In this chat, I provided the same information as in chat 2, and instructed the ai to generate the domain code first, which it did very easilly, and once again it understood the assigment extremelly well, created the domain / services/ DAOs code and started creating the step definitions of ALL the features with ONLY 2 prompts, so it has a strong memory of remembering the step definitions of eahc feature, and the code results were overall the same as in the earlier chats, with a lot of detail (not so close to the real code in many instances).

# Phase 2 Chat 3

**User**

**Act as a software developer. I have a library system application written in java and I have created some Gherkin Scenarios based on different features of the library application. I will provide you with these features/scenarios and the name of all the domain classes 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 so far. Please provide the Gherkin scenarios and the list of domain classes for your library system application. Once I have those details, I'll proceed with creating the implementation code for the Step definitions in Java, along with the necessary architecture for 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**

Got it. I'm ready for the Gherkin scenarios and features. Once you provide them, I'll start working on the implementation code for the Step definitions and the corresponding architecture.

**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 code , the services code and the DAOs code first\**

**ChatGPT**

Sure, let's start by creating the domain classes, the Data Access Objects (DAOs), and the Service classes for the library system.

### Domain Classes

#### Book.java

```java

public class Book {

private String title;

private String author;

private String isbn;

// Constructors, getters, setters, equals, and hashcode

}

```

#### Borrower.java

```java

public class Borrower {

private int borrowerNumber;

private String name;

private String email;

private BorrowerCategory category;

private List<Loan> loans;

// Constructors, getters, setters, equals, and hashcode

}

```

#### BorrowerCategory.java

```java

public class BorrowerCategory {

private String categoryName;

private int maxLendingLimit;

private int maxLendingDays;

private double finePerDay;

// Constructors, getters, setters, equals, and hashcode

}

```

#### Item.java

```java

public class Item {

private String id;

private Book book;

private ItemState state;

// Constructors, getters, setters, equals, and hashcode

}

```

#### ItemState.java

```java

public enum ItemState {

AVAILABLE,

BORROWED,

RESERVED,

LOST

}

```

#### Loan.java

```java

import java.util.Date;

public class Loan {

private Borrower borrower;

private Item item;

private Date loanDate;

private Date dueDate;

private Date returnDate;

// Constructors, getters, setters, equals, and hashcode

}

```

#### Person.java

```java

public class Person {

private String name;

private String address;

private String email;

// Constructors, getters, setters, equals, and hashcode

}

```

### Data Access Objects (DAOs)

#### BookDAO.java

```java

import java.util.List;

public interface BookDAO {

void save(Book book);

void delete(Book book);

Book findByISBN(String isbn);

List<Book> findAll();

}

```

#### BorrowerDAO.java

```java

import java.util.List;

public interface BorrowerDAO {

void save(Borrower borrower);

void delete(Borrower borrower);

Borrower findByBorrowerNumber(int borrowerNumber);

List<Borrower> findAll();

}

```

#### ItemDAO.java

```java

import java.util.List;

public interface ItemDAO {

void save(Item item);

void delete(Item item);

Item findById(String id);

List<Item> findAll();

}

```

#### LoanDAO.java

```java

import java.util.List;

public interface LoanDAO {

void save(Loan loan);

void delete(Loan loan);

Loan findById(int id);

List<Loan> findAll();

}

```

### Service Classes

#### BookService.java

```java

import java.util.List;

public class BookService {

private BookDAO bookDAO;

public BookService(BookDAO bookDAO) {

this.bookDAO = bookDAO;

}

public void registerBook(Book book) {

bookDAO.save(book);

}

public void removeBook(Book book) {

bookDAO.delete(book);

}

public Book findBookByISBN(String isbn) {

return bookDAO.findByISBN(isbn);

}

public List<Book> listAllBooks() {

return bookDAO.findAll();

}

}

```

#### BorrowerService.java

```java

import java.util.List;

public class BorrowerService {

private BorrowerDAO borrowerDAO;

public BorrowerService(BorrowerDAO borrowerDAO) {

this.borrowerDAO = borrowerDAO;

}

public void registerBorrower(Borrower borrower) {

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

if (existingBorrower == null) {

borrowerDAO.save(borrower);

} else {

throw new IllegalArgumentException("Borrower already registered");

}

}

public void updateBorrower(Borrower borrower) {

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

if (existingBorrower != null) {

borrowerDAO.save(borrower);

} else {

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

}

}

public void removeBorrower(Borrower borrower) {

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

if (existingBorrower != null && existingBorrower.getLoans().isEmpty()) {

borrowerDAO.delete(borrower);

} else if (existingBorrower != null) {

throw new IllegalArgumentException("Borrower has pending items");

} else {

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

}

}

public Borrower findBorrowerByNumber(int borrowerNumber) {

return borrowerDAO.findByBorrowerNumber(borrowerNumber);

}

public List<Borrower> listAllBorrowers() {

return borrowerDAO.findAll();

}

}

```

#### LoanService.java

```java

import java.util.Date;

import java.util.List;

public class LoanService {

private LoanDAO loanDAO;

private BorrowerDAO borrowerDAO;

private ItemDAO itemDAO;

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

this.loanDAO = loanDAO;

this.borrowerDAO = borrowerDAO;

this.itemDAO = itemDAO;

}

public void loanItem(Borrower borrower, Item item) {

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

throw new IllegalArgumentException("Borrower has reached the maximum lending limit");

}

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

throw new IllegalArgumentException("Item is not available for loan");

}

Loan loan = new Loan();

loan.setBorrower(borrower);

loan.setItem(item);

loan.setLoanDate(new Date());

loan.setDueDate(new Date(System.currentTimeMillis() + borrower.getCategory().getMaxLendingDays() \* 24 \* 60 \* 60 \* 1000L));

loanDAO.save(loan);

item.setState(ItemState.BORROWED);

itemDAO.save(item);

borrower.getLoans().add(loan);

borrowerDAO.save(borrower);

}

public void returnItem(Loan loan) {

Item item = loan.getItem();

item.setState(ItemState.AVAILABLE);

itemDAO.save(item);

Borrower borrower = loan.getBorrower();

borrower.getLoans().remove(loan);

borrowerDAO.save(borrower);

loan.setReturnDate(new Date());

loanDAO.save(loan);

if (loan.getReturnDate().after(loan.getDueDate())) {

long daysLate = (loan.getReturnDate().getTime() - loan.getDueDate().getTime()) / (24 \* 60 \* 60 \* 1000);

double fine = daysLate \* borrower.getCategory().getFinePerDay();

// Implement fine logic (e.g., record it, notify borrower, etc.)

}

}

public List<Loan> listAllLoans() {

return loanDAO.findAll();

}

}

```

With these domain classes, DAOs, and service classes in place, the next step is to implement the Step definitions for the Gherkin scenarios in Java. Let me know if you would like to proceed with this or need any adjustments in the current setup.

**User**

**proceed**

**ChatGPT**

Great, let's proceed with the implementation of the Step definitions for the Gherkin scenarios in Java using Cucumber. We'll start with the feature for handling borrowers.

### Step Definitions for Borrower Handling

#### BorrowerSteps.java

```java

import static org.junit.Assert.\*;

import java.util.HashMap;

import java.util.Map;

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

import my.library.domain.Borrower;

import my.library.domain.BorrowerCategory;

import my.library.services.BorrowerService;

public class BorrowerSteps {

private BorrowerService borrowerService;

private Map<Integer, Borrower> borrowers;

private BorrowerCategory defaultCategory;

private Borrower currentBorrower;

private String errorMessage;

public BorrowerSteps() {

borrowerService = new BorrowerService(new InMemoryBorrowerDAO());

borrowers = new HashMap<>();

defaultCategory = new BorrowerCategory("Regular", 5, 14, 0.5);

}

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

public void borrower\_is\_not\_registered(String name) {

currentBorrower = borrowerService.findBorrowerByName(name);

assertNull(currentBorrower);

}

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

public void borrower\_gets\_registered(String name) {

try {

Borrower newBorrower = new Borrower(borrowers.size() + 1, name, name.toLowerCase() + "@example.com", defaultCategory, new ArrayList<>());

borrowerService.registerBorrower(newBorrower);

currentBorrower = newBorrower;

borrowers.put(newBorrower.getBorrowerNumber(), newBorrower);

} catch (IllegalArgumentException e) {

errorMessage = e.getMessage();

}

}

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

public void system\_successfully\_stores\_details() {

assertNotNull(currentBorrower);

assertTrue(borrowers.containsKey(currentBorrower.getBorrowerNumber()));

}

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

public void borrower\_is\_registered(String name) {

Borrower registeredBorrower = new Borrower(borrowers.size() + 1, name, name.toLowerCase() + "@example.com", defaultCategory, new ArrayList<>());

borrowerService.registerBorrower(registeredBorrower);

borrowers.put(registeredBorrower.getBorrowerNumber(), registeredBorrower);

currentBorrower = registeredBorrower;

}

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

public void system\_attempts\_to\_register(String name) {

try {

Borrower duplicateBorrower = new Borrower(borrowers.size() + 1, name, name.toLowerCase() + "@example.com", defaultCategory, new ArrayList<>());

borrowerService.registerBorrower(duplicateBorrower);

} catch (IllegalArgumentException e) {

errorMessage = e.getMessage();

}

}

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

public void system\_informs\_user\_already\_exists() {

assertEquals("Borrower already registered", errorMessage);

}

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

public void borrower\_updates\_details(String name) {

try {

currentBorrower.setEmail(name.toLowerCase() + ".updated@example.com");

borrowerService.updateBorrower(currentBorrower);

} catch (IllegalArgumentException e) {

errorMessage = e.getMessage();

}

}

@Then("the system saves the changes")

public void system\_saves\_changes() {

assertEquals(currentBorrower.getEmail(), currentBorrower.getName().toLowerCase() + ".updated@example.com");

}

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

public void borrower\_tries\_to\_update\_details(String name) {

try {

Borrower nonExistentBorrower = new Borrower(borrowers.size() + 1, name, name.toLowerCase() + ".updated@example.com", defaultCategory, new ArrayList<>());

borrowerService.updateBorrower(nonExistentBorrower);

} catch (IllegalArgumentException e) {

errorMessage = e.getMessage();

}

}

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

public void system\_displays\_error\_message(String name) {

assertEquals("Borrower does not exist", errorMessage);

}

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

public void system\_deletes\_account(String name) {

try {

borrowerService.removeBorrower(currentBorrower);

borrowers.remove(currentBorrower.getBorrowerNumber());

} catch (IllegalArgumentException e) {

errorMessage = e.getMessage();

}

}

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

public void system\_removes\_details(String name) {

assertFalse(borrowers.containsKey(currentBorrower.getBorrowerNumber()));

}

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

public void system\_attempts\_to\_delete\_account(String name) {

try {

Borrower nonExistentBorrower = new Borrower(borrowers.size() + 1, name, name.toLowerCase() + "@example.com", defaultCategory, new ArrayList<>());

borrowerService.removeBorrower(nonExistentBorrower);

} catch (IllegalArgumentException e) {

errorMessage = e.getMessage();

}

}

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

public void system\_informs\_borrower\_does\_not\_exist() {

assertEquals("Borrower does not exist", errorMessage);

}

@Given("{string} has pending items")

public void borrower\_has\_pending\_items(String name) {

Loan pendingLoan = new Loan(currentBorrower, new Item("1", new Book("Harry Potter", "J.K. Rowling", "1234567890"), ItemState.BORROWED), new Date(), null, null);

currentBorrower.getLoans().add(pendingLoan);

}

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

public void system\_does\_not\_remove\_details(String name) {

assertTrue(borrowers.containsKey(currentBorrower.getBorrowerNumber()));

}

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

public void system\_informs\_pending\_items() {

assertEquals("Borrower has pending items", errorMessage);

}

}

```

### In-Memory DAO Implementation for Testing

#### InMemoryBorrowerDAO.java

```java

import java.util.\*;

public class InMemoryBorrowerDAO implements BorrowerDAO {

private Map<Integer, Borrower> borrowers = new HashMap<>();

@Override

public void save(Borrower borrower) {

borrowers.put(borrower.getBorrowerNumber(), borrower);

}

@Override

public void delete(Borrower borrower) {

borrowers.remove(borrower.getBorrowerNumber());

}

@Override

public Borrower findByBorrowerNumber(int borrowerNumber) {

return borrowers.get(borrowerNumber);

}

@Override

public List<Borrower> findAll() {

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

}

}

```

This setup includes the step definitions for handling borrowers using Cucumber in Java, as well as an in-memory DAO implementation for testing purposes. You can now write similar step definitions for the other features.

Let me know if you need any further assistance with the other features or any other part of the implementation!

**User**

**provide all the other features step definitions as well**

**ChatGPT**

Sure, I'll continue by providing the step definitions for the remaining features: Delayed Return Notification, Loaning items, and Returning a borrowed item to the library.

### Step Definitions for Delayed Return Notification

#### DelayedReturnNotificationSteps.java

```java

import static org.junit.Assert.\*;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

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

import my.library.domain.\*;

import my.library.services.\*;

public class DelayedReturnNotificationSteps {

private BorrowerService borrowerService;

private LoanService loanService;

private Map<Integer, Borrower> borrowers;

private Item item;

private Loan loan;

private String emailMessage;

public DelayedReturnNotificationSteps() {

borrowerService = new BorrowerService(new InMemoryBorrowerDAO());

loanService = new LoanService(new InMemoryLoanDAO(), new InMemoryBorrowerDAO(), new InMemoryItemDAO());

borrowers = new HashMap<>();

}

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

public void borrower\_has\_borrowed\_item(String borrowerName, String itemName) {

Borrower borrower = new Borrower(borrowers.size() + 1, borrowerName, borrowerName.toLowerCase() + "@example.com", new BorrowerCategory("Regular", 5, 14, 0.5), new ArrayList<>());

borrowers.put(borrower.getBorrowerNumber(), borrower);

item = new Item("1", new Book(itemName, "Author", "1234567890"), ItemState.BORROWED);

loan = new Loan(borrower, item, new Date(System.currentTimeMillis() - 10 \* 24 \* 60 \* 60 \* 1000L), new Date(System.currentTimeMillis() - 5 \* 24 \* 60 \* 60 \* 1000L), null);

borrower.getLoans().add(loan);

}

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

public void due\_date\_has\_passed(String itemName) {

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

}

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

public void borrower\_has\_email\_address(String borrowerName) {

assertNotNull(borrowers.get(1).getEmail());

}

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

public void system\_executes\_notification\_process() {

if (loan.getDueDate().before(new Date())) {

emailMessage = "Dear " + loan.getBorrower().getName() + ", you have delayed the return of the item " + loan.getItem().getBook().getTitle() + ". Please return it as soon as possible.";

}

}

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

public void borrower\_receives\_email\_notification(String borrowerName) {

assertNotNull(emailMessage);

}

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

public void borrower\_does\_not\_have\_email\_address(String borrowerName) {

Borrower borrower = borrowers.get(1);

borrower.setEmail(null);

}

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

public void borrower\_does\_not\_receive\_email\_notification(String borrowerName) {

assertNull(emailMessage);

}

}

```

### Step Definitions for Loaning Items

#### LoaningItemsSteps.java

```java

import static org.junit.Assert.\*;

import java.util.\*;

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

import my.library.domain.\*;

import my.library.services.\*;

public class LoaningItemsSteps {

private BorrowerService borrowerService;

private LoanService loanService;

private Map<Integer, Borrower> borrowers;

private Map<String, Item> items;

private String errorMessage;

public LoaningItemsSteps() {

borrowerService = new BorrowerService(new InMemoryBorrowerDAO());

loanService = new LoanService(new InMemoryLoanDAO(), new InMemoryBorrowerDAO(), new InMemoryItemDAO());

borrowers = new HashMap<>();

items = new HashMap<>();

}

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

public void library\_has\_item\_available(String itemName) {

Item item = new Item(UUID.randomUUID().toString(), new Book(itemName, "Author", "1234567890"), ItemState.AVAILABLE);

items.put(itemName, item);

}

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

public void borrower\_is\_registered(String borrowerName) {

Borrower borrower = new Borrower(borrowers.size() + 1, borrowerName, borrowerName.toLowerCase() + "@example.com", new BorrowerCategory("Regular", 5, 14, 0.5), new ArrayList<>());

borrowerService.registerBorrower(borrower);

borrowers.put(borrower.getBorrowerNumber(), borrower);

}

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

public void borrower\_has\_pending\_items(String borrowerName, int pendingItems) {

Borrower borrower = borrowers.get(1);

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

Loan loan = new Loan(borrower, new Item(UUID.randomUUID().toString(), new Book("Book" + i, "Author", "1234567890"), ItemState.BORROWED), new Date(), null, null);

borrower.getLoans().add(loan);

}

}

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

public void borrower\_has\_max\_lending\_limit(String borrowerName, int maxLimit) {

Borrower borrower = borrowers.get(1);

BorrowerCategory category = new BorrowerCategory("Regular", maxLimit, 14, 0.5);

borrower.setCategory(category);

}

@When("{string} borrows the item {string}")

public void borrower\_borrows\_item(String borrowerName, String itemName) {

Borrower borrower = borrowers.get(1);

Item item = items.get(itemName);

try {

loanService.loanItem(borrower, item);

} catch (IllegalArgumentException e) {

errorMessage = e.getMessage();

}

}

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

public void system\_successfully\_loans\_item(String itemName, String borrowerName) {

Borrower borrower = borrowers.get(1);

assertTrue(borrower.getLoans().stream().anyMatch(loan -> loan.getItem().getBook().getTitle().equals(itemName)));

}

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

public void borrowers\_pending\_items\_increase\_to(String borrowerName, int expectedPendingItems) {

Borrower borrower = borrowers.get(1);

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

}

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

public void borrower\_tries\_to\_borrow\_both\_items(String borrowerName, List<String> itemNames) {

Borrower borrower = borrowers.get(1);

for (String itemName : itemNames) {

Item item = items.get(itemName);

try {

loanService.loanItem(borrower, item);

} catch (IllegalArgumentException e) {

errorMessage = e.getMessage();

}

}

}

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

public void system\_successfully\_loans\_first\_item(String itemName, String borrowerName) {

Borrower borrower = borrowers.get(1);

assertTrue(borrower.getLoans().stream().anyMatch(loan -> loan.getItem().getBook().getTitle().equals(itemName)));

}

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

public void system\_does\_not\_loan\_second\_item(String itemName, String borrowerName) {

assertEquals("Borrower has reached the maximum lending limit", errorMessage);

}

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

public void item\_in\_library\_but\_not\_in\_system(String itemName) {

items.remove(itemName);

}

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

public void borrower\_tries\_to\_borrow\_nonexistent\_item(String borrowerName, String itemName) {

Borrower borrower = borrowers.get(1);

Item item = new Item(UUID.randomUUID().toString(), new Book(itemName, "Author", "1234567890"), ItemState.AVAILABLE);

try {

loanService.loanItem(borrower, item);

} catch (IllegalArgumentException e) {

errorMessage = e.getMessage();

}

}

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

public void system\_returns\_error\_due\_to\_item\_status() {

assertEquals("Item is not available for loan", errorMessage);

}

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

public void system\_withdraws\_item(String itemName) {

Item item = items.get(itemName);

item.setState(ItemState.LOST);

items.put(itemName, item);

}

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

public void borrower\_tries\_to\_borrow\_item(String borrowerName, String itemName) {

Borrower borrower = borrowers.get(1);

Item item = items.get(itemName);

try {

loanService.loanItem(borrower, item);

} catch (IllegalArgumentException e) {

errorMessage = e.getMessage();

}

}

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

**ChatGPT**

")

public void system\_doesnt\_allow\_loan() {

assertEquals("Borrower has reached the maximum lending limit", errorMessage);

}

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

public void borrowers\_pending\_items\_remain(String borrowerName, int expectedPendingItems) {

Borrower borrower = borrowers.get(1);

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

}

}

```

### Step Definitions for Returning a Borrowed Item

#### ReturningItemsSteps.java

```java

import static org.junit.Assert.\*;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

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

import my.library.domain.\*;

import my.library.services.\*;

public class ReturningItemsSteps {

private BorrowerService borrowerService;

private LoanService loanService;

private Map<Integer, Borrower> borrowers;

private Item item;

private Loan loan;

private String fineMessage;

public ReturningItemsSteps() {

borrowerService = new BorrowerService(new InMemoryBorrowerDAO());

loanService = new LoanService(new InMemoryLoanDAO(), new InMemoryBorrowerDAO(), new InMemoryItemDAO());

borrowers = new HashMap<>();

}

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

public void borrower\_borrowed\_item\_days\_prior(String borrowerName, String itemName, int daysAgo) {

Borrower borrower = new Borrower(borrowers.size() + 1, borrowerName, borrowerName.toLowerCase() + "@example.com", new BorrowerCategory("Regular", 5, 14, 0.5), new ArrayList<>());

borrowers.put(borrower.getBorrowerNumber(), borrower);

item = new Item("1", new Book(itemName, "Author", "1234567890"), ItemState.BORROWED);

loan = new Loan(borrower, item, new Date(System.currentTimeMillis() - daysAgo \* 24 \* 60 \* 60 \* 1000L), new Date(System.currentTimeMillis() - (daysAgo - 5) \* 24 \* 60 \* 60 \* 1000L), null);

borrower.getLoans().add(loan);

}

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

public void borrower\_has\_max\_lending\_days(String borrowerName, int maxDays) {

Borrower borrower = borrowers.get(1);

BorrowerCategory category = new BorrowerCategory("Regular", 5, maxDays, 0.5);

borrower.setCategory(category);

}

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

public void return\_of\_item\_is\_processed(String itemName) {

try {

loanService.returnItem(loan);

} catch (IllegalArgumentException e) {

fineMessage = e.getMessage();

}

}

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

public void system\_marks\_state\_as\_available(String itemName) {

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

}

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

public void borrower\_has\_one\_less\_pending\_item(String borrowerName) {

Borrower borrower = borrowers.get(1);

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

}

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

public void borrower\_does\_not\_pay\_fine(String borrowerName) {

assertNull(fineMessage);

}

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

public void return\_date\_is\_set\_to\_today() {

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

}

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

public void borrower\_pays\_fine\_based\_on\_category(String borrowerName) {

assertNotNull(fineMessage);

}

}

```

### In-Memory DAO Implementations for Testing

To ensure our step definitions work, let's provide in-memory implementations for `LoanDAO` and `ItemDAO`.

#### InMemoryLoanDAO.java

```java

import java.util.\*;

public class InMemoryLoanDAO implements LoanDAO {

private Map<Integer, Loan> loans = new HashMap<>();

@Override

public void save(Loan loan) {

loans.put(loans.size() + 1, loan);

}

@Override

public void delete(Loan loan) {

loans.values().remove(loan);

}

@Override

public Loan findById(int id) {

return loans.get(id);

}

@Override

public List<Loan> findAll() {

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

}

}

```

#### InMemoryItemDAO.java

```java

import java.util.\*;

public class InMemoryItemDAO implements ItemDAO {

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

@Override

public void save(Item item) {

items.put(item.getId(), item);

}

@Override

public void delete(Item item) {

items.remove(item.getId());

}

@Override

public Item findById(String id) {

return items.get(id);

}

@Override

public List<Item> findAll() {

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

}

}

```

### Summary

This setup includes the step definitions for all the specified features and scenarios in the library system, as well as the in-memory DAO implementations for testing purposes. You can now use these step definitions to run your Cucumber tests.

Let me know if you need further assistance or adjustments!