# OOP Advanced Regular Exam – Call of Duty

There is a civil war burning in the lands of Mother Russia, from the Ultranationalist movement. U.S. Force Reconnaisance Marines and British SAS commando have released their Agents and are hiring them for special missions. You have been chosen as a Lead Developer on the system that the Mission Control uses to distribute missions. In this Modern Warfare, quality is what is required, so make sure you create a good High Quality-Coded system. After all, it’s your Call of Duty.

### Overview

The software will work with Agents and Missions as its main business models. You must implement a complex logic around those entities, but don’t worry... Everything will be described to you. There was a previous developer – Ivo, who wrote some of the Mission loading logic, but unfortunately he fell into coma from excitement, when he heard McDonalds are releasing Triple Cheeseburgers. You must use his code, that’s one of the requirements.

### Structure

The structure of the software circles around Agents and Missions.

#### Agents

The Agents are initialized with an id (**String**), a name (**String**) and a rating (**Double**).

There are generally 2 types of Agents.

##### Novice Agent

The Novice Agent is just a normal agent. Nothing special here.

* The rating of the Novice Agent is always **0** upon **initialization**.

##### Master Agent

The Master Agent is initialized an id (**String**), a name (**String**), a rating (**Double**), and several additional properties:

* bounty – a **Double**.
  + The bounty is always **0** upon **initialization**.

#### Missions

The Missions are initialized with an id (**String**), a rating (**Double**) and a bounty (**Double**).  
There are generally 3 types of Missions.

##### Escort Mission

The EscortMission is just a normal mission.

* **Decreases** its given rating by **25%**.
* **Increases** its given bounty by **25%**.

##### Hunt Mission

The HuntMission is just a normal mission.

* **Increases** its given rating by **50%**.
* **Increases** its given bounty by **100%**.

##### Surveillance Mission

The SurveillanceMission is just a normal mission.

* **Decreases** its given rating by **75%**.
* **Increases** its given bounty by **50%**.

#### Mission Control

The MissionControl will be given to you in the skeleton.   
You can check more info about it in the Skeleton section.

### Functionality

The main functionality circles around the Agents and the Missions. Agents are being registered in the system, and after that they can start requesting Missions.

When an Agent requests a Mission, it is generated for him by the MissionControl, and assigned to him. An Agent can complete his assigned missions, in which case he gets rating (and bounty if he is a MasterAgent) for every mission completed. Each Mission gives the Agent that completes it some rating and bounty.

The system should also support a **functionality** for checking the **status** of an Agent or a Mission.

When an Agent completes a Mission, he doesn’t just complete 1 of his assigned missions, he completes  
**all of them**.

##### Example

Agent – Pesho, has **requested 5 missions**, and they have been **assigned** to **him**. When he **starts** **completing**, he completes **all 5 missions** and takes his rating and bounty for them.

Now that you’ve gotten the hang of the main business logic, you can check some detailed information about its components.

Check below, each section, and the functionality it describes.

#### Agents

The Agents are the main functional entities of the whole system. They accept Missions and complete them, thus earning rating… and bounty, if they are MasterAgents.

When an Agent is registered, by default he is a NoviceAgent. NoviceAgents only get rating from completing Missions, because they are not yet experienced enough to take bounties. When a NoviceAgent completes   
**3** missions, he becomes a MasterAgent, and can start taking bounties from Missions too.

In other words, the system should replace the NoviceAgent object, with a MasterAgent object with the same properties. When a NoviceAgent is promoted to a MasterAgent, he **KEEPS** his record of **completed missions**.

##### Note

If a NoviceAgent has **5 missions**, and decides to **complete** them, he **does NOT** become a MasterAgent on the **3th mission**. He completes them all, taking **ONLY** rating for their completion, and then he becomes a MasterAgent, and can start taking bounty for completing newly requested Missions.

#### Missions

The Missions have no business logic around themselves. They are just **data models**.

#### Commands

There are several commands which are given from the user input, in order to control the program.   
Here you can see how they are formed.

The **parameters** will be given in the **EXACT ORDER**, as the one **specified below**.   
You can see the exact input format in the **Input section**.

**Each** **command** will **generate an output** **result**, which you must **print**.  
You can see the exact output format in the **Output section**.

##### Agent Command

**Parameters** – **id** (string), **name** (string).

Creates a NoviceAgent with the **given id**, and the **given name**.

##### Request Command

**Parameters** – **agentId** (string), **missionId** (string), **missionRating** (double), **missionBounty** (double).

Generates a Mission, with the **given** missionId, missionRating and missionBounty, using the MissionControl, and **assigns** it to the Agent with the **given** agentId.

**Note**: You should pass the missionId, missionRating and missionBounty to the **method** for **generating** missions of the MissionControl.

##### Complete Command

**Parameters** – **agentId** (string).

Makes the Agent with the **given** agentId complete **all** his **assigned** missions.

##### Status Command

**Parameters** – **id** (string)

Brings report of the **Agent** or the **Mission** with the **given id**, providing **detailed** **information** about the corresponding object.

##### Over Command

**Exits** the program. Prints **detailed information** about the **whole** system.

### Skeleton

In this section you will be given information about the Skeleton, or the code that has been given to you.

You are allowed to change the **internal** and **private logic** of the **classes** that have been given to you.   
In other words, you can change the **body code** and the **definitions** of the **private members** in whatever   
way you like.

However. . .

You are **NOT ALLOWED** to **CHANGE** the **Interfaces** that have been provided by the **skeleton** in **ANY way**.   
You are **NOT ALLOWED** to **ADD** more **PUBLIC LOGIC**, than the **one**, **provided** by the **Interfaces**, **ASIDE FROM** the toString() method and compareTo() method.

#### Interfaces & Others

You will be given the **interfaces** for the Agent and Mission entities. You should use them when you are implementing your entities.

You will **also be given** an **interface** for the MissionControl class,   
but you will also be given the **class themselves**.

#### MissionControl

The MissionControl class **instantiates** a map with Mission classes. The classes must be **named** in a **specific way** and should be **positioned** in a **specific package**, in order for the **instantiation** to work.

The MissionControl class uses the Map with Mission classes, to dynamically instantiate objects of them. Using the **iterator** of the **map**, the MissionControl generates missions in the following order:

1. EscortMission
2. HuntMission
3. SurveillanceMission

The iterator is casually restarted, In order to generate an EscortMission again, after generating a SurveillanceMission, and continue the order again from the **start**.

The MissionControl also assignes **values** to the id, the rating and the bounty of the generated Missions, which values come from the user input.  
Those values should be in the following **ranges**:

##### id

* id – **string** which should have a **maximum length** of **8 characters**. May consist of **ANY** character.
* If the id has a **higher length** than the **maximum** allowed, it is **substringed** to the **maximum allowed length**.

##### Rating

* rating – **floating-point number** in **range** [**0, 100**].
* If the rating has a **value lower** than the **minimum** allowed, it is **set** to the **minimum allowed value**.
* If the rating has a **value higher** than the **maximum** allowed, it is **set** to the **maximum allowed value**.

##### Bounty

* bounty – **floating-point number** in **range** [**0, 100000**].
* If the bounty has a **value lower** than the **minimum** allowed, it is **set** to the **minimum allowed value**.
* If the bounty has a **value higher** than the **maximum** allowed, it is **set** to the **maximum allowed value**.

NOTE: The **evalutation** of the **minimum** and **maximum** **values** is done **BEFORE** they are passed to the Mission object.

### Input

The input consists of several commands which will be given in the format, specified below: :

* Agent {id} {name}
* Request {agentId} {missionId} {missionRating} {missionBounty}
* Complete {agentId}
* Status {id}
* Over

### Output

Each of the commands generates **output**. Here are the **output formats** of each command:

* Agent Command – registers an Agent with the given id and given name. Prints the following result:

**Registered Agent – {name}:{id}**

* Request Command – generates a Mission, with the given properties, and assigns it to the Agent, with the given id.   
  Prints the following result:

**Assigned {missionType} Mission - {missionId} to Agent - {agentName}**

* Complete Command – makes the Agent, with the given id, complete all of his assigned Missions.   
  Prints the following result:

**Agent - {name}:{id} has completed all assigned missions.**

* Status command – provides **detailed** **information** about anAgentor aMission, in one of the following formats:

|  |
| --- |
| NoviceAgent |
| {agentType} Agent – {name}  Personal Code: {id}  Assigned Missions: {assignedMissionsCount}  Completed Missions: {completedMissionsCount}  Rating: {rating} |
| MasterAgent |
| {agentType} Agent – {name}  Personal Code: {id}  Assigned Missions: {assignedMissionsCount}  Completed Missions: {completedMissionsCount}  Rating: {rating}  Bounty Earned: ${bounty} |
| Mission |
| {missionType} Mission – {id}  Status: {Open / Completed}  Rating: {rating}  Bounty: {bounty} |

* + If the Mission is still assigned (not completed) you should print “Open” as its status.
  + If the Mission is completed you should print “Completed” as its status.
* Over command – Terminates the program; **prints** detailed statistics about the whole system. The format, in which the statistics should be printed is:

Novice Agents: {noviceAgentsCount}  
Master Agents: {masterAgentsCount}  
Assigned Missions: {totalAssignedMissionsCount}  
Completed Missions: {totalCompletedMissionsCount}  
Total Rating Given: {totalRatingEarned}  
Total Bounty Given: ${totalBountyEarned}

* + The totalAssignedMissionsCount is the **total amount** of Missions that have been **assigned**.   
    **INCLUDING** the  **completed** missions.
  + The totalRatingEarned and totalBountyEarned are the **SUMS** of the ratingsandbountiesthat theAgentsearned**,** from **completing** theirMissions.

Note: All output **floating-point numbers** must be formatted to the **2nd digit** after the **decimal point**.

### Constrains

* All **numbers** in the input will be in **range [0, 1.000.000.000]**.
* All **input lines** will be **absolutely valid**.
* There will be **NO** non-existent **ids** in the input.
* There will be **NO** duplicate **ids** in the input.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Agent 007b Donald  Agent 1xbx Peter  Request 007b 1x1 25 5000  Request 007b 2x2 50 15000  Request 007b 3x4 75 55000  Status 007b  Complete 007b  Status 007b  Over | Registered Agent - Donald:007b  Registered Agent - Peter:1xbx  Assigned Escort Mission - 1x1 to Agent - Donald  Assigned Hunt Mission - 2x2 to Agent - Donald  Assigned Surveillance Mission - 3x4 to Agent - Donald  Novice Agent - Donald  Personal Code: 007b  Assigned Missions: 3  Completed Missions: 0  Rating: 0,00  Agent - Donald:007b has completed all assigned missions.  Master Agent - Donald  Personal Code: 007b  Assigned Missions: 0  Completed Missions: 3  Rating: 112,50  Bounty Earned: $0,00  Novice Agents: 1  Master Agents: 1  Assigned Missions: 3  Completed Missions: 3  Total Rating Given: 112,50  Total Bounty Given: $0,00 |
| Agent 007 James  Request 007 1q2e 120 5000  Request 007 2e1q -15 120000  Request 007 333e 20 5000  Request 007 55e 50 20000  Status 1q2e  Complete 007  Status 2e1q  Request 007 1111 20 3000  Complete 007  Over | Registered Agent - James:007  Assigned Escort Mission - 1q2e to Agent - James  Assigned Hunt Mission - 2e1q to Agent - James  Assigned Surveillance Mission - 333e to Agent - James  Assigned Escort Mission - 55e to Agent - James  Escort Mission - 1q2e  Status: Open  Rating: 75,00  Bounty: 6250,00  Agent - James:007 has completed all assigned missions.  Hunt Mission - 2e1q  Status: Completed  Rating: 0,00  Bounty: 200000,00  Assigned Hunt Mission - 1111 to Agent - James  Agent - James:007 has completed all assigned missions.  Novice Agents: 0  Master Agents: 1  Assigned Missions: 5  Completed Missions: 5  Total Rating Given: 147,50  Total Bounty Given: $6000,00 |

### Tasks

#### Task 1: High Quality Structure

##### Refactor the given Skeleton code and use it.

You have been given some **interfaces**, which you must **implement** – each and every one of them!

Ivo wrote some code before you, but he writes really bad and broken code… He somehow managed to write the MissionControl class. His work, however, is not that trustworthy, so you might have to give it an eye or two, for potential **functionality bugs** and things that **do NOT follow** the **good practices** of **Object-Oriented Programming**.

Refactor anything, which will **improve** the **code quality**, in your opinion. Be careful **NOT** to **break the code** or one of the **rules** specified in the **Skeleton** **section**.

##### High Quality Code.

Achieve good separation of concerns using abstractions and interfaces to decouple classes, while reusing code through inheritance and polymorphism. Your classes should have strong cohesion - have single responsibility and loose coupling - know about as few other classes as possible.

##### Reflection.

Since the Agent classes do not reveal a lot for their missions, you will probably have to use a some reflection for the business logic of the Complete, Status and Over commands, in order to extract the completed and assigned missions from the classes.

Note: For this task, submit only the “callofduty” folder.

#### Task 2: Correct business logic.

The given code provides some functionality, but it does not cover the entire task. Implement the rest of the business logic, using the given code, and implement everything following the requirements specification. Check your solutions in the Judge system.

Note: For this task, submit the whole “src” folder.

Note: The High-Quality Structure tests are **not connected** to the Business Logic tests, which gives you the ability to break the structure completely in order for the Business Logic to pass. You are **NOT** **allowed** to do that**. Each submission** on the **Business Logic** will be **checked** with the **High-Quality Structure** tests.

#### Task 3: Unit Testing.

Test the MissionControl class’s **PUBLIC** methods for potential bugs. There is **only 1 public method**, but you can test a **lot** of **private logic** with it.Extensive testing might require you to have some of the core logic implemented, in order to cover all cases.

When testing, use **ONLY THE CLASSES, PROVIDED** by the **SKELETON** + the Mission classes.

For this task submit the **folder** you have put your **tests** into. The **root test package** folder.

**NOTE**: You are **NOT ALLOWED** to submit **non-test classes** for this task.