OOP Advanced Exam – The Tank Game

Overview

The Tank Game is a competition between vehicles. The competition needs to be automated with a software program. **Stoyan** tried to write some code, but he is not very good and he left bugs here and there. You must finish the job.

Tasks

Task 1: Business Logic

Stoyan tried to write some code before you, but he is clumsy, so he left some bugs in the application… But he somehow managed to write the **BaseVehicle** and **TankBattleOperator** classes and all **interfaces** right.

Your first task is to find and fix all bugs.

Task 2: Input / Output

Task 3: Reflection

You need to refactor the given factories and implement new ones. Factories must use reflection, so it will be easy for us to follow the Open/Closed Principle. You are required to implement two factories:

- **PartFactory**
- **VehicleFactory**

Your task is to implement these factories in such a way that it will be easy to extend the number of concrete types of each entity.

Also, make sure that if you add a new method in the manager class, you won't have to change the **ProcessInput** method in the **CommandInterpreter** class.

NOTE: Make sure you reference the Calling Assembly, instead of the Executing Assembly, since the code that's going to be calling your factories in the tests depends on this!

No static factories are allowed!

Task 4: Unit Testing

Like you saw at the beginning, there is a class, which does not need refactoring - **BaseVehicle.** This is the class, against which you need to **write unit tests**. In your skeleton, you are provided with a **perfectly working BaseVehicle**, but it still needs to be **tested**, because in **Judge**, we have prepared some **bugs**, and you need to catch them in your unit tests.

You are provided with a **unit test project** in the **project skeleton**. **DO NOT modify its NuGet packages**.

Note: The **BaseVehicle** you need to test is in the **global namespace**, as are any entities, which it depends on, so **remove any using statements** pointing towards any entities and controllers before submitting your code.

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

Skeleton

You are allowed to change the **internal** and **private logic** of the **classes** that have been given to you.
In other words, you can change the **body code** and the **definitions** of the **private members** in whatever way you like.

SoftUni Foundation
© Software University Foundation. This work is licensed under the CC-BY-NC-SA license.
Follow us:

However. . .

You are **NOT ALLOWED** to **CHANGE** the **Interfaces** that have been provided by the **skeleton** in **ANY way**.
You are **NOT ALLOWED** to **ADD** more **PUBLIC LOGIC**, than the **one**, **provided** by the **Interfaces**, **ASIDE FROM** the **`ToString()`** method.

Guidelines

- Upload **only the `TheTankGame`** project in every problem **except Unit Tests**. For **Unit Tests**, upload **only the `TheTankGame.Tests` project** with **all using statements pointing to TheTankGame removed**.
- **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.

Below, you will find a detailed description of all entities and their methods.

Structure

The structure of the software circles around **`Vehicles`** and **`Parts`**.

Vehicles

The **`Vehicles`** are initialized with:

- **`Model`** – a **string**.
- **`Weight`** – a **floating-point number**.
- **`Price`** – a **decimal number**.
- **`Attack`** – an **integer**.
- **`Defense`** – an **integer**.
- **`HitPoints`** – an **integer**.

There are generally 2 types of **`Vehicles`**.

Vanguard

A tank-like land vehicle.

Revenger

A jet-like aerial vehicle.

Parts

The **`Parts`** are initialized with:

- **`Model`** – a **string**.
- **`Weight`** – a **floating-point number**.
- **`Price`** – a **decimal number**.

There are generally 3 types of **`Parts`**.

ArsenalPart

The **`ArsenalPart`** is initialized with an additional property:

SoftUni Foundation
© Software University Foundation. This work is licensed under the CC-BY-NC-SA license.
Follow us:

- **`AttackModifier`** – an **integer**.

ShellPart

The **`ShellPart`** is initialized with an additional property:

- **`DefenseModifier`** – an **integer**.

EndurancePart

The **`EndurancePart`** is initialized with an additional property:

- **`HitPointsModifier`** – an **integer**.

Assembler

The **`Assembler`** contains 3 collections – 1 for the **ArsenalParts**, 1 for the **ShellParts**, and 1 for the **EnduranceParts**.

The class exposes **3 methods** for adding Parts – one for the **ArsenalParts,** one for the **ShellParts**, and one for the **EnduranceParts**.

The class also exposes **3 methods** for **extracting** the **total stat modification** each type of parts gives to the **Vehicle**.

BattleOperator

The **`BattleOperator`** class exposes **1 method** for **handling Battles** – the method **accepts 2 Vehicle**s and initiates a Battle between them, ultimately **resulting** in a **winner**. The winner's **`model`** is **returned** as **result** of the **method**.

The 2 Vehicles fight in turns with the **first given Vehicle** being the **first 1** to **attack**.

First, the **attacker attacks**, **then**, the **target attacks back**. Each turn they lose **hitPoints**, due to the attack, by the following formula:

```
receivingVehicleHitPoints -= (attackingVehicleAttack - (receivingVehicleDefense + (receivingVehicleWeight / 2)))
```

As you see the **Defense** and **Weight** of the **`receivingVehicle`** **reduce** the **attack damage** of the **`attackingVehicle`**, which is a normal tactic.

The process of exchanging attacks continues, until one's **hitPoints** is **lower than** or **equal** to **0**.

Functionality

The functionality of the software involves adding **`Vehicles`**, adding **`Parts`** to the **`Vehicles`**, and so on. As you see the **`Vehicles`** and **`Parts`** are the main entities of the program. The **`model`** is the **property** that will **identify** them. The **model** will also, always be **unique** in the input.

In **some** of the **commands**, you'll receive **`models`** which may refer to a **`Vehicle`** and a **`Part`**. You must check what is the object with that **`model`**, and process the command depending on the result.

Each **`Vehicle`** has an **`Assembler`**, in which it **stores** its **`Parts`**.
The business logic of the program involves: adding vehicles and parts, inspecting vehicles and parts, fighting between vehicles.

Check below, each section, and the functionality it describes.

Vehicles

The **`Vehicles`** are the main actors in the business logic. They have **stats** which **define** their **power**. Those **stats** can be **upgraded** by **adding parts** to them, which is done through the **`Assembler`**.

© Software University Foundation. This work is licensed under the CC-BY-NC-SA license.
Follow us:

Battles are triggered **between 2 Vehicles**. The **resulting winner** of the battle, should **stay** in the data, while the loser should be **removed**.

Battles are controlled by the **`BattleOperator`**. When 2 Vehicles are passed to the **`BattleOperator`**, it **returns** the **`model`** of the **winning vehicle**. You should consider that in your logic.

Parts

The **`Parts`** have no business logic around themselves. They are just **data models**.

Commands

There are several commands which are given from the user input, in order to control the program.
Here you can see how they are formed.

The **parameters** will be given in the **EXACT ORDER**, as the one **specified below**.
You can see the exact input format in the **Input section**.

Each command will **generate an output result**, which you must **print**.
You can see the exact output format in the **Output section**.

Vehicle Command

Parameters – **type** (string), **model** (string), **weight** (double), **price** (decimal), **attack** (integer), **defense** (integer), **hitPoints** (integer).

Creates a **`Vehicle`** of the **given type**, with the **given model**.
The type will either be "**`Vanguard`**" or "**`Revenger`**".

Part Command

Parameters – **vehicleModel** (string), **type** (string), **model** (string), **weight** (double), **price** (decimal), **additionalParameter** (integer).

Creates a **`Part`** of the **given type** with the **given model** and **adds** it to the **`Assembler`** of the **Vehicle** with the **given vehicleModel**.

The **`type`** will either be "**`Arsenal`**", "**`Shell`**" or "**`Endurance`**".

Depending on the Part type, the **`additionalParameter`** will be set to a different property:

- If it's an **`ArsenalPart`** the **additionalParameter** will be set ot the **`attackModifier`**.
- If it's a **`ShellPart`** the **additionalParameter** will be set ot the **`defenseModifier`**.
- If it's an **`EndurancePart`** the **additionalParameter** will be set ot the **`hitPointsModifier`**.

Inspect Command

Parameters – **model** (string)

Brings data about the **Vehicle** or the **Part** with the **given model**, providing **detailed information** about it.

Battle Command

Parameters – **vehicle1Model** (string), **vehicle2Model** (string)

Initiates a battle between **2 Vehicles**. You should **pass** the **2 parameters** to the **`BattleOperator`**, and when you get the **resulting winner**, you should **remove** the **loser** from the **data**.

Terminate

Exits the program. Prints **detailed information** about the **current state** of the system.

© Software University Foundation. This work is licensed under the CC-BY-NC-SA license.

Input

The input consists of several commands which will be given in the format, specified below: :

- `Vehicle {vehicleType} {model} {weight} {price} {attack} {defense} {hitPoints}`
- `Part {vehicleModel} {partType} {model} {weight} {price} {additionalParameter}`
- `Inspect {model}`
- `Battle {vehicle1Model} {vehicle2Model}`
- `Terminate`

Output

Each of the commands generates **output**. Here are the **output formats** of each command:

- **Vehicle Command** – creates a **Vehicle** of the **given type**, with the **given model**. Prints the following result:

  `Created {type} Vehicle – {model}`

- **Part Command** – adds a **Part** of the **given type**, with the **given model** to a **specified Vehicle.**

  `Added {partType} - {partModel} to Vehicle - {vehicleModel}`

- **Inspect command** – provides **detailed information** about a **Vehicle** or a **Part**, in one of the following formats:

Vehicle	Part
{vehicleType} – {vehicleModel} Total Weight: {totalWeight} Total Price: {totalPrice} Attack: {totalAttack} Defense: {totalDefense} HitPoints: {totalHitPoints} Parts: {part1Model}, {part2Model}...	{partType} Part – {partModel} +{additionalParamValue} {additionalParam}

  Because of the fact, that the **Part** is not particular, the **additionalParameter** should either be "**Attack**", "**Defense**", "**HitPoints**".

  In case **there are no Parts** for the Vehicle, print "**Parts: None**".

  The **totalWeight** and **totalPrice** must be printed to the **3rd digit after** the **decimal point**.

  - The **Parts** in the output should be **ordered** by **order** of **input**.

- **Battle command** – The command should return as output the winner in the following format:

  `{vehicle1Model} versus {vehicle2Model} -> {winnerModel} Wins! Flawless Victory!`

- **Terminate command** – Terminates the program; **prints** detailed statistics about the whole system. The format, in which the statistics should be printed is:

  ```
  Remaining Vehicles: {vehicle1Model}, {vehicle2Model}...
  Defeated Vehicles: {defeatedVehicle1Model}, {defeatedVehicle2Model}...
  Currently Used Parts: {countOfCurrentlyUsedParts}
  ```

  - **Remaining Vehicles** are all Vehicles that **have not been** defeated in a battle.

SoftUni Foundation
© Software University Foundation. This work is licensed under the CC-BY-NC-SA license.
Follow us:

- **`Defeated Vehicles`** are all Vehicles that **have been** defeated in a battle.
- **`Currently Used Parts`** is the **amount** of **parts** used by the **`Remaining Vehicles`**. (Exclude those from the **`Defeated Vehicles`**).
- In case there are no **`Remaining Vehicles`** or **`Defeated Vehicles`** print "**None**".
- **All data** in the output should be **ordered** by **order** of **input**.

Constrains

- All **integers** in the input will be in **range [0, 800.000.000]**.
- All **floating-point numbers** in the input will be in **range [0, 1.000.000.000]**.
- All **strings** in the input may contain **any ASCII character**, except **space** (' ').
- All **input lines** will be **absolutely valid**.
- There will be **no** non-existent **models** or **types** in the input.

Examples

Input	Output

`Vehicle Vanguard SA-203 100 300 1000 450 2000`
`Vehicle Revenger AKU 1000 1000 1000 1000 1000`
`Part SA-203 Arsenal Cannon-KA2 300 500 450`
`Part AKU Shell Shields-PI1 200 1000 750`
`Inspect SA-203`
`Inspect AKU`
`Terminate`
	`Created Vanguard Vehicle - SA-203`
	`Created Revenger Vehicle - AKU`
	`Added Arsenal - Cannon-KA2 to Vehicle - SA-203`
	`Added Shell - Shields-PI1 to Vehicle - AKU`
	`Vanguard - SA-203`
	`Total Weight: 400.000`
	`Total Price: 800.000`
	`Attack: 1450`
	`Defense: 450`
	`HitPoints: 2000`
	`Parts: Cannon-KA2`
	`Revenger - AKU`
	`Total Weight: 1200.000`
	`Total Price: 2000.000`
	`Attack: 1000`
	`Defense: 1750`
	`HitPoints: 1000`
	`Parts: Shields-PI1`
	`Remaining Vehicles: SA-203, AKU`
	`Defeated Vehicles: None`
	`Currently Used Parts: 2`

`Vehicle Revenger Destroyer-2U 1500 100000 9500 5000 15000`
`Vehicle Revenger Falcon-303 750 55000 4500 2000 6500`
	`Created Revenger Vehicle - Destroyer-2U`
	`Created Revenger Vehicle - Falcon-303`
	`Created Vanguard Vehicle - Rhino-CE`

SoftUni Foundation
© Software University Foundation. This work is licensed under the CC-BY-NC-SA license.
Follow us:

Vehicle Vanguard Rhino-CE 3000 85000 2000 4000 20000 Part Destroyer-2U Arsenal Cannon-X 1000 50000 5000 Part Destroyer-2U Arsenal Cannon-Y 1000 50000 5000 Part Rhino-CE Shell Shield-EX 2000 45000 3000 Battle Destroyer-2U Rhino-CE Inspect Destroyer-2U Terminate	Added Arsenal - Cannon-X to Vehicle - Destroyer-2U Added Arsenal - Cannon-Y to Vehicle - Destroyer-2U Added Shell - Shield-EX to Vehicle - Rhino-CE Destroyer-2U versus Rhino-CE -> Destroyer-2U Wins! Flawless Victory! Revenger - Destroyer-2U Total Weight: 3500.000 Total Price: 200000.000 Attack: 19500 Defense: 5000 HitPoints: 15000 Parts: Cannon-X, Cannon-Y Remaining Vehicles: Destroyer-2U, Falcon-303 Defeated Vehicles: Rhino-CE Currently Used Parts: 2

SoftUni Foundation
© Software University Foundation. This work is licensed under the CC-BY-NC-SA license.
Follow us: