loaning scenarios in a library system. They facilitate the implementation of rules for checking item availability, borrower eligibility based on pending items, and lending limits. This setup can be expanded to include more complex scenarios and integrate with a database or other storage systems for persistence.

**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**

ChatGPT

For the feature "Returning a borrowed item to the library," let's develop step definitions that handle both normal and late returns of items. We'll need to account for marking the item's availability, adjusting the borrower's pending item count, and assessing fines for late returns.

### Adjusting the Domain Model

We'll continue to use the previously defined `Borrower` and `LoanItem` classes, with the assumption that `LoanItem` now includes attributes to manage due dates and loan status.

**Updated LoanItem.java**

```java
import java.time.LocalDate;

public class LoanItem {
    private String title;
    private boolean isAvailable;
    private LocalDate borrowedDate;
    private int maxLendingDays;

    // Constructors, getters, and setters
    public LoanItem(String title, LocalDate borrowedDate, int maxLendingDays) {
        this.title = title;
        this.borrowedDate = borrowedDate;
        this.maxLendingDays = maxLendingDays;
        this.isAvailable = false; // Starts as not available when loaned out
    }

    public void returnItem() {
        this.isAvailable = true;
    }

    public boolean isOverdue(LocalDate returnDate) {
        return borrowedDate.plusDays(maxLendingDays).isBefore(returnDate);
    }
}
```

### Service Layer

We'll add methods in our `LoanService` to handle the return process, including checking for overdue items and calculating fines if necessary.

**LoanService.java**

```java
import java.time.LocalDate;

public class LoanService {
    private Map<String, LoanItem> inventory;
    private Map<String, Borrower> borrowers;

    public LoanService(Map<String, LoanItem> inventory, Map<String, Borrower> borrowers) {
        this.inventory = inventory;
        this.borrowers = borrowers;
    }

    public void returnItem(String itemName, String borrowerName, LocalDate returnDate) throws Exception {
        LoanItem item = inventory.get(itemName);
        Borrower borrower = borrowers.get(borrowerName);
        if (item == null || borrower == null) {
            throw new Exception("Invalid operation");
        }
        item.returnItem();
        borrower.setPendingItems(borrower.getPendingItems() - 1);
        if (item.isOverdue(returnDate)) {
            double fine = calculateFine(borrower, item, returnDate);
            System.out.println("Fine due: $" + fine);
        }
    }

    private double calculateFine(Borrower borrower, LoanItem item, LocalDate returnDate) {
        long daysOverdue = returnDate.toEpochDay() - item.getBorrowedDate().plusDays(item.getMaxLendingDays()).toEpochDay();
        return daysOverdue * 1.50; // Fine amount calculation can vary
    }
}
```

### Step Definitions

Here's how you can implement the step definitions for this feature:

```java
import org.junit.Assert;
```

```
public class ReturnItemSteps {
    private Map<String, LoanItem> inventory = new HashMap<>();
    private Map<String, Borrower> borrowers = new HashMap<>();
    private LoanService loanService = new LoanService(inventory, borrowers);
    private String lastResultMessage;

    @Given("{string} borrowed the item {string} {int} days prior to today's date")
    public void givenItemWasBorrowedDaysAgo(String borrowerName, String itemName, int daysAgo) {
        LocalDate borrowedDate = LocalDate.now().minusDays(daysAgo);
        LoanItem item = new LoanItem(itemName, borrowedDate, 10); // Assuming max lending days as default
        inventory.put(itemName, item);
        Borrower borrower = new Borrower(borrowerName.hashCode().longValue(), borrowerName, 1, 10, null);
        borrowers.put(borrowerName, borrower);
    }

    @Given("{string} has been assigned maximum lending days of {int}")
    public void givenMaxLendingDaysAssigned(String itemName, int maxDays) {
        LoanItem item = inventory.get(itemName);
        item.setMaxLendingDays(maxDays);

    }


    @When("the return of {string} is processed")

    public void whenItemIsReturned(String itemName) {

        try {

            loanService.returnItem(itemName, "George Red", LocalDate.now());

            lastResultMessage = "Return processed";

        } catch (Exception e) {

            lastResultMessage = e.getMessage();

        }

    }


    @Then("the system marks the state of {string} as AVAILABLE")

    public void thenItemIsMarkedAvailable(String itemName) {

        Assert.assertTrue(inventory.get(itemName).isAvailable());

    }


    @Then("{string} has one less pending item")

    public void thenBorrowerHasOneLessPendingItem(String borrowerName) {

        Assert.assertEquals(0, borrowers.get(borrowerName).getPendingItems());

    }
```

```
    @Then("{string} does not pay a fine")
    public void thenBorrowerDoesNotPayAFine(String borrowerName) {
        Assert.assertTrue(lastResultMessage.contains("Return processed") && !lastResultMessage.contains("Fine due"));
    }

    @Then("{string} pays a fine based on the borrower category")
    public void thenBorrowerPaysAFine(String borrowerName) {
        Assert.assertTrue(lastResultMessage.contains("Fine due"));
    }

    @Then("the return date of the loan is set to today's date")
    public void thenReturnDateIsSetToTodaysDate() {
        Assert.assertEquals("Return processed", lastResultMessage);
    }
}
```

### Summary

These step definitions, along with the domain classes and service methods, allow the system to process the return of borrowed items, automatically update their availability status, and handle fine calculations for late returns. Adjustments might be needed based on the exact system requirements, fine policies, or additional borrower categories.

**User**

**Give me the code of every service, domain class you are using on all the above features**

ChatGPT

Below, I'll consolidate and present the full implementation of the domain classes and service classes used across all the features we discussed: borrower handling, delayed return notification, loaning items, and returning borrowed items. These classes form the backbone of the library system application.

### Domain Classes

#### Borrower.java

```java
public class Borrower {

    private Long id;

    private String name;

    private String emailAddress;

    private int pendingItems;

    private int maxLendingLimit;
    public Borrower(Long id, String name, String emailAddress, int pendingItems, int maxLendingLimit) {
        this.id = id;
        this.name = name;
        this.emailAddress = emailAddress;
        this.pendingItems = pendingItems;
        this.maxLendingLimit = maxLendingLimit;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }
```

```
public void setName(String name) {
    this.name = name;
}

public String getEmailAddress() {
    return emailAddress;
}

public void setEmailAddress(String emailAddress) {
    this.emailAddress = emailAddress;
}

public int getPendingItems() {
    return pendingItems;
}

public void setPendingItems(int pendingItems) {
    this.pendingItems = pendingItems;
}

public int getMaxLendingLimit() {
    return maxLendingLimit;
    }

    public void setMaxLendingLimit(int maxLendingLimit) {
        this.maxLendingLimit = maxLendingLimit;
    }
}
```

#### LoanItem.java

```java
import java.time.LocalDate;

public class LoanItem {
    private String title;
    private boolean isAvailable;
    private LocalDate borrowedDate;
    private int maxLendingDays;

    public LoanItem(String title, boolean isAvailable, LocalDate borrowedDate, int maxLendingDays) {
        this.title = title;
    this.isAvailable = isAvailable;

    this.borrowedDate = borrowedDate;

    this.maxLendingDays = maxLendingDays;

}



public String getTitle() {

    return title;

}



public void setTitle(String title) {

    this.title = title;

}



public boolean isAvailable() {

    return isAvailable;

}



public void setAvailable(boolean available) {

    isAvailable = available;

}



public LocalDate getBorrowedDate() {
    return borrowedDate;

}



public void setBorrowedDate(LocalDate borrowedDate) {

    this.borrowedDate = borrowedDate;

}



public int getMaxLendingDays() {

    return maxLendingDays;

}



public void setMaxLendingDays(int maxLendingDays) {

    this.maxLendingDays = maxLendingDays;

}



public void returnItem() {

    this.isAvailable = true;

}



public boolean isOverdue(LocalDate returnDate) {

    return borrowedDate.plusDays(maxLendingDays).isBefore(returnDate);

}
```

```
}
```

### Service Classes

#### BorrowerService.java

```java
public class BorrowerService {
    private BorrowerDAO borrowerDAO;

    public BorrowerService(BorrowerDAO borrowerDAO) {
        this.borrowerDAO = borrowerDAO;
    }

    public void registerBorrower(Borrower borrower) throws Exception {
        if (borrowerDAO.exists(borrower.getId())) {
            throw new Exception("Borrower already exists");
        }
        borrowerDAO.save(borrower);
    }
    public void updateBorrower(Borrower borrower) throws Exception {
        if (!borrowerDAO.exists(borrower.getId())) {
            throw new Exception("Borrower does not exist");
        }
        borrowerDAO.update(borrower);
    }

    public void deleteBorrower(Long borrowerId) throws Exception {
        if (!borrowerDAO.exists(borrowerId)) {
            throw new Exception("Borrower does not exist");
        }
        borrowerDAO.delete(borrowerId);
    }
}
```

#### LoanService.java

```java
import java.time.LocalDate;

public class LoanService {
}


private Map<String, LoanItem> inventory;

private Map<String, Borrower> borrowers;


public LoanService(Map<String, LoanItem> inventory, Map<String, Borrower> borrowers) {

    this.inventory = inventory;

    this.borrowers = borrowers;

}


public boolean loanItem(String itemName, String borrowerName) throws Exception {

    LoanItem item = inventory.get(itemName);

    Borrower borrower = borrowers.get(borrowerName);

    if (item == null || !item.isAvailable()) {

        throw new Exception("Item not available");

    }

    if (borrower.getPendingItems() >= borrower.getMaxLendingLimit()) {

        throw new Exception("Lending limit reached");

    }

    borrower.setPendingItems(borrower.getPendingItems() + 1);

    item.setAvailable(false);z

    return true;

}
```

```
    public void returnItem(String itemName, String borrowerName, LocalDate returnDate) throws Exception {
        LoanItem item = inventory.get(itemName);
        Borrower borrower = borrowers.get(borrowerName);
        if (item == null || borrower == null) {
            throw new Exception("Invalid operation");
        }

        item.returnItem();
        borrower.setPendingItems(borrower.getPendingItems() - 1);
        if (item.isOverdue(returnDate)) {
            double fine = calculateFine(borrower, item, returnDate);
            System.out.println("Fine due: $" + fine);
        }
    }

    private double calculateFine(Borrower borrower, LoanItem item, LocalDate returnDate) {
        long daysOverdue = returnDate.toEpochDay() - item.getBorrowedDate().plusDays(item.getMaxLendingDays()).toEpochDay();
        return daysOverdue * 1.50; // Fine amount calculation can vary
    }
}
```

```

### Note

This code does not include implementations for `BorrowerDAO`, which should provide the actual database interaction methods like `save`, `update`, `delete`, `exists`, etc. Also, the `LoanService` and `BorrowerService` classes contain basic logic and could be further enhanced for handling more complex scenarios and integrations, such as with real-time data handling frameworks and databases.
```