Your submission to Graded Online Exercise (Ron Tobias Spannagel)

Mode: Exam

Date: Aug 1, 2023 **Time:** 11:00 - 12:40

Duration: 1h 40min

Begin of Student Review: Aug 3, 2023 08:00

End of Student Review: Aug 4, 2023 23:59

Examiner: Prof. Pramod Bhatotia **Module number:** IN0006 **Course:** Introduction to Software Engineering

Exercises: 6 **Points:** 100

Your result will be published here as soon as the correction is finished. You can get to this page by clicking on this exam in the exam overview of this course.

Exercise 1

Containers [12 Points]

EIST SoSe23 Final Exam Containers Exercise

In this exercise you receive:

- `app.py` ~> a flask application very similar to what we see during lecture, we run a flask application using `python3 app.py`
- `requirements.txt` ~> the file listing dependencies for our application
- `Dockerfile` ~> a simple Dockerfile you need to complete (DO NOT EDIT ABOVE LINE 5)
- `setup.py` ~> a python script you SHOULD run as soon as you clone the repository

Your task

- run the setup script (`python3 setup.py`)
- check the `SOLUTION.json` looks something like this:

```
{
    "Repo": "<SOMETHING>.git",
    "Flag": "insert_your_flag_here"
}
```

- complete the `Dockerfile` (install dependencies with `pip3 install -r requirements.txt` and run the Flask application)
- build the docker container (`docker build . -t eist23-final-container`)
- run the docker container (`docker run -t eist23-final-container`) (you probably won't be able to access it, see below for an hint)
- open the flask application in your browser
- retrieve the flag and update the file `SOLUTION.json`
- commit and push all your work (`Dockerfile`,`SOLUTION.json`)

Hint: in order to be able to access the container from your machine you need to publish the ports, the flask application is running on port 8000, check the documentation to understand how to publish a port.

Last Commit Hash: 068e61a

Exercise 2

Git [12 Points]

EIST SoSe23 Final exam: Git exercise: Merge the right merge

- Notice: There are `Warnings` in the end of the problem statement. Make sure to check them out.

We provide you with the following: src/git/MergeSort.java

After cloning the repository, without making any changes to it (if you do make changes, they will be stashed): run the following:

```
python3 create_branches.py
```

And then, see a list of the branches you have.

```
git branch
```

You should now have the following branches

- 1) `main`
- 2) `feature/merge-i` | i ∈ N in [1,5]

In the main branch, You have an implementation for `Merge Sort`, but the merge function that it calls has not yet been implemented.

- You have a job as a software engineer, and you are working on a project with `5 other teammates`.
- Your teammates are inexperienced programmers and thus they make a lot of mistakes
- For the current sprint, the task of each one of your teammates was to create `a feature branch for the implementation of the 'merge' function`.
- Only **one of them** has implemented the correct merge function.
- Find which branch has the correct implementation, then **merge** it to the main branch. Your teammates trust that you will make the right choice, thus they left the choice to you!
- Feel free to put whatever you want in the main function and run the program to see that it works as expected
- Commit, and push.

We wish you **merge the right merge**!

Good luck

Warnings:

- Don't just checkout to one of the branches, copy the code, and paste it to the other branch. That won't pass our tests. **You need to use the git merge command.**
- If it sorts the given array correctly, it doesn't most necessarily mean it is the right implementation.

Last Commit Hash: 8addb6d

⌨ Exercise 3

Parallelism [14 Points]

Parallelism: A Multi-threaded Saga

Due to a recent economic stimulus package for university students announced by the German government, students are rushing to the bank to take advantage of the financial benefits provided. However, this causes internal chaos at the bank, because the `BankAccount`s are unprepared for the overwhelming activity, which can lead to potential race conditions.

Note: For the sake of simplicity, the `balance` of a `BankAccount` may also be negative.

Part 1: Race Conditions

You have the following task:

1. **Fix the Race Conditions**

Your first task is to ensure the integrity of the banking system by preventing race conditions in the face of countless `deposit` and `withdraw` requests made on a single `BankAccount`. Implement synchronization mechanisms that ensure consistent execution of `deposit` and `withdraw` transactions, even when the system is faced with many such transactions executed in parallel.

Note that we have already provided a `main()` function in the `Main` class that allows you to test your solution for race conditions.

Part 2: Deadlocks

The bank also supports transferring money from one `BankAccount` to another using the `transfer()` method. Furthermore, the bank has recently introduced a new security feature: When transferring funds between two accounts, the `securityMutex` from each of the bank accounts involved in the transaction must be locked before, and unlocked after, the actual transfer of money. However, this creates a problem in an edge case that the programmers of the banking system have not noticed. For a certain multi-threaded combination of transfer requests (i.e. when multiple transfer requests are executed at the same time on different threads), the system will encounter a deadlock.

You have the following task:

1. **Fix the Deadlock**

It is now your job to find this deadlock, and prevent it.

The `Main` class already contains some code that will help you test whether the deadlock is (still) present.

As indicated in the todos in the code, you will have to make changes to the `transfer()` method. Importantly, you are **not** allowed to change any of the code in the `secureTransfer()` method. If you change anything in the `secureTransfer()` method, you will automatically receive 0 points for this part.

Hint: The `accountId` in `BankAccount` might be helpful.

Last Commit Hash: cb3730f

⌨ Exercise 4

MVC Pattern [14 Points]

Model View Controller. "AuthorHub: A Platform for Writers to Manage and Publish Books"

AuthorHub is a web-oriented platform designed exclusively for authors, providing a structured and efficient environment to manage and publish their books, as well as read books from other authors. The platform enables authors to create, update, and delete their books while tracking the progress of their publications.

To maintain organization and efficiency in system operations, AuthorHub follows the Model-View-Controller (MVC) design pattern.

Part 1: Model

The first task is to modify the Book model. At present, the Book model doesn't have any attributes. To make this model more meaningful for an authoring platform, we need to add several attributes to this model.

You have the following tasks:

1. **Implement the Book class:** Update the Book class by adding the following attributes: `author, title, genre, year, description` (all of type `String`), and `pageCount` (of type `Integer`). Implement the `constructor` to set all attributes and fill the corresponding `getters and setters` for each attribute.

Part 2: Controller

The "list view" typically refers to a type of user interface which presents a list of items or options to the user. In this case, the BookListView is a user interface view that displays a list of books.

In the next task you need to implement two methods in the `Controller` class: `saveBook()` and `displayBook()`.

You have the following tasks:)

1. **Implement saveBook in Controller:** Implement the `saveBook(Book)` method in the `Controller`. Make sure that the book gets added to the list view and all observers get notified about this.

2. **Implement displayBook in Controller:** Implement the `displayBook(Book)` method in the `Controller`. This method should open up the 'BookDetailView' for the selected book, allowing the user to read it.

Hint: You can display a view by calling the method `show()`.

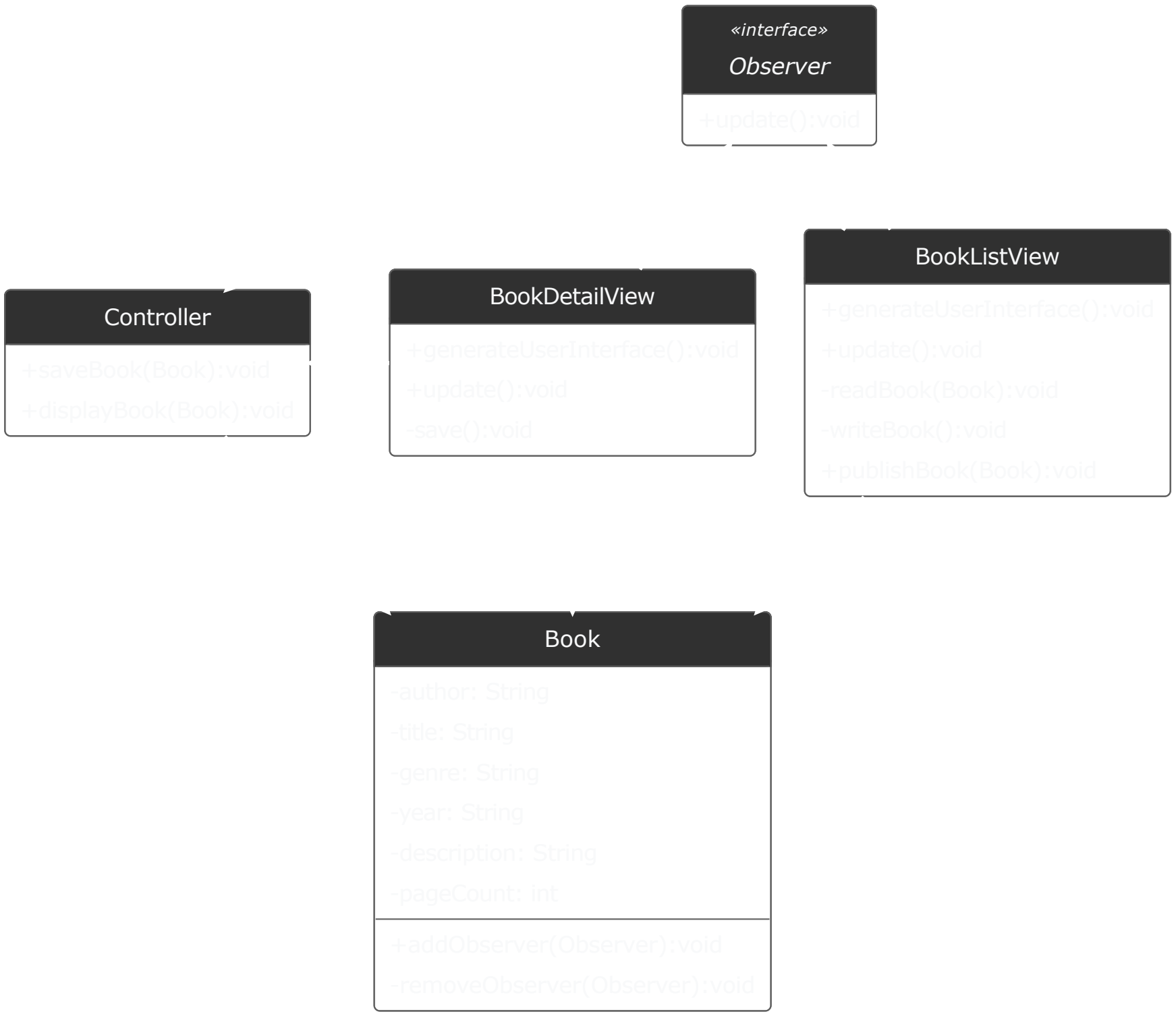
Part 3: View

The final part is to implement the `BookListView` and `BookDetailView` view components.

You have the following tasks:

1. **Implement readBook in BookListView:** In `BookListView`, implement a `readBook()` method. The `readBook()` method in `BookListView` appears to be a user-triggered action that tells the Controller to display details of the selected book.

2. **Implement writeBook in BookListView:** In `BookListView`, implement a `writeBook()` method to initiate the book creation process where the controller will be notified about a new book instance.
3. **Implement save method in BookDetailView:** Implement `save()` method in `BookDetailView`. This method should update the book with the information entered by the user and save it using the controller.
4. **Constructor BookDetailView:** Make sure that the view is an observer of each book in `BookDetailView`.



You didn't submit any solution for this exercise.

Exercise 5

Testing [20 Points]

Testing

The EIST team wants to implement a system to manage student absences for the practical course. For that we want to write several Unit Tests. We want to test functionalities that are independent of other services. Additionally, we also want to make sure to test if the correct methods of other services are called. In this exercise both students and instructors are modeled as `Person` entities and depending on if the flag `is_instructor` is set, the `Person` is either a student or an instructor.

Part 1: Unit Tests

Your job is to write tests for the `AbsenceService` class in `AbsenceServiceTest`

You have the following tasks:

1. **Implement testSaveAbsence():** We want to test the `saveAbsence(Absence absence)` method which stores an Absence into the database: Add a new test case `testSaveAbsence()` in the `AbsenceServiceTest` class. For that instantiate an `Absence` Object and save it. Afterwards, check whether that Object was saved correctly into the database. **Hint:** Utilize the `personBuilder` to instantiate and save a `Person` to the database.

2. **Implement testSaveAbsenceOnSameDate():** Write a test to make sure that we throw an `InvalidDataException` if we try to save an invalid Absence. That exception is triggered by saving a duplicate `Absence` object to the database. An `Absence` Object is a duplicate if the `date` of the absence is the same as an existing absence in the database AND it belongs to the same person. Add a new test case `testSaveAbsenceOnSameDate()` and check if the exception is thrown correctly.

Part 2: Mock Pattern

We also want to test if the `MailService` class is correctly utilized to alert the instructor of the course with an Email when an Absence is submitted. To receive an alert about a submitted absence the instructor needs to have selected in the Settings that they want to be alerted. We want to test the functionality of correctly triggering the email alert by mocking the `sendMail(String text)` method. For this we utilize the `EasyMock` framework.

You have the following tasks:

1. **Implement testSubmitAbsenceAlertInstructor():** Test the `submitAbsence(Absence absence, boolean alertInstructor)` method and mock the `sendMail(…)` method to return `true` when called and executed successfully. Write a new test case `testSubmitAbsenceAlertInstructor()` and verify that `sendMail(...)` is called when invoking `submitAbsence(absence, true)`. For that initialize two `Person` objects which correspond to a student and instructor each.

2. **Implement testSubmitAbsenceDoesNotAlertInstructor():** We also want to ensure that instructors are not alerted if they muted the alerts. Write a new test case `testSubmitAbsenceDoesNotAlertInstructor()` and verify that `sendMail(…)` is not called when invoking `submitAbsence(absence, false)`.

Important

- You can run the tests via gradles `test` task.
- Please name the test methods exactly as specified in the text below! Otherwise, the automated correction of your solution will fail and you might not get full points.
- You can find the test cases in the test folder. You only need to add code in the test methods. Do not change code in the src folder!
- All the services used in the test are set up correctly. Please DO NOT change them.

Last Commit Hash: c41788d

Exercise 6

Warning: You are viewing an illegal submission.

Facade Pattern [28 Points]

Facade with Access Policy

Problem Statement

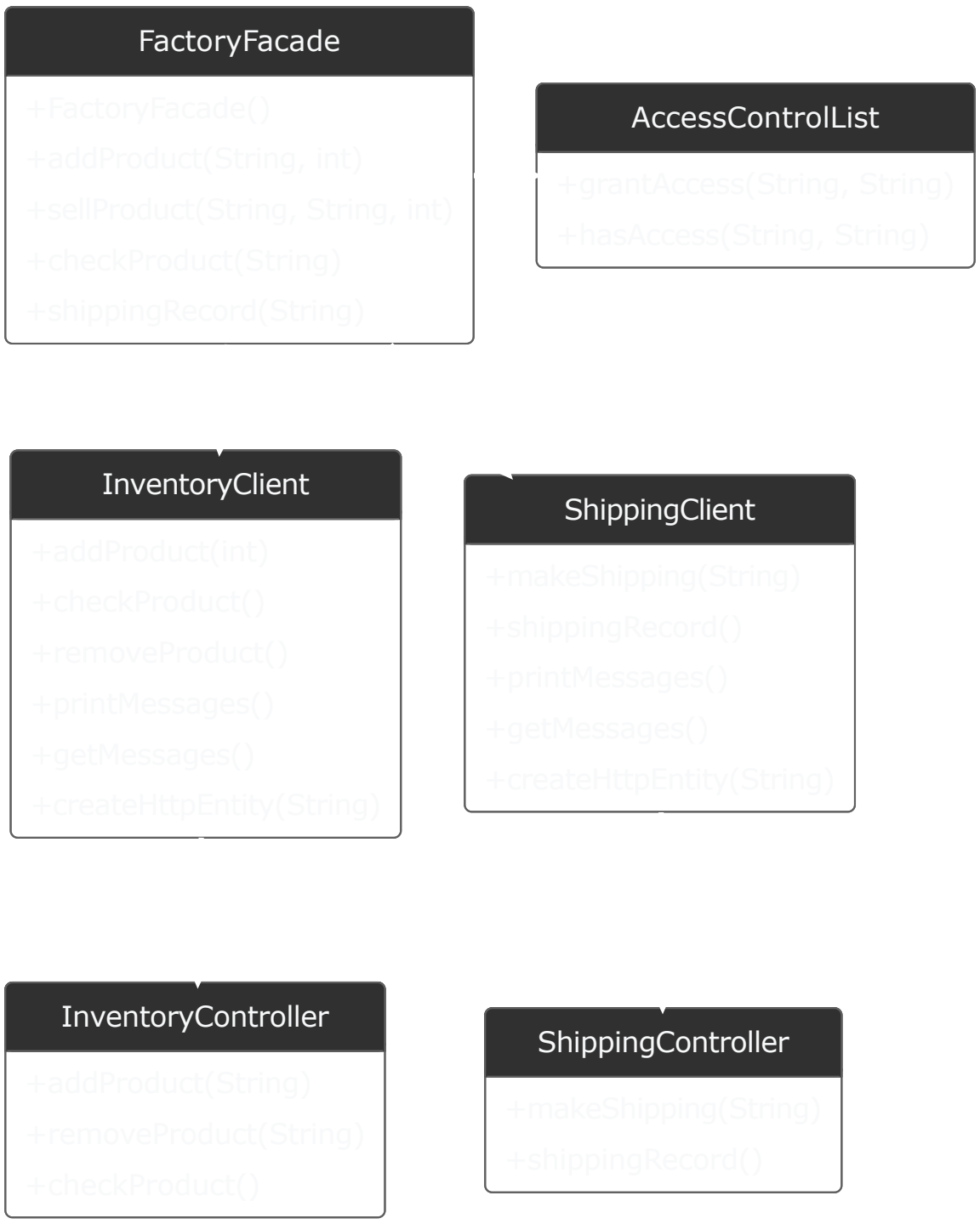
As a fresh graduate, you are hired by an old beer factory to perform some digitalization on their inventory and shipping system. The boss of the factory informed you that accessing the inventory and shipping should happen with necessary permissions. Since you should have built the system as soon as possible, initially you are given the below table for these permissions.

Role/Permission	Add	Sell	Check
SalesManager	x	x	x
SalesIntern	x		x
MarketingManager			x

From the table, you should implement each role and permissions with their exact match.

Also, as a design, you are given the below UML Diagram which shows the connections between classes and the systems. There are three different servers. The first one is the RootProject where your operations are located. Second one is the InventoryMicroservice and the third one is ShippingMicroservice. InventoryClient and ShippingClient in the RootProject are given to you as already built in. You should complete below tasks to finish missing parts of the project.

UML Diagram of the Application



Repository structure

- `InventoryMicroservice/`: Can be started via the gradle task `:InventoryMicroservice:bootRun`.
- `ShippingMicroservice/`: Can be started via the gradle task `:ShippingMicroservice:bootRun`.
- `src/eist/Client.java:main` place code to test your solution here.

Part 1: AccessControlList

First, you need to implement AccessControlList.

You have the following tasks:

1. **Implement Giving Access:** Implement the method `grantAccess(String role, String permission)` in the class `AccessControlList`. Make sure to not duplicate an existing role.

2. **Implement Control for Access:** Implement the method `hasAccess(String role, String permission)` in the class `AccessControlList`. It should check if a given role is existing with the given permission. If it is exist, return true, if it is not in the map, return false.

Part 2: FactoryFacade

Secondly, you need to implement your FactoryFacade class to provide an easy interface.

You have the following tasks:

1. **Implement Constructor:** Implement the method `FactoryFacade()` in the class `FactoryFacade`. Since you don't want to overload your Microservices with unpermitted requests, permission-checks should happen in Facade class. You should setup your AccessControlList instance in the constructor as it is defined in the above table. You can check the table and fill the ACL instance one by one with the permissions. Everytime a FactoryFacade instance is created, an AccessControlList object should be initialized with the same described permissions in the default constructor.

2. **Implement Adding New Products:** Implement the method `addProduct(String role, int product)` in the class `FactoryFacade`. A permisson check should be performed here. If the given role is permitted to perform the operation, it should continue to perform the task and add the product amount to the inventory. If the operation successful, you should return `getMessages()` method from related client. Otherwise "This role has no access" should be returned. To perform necessary operations, the provided methods of the InventoryClient or ShippingClient can be invoked.

3. **Implement Selling Products:** Implement the method `sellProduct(String role, String shippingAddress, int product)` in the class `FactoryFacade`. A permisson check should be performed here. If the given role is permitted to perform operation, it should add the shipping address to the shipping list to keep records. Moreover, it should continue to perform task and remove the product amount from the inventory and return a message with using `getMessages()` method of `InventoryClient`. If the given role has no rights to perform operation, "This role has no access" should be returned. To perform necessary operations the provided methods of the InventoryClient or ShippingClient can be invoked.

4. **Implement Check Available Products:** Implement the method `checkProduct(String role)` in the class `FactoryFacade`. A permisson check should be performed here. If the given role is permitted to perform the operation, it should continue to perform the task and return the amount of the product in the inventory. To return the message, `getMessages()` method of the related client can be used after correct method is called to perform operation. If the given role has no rights to perform operation, "This role has no access" should be returned. To perform necessary operations the provided methods of the InventoryClient or ShippingClient can be invoked.

5. **Implement Review Shipping Records:** Implement the method `shippingRecord(String role)` in the class `FactoryFacade`. A permisson check should be performed here. Similar to selling products, only people who have "Sell" permission can see the shipping records of the sold products. If the given role is permitted to perform the operation, it should continue to perform the task and return the list of records from the shipping list. To return the message, `getMessages()` method of the related client can be used after correct method is called to perform operation. If the given role has no rights to perform the operation, "This role has no access" should be returned. To perform necessary operations the provided methods of the InventoryClient or ShippingClient can be invoked.

!Important Notice: The permission checks should be implemented in a modular way. If the factory introduces new roles with the same set of permissions, checks should be able to work without any changes on the code.

Part 3: Microservices

At the end, you need to implement REST APIs inside the Microservices.

You have the following tasks:

!Important Notice/Hint! To be sure about type of your endpoints mapping and their URLs, check and understand InventoryClient and ShippingClient to see which mapping procedures and endpoints are used to reach these endpoints.

1. **Implement Add Product:** Implement the method `addProduct(String product)` in the class `InventoryController`. The given product parameter should be converted to the integer and added to the productNum. As a response, return a message as `"Products are added! New amount: <productNum>"`, where `<productNum>` is the integer instance of the InventoryController. An example message should look like this: `Products are added! New amount: 50`

2. **Implement Sell Product:** Implement the method `removeProduct(String product)` in the class `InventoryController`. The given product parameter should be converted to the integer and susbtracted from the productNum. If the given amount of the product is higher than the available one, the new product number in the inventory should be 0. At the end of the operations, return a message as `"Products are removed! New amount: <productNum>"`, where `<productNum>` is the integer instance of the InventoryController.

3. **Implement Check Product:** Implement the method `checkProduct()` in the class `InventoryController`. Return a response as `"Product amount: <productNum>"`, where `<productNum>` is the integer instance of the InventoryController.

4. **Implement Make Shipping:** Implement the method `makeShipping(String shipping)` in the class `ShippingControler`. The given shipping paramater should be added to the shippingList. Also, as a response message, `"Shipping is added to records"` should be returned.

5. **Implement Shipping Record:** Implement the method `shippingRecord()` in the class `ShippingControler`. Return the shippingList as the records of the shippings.

Last Commit Hash: 0f0a937