# Java OOP Regular Exam – 06 April 2024



# Overview

## Setup

* Upload **only the** **restaurant** package in every task **except Unit Tests**.
* **Do not modify the provided interfaces or their packages.**
* Use **strong cohesion** and **loose coupling.**
* **Use inheritance and the provided interfaces whenever possible**.
  + This includes **constructors**, **method parameters,** and **return types.**
* **Do not** violate your **interface** **implementations** by adding **more public methods** in the concrete class than the interface has defined.
* Make sure you have **no public fields** anywhere.

## Task 1: Structure (50 points)

You are given **4** interfaces and must implement their functionalities in the **correct classes**.

There are **4** types of entities in the application: **Waiter, Working, Client and TakenOrders**. There are also **2** repositories: a **WaiterRepository** and a **ClientRepository**.

### Waiter

Base**Waiter** is a **base class** for any **type of waiter** and **should not be instantiated**.

#### Data

* **name** – **String**
  + If the value of the name is either **null** or **empty** (containing only whitespaces), throw a **NullPointerException** with the following message: **"Waiter name cannot be null or empty."**
  + The values of the names are **unique.**
* **efficiency** – **int**
  + The efficiency of а waiter.
  + If the efficiency is a **negative** number, throw an **IllegalArgumentException** with the following message: **"Cannot create Waiter with negative efficiency."**
* takenOrders – TakenOrders
  + A TakenOrders field type.

#### Behaviour

##### abstract void work()

The **work()** method decreases the waiter's efficiency. Keep in mind that some Waiter types can implement the method differently.

* The method **decreases** the Waiter's efficiency **depending on** Waiter type:
* for Waiter of type **FullTimeWaiter** by **1 unit**.
* for Waiter of type **HalfTimeWaiter** by **2 units.**
* The efficiency value **should** **not** drop **below** **zero**.
* Set the value to be zero if the efficiency value drops below zero.

##### boolean canWork()

The **canWork()** method returns a **boolean**. Tell us if the efficiency is more than zero.

#### Constructor

A **BaseWaiter** should take the following values upon initialization:

String name, int efficiency

#### Child Classes

There are several concrete types of **BaseWaiter**:

**FullTimeWaiter**

Has **8 initial units of efficiency**.

The constructor should take the following values upon initialization:

Stri**ng** **name**

**HalfTimeWaiter**

Has **4 initial units of efficiency**.

The constructor should take the following values upon initialization:

String name

### TakenOrders

The TakenOrdersImpl class holds a **collection** of **orders**. It should be **instantiated**.

#### Data

* **ordersList** – a collection of **Strings**

#### Constructor

The constructor should not take any values upon initialization.

### Client

The **ClientImpl** class holds information about the **client orders** that can be ordered. It should be instantiated.

#### Data

* **name** – **String**
  + If the value of the **name** is either **null** or **empty** (containing only whitespaces), throw a **NullPointerException** with the following message: **"Invalid name!"**
* **clientOrders**– a collection of Strings

#### Constructor

The constructor should take the following values upon initialization:

String name

### Working

The **WorkingImpl** class holds the main action, which is the **takingOrders** method.

#### Behaviour

##### void takingOrders(Client client, Collection<Waiter> Waiters)

Here is how the **takingOrders** method works:

* Waiters **cannot** serve clients if their **efficiency** is **equal to or below** 0.
* They start working by **taking orders from clients,** one by one.
* If they **take an order**, they **add** it to their **getOrders** **list** and their **efficiency** is **decreased**.
* The taken **order** should then be **removed** from the **getClientOrders list** of the current client.
* Waiters **cannot** **continue** working if their **efficiency** **drops** to 0.
  + If their **efficiency** drops to 0 they get tired and **cannot** take any more orders, and the **next** waiter starts working**.**

### WaiterRepository

The **WaiterRepository** class is a **repository** for Waiters.

#### Data

* waiters – **a** **collection of Waiters**

#### Behaviour

##### void add(Waiter waiter)

* Adds a Waiter to the collection.
* Every Waiter is unique in the collection.
  + It is guaranteed that there will not be a waiter with the same name.

##### boolean remove(Waiter waiter)

* Removes a Waiter from the collection. Returns true if the deletion was successful.

##### Waiter byName(String name)

* Returns a Waiter with that name.
* If the Waiter is not in the collection, return null.

##### Collection<Waiter> getCollection()

* Returns an unmodifiable collection of Waiters.

### ClientRepository

The **ClientRepository** class is a **repository** for clients.

#### Data

* clients **– a collection of clients**

#### Behavior

##### void add(Client client)

* Adds a client to the collection.
* Every client is unique in the collection.
  + It is guaranteed that there will not be a **client** with the same **name**.

##### boolean remove(Client client)

* Removes a client from the collection. Returns true if the deletion was successful.

##### Client byName(String name)

* Returns a client with that name.
* If the client is not in the collection, return null.

##### Collection<Client> getCollection()

* Returns an unmodifiable collection of clients.

## Task 2: Business Logic (150 points)

### The Controller Class

The business logic of the program should be concentrated around several **commands**. You are given interfaces that you must implement in the correct classes.

**Note: The** ControllerImpl **class SHOULD NOT handle exceptions! The tests are designed to expect exceptions, not messages!**

The interface is Controller. You must create a ControllerImplclass, which implements the interface and implements all its methods. The constructor of ControllerImpl does **not take** any **arguments**. It should be instantiated. The given methods should have the following logic:

### Commands

There are several commands, which control the business logic of the application. They are stated below.

#### AddWaiter Command

##### Parameters

* **type – String**
* **waiterName – String**

##### Functionality

Creates a **Waiter** with the given **name** of the given **type** and saves it in the repository. If the type is invalid, throw an **IllegalArgumentException** with the following message:

**"Waiter type doesn't exist."**

Otherwise, the method should **return** the following message:

* **"Added {type}: {waiterName}."**

#### AddClient Command

##### Parameters

* **clientName** - String
* getClientOrders – String... (Varargs)

##### Functionality

Create a **client** with the provided **getClientOrders** and **name** and save it in the repository.

The method should **return** the following message:

* **"Added Client: {clientName}."**

#### RemoveWaiter Command

##### Parameters

* waiterName – String

##### Functionality

Remove the Waiter from working by removing it from the repository. If a Waiter with that name doesn’t exist, **throw IllegalArgumentException** with the following message:

* **"Waiter {waiterName} doesn't exist."**

##### If a Waiter is successfully removed, remove it from the repository and return the following message:

* **"Waiter {waiterName} was removed!"**

#### RemoveClient Command

##### Parameters

* clientName – String

##### Functionality

Remove the Client by removing it from the repository. If a Client with that name doesn’t exist, **throw IllegalArgumentException** with the following message:

* **" Client {clientName} doesn't exist."**

##### If a Client is successfully removed, remove it from the repository and return the following message:

* **" Client {clientName} was removed!"**

#### StartWorking Command

##### Parameters

* **clientName - String**

##### Functionality

When the **start working** command is called, the waiter starts taking orders from the given client.

* If you **don't have any** **Waiters**, throw an **IllegalArgumentException** with the following message: **"You must have at least one Waiter to start working."**
* After finishing work, you must **return the following message** with the **name of the served** client:

**"Client {clientName} was served."**

#### GetStatistics Command

##### Functionality

Returns the information about the waiters in the following format:

* If the waiters don't serve any clients, print **"None"** in their place.

**"{total count of served clients} client/s was/were served.**

**Waiter's statistics:**

**Name: {waiterName}**

**Efficiency: {waiterEfficiency}**

**Taken orders: {order1, order2, order3, …, ordern}**

**…**

**Name: {waiterName}**

**Efficiency: {waiterEfficiency}**

**Taken orders: {order1, order2, order3, …, ordern}"**

### Input / Output

You are provided with one interface, which will help you with the correct execution process of your program. The interface is called **Engine** and its **implementational** class should read the input. When the program finishes, the class should print the **output** to the **console**.

#### Input

These are the input commands:

* **AddWaiter** **{waiterType} {waiterName}**
* **AddClient** **{clientName} {**String... (getClientOrders)}
* **RemoveWaiter** **{waiterName}**
* **StartWorking {clientName}**
* **GetStatistics**
* **Exit**

#### Output

Print the output from each command when issued. If an exception is thrown during any of the commands' execution, print the exception message.

#### Examples

|  |
| --- |
| **Input** |
| **AddWaiter FullTimeWaiter Asen**  **AddWaiter HalfTimeWaiter George**  **AddClient Petar Spagethi Salad Shrimps Water Bread Burger**  **AddClient Kiril Cola**  **RemoveWaiter John**  **TakeOrders Petar**  **TakeOrders Kiril**  **GetStatistics**  **Exit** |
| **Output** |
| **Added FullTimeWaiter: Asen.**  **Added HalfTimeWaiter: George.**  **Added Client: Petar.**  **Added Client: Kiril.**  **Waiter John doesn't exist.**  **Client Petar was served.**  **Client Kiril was served.**  **2 client/s was/were served.**  **Waiter's statistics:**  **Name: Asen**  **Efficiency: 1**  **Taken orders: Spagethi, Salad, Shrimps, Water, Bread, Burger, Cola**  **Name: George**  **Efficiency: 4**  **Taken orders: None** |
| **Input** |
| **AddWaiter HalfTimeWaiter Zahari**  **AddWaiter FullTimeWaiter Paul**  **AddWaiter FullTimeWaiter Maria**  **AddClient Peter Steack Seabass**  **AddClient Nelly Pizza Salmon**  **AddClient Hary Wine**  **AddClient Kamen Coffee Cake**  **AddClient John Beer**  **RemoveClient John**  **RemoveWaiter Paul**  **TakeOrders Peter**  **TakeOrders Kamen**  **TakeOrders Hary**  **TakeOrders Nelly**  **GetStatistics**  **Exit** |
| **Output** |
| **Added HalfTimeWaiter: Zahari.**  **Added FullTimeWaiter: Paul.**  **Added FullTimeWaiter: Maria.**  **Added Client: Peter.**  **Added Client: Nelly.**  **Added Client: Hary.**  **Added Client: Kamen.**  **Added Client: John.**  **Client John was removed!**  **Waiter Paul was removed!**  **Client Peter was served.**  **Client Kamen was served.**  **Client Hary was served.**  **Client Nelly was served.**  **4 client/s was/were served.**  **Waiter's statistics:**  **Name: Zahari**  **Efficiency: 0**  **Taken orders: Steack, Seabass**  **Name: Maria**  **Efficiency: 3**  **Taken orders: Coffee, Cake, Wine, Pizza, Salmon** |

## Task 3: Unit Tests (100 points)

You will receive a skeleton with three classes inside – **Main**, **FullTimeWaiter,** and **Restaurant**. **The Restaurant** class will have some methods, fields, and constructors. Cover the whole class with the unit test to make sure that the class is working as intended. In Judge, you upload **.zip** to **busyWaiters (**with **RestaurantTests** inside**)** from the **skeleton**.