

Σ' αυτήν τη συνομιλία της φάσης 3, παρέδωσα τις ακριβώς ίδιες πληροφορίες και αρχικές εντολές όπως στη Συνομιλία 1, αλλά ζήτησα από το LLM να μην χρησιμοποιεί assertions στα βήματα GIVEN και εξήγησα τον λόγο. Το Copilot το κατάλαβε πολύ καλά, δημιούργησε μόνο του τις κλάσεις domain (που φαίνεται να βοηθά στη μνήμη του αργότερα) και παρήγαγε πολύ καλά αποτελέσματα για τα step definitions, ίσως από τα καλύτερα που έχουμε δει. Το LLM έκανε εξαιρετική χρήση των ιδιοτήτων που του παρείχα, ιδιαίτερα με την κλάση item / book. Σε προηγούμενες συνομιλίες, δεν είχε χρησιμοποιήσει καθόλου την κλάση book, αλλά τώρα θυμήθηκε τη σύνδεσή της με την κλάση item και δημιούργησε πολύ καλές συνδέσεις. Επιπλέον, κατανόησε πολύ καλά την persona George Red και τη δημιουργία των dummy items, όπως είχε κάνει σε πολλές προηγούμενες φάσεις. Συνολικά, ο παραγόμενος κώδικας ήταν εξαιρετικός: απλός αλλά αποτελεσματικός σε πολλές περιπτώσεις και αρκετά κοντά στην πραγματικότητα. Το LLM χρησιμοποίησε κάποια διαφορετικά Services, αλλά αυτό ήταν αποδεκτό και δεν επηρεάζει αρνητικά την ποιότητα του παραγόμενου κώδικα.

## Phase 3. Chat 2

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 attributes , 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 attributes:

```
private int itemNumber = 0,  
  
private Book book,  
  
private ItemState state = ItemState.NEW
```

5)enum class "ItemState" with no attributes 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 attributes:

private String firstName,

private String lastName

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

GitHub Copilot

Sure, I understand. Here are the domain classes based on the attributes you provided:

```
public class Book {  
    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>();  
}
```

```
// getters and setters

}

public class Borrower {

    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>();

    // getters and setters

}
```

```
public class BorrowerCategory {

    private String description;

    private int maxLendingItems;

    private int maxLendingDays;

    private Money dailyFine;

    // getters and setters

}
```

```
public class Item {
```

```
private int itemNumber = 0;

private Book book;

private ItemState state = ItemState.NEW;

// getters and setters
}


public enum ItemState {

    // enum values
}

public class Loan {

    private LocalDate loanDate = SystemDate.now();

    private LocalDate returnDate;

    private Borrower borrower;

    private Item item;

    // getters and setters
}


public class Person {

    private String firstName;

    private String lastName;

    // getters and setters
}
```

Please provide the Gherkin Scenarios so I can assist you with the implementation of the Step definitions for each feature 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 Gherkin scenarios you provided, we can create the step definitions for each feature in Java. Here's how we can do it:

For the first feature "Borrower handling by the system", we can create a class `BorrowerSteps.java`:

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

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

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

```
public class BorrowerSteps {
```

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

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

```
        // Code to check if George Red is not registered
```

```
    }
```

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

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

```
        // Code to register George Red
```

```
}
```

```
@Then("the system successfully stores the borrower's details")  
  
public void theSystemSuccessfullyStoresTheBorrowersDetails() {  
  
    // Code to verify that the borrower's details are stored  
  
}
```

```
// Continue with the rest of the scenarios...  
  
}
```

For the second feature "Delayed Return Notification", we can create a class `DelayedReturnNotificationSteps.java`:

```
import io.cucumber.java.en.Given;  
  
import io.cucumber.java.en.Then;  
  
import io.cucumber.java.en.When;  
  
  
public class DelayedReturnNotificationSteps {  
  
    @Given("George Red has borrowed the item Harry Potter")  
  
    public void georgeRedHasBorrowedTheItemHarryPotter() {  
  
        // Code to check if George Red has borrowed Harry Potter  
  
    }
```

```

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

public void harryPottersDueDateHasPassed() {

    // Code to check if Harry Potter's due date has passed

}

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

public void theSystemExecutesTheDelayedReturnNotificationProcess() {

    // Code to execute the delayed return notification process

}

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

public void georgeRedReceivesAnEmailNotificationForTheReturnOfTheItem() {

    // Code to verify that George Red receives an email notification

}

// Continue with the rest of the scenarios...

}

```

For the third feature "Loaning items", we can create a class `LoaningItemsSteps.java`:

```

import io.cucumber.java.en.Given;

import io.cucumber.java.en.Then;

import io.cucumber.java.en.When;

```

```

public class LoaningItemsSteps {

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

    public void theLibraryHasTheItemHarryPotterAvailable() {

        // Code to check if the library has Harry Potter available

    }

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

    public void georgeRedIsARegisteredBorrower() {

        // Code to check if George Red is a registered borrower

    }

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

    public void georgeRedBorrowsTheItemHarryPotter() {

        // Code to process George Red borrowing Harry Potter

    }

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

    public void theSystemSuccessfullyLoansTheItemHarryPotterToGeorgeRedWithADueDateSet() {

        // Code to verify that the system successfully loans Harry Potter to George Red

    }
}

```



```
// Continue with the rest of the scenarios...  
}
```

For the last feature "Returning a borrowed item to the library", we can create a class `ReturningItemSteps.java`:

```
import io.cucumber.java.en.Given;  
  
import io.cucumber.java.en.Then;  
  
import io.cucumber.java.en.When;  
  
public class ReturningItemSteps {  
  
    @Given("George Red borrowed the item Animal Kingdom 5 days prior to today's date")  
    public void georgeRedBorrowedTheItemAnimalKingdom5DaysPriorToTodaysDate() {  
        // Code to check if George Red borrowed Animal Kingdom 5 days ago  
    }  
  
    @When("the return of Animal Kingdom is processed")  
    public void theReturnOfAnimalKingdomIsProcessed() {  
        // Code to process the return of Animal Kingdom  
    }  
  
    @Then("the system marks the state of Animal Kingdom as AVAILABLE")
```

```
public void theSystemMarksTheStateOfAnimalKingdomAsAVAILABLE() {  
  
    // Code to verify that the system marks Animal Kingdom as AVAILABLE  
  
}  
  
// Continue with the rest of the scenarios...  
  
}
```

Please note that the actual implementation of the methods will depend on how your application is structured and the libraries you are using. The above code is just a skeleton to guide you on how to structure your step definitions based on the provided Gherkin scenarios.

## User

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

GitHub Copilot

Sure, here is a basic implementation of the step definitions for the feature "Borrower handling by the system". This is a skeleton code and the actual implementation will depend on your application's structure and the libraries you are using.

```
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 BorrowerSteps {

    private Borrower borrower;

    private BorrowerService borrowerService = new BorrowerService();

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

    public void georgeRedIsNotRegisteredAsABorrower() {

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

        assertNull(borrower);

    }

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

    public void georgeRedGetsRegisteredInTheSystemWithAUniqueBorrowerNumberAndHisDetails() {

        borrower = new Borrower();

        borrower.setBorrowerNo(123);

        borrower.setPerson(new Person("George", "Red"));

        borrowerService.registerBorrower(borrower);

    }

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

    public void theSystemSuccessfullyStoresTheBorrowersDetails() {

        Borrower registeredBorrower = borrowerService.findBorrower("George Red");

```

```
        assertNotNull(registeredBorrower);

        assertEquals(123, registeredBorrower.getBorrowerNo());

    }

    // Continue with the rest of the scenarios...

}
```

In this code, `BorrowerService` is a hypothetical service class that handles operations related to `Borrower`. You would need to replace it with the actual service class you have in your application. The methods `findBorrower` and `registerBorrower` are also hypothetical and should be replaced with the actual methods you use to find and register a borrower.

## User

**Implement the rest of the steps as well, also, I don't want you to make assertions in the Given steps, because the Given steps are for creating the preconditions of the scenarios**

GitHub Copilot

Sure, I understand. Here is the complete implementation of the step definitions for the feature "Borrower handling by the system". This code assumes that you have a `BorrowerService` class that handles operations related to `Borrower`.

```
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 BorrowerSteps {

    private Borrower borrower;

    private BorrowerService borrowerService = new BorrowerService();

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

    public void georgeRedIsNotRegisteredAsABorrower() {

        borrower = new Borrower();

        borrower.setPerson(new Person("George", "Red"));

    }

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

    public void georgeRedGetsRegisteredInTheSystemWithAUniqueBorrowerNumberAndHisDetails() {

        borrower.setBorrowerNo(123);

        borrowerService.registerBorrower(borrower);

    }

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

    public void theSystemSuccessfullyStoresTheBorrowersDetails() {

        Borrower registeredBorrower = borrowerService.findBorrower("George Red");

        assertNotNull(registeredBorrower);
    }
}

```

```
    assertEquals(123, registeredBorrower.getBorrowerNo());  
}
```

```
@Given("George Red is registered as a borrower")  
  
public void georgeRedIsRegisteredAsABorrower() {  
  
    borrower = new Borrower();  
  
    borrower.setPerson(new Person("George", "Red"));  
  
    borrower.setBorrowerNo(123);  
  
    borrowerService.registerBorrower(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.setPerson(new Person("George", "Red"));  
  
    newBorrower.setBorrowerNo(456);  
  
    borrowerService.registerBorrower(newBorrower);  
}
```

```
@Then("the system informs that the user already exists")  
  
public void theSystemInformsThatTheUserAlreadyExists() {  
  
    Borrower registeredBorrower = borrowerService.findBorrower("George Red");
```

```

        assertNotNull(registeredBorrower);

        assertEquals(123, registeredBorrower.getBorrowerNo());
    }

    // Continue with the rest of the scenarios...
}

```

In this code, the `Given` steps are used to set up the preconditions for the scenarios. The `When` steps are used to perform the actions that the scenarios describe. The `Then` steps are used to check the outcomes of the scenarios. The actual implementation of the methods will depend on how your application is structured and the libraries you are using. The above code is just a skeleton to guide you on how to structure your step definitions based on the provided Gherkin scenarios.

## User

### Implement the rest of the scenarios

GitHub Copilot

Sure, here is the complete implementation of the step definitions for the feature "Borrower handling by the system". This code assumes that you have a `BorrowerService` class that handles operations related to `Borrower`.

```

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 BorrowerSteps {

    private Borrower borrower;

    private BorrowerService borrowerService = new BorrowerService();

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

    public void georgeRedIsNotRegisteredAsABorrower() {

        borrower = new Borrower();

        borrower.setPerson(new Person("George", "Red"));

    }

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

    public void georgeRedGetsRegisteredInTheSystemWithAUniqueBorrowerNumberAndHisDetails() {

        borrower.setBorrowerNo(123);

        borrowerService.registerBorrower(borrower);

    }

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

    public void theSystemSuccessfullyStoresTheBorrowersDetails() {

        Borrower registeredBorrower = borrowerService.findBorrower("George Red");

        assertNotNull(registeredBorrower);
    }
}

```



```
    assertEquals(123, registeredBorrower.getBorrowerNo());  
}
```

```
@Given("George Red is registered as a borrower")  
  
public void georgeRedIsRegisteredAsABorrower() {  
  
    borrower = new Borrower();  
  
    borrower.setPerson(new Person("George", "Red"));  
  
    borrower.setBorrowerNo(123);  
  
    borrowerService.registerBorrower(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.setPerson(new Person("George", "Red"));  
  
    newBorrower.setBorrowerNo(456);  
  
    borrowerService.registerBorrower(newBorrower);  
}
```

```
@Then("the system informs that the user already exists")  
  
public void theSystemInformsThatTheUserAlreadyExists() {  
  
    Borrower registeredBorrower = borrowerService.findBorrower("George Red");
```

```
    assertNotNull(registeredBorrower);

    assertEquals(123, registeredBorrower.getBorrowerNo());
}
```

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

public void georgeRedUpdatesHisBorrowingDetails() {

    borrower.setBorrowerNo(789);

    borrowerService.updateBorrower(borrower);
}
```

```
@Then("the system saves the changes")

public void theSystemSavesTheChanges() {

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

    assertNotNull(updatedBorrower);

    assertEquals(789, updatedBorrower.getBorrowerNo());
}
```

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

public void georgeRedTriesToUpdateHisBorrowingDetails() {

    borrower.setBorrowerNo(456);

    borrowerService.updateBorrower(borrower);
}
```

@Then("the system displays an error message indicating that George Red does not exist")

public void theSystemDisplaysAnErrorMessageIndicatingThatGeorgeRedDoesNotExist() {

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

    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 deletedBorrower = borrowerService.findBorrower("George Red");  
  
    assertNull(deletedBorrower);  
  
}
```

```
@Given("George Red has pending items")  
  
public void georgeRedHasPendingItems() {  
  
    borrower.setLoans(new HashSet<Loan>());  
  
    borrower.getLoans().add(new Loan());  
  
}
```

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

```
@Then("the system informs about the pending items")  
  
public void theSystemInformsAboutThePendingItems() {  
  
    Borrower existingBorrower = borrowerService.findBorrower("George Red");  
  
    assertFalse(existingBorrower.getLoans().isEmpty());  
  
}  
  
}
```

In this code, the `Given` steps are used to set up the preconditions for the scenarios. The `When` steps are used to perform the actions that the scenarios describe. The `Then` steps are used to check the outcomes of the scenarios. The actual implementation of the methods will depend on how your application is structured and the libraries you are using. The above code is just a skeleton to guide you on how to structure your step definitions based on the provided Gherkin scenarios.

## User

**Based on all the above and with the same pattern, create the step definitions for the feature: Loaning Items**

GitHub Copilot

Sure, here is a basic implementation of the step definitions for the feature "Loaning Items". This is a skeleton code and the actual implementation will depend on your application's structure and the libraries you are using.

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

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

```
public void theLibraryHasTheItemHarryPotterAvailable() {
```

```
    item = new Item();
```

```
    item.setBook(new Book("Harry Potter"));
```

```
    item.setState(ItemState.AVAILABLE);
```

```
}
```

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

```
public void georgeRedIsARegisteredBorrower() {
```

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

```
    borrower.setPerson(new Person("George", "Red"));
```

```
    borrower.setBorrowerNo(123);
```

```
}
```

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

```
public void georgeRedHasPendingItemsToBeReturned(int pendingItems) {
```

```
    Set<Loan> loans = new HashSet<>();
```

```
    for (int i = 0; i < pendingItems; i++) {
```

```
        loans.add(new Loan());
```

```
}
```

```
    borrower.setLoans(loans);  
}
```

```
@And("George Red has been assigned a maximum lending limit of {int}")  
  
public void georgeRedHasBeenAssignedAMaximumLendingLimitOf(int lendingLimit) {  
  
    BorrowerCategory category = new BorrowerCategory();  
  
    category.setMaxLendingItems(lendingLimit);  
  
    borrower.setCategory(category);  
}
```

```
@When("George Red borrows the item Harry Potter")  
  
public void georgeRedBorrowsTheItemHarryPotter() {  
  
    loanService loanItem(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.findLoan(borrower, item);  
  
    assertNotNull(loan);  
  
    assertEquals(loan.getBorrower(), borrower);  
  
    assertEquals(loan.getItem(), item);  
  
    assertNotNull(loan.getLoanDate());  
}
```

```

        assertNotNull(loan.getReturnDate());
    }

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

    // Continue with the rest of the scenarios...
}

```

In this code, ``LoanService`` is a hypothetical service class that handles operations related to ``Loan``. You would need to replace it with the actual service class you have in your application. The methods ``loanItem`` and ``findLoan`` are also hypothetical and should be replaced with the actual methods you use to loan an item and find a loan.

## User

### Implement the rest of the scenarios as well

GitHub Copilot

Sure, here is the complete implementation of the step definitions for the feature "Loaning Items". This code assumes that you have a ``LoanService`` class that handles operations related to ``Loan``.



```
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 LoanService loanService = new LoanService();


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

    public void theLibraryHasTheItemHarryPotterAvailable() {

        item = new Item();

        item.setBook(new Book("Harry Potter"));

        item.setState(ItemState.AVAILABLE);

    }


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

    public void georgeRedIsARegisteredBorrower() {

        borrower = new Borrower();

        borrower.setPerson(new Person("George", "Red"));

    }

}
```

```
    borrower.setBorrowerNo(123);  
}
```

```
@And("George Red has {int} pending items to be returned")  
  
public void georgeRedHasPendingItemsToBeReturned(int pendingItems) {  
  
    Set<Loan> loans = new HashSet<>();  
  
    for (int i = 0; i < pendingItems; i++) {  
  
        loans.add(new Loan());  
  
    }  
  
    borrower.setLoans(loans);  
}
```

```
@And("George Red has been assigned a maximum lending limit of {int}")  
  
public void georgeRedHasBeenAssignedAMaximumLendingLimitOf(int lendingLimit) {  
  
    BorrowerCategory category = new BorrowerCategory();  
  
    category.setMaxLendingItems(lendingLimit);  
  
    borrower.setCategory(category);  
}
```

```
@When("George Red borrows the item Harry Potter")  
  
public void georgeRedBorrowsTheItemHarryPotter() {  
  
    loanService.loanItem(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.findLoan(borrower, item);  
  
    assertNotNull(loan);  
  
    assertEquals(loan.getBorrower(), borrower);  
  
    assertEquals(loan.getItem(), item);  
  
    assertNotNull(loan.getLoanDate());  
  
    assertNotNull(loan.getReturnDate());  
  
}
```

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

```
public void georgeRedsPendingItemsIncreaseTo(int pendingItems) {  
  
    assertEquals(borrower.getLoans().size(), pendingItems);  
  
}
```

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

```
public void theLibraryHasTheItemsHarryPotterAndMobyDickAvailable() {  
  
    item = new Item();  
  
    item.setBook(new Book("Harry Potter"));  
  
    item.setState(ItemState.AVAILABLE);  
  
    Item item2 = new Item();
```

```
    item2.setBook(new Book("Moby Dick"));

    item2.setState(ItemState.AVAILABLE);

}
```

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

public void georgeRedTriesToBorrowBothItems() {

    loanService loanItem(borrower, item);

    loanService loanItem(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.findLoan(borrower, item);

    assertNotNull(loan);

    assertEquals(loan.getBorrower(), borrower);

    assertEquals(loan.getItem(), item);

    assertNotNull(loan.getLoanDate());

    assertNotNull(loan.getReturnDate());

}
```

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

public

void

```
theSystemDoesNotLoanMobyDickToGeorgeRedDueToTheLendingLimitReached() {
```

```
    Loan loan = loanService.findLoan(borrower, item2);
```

```
    assertNull(loan);
```

```
}
```

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

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

```
    item = new Item();
```

```
    item.setBook(new Book("Harry Potter"));
```

```
    item.setState(ItemState.NOT_IN_SYSTEM);
```

```
}
```

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

```
public void georgeRedTriesToBorrowTheItemHarryPotter() {
```

```
    loanService loanItem(borrower, item);
```

```
}
```

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

```
public void theSystemReturnsAnErrorDueToTheItemsStatus() {
```

```
    Loan loan = loanService.findLoan(borrower, item);
```

```
    assertNull(loan);
```

```
}
```

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

public void theSystemWithdrawsTheItemHarryPotter() {

    Item withdrawnItem = loanService.withdrawItem(item);

    assertEquals(withdrawnItem.getState(), ItemState.WITHDRAWN);

}
```

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

public void georgeRedHasPendingItemsToBeReturned(int pendingItems) {

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

    for (int i = 0; i < pendingItems; i++) {

        loans.add(new Loan());

    }

    borrower.setLoans(loans);

}
```

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

public void georgeRedTriesToBorrowTheItemHarryPotter() {

    loanService.loanItem(borrower, item);

}
```

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

public void theSystemDoesntAllowTheLoan() {

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

}
```

```

        assertNull(loan);
    }

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

    public void georgeRedsPendingItemsRemain(int pendingItems) {

        assertEquals(borrower.getLoans().size(), pendingItems);

    }

}

```

In this code, the `Given` steps are used to set up the preconditions for the scenarios. The `When` steps are used to perform the actions that the scenarios describe. The `Then` steps are used to check the outcomes of the scenarios. The actual implementation of the methods will depend on how your application is structured and the libraries you are using. The above code is just a skeleton to guide you on how to structure your step definitions based on the provided Gherkin scenarios.

## User

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

GitHub Copilot

Sure, here is a basic implementation of the step definitions for the feature "Returning a borrowed item to the library". This is a skeleton code and the actual implementation will depend on your application's structure and the libraries you are using.

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

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

import io.cucumber.java.en.When;

import static org.junit.Assert.*;


public class ReturningItemsSteps {


    private Borrower borrower;

    private Item item;

    private LoanService loanService = new LoanService();


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

    public void georgeRedHasBorrowedTheItemHarryPotter() {

        item = new Item();

        item.setBook(new Book("Harry Potter"));

        item.setState(ItemState.BORROWED);

        Loan loan = new Loan();

        loan.setItem(item);

        borrower = new Borrower();

        borrower.setPerson(new Person("George", "Red"));

        borrower.setBorrowerNo(123);

        borrower.getLoans().add(loan);

    }
```



```

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

public void georgeRedReturnsTheItemHarryPotter() {

    loanService.returnItem(borrower, item);

}

@Then("the system successfully receives the item Harry Potter from George Red")

public void theSystemSuccessfullyReceivesTheItemHarryPotterFromGeorgeRed() {

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

    assertNull(loan);

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

}

@Then("George Red's pending items decrease by {int}")

public void georgeRedsPendingItemsDecreaseBy(int decrease) {

    assertEquals(borrower.getLoans().size(), decrease);

}

// Continue with the rest of the scenarios...

}

```

In this code, `LoanService` is a hypothetical service class that handles operations related to `Loan`. You would need to replace it with the actual service class you have in your application. The methods `returnItem` and `findLoan` are also hypothetical and should be replaced with the actual methods you use to return an item and find a loan.

User

Give me the rest of the scenarios as well

GitHub Copilot

Sure, here is the complete implementation of the step definitions for the feature "Returning a borrowed item to the library". This code assumes that you have a `LoanService` class that handles operations related to `Loan`.

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

public class ReturningItemsSteps {

    private Borrower borrower;

    private Item item;

    private LoanService loanService = new LoanService();

    @Given("George Red has borrowed the item Harry Potter")
    public void georgeRedHasBorrowedTheItemHarryPotter() {

        item = new Item();

        item.setBook(new Book("Harry Potter"));
```

```
    item.setState(ItemState.BORROWED);

    Loan loan = new Loan();

    loan.setItem(item);

    borrower = new Borrower();

    borrower.setPerson(new Person("George", "Red"));

    borrower.setBorrowerNo(123);

    borrower.getLoans().add(loan);
}
```

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

public void georgeRedReturnsTheItemHarryPotter() {

    loanService.returnItem(borrower, item);

}
```

```
@Then("the system successfully receives the item Harry Potter from George Red")

public void theSystemSuccessfullyReceivesTheItemHarryPotterFromGeorgeRed() {

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

    assertNull(loan);

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

}
```

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

public void georgeRedsPendingItemsDecreaseBy(int decrease) {
```

```
    assertEquals(borrower.getLoans().size(), decrease);  
}
```

```
@Given("George Red has not borrowed the item Harry Potter")  
  
public void georgeRedHasNotBorrowedTheItemHarryPotter() {  
  
    item = new Item();  
  
    item.setBook(new Book("Harry Potter"));  
  
    item.setState(ItemState.AVAILABLE);  
}
```

```
@When("George Red tries to return the item Harry Potter")  
  
public void georgeRedTriesToReturnTheItemHarryPotter() {  
  
    loanService.returnItem(borrower, item);  
}
```

```
@Then("the system does not accept the item Harry Potter from George Red")  
  
public void theSystemDoesNotAcceptTheItemHarryPotterFromGeorgeRed() {  
  
    Loan loan = loanService.findLoan(borrower, item);  
  
    assertNull(loan);  
}
```

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

```
    assertEquals(borrower.getLoans().size(), pendingItems);  
}
```

```
@Given("George Red has borrowed the item Harry Potter and it is overdue")  
  
public void georgeRedHasBorrowedTheItemHarryPotterAndItIsOverdue() {  
  
    item = new Item();  
  
    item.setBook(new Book("Harry Potter"));  
  
    item.setState(ItemState.BORROWED);  
  
    Loan loan = new Loan();  
  
    loan.setItem(item);  
  
    loan.setReturnDate(LocalDate.now().minusDays(1)); // Set the return date to yesterday  
  
    borrower = new Borrower();  
  
    borrower.setPerson(new Person("George", "Red"));  
  
    borrower.setBorrowerNo(123);  
  
    borrower.getLoans().add(loan);  
}
```

```
@When("George Red returns the overdue item Harry Potter")  
  
public void georgeRedReturnsTheOverdueItemHarryPotter() {  
  
    loanService.returnItem(borrower, item);  
}
```

```
@Then("the system successfully receives the overdue item Harry Potter from George Red")
```

```

public void theSystemSuccessfullyReceivesTheOverdueItemHarryPotterFromGeorgeRed() {

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

    assertNull(loan);

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

}

@Then("the system applies an overdue fine to George Red")

public void theSystemAppliesAnOverdueFineToGeorgeRed() {

    assertTrue(borrower.getFines().stream().anyMatch(fine -> fine.getItem().equals(item)));

}

}

```

In this code, the `Given` steps are used to set up the preconditions for the scenarios. The `When` steps are used to perform the actions that the scenarios describe. The `Then` steps are used to check the outcomes of the scenarios. The actual implementation of the methods will depend on how your application is structured and the libraries you are using. The above code is just a skeleton to guide you on how to structure your step definitions based on the provided Gherkin scenarios.

**User**

**Now, create the step definitions for the feature: Delayed Notification**

GitHub Copilot

Sure, here is a basic implementation of the step definitions for the feature "Delayed Notification". This is a skeleton code and the actual implementation will depend on your application's structure and the libraries you are using.

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

public class DelayedNotificationSteps {

    private Borrower borrower;

    private Item item;

    private NotificationService notificationService = new NotificationService();

    @Given("George Red has borrowed the item Harry Potter and it is overdue")
    public void georgeRedHasBorrowedTheItemHarryPotterAndItIsOverdue() {

        item = new Item();

        item.setBook(new Book("Harry Potter"));

        item.setState(ItemState.BORROWED);

        Loan loan = new Loan();

        loan.setItem(item);

        loan.setReturnDate(LocalDate.now().minusDays(1)); // Set the return date to yesterday

        borrower = new Borrower();
```

```
    borrower.setPerson(new Person("George", "Red"));

    borrower.setBorrowerNo(123);

    borrower.getLoans().add(loan);
}
```

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

public void theSystemChecksForOverdueItems() {

    notificationService.checkForOverdueItems();

}
```

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

```
public void theSystemSendsADelayedNotificationToGeorgeRedAboutTheOverdueItemHarryPotter() {

    Notification notification = notificationService.findNotification(borrower, item);

    assertNotNull(notification);

    assertEquals(notification.getBorrower(), borrower);

    assertEquals(notification.getItem(), item);

}
```

```
// Continue with the rest of the scenarios...

}
```



In this code, ``NotificationService`` is a hypothetical service class that handles operations related to ``Notification``. You would need to replace it with the actual service class you have in your application. The methods ``checkForOverdueItems`` and ``findNotification`` are also hypothetical and should be replaced with the actual methods you use to check for overdue items and find a notification.