*In this chat, I provided the same information as in Phase 2. Chat 1(features and domain classes) BUT I didn’t instruct the AI to first try and create/guess the code of the domain classes, but instead told it to just generate the step definitions. In the start, as in most of the chats I have had with it, it didn’t really generate code, and just the step definition classes, but with a small help/prompt it gave me answers about the step definitions, written in java code AND in the form of comments explaining to me what I should do. Overall, the code it provided was right in many instances, based on the functions and the variables the AI guessed the domain classes should have, but was not close to guessing the real systems functions and variables. Also, the code was pretty simple and in many cases, should have been more complicated, but it could be executable. At last, the AI , as in Phase 2. Chat 1, used all of the domain classes I provided it with, without using anything more to write the code for the step definitions, which is pretty good and means it has a clear understanding of what it should do with the tools/information it has been given.*

# Phase 2. Chat 2

**User**

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

**ChatGPT**

No questions so far. Please go ahead and provide me with the features/scenarios and the names of the domain classes. Once I have that information, I'll proceed with implementing the Step definitions in Java, following the architecture you specified.

**User**

**The Domain classes you will need are : Book, Borrower, BorrowerCategory, Item, ItemState, Loan, Person. The Dao classes you will need are: borrowerDao, loanDao, itemDao. The services you will need are :loanService, NotificationService, ReturnService. In the next message, I will provide you with all the features of the system**

**ChatGPT**

Great! Please go ahead and provide me with all the features of the system, and I'll start working on the implementation of Step definitions for each feature.

**User**

**The first feature is :**

**Feature: Borrower handling by the system**

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

**User Story: As a librarian,**

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

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

**Scenario: Registering a new borrower**

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

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

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

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

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

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

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

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

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

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

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

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

**When George Red updates his borrowing details**

**Then the system saves the changes**

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

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

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

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

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

**Scenario: Deleting a borrower**

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

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

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

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

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

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

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

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

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

**Scenario: Handling unreturned items of books**

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

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

**And George Red has pending items**

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

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

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

**The second feature is :**

**Feature: Delayed Return Notification**

**User Story:As a Library Manager**

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

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

**Scenario: Notifying the borrower via email**

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

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

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

**And George Red has an email address**

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

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

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

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

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

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

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

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

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

**The third feature is :**

**Feature: Loaning items**

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

**on some conditions of the system**

**User story: As a library member**

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

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

**Scenario: Successful loaning of an item**

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

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

**And George Red is a registered borrower**

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

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

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

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

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

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

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

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

**And George Red is a registered borrower**

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

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

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

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

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

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

**Scenario:Item not found**

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

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

**And George Red is a registered borrower**

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

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

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

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

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

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

**And George Red is a registered borrower**

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

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

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

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

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

**and the last feature is :**

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

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

**User story: As a librarian**

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

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

**Scenario:Successful return of an item**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**ChatGPT**

Thanks for providing the features. I'll start by implementing the Step definitions for the first feature, "Borrower handling by the system," based on the provided scenarios. We'll follow the architecture you mentioned, including Domain classes, DAO classes, and Service classes.

Here's the implementation of the Step definitions for the first feature:

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.When;

public class BorrowerStepDefinitions {

// Injecting necessary DAO and Service classes

private BorrowerDao borrowerDao;

private LoanService loanService;

public BorrowerStepDefinitions(BorrowerDao borrowerDao, LoanService loanService) {

this.borrowerDao = borrowerDao;

this.loanService = loanService;

}

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

public void borrowerIsNotRegistered() {

// Implement logic to check if borrower is not registered

}

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

public void registerBorrower() {

// Implement logic to register borrower

}

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

public void borrowerDetailsStoredSuccessfully() {

// Implement logic to verify successful registration

}

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

public void borrowerIsRegistered() {

// Implement logic to check if borrower is registered

}

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

public void userAlreadyExists() {

// Implement logic to inform that user already exists

}

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

public void updateBorrowerDetails() {

// Implement logic to update borrower details

}

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

public void saveChanges() {

// Implement logic to save changes

}

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

public void displayErrorMessage() {

// Implement logic to display error message

}

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

public void deleteBorrowerAccount() {

// Implement logic to delete borrower account

}

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

public void removeBorrowerDetails() {

// Implement logic to remove borrower details

}

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

public void borrowerDoesNotExist() {

// Implement logic to inform that borrower does not exist

}

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

public void borrowerWithPendingItems() {

// Implement logic to check if borrower has pending items

}

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

public void doNotRemoveBorrowerDetails() {

// Implement logic to not remove borrower details

}

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

public void informAboutPendingItems() {

// Implement logic to inform about pending items

}

}

This is the implementation of Step definitions for the first feature. I'll proceed with the next features as well. Let me know if you need any modifications or additions.

**User**

**Give me the code for the step definitions, in order for the to be executable and make any necessary assumptions**

**ChatGPT**

Certainly! To make the step definitions executable, we'll need to provide implementations for each step using the necessary domain classes, DAO classes, and service classes. Since I don't have access to your entire codebase and dependencies, I'll make some assumptions and provide you with a skeleton implementation that you can adapt to your specific setup. Here's the executable implementation:

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.When;

public class BorrowerStepDefinitions {

private BorrowerDao borrowerDao; // Assuming BorrowerDao is a DAO class for managing borrowers

private LoanService loanService; // Assuming LoanService is a service class for managing loans

private Borrower borrower; // Assuming Borrower is a domain class representing a borrower

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

public void borrowerIsNotRegistered() {

// Assuming borrower is not present in the database

// If using an in-memory database, you can clear the database or reset the state here

// Alternatively, check if George Red is not present in the database

}

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

public void registerBorrower() {

// Assuming borrower details are provided, create a new borrower object

borrower = new Borrower("George Red", "UniqueBorrowerNumber", ...); // Assuming constructor parameters

// Save the borrower details using BorrowerDao

borrowerDao.save(borrower);

}

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

public void borrowerDetailsStoredSuccessfully() {

// Assuming the borrower was successfully saved in the database

// You can verify by fetching the borrower from the database and checking if details match

Borrower savedBorrower = borrowerDao.findByBorrowerNumber(borrower.getBorrowerNumber());

assert savedBorrower != null;

assert savedBorrower.equals(borrower);

}

// Similarly, implement the rest of the step definitions following the same pattern

// You'll need to inject dependencies and handle scenarios accordingly

}

In this implementation:

- `BorrowerDao` is assumed to be a Data Access Object responsible for managing borrowers in the database.

- `LoanService` is assumed to be a Service class responsible for managing loans.

- `Borrower` is assumed to be a domain class representing a borrower, and it contains necessary attributes like name, unique borrower number, etc.

You'll need to replace placeholder comments with actual logic according to your application structure and requirements. Make sure to inject necessary dependencies, handle database operations, and verify results accordingly.

If you have any specific requirements or if you need further assistance with any part of the implementation, feel free to ask!

**User**

**I want you to create as much code as possible, with the information you are provided with , for every step definition**

**ChatGPT**

Certainly! I'll provide more detailed implementations for each step definition, including assumptions about the domain classes and services. Let's start with the first feature:

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.When;

public class BorrowerStepDefinitions {

private BorrowerDao borrowerDao;

private LoanService loanService;

private Borrower borrower;

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

public void borrowerIsNotRegistered() {

// Assuming BorrowerDao can check if borrower exists by borrower name

boolean borrowerExists = borrowerDao.existsByName("George Red");

assert !borrowerExists; // Assuming existsByName returns true if borrower exists

}

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

public void registerBorrower() {

// Assuming borrower details are provided and Borrower object is created

borrower = new Borrower("George Red", "UniqueBorrowerNumber", ...); // Assuming constructor parameters

borrowerDao.save(borrower);

}

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

public void borrowerDetailsStoredSuccessfully() {

// Assuming borrower is successfully saved in the database

Borrower savedBorrower = borrowerDao.findByBorrowerNumber(borrower.getBorrowerNumber());

assert savedBorrower != null;

assert savedBorrower.equals(borrower);

}

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

public void borrowerIsRegistered() {

// Assuming George Red is already registered in the system

// We don't need to perform any action here as it's a precondition for other scenarios

}

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

public void userAlreadyExists() {

// Assuming system provides a message indicating that user already exists

// This can be done by throwing an exception or returning a specific message

// For example:

throw new UserAlreadyExistsException("User already exists"); // Custom exception

}

// Implementations for the rest of the steps

}

In this implementation:

- We assume the existence of a `BorrowerDao` class with methods like `existsByName` and `save`.

- `LoanService` is assumed to handle operations related to loans.

- `Borrower` is assumed to be a domain class with necessary attributes and methods.

You'll need to replace placeholder comments with actual logic according to your application structure and requirements. If you need further clarification or additional assistance, feel free to ask!

**User**

**implement all the step definitions of this feature**

**ChatGPT**

Certainly! Let's implement all the step definitions for the first feature, "Borrower handling by the system":

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.When;

public class BorrowerStepDefinitions {

private BorrowerDao borrowerDao;

private LoanService loanService;

private Borrower borrower;

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

public void borrowerIsNotRegistered() {

// Assuming BorrowerDao can check if borrower exists by borrower name

boolean borrowerExists = borrowerDao.existsByName("George Red");

assert !borrowerExists; // Assuming existsByName returns true if borrower exists

}

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

public void registerBorrower() {

// Assuming borrower details are provided and Borrower object is created

borrower = new Borrower("George Red", "UniqueBorrowerNumber", ...); // Assuming constructor parameters

borrowerDao.save(borrower);

}

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

public void borrowerDetailsStoredSuccessfully() {

// Assuming borrower is successfully saved in the database

Borrower savedBorrower = borrowerDao.findByBorrowerNumber(borrower.getBorrowerNumber());

assert savedBorrower != null;

assert savedBorrower.equals(borrower);

}

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

public void borrowerIsRegistered() {

// Assuming George Red is already registered in the system

// We don't need to perform any action here as it's a precondition for other scenarios

}

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

public void userAlreadyExists() {

// Assuming system provides a message indicating that user already exists

// This can be done by throwing an exception or returning a specific message

// For example:

throw new UserAlreadyExistsException("User already exists"); // Custom exception

}

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

public void updateBorrowerDetails() {

// Assuming borrower details are updated and saved in the system

// For simplicity, let's assume we update the borrower's name

borrower.setName("New Name");

borrowerDao.update(borrower); // Assuming update method in BorrowerDao

}

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

public void saveChanges() {

// Assuming changes are successfully saved in the database

// We can verify by fetching the borrower from the database and checking if details match

Borrower updatedBorrower = borrowerDao.findByBorrowerNumber(borrower.getBorrowerNumber());

assert updatedBorrower != null;

assert updatedBorrower.getName().equals("New Name");

}

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

public void displayErrorMessage() {

// Assuming system displays an error message indicating that George Red does not exist

// This can be done by throwing an exception or returning a specific message

// For example:

throw new BorrowerNotFoundException("George Red does not exist"); // Custom exception

}

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

public void deleteBorrowerAccount() {

// Assuming borrower account is deleted from the system

borrowerDao.delete(borrower); // Assuming delete method in BorrowerDao

}

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

public void removeBorrowerDetails() {

// Assuming borrower details are successfully removed from the system

Borrower deletedBorrower = borrowerDao.findByBorrowerNumber(borrower.getBorrowerNumber());

assert deletedBorrower == null; // Verifying that borrower is deleted

}

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

public void borrowerDoesNotExist() {

// Assuming system informs that the borrower does not exist

// This can be done by throwing an exception or returning a specific message

// For example:

throw new BorrowerNotFoundException("Borrower does not exist"); // Custom exception

}

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

public void borrowerWithPendingItems() {

// Assuming George Red is registered and has pending items in the system

// We don't need to perform any action here as it's a precondition for other scenarios

}

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

public void doNotRemoveBorrowerDetails() {

// Assuming system does not remove borrower details due to pending items

// This can be done by throwing an exception or returning a specific message

// For example:

throw new PendingItemsException("Cannot remove borrower due to pending items"); // Custom exception

}

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

public void informAboutPendingItems() {

// Assuming system informs about pending items

// This can be done by throwing an exception or returning a specific message

// For example:

throw new PendingItemsException("Borrower has pending items"); // Custom exception

}

}

In this implementation, we have added detailed logic for each step definition. Assumptions are made about the existence of domain classes, DAO classes, and service classes, as well as the behavior of methods such as save, update, delete, etc. You'll need to adapt this code to fit your specific application structure and requirements. If you have any questions or need further assistance, feel free to ask!

**User**

**Based on the above and the pattern you created this feature, create the step definitions for the feature: Loaning items**

**ChatGPT**

Sure! Let's create the step definitions for the "Loaning items" feature based on the pattern we established:

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.When;

public class LoaningStepDefinitions {

private ItemDao itemDao;

private BorrowerDao borrowerDao;

private LoanService loanService;

private Borrower borrower;

private Item item;

@Given("^the library has the item (.\*) available$")

public void libraryHasItemAvailable(String itemName) {

// Assuming item is available in the library

item = itemDao.findByName(itemName);

assert item != null && item.getState() == ItemState.AVAILABLE;

}

@Given("^the library has the items (.\*) and (.\*) available$")

public void libraryHasItemsAvailable(String item1Name, String item2Name) {

// Assuming both items are available in the library

Item item1 = itemDao.findByName(item1Name);

Item item2 = itemDao.findByName(item2Name);

assert item1 != null && item1.getState() == ItemState.AVAILABLE;

assert item2 != null && item2.getState() == ItemState.AVAILABLE;

}

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

public void borrowerIsRegistered() {

// Assuming George Red is a registered borrower

borrower = borrowerDao.findByName("George Red");

assert borrower != null;

}

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

public void borrowerHasPendingItems(int pendingItemsCount) {

// Assuming George Red has pending items in the system

// We don't need to perform any action here as it's a precondition for other scenarios

}

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

public void borrowerHasMaximumLendingLimit(int maxLendingLimit) {

// Assuming George Red has been assigned a maximum lending limit

// We don't need to perform any action here as it's a precondition for other scenarios

}

@When("^George Red borrows the item (.\*)$")

public void borrowerBorrowsItem(String itemName) {

// Assuming borrower borrows the item

loanService.loanItem(borrower, item);

}

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

public void systemLoansItemToBorrower(String itemName) {

// Assuming item is successfully loaned to borrower and due date is set

// We can verify by checking the loan details in the system

Loan loan = loanService.findLoanByBorrowerAndItem(borrower, item);

assert loan != null && loan.getBorrower().equals(borrower) && loan.getItem().equals(item);

assert loan.getDueDate() != null;

}

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

public void pendingItemsIncrease(int newPendingItemsCount) {

// Assuming borrower's pending items count is updated in the system

// We can verify by fetching borrower details from the database

borrower = borrowerDao.findById(borrower.getId());

assert borrower != null && borrower.getPendingItemsCount() == newPendingItemsCount;

}

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

public void systemDoesNotLoanItemDueToLimit(String itemName) {

// Assuming system does not loan item due to lending limit reached

// This can be indicated by throwing an exception or returning a specific message

// For example:

throw new LendingLimitReachedException("Lending limit reached"); // Custom exception

}

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

public void systemReturnsErrorDueToItemStatus() {

// Assuming system returns an error due to item's status

// This can be indicated by throwing an exception or returning a specific message

// For example:

throw new ItemNotAvailableException("Item not available"); // Custom exception

}

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

public void systemDoesNotAllowLoan() {

// Assuming system doesn't allow the loan

// This can be indicated by throwing an exception or returning a specific message

// For example:

throw new LoanNotAllowedException("Loan not allowed"); // Custom exception

}

}

In this implementation:

- We have defined step definitions for each scenario in the "Loaning items" feature.

- Assumptions are made about the existence of domain classes (such as `Item`, `Borrower`, and `Loan`), DAO classes (`ItemDao` and `BorrowerDao`), and service classes (`LoanService`).

- We simulate the behavior of the system based on each step in the scenarios.

You will need to replace the placeholder logic with actual logic according to your application structure and requirements. If you have any questions or need further assistance, feel free to ask!

**User**

**Based on the above and the pattern you created this feature, create the step definitions for the feature: Returning a borrowed item to the library**

**ChatGPT**

Sure! Let's create the step definitions for the "Returning a borrowed item to the library" feature based on the pattern we established:

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.When;

public class ReturningStepDefinitions {

private LoanDao loanDao;

private ItemDao itemDao;

private BorrowerDao borrowerDao;

private Borrower borrower;

private Item item;

private Loan loan;

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

public void borrowerBorrowedItemDaysAgo(String itemName, int days) {

// Assuming loan exists in the system with given parameters

borrower = borrowerDao.findByName("George Red");

item = itemDao.findByName(itemName);

loan = loanDao.findLoanByBorrowerAndItem(borrower, item);

assert loan != null && loan.getDueDate().isBefore(LocalDate.now().minusDays(days));

}

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

public void borrowerHasMaximumLendingDays(int maxLendingDays) {

// Assuming borrower has been assigned maximum lending days

// We don't need to perform any action here as it's a precondition for other scenarios

}

@When("^the return of (.\*) is processed$")

public void processItemReturn(String itemName) {

// Assuming return of item is processed

loanService.returnItem(loan);

}

@Then("^the system marks the state of (.\*) as AVAILABLE$")

public void systemMarksItemAsAvailable(String itemName) {

// Assuming item's state is updated to AVAILABLE

item = itemDao.findByName(itemName);

assert item != null && item.getState() == ItemState.AVAILABLE;

}

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

public void borrowerHasOneLessPendingItem() {

// Assuming borrower's pending items count is updated in the system

borrower = borrowerDao.findById(borrower.getId());

assert borrower != null && borrower.getPendingItemsCount() == loan.getBorrower().getPendingItemsCount() - 1;

}

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

public void borrowerDoesNotPayFine() {

// Assuming borrower does not pay a fine

assert loan.getFineAmount() == 0;

}

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

public void returnDateSetToToday() {

// Assuming return date of the loan is updated to today's date

assert loan.getReturnDate().equals(LocalDate.now());

}

@Then("^the system marks the state of (.\*) as AVAILABLE and (.\*) as UNAVAILABLE$")

public void systemMarksItemsAsAvailableAndUnavailable(String returnedItem, String otherItem) {

// Assuming both items' states are updated accordingly

Item returnedItem = itemDao.findByName(returnedItem);

Item otherItem = itemDao.findByName(otherItem);

assert returnedItem != null && returnedItem.getState() == ItemState.AVAILABLE;

assert otherItem != null && otherItem.getState() == ItemState.UNAVAILABLE;

}

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

public void borrowerPaysFineBasedOnCategory() {

// Assuming borrower pays fine based on borrower category

assert loan.getFineAmount() > 0;

}

}

In this implementation:

- We have defined step definitions for each scenario in the "Returning a borrowed item to the library" feature.

- Assumptions are made about the existence of domain classes (such as `Item`, `Borrower`, and `Loan`), DAO classes (`ItemDao`, `BorrowerDao`, and `LoanDao`), and service classes (`LoanService`).

- We simulate the behavior of the system based on each step in the scenarios.

You will need to replace the placeholder logic with actual logic according to your application structure and requirements. If you have any questions or need further assistance, feel free to ask!

**User**

**Based on the above and the pattern you created this feature, create the step definitions for the feature: Delayed Return Notification**

**ChatGPT**

Let's create the step definitions for the "Delayed Return Notification" feature based on the established pattern:

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.When;

public class DelayedReturnNotificationStepDefinitions {

private LoanDao loanDao;

private NotificationService notificationService;

private Borrower borrower;

private Item item;

private Loan loan;

@Given("^George Red has borrowed the item (.\*)$")

public void borrowerHasBorrowedItem(String itemName) {

// Assuming George Red has borrowed the specified item

borrower = borrowerDao.findByName("George Red");

item = itemDao.findByName(itemName);

loan = loanDao.findLoanByBorrowerAndItem(borrower, item);

assert loan != null;

}

@Given("^the due date of (.\*) has passed$")

public void dueDateHasPassed(String itemName) {

// Assuming the due date of the specified item has passed

loan = loanDao.findLoanByBorrowerAndItem(borrower, item);

assert loan != null && loan.getDueDate().isBefore(LocalDate.now());

}

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

public void borrowerHasEmailAddress() {

// Assuming George Red has an email address

assert borrower != null && borrower.getEmail() != null && !borrower.getEmail().isEmpty();

}

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

public void borrowerDoesNotHaveEmailAddress() {

// Assuming George Red does not have an email address

assert borrower != null && (borrower.getEmail() == null || borrower.getEmail().isEmpty());

}

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

public void executeDelayedReturnNotificationProcess() {

// Assuming the system executes the delayed return notification process

notificationService.sendDelayedReturnNotification(borrower, item);

}

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

public void borrowerReceivesEmailNotification() {

// Assuming George Red receives an email notification for the return of the item

// We can verify by checking the notification status in the system

assert notificationService.isNotificationSent(borrower, item);

}

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

public void borrowerDoesNotReceiveEmailNotification() {

// Assuming George Red does not receive an email notification for the return of the item

// We can verify by checking the notification status in the system

assert !notificationService.isNotificationSent(borrower, item);

}

}

In this implementation:

- We define step definitions for each scenario in the "Delayed Return Notification" feature.

- Assumptions are made about the existence of domain classes (such as `Item`, `Borrower`, and `Loan`), DAO classes (`LoanDao`), and service classes (`NotificationService`).

- We simulate the behavior of the system based on each step in the scenarios.

You will need to replace the placeholder logic with actual logic according to your application structure and requirements. If you have any questions or need further assistance, feel free to ask!

**User**

**based on the above, create the code for the feature: Loaning items**

**ChatGPT**

Certainly! Below are the step definitions for the "Loaning items" feature:

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.en.When;

public class LoaningStepDefinitions {

private ItemDao itemDao;

private BorrowerDao borrowerDao;

private LoanService loanService;

private Borrower borrower;

private Item item;

@Given("^the library has the item (.\*) available$")

public void libraryHasItemAvailable(String itemName) {

item = itemDao.findByName(itemName);

assert item != null && item.getState() == ItemState.AVAILABLE;

}

@Given("^the library has the items (.\*) and (.\*) available$")

public void libraryHasItemsAvailable(String item1Name, String item2Name) {

Item item1 = itemDao.findByName(item1Name);

Item item2 = itemDao.findByName(item2Name);

assert item1 != null && item1.getState() == ItemState.AVAILABLE;

assert item2 != null && item2.getState() == ItemState.AVAILABLE;

}

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

public void borrowerIsRegistered() {

borrower = borrowerDao.findByName("George Red");

assert borrower != null;

}

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

public void borrowerHasPendingItems(int pendingItemsCount) {

// Assuming George Red has pending items in the system

}

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

public void borrowerHasMaximumLendingLimit(int maxLendingLimit) {

// Assuming George Red has been assigned a maximum lending limit

}

@When("^George Red borrows the item (.\*)$")

public void borrowerBorrowsItem(String itemName) {

item = itemDao.findByName(itemName);

loanService.loanItem(borrower, item);

}

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

public void systemLoansItemToBorrower(String itemName) {

Loan loan = loanService.findLoanByBorrowerAndItem(borrower, item);

assert loan != null && loan.getBorrower().equals(borrower) && loan.getItem().equals(item);

assert loan.getDueDate() != null;

}

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

public void pendingItemsIncrease(int newPendingItemsCount) {

borrower = borrowerDao.findById(borrower.getId());

assert borrower != null && borrower.getPendingItemsCount() == newPendingItemsCount;

}

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

public void systemDoesNotLoanItemDueToLimit(String itemName) {

// Assuming system does not loan item due to lending limit reached

}

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

public void systemReturnsErrorDueToItemStatus() {

// Assuming system returns an error due to item's status

}

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

public void systemDoesNotAllowLoan() {

// Assuming system doesn't allow the loan

}

}

In these step definitions:

- We handle scenarios such as borrowing items, checking the availability of items, verifying borrower details, and enforcing lending limits.

- Assumptions are made about the existence of domain classes (such as `Item` and `Borrower`), DAO classes (`ItemDao` and `BorrowerDao`), and service classes (`LoanService`).

- Placeholder logic is used for actions like borrowing items and checking pending items, which you will need to replace with your actual application logic.

Let me know if you need further assistance or clarification!