

Σε αυτή τη συνομιλία, παρείχα στο LLM όλα τα features του συστήματος, τα ονόματα των κλάσεων domain που πρέπει να χρησιμοποιήσει, καθώς και την αρχιτεκτονική και γενικές πληροφορίες για το σύστημα. Η διαφορά σε αυτή την περίπτωση ήταν ότι ζήτησα πρώτα από το LLM να σκεφτεί και να δημιουργήσει τον κώδικα για τις κλάσεις domain. Το LLM εκτίμησε αρκετά καλά τις σχέσεις μεταξύ των κλάσεων και μερικά features. Ξεκίνησε να δημιουργεί κώδικα από την αρχή, πράγμα που δείχνει ότι κατανοούσε πολύ καλά τι του παρείχα. Ωστόσο, χρειάστηκε και πάλι κάποια επανάληψη στις εντολές για να δημιουργήσει όλα τα step definitions, παραλείποντας μερικά. Συνολικά, δημιούργησε αρκετά περίπλοκο κώδικα, ο οποίος ήταν καλός σε ορισμένες περιπτώσεις αλλά κακός σε άλλες, επειδή ήταν υπερβολικά περίπλοκος για να κατανοηθεί εύκολα το step definition.

## 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

Great, it sounds like you have a structured plan for your library system application using Java with Gherkin scenarios for behavior-driven development (BDD). Before we proceed with implementing the step definitions, I have a few questions to ensure everything is clear and aligned with your requirements:

1. **\*\*Domain Classes:\*\*** Could you please list the domain classes you have in your application?

Knowing the classes and their relationships will help in crafting the step definitions properly.

2. **Data Access Objects (DAOs):** Do you already have the DAOs implemented, or would you like assistance in defining these as well? If they are already implemented, please share the methods available in these DAOs.

3. **Service Classes:** Similar to DAOs, do you have the service layer implemented? If yes, please share the methods available in these service classes. If not, do you need help in defining these?

4. **Gherkin Scenarios:** Could you provide the Gherkin scenarios? This will help in understanding the specific functionalities and interactions that need to be covered in the step definitions.

5. **Integration Details:** How are the step definitions expected to interact with the service layer and DAOs? This includes any specific frameworks or libraries (like Spring for dependency injection) that you're using.

6. **Test Configuration:** Are there any specific configurations or setups (like database configurations for tests, use of mock frameworks like Mockito) you are using or plan to use in the testing environment?

Understanding these details will help in providing accurate and efficient code for your step definitions in Java.

**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

Thank you for listing the domain classes. Here's a quick overview of what these classes might represent, based on common usage in a library system:

1. **Book:** Typically contains information about the book itself, such as title, author, ISBN, and possibly linked to `Item` instances.
2. **Borrower:** A subclass of `Person`, specific for users of the library who can borrow items. Might include properties specific to borrowing activities.
3. **BorrowerCategory:** Could classify borrowers, possibly affecting loan terms like duration, number of items borrowable at once, etc.
4. **Item:** Represents a physical instance of a library resource, which could be a book, a DVD, etc. This class would likely link to `ItemState`.
5. **ItemState:** Enum or class representing the state of an item (e.g., available, loaned, under repair, lost).
6. **Loan:** Represents the borrowing of an item by a borrower, including details like loan start date, due date, return date, etc.
7. **Person:** A general class for individuals associated with the library, which might be extended by `Borrower`.

For each of these classes, typical attributes and methods can be guessed, but specific details will depend on your system's requirements. Next, please provide the features/scenarios of your system so we can understand how these classes interact within your application, and how to structure the step definitions to test these interactions. Also, let me know if you'd like to see sample Java class definitions for these domain entities or any additional setup around these entities.

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

Create the domain/services/daos code first

ChatGPT

Let's start by designing the domain classes for your library system based on the descriptions you provided and the features outlined. I'll also provide a basic structure for the DAOs (Data Access Objects) and service classes. For this setup, we'll assume a typical Java environment, potentially using Spring Boot for dependency management and Hibernate for ORM.

### ### 1. Domain Classes

Here's a brief outline of each class:

##### Person

```
```java
@Entity

public class Person {

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)

    private Long id;

    private String name;

    // Constructors, getters, and setters

}
...

```

#### Borrower (extends Person)

```
```java
@Entity

public class Borrower extends Person {

    @ManyToOne

    private BorrowerCategory category;

    private String borrowerNumber; // Unique identifier for the borrower

    // Constructors, getters, and setters

}
...

```

#### BorrowerCategory

```
```java
```

@Entity

```
public class BorrowerCategory {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.AUTO)
```

```
    private Long id;
```

```
    private String categoryName;
```

```
    private Integer maxLoanLimit;
```

```
    private Integer maxLoanDays;
```

```
    // Constructors, getters, and setters
```

```
}
```

```
...
```

#### Book

```
```java
```

@Entity

```
public class Book {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.AUTO)
```

```
    private Long id;
```

```
    private String title;
```

```
private String author;

private String isbn;

// Constructors, getters, and setters

}

...
```

#### Item

```
```java

@Entity

public class Item {

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)

    private Long id;

    @ManyToOne

    private Book book;

    @Enumerated(EnumType.STRING)

    private ItemState state;

    // Constructors, getters, and setters

}

...
```

#### ItemState (Enum)

```
```java

public enum ItemState {

    AVAILABLE, LOANED, REPAIR, LOST

}

...

```

#### Loan

```
```java

@Entity

public class Loan {

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)

    private Long id;

    @ManyToOne

    private Borrower borrower;

    @ManyToOne

    private Item item;

    private LocalDate loanDate;

    private LocalDate dueDate;

    private LocalDate returnDate;

    // Constructors, getters, and setters

}

```

...

### ### 2. Data Access Objects (DAOs)

Using Spring Data JPA, you can define interfaces for your DAOs. Here's an example for `Borrower`:

```
```java
@Repository

public interface BorrowerRepository extends JpaRepository<Borrower, Long> {

    Optional<Borrower> findByBorrowerNumber(String borrowerNumber);

}
```
```

You would create similar repositories for `Book`, `Item`, `Loan`, etc.

### ### 3. Service Classes

Here's an example service class for managing borrowers:

```
```java
@Service

public class BorrowerService {
```



@Autowired

private BorrowerRepository borrowerRepository;

public Borrower registerBorrower(Borrower borrower) throws Exception {

if(borrowerRepository.findByBorrowerNumber(borrower.getBorrowerNumber()).isPresent()) {

throw new Exception("Borrower already registered.");

}

return borrowerRepository.save(borrower);

}

public Borrower updateBorrower(Borrower borrower) throws Exception {

if(!borrowerRepository.existsById(borrower.getId())) {

throw new Exception("Borrower not found.");

}

return borrowerRepository.save(borrower);

}

public void deleteBorrower(Long id) throws Exception {

if(!borrowerRepository.existsById(id)) {

throw new Exception("Borrower not found.");

}

```
        borrowerRepository.deleteById(id);  
    }  
}  
...  

```

This structure sets up your application to manage borrowers. For other functionalities, similar patterns can be followed. Please let me know if you need the services and DAOs for handling loans, items, or books. We can also proceed to implement step definitions based on this architecture.

## User

### Don't use Spring for the DAOs, just create DAO classes for the classes you think need DAOs

ChatGPT

Without using Spring Data JPA, we'll create traditional DAO classes for managing database operations. These DAOs will handle CRUD (Create, Read, Update, Delete) operations directly through JDBC or a similar framework. We'll assume a simple JDBC setup for these examples.

#### ### DAO Interfaces

We'll first define the interfaces for the DAO classes. The DAOs will focus on essential operations such as finding, adding, updating, and deleting entities.

#### #### BookDAO

```
```java
```

```
public interface BookDAO {
```

```
    Book findById(Long id);
```

```
    List<Book> findAll();
```

```
    void save(Book book);
```

```
    void update(Book book);
```

```
    void delete(Long id);
```

```
}
```

```
...
```

```
#### BorrowerDAO
```

```
```java
```

```
public interface BorrowerDAO {
```

```
    Borrower findById(Long id);
```

```
    Borrower findByBorrowerNumber(String borrowerNumber);
```

```
    List<Borrower> findAll();
```

```
    void save(Borrower borrower);
```

```
    void update(Borrower borrower);
```

```
    void delete(Long id);
```

```
}
```

```
...
```

```
#### ItemDAO
```

```
```java
```

```
public interface ItemDAO {
```

```
    Item findById(Long id);
```

```
    List<Item> findAll();
```

```
    void save(Item item);
```

```
    void update(Item item);
```

```
    void delete(Long id);
```

```
}
```

```
```
```

```
#### LoanDAO
```

```
```java
```

```
public interface LoanDAO {
```

```
    Loan findById(Long id);
```

```
    List<Loan> findAll();
```

```
    void save(Loan loan);
```

```
    void update(Loan loan);
```

```
    void delete(Long id);
```

```
}
```

```
```
```

```
### DAO Implementations
```

These implementations will use JDBC to interact with the database. Below is an example implementation for the `BorrowerDAO`. We'll assume there's a helper class for obtaining database connections.

```
#### BorrowerDAOImpl
```

```
```java
```

```
public class BorrowerDAOImpl implements BorrowerDAO {
```

```
    private Connection getConnection() throws SQLException {
```

```
        // This method should return a connection from a connection pool or database source
```

```
        return DriverManager.getConnection("jdbc:url", "username", "password");
```

```
    }
```

```
@Override
```

```
public Borrower findById(Long id) {
```

```
    String sql = "SELECT * FROM borrowers WHERE id = ?";
```

```
    try (Connection conn = getConnection();
```

```
        PreparedStatement stmt = conn.prepareStatement(sql)) {
```

```
        stmt.setLong(1, id);
```

```
        ResultSet rs = stmt.executeQuery();
```

```
        if (rs.next()) {
```

```

        return new Borrower(rs.getLong("id"), rs.getString("name"),
rs.getString("borrowerNumber"));

    }

    } catch (SQLException e) {

        e.printStackTrace();

    }

    return null;

}

```

@Override

```

public Borrower findByBorrowerNumber(String borrowerNumber) {

    String sql = "SELECT * FROM borrowers WHERE borrowerNumber = ?";

    try (Connection conn = getConnection();

        PreparedStatement stmt = conn.prepareStatement(sql)) {

        stmt.setString(1, borrowerNumber);

        ResultSet rs = stmt.executeQuery();

        if (rs.next()) {

            return new Borrower(rs.getLong("id"), rs.getString("name"),
rs.getString("borrowerNumber"));

        }

    } catch (SQLException e) {

        e.printStackTrace();

    }
}

```

```
}  
  
return null;  
  
}
```

@Override

```
public List<Borrower> findAll() {  
  
    List<Borrower> borrowers = new ArrayList<>();  
  
    String sql = "SELECT * FROM borrowers";  
  
    try (Connection conn = getConnection();  
  
        PreparedStatement stmt = conn.prepareStatement(sql);  
  
        ResultSet rs = stmt.executeQuery()) {  
  
        while (rs.next()) {  
  
            borrowers.add(new Borrower(rs.getLong("id"),  
rs.getString("name"),  
rs.getString("borrowerNumber")));  
  
        }  
  
    } catch (SQLException e) {  
  
        e.printStackTrace();  
  
    }  
  
    return borrowers;  
  
}
```

@Override

```

public void save(Borrower borrower) {

    String sql = "INSERT INTO borrowers (name, borrowerNumber) VALUES (?, ?)";

    try (Connection conn = getConnection();

        PreparedStatement stmt = conn.prepareStatement(sql)) {

        stmt.setString(1, borrower.getName());

        stmt.setString(2, borrower.getBorrowerNumber());

        stmt.executeUpdate();

    } catch (SQLException e) {

        e.printStackTrace();

    }

}

```

@Override

```

public void update(Borrower borrower) {

    String sql = "UPDATE borrowers SET name = ?, borrowerNumber = ? WHERE id = ?";

    try (Connection conn = getConnection();

        PreparedStatement stmt = conn.prepareStatement(sql)) {

        stmt.setString(1, borrower.getName());

        stmt.setString(2, borrower.getBorrowerNumber());

        stmt.setLong(3, borrower.getId());

        stmt.executeUpdate();

    } catch (SQLException e) {

```



```

        e.printStackTrace();
    }
}

@Override

public void delete(Long id) {

    String sql = "DELETE FROM borrowers WHERE id = ?";

    try (Connection conn = getConnection();

        PreparedStatement stmt = conn.prepareStatement(sql)) {

        stmt.setLong(1, id);

        stmt.executeUpdate();

    } catch (SQLException e) {

        e.printStackTrace();

    }

}

}

...

```

For the other DAOs (`BookDAOImpl`, `ItemDAOImpl`, `LoanDAOImpl`), similar patterns can be followed where the SQL queries and methods are tailored to the respective entity operations.

This setup defines a clear separation of concerns, with each DAO handling the data operations for a specific domain entity. Adjust these implementations according to your specific database schema and business logic requirements.

## User

**Now, give me the all the step definitions for the feature: Borrower Handling by the system**

ChatGPT

Let's craft the step definitions for the "Borrower Handling by the System" feature in Java using Cucumber for behavior-driven development (BDD). These step definitions will interact with the service layer (assuming there's a service class named `BorrowerService` that we've defined to handle the logic) to perform actions based on the Gherkin scenarios provided.

### Cucumber Step Definitions for "Borrower Handling by the System"

First, ensure you have a `BorrowerService` that can handle operations like registering, updating, and deleting a borrower. We'll mock or implement the methods in the service class as needed.

Here's how the step definitions might look:

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

```

public class BorrowerSteps {

    private BorrowerService borrowerService = new BorrowerService(new BorrowerDAOImpl());

    private Borrower borrower;

    private String operationResult = "";

    @Given("^{string} is not registered as a borrower$")

    public void george_red_is_not_registered_as_a_borrower(String name) {

        // Simulate that the borrower is not found in the database

        Assert.assertNull(borrowerService.findByBorrowerNumber(name));

    }

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

    public void george_red_gets_registered_in_the_system_with_a_unique_borrower_number_and_his_details(String name) {

        borrower = new Borrower();

        borrower.setName(name);

        borrower.setBorrowerNumber("BR001");

        try {

            borrowerService.registerBorrower(borrower);

```

```
        operationResult = "SUCCESS";

    } catch (Exception e) {

        operationResult = e.getMessage();

    }

}
```

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

public void the_system_successfully_stores_the_borrower_s_details() {

    Assert.assertEquals("SUCCESS", operationResult);

}
```

```
@Given("^.{string} is registered as a borrower$")

public void george_red_is_registered_as_a_borrower(String name) {

    // Assume the borrower is already registered in the database

    borrower = new Borrower();

    borrower.setName(name);

    borrower.setBorrowerNumber("BR001");

    borrowerService.saveBorrower(borrower);

}
```

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

```

public void
the_system_attempts_to_register_george_red_with_a_unique_borrower_number_and_his_de
tails(String name) {

    try {

        borrowerService.registerBorrower(borrower);

        operationResult = "SUCCESS";

    } catch (Exception e) {

        operationResult = e.getMessage();

    }

}

```

```

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

public void the_system_informs_that_the_user_already_exists() {

    Assert.assertEquals("Borrower already registered.", operationResult);

}

```

```

@When("^a {string} updates his borrowing details$")

public void george_red_updates_his_borrowing_details(String name) {

    // Assume updates are made here, like changing the borrower category

    borrower.setCategory(new BorrowerCategory("Standard", 5, 30));

    try {

        borrowerService.updateBorrower(borrower);

    }
}

```

```
        operationResult = "SUCCESS";

    } catch (Exception e) {

        operationResult = e.getMessage();

    }

}
```

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

public void the_system_saves_the_changes() {

    Assert.assertEquals("SUCCESS", operationResult);

}
```

```
@When("^{string} tries to update his borrowing details$")

public void george_red_tries_to_update_his_borrowing_details(String name) {

    // Simulate updating a non-existent borrower

    try {

        borrowerService.updateBorrower(new Borrower());

        operationResult = "SUCCESS";

    } catch (Exception e) {

        operationResult = e.getMessage();

    }

}
```

```
@Then("^the system displays an error message indicating that {string} does not exist$")

public void
the_system_displays_an_error_message_indicating_that_george_red_does_not_exist(String
name) {

    Assert.assertEquals("Borrower not found.", operationResult);

}
```

```
@When("^the system deletes {string}'s account$")

public void the_system_deletes_george_red_s_account(String name) {

    try {

        borrowerService.deleteBorrower(borrower.getId());

        operationResult = "SUCCESS";

    } catch (Exception e) {

        operationResult = e.getMessage();

    }

}
```

```
@Then("^the system removes {string}'s details$")

public void the_system_removes_george_red_s_details(String name) {

    Assert.assertEquals("SUCCESS", operationResult);

}
```

```

@When("^the system attempts to delete {string}'s account$")

public void the_system_attempts_to_delete_george_red_s_account(String name) {

    // Trying to delete an unregistered borrower

    try {

        borrowerService.deleteBorrower(999L); // assuming 999L is an ID that doesn't exist

        operationResult = "SUCCESS";

    } catch (Exception e) {

        operationResult = e.getMessage();

    }

}

```

```

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

public void the_system_informs_that_the_borrower_does_not_exist() {

    Assert.assertEquals("Borrower not found.", operationResult);

}

}

...

```

### Additional Notes



1. **\*\*Integration with Database:\*\*** This example assumes that the ``BorrowerService`` interacts with a real or mocked database. You might use a mocking framework like Mockito to simulate these interactions in a unit test environment.
2. **\*\*Service and DAO Methods:\*\*** Ensure that the ``BorrowerService`` and ``BorrowerDAOImpl`` provide methods like ``registerBorrower``, ``updateBorrower``, ``deleteBorrower``, ``findByBorrowerNumber``, and ``saveBorrower`` as used in the step definitions.
3. **\*\*Error Handling:\*\*** Proper error handling should be implemented in the service methods to throw exceptions when operations fail, which are caught and checked in the step definitions.

These step definitions now provide a solid foundation for testing the borrower handling features of your library system. Adjustments and enhancements might be needed based on your specific business rules and data management policies.

## User

**You sure there aren't step definitions missing for this feature? I think you skipped some**

ChatGPT

You're right; I initially covered many of the primary scenarios, but it seems I missed some specific scenarios regarding handling unreturned items and errors when deleting a borrower who hasn't registered yet. Let's complete those step definitions to cover all cases fully.

Here are the missing step definitions for handling unreturned items and when deleting accounts:

### Additional Step Definitions for Borrower Handling

#### Handling unreturned items before deleting a borrower

```
```java
```

```
@Given("^{string} has pending items$")
```

```
public void george_red_has_pending_items(String name) {
```

```
    // Assume George Red has items that haven't been returned
```

```
    borrower.setPendingItems(true); // Assuming there is a flag or method to set pending items
```

```
}
```

```
@When("^{the system attempts to delete {string}'s account$")
```

```
public void the_system_attempts_to_delete_borrower_account(String name) {
```

```
    // Trying to delete a borrower with unreturned items
```

```
    try {
```

```
        if (borrower.hasPendingItems()) {
```

```
            operationResult = "PENDING ITEMS";
```

```
        } else {
```

```
            borrowerService.deleteBorrower(borrower.getId());
```

```
            operationResult = "SUCCESS";
```

```
        }
```

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

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

```

    }
}

@Then("^the system does not remove {string}'s details$")

public void the_system_does_not_remove_borrower_details(String name) {

    Assert.assertNotEquals("SUCCESS", operationResult);

}

```

```

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

public void the_system_informs_about_the_pending_items() {

    Assert.assertEquals("PENDING ITEMS", operationResult);

}

...

```

#### Deleting a borrower when he hasn't registered

This scenario actually was covered but let's explicitly create a step definition for clarity:

```

```java

@When("^the system attempts to delete {string}'s account$")

public void the_system_attempts_to_delete_unregistered_borrower_account(String name) {

    // Attempting to delete a non-existing borrower

```

```

try {

    Borrower unregisteredBorrower = borrowerService.findByBorrowerNumber("non-existing-
number");

    if (unregisteredBorrower == null) {

        operationResult = "BORROWER DOES NOT EXIST";

    } else {

        borrowerService.deleteBorrower(unregisteredBorrower.getId());

        operationResult = "SUCCESS";

    }

} catch (Exception e) {

    operationResult = e.getMessage();

}

}

```

```

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

public void the_system_informs_that_the_borrower_does_not_exist() {

    Assert.assertEquals("BORROWER DOES NOT EXIST", operationResult);

}

...

```

### Review of Completion

With these additions, we now have a comprehensive set of step definitions covering all scenarios mentioned in your feature for "Borrower Handling by the System." This includes:

- Registering new borrowers and handling already registered cases.
- Updating borrower details for both registered and unregistered borrowers.
- Deleting borrowers, including those with pending items and those not registered.

This complete set ensures robust testing of the borrower management functionality in your library system. If you need further adjustments or additional scenarios, feel free to let me know!

**User**

**Now, based on the above, provide me with all the step definitions for the feature: Loaning Items**

ChatGPT

To develop the step definitions for the "Loaning Items" feature based on the scenarios you described, we'll utilize the `ItemService` and `LoanService` classes to manage the actions related to items and loans. The step definitions will reflect actions like checking item availability, registering loans, and handling borrower limits.

Here are the step definitions that correspond to each scenario described in your "Loaning Items" feature:

### Step Definitions for "Loaning Items"

```
```java
```

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

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

import io.cucumber.java.en.Then;

import org.junit.Assert;


public class LoaningItemsSteps {

    private ItemService itemService = new ItemService(new ItemDAOImpl());

    private LoanService loanService = new LoanService(new LoanDAOImpl());

    private Item item;

    private Borrower borrower;

    private String operationResult = "";


    @Given("^the library has the item {string} available$")

    public void the_library_has_the_item_available(String title) {

        item = itemService.findItemByTitle(title);

        Assert.assertNotNull(item);

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

    }


    @Given("^a {string} is a registered borrower$")

    public void george_red_is_a_registered_borrower(String name) {

        borrower = new Borrower();

        borrower.setName(name);

    }

}
```

```
borrower.setBorrowerNumber("BR001");

Assert.assertNotNull(borrower);

}
```

```
@Given("^{string} has {int} pending items to be returned$")

public void george_red_has_pending_items_to_be_returned(String name, Integer
pendingItems) {

    borrower.setPendingItemsCount(pendingItems);

}
```

```
@Given("^{string} has been assigned a maximum lending limit of {int}$")

public void george_red_has_been_assigned_a_maximum_lending_limit_of(String name,
Integer lendingLimit) {

    borrower.setMaxLendingLimit(lendingLimit);

}
```

```
@When("^{string} borrows the item {string}$")

public void george_red_borrows_the_item(String name, String title) {

    try {

        Loan newLoan = loanService loanItemToBorrower(item, borrower);

        operationResult = newLoan != null ? "SUCCESS" : "FAIL";

    } catch (Exception e) {
```

```
        operationResult = e.getMessage();
    }
}
```

```
@Then("^the system successfully loans the item {string} to {string} with a due date set$")

public void the_system_successfully_loans_the_item_to_george_red_with_a_due_date_set(String title,
String name) {

    Assert.assertEquals("SUCCESS", operationResult);
}
```

```
@Then("^{"string}'s pending items increase to {int}$")

public void george_red_s_pending_items_increase_to(String name, Integer newPendingItems)
{

    Assert.assertEquals(newPendingItems, borrower.getPendingItemsCount());
}
```

```
@When("^{"string} tries to borrow both items$")

public void george_red_tries_to_borrow_both_items(String name) {

    try {

        Loan firstLoan = loanService loanItemToBorrower(itemService.findItemByTitle("Harry
Potter"), borrower);
```



```
        Loan secondLoan = loanService loanItemToBorrower(itemService.findItemByTitle("Moby  
Dick"), borrower);
```

```
        operationResult = (firstLoan != null && secondLoan == null) ? "LIMIT REACHED" : "ERROR";
```

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

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

```
    }
```

```
}
```

```
@Then("^the system does not loan {string} to {string} due to the lending limit reached$")
```

```
public void the_system_does_not_loan_moby_dick_to_george_red_due_to_the_lending_limit_reached(String title, String name) {
```

```
    Assert.assertEquals("LIMIT REACHED", operationResult);
```

```
}
```

```
@Given("^the item {string} is in the library but not in the system$")
```

```
public void the_item_is_in_the_library_but_not_in_the_system(String title) {
```

```
    // Simulating an item that is physically there but not registered in the system
```

```
    Assert.assertNull(itemService.findItemByTitle(title));
```

```
}
```

```
@When("^{string} tries to borrow the item {string}$")
```

```

public void george_red_tries_to_borrow_the_item(String name, String title) {

    try {

        item = itemService.findItemByTitle(title);

        if (item == null || item.getState() != ItemState.AVAILABLE) {

            operationResult = "ITEM UNAVAILABLE";

        } else {

            loanService loanItemToBorrower(item, borrower);

            operationResult = "SUCCESS";

        }

    } catch (Exception e) {

        operationResult = e.getMessage();

    }

}

```

```

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

```

```

public void the_system_returns_an_error_due_to_the_item_s_status() {

    Assert.assertEquals("ITEM UNAVAILABLE", operationResult);

}

```

```

@Given("^a {string} has {int} pending items to be returned and a maximum lending limit of {int}$")

```

```

    public void borrower_has_pending_items_and_a_maximum_lending_limit(String name,
Integer pendingItems, Integer lendingLimit) {

    borrower.setPendingItemsCount(pendingItems);

    borrower.setMaxLendingLimit(lendingLimit);

}

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

public void the_system_doesn_t_allow_the_loan() {

    Assert.assertEquals("ITEM UNAVAILABLE", operationResult);

}

}

...

```

### ### Key Considerations

- **Item and Borrower State:** The step definitions assume the existence of methods like `findItemByTitle`, `loanItemToBorrower`, and that borrower and item states (e.g., `pendingItemsCount`, `maxLendingLimit`) are modifiable properties used during the loaning process.
- **Error Handling and Results:** Various outcomes like successful loans, limit checks, and unavailable items are handled through exceptions or conditional checks, reflecting their results in the step operation results.
- **Service Layer Interactions:** These steps assume interaction with well-defined service methods that encapsulate the business logic required by the scenarios, which should properly manage transactional behaviors and state changes.

This comprehensive set of step definitions ensures that each scenario around loaning items is thoroughly testable, covering both happy paths and edge cases within your library system's loan management functionality. Adjust and enhance these definitions based on specific business rules, data integrity requirements, and error management strategies.

**User**

**Now, give me all the step definitions for the feature: Returning a borrowed item in the library**

ChatGPT

To craft the step definitions for the "Returning a Borrowed Item" feature, we will simulate interactions with a `LoanService` and an `ItemService`. These services handle operations such as processing returns, updating item states, and handling fines for late returns.

### Step Definitions for "Returning a Borrowed Item"

```
```java
```

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

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

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

```
import org.junit.Assert;
```

```
public class ReturningItemsSteps {
```

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

```
    private ItemService itemService = new ItemService(new ItemDAOImpl());
```

```
private Item item;
```

```
private Borrower borrower;
```

```
private Loan loan;
```

```
private String operationResult = "";
```

```
@Given("^{string} borrowed the item {string} {int} days prior to today's date$")
```

```
public void borrower_borrowed_the_item_days_prior_to_today_s_date(String name, String  
title, int daysAgo) {
```

```
    item = itemService.findItemByTitle(title);
```

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

```
    borrower.setName(name);
```

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

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

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

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

```
    loan.setDueDate(loan.getLoanDate().plusDays(borrower.getCategory().getMaxLoanDays()));
```

```
    // Assuming a method that saves this setup for simulation purposes
```

```
    loanService.createLoan(loan);
```

```
}
```

```
@Given("^{string} has been assigned maximum lending days of {int}$")
```

```

    public void borrower_has_been_assigned_maximum_lending_days_of(String name, int
maxLendingDays) {

        BorrowerCategory category = new BorrowerCategory();

        category.setMaxLoanDays(maxLendingDays);

        borrower.setCategory(category);

    }

```

```

@When("^the return of {string} is processed$")

public void the_return_of_item_is_processed(String title) {

    try {

        operationResult = loanService.returnItem(loan);

    } catch (Exception e) {

        operationResult = e.getMessage();

    }

}

```

```

@Then("^the system marks the state of {string} as AVAILABLE$")

public void the_system_marks_the_state_of_item_as_available(String title) {

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

}

```

```

@Then("^ {string} has one less pending item$")

```

```
public void borrower_has_one_less_pending_item(String name) {  
  
    Assert.assertTrue(borrower.getPendingItemsCount() <  
loanService.findLoansByBorrower(borrower).size());  
  
}
```

```
@Then("^{string} does not pay a fine$")
```

```
public void borrower_does_not_pay_a_fine(String name) {  
  
    Assert.assertFalse(loanService.isFineApplicable(loan));  
  
}
```

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

```
public void the_return_date_of_the_loan_is_set_to_today_s_date() {  
  
    Assert.assertEquals(LocalDate.now(), loan.getReturnDate());  
  
}
```

```
@Given("^{string} borrowed the item {string} {int} days prior to today's date and has to pay a  
fine based on the borrower category$")
```

```
public void borrower_borrowed_the_item_days_prior_and_has_to_pay_a_fine(String name,  
String title, int daysAgo) {
```

```
    borrower_borrowed_the_item_days_prior_to_today_s_date(name, title, daysAgo);
```

```
    // Assuming there's a late fee calculation in the category setup
```

```
    borrower.getCategory().setFinePerDay(5.0); // Assuming a daily fine
```

```

    }

    @Then("^{string} pays a fine based on the borrower category$")

    public void borrower_pays_a_fine_based_on_the_borrower_category(String name) {

        double fine = loanService.calculateFine(loan);

        Assert.assertTrue(fine > 0);

        Assert.assertEquals("FINE APPLIED", operationResult);

    }

}

...

```

### ### Key Considerations

- **\*\*Service Layer Interactions:\*\*** These step definitions assume methods like `createLoan`, `returnItem`, `findLoansByBorrower`, `isFineApplicable`, and `calculateFine` are implemented in the `LoanService`. These methods handle the logic of loan creation, item returning, fine assessment, and updates.
- **\*\*Loan and Item Management:\*\*** The operations performed on loans and items are directly linked to their current state in the system, which the service layer handles. This ensures the steps reflect realistic scenarios.
- **\*\*Error and State Handling:\*\*** Proper management of borrower and item states, loan due dates, and fine calculations is critical and should be well encapsulated within the service methods.



These step definitions provide a solid foundation for testing the return functionality of items within your library system. Adjustments may be needed based on your specific business rules, the details of your database schema, and integration specifics of your actual services.

## User

**Now, with the same pattern, provide the step definitions of the feature: Delayed Notification**

ChatGPT

For the feature "Delayed Return Notification," we'll create step definitions that simulate interactions with the `LoanService` and possibly another service dedicated to handling notifications, such as an `EmailService`. The scenario involves notifying borrowers who have not returned items on time.

### Step Definitions for "Delayed Return Notification"

```
```java
```

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

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

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

```
import org.junit.Assert;
```

```
public class DelayedNotificationSteps {
```

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

```
private EmailService emailService = new EmailService(); // Assume EmailService handles
sending emails
```

```
private Borrower borrower;
```

```
private Item item;
```

```
private Loan loan;
```

```
private boolean emailSent = false;
```

```
@Given("^{string} has borrowed the item {string}$")
```

```
public void george_red_has_borrowed_the_item(String borrowerName, String itemName) {
```

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

```
    borrower.setName(borrowerName);
```

```
    borrower.setEmail("george.red@example.com"); // Email is set for sending notifications
```

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

```
    item.setTitle(itemName);
```

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

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

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

```
    loanService.createLoan(loan); // Assuming createLoan sets up the loan
```

```
}
```

```
@Given("^{string}'s due date has passed$")
```

```
public void item_s_due_date_has_passed(String itemName) {
```

```

        loan.setDueDate(LocalDate.now().minusDays(1)); // Set the due date to yesterday
    }

    @Given("^{string} has an email address$")

    public void borrower_has_an_email_address(String borrowerName) {

        borrower.setEmail("george.red@example.com"); // Ensure the borrower has an email
    }

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

    public void the_system_executes_the_delayed_return_notification_process() {

        if (loan.getDueDate().isBefore(LocalDate.now()) && borrower.getEmail() != null) {

            emailSent = emailService.sendEmail(borrower.getEmail(), "Delayed Return Notification",

                "This is a reminder that the item '" + loan.getItem().getTitle() + "' is overdue. Please
return it as soon as possible.");

        }

    }

    @Then("^{string} receives an email notification for the return of the item$")

    public void borrower_receives_an_email_notification_for_the_return_of_the_item(String
borrowerName) {

        Assert.assertTrue(emailSent);

    }

```

```

@Given("^{string} does not have an email address$")

public void borrower_does_not_have_an_email_address(String borrowerName) {

    borrower.setEmail(null); // Remove email to simulate this scenario

}

@Then("^{string} does not receive an email notification for the return of the item$")

public void borrower_does_not_receive_an_email_notification_for_the_return_of_the_item(String borrowerName) {

    Assert.assertFalse(emailSent);

}

}

...

```

### ### Key Considerations

- **Notification Service:** We've introduced an `EmailService`, which handles sending emails. The actual implementation would depend on your system's infrastructure but typically would involve SMTP settings or integration with an email sending service.
- **Loan and Borrower Data:** We set up borrower and loan data, including due dates and email addresses, directly related to the scenario's conditions.

- **Email Handling:** These steps handle both the scenario where an email is successfully sent and where no email is sent because the borrower lacks an email address. This ensures thorough testing of the notification logic.

This setup ensures your BDD testing for the delayed notification feature is robust and realistic, covering both success and edge cases related to notifying borrowers about overdue items. Adjustments can be made based on specific business logic, error handling, and system configurations.