

Στην πρώτη μου αλληλεπίδραση με το GitHub Copilot, δεν χρειάστηκε να παρέχω τα features του συστήματος, καθώς το σύστημα μπορούσε να διαβάσει αυτόματα τα αρχεία από τον editor. Αντίθετα, παρείχα στο LLM την αρχιτεκτονική και τη γενική γνώση του συστήματος καθώς και τα features του. Αρχικά, το LLM παρήγαγε μόνο τα Step Definitions με κενό σώμα, αλλά μετά από μερικά μηνύματα, κατανόησε την εργασία και άρχισε να δημιουργεί κώδικα με ευκολία. Για πρώτη φορά, με την περιορισμένη γνώση που του είχε δοθεί, το LLM κατάφερε να κατανοήσει τη σημασία των «περσόνων» (π.χ. George Red) και χρησιμοποίησε σωστά τις μεταβλητές Harry Potter και Moby Dick. Ωστόσο, όπως και το ChatGPT, το LLM χρησιμοποίησε συχνά assertions στα βήματα GIVEN, κάτι που δεν είναι η καλύτερη πρακτική για την γραφή των Step Definitions, καθώς τα βήματα GIVEN πρέπει να δημιουργούν το περιβάλλον και τις μεταβλητές για την επαλήθευση του σεναρίου. Συνολικά, το LLM αντελήφθη αμέσως τις εντολές και χρειάστηκε λίγα μηνύματα για να δημιουργήσει τον κώδικα, σε αντίθεση με το ChatGPT. Ο παραγόμενος κώδικας δεν ήταν τέλειος, αλλά ήταν αρκετά καλός σε πολλά σημεία. Το LLM έκανε καλή χρήση των Service κλάσεων, αλλά όχι των Data Access Objects, και παρέλειψε κάποια σημαντικά Step Definitions λόγω της πολυπλοκότητάς τους και της έλλειψης προϋπάρχουσας γνώσης που του είχε δοθεί.

Phase 1. Chat 1

User

Act as a software developer. I have a library system application written in java and I have created some Gherkin Scenarios based on different features of the library application. I will provide you with these features/scenarios, and then I will ask you to provide me the implementation code of the Step definitions (hint: the step definitions will be split into different classes based on the different feature). The architecture layers of the system are a domain layer containing the domain classes, the DAO layer containing the DAO classes and a service layer containing service classes that help with the logic of the system. In the next message , I will provide you with the features/scenarios and then I want you to implement the step definitions code for each step

GitHubCopilot

Sure, I can help with that. Please provide the features/scenarios and I will provide the step definitions code for each step.

User

For each one of these, create the step definitions based on all the information I provided you with

GitHubCopilot

Sure, let's start with the `borrower_handling.feature` file. We will create a class `BorrowerHandlingSteps` to handle the step definitions.

```
import io.cucumber.java.en.Given;
```

```
import io.cucumber.java.en.Then;
```

```
import io.cucumber.java.en.When;
```

```
public class BorrowerHandlingSteps {
```

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

```
    public void georgeRedIsNotRegisteredAsABorrower() {
```

```
        // Code to check if George Red is not registered as a borrower
```

```
    }
```

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

```
    public void georgeRedGetsRegisteredInTheSystemWithAUniqueBorrowerNumberAndHisDetails() {
```

```
        // Code to register George Red in the system
```

```
    }
```

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

public void theSystemSuccessfullyStoresTheBorrowersDetails() {

    // Code to verify that the system has stored the borrower's details

}

// Continue with the rest of the steps...

}
```

For the `item_loaning.feature` file, we will create a class `ItemLoaningSteps` to handle the step definitions.

```
public class ItemLoaningSteps {

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

    public void theLibraryHasTheItemHarryPotterAvailable() {

        // Code to check if the library has the item Harry Potter available

    }

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

    public void georgeRedBorrowsTheItemHarryPotter() {

        // Code for George Red to borrow the item Harry Potter

    }

}
```

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

```
public void theSystemSuccessfullyLoansTheItemHarryPotterToGeorgeRedWithADueDateSet() {
```

```
    // Code to verify that the system has successfully loaned the item Harry Potter to George Red
```

```
}
```

```
    // Continue with the rest of the steps...
```

```
}
```

For the `item_return.feature` file, we will create a class `ItemReturnSteps` to handle the step definitions.

```
public class ItemReturnSteps {
```

```
    @Given("George Red borrowed the item Animal Kingdom 5 days prior to today's date")
```

```
    public void georgeRedBorrowedTheItemAnimalKingdom5DaysPriorToTodaysDate() {
```

```
        // Code to check if George Red borrowed the item Animal Kingdom 5 days prior to today's date
```

```
    }
```

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

```

public void theReturnOfAnimalKingdomIsProcessed() {

    // Code to process the return of Animal Kingdom

}

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

public void theSystemMarksTheStateOfAnimalKingdomAsAVAILABLE() {

    // Code to verify that the system has marked the state of Animal Kingdom as AVAILABLE

}

// Continue with the rest of the steps...

}

```

For the `delay_notification.feature` file, we will create a class `DelayNotificationSteps` to handle the step definitions.

```

public class DelayNotificationSteps {

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

    public void georgeRedHasBorrowedTheItemHarryPotter() {

        // Code to check if George Red has borrowed the item Harry Potter

    }

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

```

```
public void theSystemExecutesTheDelayedReturnNotificationProcess() {  
  
    // Code to execute the delayed return notification process  
  
}  
  
@Then("George Red receives an email notification for the return of the item")  
  
public void georgeRedReceivesAnEmailNotificationForTheReturnOfTheItem() {  
  
    // Code to verify that George Red has received an email notification for the return of the  
item  
  
}  
  
// Continue with the rest of the steps...  
  
}
```

Please note that the actual implementation of these methods will depend on the specific details of your application. The above code is just a skeleton to guide you on how to structure your step definitions.

User

Create all the step definitions for the feature: Borrower handling by the system, with as much code as possible

Sure, let's create the step definitions for the `Borrower handling by the system` feature. We will create a class `BorrowerHandlingSteps` to handle the step definitions.

```
import io.cucumber.java.en.Given;

import io.cucumber.java.en.Then;

import io.cucumber.java.en.When;

import librarysystem.domain.Borrower;

import librarysystem.service.BorrowerService;


public class BorrowerHandlingSteps {


    private BorrowerService borrowerService = new BorrowerService();

    private Borrower borrower;


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

    public void georgeRedIsNotRegisteredAsABorrower() {

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

        if (borrower != null) {

            borrowerService.deleteBorrower(borrower);

        }

    }

}
```

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

```
public void georgeRedGetsRegisteredInTheSystemWithAUniqueBorrowerNumberAndHisDetails() {  
    borrower = new Borrower("George Red", "UniqueBorrowerNumber",  
"GeorgeRedDetails");  
    borrowerService.registerBorrower(borrower);  
}
```

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

```
public void theSystemSuccessfullyStoresTheBorrowersDetails() {  
    Borrower storedBorrower = borrowerService.findBorrower("George Red");  
    assert storedBorrower != null;  
}
```

```
// Continue with the rest of the steps...
```

```
}
```

Please note that the actual implementation of these methods will depend on the specific details of your application. The above code is just a skeleton to guide you on how to structure your step definitions. You would need to replace the `BorrowerService` with the actual service class you are using in your application. Similarly, the `Borrower` class would need to be replaced with the actual domain class you are using.

User

Give me all the step definitions

GitHubCopilot

Sure, let's create the step definitions for all the features. We will create separate classes for each feature to handle the step definitions.

For the `Borrower handling by the system` feature, we will create a class `BorrowerHandlingSteps`:

```
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import librarysystem.domain.Borrower;
import librarysystem.service.BorrowerService;

public class BorrowerHandlingSteps {

    private BorrowerService borrowerService = new BorrowerService();

    private Borrower borrower;

    @Given("George Red is not registered as a borrower")
    public void georgeRedIsNotRegisteredAsABorrower() {
```

```

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

    if (borrower != null) {

        borrowerService.deleteBorrower(borrower);

    }

}

```

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

```

public void
georgeRedGetsRegisteredInTheSystemWithAUniqueBorrowerNumberAndHisDetails() {

    borrower = new Borrower("George Red", "UniqueBorrowerNumber",
"GeorgeRedDetails");

    borrowerService.registerBorrower(borrower);

}

```

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

```

public void theSystemSuccessfullyStoresTheBorrowersDetails() {

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

    assert storedBorrower != null;

}

```

// Continue with the rest of the steps...

```

}

```

For the `Loaning items` feature, we will create a class `ItemLoaningSteps`:

```
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import librarysystem.domain.Item;
import librarysystem.service.ItemService;

public class ItemLoaningSteps {

    private ItemService itemService = new ItemService();

    private Item item;

    @Given("the library has the item Harry Potter available")
    public void theLibraryHasTheItemHarryPotterAvailable() {

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

        assert item != null;
    }

    @When("George Red borrows the item Harry Potter")
    public void georgeRedBorrowsTheItemHarryPotter() {
```

```

        itemService loanItem("Harry Potter", "George Red");
    }

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

    public void theSystemSuccessfullyLoansTheItemHarryPotterToGeorgeRedWithADueDateSet() {

        Item loanedItem = itemService.findItem("Harry Potter");

        assert loanedItem != null && loanedItem.isLoaned();

    }

    // Continue with the rest of the steps...
}

```

For the `Returning a borrowed item to the library` feature, we will create a class `ItemReturnSteps`:

```

import io.cucumber.java.en.Given;

import io.cucumber.java.en.Then;

import io.cucumber.java.en.When;

import librarysystem.domain.Item;

import librarysystem.service.ItemService;

```

```
public class ItemReturnSteps {

    private ItemService itemService = new ItemService();

    private Item item;

    @Given("George Red borrowed the item Animal Kingdom 5 days prior to today's date")
    public void georgeRedBorrowedTheItemAnimalKingdom5DaysPriorToTodaysDate() {

        item = itemService.findItem("Animal Kingdom");

        assert item != null && item.isLoaned();

    }

    @When("the return of Animal Kingdom is processed")
    public void theReturnOfAnimalKingdomIsProcessed() {

        itemService.returnItem("Animal Kingdom");

    }

    @Then("the system marks the state of Animal Kingdom as AVAILABLE")
    public void theSystemMarksTheStateOfAnimalKingdomAsAVAILABLE() {

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

        assert returnedItem != null && !returnedItem.isLoaned();

    }

}
```

```
// Continue with the rest of the steps...  
}
```

For the `Delayed Return Notification` feature, we will create a class `DelayNotificationSteps`:

```
import io.cucumber.java.en.Given;  
  
import io.cucumber.java.en.Then;  
  
import io.cucumber.java.en.When;  
  
import librarysystem.domain.Item;  
  
import librarysystem.service.ItemService;  
  
import librarysystem.service.NotificationService;  
  
public class DelayNotificationSteps {  
  
    private ItemService itemService = new ItemService();  
  
    private NotificationService notificationService = new NotificationService();  
  
    private Item item;  
  
    @Given("George Red has borrowed the item Harry Potter")  
  
    public void georgeRedHasBorrowedTheItemHarryPotter() {  
  
        item = itemService.findItem("Harry Potter");  
  
    }  
  
}
```

```

        assert item != null && item.isLoaned();
    }

    @When("the system executes the delayed return notification process")
    public void theSystemExecutesTheDelayedReturnNotificationProcess() {
        notificationService.sendDelayedReturnNotification("George Red", "Harry Potter");
    }

    @Then("George Red receives an email notification for the return of the item")
    public void georgeRedReceivesAnEmailNotificationForTheReturnOfTheItem() {
        // This step would require access to the email server to verify the email was sent.
        // As such, it might not be possible to implement a verification step here.
    }

    // Continue with the rest of the steps...
}

```

Please note that the actual implementation of these methods will depend on the specific details of your application. The above code is just a skeleton to guide you on how to structure your step definitions. You would need to replace the ``ItemService``, ``BorrowerService`` and ``NotificationService`` with the actual service classes you are using in your application. Similarly, the ``Item`` and ``Borrower`` classes would need to be replaced with the actual domain classes you are using.

User

I want you to create All the step definitions for the feature : Borrower Handling by the system, without skipping any

GitHubCopilot

Sure, let's create the step definitions for the `Borrower handling by the system` feature. We will create a class `BorrowerHandlingSteps` to handle the step definitions.

```
import io.cucumber.java.en.Given;

import io.cucumber.java.en.Then;

import io.cucumber.java.en.When;

import librarysystem.domain.Borrower;

import librarysystem.service.BorrowerService;


public class BorrowerHandlingSteps {


    private BorrowerService borrowerService = new BorrowerService();

    private Borrower borrower;


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

    public void georgeRedIsNotRegisteredAsABorrower() {

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

        if (borrower != null) {
```



```
        borrowerService.deleteBorrower(borrower);  
    }  
}
```

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

```
public void georgeRedGetsRegisteredInTheSystemWithAUniqueBorrowerNumberAndHisDetails() {  
    borrower = new Borrower("George Red", "UniqueBorrowerNumber",  
"GeorgeRedDetails");  
    borrowerService.registerBorrower(borrower);  
}
```

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

```
public void theSystemSuccessfullyStoresTheBorrowersDetails() {  
    Borrower storedBorrower = borrowerService.findBorrower("George Red");  
    assert storedBorrower != null;  
}
```

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

```
public void georgeRedIsRegisteredAsABorrower() {  
    borrower = borrowerService.findBorrower("George Red");  
    assert borrower != null;  
}
```

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

```
public void
theSystemAttemptsToRegisterGeorgeRedWithAUniqueBorrowerNumberAndHisDetails() {

    try {

        borrower = new Borrower("George Red", "UniqueBorrowerNumber",
"GeorgeRedDetails");

        borrowerService.registerBorrower(borrower);

    } catch (Exception e) {

        // Handle exception

    }

}
```

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

```
public void theSystemInformsThatTheUserAlreadyExists() {

    // This step would require access to the system's response to verify the user already
exists.

    // As such, it might not be possible to implement a verification step here.

}
```

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

```
public void georgeRedUpdatesHisBorrowingDetails() {

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

}
```

```

    if (borrower != null) {

        borrower.setDetails("UpdatedDetails");

        borrowerService.updateBorrower(borrower);

    }

}

```

```

@Then("the system saves the changes")

public void theSystemSavesTheChanges() {

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

    assert      updatedBorrower      !=      null      &&
    "UpdatedDetails".equals(updatedBorrower.getDetails());

}

```

```

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

public void georgeRedTriesToUpdateHisBorrowingDetails() {

    try {

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

        if (borrower == null) {

            throw new Exception("Borrower not found");

        }

        borrower.setDetails("UpdatedDetails");

        borrowerService.updateBorrower(borrower);

    } catch (Exception e) {

        // Handle exception
    }
}

```

```
}  
  
}
```

```
@Then("the system displays an error message indicating that George Red does not exist")  
  
public void theSystemDisplaysAnErrorMessageIndicatingThatGeorgeRedDoesNotExist() {  
  
    // This step would require access to the system's response to verify the error message.  
  
    // As such, it might not be possible to implement a verification step here.  
  
}
```

```
@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.findBorrower("George Red");  
  
    assert deletedBorrower == null;  
  
}
```

```
@When("the system attempts to delete George Red's account")  
  
public void theSystemAttemptsToDeleteGeorgeRedsAccount() {  
  
    try {
```

```
        borrowerService.deleteBorrower(borrower);

    } catch (Exception e) {

        // Handle exception

    }

}
```

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

public void theSystemInformsThatTheBorrowerDoesNotExist() {

    // This step would require access to the system's response to verify the borrower does
    not exist.

    // As such, it might not be possible to implement a verification step here.

}
```

```
@Given("George Red has pending items")

public void georgeRedHasPendingItems() {

    // This step would require access to the system's data to verify George Red has pending
    items.

    // As such, it might not be possible to implement a verification step here.

}
```

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

public void theSystemDoesNotRemoveGeorgeRedsDetails() {

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

    assert existingBorrower != null;
```

```

    }

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

    public void theSystemInformsAboutThePendingItems() {

        // This step would require access to the system's response to verify the system informs
        about the pending items.

        // As such, it might not be possible to implement a verification step here.

    }
}

```

Please note that the actual implementation of these methods will depend on the specific details of your application. The above code is just a skeleton to guide you on how to structure your step definitions. You would need to replace the `BorrowerService` with the actual service class you are using in your application. Similarly, the `Borrower` class would need to be replaced with the actual domain class you are using.

User

Now, based on the above, create all the step definitions for the feature: Item Loaning

GitHubCopilot

Sure, let's create the step definitions for the `Item Loaning` feature. We will create a class `ItemLoaningSteps` to handle the step definitions.

```
import io.cucumber.java.en.Given;

import io.cucumber.java.en.Then;

import io.cucumber.java.en.When;

import librarysystem.domain.Borrower;

import librarysystem.domain.Item;

import librarysystem.service.BorrowerService;

import librarysystem.service.ItemService;


public class ItemLoaningSteps {


    private ItemService itemService = new ItemService();

    private BorrowerService borrowerService = new BorrowerService();

    private Item item;

    private Borrower borrower;


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

    public void theLibraryHasTheItemHarryPotterAvailable() {

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

        assert item != null;

    }


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

    public void georgeRedIsARegisteredBorrower() {
```

```
borrower = borrowerService.findBorrower("George Red");

assert borrower != null;

}
```

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

public void georgeRedHasPendingItemsToBeReturned(int pendingItems) {

    assert borrower.getPendingItems() == pendingItems;

}
```

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

public void georgeRedHasBeenAssignedAMaximumLendingLimitOf(int lendingLimit) {

    assert borrower.getLendingLimit() == lendingLimit;

}
```

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

public void georgeRedBorrowsTheItemHarryPotter() {

    itemService loanItem("Harry Potter", "George Red");

}
```

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

```
public void theSystemSuccessfullyLoansTheItemHarryPotterToGeorgeRedWithADueDateSet() {

    Item loanedItem = itemService.findItem("Harry Potter");

}
```



```
    assert loanedItem != null && loanedItem.isLoaned();  
}
```

```
@Then("George Red's pending items increase to {int}")  
  
public void georgeRedsPendingItemsIncreaseTo(int pendingItems) {  
    assert borrower.getPendingItems() == pendingItems;  
}
```

```
@When("George Red tries to borrow both items")  
  
public void georgeRedTriesToBorrowBothItems() {  
    try {  
        itemService.loanItem("Harry Potter", "George Red");  
        itemService.loanItem("Moby Dick", "George Red");  
    } catch (Exception e) {  
        // Handle exception  
    }  
}
```

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

```
public void theSystemDoesNotLoanMobyDickToGeorgeRedDueToTheLendingLimitReached() {  
    Item notLoanedItem = itemService.findItem("Moby Dick");  
    assert notLoanedItem != null && !notLoanedItem.isLoaned();  
}
```

```
}
```

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

```
public void theItemHarryPotterIsInTheLibraryButNotInTheSystem() {
```

```
    item = itemService.findItem("Harry Potter");
```

```
    assert item == null;
```

```
}
```

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

```
public void georgeRedTriesToBorrowTheItemHarryPotter() {
```

```
    try {
```

```
        itemService loanItem("Harry Potter", "George Red");
```

```
    } catch (Exception e) {
```

```
        // Handle exception
```

```
    }
```

```
}
```

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

```
public void theSystemReturnsAnErrorDueToTheItemsStatus() {
```

```
    // This step would require access to the system's response to verify the error due to the  
    item's status.
```

```
    // As such, it might not be possible to implement a verification step here.
```

```
}
```

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

public void theSystemWithdrawsTheItemHarryPotter() {

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

    assert withdrawnItem == null;

}
```

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

public void georgeRedTriesToBorrowTheItemHarryPotter() {

    try {

        itemService loanItem("Harry Potter", "George Red");

    } catch (Exception e) {

        // Handle exception

    }

}
```

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

public void theSystemDoesntAllowTheLoan() {

    Item notLoanedItem = itemService.findItem("Harry Potter");

    assert notLoanedItem != null && !notLoanedItem.isLoaned();

}
```

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

public void georgeRedsPendingItemsRemain(int pendingItems) {
```

```
        assert borrower.getPendingItems() == pendingItems;
    }
}
```

Please note that the actual implementation of these methods will depend on the specific details of your application. The above code is just a skeleton to guide you on how to structure your step definitions. You would need to replace the `ItemService` and `BorrowerService` with the actual service classes you are using in your application. Similarly, the `Item` and `Borrower` classes would need to be replaced with the actual domain classes you are using.

User

Based on the above, create all the step definitions for the feature : Returning a borrowed item to the library

GitHubCopilot

Sure, let's create the step definitions for the `Returning a borrowed item to the library` feature. We will create a class `ItemReturnSteps` to handle the step definitions.

```
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import librarysystem.domain.Borrower;
import librarysystem.domain.Item;
```

```
import librarysystem.service.BorrowerService;

import librarysystem.service.ItemService;


import java.time.LocalDate;

import java.time.temporal.ChronoUnit;


public class ItemReturnSteps {


    private ItemService itemService = new ItemService();

    private BorrowerService borrowerService = new BorrowerService();

    private Item item;

    private Borrower borrower;


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

    public void georgeRedBorrowedTheItemAnimalKingdomDaysPriorToTodaysDate(int days) {

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

        assert borrower != null;

        item = itemService.findItem("Animal Kingdom");

        assert item != null && item.isLoaned();

        // Assuming the loan date is set when the item is loaned

        assert ChronoUnit.DAYS.between(item.getLoanDate(), LocalDate.now()) == days;

    }
```

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

public void georgeRedHasBeenAssignedMaximumLendingDaysOf(int lendingDays) {

    assert borrower.getLendingDays() == lendingDays;

}
```

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

public void theReturnOfAnimalKingdomIsProcessed() {

    itemService.returnItem("Animal Kingdom");

}
```

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

public void theSystemMarksTheStateOfAnimalKingdomAsAVAILABLE() {

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

    assert returnedItem != null && !returnedItem.isLoaned();

}
```

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

public void georgeRedHasOneLessPendingItem() {

    assert borrower.getPendingItems() == borrower.getPendingItems() - 1;

}
```

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

public void georgeRedDoesNotPayAFine() {
```

```
// This step would require access to the system's response to verify George Red does not pay a fine.
```

```
// As such, it might not be possible to implement a verification step here.
```

```
}
```

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

```
public void theReturnDateOfTheLoanIsSetToTodaysDate() {
```

```
    assert item.getReturnDate().isEqual(LocalDate.now());
```

```
}
```

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

```
public void georgeRedPaysAFineBasedOnTheBorrowerCategory() {
```

```
    // This step would require access to the system's response to verify George Red pays a fine based on the borrower category.
```

```
    // As such, it might not be possible to implement a verification step here.
```

```
}
```

```
}
```

Please note that the actual implementation of these methods will depend on the specific details of your application. The above code is just a skeleton to guide you on how to structure your step definitions. You would need to replace the ``ItemService`` and ``BorrowerService`` with the actual service classes you are using in your application. Similarly, the ``Item`` and ``Borrower`` classes would need to be replaced with the actual domain classes you are using.

User

Based on the above, create all the step definitions for the feature: Delayed Return Notification

GitHubCopilot

Sure, let's create the step definitions for the `Delayed Return Notification` feature. We will create a class `DelayNotificationSteps` to handle the step definitions.

```
import io.cucumber.java.en.Given;

import io.cucumber.java.en.Then;

import io.cucumber.java.en.When;

import librarysystem.domain.Borrower;

import librarysystem.domain.Item;

import librarysystem.service.BorrowerService;

import librarysystem.service.ItemService;

import librarysystem.service.NotificationService;

import java.time.LocalDate;

import java.time.temporal.ChronoUnit;

public class DelayNotificationSteps {

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



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

private NotificationService notificationService = new NotificationService();

private Item item;

private Borrower borrower;
```

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

```
public void georgeRedHasBorrowedTheItemHarryPotter() {

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

    assert borrower != null;

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

    assert item != null && item.isLoaned();

}
```

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

```
public void harryPottersDueDateHasPassed() {

    assert item.getDueDate().isBefore(LocalDate.now());

}
```

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

```
public void georgeRedHasAnEmailAddress() {

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

}
```

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

public void theSystemExecutesTheDelayedReturnNotificationProcess() {

    notificationService.sendDelayedReturnNotification(borrower.getEmail(), "Harry Potter");

}
```

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

public void georgeRedReceivesAnEmailNotificationForTheReturnOfTheItem() {

    // This step would require access to the email server to verify the email was sent.

    // As such, it might not be possible to implement a verification step here.

}
```

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

public void georgeRedDoesNotHaveAnEmailAddress() {

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

}
```

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

public void georgeRedDoesNotReceiveAnEmailNotificationForTheReturnOfTheItem() {

    // This step would require access to the email server to verify the email was not sent.

    // As such, it might not be possible to implement a verification step here.

}

}
```

Please note that the actual implementation of these methods will depend on the specific details of your application. The above code is just a skeleton to guide you on how to structure

your step definitions. You would need to replace the ``ItemService``, ``BorrowerService`` and ``NotificationService`` with the actual service classes you are using in your application. Similarly, the ``Item`` and ``Borrower`` classes would need to be replaced with the actual domain classes you are using.