*In this chat , I provided the exact same information as in the earlier chat 3,4 but I instructed the Ai to give me the domain code(even If it gives it alone as well) and the domain classes were exactly the same as before, and then I continued the conversation with the rest of the features, and the code was pretty good but not something very different from chat 3 and 4.*

# Phase 1. Chat 5

**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 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?**

As an AI, I don't ask questions, but I can certainly help you with your request. Please provide the Gherkin Scenarios and I will assist you in creating the corresponding Step definitions in Java, along with the architecture you specified.

**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**

**I want you to firstly create the domain classes , the daos and the services**

Sure, let's start by creating the domain classes. We will need classes for `Borrower`, `Item`, `Loan`, and `Notification`.

Here is the `Borrower` class:

public class Borrower {

private String name;

private String borrowerNumber;

private String emailAddress;

private int pendingItems;

private int lendingLimit;

// getters and setters

}

Next, the `Item` class:

public class Item {

private String title;

private String status;

private String dueDate;

// getters and setters

}

The `Loan` class:

public class Loan {

private Borrower borrower;

private Item item;

private String loanDate;

private String dueDate;

private String returnDate;

// getters and setters

}

And finally, the `Notification` class:

public class Notification {

private Borrower borrower;

private String message;

// getters and setters

}

Next, we will create the DAOs (Data Access Objects) for each of these classes. The DAOs will be responsible for handling the data operations related to these classes.

Here is the `BorrowerDAO`:

public class BorrowerDAO {

public void save(Borrower borrower) {

// code to save borrower

}

public void update(Borrower borrower) {

// code to update borrower

}

public void delete(Borrower borrower) {

// code to delete borrower

}

public Borrower find(String borrowerNumber) {

// code to find borrower

return null;

}

}

Similarly, we will have `ItemDAO`, `LoanDAO`, and `NotificationDAO` with similar methods.

Finally, we will create the service classes. These classes will contain the business logic of our application.

Here is the `BorrowerService`:

public class BorrowerService {

private BorrowerDAO borrowerDAO;

public BorrowerService(BorrowerDAO borrowerDAO) {

this.borrowerDAO = borrowerDAO;

}

public void registerBorrower(Borrower borrower) {

// business logic for registering a borrower

}

public void updateBorrower(Borrower borrower) {

// business logic for updating a borrower

}

public void deleteBorrower(Borrower borrower) {

// business logic for deleting a borrower

}

}

Similarly, we will have `ItemService`, `LoanService`, and `NotificationService` with methods that implement the business logic related to items, loans, and notifications respectively.

**Based on the above, create the step definitions for the feature: Borrower handling by the system**

Sure, let's create the step definitions for the feature: Borrower handling by the system. We will use Cucumber for this.

import io.cucumber.java.en.Given;

import io.cucumber.java.en.Then;

import io.cucumber.java.en.When;

import static org.junit.Assert.\*;

public class BorrowerSteps {

private Borrower borrower;

private BorrowerService borrowerService = new BorrowerService(new BorrowerDAO());

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

public void georgeRedIsNotRegisteredAsABorrower() {

borrower = borrowerService.find("George Red");

assertNull(borrower);

}

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

public void georgeRedGetsRegisteredInTheSystemWithAUniqueBorrowerNumberAndHisDetails() {

borrower = new Borrower();

borrower.setName("George Red");

borrowerService.registerBorrower(borrower);

}

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

public void theSystemSuccessfullyStoresTheBorrowerSDetails() {

Borrower storedBorrower = borrowerService.find("George Red");

assertNotNull(storedBorrower);

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

}

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

public void georgeRedIsRegisteredAsABorrower() {

borrower = borrowerService.find("George Red");

assertNotNull(borrower);

}

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

public void theSystemAttemptsToRegisterGeorgeRedWithAUniqueBorrowerNumberAndHisDetails() {

Borrower newBorrower = new Borrower();

newBorrower.setName("George Red");

borrowerService.registerBorrower(newBorrower);

}

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

public void theSystemInformsThatTheUserAlreadyExists() {

Borrower existingBorrower = borrowerService.find("George Red");

assertNotNull(existingBorrower);

assertEquals("George Red", existingBorrower.getName());

}

// Continue with the rest of the scenarios in a similar manner

}

Please note that the actual implementation of the `BorrowerService` and `BorrowerDAO` methods would depend on your specific application requirements and the underlying data storage mechanism. The above code is a simplified example to illustrate how the step definitions could be written.

**give me the rest of the scenarios as well**

Sure, let's continue with the rest of the scenarios for the feature: Borrower handling by the system.

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

public void georgeRedUpdatesHisBorrowingDetails() {

borrower.setBorrowerNumber("New Number");

borrowerService.updateBorrower(borrower);

}

@Then("the system saves the changes")

public void theSystemSavesTheChanges() {

Borrower updatedBorrower = borrowerService.find("George Red");

assertEquals("New Number", updatedBorrower.getBorrowerNumber());

}

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

public void georgeRedTriesToUpdateHisBorrowingDetails() {

borrower = new Borrower();

borrower.setName("George Red");

borrower.setBorrowerNumber("New Number");

borrowerService.updateBorrower(borrower);

}

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

public void theSystemDisplaysAnErrorMessageIndicatingThatGeorgeRedDoesNotExist() {

Borrower nonExistentBorrower = borrowerService.find("George Red");

assertNull(nonExistentBorrower);

}

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

public void theSystemDeletesGeorgeRedSAccount() {

borrowerService.deleteBorrower(borrower);

}

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

public void theSystemRemovesGeorgeRedSDetails() {

Borrower deletedBorrower = borrowerService.find("George Red");

assertNull(deletedBorrower);

}

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

public void theSystemAttemptsToDeleteGeorgeRedSAccount() {

borrowerService.deleteBorrower(borrower);

}

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

public void theSystemInformsThatTheBorrowerDoesNotExist() {

Borrower nonExistentBorrower = borrowerService.find("George Red");

assertNull(nonExistentBorrower);

}

@Given("George Red has pending items")

public void georgeRedHasPendingItems() {

borrower.setPendingItems(1);

}

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

public void theSystemDoesNotRemoveGeorgeRedSDetails() {

Borrower existingBorrower = borrowerService.find("George Red");

assertNotNull(existingBorrower);

}

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

public void theSystemInformsAboutThePendingItems() {

Borrower existingBorrower = borrowerService.find("George Red");

assertTrue(existingBorrower.getPendingItems() > 0);

}

Again, the actual implementation of the `BorrowerService` and `BorrowerDAO` methods would depend on your specific application requirements and the underlying data storage mechanism. The above code is a simplified example to illustrate how the step definitions could be written.

**now, based on the above , give me the step definitions for the feature: Item Loaning**

Sure, let's create the step definitions for the feature: Item Loaning. We will use Cucumber for this.

import io.cucumber.java.en.Given;

import io.cucumber.java.en.Then;

import io.cucumber.java.en.When;

import static org.junit.Assert.\*;

public class LoanSteps {

private Borrower borrower;

private Item item;

private Loan loan;

private BorrowerService borrowerService = new BorrowerService(new BorrowerDAO());

private ItemService itemService = new ItemService(new ItemDAO());

private LoanService loanService = new LoanService(new LoanDAO());

@Given("the library has the item Harry Potter available")

public void theLibraryHasTheItemHarryPotterAvailable() {

item = itemService.find("Harry Potter");

assertNotNull(item);

assertEquals("AVAILABLE", item.getStatus());

}

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

public void georgeRedIsARegisteredBorrower() {

borrower = borrowerService.find("George Red");

assertNotNull(borrower);

}

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

public void georgeRedHasPendingItemsToBeReturned(int pendingItems) {

assertEquals(pendingItems, borrower.getPendingItems());

}

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

public void georgeRedHasBeenAssignedAMaximumLendingLimitOf(int lendingLimit) {

assertEquals(lendingLimit, borrower.getLendingLimit());

}

@When("George Red borrows the item Harry Potter")

public void georgeRedBorrowsTheItemHarryPotter() {

loan = new Loan();

loan.setBorrower(borrower);

loan.setItem(item);

loanService.createLoan(loan);

}

@Then("the system successfully loans the item Harry Potter to George Red with a due date set")

public void theSystemSuccessfullyLoansTheItemHarryPotterToGeorgeRedWithADueDateSet() {

Loan createdLoan = loanService.find(loan.getId());

assertNotNull(createdLoan);

assertEquals("George Red", createdLoan.getBorrower().getName());

assertEquals("Harry Potter", createdLoan.getItem().getTitle());

assertNotNull(createdLoan.getDueDate());

}

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

public void georgeRedSPendingItemsIncreaseTo(int pendingItems) {

assertEquals(pendingItems, borrower.getPendingItems());

}

// Continue with the rest of the scenarios in a similar manner

}

Please note that the actual implementation of the `BorrowerService`, `ItemService`, `LoanService` and their respective DAO methods would depend on your specific application requirements and the underlying data storage mechanism. The above code is a simplified example to illustrate how the step definitions could be written.

**give me the rest of the scenarios as well**

Sure, let's continue with the rest of the scenarios for the feature: Item Loaning.

@Given("the library has the items Harry Potter and Moby Dick available")

public void theLibraryHasTheItemsHarryPotterAndMobyDickAvailable() {

Item item2 = itemService.find("Moby Dick");

assertNotNull(item);

assertNotNull(item2);

assertEquals("AVAILABLE", item.getStatus());

assertEquals("AVAILABLE", item2.getStatus());

}

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

public void georgeRedTriesToBorrowBothItems() {

Loan loan2 = new Loan();

loan2.setBorrower(borrower);

loan2.setItem(itemService.find("Moby Dick"));

loanService.createLoan(loan2);

}

@Then("the system successfully loans the item Harry Potter to George Red with a due date set")

public void theSystemSuccessfullyLoansTheItemHarryPotterToGeorgeRedWithADueDateSet() {

Loan createdLoan = loanService.find(loan.getId());

assertNotNull(createdLoan);

assertEquals("George Red", createdLoan.getBorrower().getName());

assertEquals("Harry Potter", createdLoan.getItem().getTitle());

assertNotNull(createdLoan.getDueDate());

}

@Then("the system does not loan Moby Dick to George Red due to the lending limit reached")

public void theSystemDoesNotLoanMobyDickToGeorgeRedDueToTheLendingLimitReached() {

Loan createdLoan = loanService.find(loan2.getId());

assertNull(createdLoan);

}

@Given("the item Harry Potter is in the library but not in the system")

public void theItemHarryPotterIsInTheLibraryButNotInTheSystem() {

item = itemService.find("Harry Potter");

assertNull(item);

}

@When("George Red tries to borrow the item Harry Potter")

public void georgeRedTriesToBorrowTheItemHarryPotter() {

loan = new Loan();

loan.setBorrower(borrower);

loan.setItem(item);

loanService.createLoan(loan);

}

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

public void theSystemReturnsAnErrorDueToTheItemSStatus() {

Loan createdLoan = loanService.find(loan.getId());

assertNull(createdLoan);

}

@Then("the system withdraws the item Harry Potter")

public void theSystemWithdrawsTheItemHarryPotter() {

Item withdrawnItem = itemService.find("Harry Potter");

assertNull(withdrawnItem);

}

@When("George Red tries to borrow the item Harry Potter")

public void georgeRedTriesToBorrowTheItemHarryPotter() {

loan = new Loan();

loan.setBorrower(borrower);

loan.setItem(item);

loanService.createLoan(loan);

}

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

public void theSystemDoesnTAllowTheLoan() {

Loan createdLoan = loanService.find(loan.getId());

assertNull(createdLoan);

}

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

public void georgeRedSPendingItemsRemain(int pendingItems) {

assertEquals(pendingItems, borrower.getPendingItems());

}

Again, the actual implementation of the `BorrowerService`, `ItemService`, `LoanService` and their respective DAO methods would depend on your specific application requirements and the underlying data storage mechanism. The above code is a simplified example to illustrate how the step definitions could be written.

**now, based on all the above, give me all the step definitions of the feature: Returing a borrowed item to the library**

Sure, let's create the step definitions for the feature: Returning a borrowed item to the library. We will use Cucumber for this.

import io.cucumber.java.en.Given;

import io.cucumber.java.en.Then;

import io.cucumber.java.en.When;

import static org.junit.Assert.\*;

public class ReturnSteps {

private Borrower borrower;

private Item item;

private Loan loan;

private BorrowerService borrowerService = new BorrowerService(new BorrowerDAO());

private ItemService itemService = new ItemService(new ItemDAO());

private LoanService loanService = new LoanService(new LoanDAO());

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

public void georgeRedBorrowedTheItemAnimalKingdomDaysPriorToTodaySDate(int days) {

loan = loanService.findLoanByBorrowerAndItem("George Red", "Animal Kingdom");

assertNotNull(loan);

// Assuming that the loanDate is stored as a string in the format "yyyy-MM-dd"

String expectedLoanDate = LocalDate.now().minusDays(days).toString();

assertEquals(expectedLoanDate, loan.getLoanDate());

}

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

public void georgeRedHasBeenAssignedMaximumLendingDaysOf(int lendingDays) {

assertEquals(lendingDays, borrower.getLendingLimit());

}

@When("the return of Animal Kingdom is processed")

public void theReturnOfAnimalKingdomIsProcessed() {

loanService.processReturn(loan);

}

@Then("the system marks the state of Animal Kingdom as AVAILABLE")

public void theSystemMarksTheStateOfAnimalKingdomAsAVAILABLE() {

Item returnedItem = itemService.find("Animal Kingdom");

assertEquals("AVAILABLE", returnedItem.getStatus());

}

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

public void georgeRedHasOneLessPendingItem() {

assertEquals(loan.getBorrower().getPendingItems() - 1, borrower.getPendingItems());

}

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

public void georgeRedDoesNotPayAFine() {

// Assuming that the fine is stored in the Loan object

assertEquals(0, loan.getFine());

}

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

public void theReturnDateOfTheLoanIsSetToTodaySDate() {

String expectedReturnDate = LocalDate.now().toString();

assertEquals(expectedReturnDate, loan.getReturnDate());

}

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

public void georgeRedPaysAFineBasedOnTheBorrowerCategory() {

// Assuming that the fine is calculated based on the borrower category

int expectedFine = calculateFineBasedOnBorrowerCategory(borrower.getCategory());

assertEquals(expectedFine, loan.getFine());

}

private int calculateFineBasedOnBorrowerCategory(String category) {

// Implement the logic to calculate the fine based on the borrower category

return 0;

}

}

Please note that the actual implementation of the `BorrowerService`, `ItemService`, `LoanService` and their respective DAO methods would depend on your specific application requirements and the underlying data storage mechanism. The above code is a simplified example to illustrate how the step definitions could be written.

**based on the above, give me the step definitions for the feature: Delayed notification**

Sure, let's create the step definitions for the feature: Delayed notification. We will use Cucumber for this.

import io.cucumber.java.en.Given;

import io.cucumber.java.en.Then;

import io.cucumber.java.en.When;

import static org.junit.Assert.\*;

public class NotificationSteps {

private Borrower borrower;

private Loan loan;

private BorrowerService borrowerService = new BorrowerService(new BorrowerDAO());

private LoanService loanService = new LoanService(new LoanDAO());

private NotificationService notificationService = new NotificationService();

@Given("George Red has borrowed the item Animal Kingdom {int} days ago")

public void georgeRedHasBorrowedTheItemAnimalKingdomDaysAgo(int days) {

loan = loanService.findLoanByBorrowerAndItem("George Red", "Animal Kingdom");

assertNotNull(loan);

// Assuming that the loanDate is stored as a string in the format "yyyy-MM-dd"

String expectedLoanDate = LocalDate.now().minusDays(days).toString();

assertEquals(expectedLoanDate, loan.getLoanDate());

}

@Given("George Red has not returned the item yet")

public void georgeRedHasNotReturnedTheItemYet() {

assertNull(loan.getReturnDate());

}

@When("the system checks for delayed returns")

public void theSystemChecksForDelayedReturns() {

notificationService.checkForDelayedReturns();

}

@Then("the system sends a notification to George Red about the delayed return")

public void theSystemSendsANotificationToGeorgeRedAboutTheDelayedReturn() {

Notification notification = notificationService.findNotificationByBorrowerAndLoan("George Red", loan.getId());

assertNotNull(notification);

assertEquals("Delayed return", notification.getTitle());

assertEquals("George Red", notification.getBorrower().getName());

assertEquals(loan.getId(), notification.getLoan().getId());

}

}

Please note that the actual implementation of the `BorrowerService`, `LoanService`, `NotificationService` and their respective DAO methods would depend on your specific application requirements and the underlying data storage mechanism. The above code is a simplified example to illustrate how the step definitions could be written.