

Αυτή η συνομιλία ήταν ένα πείραμα για να διαπιστώσω αν το LLM θα μου παρέχει ξανά τις κλάσεις Domain ως αρχικό σημείο, χωρίς να του το ζητήσω, κάτι που τελικά έκανε. Στη συνέχεια, συνέχισα τη συνομιλία για να αποκτήσω τον πλήρη κώδικα των step definitions. Τα αποτελέσματα ήταν συνολικά καλά και αρκετά παρόμοια με τα αποτελέσματα των προηγούμενων συνομιλιών, με βάση τις γνώσεις που παρείχα στο LLM. Το σύστημα άρχισε να δημιουργεί κώδικα αμέσως, απαιτώντας πολύ λίγες εντολές.

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 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 can certainly help you with your request. Please provide the Gherkin Scenarios and I will assist you in creating the corresponding Step definitions, Domain classes, DAOs, and Service classes 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

GitHubCopilot

Based on the provided Gherkin scenarios, we can start by creating the necessary domain classes, DAOs, and service classes.

Let's start with the domain classes. We will need classes for `Borrower`, `Item`, `Loan`, and `Notification`.

Here is the `Borrower` class:


```
public class Borrower {  
  
    private String name;  
  
    private String borrowerNumber;  
  
    private String emailAddress;  
  
    private int pendingItems;  
  
    private int lendingLimit;  
  
  
    // getters and setters  
  
}
```

Next, we have the `Item` class:

```
public class Item {  
  
    private String title;  
  
    private boolean isAvailable;  
  
    private String dueDate;  
  
  
    // getters and setters  
  
}
```

The `Loan` class:

```
public class Loan {  
  
    private Borrower borrower;  
  
    private Item item;  
  
    private String loanDate;  
  
    private String dueDate;  
  
    private String returnDate;  
  
    // getters and setters  
}
```

And the `Notification` class:

```
public class Notification {  
  
    private Borrower borrower;  
  
    private String message;  
  
    // getters and setters
```

```
}
```

Next, we will create the DAOs for these classes. The DAOs will handle the data access operations such as saving, updating, and deleting entities.

Here is the `BorrowerDAO`:

```
public class BorrowerDAO {  
  
    public void save(Borrower borrower) {  
  
        // implementation code  
  
    }  
  
    public void update(Borrower borrower) {  
  
        // implementation code  
  
    }  
  
    public void delete(Borrower borrower) {  
  
        // implementation code  
  
    }  
  
    public Borrower findByName(String name) {  
  
        // implementation code
```

```
}  
}
```

The `ItemDAO`:

```
public class ItemDAO {  
  
    public void save(Item item) {  
  
        // implementation code  
  
    }  
  
    public void update(Item item) {  
  
        // implementation code  
  
    }  
  
    public void delete(Item item) {  
  
        // implementation code  
  
    }  
  
    public Item findByTitle(String title) {  
  
        // implementation code  
  
    }  
}
```

```
}
```

The `LoanDAO`:

```
public class LoanDAO {  
  
    public void save(Loan loan) {  
  
        // implementation code  
  
    }  
  
    public void update(Loan loan) {  
  
        // implementation code  
  
    }  
  
    public void delete(Loan loan) {  
  
        // implementation code  
  
    }  
  
    public Loan findByBorrowerAndItem(Borrower borrower, Item item) {  
  
        // implementation code  
  
    }  
}
```

And the `NotificationDAO`:

```
public class NotificationDAO {  
  
    public void save(Notification notification) {  
  
        // implementation code  
  
    }  
  
    public void update(Notification notification) {  
  
        // implementation code  
  
    }  
  
    public void delete(Notification notification) {  
  
        // implementation code  
  
    }  
  
    public Notification findByBorrower(Borrower borrower) {  
  
        // implementation code  
  
    }  
}
```

Finally, we will create the service classes that will use the DAOs to perform the operations described in the Gherkin scenarios.

Here is the `BorrowerService`:

```
public class BorrowerService {  
  
    private BorrowerDAO borrowerDAO;  
  
    public void registerBorrower(String name, String borrowerNumber, String emailAddress, int  
lendingLimit) {  
  
        // implementation code  
  
    }  
  
    public void updateBorrowerDetails(String name, String borrowerNumber, String  
emailAddress, int lendingLimit) {  
  
        // implementation code  
  
    }  
  
    public void deleteBorrower(String name) {  
  
        // implementation code  
  
    }  
}
```

The `LoanService`:

```
public class LoanService {  
  
    private LoanDAO loanDAO;  
  
    private ItemDAO itemDAO;  
  
    private BorrowerDAO borrowerDAO;  
  
  
    public void loanItem(String borrowerName, String itemTitle) {  
  
        // implementation code  
  
    }  
  
  
    public void returnItem(String borrowerName, String itemTitle) {  
  
        // implementation code  
  
    }  
}
```

And the `NotificationService`:


```
public class NotificationService {  
  
    private NotificationDAO notificationDAO;  
  
    private BorrowerDAO borrowerDAO;  
  
    public void sendDelayedReturnNotification(String borrowerName) {  
  
        // implementation code  
  
    }  
}
```

Please note that the actual implementation of the methods in the DAOs and service classes will depend on the specific technology you are using for data storage (e.g., a relational database, a NoSQL database, etc.) and the specific requirements of your application.

User

Based on the above, create the step definitions for the feature: Borrower Handling By the system

GitHubCopilot

Sure, let's create the step definitions for the feature: Borrower Handling By the system. We will use Cucumber for Java to implement the step definitions.

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

```
    private Borrower borrower;
```

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

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

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

```
        assertNull(borrower);
```

```
    }
```

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

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

```
        borrowerService.registerBorrower("George Red", "uniqueBorrowerNumber",  
        "george.red@example.com", 5);
```

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

```
    }
```

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

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

```
        assertNotNull(borrower);
```

```
    assertEquals("George Red", borrower.getName());  
}
```

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

```
public void georgeRedIsRegisteredAsABorrower() {  
    borrowerService.registerBorrower("George Red", "uniqueBorrowerNumber",  
"george.red@example.com", 5);  
    borrower = borrowerService.findByName("George Red");  
    assertNotNull(borrower);  
}
```

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

```
public void  
theSystemAttemptsToRegisterGeorgeRedWithAUniqueBorrowerNumberAndHisDetails() {  
    try {  
        borrowerService.registerBorrower("George Red", "uniqueBorrowerNumber",  
"george.red@example.com", 5);  
    } catch (Exception e) {  
        // handle exception  
    }  
}
```

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

```
public void theSystemInformsThatTheUserAlreadyExists() {  
  
    borrower = borrowerService.findByName("George Red");  
  
    assertNotNull(borrower);  
  
}
```

```
@When("George Red updates his borrowing details")  
  
public void georgeRedUpdatesHisBorrowingDetails() {  
  
    borrowerService.updateBorrowerDetails("George Red", "newUniqueBorrowerNumber",  
"new.george.red@example.com", 7);  
  
    borrower = borrowerService.findByName("George Red");  
  
}
```

```
@Then("the system saves the changes")  
  
public void theSystemSavesTheChanges() {  
  
    assertEquals("newUniqueBorrowerNumber", borrower.getBorrowerNumber());  
  
    assertEquals("new.george.red@example.com", borrower.getEmailAddress());  
  
    assertEquals(7, borrower.getLendingLimit());  
  
}
```

```
@When("George Red tries to update his borrowing details")  
  
public void georgeRedTriesToUpdateHisBorrowingDetails() {  
  
    try {  
  
        borrowerService.updateBorrowerDetails("George Red",  
"newUniqueBorrowerNumber", "new.george.red@example.com", 7);
```

```
    } catch (Exception e) {  
  
        // handle exception  
  
    }  
}
```

```
@Then("the system displays an error message indicating that George Red does not exist")  
  
public void theSystemDisplaysAnErrorMessageIndicatingThatGeorgeRedDoesNotExist() {  
  
    borrower = borrowerService.findByName("George Red");  
  
    assertNull(borrower);  
  
}
```

```
@When("the system deletes George Red's account")  
  
public void theSystemDeletesGeorgeRedSAccount() {  
  
    borrowerService.deleteBorrower("George Red");  
  
    borrower = borrowerService.findByName("George Red");  
  
}
```

```
@Then("the system removes George Red's details")  
  
public void theSystemRemovesGeorgeRedSDetails() {  
  
    assertNull(borrower);  
  
}
```

```
@When("the system attempts to delete George Red's account")
```

```
public void theSystemAttemptsToDeleteGeorgeRedSAccount() {  
  
    try {  
  
        borrowerService.deleteBorrower("George Red");  
  
    } catch (Exception e) {  
  
        // handle exception  
  
    }  
  
}
```

```
@Then("the system informs that the borrower does not exist")  
  
public void theSystemInformsThatTheBorrowerDoesNotExist() {  
  
    borrower = borrowerService.findByName("George Red");  
  
    assertNull(borrower);  
  
}
```

```
// Add the remaining steps for the scenarios "Handling unreturned items of books" here  
  
}
```

Please note that the actual implementation of the methods in the `BorrowerService` class will depend on the specific requirements of your application. The above code is a basic template and may need to be adjusted to fit your needs.

User

Give me the remaining scenarios of this feature

GitHubCopilot

Sure, here are the remaining step definitions for the "Borrower Handling By the system" feature:

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

```
public class BorrowerHandlingSteps {
```

```
    // existing code...
```

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

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

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

```
        assertTrue(borrower.getPendingItems() > 0);
```

```
    }
```

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

```
    public void theSystemDoesNotRemoveGeorgeRedSDetails() {
```

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

```
        assertNotNull(borrower);
```

```
    }
```

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

```
public void theSystemInformsAboutThePendingItems() {  
  
    borrower = borrowerService.findByName("George Red");  
  
    assertTrue(borrower.getPendingItems() > 0);  
  
}  
}
```

These step definitions correspond to the scenario "Handling unreturned items of books". They check if George Red has pending items and if so, the system does not remove his details and informs about the pending items.

User

Based on all the above, give me the step definitions of the feature : Loaning Items

GitHubCopilot

Sure, let's create the step definitions for the feature: Loaning Items. We will use Cucumber for Java to implement the step definitions.

```
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 LoanService loanService = new LoanService();

    private Item item;

    private Borrower borrower;

    private Loan loan;

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

    public void theLibraryHasTheItemHarryPotterAvailable() {

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

        assertTrue(item.isAvailable());

    }

    @And("George Red is a registered borrower")

    public void georgeRedIsARegisteredBorrower() {

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

        assertNotNull(borrower);

    }

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

    public void georgeRedHasPendingItemsToBeReturned(int pendingItems) {

        assertEquals(pendingItems, borrower.getPendingItems());

    }

}
```

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

```
public void georgeRedHasBeenAssignedAMaximumLendingLimitOf(int lendingLimit) {  
    assertEquals(lendingLimit, borrower.getLendingLimit());  
}
```

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

```
public void georgeRedBorrowsTheItemHarryPotter() {  
    loanService loanItem("George Red", "Harry Potter");  
    loan = loanService.findLoanByBorrowerAndItem(borrower, item);  
}
```

```
@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().getTitle());  
    assertNotNull(loan.getDueDate());  
}
```

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

```
public void georgeRedSPendingItemsIncreaseTo(int newPendingItems) {  
    assertEquals(newPendingItems, borrower.getPendingItems());  
}
```

```
}
```

```
// Add the remaining steps for the scenarios "Borrower can borrow only one item due to  
his lending limit", "Item not found", and "The borrower is not eligible to borrow" here  
}
```

Please note that the actual implementation of the methods in the `LoanService` class will depend on the specific requirements of your application. The above code is a basic template and may need to be adjusted to fit your needs.

User

Give me these remaining steps for this feature

GitHubCopilot

Sure, here are the remaining step definitions for the "Loaning Items" feature:

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

```
public class LoaningItemsSteps {
```

```
    // existing code...
```

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

```

public void theLibraryHasTheItemsHarryPotterAndMobyDickAvailable() {

    Item itemMobyDick = loanService.findItemByTitle("Moby Dick");

    assertTrue(itemMobyDick.isAvailable());

    assertTrue(item.isAvailable());

}

```

```

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

```

```

public void georgeRedTriesToBorrowBothItems() {

    try {

        loanService loanItem("George Red", "Harry Potter");

        loanService loanItem("George Red", "Moby Dick");

    } catch (Exception e) {

        // handle exception

    }

}

```

```

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

```

```

public void theSystemSuccessfullyLoansTheItemHarryPotterToGeorgeRedWithADueDateSet() {

    loan = loanService.findLoanByBorrowerAndItem(borrower, item);

    assertNotNull(loan);

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

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

}

```

```
        assertNotNull(loan.getDueDate());  
    }  
  
}
```

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

```
public void theSystemDoesNotLoanMobyDickToGeorgeRedDueToTheLendingLimitReached() {  
    Loan loanMobyDick = loanService.findLoanByBorrowerAndItem(borrower,  
itemMobyDick);  
    assertNull(loanMobyDick);  
}  
  
}
```

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

```
public void theItemHarryPotterIsInTheLibraryButNotInTheSystem() {  
    item = loanService.findItemByTitle("Harry Potter");  
    assertNull(item);  
}  
  
}
```

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

```
public void theSystemReturnsAnErrorDueToTheItemSStatus() {  
    // This step would involve checking for an exception or error message that was thrown  
when trying to borrow the item  
}  
  
}
```

```

@And("the system withdraws the item Harry Potter")

public void theSystemWithdrawsTheItemHarryPotter() {

    // This step would involve checking that the item has been removed from the system or
    marked as unavailable

}

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

public void theSystemDoesnTAllowTheLoan() {

    loan = loanService.findLoanByBorrowerAndItem(borrower, item);

    assertNull(loan);

}

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

public void georgeRedSPendingItemsRemain(int pendingItems) {

    assertEquals(pendingItems, borrower.getPendingItems());

}

}

```

These step definitions correspond to the scenarios "Borrower can borrow only one item due to his lending limit", "Item not found", and "The borrower is not eligible to borrow". They check various conditions such as the availability of items, the borrower's lending limit, and the system's response to these conditions.

User

Now, based on all the above and with the same pattern, give me the step definitions for the feature: Returning a borrowed item to the library

GitHubCopilot

Sure, let's create the step definitions for the feature: Returning a borrowed item to the library. We will use Cucumber for Java to implement the step definitions.

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

```
    private Item item;
```

```
    private Borrower borrower;
```

```
    private Loan loan;
```

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

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

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

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

    loan = loanService.findLoanByBorrowerAndItem(borrower, item);

    assertNotNull(loan);

    // Assume that we have a method to calculate the date X days ago

    String expectedLoanDate = calculateDateDaysAgo(daysAgo);

    assertEquals(expectedLoanDate, loan.getLoanDate());

}
```

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

public void georgeRedHasBeenAssignedMaximumLendingDaysOf(int lendingDays) {

    assertEquals(lendingDays, borrower.getLendingDays());

}
```

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

public void theReturnOfAnimalKingdomIsProcessed() {

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

    loan = loanService.findLoanByBorrowerAndItem(borrower, item);

}
```

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

public void theSystemMarksTheStateOfAnimalKingdomAsAVAILABLE() {

    assertTrue(item.isAvailable());

}
```



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

```
public void georgeRedHasOneLessPendingItem() {
```

```
    int expectedPendingItems = borrower.getPendingItems() - 1;
```

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

```
}
```

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

```
public void georgeRedDoesNotPayAFine() {
```

```
    // Assume that we have a method to calculate the fine
```

```
    double fine = calculateFine(borrower, loan);
```

```
    assertEquals(0.0, fine, 0.01);
```

```
}
```

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

```
public void theReturnDateOfTheLoanIsSetToTodaySDate() {
```

```
    // Assume that we have a method to get today's date
```

```
    String today = getTodayDate();
```

```
    assertEquals(today, loan.getReturnDate());
```

```
}
```

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

```
public void georgeRedPaysAFineBasedOnTheBorrowerCategory() {
```

```
// Assume that we have a method to calculate the fine

double fine = calculateFine(borrower, loan);

assertTrue(fine > 0);

}

// Add any additional methods needed for these steps here, such as calculateDateDaysAgo,
calculateFine, and getTodayDate

}
```

Please note that the actual implementation of the methods in the `LoanService` class and the additional methods in the step definitions will depend on the specific requirements of your application. The above code is a basic template and may need to be adjusted to fit your needs.

User

Based on the above, give me the step definitions for the feature: Delayed Notification

GitHubCopilot

Sure, let's create the step definitions for the feature: Delayed Notification. We will use Cucumber for Java to implement the step definitions.

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

```
    private Borrower borrower;
```

```
    private Loan loan;
```

```
    @Given("George Red has borrowed the item Animal Kingdom {int} days ago")
```

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

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

```
        loan = notificationService.findLoanByBorrowerAndItem(borrower, "Animal Kingdom");
```

```
        assertNotNull(loan);
```

```
        // Assume that we have a method to calculate the date X days ago
```

```
        String expectedLoanDate = calculateDateDaysAgo(daysAgo);
```

```
        assertEquals(expectedLoanDate, loan.getLoanDate());
```

```
    }
```

```
    @And("George Red has not returned the item Animal Kingdom")
```

```
    public void georgeRedHasNotReturnedTheItemAnimalKingdom() {
```

```
        assertFalse(loan.isReturned());
```

```
    }
```

```

@When("the system checks for delayed returns")

public void theSystemChecksForDelayedReturns() {

    notificationService.checkForDelayedReturns();

}

@Then("the system sends a notification to George Red about the delayed return of Animal
Kingdom")

public void theSystemSendsANotificationToGeorgeRedAboutTheDelayedReturnOfAnimalKingdom() {

    Notification notification =
notificationService.getNotificationForBorrowerAndItem(borrower, "Animal Kingdom");

    assertNotNull(notification);

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

    assertEquals("Animal Kingdom", notification.getItem().getTitle());

}

// Add any additional methods needed for these steps here, such as calculateDateDaysAgo
}

```

Please note that the actual implementation of the methods in the `NotificationService` class and the additional methods in the step definitions will depend on the specific requirements of your application. The above code is a basic template and may need to be adjusted to fit your needs.

User

Add these additional methods

GitHubCopilot

Sure, here are the additional methods that were mentioned in the step definitions:

```
public class DateUtils {

    public static String calculateDateDaysAgo(int daysAgo) {

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

        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");

        return dateDaysAgo.format(formatter);

    }

    public static String getTodayDate() {

        LocalDate today = LocalDate.now();

        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");

        return today.format(formatter);

    }

}

public class LoanService {

    // existing code...
```

```
public double calculateFine(Borrower borrower, Loan loan) {  
  
    // This is a placeholder implementation. You'll need to replace it with your actual fine  
    calculation logic.  
  
    if (loan.isOverdue()) {  
  
        return 10.0; // Assume a flat fine of 10 for overdue loans  
  
    } else {  
  
        return 0.0;  
  
    }  
  
}  
  
}
```

```
public class NotificationService {  
  
    // existing code...  
  
  
    public void checkForDelayedReturns() {  
  
        // This is a placeholder implementation. You'll need to replace it with your actual logic for  
        checking delayed returns.  
  
        List<Loan> allLoans = findAllLoans();  
  
        for (Loan loan : allLoans) {  
  
            if (loan.isOverdue()) {  
  
                sendNotification(loan.getBorrower(), loan.getItem());  
  
            }  
  
        }  
  
    }  
  
}
```

```
}
```

```
private void sendNotification(Borrower borrower, Item item) {
```

```
    // This is a placeholder implementation. You'll need to replace it with your actual logic for  
    sending notifications.
```

```
    System.out.println("Sending notification to " + borrower.getName() + " about overdue  
    item " + item.getTitle());
```

```
}
```

```
public Notification getNotificationForBorrowerAndItem(Borrower borrower, String  
itemTitle) {
```

```
    // This is a placeholder implementation. You'll need to replace it with your actual logic for  
    retrieving notifications.
```

```
    return new Notification(borrower, itemTitle, "Your loan for " + itemTitle + " is overdue.");
```

```
}
```

```
}
```

Please note that these are placeholder implementations. You'll need to replace them with your actual logic.