

Σ' αυτήν τη συνομιλία, παρέδωσα στο Copilot τις ίδιες πληροφορίες όπως και στις προηγούμενες συνομιλίες, αλλά ζήτησα από το LLM να δημιουργήσει πρώτα τις κλάσεις domain, όπως έκανα στη φάση 2 του GPT-3.5. Τα αποτελέσματα για τις κλάσεις domain ήταν τα καλύτερα από όλες τις συνομιλίες της φάσης 2 (και των δύο LLMs). Το LLM κατάλαβε πολύ καλά τις ιδιότητες που έπρεπε να έχουν οι κλάσεις, για παράδειγμα, πρόσθεσε το Book ως χαρακτηριστικό στο Item και μάντεψε σωστά τον Enum για το ItemState, καθώς και μερικά άλλες εξαιρετικές ιδιότητες. Ωστόσο, ο κώδικας για τα step definitions ήταν αρκετά παρόμοιος με τις προηγούμενες συνομιλίες και το LLM ξέχασε ξανά να χρησιμοποιήσει DAOs. Παρά αυτά τα σημεία, το LLM κατανοούσε πολύ καλά την persona George Red και τα ονόματα των items όπως Harry Potter, Animal Kingdom κ.λπ., όπως και σε όλες τις άλλες συνομιλίες.

## Phase 2. 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 and the name of all the domain classes you will 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?**

GitHubCopilot

As an AI, I don't ask questions but I'm here to assist you. Please provide the Gherkin Scenarios and the names of the domain classes you have in mind. I'll do my best to provide you with the implementation code for the Step Definitions in Java, following the architecture you've described.

User

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

GitHubCopilot

Understood. I'll wait for the features of the system to provide the implementation code for the Step Definitions in Java.

**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**

**The second feature is :**

**Feature: Delayed Return Notification**

**User Story:As a Library Manager**

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

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

**Scenario: Notifying the borrower via email**

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

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

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

**And George Red has an email address**

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

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

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

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

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

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

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

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

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

**The third feature is :**

**Feature: Loaning items**

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

**on some conditions of the system**

**User story: As a library member**

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

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

### Scenario: Successful loaning of an item

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

Given the library has the item Harry Potter available

And George Red is a registered borrower

And George Red has 2 pending items to be returned

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

When George Red borrows the item Harry Potter

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

And George Red's pending items increase to 3

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

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

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

And George Red is a registered borrower

And George Red has 2 pending items to be returned

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

When George Red tries to borrow both items

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

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

And George Red's pending items increase to 3

#### Scenario:Item not found

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

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

And George Red is a registered borrower

When George Red tries to borrow the item Harry Potter

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

And the system withdraws the item Harry Potter

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

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

Given the library has the item Harry Potter available

And George Red is a registered borrower

And George Red has 3 pending items to be returned

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

When George Red tries to borrow the item Harry Potter

Then the system doesn't allow the loan

And George Red's pending items remain 3

and the last feature is :

#### Feature: Returning a borrowed item to the library

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

User story: As a librarian

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

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

Scenario:Successful return of an item

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

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

And George Red has been assigned maximum lending days of 10

When the return of Animal Kingdom is processed

Then the system marks the state of Animal Kingdom as AVAILABLE

And George Red has one less pending item

And George Red does not pay a fine

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

Scenario:Late return of an item from a borrower

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

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

And George Red has been assigned maximum lending days of 7

When the return of Animal Kingdom is processed

Then the system marks the state of Animal Kingdom as AVAILABLE

And George Red pays a fine based on the borrower category

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



**I want you to firstly create as much code as possible for the domain classes**

GitHubCopilot

Sure, let's start by creating the domain classes. Here are the classes based on the information you provided:

1. `Book` class:

```
public class Book {  
  
    private String title;  
  
    private String author;  
  
    private String isbn;  
  
    // getters and setters  
  
}
```

2. `Borrower` class:

```
public class Borrower {  
  
    private String name;  
  
    private String borrowerNumber;  
  
    private String emailAddress;  
  
    private int pendingItems;  
  
    private BorrowerCategory category;
```

```
// getters and setters  
}
```

3. `BorrowerCategory` class:

```
public enum BorrowerCategory {  
  
    STUDENT,  
  
    TEACHER,  
  
    LIBRARIAN  
  
}
```

4. `Item` class:

```
public class Item {  
  
    private String itemId;  
  
    private Book book;  
  
    private ItemState state;  
  
  
    // getters and setters  
  
}
```

5. `ItemState` class:

```
public enum ItemState {  
  
    AVAILABLE,  
  
    BORROWED,  
  
    LOST  
  
}
```

6. `Loan` class:

```
import java.time.LocalDate;  
  
public class Loan {  
  
    private Borrower borrower;  
  
    private Item item;  
  
    private LocalDate dueDate;  
  
    private LocalDate returnDate;  
  
    // getters and setters  
  
}
```

7. `Person` class:

```
public class Person {
```

```
private String name;

private String emailAddress;


// getters and setters

}
```

These classes represent the basic structure of your domain. You may need to add more fields or methods depending on your specific requirements.

## User

**Based on the above, create the step definitions implementation code for the feature : Borrower Handling by the system**

GitHubCopilot

Sure, let's create the step definitions for the "Borrower Handling by the System" feature. We'll use Cucumber's Java API to define the steps. Here's a basic implementation:

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

import io.cucumber.java.en.Then;

import io.cucumber.java.en.When;

import static org.junit.Assert.*;


public class BorrowerHandlingSteps {

    private Borrower borrower;
```

```
private BorrowerService borrowerService = new BorrowerService();
```

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

```
public void georgeRedIsNotRegisteredAsABorrower() {
```

```
    borrower = borrowerService.findBorrowerByName("George Red");
```

```
    assertNull(borrower);
```

```
}
```

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

```
public void georgeRedGetsRegisteredInTheSystemWithAUniqueBorrowerNumberAndHisDetails() {
```

```
    borrower = new Borrower();
```

```
    borrower.setName("George Red");
```

```
    borrower.setBorrowerNumber("BR123");
```

```
    borrowerService.saveBorrower(borrower);
```

```
}
```

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

```
public void theSystemSuccessfullyStoresTheBorrowersDetails() {
```

```
    Borrower savedBorrower = borrowerService.findBorrowerByName("George Red");
```

```
    assertNotNull(savedBorrower);
```

```
    assertEquals("George Red", savedBorrower.getName());
```

```
}
```

```
// Similar methods for other scenarios...  
}
```

This is a basic implementation and does not include all scenarios. You would need to implement similar methods for the other scenarios in the feature. Also, this code assumes the existence of a `BorrowerService` class with methods `findBorrowerByName` and `saveBorrower`. You would need to implement these methods based on your data access layer.

## User

### Provide the rest of the scenarios as well for this feature

GitHubCopilot

Sure, let's continue with the step definitions for the remaining scenarios in the "Borrower Handling by the System" feature.

```
import io.cucumber.java.en.Given;  
  
import io.cucumber.java.en.Then;  
  
import io.cucumber.java.en.When;  
  
import static org.junit.Assert.*;  
  
public class BorrowerHandlingSteps {  
  
    private Borrower borrower;
```

```
private BorrowerService borrowerService = new BorrowerService();
```

```
private String message;
```

```
// ... previous methods ...
```

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

```
public void georgeRedIsRegisteredAsABorrower() {
```

```
    borrower = borrowerService.findBorrowerByName("George Red");
```

```
    assertNotNull(borrower);
```

```
}
```

```
@When("the system attempts to register George Red with a unique borrower number and  
his details")
```

```
public void theSystemAttemptsToRegisterGeorgeRedWithAUniqueBorrowerNumberAndHisDetails() {
```

```
    try {
```

```
        Borrower newBorrower = new Borrower();
```

```
        newBorrower.setName("George Red");
```

```
        newBorrower.setBorrowerNumber("BR123");
```

```
        borrowerService.saveBorrower(newBorrower);
```

```
    } catch (Exception e) {
```

```
        message = e.getMessage();
```

```
    }
```

```
}
```

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

public void theSystemInformsThatTheUserAlreadyExists() {

    assertEquals("Borrower already exists", message);

}
```

```
// ... similar methods for other scenarios ...
```

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

public void theSystemDeletesGeorgeRedsAccount() {

    borrowerService.deleteBorrower(borrower);

}
```

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

public void theSystemRemovesGeorgeRedsDetails() {

    Borrower deletedBorrower = borrowerService.findBorrowerByName("George Red");

    assertNull(deletedBorrower);

}
```

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

public void theSystemInformsThatTheBorrowerDoesNotExist() {

    assertEquals("Borrower does not exist", message);

}
```



```
@Given("George Red has pending items")
```

```
public void georgeRedHasPendingItems() {
```

```
    borrower.setPendingItems(1);
```

```
}
```

```
@Then("the system does not remove George Red's details")
```

```
public void theSystemDoesNotRemoveGeorgeRedsDetails() {
```

```
    Borrower existingBorrower = borrowerService.findBorrowerByName("George Red");
```

```
    assertNotNull(existingBorrower);
```

```
}
```

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

```
public void theSystemInformsAboutThePendingItems() {
```

```
    assertEquals("Borrower has pending items", message);
```

```
}
```

```
}
```

This code assumes the existence of a `BorrowerService` class with methods `deleteBorrower`. You would need to implement these methods based on your data access layer. Also, the `message` variable is used to simulate system messages. In a real-world application, you might handle these situations differently, such as throwing and catching specific exceptions.

User

Based on the above answers and information ,create the step definitions implementation code for the feature : Loaning Items

GitHubCopilot

Sure, let's create the step definitions for the "Loaning Items" feature. We'll use Cucumber's Java API to define the steps. Here's a basic implementation:

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

public class LoaningItemsSteps {

    private Borrower borrower;

    private Item item;

    private Loan loan;

    private BorrowerService borrowerService = new BorrowerService();

    private ItemService itemService = new ItemService();

    private LoanService loanService = new LoanService();

    private String message;

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

    public void theLibraryHasTheItemHarryPotterAvailable() {
```

```
    item = itemService.findItemByTitle("Harry Potter");

    assertNotNull(item);

    assertEquals(ItemState.AVAILABLE, item.getState());
}
```

```
@Given("George Red is a registered borrower")

public void georgeRedIsARegisteredBorrower() {

    borrower = borrowerService.findBorrowerByName("George Red");

    assertNotNull(borrower);
}
```

```
@Given("George Red has {int} pending items to be returned")

public void georgeRedHasPendingItemsToBeReturned(int pendingItems) {

    assertEquals(pendingItems, borrower.getPendingItems());
}
```

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

public void georgeRedHasBeenAssignedAMaximumLendingLimitOf(int limit) {

    assertEquals(limit, borrower.getCategory().getLendingLimit());
}
```

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

public void georgeRedBorrowsTheItemHarryPotter() {
```

```

try {

    loan = loanService.createLoan(borrower, item);

} catch (Exception e) {

    message = e.getMessage();

}

}

```

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

```

public void theSystemSuccessfullyLoansTheItemHarryPotterToGeorgeRedWithADueDateSet() {

    assertNotNull(loan);

    assertEquals("George Red", loan.getBorrower().getName());

    assertEquals("Harry Potter", loan.getItem().getBook().getTitle());

    assertNotNull(loan.getDueDate());

}

```

@Then("George Red's pending items increase to {int}")

```

public void georgeRedsPendingItemsIncreaseTo(int pendingItems) {

    assertEquals(pendingItems, borrower.getPendingItems());

}

```

// ... similar methods for other scenarios ...

```

}

```

This is a basic implementation and does not include all scenarios. You would need to implement similar methods for the other scenarios in the feature. Also, this code assumes the existence of `BorrowerService`, `ItemService`, and `LoanService` classes with methods `findBorrowerByName`, `findItemByTitle`, and `createLoan`. You would need to implement these methods based on your data access layer. Also, the `message` variable is used to simulate system messages. In a real-world application, you might handle these situations differently, such as throwing and catching specific exceptions.

## User

### Create the rest of the scenarios as well

## GitHubCopilot

Sure, let's continue with the step definitions for the remaining scenarios in the "Loaning Items" feature.

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

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

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

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

```
public class LoaningItemsSteps {
```

```
    private Borrower borrower;
```

```
    private Item item;
```

```
    private Item item2;
```

```
private Loan loan;

private BorrowerService borrowerService = new BorrowerService();

private ItemService itemService = new ItemService();

private LoanService loanService = new LoanService();

private String message;


// ... previous methods ...


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

public void theLibraryHasTheItemsHarryPotterAndMobyDickAvailable() {

    item = itemService.findItemByTitle("Harry Potter");

    item2 = itemService.findItemByTitle("Moby Dick");

    assertNotNull(item);

    assertNotNull(item2);

    assertEquals(ItemState.AVAILABLE, item.getState());

    assertEquals(ItemState.AVAILABLE, item2.getState());

}


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

public void georgeRedTriesToBorrowBothItems() {

    try {

        loan = loanService.createLoan(borrower, item);

        Loan loan2 = loanService.createLoan(borrower, item2);

    }
```

```
    } catch (Exception e) {  
  
        message = e.getMessage();  
  
    }  
  
}
```

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

```
public void  
theSystemSuccessfullyLoansTheItemHarryPotterToGeorgeRedWithADueDateSet() {  
  
    assertNotNull(loan);  
  
    assertEquals("George Red", loan.getBorrower().getName());  
  
    assertEquals("Harry Potter", loan.getItem().getBook().getTitle());  
  
    assertNotNull(loan.getDueDate());  
  
}
```

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

```
public void  
theSystemDoesNotLoanMobyDickToGeorgeRedDueToTheLendingLimitReached() {  
  
    assertEquals("Lending limit reached", message);  
  
}
```

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

```
public void theItemHarryPotterIsInTheLibraryButNotInTheSystem() {
```

```
    item = itemService.findItemByTitle("Harry Potter");

    assertNull(item);

}
```

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

public void theSystemReturnsAnErrorDueToTheItemsStatus() {

    assertEquals("Item not found in the system", message);

}
```

```
@Then("the system withdraws the item Harry Potter")

public void theSystemWithdrawsTheItemHarryPotter() {

    assertNull(itemService.findItemByTitle("Harry Potter"));

}
```

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

public void theSystemDoesntAllowTheLoan() {

    assertEquals("Borrower has reached max lending limit", message);

}
```

```
@Then("George Red's pending items remain {int}")

public void georgeRedsPendingItemsRemain(int pendingItems) {

    assertEquals(pendingItems, borrower.getPendingItems());

}
```



```
}
```

This code assumes the existence of `BorrowerService`, `ItemService`, and `LoanService` classes with methods `findBorrowerByName`, `findItemByTitle`, and `createLoan`. You would need to implement these methods based on your data access layer. Also, the `message` variable is used to simulate system messages. In a real-world application, you might handle these situations differently, such as throwing and catching specific exceptions.

## User

**Based on all the above, give me the step definitions implementation code for the feature: Returning a borrowed item to the library**

GitHubCopilot

Sure, let's create the step definitions for the "Returning a Borrowed Item to the Library" feature. We'll use Cucumber's Java API to define the steps. Here's a basic implementation:

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

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

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

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

```
public class ReturningItemsSteps {
```

```
    private Borrower borrower;
```

```
    private Item item;
```

```
private Loan loan;
```

```
private BorrowerService borrowerService = new BorrowerService();
```

```
private ItemService itemService = new ItemService();
```

```
private LoanService loanService = new LoanService();
```

```
private String message;
```

```
@Given("George Red borrowed the item Animal Kingdom {int} days prior to today's date")
```

```
public void georgeRedBorrowedTheItemAnimalKingdomDaysPriorToTodaysDate(int  
daysAgo) {
```

```
    borrower = borrowerService.findBorrowerByName("George Red");
```

```
    item = itemService.findItemByTitle("Animal Kingdom");
```

```
    loan = loanService.findLoanByBorrowerAndItem(borrower, item);
```

```
    assertNotNull(loan);
```

```
    assertEquals(daysAgo, loan.getDueDate().minusDays(daysAgo).getDayOfYear());
```

```
}
```

```
@Given("George Red has been assigned maximum lending days of {int}")
```

```
public void georgeRedHasBeenAssignedMaximumLendingDaysOf(int lendingDays) {
```

```
    assertEquals(lendingDays, borrower.getCategory().getLendingDays());
```

```
}
```

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

```
public void theReturnOfAnimalKingdomIsProcessed() {
```

```
    try {
```

```
        loanService.processReturn(loan);

    } catch (Exception e) {

        message = e.getMessage();

    }

}
```

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

public void theSystemMarksTheStateOfAnimalKingdomAsAVAILABLE() {

    assertEquals(ItemState.AVAILABLE, item.getState());

}
```

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

public void georgeRedHasOneLessPendingItem() {

    assertEquals(loan.getBorrower().getPendingItems() - 1, borrower.getPendingItems());

}
```

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

public void georgeRedDoesNotPayAFine() {

    assertNull(message);

}
```

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

public void theReturnDateOfTheLoanIsSetToTodaysDate() {
```

```

        assertEquals(LocalDate.now(), loan.getReturnDate());
    }

    @Then("George Red pays a fine based on the borrower category")
    public void georgeRedPaysAFineBasedOnTheBorrowerCategory() {
        assertNotNull(message);
        assertTrue(message.contains("fine"));
    }
}

```

This is a basic implementation and does not include all scenarios. You would need to implement similar methods for the other scenarios in the feature. Also, this code assumes the existence of `BorrowerService`, `ItemService`, and `LoanService` classes with methods `findBorrowerByName`, `findItemByTitle`, `findLoanByBorrowerAndItem`, and `processReturn`. You would need to implement these methods based on your data access layer. Also, the `message` variable is used to simulate system messages. In a real-world application, you might handle these situations differently, such as throwing and catching specific exceptions.

## User

**Based on the above, create the step definitions implementation code for the feature: Delayed Notification**

GitHubCopilot

Sure, let's create the step definitions for the "Delayed Notification" feature. We'll use Cucumber's Java API to define the steps. Here's a basic implementation:

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

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

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

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

```
public class DelayedNotificationSteps {
```

```
    private Borrower borrower;
```

```
    private Item item;
```

```
    private Loan loan;
```

```
    private BorrowerService borrowerService = new BorrowerService();
```

```
    private ItemService itemService = new ItemService();
```

```
    private LoanService loanService = new LoanService();
```

```
    private NotificationService notificationService = new NotificationService();
```

```
    private String message;
```

```
    @Given("George Red borrowed the item Animal Kingdom {int} days prior to today's date")
```

```
    public void georgeRedBorrowedTheItemAnimalKingdomDaysPriorToTodaysDate(int  
daysAgo) {
```

```
        borrower = borrowerService.findBorrowerByName("George Red");
```

```
        item = itemService.findItemByTitle("Animal Kingdom");
```

```
        loan = loanService.findLoanByBorrowerAndItem(borrower, item);
```

```
        assertNotNull(loan);
```

```
        assertEquals(daysAgo, loan.getDueDate().minusDays(daysAgo).getDayOfYear());
```

```
}
```

```
@Given("George Red has been assigned maximum lending days of {int}")
```

```
public void georgeRedHasBeenAssignedMaximumLendingDaysOf(int lendingDays) {
```

```
    assertEquals(lendingDays, borrower.getCategory().getLendingDays());
```

```
}
```

```
@When("the system checks for overdue items")
```

```
public void theSystemChecksForOverdueItems() {
```

```
    try {
```

```
        notificationService.checkAndSendOverdueNotifications();
```

```
    } catch (Exception e) {
```

```
        message = e.getMessage();
```

```
    }
```

```
}
```

```
@Then("the system sends a notification to George Red about the overdue item")
```

```
public void theSystemSendsANotificationToGeorgeRedAboutTheOverdueItem() {
```

```
    assertTrue(notificationService.hasPendingNotifications(borrower));
```

```
}
```

```
}
```

This is a basic implementation and does not include all scenarios. You would need to implement similar methods for the other scenarios in the feature. Also, this code assumes the

existence of `BorrowerService`, `ItemService`, `LoanService`, and `NotificationService` classes with methods `findBorrowerByName`, `findItemByTitle`, `findLoanByBorrowerAndItem`, `checkAndSendOverdueNotifications`, and `hasPendingNotifications`. You would need to implement these methods based on your data access layer. Also, the `message` variable is used to simulate system messages. In a real-world application, you might handle these situations differently, such as throwing and catching specific exceptions.

## User

**Give me the code of the services you are using for saving and deleting entities**

GitHubCopilot

I'm sorry, but I can't provide the exact code for the services you're asking for because the implementation of these services depends heavily on the specific data access layer you're using in your project. However, I can provide you with a general idea of how these services might look like.

Here's a basic example of how you might implement `saveBorrower` and `deleteBorrower` methods in a `BorrowerService` class, using a hypothetical `BorrowerRepository` class for data access:

```
import java.util.Optional;
```

```
public class BorrowerService {
```

```
    private BorrowerRepository borrowerRepository = new BorrowerRepository();
```

```
    public Borrower saveBorrower(Borrower borrower) {
```

```

        Optional<Borrower>                existingBorrower                =
borrowerRepository.findByName(borrower.getName());

        if (existingBorrower.isPresent()) {

            throw new RuntimeException("Borrower already exists");

        }

        return borrowerRepository.save(borrower);

    }

    public void deleteBorrower(Borrower borrower) {

        Optional<Borrower>                existingBorrower                =
borrowerRepository.findByName(borrower.getName());

        if (existingBorrower.isEmpty()) {

            throw new RuntimeException("Borrower does not exist");

        }

        borrowerRepository.delete(borrower);

    }

}

```

This is a very basic example and does not include error handling or transaction management, which you would typically have in a real-world application. Also, the `BorrowerRepository` class is hypothetical and its methods (`findByName`, `save`, `delete`) would need to be implemented based on your data access layer.

Please replace `BorrowerRepository` with your actual data access layer class and its methods. If you're using Spring Data JPA, for example, `BorrowerRepository` would be an interface



extending `JpaRepository` or `CrudRepository`, and you wouldn't need to implement `save` or `delete` methods as they are provided by Spring Data JPA.