# Object-Oriented Programming – Practical Exam

## Problem 1 – War Machines

In a war machine virtual factory there are two types of **machines**: **tanks** and **fighters**. Each machine has **name**, **pilot**, **health points**, **attack points**, **defense points** and **attack targets**. Each pilot has **name** and **machines** he **engages**. Pilots make status **reports** on all machines they engage. One machine can be **engaged** by one pilot at a time. **Tanks** have **defense mode** which can be turned **on** and **off**. **Fighters** have stealth mode which can be turned **on** and **off**.

### Design the Class Hierarchy

Your **task** is to **design an object-oriented class hierarchy** to model the war machines factory, machines and pilots **using the best practices for object-oriented design (OOD) and object-oriented programming (OOP)**. Avoid duplicated code though abstraction, inheritance, and polymorphism and encapsulate correctly all fields.

You are given few C# **interfaces** that you should **obligatory** implement and use as a basis of your code:

|  |
| --- |
| namespaceWarMachines.Interfaces  {  publicinterfaceIPilot  {  string Name { get; set; }  void AddMachine(IMachinemachine);  stringReport();  }  publicinterfaceIMachine  {  string Name { get; set; }  IPilot Pilot { get; set; }  double HealthPoints { get; set; }  double AttackPoints { get; }  double DefensePoints { get; }  IList<string> Targets { get; }  void Attack(string target);  string ToString();  }  publicinterfaceITank : IMachine  {  bool DefenseMode { get; }  void ToggleDefenseMode();  }  publicinterfaceIFighter : IMachine  {  bool StealthMode { get; }  void ToggleStealthMode();  }  publicinterfaceIMachineFactory  {  IPilot HirePilot(string name);  ITank ManufactureTank(string name, double attackPoints, double defensePoints);  IFighter ManufactureFighter(string name, double attackPoints,  double defensePoints, bool stealthMode);  }  } |

All your machines should implement IMachine. Pilots should implement IPilot. Tanks and fighter courses should implement ITank and IFighter respectively. Machines and pilots should be created only through the IMachineFactory interface implemented by a class named MachineFactory. Tank’s initial health points are always 100 and fighter’s initial health points are always 200. Tank’s defense mode adds 30 defense points to the initial ones and removes 40 attack points from the initial ones. By default tanks’ defense mode is turned on. Fighters in stealth mode can be attacked only by other fighters.

The IPilot.Report() method returns the information about a pilot in the following form:

|  |
| --- |
| ***(pilot name)*–*(number of machines/”no”)(“machine”/”machines”)***  **- *(machine name)***  **\*Type: *(“Tank”/”Fighter”)***  **\*Health: *(machine health points)***  **\*Attack: *(machine attack points)***  **\*Defense: *(machine defense points)***  **\*Targets: *(machine target names/”None” – comma separated)***  **\*Defense: *(“ON”/”OFF” – when applicable)***  **- *(machine name)***  **\*Type: *(machine type)***  **\*Health: *(machine health points)***  **\*Attack: *(machine attack points)***  **\*Defense: *(machine defense points)***  **\*Targets: *(machine target names/”None” – comma separated)***  **\*Stealth: *(“ON”/”OFF” – when applicable)*** |

Look into the example below to get better understanding of the printing format.

The listed machines added to a certain pilot (though the AddMachine(…) method) should be ordered by health points then by name. If the pilot has no machines added, print **“no machines”**. If the pilot has **1** machine, print **“1 machine”** and list it. It is allowed to add the same machine more than once. Machine’s targets are separated by “, “ (comma + space). If a machine does not have targets, print **“None”**.All double type fields should be printed “**as is**”, without any formatting or rounding.

The IMachine.ToString() method returns the information about a course in the following form:

|  |
| --- |
| **- *(machine name)***  **\*Type: *(“Tank”/”Fighter”)***  **\*Health: *(machine health points)***  **\*Attack: *(machine attack points)***  **\*Defense: *(machine defense points)***  **\*Targets: *(machine target names/”None” – comma separated)***  **\*Defense: *(“ON”/”OFF” – when applicable)***  **\*Stealth: *(“ON”/”OFF” – when applicable)*** |

The Type is either “Tank“ or “Fighter”. The defense and stealth are only shown when applicable and should print “**ON**” or “**OFF**” depending on their state. The attack targets added to a certain machine (though the Attack(…) method) should appear in the order of their addition. It is allowed to attack one machine (thus adding it) more than once. If the machine has no targets added to it, print “**None**”. Do not print ,(comma) at the line end. All double type fields should be printed “**as is**”, without any formatting or rounding.

All properties in the above interfaces are mandatory (cannot be null or empty). Keep in mind that a machine can be manufactured without a pilot but **cannot** be engaged with **null** pilot afterwards.

If a null value is passed to some mandatory property, your program should throw a proper exception.

### Additional Notes

To simplify your work you are given a virtual war machines factory execution engine that executes a sequence of commands read from the console using the classes and interfaces in your project (see the fileWarMachinesSkeleton.rar). Please put your classes in namespace **WarMachines.Machines**. Implement the**MachineFactory** class in the namespace **WarMachines.Engine**.

You are only **allowed to write classes**. You are **not allowed to modify the existing interfaces and classes except the MachineFactory class**.

Current implemented commands the engine supports are:

* **HirePilot(name)** – adds a pilot with given name. Duplicate names are not allowed. As a result the command returns “**Pilot (name) hired**”.
* **Report (name)** – searches for a hired pilot with given name and returns the **IPilot.Report()** method result.
* **ManufactureTank (name) (attack) (defense)**– creates a tank with given name, attack and defense points. Duplicate names are not allowed. Initial health points are always 100. Initial defense mode is turned on. As a result the command returns “**Tank (name) manufactured - attack: (attack); defense: (defense)**”.
* **DefenseMode (name)** – searches for tank with given name and toggles its defense mode.As a result the command returns “**Tank (name) toggled defense mode**”.
* **ManufactureFighter (name) (attack) (defense) (stealth)** – creates a fighter with given name, attack and defense points. Duplicate names are not allowed. Initial health points are always 200. As a result the command returns **“Fighter (name) manufactured - attack: (attack); defense: (defense); stealth: (ON/OFF)”**
* **StealthMode (name)** - searches for fighter with given name and toggles its stealth mode. As a result the command returns “**Fighter (name) toggled stealth mode**”.
* **Engage (pilot-name) (machine-name)** –searches for a pilot and machine by given names, adds the machine to the pilot’s list of machines and initialises the machine’s pilot. As a result the command returns “**Pilot (pilot-name) engaged machine (machine-name)**”.
* **Attack (attacking-machine-name) (defending-machine-name)** – searches for two machines by given names and the first one attacks the second one if it is possible. As a result the command returns “**Machine (defending-machine-name) was attacked by machine (attacking-machine-name) - current health: (defending-machine-health).**

In case of invalid operation or error, the engine returns appropriate text messages.

### Sample Input

|  |
| --- |
| **HirePilot John**  **HirePilot Nelson**  **Report Bender**  **ManufactureTank T-72 100 100**  **ManufactureFighter Kingcobra 150 90 StealthON**  **Report John**  **Engage John T-72**  **Engage John Kingcobra**  **Report John**  **Report Nelson**  **Engage Nelson T-72**  **Engage Nelson Kingcobra**  **ManufactureFighter Boeing 180 90 StealthOFF**  **Engage Nelson Boeing**  **Attack T-72 Kingcobra**  **Attack T-72 Boeing**  **DefenseMode T-72**  **DefenseMode Kingcobra**  **DefenseMode Boeing**  **Attack T-72 Kingcobra**  **Attack T-72 Boeing**  **StealthMode Kingcobra**  **StealthMode Boeing**  **StealthMode T-72**  **Attack Kingcobra T-72**  **Attack Boeing T-72**  **Report Nelson**  **Report John** |

### Sample Output

|  |
| --- |
| **Pilot John hired**  **Pilot Nelson hired**  **Pilot Bender could not be found**  **Tank T-72 manufactured - attack: 100; defense: 100**  **Fighter Kingcobra manufactured - attack: 150; defense: 90; stealth: ON**  **John - no machines**  **Pilot John engaged machine T-72**  **Pilot John engaged machine Kingcobra**  **John - 2 machines**  **- T-72**  **\*Type: Tank**  **\*Health: 100**  **\*Attack: 60**  **\*Defense: 130**  **\*Targets: None**  **\*Defense: ON**  **- Kingcobra**  **\*Type: Fighter**  **\*Health: 200**  **\*Attack: 150**  **\*Defense: 90**  **\*Targets: None**  **\*Stealth: ON**  **Nelson - no machines**  **Machine T-72 is already occupied**  **Machine Kingcobra is already occupied**  **Fighter Boeing manufactured - attack: 180; defense: 90; stealth: OFF**  **Pilot Nelson engaged machine Boeing**  **Tank T-72 cannot attack stealth fighter Kingcobra**  **Machine Boeing was attacked by machine T-72 - current health: 200**  **Tank T-72 toggled defense mode**  **Machine Kingcobra does not support this operation**  **Machine Boeing does not support this operation**  **Tank T-72 cannot attack stealth fighter Kingcobra**  **Machine Boeing was attacked by machine T-72 - current health: 190**  **Fighter Kingcobra toggled stealth mode**  **Fighter Boeing toggled stealth mode**  **Machine T-72 does not support this operation**  **Machine T-72 was attacked by machine Kingcobra - current health: 50**  **Machine T-72 was attacked by machine Boeing - current health: 0**  **Nelson - 1 machine**  **- Boeing**  **\*Type: Fighter**  **\*Health: 190**  **\*Attack: 180**  **\*Defense: 90**  **\*Targets: T-72**  **\*Stealth: ON**  **John - 2 machines**  **- T-72**  **\*Type: Tank**  **\*Health: 0**  **\*Attack: 100**  **\*Defense: 100**  **\*Targets: Boeing, Boeing**  **\*Defense: OFF**  **- Kingcobra**  **\*Type: Fighter**  **\*Health: 200**  **\*Attack: 150**  **\*Defense: 90**  **\*Targets: T-72**  **\*Stealth: OFF** |

## Problem 2 – Trade and TravelAPI

You are given an API, which supports interactions between different actors (people) and items, occurring in different locations. You are also given a C# file, which has a Main method and uses the API for processing commands from the input.

There are some simple rules the API supports:

* Everything is an object
  + Every object has a name
* Every object is at some location (items are sometimes “inside” a person’s inventory and are then considered as not having a location)
* Locations are specified by names and can be several types (e.g. town)
* A Person can have items and money (every person has “100 money” initially)
  + The items a person has are referred to as his “inventory”
  + A person can drop all of his items at a location (at that moment, any other person can take them)
  + A person can pick up all items at a location
* A Person can be a Shopkeeper, enabling him to sell things for money
  + A Person can also sell things to a Shopkeeper
  + Any Person can fall in debt – that is, have less than 0 money
* A Person can be a Traveller, enabling him to move from one location to the other
* There can be several types of items, the API currently has “armor” implemented
* Items have “value”. Value is what determines the amount of money is spent when buying/selling an item
  + Shopkeepers have the right to determine at what price they sell or buy items
* There can be several types of locations, the API currently has “town” implemented

### Commands

There are two types of commands the Engine supports:

* Creation commands – create items, people or locations
  + Creating locations requires a location type and location name
  + Syntax: create location town *sofia*
  + Creating items requires an item type, item name and location name
  + Syntax: create item armor *coolarmorsofia*– creates an armor type item, named “coolarmor” at location “sofia”
  + Creating people requires a person type, person name and location name
  + Syntax: create traveller *Nelson sofia*– creates a traveller type of Person, with the name of Nelson
* Person commands – order a person to move, buy, sell, drop, pick up items, etc.
  + Person commands start with the person’s name and continue with the type of command
  + A Person can list his inventory
  + Syntax: Joro inventory – outputs all the names of the items in Joro’s inventory
  + A Person can show his money
  + Syntax: Joro money
  + A Person can drop all his items, leaving his inventory empty
  + Syntax: Joro drop
  + A Person can pick up all items at his location, placing them in his inventory
  + Syntax: Joro pickup
  + A Person can travel from one location to another, if he is created as a traveller
  + Syntax: Joro travel Gabrovo
  + A Shopkeeper can be bought from or sold to
  + Syntax: Joro buy coolarmor NikiTheShopman – Joro buys the “coolarmor” item from NikiTheShopman, who is a shopkeeper
    - Joro and NikiTheShopman must be in the same location for this to happen
    - NikiTheShopman must have an item named “coolarmor” for this to happen
  + Syntax: Joro sell jorosarmor NikiTheShopman – Joro sells his “jorosarmor” item to NikiTheShopman
    - Analogous rules to the “buy” command

### Tasks

You are tasked with extending the API by implementing several commands and object types. You are **not allowed to edit any existing class from the original code of the API**. You **are allowed to edit the Main method**.

* Implement a command to create a Weapon item
  + The Weapon item has a moneyvalue of 10
  + Syntax: **create item weapon *weaponname location*** - creates a weapon with the given name, at the given location
* Implement a command to create a Wood item
  + The Wood item has a money value of 2
  + The Wood item decreases its value each time it is dropped by 1, until it reaches 0
  + Syntax: **create item wood *woodname location***
* Implement a command to create an Iron item
  + The Iron item has a money value of 3
  + Syntax: **create item iron *ironname location***
* Implement a command to create a Mine location
  + Syntax: **create location mine *BobovDol***– creates a location, which is a mine with the name of *BobovDol*
* Implement a command to create a Forest location
  + Syntax: **create location forest *Cherokee*** – creates a location, which is a forest, with the name *Cherokee*
* Implement a “gather” command
  + Gathering means a Person takes an item from a special location
  + A Person should be able to gather from mines and from forests
  + A Person can gather from a forest only if he has a Weapon in his inventory
    - Gathering from a forests results in adding a Wood item in the Person’s inventory
  + A Person can gather from a mine only if he has an Armor in his inventory
    - Gathering from a mine results in adding an Iron item in the Person’s inventory
  + Syntax: **Joro gather*newItemName***– gathers an item, naming it *newItemName*if the Person *Joro* is at a mine or forest, and respectively has an Armor or Weapon
* Implement a “craft” command
  + A Person can craft items, provided he has some items in his inventory
  + A Person should be able to craft Weapons and Armor
  + Crafting an Armor requires that the Person has Iron in his inventory
    - Results in adding an Armor item in the Person’s inventory
  + Crafting a Weapon requires that the Person has Iron and Wood in his inventory
  + Syntax: **Joro craft *newItemName***- gathers an item, naming it *newItemName* if the Person *Joro* has the necessary
* Implement a command to create a Merchant
  + A merchant is a Shopkeeper, supporting all of the shopkeeper’s abilities, but can also travel from one location to another
  + Syntax: **create merchant *Joro sofia***–creates a merchant with the name *Joro* at the location *sofia*

### Input and Output Data

You should not concern yourself with handling input and output data – the engine does it for you. You should only consider how to implement the required commands. See the existing API code for hints. Also:

* The names in the commands will always consist of upper and lowercase English letters only.
* In the input, all locations will be created before all other objects
* If for some reason a command is illegal (i.e. trying to sell to someone in a different location), just skip it

|  |  |
| --- | --- |
| Sample Input | Sample Output |
| **create location town whiterun**  **create location town riften**  **create location mine cidna**  **create location forest blackmarsh**  **create item armor theArmor whiterun**  **create item weapon Axe blackmarsh**  **create item armor MineClothes blackmarsh**  **create traveller pesho whiterun**  **create merchant kiro whiterun**  **pesho inventory**  **pesho money**  **pesho pickup**  **pesho inventory**  **pesho travel riften**  **pesho drop**  **create shopkeeper joro riften**  **joro pickup**  **joro inventory**  **pesho buy theArmor joro**  **pesho money**  **pesho sell theArmor joro**  **pesho inventory**  **kiro travel riften**  **kiro buy theArmor joro**  **pesho buy theArmor kiro**  **kiro money**  **kiro travel blackmarsh**  **kiro gather x**  **kiro inventory**  **kiro pickup**  **kiro gather gatheredAtBlackmarsh**  **kiro travel cidna**  **kiro gather gatheredAtCidna**  **kiro inventory**  **kiro craft weapon craftedWeapon**  **kiro craft armor craftedArmor**  **kiro inventory**  **end** | **empty**  **100**  **theArmor**  **theArmor**  **95**  **empty**  **100**  **empty**  **Axe**  **MineClothes**  **gatheredAtBlackmarsh**  **gatheredAtCidna**  **Axe**  **MineClothes**  **gatheredAtBlackmarsh**  **gatheredAtCidna**  **craftedWeapon**  **craftedArmor** |