

Σ' αυτήν τη συνομιλία, παρείχα στο LLM όλες τις ίδιες πληροφορίες όπως στις άλλες συνομιλίες της φάσης 3. Ωστόσο, ζήτησα να δημιουργήσει και τον κώδικα για τα Services και τα DAOs μαζί με τον κώδικα για τις κλάσεις domain (σε άλλες συνομιλίες, είχα ζητήσει μόνο τον κώδικα για τις κλάσεις domain). Κάτι τέτοιο είχα κάνει επίσης στη φάση 2, Συνομιλία 1 και στη φάση 2, Συνομιλία 4. Τα αποτελέσματα ήταν πολύ καλά για τον κώδικα των κλάσεων domain, καθώς και για τα DAOs. Ωστόσο, για τα Services, το LLM δημιούργησε μερικές παραπάνω κλάσεις από όσες ήταν απαραίτητες, αν και οι κλάσεις αυτές ήταν ακόμα καλές. Στη συνέχεια, το LLM άρχισε αμέσως να δημιουργεί κώδικα (όπως συνήθως κάνει όταν του ζητώ πρώτα να δημιουργήσει τον κώδικα για τις κλάσεις domain), και τα αποτελέσματα ήταν αρκετά καλά σε σύγκριση με τις προηγούμενες συνομιλίες. Ειδικότερα, το LLM κατάλαβε ότι ο "George Red" είναι μια persona και δεν τον προσδιόρισε ως κανονική έκφραση, αλλά δημιούργησε μια μεταβλητή με αυτό το όνομα. Επίσης, χρησιμοποίησε την κλάση Item όπως χρειαζόταν (αυτή τη φορά ξέχασε την κλάση Book, ενώ σε άλλες συνομιλίες είχε ξεχάσει την κλάση Item) και χρησιμοποίησε τις κλάσεις EmailService για τη λειτουργία ειδοποίησης καθυστέρησης. Συνολικά, τα αποτελέσματα αυτής της συνομιλίας ήταν πολύ καλά και χρήσιμα με βάση τον τρέχοντα κώδικα, με κάποιες μικρές αλλαγές. Φαίνεται ότι η υπενθύμιση στο LLM να δημιουργήσει τον κώδικα για τα DAOs και τα Services το βοηθάει να τα χρησιμοποιεί περισσότερο, σε σύγκριση με το να μην αναφέρεται τίποτα (σε άλλες συνομιλίες, απλά ζητούσα από το LLM να τα χρησιμοποιεί στην αρχιτεκτονική του στην πρώτη εντολή).

## Phase 3.Chat 3

User

**Act as a software developer. I have a library system application written in java and I have created some Gherkin Scenarios based on different features of the library application. I will provide you with these features/scenarios and the name of all the domain classes 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?**

ChatGPT

Sure, I understand the requirement. Please provide me with the features/scenarios along with the domain classes and their attributes, and we can proceed with implementing the Step definitions for each feature. If you have any specific preferences or constraints regarding the architecture or implementation details, feel free to let me know.

## 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**

**ChatGPT**

Great, thanks for providing the domain classes. Please go ahead and provide the features/scenarios, and I'll assist you in implementing the Step definitions for each feature.

**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**

**I want you to firstly create all the domain classes and the implementation code for them, as well as the DAO classes and the services**

ChatGPT

To implement the Step definitions for each feature, we'll first need to create the domain classes, DAO classes, and service classes as per your architecture requirements. Let's start by creating the domain classes and then proceed with the DAO and service classes.

Here's the implementation code for the domain classes:

```
import java.time.LocalDate;
```

```
import java.util.HashSet;

import java.util.Set;


public class Book {

    private ISBN isbn;

    private String title;

    private String publication;

    private int publicationYear;

    private Publisher publisher;

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

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


    // Constructor, getters, 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<>();

}
```

```
    // Constructor, getters, setters  
}
```

```
public class BorrowerCategory {  
  
    private String description;  
  
    private int maxLendingItems;  
  
    private int maxLendingDays;  
  
    private Money dailyFine;
```

```
    // Constructor, getters, setters  
}
```

```
public class Item {  
  
    private int itemNumber = 0;  
  
    private Book book;  
  
    private ItemState state = ItemState.NEW;
```

```
    // Constructor, getters, setters  
}
```

```
public enum ItemState {  
  
    NEW,
```

```
// Add more states if necessary  
  
}  
  
public class Loan {  
  
    private LocalDate loanDate = LocalDate.now();  
  
    private LocalDate returnDate;  
  
    private Borrower borrower;  
  
    private Item item;  
  
    // Constructor, getters, setters  
  
}
```

```
public class Person {  
  
    private String firstName;  
  
    private String lastName;  
  
    // Constructor, getters, setters  
  
}
```

Next, let's create the Data Access Object (DAO) classes for each domain class:

```
public interface BookDAO {  
  
    void save(Book book);  
  
    void update(Book book);  
  
}
```

```
void delete(Book book);

Book getByISBN(ISBN isbn);

// Other methods if needed
}
```

```
public interface BorrowerDAO {

    void save(Borrower borrower);

    void update(Borrower borrower);

    void delete(Borrower borrower);

    Borrower getByBorrowerNo(int borrowerNo);

    // Other methods if needed
}
```

```
public interface BorrowerCategoryDAO {

    void save(BorrowerCategory category);

    void update(BorrowerCategory category);

    void delete(BorrowerCategory category);

    // Other methods if needed
}
```

```
public interface ItemDAO {

    void save(Item item);

    void update(Item item);
```

```
void delete(Item item);

Item getByItemNumber(int itemNumber);

// Other methods if needed
}
```

```
public interface LoanDAO {

    void save(Loan loan);

    void update(Loan loan);

    void delete(Loan loan);

    // Other methods if needed
}
```

Now, let's create the service classes:

```
public class BookService {

    private BookDAO bookDAO;

    public BookService(BookDAO bookDAO) {

        this.bookDAO = bookDAO;

    }

    // Implement service methods
}
```

```
public class BorrowerService {  
  
    private BorrowerDAO borrowerDAO;  
  
    public BorrowerService(BorrowerDAO borrowerDAO) {  
  
        this.borrowerDAO = borrowerDAO;  
  
    }  
  
    // Implement service methods  
}
```

```
public class BorrowerCategoryService {  
  
    private BorrowerCategoryDAO borrowerCategoryDAO;  
  
    public BorrowerCategoryService(BorrowerCategoryDAO borrowerCategoryDAO) {  
  
        this.borrowerCategoryDAO = borrowerCategoryDAO;  
  
    }  
  
    // Implement service methods  
}
```

```
public class ItemService {  
  
    private ItemDAO itemDAO;
```



```
public ItemService(ItemDAO itemDAO) {  
  
    this.itemDAO = itemDAO;  
  
}  
  
// Implement service methods  
}
```

```
public class LoanService {  
  
    private LoanDAO loanDAO;  
  
    public LoanService(LoanDAO loanDAO) {  
  
        this.loanDAO = loanDAO;  
  
    }  
  
    // Implement service methods  
}
```

Once we have these classes in place, we can proceed with implementing the Step definitions for each feature using these services to interact with the domain objects. Let me know if you need further assistance with that!

**User**

**Based on the domain classes I provided you, create some extra classes you think will be used**

ChatGPT

Certainly! Considering the domain classes provided, we can identify some additional classes that might be useful in the library system:

1. **\*\*Publisher\*\***: A class representing the publisher of a book.

```
public class Publisher {  
  
    private String name;  
  
    // Other attributes, constructors, getters, setters  
  
}
```

2. **\*\*Author\*\***: A class representing the author of a book.

```
public class Author {  
  
    private String name;  
  
    // Other attributes, constructors, getters, setters  
  
}
```

3. **\*\*TelephoneNumber\*\***: A class representing the telephone number of a borrower.

```
public class TelephoneNumber {  
  
    private String phoneNumber;  
  
    // Other attributes, constructors, getters, setters  
  
}
```

4. **\*\*EmailAddress\*\***: A class representing the email address of a borrower.

```
public class EmailAddress {
```

```
private String email;

// Other attributes, constructors, getters, setters
}
```

5. **\*\*Address\*\***: A class representing the address of a borrower.

```
public class Address {

    private String street;

    private String city;

    private String state;

    private String zipCode;

    // Other attributes, constructors, getters, setters
}
```

6. **\*\*Money\*\***: A class representing money with a specific currency.

```
public class Money {

    private double amount;

    private String currency;

    // Other attributes, constructors, getters, setters
}
```

7. **\*\*ISBN\*\***: A class representing the International Standard Book Number of a book.

```
public class ISBN {  
  
    private String number;  
  
    // Other attributes, constructors, getters, setters  
  
}
```

These classes represent various entities and value objects that are integral to the library system domain. They provide necessary abstractions for handling book details, borrower information, and other related entities. Let me know if you need further clarification or assistance with anything else!

## User

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

ChatGPT

To implement the Step definitions for the "Borrower handling by the system" feature, we'll create a class containing methods corresponding to each step defined in the Gherkin scenarios. These methods will interact with the service classes to perform the required actions on borrowers. Here's how you can structure the Step definitions:

```
import cucumber.api.java.en.Given;  
  
import cucumber.api.java.en.When;  
  
import cucumber.api.java.en.Then;  
  
  
public class BorrowerSteps {
```

```
private BorrowerService borrowerService;
```

```
private Borrower registeredBorrower;
```

```
// Inject the BorrowerService instance via constructor or setter
```

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

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

```
    // Perform any necessary setup
```

```
    registeredBorrower = null;
```

```
}
```

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

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

```
    // Assuming borrower details are provided in the step, create a new borrower
```

```
    Borrower georgeRed = new Borrower();
```

```
    // Set borrower details
```

```
    borrowerService.registerBorrower(georgeRed);
```

```
    registeredBorrower = georgeRed;
```

```
}
```

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

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

```
// Verify that borrower details are stored in the system

assert borrowerService.getBorrowerById(registeredBorrower.getBorrowerNo()) != null;

}
```

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

public void georgeRedIsRegisteredAsABorrower() {

    // Assuming George Red is already registered as a borrower

    registeredBorrower = new Borrower();

    borrowerService.registerBorrower(registeredBorrower);

}
```

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

```
public void theSystemAttemptsToRegisterGeorgeRedWithAUniqueBorrowerNumberAndHisDetails() {

    // Attempt to register George Red again

    try {

        borrowerService.registerBorrower(registeredBorrower);

    } catch (Exception e) {

        // Handle the exception or assert an error message

    }

}
```

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

```
public void theSystemInformsThatTheUserAlreadyExists() {  
  
    // Verify that system informs about existing user  
  
    // Assert error message or handle the exception  
  
}  
  
// Implement other Step definitions similarly for other scenarios  
  
}
```

In this code:

- Each Step definition method corresponds to a step in the Gherkin scenario.
- The `BorrowerService` is used to interact with the domain logic related to borrowers, such as registering a borrower.
- Assertions or exception handling is included to verify the behavior of the system.

You'll need to inject the appropriate service instances into the `BorrowerSteps` class to make it work. Let me know if you need further clarification or assistance!

**User**

**Create all the step definitions code for this feature because you have skipped some**

ChatGPT

Certainly! Let's complete the Step definitions for all the scenarios in the "Borrower handling by the system" feature:

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

import cucumber.api.java.en.When;

import cucumber.api.java.en.Then;


public class BorrowerSteps {


    private BorrowerService borrowerService;

    private Borrower registeredBorrower;

    private String errorMessage;


    // Inject the BorrowerService instance via constructor or setter


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

    public void georgeRedIsNotRegisteredAsABorrower() {

        // Perform any necessary setup

        registeredBorrower = null;

    }


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

    public void georgeRedGetsRegisteredInTheSystemWithAUniqueBorrowerNumberAndHisDetails() {

        // Assuming borrower details are provided in the step, create a new borrower

        Borrower georgeRed = new Borrower();
    }
}
```



```
// Set borrower details

borrowerService.registerBorrower(georgeRed);

registeredBorrower = georgeRed;
}
```

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

public void theSystemSuccessfullyStoresTheBorrowersDetails() {

    // Verify that borrower details are stored in the system

    assert borrowerService.getBorrowerById(registeredBorrower.getBorrowerNo()) != null;
}
```

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

public void georgeRedIsRegisteredAsABorrower() {

    // Assuming George Red is already registered as a borrower

    registeredBorrower = new Borrower();

    borrowerService.registerBorrower(registeredBorrower);
}
```

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

```
public void theSystemAttemptsToRegisterGeorgeRedWithAUniqueBorrowerNumberAndHisDetails() {

    // Attempt to register George Red again

    try {
```

```
        borrowerService.registerBorrower(registeredBorrower);

    } catch (Exception e) {

        errorMessage = e.getMessage();

    }

}
```

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

public void theSystemInformsThatTheUserAlreadyExists() {

    // Verify that system informs about existing user

    assert errorMessage != null && errorMessage.equals("User already exists");

}
```

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

public void georgeRedUpdatesHisBorrowingDetails() {

    // Update George Red's borrowing details

    registeredBorrower.setAddress(new Address("New Street", "New City", "New State",
"New Zip"));

    borrowerService.updateBorrower(registeredBorrower);

}
```

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

public void theSystemSavesTheChanges() {

    // Verify that system successfully saves the changes
```

```
        assert  
        borrowerService.getBorrowerById(registeredBorrower.getBorrowerNo()).getAddress().getStreet().equals("New Street");  
    }  
  

```

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

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

```
    // Perform any necessary setup
```

```
    registeredBorrower = null;
```

```
}
```

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

```
public void georgeRedTriesToUpdateHisBorrowingDetails() {
```

```
    // Attempt to update borrowing details for non-registered borrower
```

```
    try {
```

```
        borrowerService.updateBorrower(registeredBorrower);
```

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

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

```
    }
```

```
}
```

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

```
public void theSystemDisplaysAnErrorMessageIndicatingThatGeorgeRedDoesNotExist() {
```

```
    // Verify that system displays appropriate error message
```

```
    assert errorMessage != null && errorMessage.equals("User does not exist");  
}
```

```
@Given("^George Red is registered as a borrower$")  
  
public void georgeRedIsRegisteredAsABorrower() {  
  
    // Assuming George Red is already registered as a borrower  
  
    registeredBorrower = new Borrower();  
  
    borrowerService.registerBorrower(registeredBorrower);  
}
```

```
@When("^the system deletes George Red's account$")  
  
public void theSystemDeletesGeorgeRedsAccount() {  
  
    // Delete George Red's account  
  
    borrowerService.deleteBorrower(registeredBorrower);  
}
```

```
@Then("^the system removes George Red's details$")  
  
public void theSystemRemovesGeorgeRedsDetails() {  
  
    // Verify that George Red's details are removed from the system  
  
    assert borrowerService.getBorrowerById(registeredBorrower.getBorrowerNo()) == null;  
}
```

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

```
public void georgeRedIsNotRegisteredAsABorrower() {  
  
    // Perform any necessary setup  
  
    registeredBorrower = null;  
  
}
```

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

```
public void theSystemAttemptsToDeleteGeorgeRedsAccount() {  
  
    // Attempt to delete account of non-registered borrower  
  
    try {  
  
        borrowerService.deleteBorrower(registeredBorrower);  
  
    } catch (Exception e) {  
  
        errorMessage = e.getMessage();  
  
    }  
  
}
```

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

```
public void theSystemInformsThatTheBorrowerDoesNotExist() {  
  
    // Verify that system informs about non-existing borrower  
  
    assert errorMessage != null && errorMessage.equals("Borrower does not exist");  
  
}
```

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

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

```
// Assuming George Red is already registered as a borrower

registeredBorrower = new Borrower();

borrowerService.registerBorrower(registeredBorrower);

}
```

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

public void georgeRedHasPendingItems() {

    // Assuming George Red has pending items

    registeredBorrower.setLoans(new HashSet<>());

    registeredBorrower.getLoans().add(new Loan());

}
```

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

public void theSystemDoesNotRemoveGeorgeRedsDetails() {

    // Verify that George Red's details are not removed

    assert borrowerService.getBorrowerById(registeredBorrower.getBorrowerNo()) != null;

}
```

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

public void theSystemInformsAboutThePendingItems() {

    // Verify that system informs about pending items

    assert errorMessage != null && errorMessage.equals("Borrower has pending items");

}
```

```
}
```

In this code:

- Each Step definition method corresponds to a step in the Gherkin scenario.
- The `BorrowerService` is used to interact with the domain logic related to borrowers, such as registering, updating, and deleting borrowers.
- Assertions or exception handling is included to verify the behavior of the system.

Make sure to adjust the implementation according to your actual service methods and error handling strategy. Let me know if you need further assistance!

**User**

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

ChatGPT

Sure, let's create the Step definitions for the "Delayed Return Notification" feature:

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

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

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

```
public class DelayedReturnNotificationSteps {
```

```
    private BorrowerService borrowerService;
```

```
    private ItemService itemService;
```

```
private EmailService emailService;
```

```
private Borrower borrower;
```

```
private Item item;
```

```
// Inject the required services via constructor or setter
```

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

```
public void georgeRedHasBorrowedTheItemHarryPotter() {
```

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

```
    // Assuming borrower is registered and has borrowed Harry Potter
```

```
    borrowerService.registerBorrower(borrower);
```

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

```
    // Assuming Harry Potter item is available and assigned to George Red
```

```
    itemService.assignItemToBorrower(item, borrower);
```

```
}
```

```
@Given("^Harry Potter's due date has passed$")
```

```
public void harryPottersDueDateHasPassed() {
```

```
    // Set the due date to a past date
```

```
    item.setDueDate(LocalDate.now().minusDays(1));
```

```
    itemService.updateItem(item);
```

```
}
```



```
@Given("^George Red has an email address$")
```

```
public void georgeRedHasAnEmailAddress() {
```

```
    // Assuming George Red has provided an email address
```

```
    borrower.setEmailAddress(new EmailAddress("george.red@example.com"));
```

```
    borrowerService.updateBorrower(borrower);
```

```
}
```

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

```
public void theSystemExecutesTheDelayedReturnNotificationProcess() {
```

```
    // Execute delayed return notification process
```

```
    emailService.sendNotificationEmail(borrower.getEmailAddress(), "Reminder: Return  
Harry Potter");
```

```
}
```

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

```
public void georgeRedReceivesAnEmailNotificationForTheReturnOfTheItem() {
```

```
    // Verify that George Red receives the email notification
```

```
    assert emailService.isEmailSentSuccessfully();
```

```
}
```

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

```
public void georgeRedHasBorrowedTheItemHarryPotter() {
```

```
    // Similar to the first Given step, but assuming George Red has no email address
```

```
}
```

```
@Given("^Harry Potter's due date has passed$")
```

```
public void harryPottersDueDateHasPassed() {
```

```
    // Similar to the previous Given step
```

```
}
```

```
@Given("^George Red does not have an email address$")
```

```
public void georgeRedDoesNotHaveAnEmailAddress() {
```

```
    // Assuming George Red does not have an email address
```

```
    borrower.setEmailAddress(null);
```

```
    borrowerService.updateBorrower(borrower);
```

```
}
```

```
@Then("^George Red does not receive an email notification for the return of the item$")
```

```
public void georgeRedDoesNotReceiveAnEmailNotificationForTheReturnOfTheItem() {
```

```
    // Verify that George Red does not receive the email notification
```

```
    assert !emailService.isEmailSentSuccessfully();
```

```
}
```

```
}
```

In this code:

- Each Step definition method corresponds to a step in the Gherkin scenario.

- The `BorrowerService`, `ItemService`, and `EmailService` are used to interact with the domain logic related to borrowers, items, and email notifications.
- Assertions are included to verify the behavior of the system.

Ensure to adjust the implementation according to your actual service methods and error handling strategy. Let me know if you need further assistance!

**User**

**Based on the above, create the step definitions for the feature: Loaning items**

ChatGPT

Sure, let's create the Step definitions for the "Loaning items" feature:

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

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

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

```
public class LoaningItemsSteps {
```

```
    private BorrowerService borrowerService;
```

```
    private ItemService itemService;
```

```
    private LoanService loanService;
```

```
    private Borrower borrower;
```

```
    private Item item;
```

```
    // Inject the required services via constructor or setter
```

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

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

```
    // Assuming Harry Potter item is available in the library
```

```
    item = new Item("Harry Potter");
```

```
    itemService.addItem(item);
```

```
}
```

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

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

```
    // Assuming George Red is a registered borrower
```

```
    borrower = new Borrower("George Red");
```

```
    borrowerService.registerBorrower(borrower);
```

```
}
```

```
@Given("^George Red has (\\d+) pending items to be returned$")
```

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

```
    // Assuming George Red has pending items as per the given count
```

```
    borrower.setPendingItems(pendingItems);
```

```
    borrowerService.updateBorrower(borrower);
```

```
}
```

```
@Given("^George Red has been assigned a maximum lending limit of (\\d+)$")
```

```
public void georgeRedHasBeenAssignedAMaximumLendingLimitOf(int maxLendingLimit) {  
  
    // Assuming George Red's maximum lending limit is set  
  
    borrower.setMaxLendingLimit(maxLendingLimit);  
  
    borrowerService.updateBorrower(borrower);  
  
}
```

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

```
public void georgeRedBorrowsTheItemHarryPotter() {  
  
    // George Red borrows the item Harry Potter  
  
    Loan loan = new Loan(borrower, item);  
  
    loanService.processLoan(loan);  
  
}
```

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

```
public void theSystemSuccessfullyLoansTheItemHarryPotterToGeorgeRedWithADueDateSet() {  
  
    // Verify that the item is loaned to George Red with a due date set  
  
    assert loanService.isLoanProcessedSuccessfully();  
  
}
```

```
@Then("^George Red's pending items increase to (\\d+)$")
```

```
public void georgeRedSPendingItemsIncreaseTo(int newPendingItems) {  
  
    // Verify that George Red's pending items are updated
```

```
        assert borrowerService.getBorrowerById(borrower.getId()).getPendingItems() ==
newPendingItems;

    }
```

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

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

```
    // George Red tries to borrow the item Harry Potter
```

```
    try {
```

```
        Loan loan = new Loan(borrower, item);
```

```
        loanService.processLoan(loan);
```

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

```
        // Handle the exception or assert an error message
```

```
    }
```

```
}
```

```
@Then("^the system does not loan the item Harry Potter to George Red due to the lending
limit reached$")
```

```
public void theSystemDoesNotLoanTheItemHarryPotterToGeorgeRedDueToTheLendingLimitReached() {
```

```
    // Verify that the system does not loan the item to George Red
```

```
    assert !loanService.isLoanProcessedSuccessfully();
```

```
}
```

```
// Implement other Step definitions similarly for other scenarios
```

```
}
```

In this code:

- Each Step definition method corresponds to a step in the Gherkin scenario.
- The `BorrowerService`, `ItemService`, and `LoanService` are used to interact with the domain logic related to borrowers, items, and loans.
- Assertions or exception handling is included to verify the behavior of the system.

Ensure to adjust the implementation according to your actual service methods and error handling strategy. Let me know if you need further assistance!

## User

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

ChatGPT

Certainly! Here are the Step definitions for the "Returning a borrowed item to the library" feature:

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

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

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

```
public class ReturningItemSteps {
```

```
private BorrowerService borrowerService;
```

```
private ItemService itemService;
```

```
private LoanService loanService;
```

```
private Borrower borrower;
```

```
private Item item;
```

```
private Loan loan;
```

```
// Inject the required services via constructor or setter
```

```
@Given("^George Red borrowed the item Animal Kingdom (\\d+) days prior to today's date$")
```

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

```
    borrower = new Borrower("George Red");
```

```
    borrowerService.registerBorrower(borrower);
```

```
    item = new Item("Animal Kingdom");
```

```
    itemService.addItem(item);
```

```
    loan = new Loan(borrower, item);
```

```
    loan.setLoanDate(LocalDate.now().minusDays(days));
```

```
    loanService.processLoan(loan);
```

```
}
```

```
@Given("^George Red has been assigned maximum lending days of (\\d+)$")
```



```
public void georgeRedHasBeenAssignedMaximumLendingDaysOf(int maxLendingDays) {  
  
    borrower.setMaxLendingDays(maxLendingDays);  
  
    borrowerService.updateBorrower(borrower);  
  
}
```

```
@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() {  
  
    assert itemService.getItemById(item.getId()).getState() == ItemState.AVAILABLE;  
  
}
```

```
@Then("^George Red has one less pending item$")  
  
public void georgeRedHasOneLessPendingItem() {  
  
    assert borrowerService.getBorrowerById(borrower.getId()).getPendingItems() == 0;  
  
}
```

```
@Then("^George Red does not pay a fine$")  
  
public void georgeRedDoesNotPayAFine() {  
  
    // Assert that George Red does not pay a fine
```

```
// You might need to add some logic to check if George Red pays a fine based on specific conditions
```

```
}
```

```
@Given("^George Red borrowed the item Animal Kingdom (\\d+) days prior to today's date$")
```

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

```
    // Similar to the previous Given step
```

```
}
```

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

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

```
    // Process the return of Animal Kingdom
```

```
    // Assuming George Red returns the item late
```

```
    loanService.processReturnLate(loan);
```

```
}
```

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

```
public void theSystemMarksTheStateOfAnimalKingdomAsAVAILABLE() {
```

```
    // Verify that the state of Animal Kingdom is marked as AVAILABLE
```

```
    assert itemService.getItemById(item.getId()).getState() == ItemState.AVAILABLE;
```

```
}
```

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

```
public void georgeRedPaysAFineBasedOnTheBorrowerCategory() {  
  
    // Assert that George Red pays a fine based on borrower category  
  
    // You might need to add some logic to check if George Red pays a fine based on specific  
conditions  
  
    }  
  
}
```

In this code:

- Each Step definition method corresponds to a step in the Gherkin scenario.
- The `BorrowerService`, `ItemService`, and `LoanService` are used to interact with the domain logic related to borrowers, items, and loans.
- Assertions are included to verify the behavior of the system.

Adjust the implementation according to your actual service methods and error handling strategy. Let me know if you need further assistance!