Σ' αυτήν τη συνομιλία, παρέδωσα στο LLM τα features του συστήματος και τις κλάσεις domain με τις ιδιότητές τους.. Το Copilot κατάλαβε πολύ καλά την εργασία και άρχισε να δημιουργεί κώδικα αμέσως, χωρίς να χρειαστεί να του ζητήσω ρητά να δημιουργήσει κώδικα για τις κλάσεις domain. Στη συνέχεια, με μερικές επιπλέον μικρές εντολές, δημιούργησε πολύ γρήγορα όλα τα step definitions. Συνολικά, το LLM χρησιμοποίησε πολλές από τις ιδιότητες των κλάσεων που του παρείχα και κατάλαβε πολύ καλύτερα πώς να τα χρησιμοποιήσει σε σύγκριση με το GPT-3.5. Χρησιμοποίησε επίσης τα Services και τα DAOs, αν και τα αποτελέσματα δεν ήταν εξαιρετικά καλύτερα από τις συνομιλίες της φάσης 2, κυρίως λόγω της χρήσης πολλών assertions στα βήματα GIVEN. Στην επόμενη συνομιλία, θα ζητήσω από το LLM να μην χρησιμοποιεί assertions στα βήματα GIVEN και θα δούμε τα αποτελέσματα. Παρά ταύτα, πολλά από τα step definitions ήταν πολύ χρηστικά.

Phase 3. Chat 1

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 with their fields , 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?

GitHub Copilot

As an AI, I don't ask questions but I'm here to assist you. Please provide the Gherkin Scenarios and the details of the domain classes. I'll do my best to help you with the implementation of the Step definitions for each feature in Java.

User

The domain classes are:

1)class "Book" with the attributes:

private ISBN isbn,

private String title,

private String publication,

private int publicationyear,

private Publisher publisher,

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

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

2)class "Borrower" with the attributes:

private int borrowerNo,

private TelephoneNumber telephone,

private EmailAddress emailAddress,

private Address address,

private BorrowerCategory category,

private Person person = new Person(),

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

3)class "BorrowerCategory" with the attributes:

private String description,

private int maxLendingItems,

private int maxLendingDays,

private Money dailyFine

4)class "Item" with the fields:

private int itemNumber = 0,

private Book book,

private ItemState state = ItemState.NEW

5)enum class "ItemState" with no attrbutes outside the enum values

6)class "Loan" with the attributes:

private LocalDate loanDate = SystemDate.now(),

private LocalDate returnDate,

private Borrower borrower,

private Item item

7)class "Person" with the fields:

private String firstName,

private String lastName

In the next message I will provide you with the features/scenarios

GitHub Copilot

Thank you for providing the domain classes. I will wait for the Gherkin Scenarios to proceed with the implementation of the Step definitions in Java.

User

The Gherkin scenarios are :

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

GitHub Copilot

Based on the provided Gherkin scenarios, we can start implementing the Step Definitions for each feature in Java. We will use the Cucumber library for this purpose.

For the first feature "Borrower handling by the system", the step definitions could look like this:

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

import io.cucumber.java.en.Then;

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

```
public class BorrowerHandlingSteps {

    private BorrowerService borrowerService;
    private Borrower borrower;

    @Given("George Red is not registered as a borrower")
    public void georgeRedIsNotRegisteredAsABorrower() {
        borrower = borrowerService.findBorrowerByName("George Red");
        assert borrower == null;
    }

    @When("George Red gets registered in the system with a unique borrower number and his details")
    public void georgeRedGetsRegisteredInTheSystemWithAUniqueBorrowerNumberAndHisDetails() {
        borrower = new Borrower();
        borrower.setBorrowerNo(UniqueNumberGenerator.generate());
        borrower.setPerson(new Person("George", "Red"));
        borrowerService.saveBorrower(borrower);
    }

    @Then("the system successfully stores the borrower's details")
    public void theSystemSuccessfullyStoresTheBorrowersDetails() {
        Borrower savedBorrower = borrowerService.findBorrowerByName("George Red");
        assert savedBorrower != null;
    }

    // Implement the rest of the steps in a similar manner
}
```

For the second feature "Delayed Return Notification", the step definitions could look like this:

```
public class DelayedReturnNotificationSteps {

    private LoanService loanService;
    private EmailService emailService;
    private Borrower borrower;
    private Item item;

    @Given("George Red has borrowed the item Harry Potter")
    public void georgeRedHasBorrowedTheItemHarryPotter() {
        borrower = borrowerService.findBorrowerByName("George Red");
        item = itemService.findItemByName("Harry Potter");
        assert borrower != null && item != null;
    }

    @And("Harry Potter's due date has passed")
    public void harryPottersDueDateHasPassed() {

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

        assert loan != null && loan.getReturnDate().isBefore(LocalDate.now());

    }


    @When("the system executes the delayed return notification process")

    public void theSystemExecutesTheDelayedReturnNotificationProcess() {

        emailService.sendDelayedReturnNotification(borrower, item);

    }


    @Then("George Red receives an email notification for the return of the item")

    public void georgeRedReceivesAnEmailNotificationForTheReturnOfTheItem() {

        // This step might be hard to implement as it involves checking an external system (email)

        // You might need to mock the email service or check the sent emails in a different way

    }


    // Implement the rest of the steps in a similar manner

}
```

For the third feature "Loaning items", the step definitions could look like this:

```
public class LoaningItemsSteps {
```

```
private BorrowerService borrowerService;

private ItemService itemService;

private LoanService loanService;

private Borrower borrower;

private Item item;


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

public void theLibraryHasTheItemHarryPotterAvailable() {

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

   assert item != null && item.getState() == ItemState.AVAILABLE;

}


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

public void georgeRedIsARegisteredBorrower() {

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

   assert borrower != null;

}


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

public void georgeRedBorrowsTheItemHarryPotter() {

   loanService.createLoan(borrower, item);

}
```

```
    @Then("the system successfully loans the item Harry Potter to George Red with a due date set")
    public void theSystemSuccessfullyLoansTheItemHarryPotterToGeorgeRedWithADueDateSet() {
        Loan loan = loanService.findLoanByBorrowerAndItem(borrower, item);
        assert loan != null && loan.getReturnDate() != null;
    }

    // Implement the rest of the steps in a similar manner
}
```

For the last feature "Returning a borrowed item to the library", the step definitions could look like this:

```
public class ReturningBorrowedItemSteps {

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

    @Given("George Red borrowed the item Animal Kingdom 5 days prior to today's date")
    public void georgeRedBorrowedTheItemAnimalKingdom5DaysPriorToTodaysDate() {
        borrower = borrowerService.findBorrowerByName("George Red");
        item = itemService.findItemByName("Animal Kingdom");
        assert borrower != null && item != null;
    }

    @When("the return of Animal Kingdom is processed")
    public void theReturnOfAnimalKingdomIsProcessed() {
        loanService.returnItem(borrower, item);
    }

    @Then("the system marks the state of Animal Kingdom as AVAILABLE")
    public void theSystemMarksTheStateOfAnimalKingdomAsAVAILABLE() {
        assert item.getState() == ItemState.AVAILABLE;
    }

    // Implement the rest of the steps in a similar manner
}
```

Please note that the actual implementation of the services (`BorrowerService`, `ItemService`, `LoanService`, `EmailService`) and the way you check the results of the steps (the assertions) might vary based on your actual application architecture and requirements.

User

Implement all the step definitions implementation code for the feature: Borrower Handling by the system

GitHub Copilot

Sure, here is the implementation of the step definitions for the feature "Borrower handling by the system".

```
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import 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.findBorrowerByName("George Red");
        Assert.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.setBorrowerNo(UniqueNumberGenerator.generate());
        borrower.setPerson(new Person("George", "Red"));
        borrowerService.saveBorrower(borrower);
    }

    @Then("the system successfully stores the borrower's details")
    public void theSystemSuccessfullyStoresTheBorrowersDetails() {
        Borrower savedBorrower = borrowerService.findBorrowerByName("George Red");
        Assert.assertNotNull(savedBorrower);
    }

    @Given("George Red is registered as a borrower")
    public void georgeRedIsRegisteredAsABorrower() {
        borrower = borrowerService.findBorrowerByName("George Red");
        Assert.assertNotNull(borrower);
    }

    @When("the system attempts to register George Red with a unique borrower number and his details")
    public                                                                 void
  theSystemAttemptsToRegisterGeorgeRedWithAUniqueBorrowerNumberAndHisDetails() {
        Borrower newBorrower = new Borrower();
        newBorrower.setBorrowerNo(UniqueNumberGenerator.generate());
        newBorrower.setPerson(new Person("George", "Red"));
        borrowerService.saveBorrower(newBorrower);
    }

    @Then("the system informs that the user already exists")
    public void theSystemInformsThatTheUserAlreadyExists() {
        Borrower existingBorrower = borrowerService.findBorrowerByName("George Red");
        Assert.assertNotNull(existingBorrower);
    }

    @When("George Red updates his borrowing details")
    public void georgeRedUpdatesHisBorrowingDetails() {
        borrower.setAddress(new Address("New Street", "New City", "New Country"));
        borrowerService.updateBorrower(borrower);
    }

    @Then("the system saves the changes")
    public void theSystemSavesTheChanges() {
        Borrower updatedBorrower = borrowerService.findBorrowerByName("George Red");
        Assert.assertEquals("New Street", updatedBorrower.getAddress().getStreet());
```

```
}

@When("George Red tries to update his borrowing details")
public void georgeRedTriesToUpdateHisBorrowingDetails() {
    borrower = new Borrower();
    borrower.setPerson(new Person("George", "Red"));
    borrower.setAddress(new Address("New Street", "New City", "New Country"));
    borrowerService.updateBorrower(borrower);
}

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

@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");
    Assert.assertNull(deletedBorrower);
}

@When("the system attempts to delete George Red's account")
public void theSystemAttemptsToDeleteGeorgeRedsAccount() {
    borrowerService.deleteBorrower(borrower);
}

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

@Given("George Red has pending items")
public void georgeRedHasPendingItems() {
    borrower = borrowerService.findBorrowerByName("George Red");
    Assert.assertTrue(borrower.getLoans().size() > 0);
}

@Then("the system does not remove George Red's details")
public void theSystemDoesNotRemoveGeorgeRedsDetails() {
        Borrower existingBorrower = borrowerService.findBorrowerByName("George Red");
        Assert.assertNotNull(existingBorrower);
    }

    @Then("the system informs about the pending items")
    public void theSystemInformsAboutThePendingItems() {
        Borrower existingBorrower = borrowerService.findBorrowerByName("George Red");
        Assert.assertTrue(existingBorrower.getLoans().size() > 0);
    }
}
```

Please note that the actual implementation of the services (`BorrowerService`) and the way you check the results of the steps (the assertions) might vary based on your actual application architecture and requirements.

User

Now, based on all the above, give me the step definitions implementation code for the feature: Loaning Items

GitHub Copilot

Sure, here is the implementation of the step definitions for the feature "Loaning items".

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

import org.junit.Assert;


public class LoaningItemsSteps {


    private BorrowerService borrowerService = new BorrowerService();

    private ItemService itemService = new ItemService();

    private LoanService loanService = new LoanService();

    private Borrower borrower;

    private Item item;


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

    public void theLibraryHasTheItemHarryPotterAvailable() {

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

        Assert.assertNotNull(item);

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

    }


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

    public void georgeRedIsARegisteredBorrower() {

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

        Assert.assertNotNull(borrower);

    }
    @And("George Red has {int} pending items to be returned")
    public void georgeRedHasPendingItemsToBeReturned(int pendingItems) {
        Assert.assertEquals(pendingItems, borrower.getLoans().size());
    }

    @And("George Red has been assigned a maximum lending limit of {int}")
    public void georgeRedHasBeenAssignedAMaximumLendingLimitOf(int lendingLimit) {
        BorrowerCategory borrowerCategory = borrower.getCategory();
        Assert.assertEquals(lendingLimit, borrowerCategory.getMaxLendingItems());
    }

    @When("George Red borrows the item Harry Potter")
    public void georgeRedBorrowsTheItemHarryPotter() {
        loanService.createLoan(borrower, item);
    }

    @Then("the system successfully loans the item Harry Potter to George Red with a due date
set")
    public void
theSystemSuccessfullyLoansTheItemHarryPotterToGeorgeRedWithADueDateSet() {
        Loan loan = loanService.findLoanByBorrowerAndItem(borrower, item);
        Assert.assertNotNull(loan);
        Assert.assertNotNull(loan.getReturnDate());
    }

    @And("George Red's pending items increase to {int}")
    public void georgeRedsPendingItemsIncreaseTo(int pendingItems) {
        Assert.assertEquals(pendingItems, borrower.getLoans().size());
    }

    @When("George Red tries to borrow both items")
    public void georgeRedTriesToBorrowBothItems() {
        Item item2 = itemService.findItemByName("Moby Dick");
        loanService.createLoan(borrower, item);
        loanService.createLoan(borrower, item2);
    }

    @Then("the system successfully loans the item Harry Potter to George Red with a due date set")
    public void theSystemSuccessfullyLoansTheItemHarryPotterToGeorgeRedWithADueDateSet() {
        Loan loan = loanService.findLoanByBorrowerAndItem(borrower, item);
        Assert.assertNotNull(loan);
        Assert.assertNotNull(loan.getReturnDate());
    }
    @And("the system does not loan Moby Dick to George Red due to the lending limit
reached")
    public void
theSystemDoesNotLoanMobyDickToGeorgeRedDueToTheLendingLimitReached() {
        Loan loan = loanService.findLoanByBorrowerAndItem(borrower,
itemService.findItemByName("Moby Dick"));
        Assert.assertNull(loan);
    }

    @Given("the item Harry Potter is in the library but not in the system")
    public void theItemHarryPotterIsInTheLibraryButNotInTheSystem() {
        item = itemService.findItemByName("Harry Potter");
        Assert.assertNull(item);
    }

    @Then("the system returns an error due to the item's status")
    public void theSystemReturnsAnErrorDueToTheItemsStatus() {
        // This step might be hard to implement as it involves checking an error message
        // You might need to throw an exception in the service and catch it here
    }

    @And("the system withdraws the item Harry Potter")
    public void theSystemWithdrawsTheItemHarryPotter() {
        itemService.withdrawItem("Harry Potter");
        Item withdrawnItem = itemService.findItemByName("Harry Potter");
        Assert.assertNull(withdrawnItem);
    }

    @When("George Red tries to borrow the item Harry Potter")
    public void georgeRedTriesToBorrowTheItemHarryPotter() {
        loanService.createLoan(borrower, item);
    }

    @Then("the system doesn't allow the loan")
    public void theSystemDoesntAllowTheLoan() {
        Loan loan = loanService.findLoanByBorrowerAndItem(borrower, item);
        Assert.assertNull(loan);
    }

    @And("George Red's pending items remain {int}")
    public void georgeRedsPendingItemsRemain(int pendingItems) {
        Assert.assertEquals(pendingItems, borrower.getLoans().size());
    }
}
```

Please note that the actual implementation of the services (`BorrowerService`, `ItemService`, `LoanService`) and the way you check the results of the steps (the assertions) might vary based on your actual application architecture and requirements.

based on the above, give me the step definitions for the feature: Returning a borrowed item to the library

GitHub Copilot

Sure, here is the implementation of the step definitions for the feature "Returning a borrowed item to the library".

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

public class ReturningBorrowedItemSteps {

    private BorrowerService borrowerService = new BorrowerService();
    private ItemService itemService = new ItemService();
    private LoanService loanService = new LoanService();
    private Borrower borrower;
    private Item item;

    @Given("George Red borrowed the item Animal Kingdom {int} days prior to today's date")
    public void georgeRedBorrowedTheItemAnimalKingdomDaysPriorToTodaysDate(int days) {
        borrower = borrowerService.findBorrowerByName("George Red");
        item = itemService.findItemByName("Animal Kingdom");

        Assert.assertNotNull(borrower);

        Assert.assertNotNull(item);

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

        Assert.assertNotNull(loan);

        Assert.assertEquals(LocalDate.now().minusDays(days), loan.getLoanDate());

    }


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

    public void georgeRedHasBeenAssignedMaximumLendingDaysOf(int lendingDays) {

        BorrowerCategory borrowerCategory = borrower.getCategory();

        Assert.assertEquals(lendingDays, borrowerCategory.getMaxLendingDays());

    }


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

    public void theReturnOfAnimalKingdomIsProcessed() {

        loanService.returnItem(borrower, item);

    }


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

    public void theSystemMarksTheStateOfAnimalKingdomAsAVAILABLE() {

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

    }
```

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

public void georgeRedHasOneLessPendingItem() {

    int pendingItems = borrower.getLoans().size();

    Assert.assertEquals(pendingItems, borrower.getLoans().size() - 1);

}


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

public void georgeRedDoesNotPayAFine() {

    // This step might be hard to implement as it involves checking a fine

    // You might need to implement a method in the service to check if a fine was paid

}


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

public void theReturnDateOfTheLoanIsSetToTodaysDate() {

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

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

}


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

public void georgeRedPaysAFineBasedOnTheBorrowerCategory() {

    // This step might be hard to implement as it involves checking a fine

    // You might need to implement a method in the service to check if a fine was paid
    }
}
```

Please note that the actual implementation of the services (`BorrowerService`, `ItemService`, `LoanService`) and the way you check the results of the steps (the assertions) might vary based on your actual application architecture and requirements.

User

Now , give me the step definitions for the feature: Delayed Notification

GitHub Copilot

Sure, here is the implementation of the step definitions for the feature "Delayed Return Notification".

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

public class DelayedReturnNotificationSteps {

    private BorrowerService borrowerService = new BorrowerService();
    private ItemService itemService = new ItemService();
    private LoanService loanService = new LoanService();
private EmailService emailService = new EmailService();

private Borrower borrower;

private Item item;


@Given("George Red has borrowed the item Harry Potter")

public void georgeRedHasBorrowedTheItemHarryPotter() {

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

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

   Assert.assertNotNull(borrower);

   Assert.assertNotNull(item);

}


@And("Harry Potter's due date has passed")

public void harryPottersDueDateHasPassed() {

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

   Assert.assertNotNull(loan);

   Assert.assertTrue(loan.getReturnDate().isBefore(LocalDate.now()));

}


@And("George Red has an email address")

public void georgeRedHasAnEmailAddress() {

   Assert.assertNotNull(borrower.getEmailAddress());

}
```

```
    @When("the system executes the delayed return notification process")
    public void theSystemExecutesTheDelayedReturnNotificationProcess() {
        emailService.sendDelayedReturnNotification(borrower, item);
    }

    @Then("George Red receives an email notification for the return of the item")
    public void georgeRedReceivesAnEmailNotificationForTheReturnOfTheItem() {
        // This step might be hard to implement as it involves checking an external system (email)
        // You might need to mock the email service or check the sent emails in a different way
    }

    @And("George Red does not have an email address")
    public void georgeRedDoesNotHaveAnEmailAddress() {
        Assert.assertNull(borrower.getEmailAddress());
    }

    @Then("George Red does not receive an email notification for the return of the item")
    public void georgeRedDoesNotReceiveAnEmailNotificationForTheReturnOfTheItem() {
        // This step might be hard to implement as it involves checking an external system (email)
        // You might need to mock the email service or check the sent emails in a different way
    }
}
```

Please note that the actual implementation of the services (`BorrowerService`, `ItemService`, `LoanService`, `EmailService`) and the way you check the results of the steps (the assertions) might vary based on your actual application architecture and requirements.