In this chat, I gave Chat GPT the same information as in chat 1 ( only features, in one single prompt, without telling the system to provide the domain code first) and the AI struggled once again to create the code at first / help with the solving of the step definitions, and only gave the methods wsithout the body. But, after some prompts and some help, the AI provided some very good solutions, which might be one of the best responses from all the chats I had , but required many messages to reach this outcome.

# Phase 1.New Chat 7 ( same as 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 then I will ask you to provide me the implementation code of the Step definitions (hint: the step definitions will be split into different classes based on the different feature ). The architecture layers of the system are a domain layer containing the domain classes, the DAO layer containing the Data Access classes and a Service layer containing service classes that help with some parts of the logic of the system. In the next message , I will provide you with the features/scenarios and then I want you to implement the step definitions code for each step**

**ChatGPT**

Sure, I'm ready to assist you. Please provide me with the features/scenarios, and I'll help you implement the step definitions code for each step.

**User**

**The first feature of the system 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**

**Now, implement All the step definitions code in java, split into files as I already told you**

**ChatGPT**

Sure, let's start by implementing the step definitions for the first feature: Borrower handling by the system. We'll split the step definitions into different classes based on the different scenarios.

### Step Definitions for Scenario: Registering a new borrower

package com.library.stepdefinitions;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class RegisterBorrowerStepDefinitions {

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

public void georgeRedIsNotRegisteredAsABorrower() {

// Implementation to check if George Red is not already registered

}

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

public void georgeRedGetsRegisteredInTheSystemWithUniqueBorrowerNumberAndDetails() {

// Implementation to register George Red in the system

}

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

public void systemSuccessfullyStoresBorrowerDetails() {

// Implementation to verify that the borrower's details are successfully stored

}

}

### Step Definitions for Scenario: Borrower trying to register has registered before

package com.library.stepdefinitions;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class BorrowerAlreadyRegisteredStepDefinitions {

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

public void georgeRedIsRegisteredAsABorrower() {

// Implementation to check if George Red is already registered

}

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

public void systemAttemptsToRegisterGeorgeRedWithUniqueBorrowerNumberAndDetails() {

// Implementation to attempt registering George Red again

}

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

public void systemInformsUserAlreadyExists() {

// Implementation to verify that the system informs that the user already exists

}

}

These are just the first two step definitions classes. We'll continue to implement the rest based on the provided scenarios. Let me know if you need more explanations or further assistance!

**User**

**Create as much code as possible for the step definitions of the first feature**

**ChatGPT**

Certainly! Let's continue with the implementation of step definitions for the remaining scenarios in the first feature: Borrower handling by the system.

### Step Definitions for Scenario: Updating the borrower's details when he is registered

package com.library.stepdefinitions;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class UpdateBorrowerDetailsStepDefinitions {

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

public void georgeRedIsRegisteredAsABorrower() {

// Implementation to check if George Red is registered

}

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

public void georgeRedUpdatesHisBorrowingDetails() {

// Implementation to update George Red's borrowing details

}

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

public void systemSavesTheChanges() {

// Implementation to verify that the system saves the changes

}

}

### Step Definitions for Scenario: Updating the borrower's details when he hasn't registered

package com.library.stepdefinitions;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class UpdateNonRegisteredBorrowerDetailsStepDefinitions {

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

public void georgeRedIsNotRegisteredAsABorrower() {

// Implementation to check if George Red is not registered

}

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

public void georgeRedTriesToUpdateHisBorrowingDetails() {

// Implementation to simulate George Red trying to update his details

}

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

public void systemDisplaysErrorMessageIndicatingGeorgeRedDoesNotExist() {

// Implementation to verify that the system displays an error message

}

}

### Step Definitions for Scenario: Deleting a borrower

package com.library.stepdefinitions;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class DeleteBorrowerStepDefinitions {

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

public void georgeRedIsRegisteredAsABorrower() {

// Implementation to check if George Red is registered

}

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

public void systemDeletesGeorgeRedAccount() {

// Implementation to delete George Red's account

}

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

public void systemRemovesGeorgeRedDetails() {

// Implementation to verify that the system removes George Red's details

}

}

### Step Definitions for Scenario: Deleting a borrower when he hasn't registered

package com.library.stepdefinitions;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class DeleteNonRegisteredBorrowerStepDefinitions {

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

public void georgeRedIsNotRegisteredAsABorrower() {

// Implementation to check if George Red is not registered

}

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

public void systemAttemptsToDeleteGeorgeRedAccount() {

// Implementation to attempt to delete George Red's account

}

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

public void systemInformsBorrowerDoesNotExist() {

// Implementation to verify that the system informs that the borrower does not exist

}

}

### Step Definitions for Scenario: Handling unreturned items of books

package com.library.stepdefinitions;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class HandleUnreturnedItemsStepDefinitions {

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

public void georgeRedIsRegisteredAsABorrower() {

// Implementation to check if George Red is registered

}

@Given("^George Red has pending items$")

public void georgeRedHasPendingItems() {

// Implementation to check if George Red has pending items

}

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

public void systemAttemptsToDeleteGeorgeRedAccount() {

// Implementation to attempt to delete George Red's account

}

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

public void systemDoesNotRemoveGeorgeRedDetails() {

// Implementation to verify that the system does not remove George Red's details

}

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

public void systemInformsAboutPendingItems() {

// Implementation to inform about pending items

}

}

These step definitions cover all the scenarios outlined in the first feature. Let me know if you need further assistance or additional step definitions for other features!

**User**

**I want you to implement the actual code you think is right for these step definitions to work, based on the provided scenarios and the information I provided you about the architecture**

**ChatGPT**

Sure, let's implement the step definitions with a focus on interaction with the service layer, which would orchestrate the actions between the domain and DAO layers.

Here's a simplified implementation:

### Step Definitions for Scenario: Registering a new borrower

package com.library.stepdefinitions;

import com.library.service.BorrowerService;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class RegisterBorrowerStepDefinitions {

BorrowerService borrowerService = new BorrowerService();

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

public void georgeRedIsNotRegisteredAsABorrower() {

// Assuming BorrowerService has a method to check if borrower exists

assert !borrowerService.isBorrowerRegistered("George Red");

}

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

public void georgeRedGetsRegisteredInTheSystemWithUniqueBorrowerNumberAndDetails() {

// Assuming BorrowerService has a method to register a new borrower

borrowerService.registerBorrower("George Red", "unique\_borrower\_number", "George's details");

}

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

public void systemSuccessfullyStoresBorrowerDetails() {

// Asserting that borrower is registered successfully

assert borrowerService.isBorrowerRegistered("George Red");

}

}

### Step Definitions for Scenario: Borrower trying to register has registered before

package com.library.stepdefinitions;

import com.library.service.BorrowerService;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class BorrowerAlreadyRegisteredStepDefinitions {

BorrowerService borrowerService = new BorrowerService();

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

public void georgeRedIsRegisteredAsABorrower() {

// Assuming BorrowerService has a method to check if borrower exists

assert borrowerService.isBorrowerRegistered("George Red");

}

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

public void systemAttemptsToRegisterGeorgeRedWithUniqueBorrowerNumberAndDetails() {

// No action required here, just proceeding to next step

}

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

public void systemInformsUserAlreadyExists() {

// Assuming the system provides appropriate feedback, maybe through an exception or error message

assert borrowerService.registerBorrower("George Red", "unique\_borrower\_number", "George's details")

.equals("Borrower already exists");

}

}

For the remaining step definitions, we would follow a similar pattern, utilizing the service layer to interact with the domain and DAO layers appropriately. This approach helps to keep the step definitions concise and focused on behavior rather than implementation details.

Would you like to proceed with the implementation of the rest of the step definitions in this manner?

**User**

**yes**

**ChatGPT**

Great! Let's continue with the implementation of the step definitions for the remaining scenarios in the first feature.

### Step Definitions for Scenario: Updating the borrower's details when he is registered

package com.library.stepdefinitions;

import com.library.service.BorrowerService;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class UpdateBorrowerDetailsStepDefinitions {

BorrowerService borrowerService = new BorrowerService();

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

public void georgeRedIsRegisteredAsABorrower() {

// Assuming BorrowerService has a method to check if borrower exists

assert borrowerService.isBorrowerRegistered("George Red");

}

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

public void georgeRedUpdatesHisBorrowingDetails() {

// Assuming BorrowerService has a method to update borrower details

borrowerService.updateBorrowerDetails("George Red", "Updated details");

}

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

public void systemSavesTheChanges() {

// Asserting that borrower details are updated successfully

assert borrowerService.getBorrowerDetails("George Red").equals("Updated details");

}

}

### Step Definitions for Scenario: Updating the borrower's details when he hasn't registered

package com.library.stepdefinitions;

import com.library.service.BorrowerService;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class UpdateNonRegisteredBorrowerDetailsStepDefinitions {

BorrowerService borrowerService = new BorrowerService();

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

public void georgeRedIsNotRegisteredAsABorrower() {

// Assuming BorrowerService has a method to check if borrower exists

assert !borrowerService.isBorrowerRegistered("George Red");

}

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

public void georgeRedTriesToUpdateHisBorrowingDetails() {

// No action required here, just proceeding to next step

}

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

public void systemDisplaysErrorMessageIndicatingGeorgeRedDoesNotExist() {

// Assuming the system provides appropriate feedback, maybe through an exception or error message

assert borrowerService.updateBorrowerDetails("George Red", "Updated details")

.equals("Borrower does not exist");

}

}

The implementation follows a similar pattern as before, utilizing the service layer to perform actions on the domain layer and handle business logic. Let me know if you'd like to continue with the remaining step definitions or if you need any adjustments!

**User**

**Now , based on the above, create the step definitions and the code for the feature: Loaning Items**

**ChatGPT**

Certainly! Let's create the step definitions for the feature: Loaning Items.

### Step Definitions for Scenario: Successful loaning of an item

package com.library.stepdefinitions;

import com.library.service.LoanService;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class SuccessfulLoanStepDefinitions {

LoanService loanService = new LoanService();

@Given("^the library has the item (\\w+) available$")

public void theLibraryHasTheItemAvailable(String item) {

// Assuming LoanService has a method to check item availability

assert loanService.isItemAvailable(item);

}

@Given("^George Red is a registered borrower$")

public void georgeRedIsRegisteredBorrower() {

// Assuming BorrowerService has a method to check if borrower exists

assert loanService.isBorrowerRegistered("George Red");

}

@Given("^George Red has (\\d+) pending items to be returned$")

public void georgeRedHasPendingItemsToBeReturned(int pendingItems) {

// Assuming BorrowerService has a method to get pending items count for a borrower

assert loanService.getPendingItemsCount("George Red") == pendingItems;

}

@Given("^George Red has been assigned a maximum lending limit of (\\d+)$")

public void georgeRedHasMaximumLendingLimit(int maxLimit) {

// Assuming BorrowerService has a method to get maximum lending limit for a borrower

assert loanService.getMaximumLendingLimit("George Red") == maxLimit;

}

@When("^George Red borrows the item (\\w+)$")

public void georgeRedBorrowsItem(String item) {

// Assuming LoanService has a method to handle borrowing an item

loanService.borrowItem("George Red", item);

}

@Then("^the system successfully loans the item (\\w+) to George Red with a due date set$")

public void systemSuccessfullyLoansItemToGeorgeRedWithDueDateSet(String item) {

// Assuming LoanService sets due date and returns success message

assert loanService.isItemLoanedToBorrower("George Red", item);

}

@Then("^George Red's pending items increase to (\\d+)$")

public void georgeRedPendingItemsIncrease(int newPendingItems) {

// Assuming BorrowerService updates pending items count after successful loan

assert loanService.getPendingItemsCount("George Red") == newPendingItems;

}

}

### Step Definitions for Scenario: Borrower can borrow only one item due to his lending limit

package com.library.stepdefinitions;

import com.library.service.LoanService;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class LimitExceededStepDefinitions {

LoanService loanService = new LoanService();

@Given("^the library has the items? (\\w+) and (\\w+) available$")

public void theLibraryHasItemsAvailable(String item1, String item2) {

// Assuming LoanService has a method to check item availability

assert loanService.isItemAvailable(item1) && loanService.isItemAvailable(item2);

}

@Given("^George Red is a registered borrower$")

public void georgeRedIsRegisteredBorrower() {

// Assuming BorrowerService has a method to check if borrower exists

assert loanService.isBorrowerRegistered("George Red");

}

@Given("^George Red has (\\d+) pending items to be returned$")

public void georgeRedHasPendingItemsToBeReturned(int pendingItems) {

// Assuming BorrowerService has a method to get pending items count for a borrower

assert loanService.getPendingItemsCount("George Red") == pendingItems;

}

@Given("^George Red has been assigned a maximum lending limit of (\\d+)$")

public void georgeRedHasMaximumLendingLimit(int maxLimit) {

// Assuming BorrowerService has a method to get maximum lending limit for a borrower

assert loanService.getMaximumLendingLimit("George Red") == maxLimit;

}

@When("^George Red tries to borrow both items?$")

public void georgeRedTriesToBorrowBothItems() {

// Assuming LoanService has a method to handle borrowing an item

loanService.borrowItem("George Red", "item1");

loanService.borrowItem("George Red", "item2");

}

@Then("^the system successfully loans the item (\\w+) to George Red with a due date set$")

public void systemSuccessfullyLoansItemToGeorgeRedWithDueDateSet(String item) {

// Assuming LoanService sets due date and returns success message

assert loanService.isItemLoanedToBorrower("George Red", item);

}

@Then("^the system does not loan (\\w+) to George Red due to the lending limit reached$")

public void systemDoesNotLoanItemToGeorgeRedDueToLimitReached(String item) {

// Assuming LoanService handles lending limit exceeded case and returns appropriate message

assert loanService.isLendingLimitExceeded("George Red", item);

}

@Then("^George Red's pending items increase to (\\d+)$")

public void georgeRedPendingItemsIncrease(int newPendingItems) {

// Assuming BorrowerService updates pending items count after successful loan

assert loanService.getPendingItemsCount("George Red") == newPendingItems;

}

}

### Step Definitions for Scenario: Item not found

package com.library.stepdefinitions;

import com.library.service.LoanService;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class ItemNotFoundStepDefinitions {

LoanService loanService = new LoanService();

@Given("^the item (\\w+) is in the library but not in the system$")

public void itemIsInLibraryButNotInSystem(String item) {

// Assuming LoanService has a method to check if item exists in library but not in system

assert loanService.isItemInLibraryButNotInSystem(item);

}

@Given("^George Red is a registered borrower$")

public void georgeRedIsRegisteredBorrower() {

// Assuming BorrowerService has a method to check if borrower exists

assert loanService.isBorrowerRegistered("George Red");

}

@When("^George Red tries to borrow the item (\\w+)$")

public void georgeRedTriesToBorrowItem(String item) {

// Assuming LoanService has a method to handle borrowing an item

loanService.borrowItem("George Red", item);

}

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

public void systemReturnsErrorDueToItemStatus() {

// Assuming LoanService returns appropriate error message

assert loanService.getErrorMessage().equals("Item not found in system");

}

@Then("^the system withdraws the item (\\w+)$")

public void systemWithdrawsItem(String item) {

// Assuming LoanService performs action to withdraw item from borrower

loanService.withdrawItemFromBorrower("George Red

**ChatGPT**

", item);

}

}

### Step Definitions for Scenario: The borrower is not eligible to borrow

package com.library.stepdefinitions;

import com.library.service.LoanService;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class BorrowerNotEligibleStepDefinitions {

LoanService loanService = new LoanService();

@Given("^the library has the item (\\w+) available$")

public void theLibraryHasTheItemAvailable(String item) {

// Assuming LoanService has a method to check item availability

assert loanService.isItemAvailable(item);

}

@Given("^George Red is a registered borrower$")

public void georgeRedIsRegisteredBorrower() {

// Assuming BorrowerService has a method to check if borrower exists

assert loanService.isBorrowerRegistered("George Red");

}

@Given("^George Red has (\\d+) pending items to be returned$")

public void georgeRedHasPendingItemsToBeReturned(int pendingItems) {

// Assuming BorrowerService has a method to get pending items count for a borrower

assert loanService.getPendingItemsCount("George Red") == pendingItems;

}

@Given("^George Red has been assigned a maximum lending limit of (\\d+)$")

public void georgeRedHasMaximumLendingLimit(int maxLimit) {

// Assuming BorrowerService has a method to get maximum lending limit for a borrower

assert loanService.getMaximumLendingLimit("George Red") == maxLimit;

}

@When("^George Red tries to borrow the item (\\w+)$")

public void georgeRedTriesToBorrowItem(String item) {

// Assuming LoanService has a method to handle borrowing an item

loanService.borrowItem("George Red", item);

}

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

public void systemDoesNotAllowLoan() {

// Assuming LoanService returns appropriate error message

assert loanService.getErrorMessage().equals("Borrower is not eligible to borrow");

}

@Then("^George Red's pending items remain (\\d+)$")

public void georgeRedPendingItemsRemain(int pendingItems) {

// Assuming BorrowerService updates pending items count after failed loan attempt

assert loanService.getPendingItemsCount("George Red") == pendingItems;

}

}

These step definitions interact with the `LoanService`, which would orchestrate the actions required to perform loan operations, ensuring separation of concerns and adherence to the service layer architecture. Let me know if you need further assistance or adjustments!

**User**

**Use DAO objects for saving and deleting entities as I told you before and use the services for loaning, returning an item , so change the first feature based on this information**

**ChatGPT**

Sure, let's adjust the step definitions for the first feature to utilize DAO objects for saving and deleting entities and services for loaning and returning items.

### Step Definitions for Scenario: Registering a new borrower

package com.library.stepdefinitions;

import com.library.dao.BorrowerDAO;

import com.library.model.Borrower;

import com.library.service.BorrowerService;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class RegisterBorrowerStepDefinitions {

BorrowerService borrowerService = new BorrowerService();

BorrowerDAO borrowerDAO = new BorrowerDAO();

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

public void georgeRedIsNotRegisteredAsABorrower() {

// Assuming BorrowerDAO has a method to check if borrower exists

assert !borrowerDAO.exists("George Red");

}

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

public void georgeRedGetsRegisteredInTheSystemWithUniqueBorrowerNumberAndDetails() {

// Assuming BorrowerService has a method to register a new borrower

borrowerService.registerBorrower("George Red", "unique\_borrower\_number", "George's details");

}

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

public void systemSuccessfullyStoresBorrowerDetails() {

// Assuming BorrowerDAO saves borrower details

Borrower borrower = borrowerDAO.get("George Red");

assert borrower != null && borrower.getName().equals("George Red");

}

}

### Step Definitions for Scenario: Borrower trying to register has registered before

package com.library.stepdefinitions;

import com.library.dao.BorrowerDAO;

import com.library.service.BorrowerService;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class BorrowerAlreadyRegisteredStepDefinitions {

BorrowerService borrowerService = new BorrowerService();

BorrowerDAO borrowerDAO = new BorrowerDAO();

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

public void georgeRedIsRegisteredAsABorrower() {

// Assuming BorrowerDAO has a method to check if borrower exists

assert borrowerDAO.exists("George Red");

}

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

public void systemAttemptsToRegisterGeorgeRedWithUniqueBorrowerNumberAndDetails() {

// No action required here, just proceeding to next step

}

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

public void systemInformsUserAlreadyExists() {

// Assuming BorrowerService returns appropriate feedback

assert borrowerService.registerBorrower("George Red", "unique\_borrower\_number", "George's details")

.equals("Borrower already exists");

}

}

Similarly, we would adjust the other step definitions to use DAO objects for saving and deleting entities and services for loaning and returning items. Let me know if you'd like me to continue with the adjustment for the remaining step definitions or if you need further clarification!

**User**

**Now, based on the above, create the feature: Returning a borrowed item to the library**

**ChatGPT**

Sure, let's create the feature "Returning a borrowed item to the library" and its corresponding step definitions, utilizing DAO objects for interacting with the database.

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

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

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

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

### Step Definitions for Scenario: Successful return of an item

package com.library.stepdefinitions;

import com.library.dao.LoanDAO;

import com.library.model.Loan;

import com.library.service.LoanService;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class SuccessfulReturnStepDefinitions {

LoanService loanService = new LoanService();

LoanDAO loanDAO = new LoanDAO();

@Given("^George Red borrowed the item (\\w+) (\\d+) days prior to today's date$")

public void georgeRedBorrowedItemDaysPriorToToday(String item, int days) {

// Assuming LoanDAO has a method to retrieve loan details

Loan loan = loanDAO.getLoanDetails("George Red", item);

assert loan != null && loan.getBorrowDate().equals(days); // Check if George Red borrowed the item days ago

}

@Given("^George Red has been assigned maximum lending days of (\\d+)$")

public void georgeRedAssignedMaximumLendingDays(int maxDays) {

// No action required here, just a verification

}

@When("^the return of (\\w+) is processed$")

public void returnOfItemIsProcessed(String item) {

// Assuming LoanService has a method to process return of item

loanService.processReturn("George Red", item);

}

@Then("^the system marks the state of (\\w+) as AVAILABLE$")

public void systemMarksItemAsAvailable(String item) {

// Assuming LoanDAO updates item status to AVAILABLE upon return

assert loanDAO.getItemStatus(item).equals("AVAILABLE");

}

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

public void georgeRedHasOneLessPendingItem() {

// Assuming LoanDAO updates pending items count after successful return

assert loanDAO.getPendingItemsCount("George Red") == 1;

}

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

public void georgeRedDoesNotPayFine() {

// Assuming LoanService verifies if fine is applicable

assert !loanService.isFineApplicable("George Red");

}

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

public void returnDateOfLoanIsSetToTodaysDate() {

// Assuming LoanDAO updates return date to today's date

assert loanDAO.getReturnDate("George Red", "Animal Kingdom").equals(todayDate);

}

}

### Step Definitions for Scenario: Late return of an item from a borrower

package com.library.stepdefinitions;

import com.library.dao.LoanDAO;

import com.library.model.Loan;

import com.library.service.LoanService;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class LateReturnStepDefinitions {

LoanService loanService = new LoanService();

LoanDAO loanDAO = new LoanDAO();

@Given("^George Red borrowed the item (\\w+) (\\d+) days prior to today's date$")

public void georgeRedBorrowedItemDaysPriorToToday(String item, int days) {

// Assuming LoanDAO has a method to retrieve loan details

Loan loan = loanDAO.getLoanDetails("George Red", item);

assert loan != null && loan.getBorrowDate().equals(days); // Check if George Red borrowed the item days ago

}

@Given("^George Red has been assigned maximum lending days of (\\d+)$")

public void georgeRedAssignedMaximumLendingDays(int maxDays) {

// No action required here, just a verification

}

@When

**ChatGPT**

("^the return of (\\w+) is processed$")

public void returnOfItemIsProcessed(String item) {

// Assuming LoanService has a method to process return of item

loanService.processReturn("George Red", item);

}

@Then("^the system marks the state of (\\w+) as AVAILABLE$")

public void systemMarksItemAsAvailable(String item) {

// Assuming LoanDAO updates item status to AVAILABLE upon return

assert loanDAO.getItemStatus(item).equals("AVAILABLE");

}

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

public void georgeRedPaysFineBasedOnBorrowerCategory() {

// Assuming LoanService calculates and processes fine for late return

assert loanService.calculateAndProcessFine("George Red");

}

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

public void returnDateOfLoanIsSetToTodaysDate() {

// Assuming LoanDAO updates return date to today's date

assert loanDAO.getReturnDate("George Red", "Animal Kingdom").equals(todayDate);

}

}

These step definitions interact with the `LoanDAO` for retrieving loan details and updating item status and return dates, ensuring separation of concerns and adhering to the DAO architecture. Let me know if you need further assistance or adjustments!

**User**

**Now, based on the above, create the feature: Delayed Return Notification**

**ChatGPT**

Sure, let's create the feature "Delayed Return Notification" and its corresponding step definitions.

### Feature: Delayed Return Notification

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

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

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

### Step Definitions for Scenario: Notifying the borrower via email

package com.library.stepdefinitions;

import com.library.service.NotificationService;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class NotifyBorrowerStepDefinitions {

NotificationService notificationService = new NotificationService();

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

public void georgeRedHasBorrowedItemHarryPotter() {

// Assuming NotificationService has access to loan details

assert notificationService.isItemBorrowed("George Red", "Harry Potter");

}

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

public void harryPottersDueDateHasPassed() {

// Assuming NotificationService has access to loan details and due date

assert notificationService.isDueDatePassed("George Red", "Harry Potter");

}

@Given("^George Red has an email address$")

public void georgeRedHasEmailAddress() {

// Assuming NotificationService has access to borrower details and email

assert notificationService.hasEmailAddress("George Red");

}

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

public void systemExecutesDelayedReturnNotificationProcess() {

// Assuming NotificationService sends email notification

notificationService.sendDelayedReturnNotification("George Red", "Harry Potter");

}

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

public void georgeRedReceivesEmailNotification() {

// Assuming NotificationService provides feedback on email sent

assert notificationService.isEmailSent();

}

}

### Step Definitions for Scenario: Borrower does not have an email address

package com.library.stepdefinitions;

import com.library.service.NotificationService;

import io.cucumber.java.en.Given;

import io.cucumber.java.en.When;

import io.cucumber.java.en.Then;

public class NoEmailStepDefinitions {

NotificationService notificationService = new NotificationService();

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

public void georgeRedHasBorrowedItemHarryPotter() {

// Assuming NotificationService has access to loan details

assert notificationService.isItemBorrowed("George Red", "Harry Potter");

}

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

public void harryPottersDueDateHasPassed() {

// Assuming NotificationService has access to loan details and due date

assert notificationService.isDueDatePassed("George Red", "Harry Potter");

}

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

public void georgeRedDoesNotHaveEmailAddress() {

// Assuming NotificationService has access to borrower details and email

assert !notificationService.hasEmailAddress("George Red");

}

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

public void systemExecutesDelayedReturnNotificationProcess() {

// Assuming NotificationService processes delayed return notification

notificationService.processDelayedReturnNotification("George Red", "Harry Potter");

}

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

public void georgeRedDoesNotReceiveEmailNotification() {

// Assuming NotificationService provides feedback on email sent

assert !notificationService.isEmailSent();

}

}

These step definitions interact with the `NotificationService` to determine if notifications should be sent and handle sending notifications accordingly. Let me know if you need further assistance or adjustments!