# Zombow

### Tasks

You are given an almost working application.

Use the problem description to get familiar with the project and what it is expected to do.

#### Task 1: Refactor Bow class – 25 pts

Refactor Bow class to follow the OOP principles.

#### Task 2: Find and Fix Bugs – 25 pts

Find all business logic bugs and fix them.

You can use the **input** and **output** sections to check the business logic validity.

#### Task 3: Command Pattern – 50 pts

Implement the [command pattern](https://www.dofactory.com/net/command-design-pattern).

You are given ICommand and ICommandInterpreter interfaces, but you can modify them if you want to.

### Project Overview

### Structure

There are **4** types of entities in the application: **Player, Bow, ZombieHorde** and **BowRepository**:

#### 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'sname cannot be null or a whitespace!"
  + All names are unique
* **LifePoints** – **int**
  + The health of а player
  + If the health is below **0,** throw an **ArgumentException** with message:

"Player life points cannot be below zero!"

* **BowRepository - IRepository<Bow>**
  + Generic repository of all **player's** bows
* 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 **"Andrew".** The constructorshould not take name and life points values upon initialization.

###### Zombie

Has **50 initial life points**.

Constructorshould take the following values upon initialization:

string name

#### Bow

The Bow is a base class for any type of bow and it should not be able to be instantiated.

##### Data

* **Name – string** 
  + If the name of the bow **is null or empty,** throw an ArgumentException with message:

"Name cannot be null or a white space!"

* + All names are unique
* **QuiverCapacity – int**
  + If the bullets are **below zero,** throw an ArgumentException with message:

"Arrows cannot be below zero!"

* + The **initial** **QuiverCapacity** **count** is the actual **capacity**!
* **TotalArrows - int** 
  + If the total bullets are **below zero,** throw an ArgumentException with message:

"Total arrows cannot be below zero!"

* CanShoot – calculated property, which returns **bool**

##### Behavior

###### int Shoot()

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

* Your have a **quiver**, which has a certain **capacity** **for arrows** and you shoot a different **count of arrows** when you **shoot.**
* **If your quiver becomes empty**, you need to **refill** before you can shoot again.
* **Reloading** is done by refilling the **quiver (Keep in mind, that the quiver has capacity).**
* The arrows you **take for reloading** are **taken** from the player’s **total arrows stock**.

Keep in mind, that every type of bow shoots **different count** of **arrows**, when you **shoot**!

##### Constructor

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

string name, int quiverCapacity, int totalArrows

##### Child Classes

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

###### Recurve

Has **10 quiver capacity** and **100 total arrows**.

##### Behavior

###### int Shoot()

The recurve shoots only **one** **arrow**.

##### Constructor

Constructorshould take the following values upon initialization:

string name

###### Takedown

Has **50 quiver capacity** and **500** **total arrows**.

##### Behavior

###### int Shoot()

The takedown can shoot with **5 arrows**.

##### Constructor

Constructorshould take the following values upon initialization:

string name

#### ZombieHorde

The zombie horde is the place where the shooting happens. It should inherit the **IZombieHorde** interface.

##### Behavior

**void Action(IPlayer mainPlayer, ICollection<IPlayer> zombies)**

That's the most interesting method.

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

* He takes a bow from his bows.
* Every time he shoots, he **takes life points** from the zombie, which are **equal** to the **arrows that the current bow** shoots.
* If the **quiver of his bow becomes empty**, he **reloads** from his **arrows stock** and **continues** **shooting** with the **current bow**, **until he uses all of its arrows**.
* If his **bow runs out** of total **arrows**, he takes **the next bow** he has and **continues** shooting.
* He shoots at the **current zombie** **until it is alive**.
* If the zombie **dies**, he starts **shooting at the next one**.
* The **main player stops shooting only if he runs out of bows** or **until all the zombie are dead**.

**The zombies** (**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 zombie takes one of his bows and starts shooting at the main player.
* Every time it shoots, he **takes life points** from the main player, which are **equal** to the **arrows that the current bow shoots**.
* If **its quiver becomes empty**, it **reloads** from its **arrows** stock and **continues** **shooting** with the **current bow**, **until it uses all of its arrows**.
* If its current **bow runs out of all its arrows**, he takes **the next bow** he has and **continues** shooting.
* If a **zombie** runs out of **bows**, the **next zombie begins shooting**.
* If the main player **dies**, the whole **action** ends.

#### BowRepository

The bow repository holds information about a player's bows.

##### Data

* Models – **collection of bows (unmodifiable)**

##### Behavior

**void Add(IBow model)**

* Adds a bow in the collection.
* If the bow already exists in the player's collection of bows, don't add it.
* Every bow is unique.

**bool Remove(IBow model)**

* Removes a bow from the collection.

**IBow Find(string name)**

* Returns a bow with that name.
* It is guaranteed that the bow exists in the collection.

### Business Logic

#### The Controller Class

#### Commands

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

##### AddZombie Command

###### Parameters

* **Name** – **string**

###### Functionality

Creates a zombie with the given name.

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

* "Successfully added zombie: {zombieName}!"

##### AddZombie Command

###### Parameters

* Type - string
* Name - string

###### Functionality

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

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

* "Invalid bow type!"

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

* "Successfully added {name} of type: {type}".

##### AddBowToPlayer Command

###### Parameters

* Name – string (player's name)

###### Functionality

###### Adds the first added bow to the player's bow repository.

* If there no bows in the queue, return the following message:

"There are no bows in the queue!"

###### If the name of the player is "Andrew", you need to add the bow to the main player's repository and return the following message:

"Successfully added {bowName} to the Main Player: Andrew"

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

"Zombie with that name doesn't exists!"

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

"Successfully added {bowName} to the Zombie: {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 zombies. When a zombie 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 zombies, 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:"

"Andrew live points: {mainPlayerLifePoints}!"

"Andrew has killed: {deadZombies} zombies!"

"Left Zombies: {zombiesCount}!"

**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:

* **AddZombie** **{player username}**
* **AddBow** **{bow type} {bow name}**
* **AddBowToPlayer** **{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. Recurve = recurve; takedown = takedown

##### Examples

|  |
| --- |
| **Input** |
| **AddBow Recurve Colt**  **AddBow Takedown SniperTakedown**  **AddZombie Alfie**  **AddZombie Alexis**  **AddZombie Bean**  **AddZombie Beck**  **AddZombie Camber**  **AddZombie Burney**  **Fight**  **AddBowToPlayer Andrew**  **AddBowToPlayer Andrew**  **AddBowToPlayer Andrew**  **Fight**  **Exit** |
| **Output** |
| **Successfully added Colt of type: Recurve**  **Successfully added SniperTakedown of type: Takedown**  **Successfully added zombie: Alfie!**  **Successfully added zombie: Alexis!**  **Successfully added zombie: Bean!**  **Successfully added zombie: Beck!**  **Successfully added zombie: Camber!**  **Successfully added zombie: Burney!**  **Everything is okay!**  **Successfully added Colt to the Main Player: Andrew**  **Successfully added SniperTakedown to the Main Player: Andrew**  **There are no bows in the queue!**  **A fight happened:**  **Andrew live points: 100!**  **Andrew has killed: 6 players!**  **Left Zombies: 0!** |

|  |
| --- |
| **Input** |
| **AddBow Recurve Colt**  **AddBow Recurve ColtPython**  **AddBow Takedown SniperTakedown**  **AddBow Takedown PSGSniper**  **AddBow Shotbow Spaz**  **AddZombie Alfie**  **AddZombie Alexis**  **AddZombie Bean**  **AddZombie Beck**  **AddZombie Camber**  **AddZombie Burney**  **AddBowToPlayer Bean**  **AddBowToPlayer Andrew**  **AddBowToPlayer Alfie**  **AddBowToPlayer Arthur**  **AddBowToPlayer Alfie**  **AddBowToPlayer Burney**  **Fight**  **Exit** |
| **Output** |
| **Successfully added Colt of type: Recurve**  **Successfully added ColtPython of type: Recurve**  **Successfully added SniperTakedown of type: Takedown**  **Successfully added PSGSniper of type: Takedown**  **Invalid bow type!**  **Successfully added zombie: Alfie!**  **Successfully added zombie: Alexis!**  **Successfully added zombie: Bean!**  **Successfully added zombie: Beck!**  **Successfully added zombie: Camber!**  **Successfully added zombie: Burney!**  **Successfully added Colt to the Zombie: Bean**  **Successfully added ColtPython to the Main Player: Andrew**  **Successfully added SniperTakedown to the Zombie: Alfie**  **Zombie with that name doesn't exists!**  **Successfully added PSGSniper to the Zombie: Alfie**  **There are no bows in the queue!**  **A fight happened:**  **Andrew live points: 0!**  **Andrew has killed: 2 players!**  **Left Zombies: 4!** |