# C# OOP Retake Exam – Robot Service

### Overview

In this exam, you need to build a robot service project, which has support for **robots**, **procedures** for storing procedures of robots,and a **garage** for storing robots in the robot service. The project will consist of **model classes** and a **controller class**, which manages the **interaction** between the **robots**, **procedures** and **garage**.

## Setup

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

## Task 1: Structure (50 points)

For this task’s evaluation logic in the methods isn’t included.

You are given interfaces, and you have to implement their functionality in the **correct classes**.

There are **3** types of entities in the application: **Robot, Procedure, Garage**.

### Robots

The Robot is a **base class** for any **robot** and it **should not be able to be instantiated**.

#### Data

* Name – **string**
* Happiness – **int (**can't be **less** **than** **0** or **more than** **100**. In these cases throw **ArgumentException** with message "Invalid happiness"**)**
* **Energy – int (**can't be **less than** **0** or **more than** **100**. In these cases throw **ArgumentException** with message "Invalid energy"**)**
* **ProcedureTime – int**
* **Owner – string (**by default: **"**Service**")**
* **IsBought – bool (**by default: **false)**
* **IsChipped – bool (**by default: **false)**
* **IsChecked – bool (**by default: **false)**

#### Constructor

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

string name, int energy, int happiness, int procedureTime

**Override** ToString() **method in the format:**

**"**Robot type: {robot type} - {robot name} - Happiness: {robot happiness} - Energy: {robot energy}**"**

**Note: There is a space in the beginning of the sentence!**

#### Child Classes

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

* HouseholdRobot
* WalkerRobot
* PetRobot

### Procedures

The Procedure is a **base class** for any **procedures** and it **should not be able to be instantiated**.

#### Data

* Robots – collection of **Robots** accessible only by the child classes. Collection should contains all robots which has visited specific procedure.

#### Constructor

A **procedure** should not take any values upon initialization

#### Behavior

##### string History()

Returns a string with information about **current procedure** and its robots. The returned string must be in the following format:

**"**{procedure type}**"**

**"** Robot type: {robot type} - {robot name} - Happiness: {happiness} - Energy: {energy}**"**

**Note: You should append robot information for each robot in the collection!**

**void DoService(IRobot robot, int procedureTime)**

**Each procedure** implements its own DoService()method with different logic, which is explained below.

Each procedure must check if the robot procedure time is **more than or equal to** the time each procedure will take. If robot procedure time is lower than the time for the current procedure throw ArgumentException with message **"**Robot doesn't have enough procedure time**"**

Every time a procedure command is called, the time the procedure took is **subtracted** from the **robot’s** **allowed procedure time**.

#### Child Classes

There are several concrete types of **procedures**, which execute **different logic** when DoService() is called:

* Chip – removes **5** happiness and chips current robot. Robot can be chipped once. If robot is already chipped throw an ArgumentException with message **"**{robot name} is already chipped**"**
* Charge – adds **12** happiness and **10** energy
* Rest – removes **3** happiness and adds **10** energy
* Polish – removes **7** happiness
* Work – removes **6** energy and adds **12** happiness
* TechCheck – removes **8** energy and **checks** current robot (set **IsChecked** to **true**). If robot is already checked, just remove 8 energy again.

### Garage

The Garage is a **class** which **should be able to be instantiated**.

The **garage** is a building, which holds **robots**.

#### Data

* Capacity – **int** with a constant value of **10**
* Robots – Collection with the robot's **name** as the **key** and the **robot itself** as the **value**

#### Constructor

A **garage** should not take any values upon initialization:

#### Behavior

##### void Manufacture(IRobot robot)

If there isn't enough capacity in the garage throw an InvalidOperationException with the message **"**Not enough capacity**"**

If a robot with this name already exists in the garage, throw an ArgumentException with the message **"**Robot {robot name} already exist**"**

In any other case, add the current robot to the garage.

##### void Sell(string robotName, string owner)

If the provided robot name does not exist in the garage, throw an ArgumentException with the message **"**Robot {robot name} does not exist**"**

If a robot with that name exists, **change its owner**, its **bought status** and **remove the robot from the garage**.

## 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, which you have to implement in the correct classes.

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

The first interface is **I**Controller. You must create a Controllerclass, which implements the interface and implements all of its methods. The constructor of Controllerdoes not take any arguments. The given methods should have the logic described for each in the Commands section.

### Commands

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

**NOTE:** For each command except for **"**Manufacture**" and "History"**, you must check if a robot with that name exist in the garage. If it doesn't, throw an ArgumentException with the message **"**Robot {robot name} does not exist**"**.

#### Manufacture Command

##### Parameters

* RobotType – string
* Name – string
* Energy – int
* Happiness – int
* ProcedureTime – int

##### Functionality

Creates a robot with the correct type and **registers** it in the garage.

If the robot type is invalid throw ArgumentException with the message "{robotType} type doesn't exist"

If the robot happiness or energy are invalid throw ArgumentException for Invalid **energy**/**happiness**.

If the garage capacity is not enough, don’t register the robot and throw InvalidOperationException, with the correct message for capacity.

If a robot with the **same name** already exist in the garage, don’t register it and throw ArgumentException with message **"**Robot {robot name} already exist**"**

If successful, returns "Robot {robot name} registered successfully".

#### Chip Command

##### Parameters

* Name - string
* ProcedureTime - int

##### Functionality

Calls the Chip procedure with parameters **currentRobot** and **procedureTime**.

Returns "{current robot name} had chip procedure".

#### TechCheck Command

##### Parameters

* Name - string
* ProcedureTime - int

##### Functionality

Calls the TechCheck procedure with parameters **currentRobot** and **procedureTime**.

Returns "{current robot name} had tech check procedure".

#### Rest Command

##### Parameters

* Name - string
* ProcedureTime - int

##### Functionality

Calls the Rest procedure with parameters **currentRobot** and **procedureTime**.

Returns "{current robot name} had rest procedure".

#### Work Command

##### Parameters

* Name - string
* ProcedureTime - int

##### Functionality

Calls the Work procedure with parameters **currentRobot** and **procedureTime**.

Returns "{current robot name} was working for {procedure time} hours".

#### Charge Command

##### Parameters

* Name - string
* ProcedureTime - int

##### Functionality

Calls the Charge procedure with parameters **currentRobot** and **procedureTime**.

Returns "{current robot name} had charge procedure".

#### Polish Command

##### Parameters

* Name - string
* ProcedureTime - int

##### Functionality

Calls the Polish procedure with parameters **currentRobot** and **procedureTime**.

Returns "{current robot name} had polish procedure".

#### Sell Command

##### Parameters

* robotName - string
* ownerName - string

##### Functionality

Finds the robot with that name in the garage and sells it.

Returns:

If the current robot **is** chipped:

**"**{owner} bought robot with chip**"**

If the current robot **is not** chipped:

**"**{owner} bought robot without chip**"**

#### History Command

##### Parameters

* procedureType - string

##### Functionality

Returns information about **all robots** which had current procedure type in the format:

"{procedure type}"

" Robot type: {robot type1} - {robot name1} - Happiness: {robot happiness1} - Energy: {robot energy1}"

" Robot type: {robot type2} - {robot name2} - Happiness: {robot happiness2} - Energy: {robot energy2}"

#### Exit Command

##### Functionality

Ends the program.

## Input / Output

You are provided with one interface, which will help you with the correct execution process of your program. The interface is IEngine and the class implementing this interface should read the input and when the program finishes, this class should print the output.

You are given the **Engine** class with written logic in it. In order the code to be **compiled**, some parts are **commented**, **don’t forget to comment them out**. The **try-catch block** is also **commented** in order for the program to **throw exceptions and for you to see them**, **comment it out** when you are **ready** with this too.

### Input

Below, you can see the **format** in which **each command** will be given in the input:

* Manufacture {type} {name} {energy} {happiness} {procedureTime}
* Chip {name} {procedureTime}
* TechCheck {name} {procedureTime}
* Rest {name} {procedureTime}
* Work {name} {procedureTime}
* Charge {name} {procedureTime}
* Polish {name} {procedureTime}
* Sell {robot name} {owner}
* History {procedureType}
* 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** |
| Manufacture HouseholdRobot Cortana 30 50 6  Chip Cortana 3  Sell Cortana John  Manufacture PetRobot Alexa 20 40 5  Chip Alexa 3  Work Alexa 1  Manufacture Lion Cortana 30 50 6  Polish Alexa 4  Charge Alexa 6  History Chip  Sell Alexa John  Exit |
| **Output** |
| Robot Cortana registered successfully  Cortana had chip procedure  John bought robot with chip  Robot Alexa registered successfully  Alexa had chip procedure  Alexa was working for 1 hours  Lion type doesn't exist  Robot doesn't have enough procedure time  Robot doesn't have enough procedure time  Chip  Robot type: HouseholdRobot - Cortana - Happiness: 45 - Energy: 30  Robot type: PetRobot - Alexa - Happiness: 47 - Energy: 14  John bought robot with chip |

|  |
| --- |
| **Input** |
| Manufacture PetRobot Siri 100 50 100  Manufacture HouseholdRobot Alexa 50 40 80  Manufacture HouseholdRobot Sophia 60 40 60  Manufacture WalkerRobot Cortana 10 20 15  Manufacture WalkerRobot Cortana 10 20 14  Manufacture InvalidRobot FalseName 20 40 15  Manufacture HouseholdRobot InvalidEnergy -20 40 15  Manufacture WalkerRobot InvalidHappines 20 -40 15  Chip Cortana 10  Chip Cortana 10  Chip Alexa 10  Chip Sophia 10  Rest Sophia 10  Rest InvalidName 20  Work Siri 20  Sell Siri John  Sell Cortana Alex  Sell Invalid Name Alex  Manufacture HouseholdRobot Koly 10 20 13  Manufacture PetRobot Willy 43 20 100  Manufacture WalkerRobot Jack 10 34 55  Manufacture HouseholdRobot Fast 80 20 14  Manufacture WalkerRobot Zoom 22 12 90  Manufacture PetRobot Spider 50 60 100  Manufacture HouseholdRobot Robot010 10 44 50  Manufacture WalkerRobot Oliver 35 55 10  History Chip  Exit |
| **Output** |
| Robot Siri registered successfully  Robot Alexa registered successfully  Robot Sophia registered successfully  Robot Cortana registered successfully  Robot Cortana already exist  InvalidRobot type doesn't exist  Invalid energy  Invalid happiness  Cortana had chip procedure  Robot doesn't have enough procedure time  Alexa had chip procedure  Sophia had chip procedure  Sophia had rest procedure  Robot InvalidName does not exist  Siri was working for 20 hours  John bought robot without chip  Alex bought robot with chip  Robot Invalid does not exist  Robot Koly registered successfully  Robot Willy registered successfully  Robot Jack registered successfully  Robot Fast registered successfully  Robot Zoom registered successfully  Robot Spider registered successfully  Robot Robot010 registered successfully  Robot Oliver registered successfully  Chip  Robot type: WalkerRobot - Cortana - Happiness: 15 - Energy: 10  Robot type: HouseholdRobot - Alexa - Happiness: 35 - Energy: 50  Robot type: HouseholdRobot - Sophia - Happiness: 32 - Energy: 70 |

## Task 3: Unit testing (100 points)

You will receive a skeleton with one class inside. The class will have some methods, properties, fields and constructors. Cover the whole class with unit test to make sure that the class is working as intended.