

Στη συνομιλία αυτή, παρέθεσα στο LLM τα features του συστήματος, την περιγραφή του, καθώς και τα ονόματα των κλάσεων Domain που χρειαζόνταν. Το LLM κατανόησε άριστα την εργασία, όπως και στην προηγούμενη φάση, και άρχισε να δημιουργεί τον κώδικα για τις κλάσεις Domain (χωρίς προηγούμενη οδηγία για τη δημιουργία του κώδικα Domain), με χαρακτηριστικά που μάντεψε εξαιρετικά καλά, σχεδόν ακριβώς όπως τα πραγματικά χαρακτηριστικά των κλάσεων Domain. Επίσης, δημιούργησε αμέσως τον κώδικα για τα features με πολύ λίγες οδηγίες, καθώς είχε σαφή κατανόηση του τι έπρεπε να κάνει. Συνολικά, ο κώδικας ήταν πολύ καλύτερος σε σύγκριση με την προηγούμενη φάση, με το LLM να παράγει κώδικα με πολλές λεπτομέρειες και καλή κατανόηση του συστήματος και των Step Definitions. Συμπερασματικά, η πρόσθετη γνώση των ονομάτων των κλάσεων Domain είχε σημαντική θετική επίδραση στο αποτέλεσμα.

Phase 2 Chat 1

User

Act as a software developer. I have a library system application written in java and I have created some Gherkin Scenarios based on different features of the library application. I will provide you with these features/scenarios and the name of all the domain classes you will need, and then I will ask you to provide me the implementation code of the Step definitions for each feature, in java. I want your architecture to have Domain classes, Data Access Objects for saving/deleting entities and Service classes. Do you have any questions?

ChatGPT

No questions at the moment. Please provide the Gherkin scenarios, the list of domain classes, and any other relevant details for the library system application. Once I have that information, I'll proceed with implementing the Step definitions in Java, following the specified architecture with Domain classes, Data Access Objects (DAOs), and Service classes.

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, I'm ready to proceed. Please provide the Gherkin scenarios for the features of your library system application, and I'll start working on the implementation code for the 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: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

ChatGPT

Great, I'll start by outlining the structure of the Java code for handling these features. We'll have the following components:

1. ****Domain Classes****:

- `Book`
- `Borrower`
- `BorrowerCategory`
- `Item`
- `ItemState`
- `Loan`
- `Person`

2. ****Data Access Objects (DAOs)****:

- `BookDAO`
- `BorrowerDAO`
- `ItemDAO`
- `LoanDAO`
- `PersonDAO`

3. ****Services****:

- `BorrowerService`
- `LoanService`
- `NotificationService`

4. ****Step Definitions**** for the Gherkin scenarios.

I'll start by defining the domain classes, followed by the DAOs, services, and then the step definitions for the Gherkin scenarios.

Domain Classes

`Person.java`

```
```java
```

```
public class Person {
```

```
 private String name;
```

```
 private String email;
```

```
 // Getters and setters
```

```
}
```

```
```
```

`Borrower.java`

```
```java
```

```
public class Borrower extends Person {
```

```
 private String borrowerNumber;
```

```
 private BorrowerCategory category;
```

```
 private List<Loan> loans;
```

```
// Getters and setters

}

...

`BorrowerCategory.java`

```java

public class BorrowerCategory {

    private String categoryName;

    private int maxLendingDays;

    private int maxLendingLimit;

    private double finePerDay;

    // Getters and setters

}

...


```

```
#### `Book.java`

```java

public class Book {

 private String title;

 private String author;

 // Getters and setters


```

```
}
```

```
...
```

```
`Item.java`
```

```
```java
```

```
public class Item {
```

```
    private Book book;
```

```
    private ItemState state;
```

```
    // Getters and setters
```

```
}
```

```
...
```

```
#### `ItemState.java`
```

```
```java
```

```
public enum ItemState {
```

```
 AVAILABLE, BORROWED, LOST
```

```
}
```

```
...
```

```
`Loan.java`
```

```
```java
```

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

```
public class Loan {  
  
    private Item item;  
  
    private Borrower borrower;  
  
    private LocalDate loanDate;  
  
    private LocalDate dueDate;  
  
    private LocalDate returnDate;  
  
  
    // Getters and setters  
  
}  
...
```

Data Access Objects (DAOs)

`GenericDAO.java`

```java

import java.util.List;

```
public interface GenericDAO<T> {

 void save(T entity);

 void update(T entity);

 void delete(T entity);

 T findById(int id);
}
```

```
List<T> findAll();

}
...
```

#### `BorrowerDAO.java`

```
```java
```

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

```
public interface BorrowerDAO extends GenericDAO<Borrower> {
```

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

```
    List<Borrower> findByName(String name);
```

```
}
```

```
...
```

Services

`BorrowerService.java`

```
```java
```

```
public class BorrowerService {
```

```
 private BorrowerDAO borrowerDAO;
```

```
 public BorrowerService(BorrowerDAO borrowerDAO) {
```

```
 this.borrowerDAO = borrowerDAO;
```

```
}
```

```
public void registerBorrower(Borrower borrower) {

 if (borrowerDAO.findByBorrowerNumber(borrower.getBorrowerNumber()) != null) {

 throw new IllegalArgumentException("Borrower already exists");

 }

 borrowerDAO.save(borrower);

}
```

```
public void updateBorrower(Borrower borrower) {

 Borrower existingBorrower =
borrowerDAO.findByBorrowerNumber(borrower.getBorrowerNumber());

 if (existingBorrower == null) {

 throw new IllegalArgumentException("Borrower does not exist");

 }

 borrowerDAO.update(borrower);

}
```

```
public void deleteBorrower(String borrowerNumber) {

 Borrower borrower = borrowerDAO.findByBorrowerNumber(borrowerNumber);

 if (borrower == null) {

 throw new IllegalArgumentException("Borrower does not exist");

 }

 borrowerDAO.delete(borrower);

}
```

```
}
```

```
public Borrower getBorrower(String borrowerNumber) {
 return borrowerDAO.findByBorrowerNumber(borrowerNumber);
}
```

```
}
```

```
...
```

### Step Definitions for Gherkin Scenarios

#### `BorrowerSteps.java`

```
```java
```

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

```
import static org.junit.Assert.*;
```

```
public class BorrowerSteps {
```

```
    private BorrowerService borrowerService;
```

```
    private Borrower borrower;
```

```
    public BorrowerSteps() {
```

```
        BorrowerDAO borrowerDAO = new InMemoryBorrowerDAO();
```

```
        this.borrowerService = new BorrowerService(borrowerDAO);
```



```
}
```

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

```
public void george_red_is_not_registered_as_a_borrower() {
```

```
    borrower = borrowerService.getBorrower("GR123");
```

```
    assertNull(borrower);
```

```
}
```

```
@When("George Red 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() {
```

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

```
    borrower.setName("George Red");
```

```
    borrower.setBorrowerNumber("GR123");
```

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

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

```
}
```

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

```
public void the_system_successfully_stores_the_borrower_s_details() {
```

```
    Borrower storedBorrower = borrowerService.getBorrower("GR123");
```

```
    assertNotNull(storedBorrower);
```

```
        assertEquals("George Red", storedBorrower.getName());
    }
}
```

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

public void george_red_is_registered_as_a_borrower() {

    borrower = new Borrower();

    borrower.setName("George Red");

    borrower.setBorrowerNumber("GR123");

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

    borrowerService.registerBorrower(borrower);

    assertNotNull(borrowerService.getBorrower("GR123"));

}
}
```

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

```
public void
the_system_attempts_to_register_george_red_with_a_unique_borrower_number_and_his_details() {

    try {

        borrowerService.registerBorrower(borrower);

    } catch (IllegalArgumentException e) {

        assertEquals("Borrower already exists", e.getMessage());

    }

}
}
```

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

public void the_system_informs_that_the_user_already_exists() {

    // Exception handling in the previous step verifies this scenario

}
```

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

public void george_red_updates_his_borrowing_details() {

    borrower.setEmail("new.email@example.com");

    borrowerService.updateBorrower(borrower);

}
```

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

public void the_system_saves_the_changes() {

    Borrower updatedBorrower = borrowerService.getBorrower("GR123");

    assertEquals("new.email@example.com", updatedBorrower.getEmail());

}
```

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

public void george_red_tries_to_update_his_borrowing_details() {

    borrower = new Borrower();

    borrower.setName("George Red");

    borrower.setBorrowerNumber("GR123");

}
```

```
borrower.setEmail("new.email@example.com");

try {

    borrowerService.updateBorrower(borrower);

} 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 the_system_displays_an_error_message_indicating_that_george_red_does_not_exist() {

    // Exception handling in the previous step verifies this scenario

}
```

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

public void the_system_deletes_george_red_s_account() {

    borrowerService.deleteBorrower("GR123");

}
```

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

public void the_system_removes_george_red_s_details() {

    Borrower deletedBorrower = borrowerService.getBorrower("GR123");

    assertNull(deletedBorrower);

}
```

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

public void the_system_attempts_to_delete_george_red_s_account() {

    try {

        borrowerService.deleteBorrower("GR123");

    } catch (IllegalArgumentException e) {

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

    }

}
```

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

public void the_system_informs_that_the_borrower_does_not_exist() {

    // Exception handling in the previous step verifies this scenario

}
```

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

public void george_red_has_pending_items() {

    Loan loan = new Loan();

    loan.setBorrower(borrower);

    loan.setItem(new Item());

    borrower.getLoans().add(loan);

    assertTrue(borrower.getLoans().size() > 0);

}
```

```

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

public void the_system_does_not_remove_george_red_s_details() {

    Borrower existingBorrower = borrowerService.getBorrower("GR123");

    assertNotNull(existingBorrower);

}

@Then("the system informs about the pending items")

public void the_system_informs_about_the_pending_items() {

    // Handle this in the service layer logic when deleting

}

}

...

```

For the remaining features and their respective step definitions, the structure will be similar. The DAO implementations can be created in-memory for the sake of simplicity here, but they should ideally interface with

ChatGPT

a database in a real-world scenario.

If you need further assistance with specific step definitions or additional features, please let me know!

User

Based on the above, provide me with the step definitions for the feature: Loaning Items

ChatGPT

Sure, let's outline the `LoanService` first and then the step definitions for the feature: Loaning Items.

```
### `LoanService.java`
```

```
```java
```

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

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

```
public class LoanService {
```

```
 private BorrowerDAO borrowerDAO;
```

```
 private ItemDAO itemDAO;
```

```
 private LoanDAO loanDAO;
```

```
 public LoanService(BorrowerDAO borrowerDAO, ItemDAO itemDAO, LoanDAO loanDAO) {
```

```
 this.borrowerDAO = borrowerDAO;
```

```
 this.itemDAO = itemDAO;
```

```
 this.loanDAO = loanDAO;
```

```
 }
```

```
 public void loanItem(String borrowerNumber, String itemTitle) {
```

```
 Borrower borrower = borrowerDAO.findByBorrowerNumber(borrowerNumber);
```

```
 if (borrower == null) {
```

```
 throw new IllegalArgumentException("Borrower does not exist");
```

```

 }

 if (borrower.getLoans().size() >= borrower.getCategory().getMaxLendingLimit()) {

 throw new IllegalArgumentException("Lending limit reached");

 }

 Item item = itemDAO.findByTitle(itemTitle);

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

 throw new IllegalArgumentException("Item not available");

 }

 Loan loan = new Loan();

 loan.setBorrower(borrower);

 loan.setItem(item);

 loan.setLoanDate(LocalDate.now());

 loan.setDueDate(LocalDate.now().plusDays(borrower.getCategory().getMaxLendingDays()));

 loanDAO.save(loan);

 item.setState(ItemState.BORROWED);

 itemDAO.update(item);

 borrower.getLoans().add(loan);

 borrowerDAO.update(borrower);

}

}

...

```



### ### Step Definitions for Loaning Items

#### `LoanSteps.java`

```java

import io.cucumber.java.en.*;

import static org.junit.Assert.*;

public class LoanSteps {

private LoanService loanService;

private BorrowerService borrowerService;

private ItemDAO itemDAO;

private Borrower borrower;

private Item item1;

private Item item2;

private Exception exception;

public LoanSteps() {

BorrowerDAO borrowerDAO = new InMemoryBorrowerDAO();

this.itemDAO = new InMemoryItemDAO();

LoanDAO loanDAO = new InMemoryLoanDAO();

this.loanService = new LoanService(borrowerDAO, itemDAO, loanDAO);

```
    this.borrowerService = new BorrowerService(borrowerDAO);  
}
```

```
@Given("the library has the item {string} available")  
  
public void the_library_has_the_item_available(String itemTitle) {  
  
    item1 = new Item();  
  
    item1.setBook(new Book(itemTitle, "Author"));  
  
    item1.setState(ItemState.AVAILABLE);  
  
    itemDAO.save(item1);  
  
    assertNotNull(itemDAO.findByTitle(itemTitle));  
}
```

```
@Given("George Red is a registered borrower")  
  
public void george_red_is_a_registered_borrower() {  
  
    borrower = new Borrower();  
  
    borrower.setName("George Red");  
  
    borrower.setBorrowerNumber("GR123");  
  
    borrower.setEmail("george.red@example.com");  
  
    borrower.setCategory(new BorrowerCategory("Regular", 10, 5, 0.5));  
  
    borrowerService.registerBorrower(borrower);  
  
    assertNotNull(borrowerService.getBorrower("GR123"));  
}
```

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

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

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

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

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

```
        loan.setItem(new Item());
```

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

```
    }
```

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

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

```
}
```

```
@Given("George Red has been assigned a maximum lending limit of {int}")
```

```
public      void      george_red_has_been_assigned_a_maximum_lending_limit_of(int  
lendingLimit) {
```

```
    borrower.getCategory().setMaxLendingLimit(lendingLimit);
```

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

```
    assertEquals(lendingLimit, borrower.getCategory().getMaxLendingLimit());
```

```
}
```

```
@When("George Red borrows the item {string}")
```

```
public void george_red_borrows_the_item(String itemTitle) {
```

```
    try {
```

```
        loanService.loanItem(borrower.getBorrowerNumber(), itemTitle);
```

```
    } catch (Exception e) {  
  
        exception = e;  
  
    }  
  
}
```

```
@Then("the system successfully loans the item {string} to George Red with a due date set")  
  
public void the_system_successfully_loans_the_item_to_george_red_with_a_due_date_set(String  
itemTitle) {  
  
    assertNull(exception);  
  
    Loan loan = loanService.findLoanByBorrowerAndItem(borrower.getBorrowerNumber(),  
itemTitle);  
  
    assertNotNull(loan);  
  
    assertNotNull(loan.getDueDate());  
  
}
```

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

```
@When("George Red tries to borrow both items {string} and {string}")  
  
public void george_red_tries_to_borrow_both_items(String itemTitle1, String itemTitle2) {  
  
    item2 = new Item();  
  
}
```

```
item2.setBook(new Book(itemTitle2, "Author"));
```

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

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

```
try {
```

```
    loanService loanItem(borrower.getBorrowerNumber(), itemTitle1);
```

```
    loanService loanItem(borrower.getBorrowerNumber(), itemTitle2);
```

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

```
    exception = e;
```

```
}
```

```
}
```

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

```
public void
```

```
the_system_does_not_loan_to_george_red_due_to_the_lending_limit_reached(String  
itemTitle) {
```

```
    assertNotNull(exception);
```

```
    assertEquals("Lending limit reached", exception.getMessage());
```

```
}
```

```
@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 itemTitle) {
```

```
    // No action needed as the item is not added to the system
```

```
}
```

```
@When("George Red tries to borrow the item {string}")

public void george_red_tries_to_borrow_the_item(String itemTitle) {

    try {

        loanService loanItem(borrower.getBorrowerNumber(), itemTitle);

    } catch (Exception e) {

        exception = e;

    }

}
```

```
@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() {

    assertNotNull(exception);

    assertEquals("Item not available", exception.getMessage());

}
```

```
@Then("the system withdraws the item {string}")

public void the_system_withdraws_the_item(String itemTitle) {

    Item withdrawnItem = itemDAO.findByTitle(itemTitle);

    assertNotNull(withdrawnItem);

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

}
```

```

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

public void the_system_doesn_t_allow_the_loan() {

    assertNotNull(exception);

    assertEquals("Lending limit reached", exception.getMessage());

}

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

public void george_red_s_pending_items_remain(int pendingItems) {

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

}

}

'''

```

In this code, the step definitions for the `Loaning Items` feature are provided. The `LoanService` and `BorrowerService` are used to manage the loaning process and ensure the business logic is correctly followed according to the given Gherkin scenarios. The tests check the expected behavior, such as successful loaning, handling lending limits, and managing item availability.

User

Now, based on the above, create the step definitions for the feature: Returning a borrowed item in the library

ChatGPT

Certainly! Below are the step definitions for the feature: Returning a borrowed item in the library.

Step Definitions for Returning a Borrowed Item

`ReturnSteps.java`

```
```java
```

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

```
import static org.junit.Assert.*;
```

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

```
public class ReturnSteps {
```

```
 private LoanService loanService;
```

```
 private BorrowerService borrowerService;
```

```
 private ItemDAO itemDAO;
```

```
 private LoanDAO loanDAO;
```

```
 private Borrower borrower;
```

```
 private Item item;
```

```
 private Loan loan;
```

```
 private Exception exception;
```

```
 public ReturnSteps() {
```



```
BorrowerDAO borrowerDAO = new InMemoryBorrowerDAO();

this.itemDAO = new InMemoryItemDAO();

this.loanDAO = new InMemoryLoanDAO();

this.loanService = new LoanService(borrowerDAO, itemDAO, loanDAO);

this.borrowerService = new BorrowerService(borrowerDAO);

}
```

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

public void george_red_borrowed_the_item_days_prior_to_today_s_date(String
itemTitle, int daysPrior) {

 borrower = new Borrower();

 borrower.setName("George Red");

 borrower.setBorrowerNumber("GR123");

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

 borrower.setCategory(new BorrowerCategory("Regular", 10, 5, 0.5));

 borrowerService.registerBorrower(borrower);

 item = new Item();

 item.setBook(new Book(itemTitle, "Author"));

 item.setState(ItemState.BORROWED);

 itemDAO.save(item);

 loan = new Loan();

 loan.setBorrower(borrower);
```

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

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

```
loan.setDueDate(LocalDate.now().plusDays(borrower.getCategory().getMaxLendingDays() -
daysPrior));
```

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

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

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

```
}
```

```
@Given("George Red has been assigned maximum lending days of {int}")
```

```
public void george_red_has_been_assigned_maximum_lending_days_of(int
maxLendingDays) {
```

```
 borrower.getCategory().setMaxLendingDays(maxLendingDays);
```

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

```
 assertEquals(maxLendingDays, borrower.getCategory().getMaxLendingDays());
```

```
}
```

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

```
public void the_return_of_is_processed(String itemTitle) {
```

```
 try {
```

```
 Loan existingLoan =
```

```
loanDAO.findByBorrowerAndItem(borrower.getBorrowerNumber(), itemTitle);
```

```

 if (existingLoan == null) {

 throw new IllegalArgumentException("Loan does not exist");

 }

 existingLoan.setReturnDate(LocalDate.now());

 loanDAO.update(existingLoan);

 Item returnedItem = existingLoan.getItem();

 returnedItem.setState(ItemState.AVAILABLE);

 itemDAO.update(returnedItem);

 borrower.getLoans().remove(existingLoan);

 borrowerService.updateBorrower(borrower);

 } catch (Exception e) {

 exception = e;

 }

}

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

public void the_system_marks_the_state_of_as_available(String itemTitle) {

 assertNull(exception);

 Item returnedItem = itemDAO.findByTitle(itemTitle);

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

```

```
}
```

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

```
public void george_red_has_one_less_pending_item() {
```

```
 assertEquals(0, borrower.getLoans().size());
```

```
}
```

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

```
public void george_red_does_not_pay_a_fine() {
```

```
 // This step can be validated by ensuring no fine is calculated in the loan return process
```

```
}
```

```
@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() {
```

```
 Loan returnedLoan = loanDAO.findByBorrowerAndItem(borrower.getBorrowerNumber(),
item.getBook().getTitle());
```

```
 assertEquals(LocalDate.now(), returnedLoan.getReturnDate());
```

```
}
```

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

```
public void george_red_pays_a_fine_based_on_the_borrower_category() {
```

```
 double fine = (LocalDate.now().toEpochDay() - loan.getDueDate().toEpochDay()) *
borrower.getCategory().getFinePerDay();
```

```
 assertTrue(fine > 0);
```

```
}

}

...
```

In this code, the step definitions for the `Returning a borrowed item in the library` feature are provided. The `LoanService` and `BorrowerService` are used to manage the returning process and ensure the business logic is correctly followed according to the given Gherkin scenarios. The tests check the expected behavior, such as successful returns, handling fines for late returns, and updating the state of items.

If you need further assistance with specific step definitions or additional features, please let me know!

**User**

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

ChatGPT

Certainly! Below are the step definitions for the feature: Delayed Return Notification.

### Step Definitions for Delayed Return Notification

#### `NotificationSteps.java`

```
```java
```

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

```
import static org.junit.Assert.*;
```

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

```
public class NotificationSteps {
```

```
    private LoanService loanService;
```

```
    private BorrowerService borrowerService;
```

```
    private NotificationService notificationService;
```

```
    private ItemDAO itemDAO;
```

```
    private LoanDAO loanDAO;
```

```
    private Borrower borrower;
```

```
    private Item item;
```

```
    private Loan loan;
```

```
    private boolean emailSent;
```

```
    public NotificationSteps() {
```

```
        BorrowerDAO borrowerDAO = new InMemoryBorrowerDAO();
```

```
        this.itemDAO = new InMemoryItemDAO();
```

```
        this.loanDAO = new InMemoryLoanDAO();
```

```
        this.loanService = new LoanService(borrowerDAO, itemDAO, loanDAO);
```

```
        this.borrowerService = new BorrowerService(borrowerDAO);
```

```
        this.notificationService = new NotificationService();
```

```
    }
```

```
@Given("George Red has borrowed the item {string}")

public void george_red_has_borrowed_the_item(String itemTitle) {

    borrower = new Borrower();

    borrower.setName("George Red");

    borrower.setBorrowerNumber("GR123");

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

    borrower.setCategory(new BorrowerCategory("Regular", 10, 5, 0.5));

    borrowerService.registerBorrower(borrower);


    item = new Item();

    item.setBook(new Book(itemTitle, "Author"));

    item.setState(ItemState.BORROWED);

    itemDAO.save(item);


    loan = new Loan();

    loan.setBorrower(borrower);

    loan.setItem(item);

    loan.setLoanDate(LocalDate.now().minusDays(15)); // Set a past date

    loan.setDueDate(LocalDate.now().minusDays(5)); // Due date in the past

    loanDAO.save(loan);


    borrower.getLoans().add(loan);

    borrowerService.updateBorrower(borrower);
```

```
}
```

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

```
public void item_due_date_has_passed(String itemTitle) {
```

```
    Loan existingLoan = loanDAO.findByBorrowerAndItem(borrower.getBorrowerNumber(),  
itemTitle);
```

```
    assertTrue(LocalDate.now().isAfter(existingLoan.getDueDate()));
```

```
}
```

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

```
public void george_red_has_an_email_address() {
```

```
    assertNotNull(borrower.getEmail());
```

```
}
```

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

```
public void the_system_executes_the_delayed_return_notification_process() {
```

```
    emailSent = notificationService.sendDelayedReturnNotification(borrower, loan);
```

```
}
```

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

```
public void george_red_receives_an_email_notification_for_the_return_of_the_item() {
```

```
    assertTrue(emailSent);
```

```
}
```



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

```
public void george_red_does_not_have_an_email_address() {
```

```
    borrower.setEmail(null);
```

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

```
    assertNull(borrower.getEmail());
```

```
}
```

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

```
public void george_red_does_not_receive_an_email_notification_for_the_return_of_the_item() {
```

```
    assertFalse(emailSent);
```

```
}
```

```
}
```

```
...
```

```
### Notification Service
```

```
#### `NotificationService.java`
```

```
```java
```

```
public class NotificationService {
```

```
 public boolean sendDelayedReturnNotification(Borrower borrower, Loan loan) {
```

```
 if (borrower.getEmail() == null || borrower.getEmail().isEmpty()) {
```

```
 return false;
```

```
 }
```

```

 // Simulate sending an email

 System.out.println("Email sent to " + borrower.getEmail() + " about the delayed return of " + loan.getItem().getBook().getTitle());

 return true;
 }
}
...

```

### In-Memory DAOs for Testing

#### `InMemoryBorrowerDAO.java`

```
```java
```

```
import java.util.*;
```

```
public class InMemoryBorrowerDAO implements BorrowerDAO {
```

```
    private Map<String, Borrower> borrowerMap = new HashMap<>();
```

```
    @Override
```

```
    public void save(Borrower entity) {
```

```
        borrowerMap.put(entity.getBorrowerNumber(), entity);
```

```
    }
```

```
    @Override
```

```
public void update(Borrower entity) {  
  
    borrowerMap.put(entity.getBorrowerNumber(), entity);  
  
}
```

@Override

```
public void delete(Borrower entity) {  
  
    borrowerMap.remove(entity.getBorrowerNumber());  
  
}
```

@Override

```
public Borrower findById(int id) {  
  
    return null; // Not used in this example  
  
}
```

@Override

```
public List<Borrower> findAll() {  
  
    return new ArrayList<>(borrowerMap.values());  
  
}
```

@Override

```
public Borrower findByBorrowerNumber(String borrowerNumber) {  
  
    return borrowerMap.get(borrowerNumber);  
  
}
```

@Override

```
public List<Borrower> findByName(String name) {  
  
    List<Borrower> result = new ArrayList<>();  
  
    for (Borrower borrower : borrowerMap.values()) {  
  
        if (borrower.getName().equals(name)) {  
  
            result.add(borrower);  
  
        }  
  
    }  
  
    return result;  
  
}
```

...

`InMemoryItemDAO.java`

```java

import java.util.\*;

```
public class InMemoryItemDAO implements ItemDAO {

 private Map<String, Item> itemMap = new HashMap<>();

 @Override

 public void save(Item entity) {
```

```
 itemMap.put(entity.getBook().getTitle(), entity);
 }
}
```

```
@Override

public void update(Item entity) {

 itemMap.put(entity.getBook().getTitle(), entity);

}
```

```
@Override

public void delete(Item entity) {

 itemMap.remove(entity.getBook().getTitle());

}
```

```
@Override

public Item findById(int id) {

 return null; // Not used in this example

}
```

```
@Override

public List<Item> findAll() {

 return new ArrayList<>(itemMap.values());

}
```

```

@Override

public Item findByTitle(String title) {

 return itemMap.get(title);

}

}

...

```

#### `InMemoryLoanDAO.java`

```

```java

```

```

import java.util.*;

```

```

public class InMemoryLoanDAO implements LoanDAO {

    private Map<String, Loan> loanMap = new HashMap<>();

```

```

@Override

public void save(Loan entity) {

    loanMap.put(entity.getBorrower().getBorrowerNumber() + "-" +
entity.getItem().getBook().getTitle(), entity);

}

```

```

@Override

public void update(Loan entity) {

    loanMap.put(entity.getBorrower().getBorrowerNumber() + "-" +
entity.getItem().getBook().getTitle(), entity);

```

```
}
```

```
@Override
```

```
public void delete(Loan entity) {
```

```
    loanMap.remove(entity.getBorrower().getBorrowerNumber() + "-" +  
entity.getItem().getBook().getTitle());
```

```
}
```

```
@Override
```

```
public Loan findById(int id) {
```

```
    return null; // Not used in this example
```

```
}
```

```
@Override
```

```
public List<Loan> findAll() {
```

```
    return new ArrayList<>(loanMap.values());
```

```
}
```

```
@Override
```

```
public Loan findByBorrowerAndItem(String borrowerNumber, String itemTitle) {
```

```
    return loanMap.get(borrowerNumber + "-" + itemTitle);
```

```
}
```

```
}
```

```
...
```

In this code, the step definitions for the `Delayed Return Notification` feature are provided. The `LoanService`, `BorrowerService`, and `NotificationService` are used to manage the notification process and ensure the business logic is correctly followed according to the given Gherkin scenarios. The tests check the expected behavior, such as sending email notifications when the return is delayed and handling cases where the borrower does not have an email address.