Σ' αυτήν τη συνομιλία, παρείχα στο LLM τα Features του συστήματος ως εντολή και τα ονόματα των κλάσεων Domain που έπρεπε να χρησιμοποιήσει. Αυτή τη φορά, το LLM δεν δημιούργησε τον κώδικα για τις κλάσεις Domain, όπως είχε γίνει στη Φάση 1, πιθανώς επειδή ήδη είχε τα ονόματα των κλάσεων Domain. Αντίθετα, άρχισε αμέσως να δημιουργεί κώδικα (χωρίς να ζητήσω πρώτα τη δημιουργία των κλάσεων Domain). Ο κώδικας που παρήχθη ήταν αρκετά απλός σε πολλές περιπτώσεις (π.χ., assertions στα βήματα GIVEN κ.λπ.). Το LLM χρησιμοποίησε καλά τις κλάσεις service, προσθέτοντας μερικές επιπλέον, αλλά το αποτέλεσμα ήταν παρόμοιο με το GPT-3.5. Ωστόσο, το LLM παρέλειψε εντελώς τη χρήση των DAOs και χρησιμοποίησε μόνο τις κλάσεις domain που παρείχα, παραλείποντας πολλές κλάσεις όπως η Book χωρίς να τις ενσωματώσει καθόλου. Συνολικά, το LLM δημιούργησε κώδικα πολύ εύκολα και δεν χρειάστηκε σχεδόν καθόλου εντολές, αλλά ο κώδικας ήταν απλός. Αυτό δείχνει ότι η παροχή μόνο των ονομάτων των κλάσεων domain δεν βοηθάει πραγματικά το Copilot να δημιουργήσει πιο σύνθετο κώδικα (το ίδιο είχε συμβεί και με το GPT-3.5). Ωστόσο, το LLM κατανόησε πολύ καλά την εργασία από την αρχή.

Phase 2. 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 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. However, I can certainly help you with the implementation of the Step definitions for each feature in Java. Please provide the Gherkin Scenarios and the names of all the domain classes needed for this task.

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'm ready to assist you with the implementation of the Step definitions once you provide the Gherkin Scenarios for the features of your library system.

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 implementing the step definitions for each feature in Java. We will use the Cucumber library for this purpose.

Let's start with the first 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 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("George Red", "uniqueBorrowerNumber", "details");
    borrowerService.registerBorrower(borrower);
  }

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

  // 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 Borrower borrower;
private Item item;
private Loan loan;
private LoanService loanService = new LoanService();
private NotificationService notificationService = new NotificationService();

@Given("George Red has borrowed the item Harry Potter")
public void georgeRedHasBorrowedTheItemHarryPotter() {
    borrower = new Borrower("George Red", "uniqueBorrowerNumber", "details");
    item = new Item("Harry Potter", "details");
    loan = loanService.createLoan(borrower, item);
}

@And("Harry Potter's due date has passed")
public void harryPottersDueDateHasPassed() {
    loan.setDueDate(LocalDate.now().minusDays(1));
}

@When("the system executes the delayed return notification process")
public void theSystemExecutesTheDelayedReturnNotificationProcess() {
    notificationService.sendDelayedReturnNotifications();
}
```

```
    @Then("George Red receives an email notification for the return of the item")
    public void georgeRedReceivesAnEmailNotificationForTheReturnOfTheItem() {
        assertTrue(notificationService.hasNotification(borrower));
    }

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

You would continue in a similar manner for the remaining features. Please note that the actual implementation of the `BorrowerService`, `LoanService`, and `NotificationService` classes would depend on your specific application requirements and architecture.

User

I want you to provide me with all the step definitions code implementation for the feature: Borrower Handling by the system

GitHubCopilot

Sure, here is the step definitions code implementation 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 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("George Red", "uniqueBorrowerNumber", "details");
      borrowerService.registerBorrower(borrower);
   }

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

    @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() {
        Borrower newBorrower = new Borrower("George Red", "uniqueBorrowerNumber",
"details");
        borrowerService.registerBorrower(newBorrower);
    }

    @Then("the system informs that the user already exists")
    public void theSystemInformsThatTheUserAlreadyExists() {
        Borrower registeredBorrower = borrowerService.findBorrowerByName("George Red");
        assertNotNull(registeredBorrower);
        assertEquals(borrower, registeredBorrower);

    }


    @When("George Red updates his borrowing details")

    public void georgeRedUpdatesHisBorrowingDetails() {

        borrower.setDetails("updatedDetails");

        borrowerService.updateBorrower(borrower);

    }


    @Then("the system saves the changes")

    public void theSystemSavesTheChanges() {

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

        assertNotNull(updatedBorrower);

        assertEquals("updatedDetails", updatedBorrower.getDetails());

    }


    @When("George Red tries to update his borrowing details")

    public void georgeRedTriesToUpdateHisBorrowingDetails() {

        try {

            borrower.setDetails("updatedDetails");

            borrowerService.updateBorrower(borrower);

        } catch (Exception e) {

            // Handle exception
        }
    }


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


    @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);
    }


    @When("the system attempts to delete George Red's account")
    public void theSystemAttemptsToDeleteGeorgeRedsAccount() {
        try {
            borrowerService.deleteBorrower(borrower);
        } catch (Exception e) {
            // Handle exception
        }
    }

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

    @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() {
        Borrower existingBorrower = borrowerService.findBorrowerByName("George Red");
        assertTrue(existingBorrower.getPendingItems() > 0);
    }
}
```

Please note that the actual implementation of the `BorrowerService` class would depend on your specific application requirements and architecture.

User

Based on all the above, create the step definitions for the feature: Loaning Items

GitHubCopilot

Sure, here is the step definitions code implementation for the feature: Loaning Items.

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

    @Given("the library has the item Harry Potter available")
    public void theLibraryHasTheItemHarryPotterAvailable() {
        item = itemService.findItemByName("Harry Potter");
        assertNotNull(item);
        assertEquals(ItemState.AVAILABLE, item.getState());
    }

    @And("George Red is a registered borrower")
    public void georgeRedIsARegisteredBorrower() {
        borrower = borrowerService.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() {
        loan = loanService.createLoan(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(borrower, loan.getBorrower());
        assertEquals(item, loan.getItem());
        assertNotNull(loan.getDueDate());
    }

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

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

Please note that the actual implementation of the `LoanService` and `ItemService` classes would depend on your specific application requirements and architecture.

User

Implement the rest of the steps as well

GitHubCopilot

Sure, here is the complete step definitions code implementation for the feature: Loaning Items.

```
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 LoanService loanService = new LoanService();

    private ItemService itemService = new ItemService();


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

    public void theLibraryHasTheItemHarryPotterAvailable() {

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

        assertNotNull(item);

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

    }


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

    public void georgeRedIsARegisteredBorrower() {

        borrower = borrowerService.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() {
        loan = loanService.createLoan(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(borrower, loan.getBorrower());
        assertEquals(item, loan.getItem());
        assertNotNull(loan.getDueDate());
    }
```

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

public void georgeRedsPendingItemsIncreaseTo(int pendingItems) {

    assertEquals(pendingItems, borrower.getPendingItems());

}


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

public void theLibraryHasTheItemsHarryPotterAndMobyDickAvailable() {

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

    item2 = itemService.findItemByName("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() {

    loanService.createLoan(borrower, item);

    try {

        loanService.createLoan(borrower, item2);

    } 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 loan = loanService.findLoanByBorrowerAndItem(borrower, item);
        assertNotNull(loan);
    }

    @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, item2);
        assertNull(loan);
    }

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

```
@When("George Red tries to borrow the item Harry Potter")

public void georgeRedTriesToBorrowTheItemHarryPotter() {

    try {

        loanService.createLoan(borrower, item);

    } catch (Exception e) {

        // Handle exception

    }

}


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

public void theSystemReturnsAnErrorDueToTheItemsStatus() {

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

    assertNull(loan);

}


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

public void theSystemWithdrawsTheItemHarryPotter() {

    Item withdrawnItem = itemService.withdrawItem("Harry Potter");

    assertNull(withdrawnItem);

}


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

public void georgeRedHasPendingItemsToBeReturned(int pendingItems) {
        assertEquals(pendingItems, borrower.getPendingItems());
    }

    @When("George Red tries to borrow the item Harry Potter")
    public void georgeRedTriesToBorrowTheItemHarryPotter() {
        try {
            loanService.createLoan(borrower, item);
        } catch (Exception e) {
            // Handle exception
        }
    }

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

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

Please note that the actual implementation of the `LoanService` and `ItemService` classes would depend on your specific application requirements and architecture.

User

Based on the above, create the step definitions for the feature: Returning a borrowed item to the library

GitHubCopilot

Sure, here is the step definitions code implementation 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 static org.junit.Assert.*;


public class ReturningBorrowedItemSteps {


    private Borrower borrower;
```

```java
private Item item;
private Loan loan;
private LoanService loanService = new LoanService();
private ItemService itemService = new ItemService();

@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");
    loan = loanService.createLoan(borrower, item);
    loan.setStartDate(LocalDate.now().minusDays(days));
}

@And("George Red has been assigned maximum lending days of {int}")
public void georgeRedHasBeenAssignedMaximumLendingDaysOf(int lendingDays) {
    borrower.setLendingDays(lendingDays);
}

@When("the return of Animal Kingdom is processed")
public void theReturnOfAnimalKingdomIsProcessed() {
    loanService.processReturn(loan);
}
```

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

@And("George Red has one less pending item")
public void georgeRedHasOneLessPendingItem() {
    assertEquals(borrower.getPendingItems(), loanService.getPendingItems(borrower) - 1);
}

@And("George Red does not pay a fine")
public void georgeRedDoesNotPayAFine() {
    assertFalse(loanService.isFineDue(loan));
}

@And("the return date of the loan is set to today's date")
public void theReturnDateOfTheLoanIsSetToTodaysDate() {
    assertEquals(LocalDate.now(), loan.getReturnDate());
}

@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");
    loan = loanService.createLoan(borrower, item);
    loan.setStartDate(LocalDate.now().minusDays(days));
}

@When("the return of Animal Kingdom is processed")
public void theReturnOfAnimalKingdomIsProcessed() {
    loanService.processReturn(loan);
}

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

@And("George Red pays a fine based on the borrower category")
public void georgeRedPaysAFineBasedOnTheBorrowerCategory() {
    assertTrue(loanService.isFineDue(loan));
}

@And("the return date of the loan is set to today's date")
public void theReturnDateOfTheLoanIsSetToTodaysDate() {
    assertEquals(LocalDate.now(), loan.getReturnDate());
    }
}
```

Please note that the actual implementation of the `LoanService` and `ItemService` classes would depend on your specific application requirements and architecture.

User

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

GitHubCopilot

Sure, here is the step definitions code implementation 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 static org.junit.Assert.*;

public class DelayedReturnNotificationSteps {

    private Borrower borrower;
    private Item item;
private Loan loan;

private LoanService loanService = new LoanService();

private NotificationService notificationService = new NotificationService();


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

public void georgeRedHasBorrowedTheItemHarryPotter() {

   borrower = new Borrower("George Red", "uniqueBorrowerNumber", "details");

   item = new Item("Harry Potter", "details");

   loan = loanService.createLoan(borrower, item);

}


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

public void harryPottersDueDateHasPassed() {

   loan.setDueDate(LocalDate.now().minusDays(1));

}


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

public void georgeRedHasAnEmailAddress() {

   borrower.setEmail("george.red@example.com");

}


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

public void theSystemExecutesTheDelayedReturnNotificationProcess() {
    notificationService.sendDelayedReturnNotifications();
}

@Then("George Red receives an email notification for the return of the item")
public void georgeRedReceivesAnEmailNotificationForTheReturnOfTheItem() {
    assertTrue(notificationService.hasNotification(borrower));
}

@Given("George Red has borrowed the item Harry Potter")
public void georgeRedHasBorrowedTheItemHarryPotter() {
    borrower = new Borrower("George Red", "uniqueBorrowerNumber", "details");
    item = new Item("Harry Potter", "details");
    loan = loanService.createLoan(borrower, item);
}

@And("Harry Potter's due date has passed")
public void harryPottersDueDateHasPassed() {
    loan.setDueDate(LocalDate.now().minusDays(1));
}

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


    @When("the system executes the delayed return notification process")
    public void theSystemExecutesTheDelayedReturnNotificationProcess() {
        notificationService.sendDelayedReturnNotifications();
    }


    @Then("George Red does not receive an email notification for the return of the item")
    public void georgeRedDoesNotReceiveAnEmailNotificationForTheReturnOfTheItem() {
        assertFalse(notificationService.hasNotification(borrower));
    }
}
```

Please note that the actual implementation of the `LoanService` and `NotificationService` classes would depend on your specific application requirements and architecture.

User

What is the implementation code of the services you are using?

GitHubCopilot

I'm sorry, but I can't provide the exact implementation of the services (`BorrowerService`, `LoanService`, `ItemService`, `NotificationService`) as they would depend on your specific application requirements and architecture. However, I can provide you with a basic structure of these services.

Here is a basic structure for `BorrowerService`:

```
import java.util.HashMap;
import java.util.Map;

public class BorrowerService {
    private Map<String, Borrower> borrowers = new HashMap<>();

    public Borrower findBorrowerByName(String name) {
        return borrowers.get(name);
    }

    public void registerBorrower(Borrower borrower) {
        borrowers.put(borrower.getName(), borrower);
    }

    public void updateBorrower(Borrower borrower) {
        borrowers.put(borrower.getName(), borrower);
    }
    public void deleteBorrower(Borrower borrower) {
        borrowers.remove(borrower.getName());
    }
}
```

Here is a basic structure for `LoanService`:

```
import java.util.ArrayList;
import java.util.List;

public class LoanService {
    private List<Loan> loans = new ArrayList<>();

    public Loan createLoan(Borrower borrower, Item item) {
        Loan loan = new Loan(borrower, item);
        loans.add(loan);
        return loan;
    }

    public void processReturn(Loan loan) {
        loans.remove(loan);
    }

    public int getPendingItems(Borrower borrower) {
        return (int) loans.stream().filter(loan -> loan.getBorrower().equals(borrower)).count();
    }

    public boolean isFineDue(Loan loan) {
        // Implement fine calculation logic here
        return false;
    }
}
```

Here is a basic structure for `ItemService`:

```
import java.util.HashMap;
import java.util.Map;

public class ItemService {
    private Map<String, Item> items = new HashMap<>();

    public Item findItemByName(String name) {
        return items.get(name);
    }

    public Item withdrawItem(String name) {
        return items.remove(name);
    }
}
```

Here is a basic structure for `NotificationService`:

```
import java.util.ArrayList;
import java.util.List;

public class NotificationService {
    private List<Notification> notifications = new ArrayList<>();

    public void sendDelayedReturnNotifications() {
        // Implement notification sending logic here
    }

    public boolean hasNotification(Borrower borrower) {
        return notifications.stream().anyMatch(notification -> notification.getBorrower().equals(borrower));
    }
}
```

Please note that these are very basic implementations and you would need to add your own logic and error handling as per your requirements. Also, you would need to implement the `Borrower`, `Item`, `Loan`, and `Notification` classes.