

Στη συνομιλία αυτή, παρέθεσα στο LLM τις ίδιες ακριβώς πληροφορίες όπως στη Συνομιλία 1, αλλά ήθελα να του ζητήσω να δημιουργήσει πρώτα τον κώδικα για το Domain (κάτι που είχε κάνει αυτόματα σε όλες τις προηγούμενες περιπτώσεις). Παρ' όλα αυτά, το LLM δημιούργησε τον κώδικα Domain χωρίς να του δώσω τα features, χρησιμοποιώντας μόνο τα ονόματα των κλάσεων Domain. Στη συνέχεια, του παρείχα τα features και ζήτησα να δημιουργήσει ξανά τον επικαιροποιημένο κώδικα Domain. Συνολικά, ο κώδικας των κλάσεων Domain ήταν κοντά στον κώδικα του συστήματος, αλλά περιείχε πολλά επιπλέον χαρακτηριστικά. Το LLM χρησιμοποίησε μόνο ένα Service, συγχωνεύοντας όλες τις κλάσεις Services σε αυτό (αντί να δημιουργήσει περισσότερα Services, όπως θα ήταν πιο σωστό). Εντυπωσιακά, για πρώτη φορά το LLM δημιούργησε κώδικα για κάθε Step Definition όλων των features ταυτόχρονα (σε μία μόνο απάντηση) χωρίς να παραλείψει τίποτα, κάτι που είναι εξαιρετικό και πολύ αποδοτικό χρονικά. Συνολικά, ο κώδικας που παρείχε δεν ήταν καλύτερος από τον κώδικα της Συνομιλίας 1, αλλά περιλάμβανε πολλές λεπτομέρειες στην υλοποίησή του και έδειξε βαθιά κατανόηση του κώδικα και των απαιτήσεων. Ενδέχεται, με περισσότερες πληροφορίες, να παραχθούν ακόμα καλύτερα αποτελέσματα.

Phase 2. 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 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?

ChatGPT

I understand the structure and requirements for your library system application. Before we proceed with the step definitions, let's clarify a few points:

1. ****Domain Classes****: I'll need the names and attributes of the domain classes you have.
2. ****Data Access Objects (DAOs)****: I'll assume basic CRUD operations for these. If you have specific methods or requirements for these, please let me know.

3. ****Service Classes****: These will contain the business logic. Please specify any particular services or methods that are essential for the scenarios.
4. ****Gherkin Scenarios****: Please provide the Gherkin scenarios for each feature.

Once I have this information, I can proceed to provide the implementation code of the step definitions for each feature in Java.

Do you have any specific frameworks or libraries in mind for the implementation (e.g., JUnit, Mockito for testing, Hibernate for ORM)? Let me know if there are any additional details or preferences I should consider.

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

ChatGPT

Great, thanks for the information. Before you provide the features, I'll outline a basic structure for the domain classes, DAOs, and services, which we can refine based on your specific needs and the Gherkin scenarios you'll provide.

Domain Classes

1. ****Book****

```
```java
```

```
public class Book {

 private Long id;

 private String title;
```

```
private String author;

private String isbn;

private String publisher;

private Date publishedDate;

private ItemState state;

// Getters and Setters

}

...

```

## 2. **\*\*Borrower\*\***

```
```java

public class Borrower {

    private Long id;

    private Person person;

    private BorrowerCategory category;


// Getters and Setters

}

...

```

3. ****BorrowerCategory****

```
```java

```

```
public class BorrowerCategory {

 private Long id;

 private String name;

 private int maxLoans;

 // Getters and Setters

}
...
```

#### 4. **\*\*Item\*\***

```
```java  
  
public class Item {  
  
    private Long id;  
  
    private Book book;  
  
    private ItemState state;  
  
    // Getters and Setters  
  
}  
...
```

5. ****ItemState****

```
```java  

public enum ItemState {
```

```
 AVAILABLE,
 BORROWED,
 RESERVED,
 LOST
}
...
```

#### 6. **\*\*Loan\*\***

```
```java  
  
public class Loan {  
  
    private Long id;  
  
    private Borrower borrower;  
  
    private Item item;  
  
    private Date loanDate;  
  
    private Date dueDate;  
  
    private Date returnDate;  
  
  
    // Getters and Setters  
  
}  
...
```

7. ****Person****

```
```java
```

```
public class Person {

 private Long id;

 private String firstName;

 private String lastName;

 private String email;

 private String phoneNumber;
```

```
 // Getters and Setters

}
...
```

### ### DAO Interfaces

#### 1. **\*\*BookDAO\*\***

```
```java  
  
public interface BookDAO {  
  
    void save(Book book);  
  
    void delete(Book book);  
  
    Book findById(Long id);  
  
    List<Book> findAll();  
  
}  
...
```

2. ****BorrowerDAO****

```
```java

public interface BorrowerDAO {

 void save(Borrower borrower);

 void delete(Borrower borrower);

 Borrower findById(Long id);

 List<Borrower> findAll();

}

```
```

3. ****ItemDAO****

```
```java

public interface ItemDAO {

 void save(Item item);

 void delete(Item item);

 Item findById(Long id);

 List<Item> findAll();

}

```
```

4. ****LoanDAO****

```
```java

public interface LoanDAO {
```

```
void save(Loan loan);

void delete(Loan loan);

Loan findById(Long id);

List<Loan> findAll();

}

...

```

### ### Service Interfaces

#### 1. **\*\*LibraryService\*\***

```
```java

public interface LibraryService {

    void addBook(Book book);

    void removeBook(Book book);

    Book getBook(Long id);

    List<Book> getAllBooks();

    void borrowItem(Long borrowerId, Long itemId);

    void returnItem(Long borrowerId, Long itemId);

}

...

```

Service Implementations

1. ****LibraryServiceImpl****

```
``java

public class LibraryServiceImpl implements LibraryService {

    private BookDAO bookDAO;

    private BorrowerDAO borrowerDAO;

    private ItemDAO itemDAO;

    private LoanDAO loanDAO;

    // Constructor with DAOs injection

    public LibraryServiceImpl(BookDAO bookDAO, BorrowerDAO borrowerDAO, ItemDAO
itemDAO, LoanDAO loanDAO) {

        this.bookDAO = bookDAO;

        this.borrowerDAO = borrowerDAO;

        this.itemDAO = itemDAO;

        this.loanDAO = loanDAO;

    }

    @Override

    public void addBook(Book book) {

        bookDAO.save(book);

    }

    @Override
```

```
public void removeBook(Book book) {  
  
    bookDAO.delete(book);  
  
}
```

@Override

```
public Book getBook(Long id) {  
  
    return bookDAO.findById(id);  
  
}
```

@Override

```
public List<Book> getAllBooks() {  
  
    return bookDAO.findAll();  
  
}
```

@Override

```
public void borrowItem(Long borrowerId, Long itemId) {  
  
    Borrower borrower = borrowerDAO.findById(borrowerId);  
  
    Item item = itemDAO.findById(itemId);  
  
    if (item.getState() == ItemState.AVAILABLE) {  
  
        item.setState(ItemState.BORROWED);  
  
        Loan loan = new Loan();  
  
        loan.setBorrower(borrower);  
  
        loan.setItem(item);  
  
    }  
  
}
```

```

        loan.setLoanDate(new Date());

        // Assume 2 weeks loan period

        loan.setDueDate(new Date(System.currentTimeMillis() + 1209600000L));

        loanDAO.save(loan);
    }
}

```

@Override

```

public void returnItem(Long borrowerId, Long itemId) {

    Item item = itemDAO.findById(itemId);

    if (item.getState() == ItemState.BORROWED) {

        item.setState(ItemState.AVAILABLE);

        Loan loan = loanDAO.findById(itemId); // Simplified for example

        loan.setReturnDate(new Date());

        loanDAO.save(loan);

    }

}

}

...

```

Step Definitions Placeholder

```java

```
public class LibrarySteps {

 private LibraryService libraryService;

 private Book book;

 private Borrower borrower;

 private Item item;

 private Loan loan;

 public LibrarySteps() {

 // Initialize DAOs with mock implementations or actual implementations

 BookDAO bookDAO = new BookDAOImpl();

 BorrowerDAO borrowerDAO = new BorrowerDAOImpl();

 ItemDAO itemDAO = new ItemDAOImpl();

 LoanDAO loanDAO = new LoanDAOImpl();

 libraryService = new LibraryServiceImpl(bookDAO, borrowerDAO, itemDAO, loanDAO);

 }

 // Step definitions will go here

}

...

```

Now, please provide the Gherkin scenarios for each feature, and I'll proceed with the detailed step definitions in Java.

**User**

**The first feature is :**

**Feature: Borrower handling by the system**

**The system can register a new person, modify their credentials or delete their account**

**User Story: As a librarian,**

**I want to efficiently manage the loaning process of books to registered borrowers**

**So that borrowers can easily borrow books from the library.**

**Scenario: Registering a new borrower**

**This scenario describes the process of registering a new borrower in the library system**

**Given George Red is not registered as a borrower**

**When George Red gets registered in the system with a unique borrower number and his details**

**Then the system successfully stores the borrower's details**

**Scenario: Borrower trying to register has registered before**

**This scenario describes what happens when the system tries to register a new borrower who has already registered before**

**Given George Red is registered as a borrower**

**When the system attempts to register George Red with a unique borrower number and his details**

**Then the system informs that the user already exists**

**Scenario: Updating the borrower's details when he is registered**

**This scenario describes the process of updating the details of a borrower who has already registered before**

**Given George Red is registered as a borrower**

**When George Red updates his borrowing details**

**Then the system saves the changes**

**Scenario: Updating the borrower's details when he hasn't registered**

**This scenario describes what happens when the system tries to update the details of a borrower who has not registered before**

**Given George Red is not registered as a borrower**

**When George Red tries to update his borrowing details**

**Then the system displays an error message indicating that George Red does not exist**

**Scenario: Deleting a borrower**

**This scenario describes the process of deleting the account of a borrower who has already registered**

**Given George Red is registered as a borrower**

**When the system deletes George Red's account**

**Then the system removes George Red's details**

**Scenario: Deleting a borrower when he hasn't registered**

**This scenario describes what happens when the system tries to delete the account of a borrower who has not registered before**

**Given George Red is not registered as a borrower**

**When the system attempts to delete George Red's account**

**Then the system informs that the borrower does not exist**

### **Scenario: Handling unreturned items of books**

**This scenario describes what happens when the system tries to delete the account of a borrower who has pending items to return**

**Given George Red is registered as a borrower**

**And George Red has pending items**

**When the system attempts to delete George Red's account**

**Then the system does not remove George Red's details**

**And the system informs about the pending items**

**The second feature is :**

### **Feature: Delayed Return Notification**

#### **User Story**

**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

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**

**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

### 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

### 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

Update the domain/services/daos code and provide it again based on the features above

ChatGPT

Based on the provided features and scenarios, I'll update the domain classes, DAOs, and service implementations to accommodate the requirements. I'll also provide a skeleton for the step definitions.

### ### Updated Domain Classes

#### 1. **\*\*Book\*\***

```
```java

public class Book {

    private Long id;

    private String title;

    private String author;

    private String isbn;

    private String publisher;

    private Date publishedDate;

    private ItemState state;


    // Getters and Setters

}

```
```

#### 2. **\*\*Borrower\*\***

```
```java

public class Borrower {
```

```
private Long id;

private Person person;

private BorrowerCategory category;

private List<Loan> loans;


// Getters and Setters

}

...

```

3. ****BorrowerCategory****

```
```java

public class BorrowerCategory {

 private Long id;

 private String name;

 private int maxLoans;

 private double fineRate;

 // Getters and Setters

}

...

```

### 4. **\*\*Item\*\***

```
```java

```

```
public class Item {  
  
    private Long id;  
  
    private Book book;  
  
    private ItemState state;  
  
    // Getters and Setters  
  
}  
...
```

5. ****ItemState****

```
```java  

public enum ItemState {

 AVAILABLE,

 BORROWED,

 RESERVED,

 LOST

}
...
```

#### 6. **\*\*Loan\*\***

```
```java  
  
public class Loan {  
  
    private Long id;
```

```
    private Borrower borrower;

    private Item item;

    private Date loanDate;

    private Date dueDate;

    private Date returnDate;


    // Getters and Setters

}

...

```

7. ****Person****

```
```java

public class Person {

 private Long id;

 private String firstName;

 private String lastName;

 private String email;

 private String phoneNumber;

 // Getters and Setters

}

...

```

### ### Updated DAO Interfaces

#### 1. **\*\*BookDAO\*\***

```
```java

public interface BookDAO {

    void save(Book book);

    void delete(Book book);

    Book findById(Long id);

    List<Book> findAll();

}

```
```

#### 2. **\*\*BorrowerDAO\*\***

```
```java

public interface BorrowerDAO {

    void save(Borrower borrower);

    void delete(Borrower borrower);

    Borrower findById(Long id);

    List<Borrower> findAll();

    Borrower findByPerson(Person person);

}

```
```



### 3. **\*\*ItemDAO\*\***

```
```java

public interface ItemDAO {

    void save(Item item);

    void delete(Item item);

    Item findById(Long id);

    List<Item> findAll();

    Item findByBook(Book book);

}

```
```

### 4. **\*\*LoanDAO\*\***

```
```java

public interface LoanDAO {

    void save(Loan loan);

    void delete(Loan loan);

    Loan findById(Long id);

    List<Loan> findAll();

    List<Loan> findByBorrower(Borrower borrower);

    List<Loan> findByItem(Item item);

}

```
```

### ### Updated Service Interfaces

#### 1. **\*\*LibraryService\*\***

```
```java

public interface LibraryService {

    void registerBorrower(Person person, BorrowerCategory category);

    void updateBorrowerDetails(Person person, BorrowerCategory category);

    void deleteBorrower(Person person);

    void addBook(Book book);

    void removeBook(Book book);

    Book getBook(Long id);

    List<Book> getAllBooks();

    void borrowItem(Long borrowerId, Long itemId);

    void returnItem(Long borrowerId, Long itemId);

    void notifyDelayedReturns();

}

```
```

### ### Updated Service Implementations

#### 1. **\*\*LibraryServiceImpl\*\***

```
```java

public class LibraryServiceImpl implements LibraryService {
```

```
private BookDAO bookDAO;

private BorrowerDAO borrowerDAO;

private ItemDAO itemDAO;

private LoanDAO loanDAO;


// Constructor with DAOs injection

public LibraryServiceImpl(BookDAO bookDAO, BorrowerDAO borrowerDAO, ItemDAO
itemDAO, LoanDAO loanDAO) {

    this.bookDAO = bookDAO;

    this.borrowerDAO = borrowerDAO;

    this.itemDAO = itemDAO;

    this.loanDAO = loanDAO;

}


@Override

public void registerBorrower(Person person, BorrowerCategory category) {

    Borrower existingBorrower = borrowerDAO.findByPerson(person);

    if (existingBorrower != null) {

        throw new IllegalArgumentException("Borrower already exists");

    }

    Borrower borrower = new Borrower();

    borrower.setPerson(person);

    borrower.setCategory(category);
```

```
        borrowerDAO.save(borrower);  
    }  
}
```

@Override

```
public void updateBorrowerDetails(Person person, BorrowerCategory category) {  
  
    Borrower borrower = borrowerDAO.findByPerson(person);  
  
    if (borrower == null) {  
  
        throw new IllegalArgumentException("Borrower does not exist");  
  
    }  
  
    borrower.setCategory(category);  
  
    borrowerDAO.save(borrower);  
  
}
```

@Override

```
public void deleteBorrower(Person person) {  
  
    Borrower borrower = borrowerDAO.findByPerson(person);  
  
    if (borrower == null) {  
  
        throw new IllegalArgumentException("Borrower does not exist");  
  
    }  
  
    List<Loan> loans = loanDAO.findByBorrower(borrower);  
  
    for (Loan loan : loans) {  
  
        if (loan.getReturnDate() == null) {  
  
            throw new IllegalStateException("Borrower has pending items");  
  
        }  
  
    }  
  
}
```

```
    }  
}  
  
    borrowerDAO.delete(borrower);  
}
```

```
@Override  
  
public void addBook(Book book) {  
  
    bookDAO.save(book);  
}
```

```
@Override  
  
public void removeBook(Book book) {  
  
    bookDAO.delete(book);  
}
```

```
@Override  
  
public Book getBook(Long id) {  
  
    return bookDAO.findById(id);  
}
```

```
@Override  
  
public List<Book> getAllBooks() {  
  
    return bookDAO.findAll();  
}
```

```
}
```

```
@Override
```

```
public void borrowItem(Long borrowerId, Long itemId) {

    Borrower borrower = borrowerDAO.findById(borrowerId);

    Item item = itemDAO.findById(itemId);

    if (item.getState() == ItemState.AVAILABLE && borrower.getLoans().size() <
borrower.getCategory().getMaxLoans()) {

        item.setState(ItemState.BORROWED);

        Loan loan = new Loan();

        loan.setBorrower(borrower);

        loan.setItem(item);

        loan.setLoanDate(new Date());

        // Assume 2 weeks loan period

        loan.setDueDate(new Date(System.currentTimeMillis() + 1209600000L));

        loanDAO.save(loan);

    } else {

        throw new IllegalStateException("Item is not available or borrower has reached
maximum loans");

    }

}
```

```
@Override
```

```
public void returnItem(Long borrowerId, Long itemId) {
```

```

Item item = itemDAO.findById(itemId);

List<Loan> loans = loanDAO.findByItem(item);

Loan loan = loans.stream().filter(l -> l.getReturnDate() == null).findFirst().orElse(null);

if (loan != null) {

    item.setState(ItemState.AVAILABLE);

    loan.setReturnDate(new Date());

    loanDAO.save(loan);

} else {

    throw new IllegalArgumentException("No active loan found for the item");

}

}

```

@Override

```

public void notifyDelayedReturns() {

    List<Loan> loans = loanDAO.findAll();

    for (Loan loan : loans) {

        if (loan.getDueDate().before(new Date()) && loan.getReturnDate() == null) {

            Borrower borrower = loan.getBorrower();

            Person person = borrower.getPerson();

            if (person.getEmail() != null) {

                // Send email (pseudo-code)

                // EmailService.send(person.getEmail(), "Return overdue item: " +
loan.getItem().getBook().getTitle());

            }

}

```

```
    }  
    }  
    }  
}  
...
```

Step Definitions Skeleton

Here are the step definitions based on the provided features and scenarios:

```
```java  
public class LibrarySteps {

 private LibraryService libraryService;

 private Book book;

 private Borrower borrower;

 private Person person;

 private BorrowerCategory category;

 private Item item;

 private Loan loan;

 public LibrarySteps() {

 // Initialize DAOs with mock implementations or actual implementations
```



```
BookDAO bookDAO = new BookDAOImpl();

BorrowerDAO borrowerDAO = new BorrowerDAOImpl();

ItemDAO itemDAO = new ItemDAOImpl();

LoanDAO loanDAO = new LoanDAOImpl();

libraryService = new LibraryServiceImpl(bookDAO, borrowerDAO, itemDAO, loanDAO);

}
```

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

public void georgeRedIsNotRegistered() {

 person = new Person();

 person.setFirstName("George");

 person.setLastName("Red");

 Borrower existingBorrower = borrowerDAO.findByPerson(person);

 if (existingBorrower != null) {

 borrowerDAO.delete(existingBorrower);

 }

}
```

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

public void georgeRedGetsRegistered() {

 person = new Person();

 person.setFirstName("George");

}
```

```

 person.setLastName("Red");

 category = new BorrowerCategory();

 category.setName("Regular");

 libraryService.registerBorrower(person, category);

}

```

```

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

public void theSystemSuccessfullyStoresTheBorrowerDetails() {

 Borrower storedBorrower = borrowerDAO.findByPerson(person);

```

ChatGPT

```

 assertNotNull(storedBorrower);

}

```

```

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

public void georgeRedIsRegistered() {

 person = new Person();

 person.setFirstName("George");

 person.setLastName("Red");

 category = new BorrowerCategory();

 category.setName("Regular");

 libraryService.registerBorrower(person, category);

}

```

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

```
public void theSystemAttemptsToRegisterGeorgeRedAgain() {
 try {
 libraryService.registerBorrower(person, category);
 } catch (IllegalArgumentException e) {
 assertEquals("Borrower already exists", e.getMessage());
 }
}
```

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

```
public void theSystemInformsUserAlreadyExists() {
 // Assertion already made in the exception handling
}
```

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

```
public void georgeRedUpdatesHisDetails() {
 category.setName("Premium");
 libraryService.updateBorrowerDetails(person, category);
}
```

@Then("the system saves the changes")

```
public void theSystemSavesTheChanges() {
 Borrower updatedBorrower = borrowerDAO.findByPerson(person);
```

```
 assertEquals("Premium", updatedBorrower.getCategory().getName());
 }
}
```

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

public void georgeRedTriesToUpdateHisDetails() {

 try {

 libraryService.updateBorrowerDetails(person, category);

 } catch (IllegalArgumentException e) {

 assertEquals("Borrower does not exist", e.getMessage());

 }

}
```

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

public void theSystemDisplaysErrorBorrowerDoesNotExist() {

 // Assertion already made in the exception handling

}
```

```
@When("the system deletes George Red's account")

public void theSystemDeletesGeorgeRedsAccount() {

 libraryService.deleteBorrower(person);

}
```

```
@Then("the system removes George Red's details")
```

```
public void theSystemRemovesGeorgeRedsDetails() {

 Borrower deletedBorrower = borrowerDAO.findByPerson(person);

 assertNull(deletedBorrower);

}
```

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

public void theSystemAttemptsToDeleteGeorgeRedsAccount() {

 try {

 libraryService.deleteBorrower(person);

 } catch (IllegalArgumentException e) {

 assertEquals("Borrower does not exist", e.getMessage());

 }

}
```

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

public void theSystemInformsBorrowerDoesNotExist() {

 // Assertion already made in the exception handling

}
```

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

public void georgeRedHasPendingItems() {

 item = new Item();

 item.setBook(new Book());

}
```

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

 loan = new Loan();

 loan.setBorrower(borrower);

 loan.setItem(item);

 loan.setLoanDate(new Date());

 loanDAO.save(loan);
 }
}
```

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

public void theSystemDoesNotRemoveGeorgeRedsDetails() {

 try {

 libraryService.deleteBorrower(person);

 } catch (IllegalStateException e) {

 assertEquals("Borrower has pending items", e.getMessage());

 }

}
```

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

public void theSystemInformsAboutPendingItems() {

 // Assertion already made in the exception handling

}
```

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

```
public void georgeRedHasBorrowedHarryPotter() {

 book = new Book();

 book.setTitle("Harry Potter");

 item = new Item();

 item.setBook(book);

 item.setState(ItemState.BORROWED);

 loan = new Loan();

 loan.setBorrower(borrower);

 loan.setItem(item);

 loan.setLoanDate(new Date(System.currentTimeMillis() - 86400000L)); // 1 day ago

 loan.setDueDate(new Date(System.currentTimeMillis() - 1000L)); // 1 second ago

 loanDAO.save(loan);

}
```

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

public void harryPottersDueDateHasPassed() {

 assertTrue(loan.getDueDate().before(new Date()));

}
```

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

public void georgeRedHasAnEmailAddress() {

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

}
```

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

```
public void theSystemExecutesDelayedReturnNotification() {
```

```
 libraryService.notifyDelayedReturns();
```

```
}
```

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

```
public void georgeRedReceivesEmailNotification() {
```

```
 // Assume email service works correctly. This would be verified in integration testing.
```

```
}
```

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

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

```
 person.setEmail(null);
```

```
}
```

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

```
public void georgeRedDoesNotReceiveEmailNotification() {
```

```
 // Email service would not send an email. This would be verified in integration testing.
```

```
}
```

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

```
public void theLibraryHasItemHarryPotterAvailable() {
```



```
book = new Book();

book.setTitle("Harry Potter");

item = new Item();

item.setBook(book);

item.setState(ItemState.AVAILABLE);

itemDAO.save(item);

}
```

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

public void georgeRedBorrowsHarryPotter() {

 libraryService.borrowItem(borrower.getId(), item.getId());

}
```

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

public void theSystemSuccessfullyLoansHarryPotter() {

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

}
```

```
@Then("George Red's pending items increase to 3")

public void georgeRedsPendingItemsIncreaseTo3() {

 assertEquals(3, borrower.getLoans().size());

}
```

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

```
public void theLibraryHasItemsHarryPotterAndMobyDick() {
```

```
 Book book1 = new Book();
```

```
 book1.setTitle("Harry Potter");
```

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

```
 item.setBook(book1);
```

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

```
 itemDAO.save(item);
```

```
 Book book2 = new Book();
```

```
 book2.setTitle("Moby Dick");
```

```
 Item item2 = new Item();
```

```
 item2.setBook(book2);
```

```
 item2.setState(ItemState.AVAILABLE);
```

```
 itemDAO.save(item2);
```

```
}
```

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

```
public void georgeRedTriesToBorrowBothItems() {
```

```
 libraryService.borrowItem(borrower.getId(), item.getId()); // Harry Potter
```

```
 try {
```

```
 libraryService.borrowItem(borrower.getId(), item.getId()); // Moby Dick
```

```
 } catch (IllegalStateException e) {
```

```
 assertEquals("Item is not available or borrower has reached maximum loans",
e.getMessage());

 }

}
```

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

```
public void theSystemDoesNotLoanMobyDickToGeorgeRed() {

 Item mobyDickItem = itemDAO.findByBook(book);

 assertNotEquals(ItemState.BORROWED, mobyDickItem.getState());

}
```

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

```
public void theItemHarryPotterIsInLibraryButNotInSystem() {

 book = new Book();

 book.setTitle("Harry Potter");

 item = new Item();

 item.setBook(book);

 item.setState(ItemState.AVAILABLE);

 // Not saving the item to the system

}
```

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

```
public void georgeRedTriesToBorrowHarryPotter() {
```

```
try {

 libraryService.borrowItem(borrower.getId(), item.getId());

} catch (IllegalArgumentException e) {

 assertEquals("Item not found in the system", e.getMessage());

}
}
```

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

public void theSystemReturnsErrorDueToItemStatus() {

 // Assertion already made in the exception handling

}
```

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

public void theSystemWithdrawsHarryPotter() {

 Item withdrawnItem = itemDAO.findByBook(book);

 assertNull(withdrawnItem);

}
```

```
@Given("George Red has 3 pending items to be returned")

public void georgeRedHas3PendingItems() {

 for (int i = 0; i < 3; i++) {

 Item borrowedItem = new Item();

 borrowedItem.setBook(new Book());

 }

}
```

```

 borrowedItem.setState(ItemState.BORROWED);

 Loan newLoan = new Loan();

 newLoan.setBorrower(borrower);

 newLoan.setItem(borrowedItem);

 newLoan.setLoanDate(new Date());

 loanDAO.save(newLoan);

 }
}

```

```

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

```

```

 public void georgeRedTriesToBorrowHarry

```

ChatGPT

```

Potter() {

 try {

 libraryService.borrowItem(borrower.getId(), item.getId());

 } catch (IllegalStateException e) {

 assertEquals("Item is not available or borrower has reached maximum loans",
e.getMessage());

 }

}
}

```

```

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

```

```

 public void theSystemDoesNotAllowLoan() {

```

```

 // Assertion already made in the exception handling

```

```
}
```

```
@Then("George Red's pending items remain 3")
```

```
public void georgeRedsPendingItemsRemain3() {
```

```
 assertEquals(3, borrower.getLoans().size());
```

```
}
```

```
@Given("George Red borrowed the item Animal Kingdom 5 days prior to today's date")
```

```
public void georgeRedBorrowedAnimalKingdom5DaysAgo() {
```

```
 book = new Book();
```

```
 book.setTitle("Animal Kingdom");
```

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

```
 item.setBook(book);
```

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

```
 loan = new Loan();
```

```
 loan.setBorrower(borrower);
```

```
 loan.setItem(item);
```

```
 loan.setLoanDate(new Date(System.currentTimeMillis() - 5 * 86400000L)); // 5 days ago
```

```
 loan.setDueDate(new Date(System.currentTimeMillis() + 5 * 86400000L)); // 5 days in the
future
```

```
 loanDAO.save(loan);
```

```
}
```

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

```
public void theReturnOfAnimalKingdomIsProcessed() {

 libraryService.returnItem(borrower.getId(), item.getId());

}
```

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

```
public void theSystemMarksAnimalKingdomAsAvailable() {

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

}
```

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

```
public void georgeRedHasOneLessPendingItem() {

 assertEquals(2, borrower.getLoans().size());

}
```

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

```
public void georgeRedDoesNotPayFine() {

 // No fine calculation needed as the return is within the lending days

}
```

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

```
public void theReturnDateOfLoanIsSetToToday() {

 assertEquals(new Date(), loan.getReturnDate());

}
```

```

@Given("George Red borrowed the item Animal Kingdom 10 days prior to today's date")

public void georgeRedBorrowedAnimalKingdom10DaysAgo() {

 loan.setLoanDate(new Date(System.currentTimeMillis() - 10 * 86400000L)); // 10 days
ago

 loan.setDueDate(new Date(System.currentTimeMillis() - 3 * 86400000L)); // 3 days ago

 loanDAO.save(loan);

}

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

public void georgeRedPaysFine() {

 double fine = (new Date().getTime() - loan.getDueDate().getTime()) / 86400000L *
borrower.getCategory().getFineRate();

 // Assume fine calculation is correct. This would be verified in integration testing.

}

}

...

```

This skeleton provides a basis for the step definitions based on your provided features and scenarios. You'll need to implement the actual DAO classes (e.g., `BookDAOImpl`, `BorrowerDAOImpl`, etc.) and any necessary utility classes or methods for a complete system.