Στη συνομιλία αυτή, παρέθεσα στο LLM όλα τα features του συστήματος, τα ονόματα, τις ιδιότητες και τις μεθόδους των κλάσεων Domain, καθώς και την αρχιτεκτονική και γενικές γνώσεις του συστήματος. Το LLM δημιούργησε αρκετά καλά αποτελέσματα, αν και χρειάστηκε κάποιες επαναλαμβανόμενες καθοδηγήσεις για να θυμηθεί να χρησιμοποιήσει τα DAOs και να δημιουργήσει όλα τα Step Definitions. Παρά αυτές τις μικρές δυσκολίες, τα αποτελέσματα ήταν παρόμοια με εκείνα της προηγούμενης συνομιλίας αυτής της φάσης και ήταν ικανοποιητικά.

Phase 4 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 code of all the domain classes with their attributes and methods that you 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

I understand your requirements. To provide you with the implementation of the step definitions for your Gherkin scenarios, we will follow an architecture that includes:

1. **Domain Classes**: These will represent the core entities of your library system.

2. **Data Access Objects (DAOs)**: These will handle the database operations such as saving and deleting entities.

3. **Service Classes**: These will contain the business logic and will interact with the DAOs.

Here is a high-level overview of the steps we'll follow:

1. **Review the Gherkin scenarios**: To understand the requirements and the features to be implemented.

2. **Domain Classes**: Define the Java classes representing the core entities with their attributes and methods.

3. **Data Access Objects (DAOs)**: Implement the DAO interfaces and classes for CRUD operations.

4. **Service Classes**: Implement the service classes that contain the business logic.

5. **Step Definitions**: Write the step definitions in Java for the provided Gherkin scenarios.

Please provide the Gherkin scenarios and the code of the domain classes with their attributes and methods that you have already created. This will help me understand the specific features and functionalities we need to implement in the step definitions.

User

The domain classes are:

1)class "Book" with the attributes:

private ISBN isbn,

private String title,

private String publication,

private int publicationyear,

private Publisher publisher,

private Set<Item> items = new HashSet<Item>(),

private Set<Author> authors = new HashSet<Author>(),

and with the methods:

Setters and Getters for every attribute and a constructor,

public Set<Item> getItems(),

public void addItem(Item item),

public void removeItem(Item item),

Set<Item> friendItems(),

public void addAuthor(Author author),

public void removeAuthor(Author author),

Set<Author> friendAuthors()

2)class "Borrower" with the attributes:

private int borrowerNo,

private TelephoneNumber telephone,

private EmailAddress emailAddress,

private Address address,

private BorrowerCategory category,

private Person person = new Person(),

private Set<Loan> loans = new HashSet<Loan>(),

and with the methods:

Setters and Getters for every attribute and a constructor,

public Set<Loan> getLoans(),

Set<Loan> friendLoans(),

public int countPendingItems(),

public boolean canBorrow(),

public LocalDate getLoanDue(LocalDate loanDate),

public Money getDailyFine()

3)class "BorrowerCategory" with the attributes:

private String description,

private int maxLendingItems,

private int maxLendingDays,

private Money dailyFine,

and with the methods:

Setters and Getters for every attribute and a constructor,

public LocalDate getLoanDue(LocalDate loanDate),

public boolean canBorrow(int pendingItems)

4)class "Item" with the attributes:

private int itemNumber = 0,

private Book book,

private ItemState state = ItemState.NEW

and with the methods:

Setters and Getters for every attribute and a constructor,

public Loan borrow(Borrower borrower),

public void available(),

public void withdraw(),

public void lost(),

5)enum class "ItemState" with the Enum values :

NEW,AVAILABLE,LOANED,LOST,WITHDRAWN

6)class "Loan" with the attributes:

private LocalDate loanDate = SystemDate.now(),

private LocalDate returnDate,

private Borrower borrower,

private Item item

and with the methods:

Setters and Getters for every attribute and a constructor,

public LocalDate getDue(),

public boolean isPending(),

public void returnItem(),

public boolean isOverdue(),

public long daysToDue(),

public long getOverdue(),

public Money getFine(),

7)class "Person" with the attributes:

private String firstName,

private String lastName,

and with the methods:

Setters and Getters for every attribute and a constructor

In the next message I will provide you with the features/scenarios

ChatGPT

Great! Thanks for providing the details of the domain classes for your library system application. These details help set the foundation for understanding the structure and relationships within your application. I now have a good grasp of the entities involved, such as books, borrowers, loans, and the state of items, as well as their interactions through various methods.

Before moving ahead, it will be helpful to see the Gherkin scenarios and any additional specifics about the features you'd like to implement. This will guide the creation of the corresponding step definitions, ensuring they align with the intended behaviors described in your scenarios.

Once you provide the scenarios, we can proceed with outlining the DAOs and service classes necessary to support the step definitions within the specified architecture. If there are any specific rules or business logic details that need to be considered (e.g., borrowing restrictions based on category, fine calculations), please include those as well.

Looking forward to your next message with the scenarios!

User

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

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

Thank you for providing the detailed Gherkin scenarios for your library system application. These scenarios cover a variety of important features such as borrower registration, loaning items, returning items, and handling notifications for delayed returns.

To implement these scenarios, we'll need to develop Java step definitions that interact with the underlying service and data layers. This will involve writing step definitions that utilize service classes to manipulate domain objects and DAOs to interact with the database.

Here's a plan for implementing the step definitions:

Prerequisites

1. **Set up the testing framework**: Ensure that Cucumber and its dependencies are added to your project.

2. **Prepare the data layer**: Implement DAOs for accessing and manipulating `Borrower`, `Item`, `Loan`, etc.

3. **Service layer implementation**: Create service classes that encapsulate business logic described in your scenarios (e.g., registering a borrower, loaning items, etc.).

Step Definition Examples

I'll provide a sample implementation for a few scenarios:

1. Registering a new borrower

```java
@Given("^{string} is not registered as a borrower")
public void george_is_not_registered_as_a_borrower(String name) {
    assertThat(borrowerService.findByName(name)).isNull();
}

@When("^{string} gets registered in the system with a unique borrower number and his details")
public void george_gets_registered_in_the_system_with_details(String name) {
    Borrower newBorrower = new Borrower();
    newBorrower.setName(name);
    newBorrower.setBorrowerNo(borrowerService.generateUniqueNumber());
    newBorrower.setEmailAddress(new EmailAddress("george@example.com"));
    newBorrower.setAddress(new Address("Some street", "City", "Zip"));
    borrowerService.registerBorrower(newBorrower);
}

@Then("the system successfully stores the borrower's details")
public void the_system_successfully_stores_the_borrowers_details() {
    Borrower registered = borrowerService.findByName("George Red");
    assertThat(registered).isNotNull();

    assertThat(registered.getName()).isEqualTo("George Red");

}
```

2. Handling unreturned items of books

```java
@Given("^{string} has pending items")

public void george_has_pending_items(String name) {

    Borrower borrower = borrowerService.findByName(name);

    Item item = new Item(1, new Book("Harry Potter"), ItemState.LOANED);

    Loan loan = new Loan(LocalDate.now().minusDays(5), null, borrower, item);

    borrower.getLoans().add(loan);

}


@When("the system attempts to delete {string}'s account")

public void the_system_attempts_to_delete_account(String name) {

    try {

        borrowerService.deleteBorrower(name);

    } catch (Exception e) {

        lastException = e;
    }
}

@Then("the system does not remove {string}'s details")
public void the_system_does_not_remove_details(String name) {
    assertThat(borrowerService.findByName(name)).isNotNull();
}

@And("the system informs about the pending items")
public void the_system_informs_about_the_pending_items() {
    assertThat(lastException.getMessage()).contains("pending items");
}
```

Next Steps

1. **Implement DAOs**: Create DAO classes for all entities (`BorrowerDao`, `ItemDao`, etc.) using an ORM framework like Hibernate or just plain JDBC.

2. **Implement Service Classes**: Each service class should handle the business logic for each domain, like `BorrowerService`, `LoanService`, etc.

3. **Complete Step Definitions**: Based on the provided examples, write step definitions for all scenarios using a similar pattern.

If you need help with specific scenarios, DAO implementations, or service logic, feel free to ask!

User

Implement all the step definitions of the feature: Borrower Handling by the system

ChatGPT

To implement the step definitions for the "Borrower Handling by the System" feature in your Java-based library system, we'll first outline the necessary service methods and then write the step definitions. For the purposes of this example, we'll assume that you have services like `BorrowerService` which provide functionality such as registering, updating, and deleting borrowers, and checking if a borrower is registered.

Service Class Interfaces

Here are some methods that the `BorrowerService` might need to support these scenarios:

- **registerBorrower(Borrower borrower)**: Registers a new borrower.

- **findBorrowerByName(String name)**: Finds a borrower by name.

- **updateBorrower(Borrower borrower)**: Updates borrower's details.

- **deleteBorrower(String name)**: Deletes a borrower by name.

- **isBorrowerRegistered(String name)**: Checks if the borrower is registered.

Step Definitions for Borrower Handling

```java
import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import static org.junit.Assert.*;

public class BorrowerSteps {
    private BorrowerService borrowerService = new BorrowerService();
    private Borrower borrower;
    private String lastOperationResult;

    @Given("{string} is not registered as a borrower")
    public void borrower_is_not_registered(String name) {
        assertNull(borrowerService.findBorrowerByName(name));
    }

    @When("{string} gets registered in the system with a unique borrower number and his details")
    public void register_borrower(String name) {
        if (!borrowerService.isBorrowerRegistered(name)) {
            borrower = new Borrower();
            borrower.setName(name);
            borrower.setBorrowerNo(borrowerService.generateUniqueBorrowerNumber());
            borrower.setEmail(new EmailAddress("george@example.com")); // Assuming EmailAddress class exists
            borrower.setAddress(new Address("123 Main St", "Anytown", "12345")); // Assuming Address class exists
            borrowerService.registerBorrower(borrower);
            lastOperationResult = "Success";
        } else {
            lastOperationResult = "Already Registered";
        }
    }

    @Then("the system successfully stores the borrower's details")
    public void borrower_details_stored() {
        assertEquals("Success", lastOperationResult);
        assertNotNull(borrowerService.findBorrowerByName(borrower.getName()));
    }

    @Given("{string} is registered as a borrower")
    public void borrower_is_registered(String name) {
        borrower = borrowerService.findBorrowerByName(name);
        assertNotNull(borrower);
    }
```

```
}



@When("{string} updates his borrowing details")
public void update_borrower_details(String name) {
    Borrower existingBorrower = borrowerService.findBorrowerByName(name);
    if (existingBorrower != null) {
        existingBorrower.setEmail(new EmailAddress("updated@example.com"));
        borrowerService.updateBorrower(existingBorrower);
        lastOperationResult = "Updated";
    } else {
        lastOperationResult = "Not Registered";
    }
}


@Then("the system saves the changes")
public void system_saves_changes() {
    assertEquals("Updated", lastOperationResult);
}


@When("{string} tries to update his borrowing details")
public void tries_to_update_borrowing_details(String name) {
    update_borrower_details(name);
}
```

```
@Then("the system displays an error message indicating that {string} does not exist")
public void system_displays_error_message(String name) {
    assertEquals("Not Registered", lastOperationResult);
}

@When("the system deletes {string}'s account")
public void delete_borrower_account(String name) {
    if (borrowerService.isBorrowerRegistered(name)) {
        borrowerService.deleteBorrower(name);
        lastOperationResult = "Deleted";
    } else {
        lastOperationResult = "Not Registered";
    }
}

@Then("the system removes {string}'s details")
public void system_removes_details(String name) {
    assertEquals("Deleted", lastOperationResult);
    assertNull(borrowerService.findBorrowerByName(name));
}
```

```
    @Then("the system informs that the borrower does not exist")
    public void system_informs_borrower_does_not_exist() {
        assertEquals("Not Registered", lastOperationResult);
    }
}
```

Explanation

1. **Assertions**: We use `assertNull`, `assertNotNull`, and `assertEquals` to check conditions expected by each scenario.

2. **Service Interaction**: These steps assume that methods like `registerBorrower`, `findBorrowerByName`, `updateBorrower`, and `deleteBorrower` interact with a database or a simulated data structure.

3. **Results Handling**: A variable `lastOperationResult` is used to record the outcome of operations to validate against expected outcomes.

Make sure your service class implementations handle the business logic correctly, including managing IDs, checking for existing records, and updating or deleting data appropriately. This code assumes such functionalities are implemented in the `BorrowerService` class.

User

You are missing some step definitions, give them all

ChatGPT

Apologies for the oversight. Let's complete all the step definitions required for the "Borrower Handling by the System" feature. We'll cover each scenario outlined in your Gherkin specifications, ensuring comprehensive coverage for registering, updating, and deleting borrowers, along with error handling when a borrower does not exist or has unreturned items.

Complete Step Definitions

Here is the Java implementation for the remaining scenarios, including those for handling unreturned items, which require interacting with the `Loan` domain:

```java
import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import static org.junit.Assert.*;

public class BorrowerSteps {
    private BorrowerService borrowerService = new BorrowerService();
    private Borrower borrower;
    private String lastOperationResult;
    private Exception lastException;

    // Initial setup and registration scenarios
    @Given("{string} is not registered as a borrower")
    public void borrower_is_not_registered(String name) {
        assertNull(borrowerService.findBorrowerByName(name));
    }


    @When("{string} gets registered in the system with a unique borrower number and his details")
    public void register_borrower(String name) {
        if (!borrowerService.isBorrowerRegistered(name)) {
            borrower = new Borrower();
            borrower.setName(name);
            borrower.setBorrowerNo(borrowerService.generateUniqueBorrowerNumber());
            borrower.setEmail(new EmailAddress("george@example.com")); // Assuming EmailAddress class exists
            borrower.setAddress(new Address("123 Main St", "Anytown", "12345")); // Assuming Address class exists
            borrowerService.registerBorrower(borrower);
            lastOperationResult = "Success";
        } else {
            lastOperationResult = "Already Registered";
        }
    }
    @Then("the system successfully stores the borrower's details")
    public void borrower_details_stored() {
        assertEquals("Success", lastOperationResult);
        assertNotNull(borrowerService.findBorrowerByName(borrower.getName()));
    }

    // Handling scenarios where the borrower is already registered
    @Given("{string} is registered as a borrower")
    public void borrower_is_registered(String name) {
        assertNotNull(borrowerService.findBorrowerByName(name));
    }

    @When("the system attempts to register {string} with a unique borrower number and his details")
    public void attempt_to_register_borrower(String name) {
        register_borrower(name); // Reuse registration logic to check if re-registration happens
    }

    @Then("the system informs that the user already exists")
    public void inform_user_already_exists() {
        assertEquals("Already Registered", lastOperationResult);
    }
```

```
// Updating borrower details

@When("{string} updates his borrowing details")

public void update_borrower_details(String name) {

    Borrower existingBorrower = borrowerService.findBorrowerByName(name);

    if (existingBorrower != null) {

        existingBorrower.setEmail(new EmailAddress("updated@example.com"));

        borrowerService.updateBorrower(existingBorrower);

        lastOperationResult = "Updated";

    } else {

        lastOperationResult = "Not Registered";

    }

}


@Then("the system saves the changes")

public void system_saves_changes() {

    assertEquals("Updated", lastOperationResult);

}


// Handling scenarios where the borrower tries to update details but is not registered

@When("{string} tries to update his borrowing details")

public void tries_to_update_borrowing_details(String name) {
    update_borrower_details(name);

}



@Then("the system displays an error message indicating that {string} does not exist")

public void system_displays_error_message(String name) {

    assertEquals("Not Registered", lastOperationResult);

}



// Deleting a borrower

@When("the system deletes {string}'s account")

public void delete_borrower_account(String name) {

    if (borrowerService.isBorrowerRegistered(name)) {

        borrowerService.deleteBorrower(name);

        lastOperationResult = "Deleted";

    } else {

        lastOperationResult = "Not Registered";

    }

}



@Then("the system removes {string}'s details")

public void system_removes_details(String name) {

    assertEquals("Deleted", lastOperationResult);
    assertNull(borrowerService.findBorrowerByName(name));
}

// Attempting to delete an unregistered borrower
@Then("the system informs that the borrower does not exist")
public void system_informs_borrower_does_not_exist() {
    assertEquals("Not Registered", lastOperationResult);
}

// Handling unreturned items when deleting a borrower
@Given("{string} has pending items")
public void borrower_has_pending_items(String name) {
    Borrower borrower = borrowerService.findBorrowerByName(name);
    if (borrower != null) {
        // Mock a situation where the borrower has unreturned items
        Loan loan = new Loan(LocalDate.now().minusDays(5), null, borrower, new Item());
        borrower.getLoans().add(loan);
    }
}

@When("the system attempts to delete {string}'s account")
public void attempt_to_delete_borrower_with_pending_items(String name) {
        try {
            delete_borrower_account(name); // Reuse deletion logic
        } catch (Exception e) {
            lastException = e;
            lastOperationResult = "Has Pending Items";
        }
    }

    @Then("the system does not remove {string}'s details")
    public void system_does_not_remove_details_due_to_pending_items(String name) {
        assertEquals("Has Pending Items", lastOperationResult);
    }

    @And("the system informs about the pending items")
    public void system_informs_about_pending_items() {
        assertTrue(lastException.getMessage().contains("pending items"));
    }
}
```

This implementation covers all scenarios from the "Borrower Handling by the System" feature, ensuring that you have a robust set of step definitions to test the various states and behaviors of your borrower management functionality.

User

What is the code for the borrower service you are using?

ChatGPT

To make the step definitions operational, we'll need a `BorrowerService` class that handles the logic for borrower registration, updates, deletions, and queries. Below, I'll provide an example of how you might implement this service in Java, using a simple in-memory storage mechanism with a `HashMap`. This example is for demonstration purposes and assumes that you are not yet integrating with a real database.

Example `BorrowerService` Java Implementation

Here's a basic implementation of `BorrowerService`, which provides the functionality needed for your step definitions:

```java
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.atomic.AtomicInteger;

public class BorrowerService {
```

```
private Map<String, Borrower> borrowers = new HashMap<>();

private AtomicInteger borrowerIdGenerator = new AtomicInteger(0);


public boolean isBorrowerRegistered(String name) {

    return borrowers.containsKey(name);

}


public Borrower findBorrowerByName(String name) {

    return borrowers.get(name);

}


public void registerBorrower(Borrower borrower) {

    if (isBorrowerRegistered(borrower.getName())) {

        throw new IllegalStateException("Borrower already registered.");

    }

    borrower.setBorrowerNo(borrowerIdGenerator.incrementAndGet());

    borrowers.put(borrower.getName(), borrower);

}


public void updateBorrower(Borrower borrower) {

    if (!isBorrowerRegistered(borrower.getName())) {

        throw new IllegalStateException("Borrower not registered.");
        }
        borrowers.put(borrower.getName(), borrower);
    }

    public void deleteBorrower(String name) {
        if (!isBorrowerRegistered(name)) {
            throw new IllegalStateException("Borrower not registered.");
        }
        borrowers.remove(name);
    }

    public int generateUniqueBorrowerNumber() {
        return borrowerIdGenerator.incrementAndGet();
    }
}
```

Explanation of the `BorrowerService` Class

- **Data Storage**: The `borrowers` `HashMap` stores `Borrower` objects keyed by the borrower's name. This simple key-value storage simulates a database for demonstration purposes.

- **Registration**: `registerBorrower` adds a new `Borrower` to the map if they aren't already registered, else it throws an exception.

- **Update**: `updateBorrower` updates the borrower details if they are registered; otherwise, it throws an exception.

- **Deletion**: `deleteBorrower` removes a borrower from the map if they are registered; otherwise, it throws an exception.

- **Unique ID Generation**: `borrowerIdGenerator` is an `AtomicInteger` used to generate unique IDs, simulating database auto-increment functionality.

Integration Considerations

This service class uses in-memory storage, which is suitable for demonstration and testing but not for production. For a real-world application, you would likely use a database with an ORM (Object-Relational Mapping) framework such as Hibernate. This would involve configuring entity classes with annotations and using a DAO (Data Access Object) pattern to abstract the database operations.

To transition this example to a production-ready implementation, you would:

- Replace the in-memory map with a database-backed DAO.

- Handle transactions, error management, and concurrency appropriately.

- Add more robust error checking and possibly logging for operations.

This example provides a solid foundation for understanding how to implement the required services for your Cucumber step definitions. If you need further details on integrating with a database or expanding the functionality, feel free to ask!

User

I instructed you to use DAOs, use them

ChatGPT

You're right; the initial implementation I provided was simplified and didn't incorporate Data Access Objects (DAOs), which are crucial for separating the persistence logic from the business logic in a well-structured application. Let's refactor the `BorrowerService` to utilize a DAO layer, which will manage the database interactions for `Borrower` entities. Here, I'll assume a basic in-memory store but structured in a DAO pattern that can easily be adapted to use an actual database like SQL or NoSQL databases.

DAO Interface

First, let's define the DAO interface for the `Borrower` entity:

```java
public interface BorrowerDao {
    Borrower findByName(String name);
    void save(Borrower borrower);
    void update(Borrower borrower);
    void deleteByName(String name);
    boolean existsByName(String name);
}
```

In-Memory DAO Implementation

Here’s an implementation of the `BorrowerDao` that uses an in-memory `HashMap` to store borrower data:

```java
import java.util.HashMap;
import java.util.Map;

public class BorrowerDaoImpl implements BorrowerDao {
    private Map<String, Borrower> borrowerMap = new HashMap<>();

    @Override
    public Borrower findByName(String name) {
        return borrowerMap.get(name);
    }

    @Override
```

```
public void save(Borrower borrower) {

    borrowerMap.put(borrower.getName(), borrower);

}


@Override

public void update(Borrower borrower) {

    if (existsByName(borrower.getName())) {

        borrowerMap.put(borrower.getName(), borrower);

    } else {

        throw new IllegalStateException("Cannot update non-existing borrower.");

    }

}


@Override

public void deleteByName(String name) {

    if (!existsByName(name)) {

        throw new IllegalStateException("Cannot delete non-existing borrower.");

    }

    borrowerMap.remove(name);

}


@Override
```

```
    public boolean existsByName(String name) {
        return borrowerMap.containsKey(name);
    }
}
```

Service Class Using DAO

With the DAO implemented, we can now refactor the `BorrowerService` to use this DAO:

```java
import java.util.concurrent.atomic.AtomicInteger;

public class BorrowerService {
    private BorrowerDao borrowerDao = new BorrowerDaoImpl();
    private AtomicInteger borrowerIdGenerator = new AtomicInteger(0);

    public boolean isBorrowerRegistered(String name) {
        return borrowerDao.existsByName(name);
    }

    public Borrower findBorrowerByName(String name) {
```

```
    return borrowerDao.findByName(name);
}

public void registerBorrower(Borrower borrower) {
    if (isBorrowerRegistered(borrower.getName())) {
        throw new IllegalStateException("Borrower already registered.");
    }
    borrower.setBorrowerNo(borrowerIdGenerator.incrementAndGet());
    borrowerDao.save(borrower);
}

public void updateBorrower(Borrower borrower) {
    borrowerDao.update(borrower);
}

public void deleteBorrower(String name) {
    borrowerDao.deleteByName(name);
}

public int generateUniqueBorrowerNumber() {
    return borrowerIdGenerator.incrementAndGet();
}
```

```
}
```

Integration and Usage

With the DAO pattern implemented:

- The `BorrowerService` interacts with the `BorrowerDaoImpl` to handle all data storage operations, cleanly separating business logic from data access logic.
- Modifications to switch from an in-memory store to a real database would now primarily involve changes within the `BorrowerDaoImpl` class, without affecting the `BorrowerService`.

This structure adheres to good software design principles, promoting modularity and making the system easier to maintain and test. You can easily swap out the `BorrowerDaoImpl` with another DAO that implements the same interface but handles interactions with a real database, adjusting for production needs without modifying the service logic.

User

Now, based on the above, give me the step definitions of the feature: Loaning Items

ChatGPT

To write step definitions for the "Loaning Items" feature of your library system, we will assume that there are appropriate service classes and DAOs in place for handling the loaning process. Specifically, these might include services and DAOs for managing items, loans, and borrower details.

Here are step definitions for the scenarios provided in your "Loaning Items" feature. These definitions will rely on services for `ItemService`, `BorrowerService`, and `LoanService`, which must be capable of handling the operations required by each step.

Step Definitions for the "Loaning Items" Feature

Let's write the Java Cucumber step definitions:

```java
import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import static org.junit.Assert.*;

public class LoaningItemsSteps {
    private BorrowerService borrowerService = new BorrowerService();  // Assumes services are already defined
    private ItemService itemService = new ItemService();
    private LoanService loanService = new LoanService();
    private Item item;
    private Borrower borrower;
    private Loan loan;
private String operationResult;


@Given("the library has the item {string} available")

public void the_library_has_the_item_available(String itemName) {

    item = itemService.findItemByName(itemName);

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

}


@Given("{string} is a registered borrower")

public void registered_borrower(String name) {

    borrower = borrowerService.findBorrowerByName(name);

    assertNotNull(borrower);

}


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

public void borrower_has_pending_items(String name, int pendingItems) {

    borrower = borrowerService.findBorrowerByName(name);

    assertEquals(pendingItems, loanService.countPendingItems(borrower));

}


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

public void borrower_has_maximum_lending_limit(String name, int limit) {
        borrower = borrowerService.findBorrowerByName(name);
        assertEquals(limit, borrower.getMaxLendingItems());
    }

    @When("{string} borrows the item {string}")
    public void borrower_borrows_the_item(String borrowerName, String itemName) {
        borrower = borrowerService.findBorrowerByName(borrowerName);
        item = itemService.findItemByName(itemName);
        try {
            loan = loanService.loanItemToBorrower(item, borrower);
            operationResult = "Loan Successful";
        } catch (Exception e) {
            operationResult = e.getMessage();
        }
    }

    @Then("the system successfully loans the item {string} to {string} with a due date set")
    public void system_successfully_loans_item(String itemName, String borrowerName) {
        assertEquals("Loan Successful", operationResult);
        assertNotNull(loan);
        assertEquals(LocalDate.now().plusDays(borrower.getCategory().getMaxLendingDays()), loan.getDueDate());
```

```
}


@And("{string}'s pending items increase to {int}")
public void borrowers_pending_items_increase(String name, int totalPendingItems) {
    assertEquals(totalPendingItems, loanService.countPendingItems(borrower));
}


@Given("the library has the items {string} and {string} available")
public void library_has_two_items_available(String firstItemName, String secondItemName) {
    Item firstItem = itemService.findItemByName(firstItemName);
    Item secondItem = itemService.findItemByName(secondItemName);
    assertTrue(firstItem != null && firstItem.getState() == ItemState.AVAILABLE);
    assertTrue(secondItem != null && secondItem.getState() == ItemState.AVAILABLE);
}


@When("{string} tries to borrow both items")
public void borrower_tries_to_borrow_both_items(String name) {
    Borrower borrower = borrowerService.findBorrowerByName(name);
    Item firstItem = itemService.findItemByName("Harry Potter");
    Item secondItem = itemService.findItemByName("Moby Dick");
    try {
        loanService.loanItemToBorrower(firstItem, borrower);
        loanService.loanItemToBorrower(secondItem, borrower);

    } catch (Exception e) {

        operationResult = e.getMessage();

    }

}


@And("the system does not loan {string} to {string} due to the lending limit reached")

public void system_does_not_loan_due_to_limit(String itemName, String borrowerName) {

    assertFalse(loanService.isLoanSuccessful(itemName, borrowerName));

}


@Given("the item {string} is in the library but not in the system")

public void item_not_in_system(String itemName) {

    assertNull(itemService.findItemByName(itemName));

}


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

public void system_returns_error_due_to_status() {

    assertTrue(operationResult.contains("Error"));

}


@And("the system withdraws the item {string}")
```

```
    public void system_withdraws_item(String itemName) {

        assertEquals(ItemState.WITHDRAWN, itemService.findItemByName(itemName).getState());

    }

}
```

Explanation of Step Definitions

1. **Assertions**: These use JUnit assertions to validate conditions, such as the availability of items and the registration status of borrowers.

2. **Service Interactions**: Interactions with `BorrowerService`, `ItemService`, and `LoanService` assume methods like `findItemByName`, `loanItemToBorrower`, and `countPendingItems` are properly implemented in the respective services.

3. **Error Handling**: Error messages are captured and checked against expected outcomes, ensuring that business rules (like lending limits) are enforced.

These steps assume that you have placeholder methods and proper error handling in your service classes. If you need implementations for these service methods or further refinement of these definitions, feel free to ask!

User

Give me the item service code

ChatGPT

To implement the `ItemService` class for your library system, we'll create a service that handles operations related to library items, such as checking item availability, updating item states, and finding items by their name. This service will interact with an `ItemDao` that handles the actual data storage and retrieval.

Below, I'll provide a sample implementation of both the `ItemService` and the corresponding `ItemDao`. This example will use a simple in-memory store for demonstration, but you can adapt it to use a database if needed.

ItemDao Interface

First, let's define the `ItemDao` interface:

```java
public interface ItemDao {
    Item findByName(String name);
    void save(Item item);
    void update(Item item);
    void deleteByName(String name);
    boolean existsByName(String name);
}
```

In-Memory ItemDao Implementation

Now, we'll implement this DAO using an in-memory `HashMap`:

```java
import java.util.HashMap;
import java.util.Map;

public class ItemDaoImpl implements ItemDao {
    private Map<String, Item> items = new HashMap<>();

    @Override
    public Item findByName(String name) {
        return items.get(name);
    }

    @Override
    public void save(Item item) {
        items.put(item.getBook().getTitle(), item);
    }

    @Override
    public void update(Item item) {
        if (existsByName(item.getBook().getTitle())) {
            items.put(item.getBook().getTitle(), item);
        } else {
            throw new IllegalStateException("Item does not exist and thus cannot be updated.");
        }
    }

    @Override
    public void deleteByName(String name) {
        if (!existsByName(name)) {
            throw new IllegalStateException("Item not found and thus cannot be deleted.");
        }
        items.remove(name);
    }

    @Override
    public boolean existsByName(String name) {
        return items.containsKey(name);
    }
}
```

ItemService Class

Next, the `ItemService` class uses `ItemDao` to manage `Item` entities:

```java
public class ItemService {
    private ItemDao itemDao = new ItemDaoImpl(); // Replace with a real implementation if using a database

    public Item findItemByName(String name) {
        return itemDao.findByName(name);
    }

    public void addItem(Item item) {
        if (!itemDao.existsByName(item.getBook().getTitle())) {
            itemDao.save(item);
        } else {
            throw new IllegalArgumentException("Item already exists.");
        }
    }
```

```
public void updateItem(Item item) {
    itemDao.update(item);
}


public void deleteItem(String name) {
    itemDao.deleteByName(name);
}


public void changeItemState(String itemName, ItemState state) {
    Item item = findItemByName(itemName);
    if (item != null) {
        item.setState(state);
        itemDao.update(item);
    } else {
        throw new IllegalArgumentException("Item not found.");
    }
}


public boolean isLoanSuccessful(String itemName, String borrowerName) {
    Item item = findItemByName(itemName);
    if (item != null && item.getState() == ItemState.LOANED) {
        // Assuming logic to verify borrower as well
```

```
            return true;
        }
        return false;
    }
}
```

Usage and Integration

This `ItemService`:

- Manages `Item` operations, centralizing logic for adding, updating, deleting, and changing the state of items.
- Utilizes `ItemDao` for data interaction, which allows for easier transition to a database-backed storage by merely changing the DAO implementation.
- Provides methods to support the step definitions for item-related scenarios in your Gherkin specifications.

Adaptation for Production

For a production environment, you would:

- Implement the DAO with a real database, using JDBC, JPA, or an ORM framework like Hibernate.
- Ensure transaction management, particularly for methods that change item state or handle loans.

- Add error handling and possibly logging for better operation visibility and debugging.

This setup lays the groundwork for a modular and easily maintainable code structure in your application. If you need further assistance or more specific implementations, feel free to ask!

User

Now, based on the above, create the step definitions for the feature: Returning a borrowed item in the library

ChatGPT

For the feature "Returning a Borrowed Item to the Library," we will create step definitions that interact with your library system, focusing on the scenarios described in your Gherkin document. These step definitions will use methods likely found in your `ItemService` and `LoanService` to manage the return of items, update their status, and handle any fines due for late returns.

Assumptions

- **LoanService** provides methods like `returnItem(Item item, Borrower borrower)` which handles the actual return logic including updating the loan status, setting the return date, and calculating any fines.

- **ItemService** may need methods to update the state of an item, such as marking it as available.

- **BorrowerService** might be used to update borrower details after returning an item, like decreasing the count of pending items.

Step Definitions for Returning a Borrowed Item

Here's the Java Cucumber step definitions for the feature "Returning a Borrowed Item":

```java
import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import static org.junit.Assert.*;

public class ReturnItemSteps {
    private ItemService itemService = new ItemService();
    private LoanService loanService = new LoanService();
    private BorrowerService borrowerService = new BorrowerService();
    private Item item;
    private Borrower borrower;
    private Loan loan;
    private boolean isLate;
    private Money fine;

    @Given("{string} borrowed the item {string} {int} days prior to today's date")
    public void borrowed_the_item_days_ago(String borrowerName, String itemName, int daysAgo) {
        borrower = borrowerService.findBorrowerByName(borrowerName);
        item = itemService.findItemByName(itemName);

        LocalDate loanDate = LocalDate.now().minusDays(daysAgo);

        loan = new Loan(loanDate, null, borrower, item);

        loanService.createLoan(loan); // Assume this method sets up the loan

    }


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

    public void assigned_maximum_lending_days(String borrowerName, int maxDays) {

        borrower = borrowerService.findBorrowerByName(borrowerName);

        borrower.setCategory(new BorrowerCategory(maxDays)); // Assumes a method to set this

    }


    @When("the return of {string} is processed")

    public void the_return_of_item_is_processed(String itemName) {

        isLate = loanService.isLoanOverdue(loan);

        fine = loanService.returnItem(item, borrower);

    }


    @Then("the system marks the state of {string} as AVAILABLE")

    public void system_marks_item_as_available(String itemName) {

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

    }
```

```
    @And("{string} has one less pending item")
    public void borrower_has_one_less_pending_item(String borrowerName) {
        assertEquals(borrower.getLoans().size() - 1, loanService.countPendingItems(borrower));
    }

    @And("{string} does not pay a fine")
    public void borrower_does_not_pay_a_fine(String borrowerName) {
        assertEquals(new Money(0), fine); // Assuming Money class has appropriate equals method
    }

    @And("the return date of the loan is set to today's date")
    public void return_date_set_to_today() {
        assertEquals(LocalDate.now(), loan.getReturnDate());
    }

    @Then("{string} pays a fine based on the borrower category")
    public void borrower_pays_a_fine_based_on_category(String borrowerName) {
        assertTrue(fine.getAmount() > 0);
    }
}
```

Explanation of Step Definitions

- **Setup Steps**: Initialize the state for a borrower who has borrowed an item, including the configuration for the maximum lending days.

- **Return Processing**: When processing the return, the system needs to check if the loan is overdue and then execute the return logic which may involve calculating a fine.

- **Assertions**: Verify that the item state is updated to AVAILABLE, the borrower's pending items count is decremented, and any fines are handled as expected.

Additional Implementation Notes

- **LoanService and BorrowerService methods**: These steps assume that your services can calculate fines, determine if a loan is overdue, and manage loan and borrower state updates. If these methods are not yet implemented, they will need to be developed.

- **Money and BorrowerCategory classes**: These should be defined to handle monetary values and category-specific rules like lending limits and fines.

This setup aligns with the expected library system functionalities, promoting a clear, testable approach to managing book returns. If further details or adjustments are needed, feel free to ask for additional help!

User

Based on the above, give me all the step definitions of the feature: Delayed Notification

ChatGPT

For the feature "Delayed Return Notification," we need step definitions that effectively describe the process of notifying borrowers about overdue items. These step definitions will rely on services such as `LoanService` for handling overdue loans and possibly an `EmailService` or a similar notification service for sending emails.

Assumptions

- **LoanService** provides methods like `isLoanOverdue(Loan loan)` to check if a loan is overdue and `findOverdueLoans()` to fetch all overdue loans.
- **EmailService** or similar is used to handle the sending of email notifications.

Step Definitions for Delayed Return Notification

Let's outline the step definitions for notifying borrowers when an item has not been returned by the due date:

```java
import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import static org.junit.Assert.*;

public class DelayedNotificationSteps {
    private BorrowerService borrowerService = new BorrowerService();
    private ItemService itemService = new ItemService();
    private LoanService loanService = new LoanService();
    private EmailService emailService = new EmailService();
    private Borrower borrower;
    private Item item;
    private Loan loan;
    private boolean emailSent;

    @Given("{string} has borrowed the item {string}")
    public void borrower_has_borrowed_item(String borrowerName, String itemName) {
        borrower = borrowerService.findBorrowerByName(borrowerName);
        item = itemService.findItemByName(itemName);
        loan = new Loan(LocalDate.now().minusWeeks(2), null, borrower, item); // Assume item was borrowed 2 weeks ago
        loanService.createLoan(loan);
    }

    @Given("{string}'s due date has passed")
    public void items_due_date_has_passed(String itemName) {
        assertTrue("The loan should be overdue", loanService.isLoanOverdue(loan));
    }

    @Given("{string} has an email address")
    public void borrower_has_an_email_address(String borrowerName) {
        borrower.setEmail(new EmailAddress("borrower@example.com"));
    }

    @When("the system executes the delayed return notification process")
    public void execute_delayed_return_notification_process() {
        if (loanService.isLoanOverdue(loan) && borrower.getEmail() != null) {
            emailSent = emailService.sendEmail(borrower.getEmail(), "Your loan is overdue", "Please return the item as soon as possible.");
        } else {
            emailSent = false;
        }
    }

    @Then("{string} receives an email notification for the return of the item")
    public void borrower_receives_email_notification(String borrowerName) {
        assertTrue("The email should have been sent", emailSent);
    }

    @Given("{string} does not have an email address")
    public void borrower_does_not_have_an_email_address(String borrowerName) {
        borrower.setEmail(null);
    }


    @Then("{string} does not receive an email notification for the return of the item")
    public void borrower_does_not_receive_email_notification(String borrowerName) {
        assertFalse("No email should have been sent", emailSent);
    }
}
```

Explanation of Step Definitions

- **Loan Setup**: The steps set up the scenario where a borrower has borrowed an item that is now overdue.

- **Notification Conditions**: Before executing the notification process, conditions such as whether the loan is overdue and if the borrower has an email are checked.

- **Notification Execution**: The `EmailService.sendEmail()` method is called to send notifications. This method would ideally handle the actual email sending logic and return true if the email was sent successfully.

- **Assertions**: These check if the email was sent or not, depending on the scenario requirements.

EmailService

This example assumes an `EmailService` class that handles email notifications. Here is a basic structure for such a service:

```java
public class EmailService {
    public boolean sendEmail(EmailAddress emailAddress, String subject, String message) {
        // Simulate sending an email
        System.out.println("Sending email to: " + emailAddress);
        System.out.println("Subject: " + subject);
        System.out.println("Message: " + message);
        return true; // Assume the email is always successfully sent
    }
}
```

Integration and Usage

This setup provides a functional and testable approach to managing overdue notifications in your library system. The actual implementation of `EmailService` would depend on your specific backend technology stack (e.g., using SMTP servers or third-party email services). These step definitions can be directly integrated into your Cucumber test suite to automate the verification of overdue notification functionalities. If you need further help with implementing these services or expanding the testing scenarios, feel free to ask!

User

What is the code for the loan service

ChatGPT

For implementing the `LoanService` in your library system, we need a service that manages loans, including creating loans, checking for overdue loans, returning items, and calculating fines. This service will interact with a `LoanDao` for persisting loan data and potentially a `BorrowerDao` and `ItemDao` for updating related entities.

LoanDao Interface

First, let's define the `LoanDao` interface, which provides basic CRUD operations plus some specific functionalities:

```java
import java.util.List;

public interface LoanDao {
    Loan findById(int id);
    void save(Loan loan);
    void update(Loan loan);
    void delete(int id);
    List<Loan> findLoansByBorrower(Borrower borrower);
    List<Loan> findOverdueLoans();
```

}
~~~

In-Memory LoanDao Implementation

Here’s an example of an in-memory `LoanDaoImpl` using a `HashMap`:

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class LoanDaoImpl implements LoanDao {
    private Map<Integer, Loan> loans = new HashMap<>();
    private int currentId = 1;

    @Override
    public Loan findById(int id) {
        return loans.get(id);
    }
~~~

```
@Override

public void save(Loan loan) {

    loan.setId(currentId);

    loans.put(currentId++, loan);

}


@Override

public void update(Loan loan) {

    loans.put(loan.getId(), loan);

}


@Override

public void delete(int id) {

    loans.remove(id);

}


@Override

public List<Loan> findLoansByBorrower(Borrower borrower) {

    List<Loan> result = new ArrayList<>();

    for (Loan loan : loans.values()) {

        if (loan.getBorrower().equals(borrower)) {

            result.add(loan);
            }
        }
        return result;
    }

    @Override
    public List<Loan> findOverdueLoans() {
        List<Loan> overdueLoans = new ArrayList<>();
        for (Loan loan : loans.values()) {
            if (loan.isOverdue()) {
                overdueLoans.add(loan);
            }
        }
        return overdueLoans;
    }
}
```

LoanService Class

The `LoanService` uses the `LoanDao` to manage loan-related operations. Here’s an implementation:

```java
public class LoanService {
    private LoanDao loanDao = new LoanDaoImpl();

    public Loan createLoan(LocalDate loanDate, Borrower borrower, Item item) {
        Loan loan = new Loan();
        loan.setLoanDate(loanDate);
        loan.setBorrower(borrower);
        loan.setItem(item);
        loan.setDueDate(loanDate.plusDays(borrower.getCategory().getMaxLendingDays()));
        loanDao.save(loan);
        return loan;
    }

    public boolean isLoanOverdue(Loan loan) {
        return loan.getDueDate().isBefore(LocalDate.now());
    }

    public Money returnItem(Item item, Borrower borrower) {
        List<Loan> loans = loanDao.findLoansByBorrower(borrower);
        for (Loan loan : loans) {
            if (loan.getItem().equals(item) && !loan.isReturned()) {
                loan.setReturnDate(LocalDate.now());
                loanDao.update(loan);
                itemService.changeItemState(item.getBook().getTitle(), ItemState.AVAILABLE);
                if (isLoanOverdue(loan)) {
                    return calculateFine(loan);
                }
                return new Money(0);  // No fine due
            }
        }
        throw new IllegalArgumentException("No loan found for the item to be returned.");
    }

    private Money calculateFine(Loan loan) {
        long daysOverdue = ChronoUnit.DAYS.between(loan.getDueDate(), LocalDate.now());
        return new Money(daysOverdue *
loan.getBorrower().getCategory().getDailyFine().getAmount());
    }
}
```

Integration and Usage

- **Error Handling**: This service handles errors like no loan found for a returning item.

- **Dependencies**: The service depends on an `ItemService` which isn't provided here but would be necessary for changing the state of the item.

- **Fine Calculation**: Assumes that `Money` class and `BorrowerCategory` have methods to handle monetary calculations.

This setup provides a complete and testable approach for managing loans in your library system. Adjustments can be made to fit your exact system configuration and requirements. If you need further implementation details or expansion, feel free to ask!