C# OOP Exam - 11 August 2019

1. Overview

GTA Vice city is one of the greatest games ever. It's a action-adventure game with main player Tommy Vercetti. You have been asked from "Rockstar Games" to create an extension. Your task is to create a entity which will be tha main player and one more which will be the same - civil player. You GTA witout guns is very boring, so that's why you need to crete enities which will represent guns. Finally the fight is happen in the gangsta neighbourhood.

2. Setup

- Upload **only the `ViceCity`** project 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

3. Task 1: Structure (50 points)

You are given **4** interfaces, and you have to implement their functionality in the **correct classes**.

There are **4** types of entities in the application: **`Player`, `Gun`, `Neighborhood`** and **`GunRepository`**:

Player

`Player` is a **base class** or any **type of player** and it **should not be able to be instantiated**.

Data

- **`Name`** – **`string`**
  - If the username **is null or whitespace,** throw an **`ArgumentNullException`** with message: `"Player's name cannot be null or a whitespace!"`
  - All names are unique
- **`LifePoints`** – **`int`**
  - The health of a player
  - If the health is below **0,** throw an **`ArgumentException`** with message: `"Player life points cannot be below zero!"`
- **`GunRepository - IRepository<Gun>`**
  - Generic repository of all **player's** guns
- **`IsAlive –`** calculated property, which returns **bool**

Behavior

`void TakeLifePoints(int points)`

The **`TakeLifePoints`** method decreases players' life points.

- Player's life points should not drop below zero

Constructor

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

```
string name, int lifePoints
```

Child Classes

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

MainPlayer

Has **100 initial life points** and the main player has only one name **"Tommy Vercetti".** The constructor should not take name and life points values upon initialization.

CivilPlayer

Has **50 initial life points**.

Constructor should take the following values upon initialization:

```
string name
```

Gun

The **Gun** is a base class for any type of gun and it should not be able to be instantiated.

Data

- **`Name – string`**
  - If the name of the gun **is null or empty,** throw an **`ArgumentException`** with message: `"Name cannot be null or a white space!"`
  - All names are unique
- **`BulletsPerBarrel – int`**
  - If the bullets are **below zero,** throw an **`ArgumentException`** with message: `"Bullets cannot be below zero!"`
  - The **initial `BulletsPerBarrel` count** is the actual **capacity** of the **barrel**!
- **`TotalBullets - int`**
  - If the total bullets are **below zero,** throw an **`ArgumentException`** with message: `"Total bullets cannot be below zero!"`
- **`CanFire –`** calculated property, which returns **bool**

Behavior

`int Fire()`

The Fire method acts **different** in all **child classes**. It shoots bullets and returns the number of bullets that were shot. Here is how it works:

- Your guns have a **barrel**, which have a certain **capacity for bullets** and you shoot a different **count of bullets** when you **pull the trigger.**
- **If your barrel becomes empty**, you need to **reload** before you can shoot again.
- **Reloading** is done by refilling the **whole barrel of the gun (Keep in mind, that every barrel has capacity).**
- The bullets you **take for reloading** are **taken** from the gun's **total bullets stock**.

Keep in mind, that every type of gun shoots **different count** of **bullets**, when you **pull the trigger**!

Constructor

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

```
string name, int bulletsPerBarrel, int totalBullets
```

Child Classes

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

Pistol

Has **10 bullets per barrel** and **100 total bullets**.

Behavior

`int Fire()`

The pistol shoots only **one bullet**.

Constructor

Constructor should take the following values upon initialization:

```
string name
```

Rifle

Has **50 bullets** per **barrel** and **500 total bullets**.

Behavior

`int Fire()`

The rifle can shoot with **5 bullets**.

Constructor

Constructor should take the following values upon initialization:

```
string name
```

GangNeighbourhood

The gang neighbourhood is the place where the shooting happens. It should inherit the **`INeighbourhood`** interface.

Behavior

`void Action(IPlayer mainPlayer, ICollection<IPlayer> civilPlayers)`

That's the most interesting method.

The main player starts shooting at all the civil players. When he starts shooting at a civil player, the following **rules** apply:

- He takes a gun from his guns.
- Every time he shoots, he **takes life points** from the civil player, which are **equal** to the **bullets that the current gun** shoots when the trigger is pulled.
- If the **barrel of his gun becomes empty**, he **reloads** from his **bullets stock** and **continues shooting** with the **current gun**, **until he uses all of its bullets**.
- If his **gun runs out** of total **bullets**, he takes **the next gun** he has and **continues** shooting.
- He shoots at the **current civil** player **until he / she is alive**.
- If the civil player **dies**, he starts **shooting at the next one**.
- The **main player stops shooting only if he runs out of guns** or **until all the civil players are dead**.

The civil players (**the ones that have stayed alive after the main player's attack**) attack second. They start shooting at him **one after another** and the following **rules** apply:

- A civil player takes one of his guns and starts shooting at the main player.
- Every time he shoots, he **takes life points** from the main player, which are **equal** to the **bullets that the current gun shoots** when the trigger is pulled.
- If the **barrel of his gun becomes empty**, he **reloads** from his **bullets stock** and **continues shooting** with the **current gun**, **until he uses all of its bullets**.
- If his current **gun runs out of all its bullets**, he takes **the next gun** he has and **continues** shooting.
- If a **civil player** runs out of **guns**, the **next civil player begins shooting**.
- If the main player **dies**, the whole **action** ends.

GunRepository

The gun repository holds information about a player's guns.

Data

- **`Models` – collection of guns (unmodifiable)**

Behavior

`void Add(IGun model)`

- Adds a gun in the collection.
- If the gun already exists in the player's collection of guns, don't add it.
- Every gun is unique.

`bool Remove(IGun model)`

- Removes a gun from the collection.

`IGun Find(string name)`

- Returns a gun with that name.
- It is guaranteed that the guns exists in the collection.

4. 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 **`IController`**. You must create a **`Controller`** class, which implements the interface and implements all of its methods. The constructor of **`Controller`** does not take any arguments. When a controller is initialized, the main player is created. 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.

AddPlayer Command

Parameters

- **Name – string**

Functionality

Creates a civil player with the given name.

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

- `"Successfully added civil player: {civilPlayerName}!"`

AddGun Command

Parameters

- **Type - string**
- **Name - string**

Functionality

Creates a **gun** with the provided **type** and **name**.

If the gun type is invalid, the method should return the following message:

- `"Invalid gun type!"`

If the gun type is correct, keep the gun and **return** the following message:

- `"Successfully added {name} of type: {type}"`.

AddGunToPlayer Command

Parameters

- **Name – string (player's name)**

Functionality

Adds the first added gun to the **player's** gun **repository**.

- If there no guns in the queue, return the following message:

  `"There are no guns in the queue!"`

- If the name of the player is **"Vercetti"**, you need to add the gun to the main player's repository and return the following message:

  `"Successfully added {gunName} to the Main Player: Tommy Vercetti"`

- If you receive a name which doesn't exist, you should return the following message:

  `"Civil player with that name doesn't exists!"`

- If everything is successful, you should add the gun to the player's repository and return the following message:

  `"Successfully added {gunName} to the Civil Player: {playerName}"`

Fight Command

Functionality

When the fight command is called, the action happens. You should start the action between the main player and all the civil players. When a civil player is killed, it should be removed from the repository.

- If the main player still has all of his points and no one is dead or harmed from the civil players, you should return the following messages:

  ```
  "Everything is okay!"
  ```

- If any of the players has different life points, you should return the following message:

  ```
  "A fight happened:"
  "Tommy live points: {mainPlayerLifePoints}!"
  "Tommy has killed: {deadCivilPlayers} players!"
  "Left Civil Players: {civilPlayersCount}!"
  ```

Note: Use \r\n or Environment.NewLine for a new line.

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.

Input

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

- **`AddPlayer {player username}`**
- **`AddGun {gun type} {gun name}`**
- **`AddGunToPlayer {player name}`**
- **`Fight`**

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
AddGun Pistol Colt **AddGun Rifle SniperRifle** **AddPlayer Alfie** **AddPlayer Alexis** **AddPlayer Bean** **AddPlayer Beck** **AddPlayer Camber** **AddPlayer Burney** **Fight** **AddGunToPlayer Vercetti** **AddGunToPlayer Vercetti** **AddGunToPlayer Vercetti** **Fight** **Exit**

Output

```
Successfully added Colt of type: Pistol
Successfully added SniperRifle of type: Rifle
Successfully added civil player: Alfie!
Successfully added civil player: Alexis!
Successfully added civil player: Bean!
Successfully added civil player: Beck!
Successfully added civil player: Camber!
Successfully added civil player: Burney!
Everything is okay!
Successfully added Colt to the Main Player: Tommy Vercetti
Successfully added SniperRifle to the Main Player: Tommy Vercetti
There are no guns in the queue!
A fight happened:
Tommy live points: 100!
Tommy has killed: 6 players!
Left Civil Players: 0!
```

Input

```
AddGun Pistol Colt
AddGun Pistol ColtPython
AddGun Rifle SniperRifle
AddGun Rifle PSGSniper
AddGun Shotgun Spaz
AddPlayer Alfie
AddPlayer Alexis
AddPlayer Bean
AddPlayer Beck
AddPlayer Camber
AddPlayer Burney
AddGunToPlayer Bean
AddGunToPlayer Vercetti
AddGunToPlayer Alfie
AddGunToPlayer Arthur
AddGunToPlayer Alfie
AddGunToPlayer Burney
Fight
Exit
```

Output

```
Successfully added Colt of type: Pistol
Successfully added ColtPython of type: Pistol
Successfully added SniperRifle of type: Rifle
Successfully added PSGSniper of type: Rifle
Invalid gun type!
Successfully added civil player: Alfie!
Successfully added civil player: Alexis!
Successfully added civil player: Bean!
Successfully added civil player: Beck!
Successfully added civil player: Camber!
Successfully added civil player: Burney!
Successfully added Colt to the Civil Player: Bean
Successfully added ColtPython to the Main Player: Tommy Vercetti
Successfully added SniperRifle to the Civil Player: Alfie
```

```
Civil player with that name doesn't exists!
Successfully added PSGSniper to the Civil Player: Alfie
There are no guns in the queue!
A fight happened:
Tommy live points: 0!
Tommy has killed: 2 players!
Left Civil Players: 4!
```

5. Task 3: Unit Tests (100 points)

You will receive a skeleton with **`Astronaut`** and **`Spaceship`** classes inside. The class will have some methods, fields and one constructor, which are working properly. You are **NOT ALLOWED** to change any class. Cover the whole class 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!