*In this chat, I provided the Ai the same information as in Phase 2. Chat 3, but I instructed the Ai to create the domain code AS well AS the code for the daos and services in order for the system to actually use DAOs. As expected, it did a great job generating the domain classes, and the DAOs and services as well( this was the first chat that the Ai created an interface for the daos). Then I told the Ai to create the code, which was almost copy paste from previous chats 1,2 and 3. Overall, telling the AI to also create the DAOs code helps it use them later.*

# Phase 2. Chat 4

**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?**

As an AI, I don't ask questions but I'm here to assist you. Please provide the Gherkin Scenarios and the names of the domain classes you have. I'll help you create the implementation code for the Step definitions in Java.

**The Domain classes you will need are : Book, Borrower, BorrowerCategory, Item, ItemState, Loan, Person.In the next message, I will provide you with all the features of the system**

Understood. I'm ready to assist you with the implementation of the Step definitions once you provide the features of the system.

**The Gherkin scenarios are :**

**The first feature is :**

**Feature: Borrower handling by the system**

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

**User Story: As a librarian,**

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

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

**Scenario: Registering a new borrower**

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

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

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

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

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

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

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

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

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

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

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

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

**When George Red updates his borrowing details**

**Then the system saves the changes**

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

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

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

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

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

**Scenario: Deleting a borrower**

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

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

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

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

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

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

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

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

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

**Scenario: Handling unreturned items of books**

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

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

**And George Red has pending items**

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

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

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

**The second feature is :**

**Feature: Delayed Return Notification**

**User Story: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 all the domain classes and the implementation code for them, as well as the DAO classes and the services**

Sure, let's start by creating the domain classes.

Here is the `Person` class:

public class Person {

private String name;

private String email;

// constructors, getters and setters

}

The `Borrower` class:

public class Borrower extends Person {

private String borrowerNumber;

private int pendingItems;

private BorrowerCategory category;

// constructors, getters and setters

}

The `BorrowerCategory` class:

public class BorrowerCategory {

private String categoryName;

private int maxLendingLimit;

private int maxLendingDays;

// constructors, getters and setters

}

The `Item` class:

public class Item {

private String itemName;

private ItemState state;

// constructors, getters and setters

}

The `ItemState` enum:

public enum ItemState {

AVAILABLE,

BORROWED

}

The `Loan` class:

import java.time.LocalDate;

public class Loan {

private Borrower borrower;

private Item item;

private LocalDate dueDate;

private LocalDate returnDate;

// constructors, getters and setters

}

Now, let's create the DAO classes.

Here is the `BorrowerDAO` class:

import java.util.List;

public interface BorrowerDAO {

void save(Borrower borrower);

void update(Borrower borrower);

void delete(Borrower borrower);

Borrower findByBorrowerNumber(String borrowerNumber);

List<Borrower> findAll();

}

The `ItemDAO` class:

import java.util.List;

public interface ItemDAO {

void save(Item item);

void update(Item item);

void delete(Item item);

Item findByName(String itemName);

List<Item> findAll();

}

The `LoanDAO` class:

import java.util.List;

public interface LoanDAO {

void save(Loan loan);

void update(Loan loan);

void delete(Loan loan);

Loan findByBorrowerAndItem(Borrower borrower, Item item);

List<Loan> findAll();

}

Finally, let's create the service classes.

Here is the `BorrowerService` class:

public class BorrowerService {

private BorrowerDAO borrowerDAO;

public BorrowerService(BorrowerDAO borrowerDAO) {

this.borrowerDAO = borrowerDAO;

}

public void registerBorrower(Borrower borrower) {

// implementation

}

public void updateBorrower(Borrower borrower) {

// implementation

}

public void deleteBorrower(Borrower borrower) {

// implementation

}

}

The `LoanService` class:

public class LoanService {

private LoanDAO loanDAO;

public LoanService(LoanDAO loanDAO) {

this.loanDAO = loanDAO;

}

public void loanItem(Borrower borrower, Item item) {

// implementation

}

public void returnItem(Borrower borrower, Item item) {

// implementation

}

}

The `NotificationService` class:

public class NotificationService {

public void notifyDelayedReturn(Borrower borrower, Item item) {

// implementation

}

}

Please note that the actual implementation of the methods in the service classes will depend on the specific business logic of your application.

**Based on the above, create the step definitions implementation code for the feature: Borrower Handling by the system**

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

Here is the `BorrowerStepDefs` class:

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 BorrowerStepDefs {

private BorrowerService borrowerService;

private Borrower borrower;

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

public void georgeRedIsNotRegisteredAsABorrower() {

borrower = borrowerService.findByBorrowerNumber("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.findByBorrowerNumber(borrower.getBorrowerNumber());

assertNotNull(storedBorrower);

assertEquals(borrower, storedBorrower);

}

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

public void georgeRedIsRegisteredAsABorrower() {

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

assertNotNull(existingBorrower);

assertEquals(borrower, existingBorrower);

}

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

public void georgeRedUpdatesHisBorrowingDetails() {

borrower.setEmail("george.red@example.com");

borrowerService.updateBorrower(borrower);

}

@Then("the system saves the changes")

public void theSystemSavesTheChanges() {

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

assertEquals("george.red@example.com", updatedBorrower.getEmail());

}

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

public void georgeRedTriesToUpdateHisBorrowingDetails() {

borrower = new Borrower();

borrower.setName("George Red");

borrower.setEmail("george.red@example.com");

borrowerService.updateBorrower(borrower);

}

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

public void theSystemDisplaysAnErrorMessageIndicatingThatGeorgeRedDoesNotExist() {

Borrower nonExistingBorrower = borrowerService.findByBorrowerNumber("George Red");

assertNull(nonExistingBorrower);

}

@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.findByBorrowerNumber("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 nonExistingBorrower = borrowerService.findByBorrowerNumber("George Red");

assertNull(nonExistingBorrower);

}

@Given("George Red has pending items")

public void georgeRedHasPendingItems() {

assertTrue(borrower.getPendingItems() > 0);

}

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

public void theSystemDoesNotRemoveGeorgeRedsDetails() {

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

assertNotNull(existingBorrower);

}

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

public void theSystemInformsAboutThePendingItems() {

assertTrue(borrower.getPendingItems() > 0);

}

}

Please note that the actual implementation of the methods in the step definitions will depend on the specific business logic of your application.

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

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

Here is the `ItemLoaningStepDefs` class:

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 ItemLoaningStepDefs {

private LoanService loanService;

private Borrower borrower;

private Item item;

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

public void theLibraryHasTheItemHarryPotterAvailable() {

item = new Item();

item.setItemName("Harry Potter");

item.setState(ItemState.AVAILABLE);

}

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

public void georgeRedIsARegisteredBorrower() {

borrower = new Borrower();

borrower.setName("George Red");

}

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

public void georgeRedHasPendingItemsToBeReturned(int pendingItems) {

borrower.setPendingItems(pendingItems);

}

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

public void georgeRedHasBeenAssignedAMaximumLendingLimitOf(int maxLendingLimit) {

BorrowerCategory category = new BorrowerCategory();

category.setMaxLendingLimit(maxLendingLimit);

borrower.setCategory(category);

}

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

public void georgeRedBorrowsTheItemHarryPotter() {

loanService.loanItem(borrower, item);

}

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

public void theSystemSuccessfullyLoansTheItemHarryPotterToGeorgeRedWithADueDateSet() {

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

assertNotNull(loan);

assertEquals(borrower, loan.getBorrower());

assertEquals(item, loan.getItem());

assertNotNull(loan.getDueDate());

}

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

public void georgeRedsPendingItemsIncreaseTo(int pendingItems) {

assertEquals(pendingItems, borrower.getPendingItems());

}

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

public void theLibraryHasTheItemsHarryPotterAndMobyDickAvailable() {

// similar to the previous Given step, but for two items

}

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

public void georgeRedTriesToBorrowBothItems() {

// similar to the previous When step, but for two items

}

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

public void theSystemSuccessfullyLoansTheItemHarryPotterToGeorgeRedWithADueDateSet() {

// similar to the previous Then step

}

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

public void theSystemDoesNotLoanMobyDickToGeorgeRedDueToTheLendingLimitReached() {

Loan loan = loanService.findByBorrowerAndItem(borrower, new Item("Moby Dick"));

assertNull(loan);

}

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

public void theItemHarryPotterIsInTheLibraryButNotInTheSystem() {

item = null;

}

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

public void georgeRedTriesToBorrowTheItemHarryPotter() {

try {

loanService.loanItem(borrower, item);

} catch (Exception e) {

// handle exception

}

}

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

public void theSystemReturnsAnErrorDueToTheItemsStatus() {

// verify that an exception was thrown in the previous When step

}

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

public void theSystemWithdrawsTheItemHarryPotter() {

// implementation depends on the business logic

}

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

public void theSystemDoesntAllowTheLoan() {

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

assertNull(loan);

}

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

public void georgeRedsPendingItemsRemain(int pendingItems) {

assertEquals(pendingItems, borrower.getPendingItems());

}

}

Please note that the actual implementation of the methods in the step definitions will depend on the specific business logic of your application.

**give me as much code as possible for the scenarios of the feature above that you didnt give me**

Sure, let's continue with the step definitions for the "Item Loaning" feature.

Here is the `ItemLoaningStepDefs` class:

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 ItemLoaningStepDefs {

private LoanService loanService;

private Borrower borrower;

private Item item;

// ... previous step definitions ...

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

public void theLibraryHasTheItemsHarryPotterAndMobyDickAvailable() {

Item item1 = new Item();

item1.setItemName("Harry Potter");

item1.setState(ItemState.AVAILABLE);

Item item2 = new Item();

item2.setItemName("Moby Dick");

item2.setState(ItemState.AVAILABLE);

}

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

public void georgeRedTriesToBorrowBothItems() {

loanService.loanItem(borrower, new Item("Harry Potter"));

loanService.loanItem(borrower, new Item("Moby Dick"));

}

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

public void theSystemDoesNotLoanMobyDickToGeorgeRedDueToTheLendingLimitReached() {

Loan loan = loanService.findByBorrowerAndItem(borrower, new Item("Moby Dick"));

assertNull(loan);

}

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

public void theItemHarryPotterIsInTheLibraryButNotInTheSystem() {

item = null;

}

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

public void georgeRedTriesToBorrowTheItemHarryPotter() {

try {

loanService.loanItem(borrower, item);

} catch (Exception e) {

// handle exception

}

}

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

public void theSystemReturnsAnErrorDueToTheItemsStatus() {

// verify that an exception was thrown in the previous When step

}

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

public void theSystemWithdrawsTheItemHarryPotter() {

// implementation depends on the business logic

}

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

public void theSystemDoesntAllowTheLoan() {

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

assertNull(loan);

}

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

public void georgeRedsPendingItemsRemain(int pendingItems) {

assertEquals(pendingItems, borrower.getPendingItems());

}

}

Please note that the actual implementation of the methods in the step definitions will depend on the specific business logic of your application.

**based on the above, give me the step definitions for the feature: Returning a borrowed item to the library**

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

Here is the `ItemReturningStepDefs` class:

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 ItemReturningStepDefs {

private LoanService loanService;

private Borrower borrower;

private Item item;

private Loan loan;

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

public void georgeRedBorrowedTheItemAnimalKingdomDaysPriorToTodaysDate(int daysAgo) {

borrower = new Borrower();

borrower.setName("George Red");

item = new Item();

item.setItemName("Animal Kingdom");

item.setState(ItemState.BORROWED);

loan = new Loan();

loan.setBorrower(borrower);

loan.setItem(item);

loan.setDueDate(LocalDate.now().minusDays(daysAgo));

}

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

public void georgeRedHasBeenAssignedMaximumLendingDaysOf(int maxLendingDays) {

BorrowerCategory category = new BorrowerCategory();

category.setMaxLendingDays(maxLendingDays);

borrower.setCategory(category);

}

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

public void theReturnOfAnimalKingdomIsProcessed() {

loanService.returnItem(borrower, item);

}

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

public void theSystemMarksTheStateOfAnimalKingdomAsAVAILABLE() {

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

}

@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() {

// implementation depends on the business logic

}

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

public void theReturnDateOfTheLoanIsSetToTodaysDate() {

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

}

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

public void georgeRedPaysAFineBasedOnTheBorrowerCategory() {

// implementation depends on the business logic

}

}

Please note that the actual implementation of the methods in the step definitions will depend on the specific business logic of your application.

**based on the above, give me as much code as possible for the feature: Delayed Notification**

Sure, let's create the step definitions for the "Delayed Return Notification" feature. We will use Cucumber for this.

Here is the `DelayedNotificationStepDefs` class:

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 DelayedNotificationStepDefs {

private NotificationService notificationService;

private Borrower borrower;

private Item item;

private Loan loan;

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

public void georgeRedHasBorrowedTheItemHarryPotter() {

borrower = new Borrower();

borrower.setName("George Red");

item = new Item();

item.setItemName("Harry Potter");

item.setState(ItemState.BORROWED);

loan = new Loan();

loan.setBorrower(borrower);

loan.setItem(item);

}

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

public void harryPottersDueDateHasPassed() {

loan.setDueDate(LocalDate.now().minusDays(1));

}

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

public void georgeRedHasAnEmailAddress() {

borrower.setEmail("george.red@example.com");

}

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

public void theSystemExecutesTheDelayedReturnNotificationProcess() {

notificationService.notifyDelayedReturn(borrower, item);

}

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

public void georgeRedReceivesAnEmailNotificationForTheReturnOfTheItem() {

// implementation depends on the business logic

}

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

public void georgeRedDoesNotHaveAnEmailAddress() {

borrower.setEmail(null);

}

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

public void georgeRedDoesNotReceiveAnEmailNotificationForTheReturnOfTheItem() {

// implementation depends on the business logic

}

}

Please note that the actual implementation of the methods in the step definitions will depend on the specific business logic of your application.