In this chat, I provided the Ai with all the features of the system, the domain classes with their respective attributes and method names as well as with the description of the system and asked the Ai to first provide the domain code (which is unnecessary because it does it even if I don’t instruct it to) of the system. As always with this LLM, it understood the assigment extremelly good, provided all the code for the domain/daos/services with a lot of details and then it provided me with the step deifnitions code, which in many cases/scenarios, was exactly the same as the real solution, and in some other cases it generated a better solution as well. Overall, the results were pretty good, it required very little prompts and was very good on understanding what to do.

# Phase 4 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 code of all the domain classes with their attributes and methods that you 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**

That sounds like a well-structured approach to your library system application. Before we start, I'd like to clarify a few things to ensure that I can provide you with accurate and efficient help:

1. \*\*Scope of Scenarios:\*\* How many Gherkin scenarios do you have, and what features do they cover? This will help me understand the breadth of the implementation needed.

2. \*\*Existing Domain Classes:\*\* Please provide the code of the domain classes that you have. This includes their attributes and methods.

3. \*\*Persistence Mechanism:\*\* Are you using a specific framework for Data Access Objects (DAOs), such as JPA, Hibernate, or another library? This affects how the DAOs will be implemented.

4. \*\*Service Classes:\*\* Do you already have service interfaces or classes defined, or should I create them from scratch based on the domain classes and scenarios?

5. \*\*Testing Framework:\*\* Are you using a specific framework for your Gherkin scenarios? Common options include Cucumber or JBehave. This impacts how the step definitions are written.

Once you provide the Gherkin scenarios and the existing code for the domain classes, I'll be able to create the step definitions for each feature, along with the necessary DAOs and service classes.

**User**

**The domain classes are:**

**1)class "Book" with the attributes:**

**private ISBN isbn,**

**private String title,**

**private String publication,**

**private int publicationyear,**

**private Publisher publisher,**

**private Set<Item> items = new HashSet<Item>(),**

**private Set<Author> authors = new HashSet<Author>(),**

**and with the methods:**

**Setters and Getters for every attribute and a constructor,**

**public Set<Item> getItems(),**

**public void addItem(Item item),**

**public void removeItem(Item item),**

**Set<Item> friendItems(),**

**public void addAuthor(Author author),**

**public void removeAuthor(Author author),**

**Set<Author> friendAuthors()**

**2)class "Borrower" with the attributes:**

**private int borrowerNo,**

**private TelephoneNumber telephone,**

**private EmailAddress emailAddress,**

**private Address address,**

**private BorrowerCategory category,**

**private Person person = new Person(),**

**private Set<Loan> loans = new HashSet<Loan>(),**

**and with the methods:**

**Setters and Getters for every attribute and a constructor,**

**public Set<Loan> getLoans(),**

**Set<Loan> friendLoans(),**

**public int countPendingItems(),**

**public boolean canBorrow(),**

**public LocalDate getLoanDue(LocalDate loanDate),**

**public Money getDailyFine()**

**3)class "BorrowerCategory" with the attributes:**

**private String description,**

**private int maxLendingItems,**

**private int maxLendingDays,**

**private Money dailyFine,**

**and with the methods:**

**Setters and Getters for every attribute and a constructor,**

**public LocalDate getLoanDue(LocalDate loanDate),**

**public boolean canBorrow(int pendingItems)**

**4)class "Item" with the attributes:**

**private int itemNumber = 0,**

**private Book book,**

**private ItemState state = ItemState.NEW**

**and with the methods:**

**Setters and Getters for every attribute and a constructor,**

**public Loan borrow(Borrower borrower),**

**public void available(),**

**public void withdraw(),**

**public void lost(),**

**5)enum class "ItemState" with the Enum values :**

**NEW,AVAILABLE,LOANED,LOST,WITHDRAWN**

**6)class "Loan" with the attributes:**

**private LocalDate loanDate = SystemDate.now(),**

**private LocalDate returnDate,**

**private Borrower borrower,**

**private Item item**

**and with the methods:**

**Setters and Getters for every attribute and a constructor,**

**public LocalDate getDue(),**

**public boolean isPending(),**

**public void returnItem(),**

**public boolean isOverdue(),**

**public long daysToDue(),**

**public long getOverdue(),**

**public Money getFine(),**

**7)class "Person" with the attributes:**

**private String firstName,**

**private String lastName,**

**and with the methods:**

**Setters and Getters for every attribute and a constructor**

**In the next message I will provide you with the features/scenarios**

**ChatGPT**

Great, thank you for providing the domain classes. I will now wait for the features/scenarios in Gherkin syntax. Once you provide those, I will proceed with the implementation of the step definitions in Java, along with the necessary DAOs and service classes to support those scenarios.

**User**

**The Gherkin scenarios are :**

**The first feature is :**

**Feature: Borrower handling by the system**

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

**User Story: As a librarian,**

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

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

**Scenario: Registering a new borrower**

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

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

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

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

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

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

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

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

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

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

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

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

**When George Red updates his borrowing details**

**Then the system saves the changes**

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

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

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

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

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

**Scenario: Deleting a borrower**

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

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

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

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

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

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

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

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

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

**Scenario: Handling unreturned items of books**

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

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

**And George Red has pending items**

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

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

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

**The second feature is :**

**Feature: Delayed Return Notification**

**User Story**

**a Library Manager**

**I want the loan system to automatically notify via email those who have delayed the return of an item**

**So that borrowers are informed when they have delayed the return of an item**

**Scenario: Notifying the borrower via email**

**This scenario handles the case where an item hasn't been returned on time and the system must notify the borrower via email**

**Given George Red has borrowed the item Harry Potter**

**And Harry Potter's due date has passed**

**And George Red has an email address**

**When the system executes the delayed return notification process**

**Then George Red receives an email notification for the return of the item**

**Scenario: Borrower does not have an email address**

**This scenario handles the case where an item hasn't been returned on time and the system must notify the user via email but he doesn't have one**

**Given George Red has borrowed the item Harry Potter**

**And Harry Potter's due date has passed**

**And George Red does not have an email address**

**When the system executes the delayed return notification process**

**Then George Red does not receive an email notification for the return of the item**

**The third feature is :**

**Feature: Loaning items**

**The library application allows for the librarian to loan an item of a book to a borrower based**

**on some conditions of the system**

**User story: As a library member**

**I want to be able to borrow items**

**So that I can study them at home**

**Scenario: Successful loaning of an item**

**This scenario describes the successful process of loaning an item to a borrower that is entitled to borrow**

**Given the library has the item Harry Potter available**

**And George Red is a registered borrower**

**And George Red has 2 pending items to be returned**

**And George Red has been assigned a maximum lending limit of 5**

**When George Red borrows the item Harry Potter**

**Then the system successfully loans the item Harry Potter to George Red with a due date set**

**And George Red's pending items increase to 3**

**Scenario: Borrower can borrow only one item due to his lending limit**

**This scenario describes the successful process of loaning only one but not two items to a borrower that is entitled to borrow only one item**

**Given the library has the items Harry Potter and Moby Dick available**

**And George Red is a registered borrower**

**And George Red has 2 pending items to be returned**

**And George Red has been assigned a maximum lending limit of 3**

**When George Red tries to borrow both items**

**Then the system successfully loans the item Harry Potter to George Red with a due date set**

**And the system does not loan Moby Dick to George Red due to the lending limit reached**

**And George Red's pending items increase to 3**

**Scenario: Item not found**

**This scenario describes the edge case where the library system cannot find the item, so the loan isn't happening**

**Given the item Harry Potter is in the library but not in the system**

**And George Red is a registered borrower**

**When George Red tries to borrow the item Harry Potter**

**Then the system returns an error due to the item's status**

**And the system withdraws the item Harry Potter**

**Scenario: The borrower is not eligible to borrow**

**This scenario describes the unsuccessful process of loaning an item to a borrower that has reached his max lending limit**

**Given the library has the item Harry Potter available**

**And George Red is a registered borrower**

**And George Red has 3 pending items to be returned**

**And George Red has been assigned a maximum lending limit of 3**

**When George Red tries to borrow the item Harry Potter**

**Then the system doesn't allow the loan**

**And George Red's pending items remain 3**

**and the last feature is :**

**Feature: Returning a borrowed item to the library**

**The borrower returns the book copy in the library and then the system needs to accept it based on some conditions**

**User story: As a librarian**

**I want to accurately record the return of one or more books**

**So that books are returned on time, and late returns are fined**

**Scenario: Successful return of an item**

**This scenario describes the successful process of returning an item by a borrower**

**Given George Red borrowed the item Animal Kingdom 5 days prior to today's date**

**And George Red has been assigned maximum lending days of 10**

**When the return of Animal Kingdom is processed**

**Then the system marks the state of Animal Kingdom as AVAILABLE**

**And George Red has one less pending item**

**And George Red does not pay a fine**

**And the return date of the loan is set to today's date**

**Scenario: Late return of an item from a borrower**

**This scenario describes what happens when a borrower returns a loaned item late and has to pay a fine**

**Given George Red borrowed the item Animal Kingdom 10 days prior to today's date**

**And George Red has been assigned maximum lending days of 7**

**When the return of Animal Kingdom is processed**

**Then the system marks the state of Animal Kingdom as AVAILABLE**

**And George Red pays a fine based on the borrower category**

**And the return date of the loan is set to today's date**

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

**ChatGPT**

Sure, let's start by creating the domain classes based on the provided information. We'll then create the DAO classes for each entity and the corresponding service classes.

### Domain Classes

#### Book.java

```java

import java.util.HashSet;

import java.util.Set;

public class Book {

private String isbn;

private String title;

private String publication;

private int publicationYear;

private Publisher publisher;

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

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

// Constructors, Getters, and Setters

public Book(String isbn, String title, String publication, int publicationYear, Publisher publisher) {

this.isbn = isbn;

this.title = title;

this.publication = publication;

this.publicationYear = publicationYear;

this.publisher = publisher;

}

public String getIsbn() {

return isbn;

}

public void setIsbn(String isbn) {

this.isbn = isbn;

}

// Other getters and setters...

public Set<Item> getItems() {

return items;

}

public void addItem(Item item) {

items.add(item);

}

public void removeItem(Item item) {

items.remove(item);

}

public Set<Item> friendItems() {

return items;

}

public void addAuthor(Author author) {

authors.add(author);

}

public void removeAuthor(Author author) {

authors.remove(author);

}

public Set<Author> friendAuthors() {

return authors;

}

}

```

#### Borrower.java

```java

import java.time.LocalDate;

import java.util.HashSet;

import java.util.Set;

public class Borrower {

private int borrowerNo;

private TelephoneNumber telephone;

private EmailAddress emailAddress;

private Address address;

private BorrowerCategory category;

private Person person = new Person();

private Set<Loan> loans = new HashSet<>();

// Constructors, Getters, and Setters

public Borrower(int borrowerNo, TelephoneNumber telephone, EmailAddress emailAddress, Address address, BorrowerCategory category, Person person) {

this.borrowerNo = borrowerNo;

this.telephone = telephone;

this.emailAddress = emailAddress;

this.address = address;

this.category = category;

this.person = person;

}

public int getBorrowerNo() {

return borrowerNo;

}

public void setBorrowerNo(int borrowerNo) {

this.borrowerNo = borrowerNo;

}

// Other getters and setters...

public Set<Loan> getLoans() {

return loans;

}

public Set<Loan> friendLoans() {

return loans;

}

public int countPendingItems() {

return loans.size();

}

public boolean canBorrow() {

return category.canBorrow(countPendingItems());

}

public LocalDate getLoanDue(LocalDate loanDate) {

return category.getLoanDue(loanDate);

}

public Money getDailyFine() {

return category.getDailyFine();

}

}

```

#### BorrowerCategory.java

```java

import java.time.LocalDate;

public class BorrowerCategory {

private String description;

private int maxLendingItems;

private int maxLendingDays;

private Money dailyFine;

// Constructors, Getters, and Setters

public BorrowerCategory(String description, int maxLendingItems, int maxLendingDays, Money dailyFine) {

this.description = description;

this.maxLendingItems = maxLendingItems;

this.maxLendingDays = maxLendingDays;

this.dailyFine = dailyFine;

}

public String getDescription() {

return description;

}

public void setDescription(String description) {

this.description = description;

}

// Other getters and setters...

public LocalDate getLoanDue(LocalDate loanDate) {

return loanDate.plusDays(maxLendingDays);

}

public boolean canBorrow(int pendingItems) {

return pendingItems < maxLendingItems;

}

}

```

#### Item.java

```java

public class Item {

private int itemNumber;

private Book book;

private ItemState state = ItemState.NEW;

// Constructors, Getters, and Setters

public Item(int itemNumber, Book book) {

this.itemNumber = itemNumber;

this.book = book;

}

public int getItemNumber() {

return itemNumber;

}

public void setItemNumber(int itemNumber) {

this.itemNumber = itemNumber;

}

// Other getters and setters...

public Loan borrow(Borrower borrower) {

this.state = ItemState.LOANED;

return new Loan(borrower, this);

}

public void available() {

this.state = ItemState.AVAILABLE;

}

public void withdraw() {

this.state = ItemState.WITHDRAWN;

}

public void lost() {

this.state = ItemState.LOST;

}

}

```

#### ItemState.java

```java

public enum ItemState {

NEW, AVAILABLE, LOANED, LOST, WITHDRAWN

}

```

#### Loan.java

```java

import java.time.LocalDate;

public class Loan {

private LocalDate loanDate = SystemDate.now();

private LocalDate returnDate;

private Borrower borrower;

private Item item;

// Constructors, Getters, and Setters

public Loan(Borrower borrower, Item item) {

this.borrower = borrower;

this.item = item;

}

public LocalDate getLoanDate() {

return loanDate;

}

public void setLoanDate(LocalDate loanDate) {

this.loanDate = loanDate;

}

// Other getters and setters...

public LocalDate getDue() {

return borrower.getLoanDue(loanDate);

}

public boolean isPending() {

return returnDate == null;

}

public void returnItem() {

this.returnDate = SystemDate.now();

}

public boolean isOverdue() {

return LocalDate.now().isAfter(getDue());

}

public long daysToDue() {

return getDue().toEpochDay() - LocalDate.now().toEpochDay();

}

public long getOverdue() {

return LocalDate.now().toEpochDay() - getDue().toEpochDay();

}

public Money getFine() {

return new Money(getOverdue() \* borrower.getDailyFine().getAmount());

}

}

```

#### Person.java

```java

public class Person {

private String firstName;

private String lastName;

// Constructors, Getters, and Setters

public Person(String firstName, String lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

// Other getters and setters...

}

```

#### SystemDate.java (for handling system date, assuming a singleton for simplicity)

```java

import java.time.LocalDate;

public class SystemDate {

private static LocalDate now = LocalDate.now();

public static LocalDate now() {

return now;

}

public static void setDate(LocalDate date) {

now = date;

}

}

```

### DAO Classes

#### GenericDAO.java (Base DAO class)

```java

import java.util.HashMap;

import java.util.Map;

public class GenericDAO<T> {

protected Map<Integer, T> datastore = new HashMap<>();

public void save(int id, T entity) {

datastore.put(id, entity);

}

public T findById(int id) {

return datastore.get(id);

}

public void delete(int id) {

datastore.remove(id);

}

public boolean exists(int id) {

return datastore.containsKey(id);

}

}

```

#### BorrowerDAO.java

```java

public class BorrowerDAO extends GenericDAO<Borrower> {

// Additional methods if necessary

}

```

#### BookDAO.java

```java

public class BookDAO extends GenericDAO<Book> {

// Additional methods if necessary

}

```

#### ItemDAO.java

```java

public class ItemDAO extends GenericDAO<Item> {

// Additional methods if necessary

}

```

#### LoanDAO.java

```java

public class LoanDAO extends GenericDAO<Loan> {

// Additional methods if necessary

}

```

### Service Classes

#### BorrowerService.java

```java

public class BorrowerService {

private BorrowerDAO borrowerDAO = new BorrowerDAO();

public void registerBorrower(Borrower borrower) {

if (!borrowerDAO.exists(borrower.getBorrowerNo())) {

borrowerDAO.save(borrower.getBorrowerNo(), borrower);

} else {

throw new RuntimeException("Borrower already exists");

}

}

public void updateBorrower(Borrower borrower) {

if (borrowerDAO.exists(borrower.getBorrowerNo())) {

borrowerDAO.save(borrower.getBorrowerNo(), borrower);

} else {

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

}

}

public void deleteBorrower(int borrowerNo) {

if (borrowerDAO.exists(borrowerNo)) {

Borrower borrower = borrowerDAO.findById(borrowerNo);

if (borrower.countPendingItems() == 0) {

borrowerDAO.delete(borrowerNo);

} else {

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

}

} else {

**ChatGPT**

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

}

}

public Borrower findBorrower(int borrowerNo) {

return borrowerDAO.findById(borrowerNo);

}

}

```

#### BookService.java

```java

public class BookService {

private BookDAO bookDAO = new BookDAO();

public void addBook(Book book) {

if (!bookDAO.exists(book.getIsbn().hashCode())) {

bookDAO.save(book.getIsbn().hashCode(), book);

} else {

throw new RuntimeException("Book already exists");

}

}

public Book findBook(String isbn) {

return bookDAO.findById(isbn.hashCode());

}

public void deleteBook(String isbn) {

bookDAO.delete(isbn.hashCode());

}

}

```

#### LoanService.java

```java

public class LoanService {

private LoanDAO loanDAO = new LoanDAO();

public void createLoan(Loan loan) {

loanDAO.save(loan.hashCode(), loan);

}

public Loan findLoan(int id) {

return loanDAO.findById(id);

}

public void returnLoan(int loanId) {

Loan loan = loanDAO.findById(loanId);

loan.returnItem();

loanDAO.save(loanId, loan);

}

}

```

### Gherkin Step Definitions

Now that we have the domain classes, DAOs, and services, we can create the step definitions. Let's start with the step definitions for the first feature: Borrower handling by the system.

#### BorrowerStepDefinitions.java

```java

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

import static org.junit.jupiter.api.Assertions.\*;

public class BorrowerStepDefinitions {

private BorrowerService borrowerService = new BorrowerService();

private Borrower borrower;

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

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

borrower = new Borrower(1, new TelephoneNumber("123456789"), new EmailAddress("george@example.com"), new Address("Street 1"), new BorrowerCategory("Standard", 5, 14, new Money(1)), new Person("George", "Red"));

try {

borrowerService.findBorrower(borrower.getBorrowerNo());

fail("Borrower should not be registered");

} catch (RuntimeException e) {

// Expected

}

}

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

public void borrower\_gets\_registered\_in\_the\_system(String name) {

borrowerService.registerBorrower(borrower);

}

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

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

Borrower foundBorrower = borrowerService.findBorrower(borrower.getBorrowerNo());

assertNotNull(foundBorrower);

}

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

public void borrower\_is\_registered\_as\_a\_borrower(String name) {

borrowerService.registerBorrower(borrower);

Borrower foundBorrower = borrowerService.findBorrower(borrower.getBorrowerNo());

assertNotNull(foundBorrower);

}

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

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

try {

borrowerService.registerBorrower(borrower);

fail("Borrower should already be registered");

} catch (RuntimeException e) {

// Expected

}

}

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

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

// Implementation here

}

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

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

borrower.setTelephone(new TelephoneNumber("987654321"));

borrowerService.updateBorrower(borrower);

}

@Then("the system saves the changes")

public void system\_saves\_the\_changes() {

Borrower updatedBorrower = borrowerService.findBorrower(borrower.getBorrowerNo());

assertEquals("987654321", updatedBorrower.getTelephone().getNumber());

}

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

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

borrower = new Borrower(2, new TelephoneNumber("123456789"), new EmailAddress("george@example.com"), new Address("Street 1"), new BorrowerCategory("Standard", 5, 14, new Money(1)), new Person("George", "Red"));

try {

borrowerService.updateBorrower(borrower);

fail("Borrower should not exist");

} catch (RuntimeException e) {

// Expected

}

}

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

public void system\_displays\_error\_message\_borrower\_does\_not\_exist(String name) {

// Implementation here

}

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

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

borrowerService.deleteBorrower(borrower.getBorrowerNo());

}

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

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

try {

borrowerService.findBorrower(borrower.getBorrowerNo());

fail("Borrower should not exist");

} catch (RuntimeException e) {

// Expected

}

}

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

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

try {

borrowerService.deleteBorrower(borrower.getBorrowerNo());

fail("Borrower should not exist");

} catch (RuntimeException e) {

// Expected

}

}

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

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

// Implementation here

}

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

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

// Add loans to borrower

Loan loan = new Loan(borrower, new Item(1, new Book("123", "Harry Potter", "Publisher", 2000, new Publisher())));

borrower.getLoans().add(loan);

}

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

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

try {

borrowerService.findBorrower(borrower.getBorrowerNo());

} catch (RuntimeException e) {

fail("Borrower should still exist");

}

}

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

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

// Implementation here

}

}

```

The above code provides the domain classes, DAOs, services, and step definitions for the first feature. We can similarly create the step definitions for the other features and scenarios. Let me know if you want to proceed with the next feature or need any modifications to the existing setup.

**User**

**Give me all the step definitions of all the features provided**

**ChatGPT**

Sure, I'll provide step definitions for all the features and scenarios.

### BorrowerStepDefinitions.java

```java

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

import static org.junit.jupiter.api.Assertions.\*;

public class BorrowerStepDefinitions {

private BorrowerService borrowerService = new BorrowerService();

private Borrower borrower;

private String errorMessage;

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

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

borrower = new Borrower(1, new TelephoneNumber("123456789"), new EmailAddress("george@example.com"), new Address("Street 1"), new BorrowerCategory("Standard", 5, 14, new Money(1)), new Person("George", "Red"));

assertFalse(borrowerService.exists(borrower.getBorrowerNo()));

}

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

public void borrower\_gets\_registered\_in\_the\_system(String name) {

borrowerService.registerBorrower(borrower);

}

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

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

assertNotNull(borrowerService.findBorrower(borrower.getBorrowerNo()));

}

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

public void borrower\_is\_registered\_as\_a\_borrower(String name) {

borrowerService.registerBorrower(borrower);

assertNotNull(borrowerService.findBorrower(borrower.getBorrowerNo()));

}

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

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

try {

borrowerService.registerBorrower(borrower);

} catch (RuntimeException e) {

errorMessage = e.getMessage();

}

}

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

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

assertEquals("Borrower already exists", errorMessage);

}

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

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

borrower.setTelephone(new TelephoneNumber("987654321"));

borrowerService.updateBorrower(borrower);

}

@Then("the system saves the changes")

public void system\_saves\_the\_changes() {

assertEquals("987654321", borrowerService.findBorrower(borrower.getBorrowerNo()).getTelephone().getNumber());

}

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

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

borrower = new Borrower(2, new TelephoneNumber("123456789"), new EmailAddress("george@example.com"), new Address("Street 1"), new BorrowerCategory("Standard", 5, 14, new Money(1)), new Person("George", "Red"));

try {

borrowerService.updateBorrower(borrower);

} catch (RuntimeException e) {

errorMessage = e.getMessage();

}

}

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

public void system\_displays\_error\_message\_borrower\_does\_not\_exist(String name) {

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

}

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

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

borrowerService.deleteBorrower(borrower.getBorrowerNo());

}

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

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

assertFalse(borrowerService.exists(borrower.getBorrowerNo()));

}

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

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

try {

borrowerService.deleteBorrower(borrower.getBorrowerNo());

} catch (RuntimeException 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 loan = new Loan(borrower, new Item(1, new Book("123", "Harry Potter", "Publisher", 2000, new Publisher())));

borrower.getLoans().add(loan);

}

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

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

assertTrue(borrowerService.exists(borrower.getBorrowerNo()));

}

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

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

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

}

}

```

### DelayedReturnNotificationStepDefinitions.java

```java

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

import static org.junit.jupiter.api.Assertions.\*;

public class DelayedReturnNotificationStepDefinitions {

private BorrowerService borrowerService = new BorrowerService();

private Borrower borrower;

private LoanService loanService = new LoanService();

private Loan loan;

private String emailNotification;

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

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

borrower = new Borrower(1, new TelephoneNumber("123456789"), new EmailAddress("george@example.com"), new Address("Street 1"), new BorrowerCategory("Standard", 5, 14, new Money(1)), new Person("George", "Red"));

borrowerService.registerBorrower(borrower);

Item item = new Item(1, new Book("123", itemName, "Publisher", 2000, new Publisher()));

loan = new Loan(borrower, item);

loanService.createLoan(loan);

}

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

public void item\_due\_date\_has\_passed(String itemName) {

SystemDate.setDate(SystemDate.now().plusDays(15)); // Simulating 15 days passed

}

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

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

assertNotNull(borrower.getEmailAddress());

}

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

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

if (loan.isOverdue() && borrower.getEmailAddress() != null) {

emailNotification = "Notification sent to " + borrower.getEmailAddress().getEmail();

}

}

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

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

assertEquals("Notification sent to george@example.com", emailNotification);

}

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

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

borrower.setEmailAddress(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(emailNotification);

}

}

```

### LoaningItemsStepDefinitions.java

```java

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

import static org.junit.jupiter.api.Assertions.\*;

public class LoaningItemsStepDefinitions {

private BorrowerService borrowerService = new BorrowerService();

private BookService bookService = new BookService();

private LoanService loanService = new LoanService();

private Borrower borrower;

private Item item;

private String errorMessage;

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

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

Book book = new Book("123", itemName, "Publisher", 2000, new Publisher());

bookService.addBook(book);

item = new Item(1, book);

item.available();

}

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

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

borrower = new Borrower(1, new TelephoneNumber("123456789"), new EmailAddress("george@example.com"), new Address("Street 1"), new BorrowerCategory("Standard", 5, 14, new Money(1)), new Person("George", "Red"));

borrowerService.registerBorrower(borrower);

}

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

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

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

Loan loan = new Loan(borrower, new Item(i, new Book("123", "Book" + i, "Publisher", 2000, new Publisher())));

borrower.getLoans().add(loan);

}

}

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

public void borrower\_has\_maximum\_lending\_limit(String borrowerName, int limit) {

borrower.getCategory().setMaxLendingItems(limit);

}

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

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

if (borrower.canBorrow()) {

loanService.createLoan(item.borrow(borrower));

} else {

errorMessage = "Borrowing limit reached";

}

}

@Then("the

**ChatGPT**

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

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

Loan loan = loanService.findLoan(item.hashCode());

assertNotNull(loan);

assertEquals(borrower.getBorrowerNo(), loan.getBorrower().getBorrowerNo());

}

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

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

assertEquals(pendingItems, borrower.countPendingItems());

}

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

public void borrower\_tries\_to\_borrow\_both\_items(String borrowerName) {

// Trying to borrow the first item

if (borrower.canBorrow()) {

loanService.createLoan(item.borrow(borrower));

} else {

errorMessage = "Borrowing limit reached for first item";

}

// Trying to borrow the second item

Item secondItem = new Item(2, new Book("124", "Moby Dick", "Publisher", 2000, new Publisher()));

secondItem.available();

if (borrower.canBorrow()) {

loanService.createLoan(secondItem.borrow(borrower));

} else {

errorMessage = "Borrowing limit reached for second item";

}

}

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

public void system\_does\_not\_loan\_item\_due\_to\_lending\_limit(String itemName, String borrowerName) {

assertEquals("Borrowing limit reached for second item", 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) {

item = new Item(3, new Book("125", itemName, "Publisher", 2000, new Publisher()));

}

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

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

try {

loanService.createLoan(item.borrow(borrower));

errorMessage = null;

} catch (RuntimeException e) {

errorMessage = e.getMessage();

item.withdraw();

}

}

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

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

assertEquals("Item not found in system", errorMessage);

}

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

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

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

}

@When("{string} tries to borrow the item {string} but has reached his max lending limit")

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

borrower.getCategory().setMaxLendingItems(3);

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

Loan loan = new Loan(borrower, new Item(i, new Book("126", "Book" + i, "Publisher", 2000, new Publisher())));

borrower.getLoans().add(loan);

}

try {

loanService.createLoan(item.borrow(borrower));

} catch (RuntimeException e) {

errorMessage = e.getMessage();

}

}

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

public void system\_does\_not\_allow\_loan() {

assertEquals("Borrowing limit reached", errorMessage);

}

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

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

assertEquals(pendingItems, borrower.countPendingItems());

}

}

```

### ReturningItemsStepDefinitions.java

```java

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

import static org.junit.jupiter.api.Assertions.\*;

public class ReturningItemsStepDefinitions {

private BorrowerService borrowerService = new BorrowerService();

private BookService bookService = new BookService();

private LoanService loanService = new LoanService();

private Borrower borrower;

private Item item;

private Loan loan;

@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 days) {

borrower = new Borrower(1, new TelephoneNumber("123456789"), new EmailAddress("george@example.com"), new Address("Street 1"), new BorrowerCategory("Standard", 5, 14, new Money(1)), new Person("George", "Red"));

borrowerService.registerBorrower(borrower);

item = new Item(1, new Book("123", itemName, "Publisher", 2000, new Publisher()));

item.available();

loan = new Loan(borrower, item);

loan.setLoanDate(SystemDate.now().minusDays(days));

loanService.createLoan(loan);

borrower.getLoans().add(loan);

}

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

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

borrower.getCategory().setMaxLendingDays(maxDays);

}

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

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

loan.returnItem();

loanService.returnLoan(loan.hashCode());

}

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

public void system\_marks\_item\_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) {

assertEquals(0, borrower.countPendingItems());

}

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

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

assertFalse(loan.isOverdue());

}

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

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

assertEquals(SystemDate.now(), loan.getReturnDate());

}

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

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

assertTrue(loan.isOverdue());

assertEquals(new Money(loan.getOverdue() \* borrower.getDailyFine().getAmount()), loan.getFine());

}

}

```

These step definitions cover all the provided Gherkin scenarios. They use the services and domain classes we defined earlier to perform the necessary actions and assertions. Each step definition method corresponds to a specific step in the Gherkin scenarios, ensuring that the tests accurately simulate the expected behavior of the library system.