## C# OOP Exam - 14 August 2022

## Overview

The Universe has entered one of its darkest eras. The planets are running out of resources and the species are suffering from starvation. New Commanders are leading powerful military units and arm their planets with devastating weapons. The war for the last energy sources will be ruthless.

## Setup

* Upload **only the** PlanetWarsproject 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
* **Exception messages** and **output messages** can be found in the **"Utilities"** folder.
* For solving this problem use **Visual Studio 2019 or Visual Studio 2022,** and **netcoreapp 3.1.**

## Task 1: Structure (50 points)

You are given **5** interfaces, and you have to implement their functionality in the **correct classes**.

There are **3** types of entities in the application: **Weapon,** **MilitaryUnit and Planet**. There should also be a **WeaponRepository, UnitRepository** as well as **PlanetRepository**.

### Weapon

Weapon is a **base class** of any **type of weapon** and it **should not be able to be instantiated**. Every **Planet** can possess only one **Weapon** from each type in it’s collection of weapons.

#### Data

* **Price – double -** in billions QUID (Quasi Universal Intergalactic Denomination)
* **DestructionLevel – int**
  + The destruction level is a value between 1 and 10.
  + If the level is below **1,** throw an **ArgumentException** with a message:

"Destruction level cannot be zero or negative."

* + If **exceeds 10,**  throw an **ArgumentException** with a message: "Destruction level cannot exceed 10 power points."

#### Constructor

The constructor should take the following values upon initialization:

int destructionLevel, double price

#### Child Classes

There are three concrete types of Weapons:

##### BioChemicalWeapon

**Price** of **3.2 billion QUID**.

The constructor should take the following values upon initialization: int destructionLevel

##### NuclearWeapon

**Price** of **15 billion QUID**.

The constructor should take the following values upon initialization: int destructionLevel

##### SpaceMissiles

**Price** of **8.75 billion QUID**.

The constructor should take the following values upon initialization: int destructionLevel

### MilitaryUnit

MilitaryUnit is a **base class** of any **type of military unit** and it **should not be able to be instantiated**. Every **Planet** can possess only one **MilitaryUnit** from each type in it’s collection of military units.

#### Data

* **Cost – double -** in billions QUID (Quasi Universal Intergalactic Denomination)
* **EnduranceLevel** **– int**
  + The initial value of every **MilitaryUnit** is equal to 1 power point.

#### Behavior

##### void IncreaseEndurance()

* Increases the endurance of the unit by 1 power point.
* If **exceeds 20,** set the level to 20 and throw an **ArgumentException** with a message: "Endurance level cannot exceed 20 power points."

#### Constructor

The constructor should take the following values upon initialization:

double cost

#### Child Classes

There are three concrete types of MilitaryUnits:

##### StormTroopers

**Cost** of **2.5 billion QUID**.

The constructor should not take any values upon initialization.

##### SpaceForces

**Cost** of **11 billion QUID**.

The constructor should not take any values upon initialization.

##### AnonymousImpactUnit

**Cost** of **30 billion QUID**.

The constructor should not take any values upon initialization.

### Planet

The **Planet** is a class that holds information about the budget of the planet, the army and the military equipment. **It should** be able to be **instantiated**.

You need to keep track of some things, this is why you need some private fields in your Planet class:

* **units - UnitRepository**
* **weapons - WeaponRepository**

#### Data

* **Name – string**
  + If the name **is null or whitespace,** throw an **ArgumentException** with message: "Planet name cannot be null or empty."
* **Budget** - **double**
  + The initial available budget in billions QUID (Quasi Universal Intergalactic Denomination)
  + If the budget is less than **0,** throw an **ArgumentException** with a message:

"Budget's amount cannot be negative."

* **MilitaryPower – double, rounded to the third decimal places.** A calculated property. In order to calculate precise value, follow the sequence of the following operations:
  + **Total amount = (sum of unit endurances + sum of weapon destruction levels)**
  + If the planet has **AnonymousImpactUnit** in its military units (**Army**), **total amount** increases with 30%
  + If the planet has **NuclearWeapon** in its **Weapons** , **total amount** increases with 45%
  + First check for **AnonymousImpactUnit** and then for **NuclearWeapon**
  + Remember to keep the property’s setter private

HINT: The property may be calculated in a separate private method. In order to fulfill the requirements, use **Math.Round(value, 3)** for the returned value.

##### IReadonlyCollection<IMilitaryUnit> Army

Returns a collection of military units (UnitRepository)

##### IReadonlyCollection<IWeapon> Weapons

Returns a collection of weapons (WeaponRepository)

### Behavior

##### void AddUnit(IMilitaryUnit unit)

Adds new **MilitaryUnit** to the **Army**.

##### void AddWeapon(IWeapon weapon)

Adds new **Weapon** to the **Weapons**.

##### void TrainArmy()

The **TrainArmy()** method should increase the **EnduranceLevel** of all forces in the **Army** by 1 power point.

##### void Spend(double amount)

The **Spend()** method should decrease the **Budget** by the given amount.

* If the **Budget** is not enough to finish the purchase, throw an **InvalidOperationException** with a message: "Budget too low!"

##### void Profit(double amount)

The **Profit()** method should increase the **Budget** by the given amount.

##### string PlanetInfo()

**Returns** a **string** with **information** about the **planet** in the format below:

"Planet: {planetName}

--Budget: {budgetAmount}billion QUID  
--Forces: {militaryUnitName1}, {militaryUnitName2}, {militaryUnitName3} (…) / No units

--Combat equipment: {weaponName1}, {weaponName2}, {weaponName3} (…) / No weapons  
--Military Power: {militaryPower}"

**Note: Do not use** "\r\n" **for a new line.**

#### Constructor

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

string name, double budget

### WeaponRepository

The **WeaponRepository** is a **class** which represents collection of weapons.

#### Data

* Models – IReadOnlyCollection<IWeapon>
* Some private field might be helpful

#### Behavior

**IWeapon FindByName(string weaponTypeName)**

* Returns a weapon with the given type name, if it exists. If it doesn't, returns null.

##### void AddItem(IWeapon weapon)

* Adds new weapon to the repository.

**bool RemoveItem(string weaponTypeName)**

* Removes a weapon with the given typeName from the collection. Returns **true** if the deletion was sucessful. Otherwise returns **false.**

#### Constructor

### The constructor should not take any values upon initialization.

### UnitRepository

The **UnitRepository** is a **class** which represents collection of military units.

#### Data

* Models – **a** **collection IMilitaryUnit (unmodifiable)**
* Some private field might be helpful

#### Behavior

**IMilitaryUnit FindByName(string unitTypeName)**

* Returns a unit with the given type name, if it exists. If it doesn't, returns null.

##### void AddItem(IMilitaryUnit unit)

* Adds a unit to the repository.

**bool RemoveItem(string unitTypeName)**

* Removes an unit with the given typeName from the collection. Returns **true** if the deletion was sucessful. Otherwise returns **false.**

#### Constructor

### The constructor should not take any values upon initialization.

### PlanetRepository

The **PlanetRepository** is a **class** which represents collection of planets.

#### Data

* Models – **a** **collection IPlanet (unmodifiable)**
* Some private field might be helpful

#### Behavior

**IPlanet FindByName(string name)**

* Returns a planet with the given name, if it exists. If it doesn't, returns null.

##### void AddItem(IPlanet planet)

* Adds a planet to the repository.

**bool RemoveItem(string planetName)**

* Removes a planet with the given name from the collection. Returns **true** if the deletion was sucessful. Otherwise returns **false.**

#### Constructor

### The constructor should not take any values upon initialization.

## 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!**

**NOTE:** When you create the Controllerclass, go into the **Engine** class constructor and uncomment the "this.controller = new Controller();" line.

The first interface is **I**Controller. You must create a Controllerclass, which implements the interface and implements all of its methods. The constructor of Controller does not take any arguments. The given methods should have the following logic:

### Data

You need to keep track of some things, this is why you need a private field in your controller class:

* **planets - PlanetRepository**

### Commands

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

#### CreatePlanet Command

##### Parameters

* planetName - string
* budget – double

##### Functionality

Creates a **planet** with the provided **name** and **budget**.

* If a Planet with the same name is already created, return the following message: "Planet {planetName} is already added!"
* If the planet is valid, keep it in the repository of planets and **return** the following message: "Successfully added Planet: {planetName}".

#### AddUnit Command

##### Parameters

* unitTypeName – string
* planetName - string

##### Functionality

##### Creates a MilitaryUnit from the given type and adds it to the Army of the Planet with the given name. Every unit is unique. A Planet can have only one MilitaryUnit from a specific type:

##### If a Planet with the given name doesn’t exist in the PlanetReposotiry, throw an InvalidOperationException with the following message: "Planet {planetName} does not exist!"

* If the **MilitaryUnit** is not available in our application (no such type of MilitaryUnit exists in the child classes), throw an **InvalidOperationException** with the following message: "{unitTypeName} still not available!"

##### If the same MilitaryUnit is already added, throw an InvalidOperationException with the following message: "{unitTypeName} already added to the Army of {planetName}!"

* If the **MilitaryUnit** is valid, add it to the UnitRepository of the planet. Planet’s **Budget** is reduced with the **price of the unit** and the following message is returned: "{unitTypeName} added successfully to the Army of {planetName}!"

#### AddWeapon Command

##### Parameters

* planetName – string
* weaponTypeName – string
* destructionLevel - int

##### Functionality

##### Creates a Weapon from the given type and adds it to the Weapons of the Planet with the given name. Every weapon is unique. A Planet can have only one Weapon from a specific type:

##### If a Planet with the given name doesn’t exist in the PlanetRepository, throw an InvalidOperationException with the following message: "Planet {planetName} does not exist!"

##### If the same Weapon is already added, throw an InvalidOperationException with the following message: "{weaponTypeName} already added to the Weapons of {planetName}!"

* If the **Weapon** is not available in our application (no such type of Weapon exists in the child classes), throw an **InvalidOperationException** with the following message: "{weaponTypeName} still not available!"
* If the **Weapon** is valid, add it to the WeaponRepository of the planet. Planet’s **Budget** is reduced with the **price of the weapon** and the following message is returned: "{planetName} purchased {weaponTypeName}!"

#### SpecializeForces Command

##### Parameters

* planetName – string

##### Functionality

##### Increases the EnduranceLevel of the Army of the specific Planet:

##### If a Planet with the given name doesn’t exist, throw an InvalidOperationException with the following message: "Planet {planetName} does not exist!"

* If there are no Military units added still, throw an **InvalidOperationException** with the following message: "No units available for upgrade!"
* If the action is completed successfully, reduce the **Budget** by 1.25 billion QUID, train the army of the given Planet and return the following message: "{planetName} has upgraded its forces!".

#### SpaceCombat Command

##### Parameters

* firstPlanetName – string
* secondPlanetName – string

##### Functionality

##### The first planet declares war to the second. You will receive only planets, which are added to the universe, already. The planet with the bigger MilitaryPower wins the war:

* If both have the same **MilitaryPower** the winner is the **Planet** which owns **NuclearWeapon**
* If both have the same **MilitaryPower** and both **own** **NuclearWeapon** ORboth do **NOT own NuclearWeapon**, no one wins the war and **both lose** half of their **Budget**. The following message should be returned: "The only winners from the war are the ones who supply the bullets and the bandages!"

##### The winner loses half of its own Budget. Then takes half of the Budget left from the losing planet. The winner also adds the sum of all forces costs and weapons prices possessed by the losing planet to its Budget. The losing planet is deleted from the PlanetRepository of the Universe. The following message is returned: "{winningPlanetName} destructed {losingPlanetName}!"

#### ForcesReport Command

##### Functionality

Returns information about the military state in the **Universe**. You can use the **PlanetInfo** method. The planets must be ordered in descending order, by their **MilitaryPower**. Then by their names in ascending alphabetical order. Military units and weapons are arranged in the order of purchasing.

\*\*\*UNIVERSE PLANET MILITARY REPORT\*\*\*

Planet: {planetName1}

--Budget: {budgetAmount}billion QUID  
--Forces: {militaryUnitTypeName1}, {militaryUnitTypeName2}, {militaryUnitTypeName3} (…) / No units

--Combat equipment: {weaponTypeName1}, {weaponTypeName2}, {weaponTypeName3} (…) / No weapons  
--Military Power: {militaryPower}

Planet: {planetName2}

--Budget: {budgetAmount}billion QUID  
--Forces: {militaryUnitTypeName1}, {militaryUnitTypeName2}, {militaryUnitTypeName3} (…) / No units

--Combat equipment: {weaponTypeName1}, {weaponTypeName2}, {weaponTypeName3} (…) / No weapons  
--Military Power: {militaryPower}

…

Planet: {planetNameN}

--Budget: {budgetAmount}billion QUID  
--Forces: {militaryUnitTypeName1}, {militaryUnitTypeName2}, {militaryUnitTypeName3} (…) / No units

--Combat equipment: {weaponTypeName1}, {weaponTypeName2}, {weaponTypeName3} (…) / No weapons  
--Military Power: {militaryPower}

**Note: Do not use** "\r\n" **for a new line.**

**Hint: You can use StringBuilder.AppendLine()**

#### Peace 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.

#### Input

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

* **CreatePlanet** **{planetName} {budget}**
* **AddUnit** **{unitTypeName} {planetName}**
* **AddWeapon** **{planetName} {weaponTypeName} {destructionLevel}**
* **SpecializeForces {planetName}**
* **SpaceCombat {firstPlanetName} {secondPlanetName}**
* **ForcesReport**
* **Peace**

#### 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** |
| CreatePlanet Zomitune 300  CreatePlanet Cippe-C77T 270  CreatePlanet Xunrichi 20.52  AddUnit StormTroopers Dotrienides  AddUnit SpaceForces Zomitune  AddUnit SpaceForces Zomitune  AddWeapon Xunrichi SpaceMissiles 2  AddWeapon Xunrichi BioChemicalWeapon 6  AddWeapon Xunrichi NuclearWeapon 8  AddWeapon Cippe-C77T NuclearWeapon 11  AddWeapon Cippe-C77T NuclearWeapon 10  SpecializeForces Zomitune  AddUnit AnonymousImpactUnit Xunrichi  AddUnit StormTroopers Xunrichi  SpaceCombat Xunrichi Zomitune  ForcesReport  Peace |
| **Output** |
| Successfully added Planet: Zomitune  Successfully added Planet: Cippe-C77T  Successfully added Planet: Xunrichi  Planet Dotrienides does not exist!  SpaceForces added successfully to the Army of Zomitune!  SpaceForces already added to the Army of Zomitune!  Xunrichi purchased SpaceMissiles!  Xunrichi purchased BioChemicalWeapon!  Budget too low!  Destruction level cannot exceed 10 power points.  Cippe-C77T purchased NuclearWeapon!  Zomitune has upgraded its forces!  Budget too low!  StormTroopers added successfully to the Army of Xunrichi!  Xunrichi destructed Zomitune!  \*\*\*UNIVERSE PLANET MILITARY REPORT\*\*\*  Planet: Cippe-C77T  --Budget: 255 billion QUID  --Forces: No units  --Combat equipment: NuclearWeapon  --Military Power: 14.5  Planet: Xunrichi  --Budget: 157.91 billion QUID  --Forces: StormTroopers  --Combat equipment: SpaceMissiles, BioChemicalWeapon  --Military Power: 9 |

## Task 3: Unit Tests (100 points)

You will receive a skeleton with **Planet** and **Weapon** classes inside. The classes will have some methods, fields and one constructor, which are working properly. You are **NOT ALLOWED** to change any class. Cover the both classes with unit tests to make sure that the class is working as intended.

You are provided with a **unit test project** in the **project skeleton**.

Do **NOT** use **Mocking** in your unit tests!