# Java OOP Advanced Exam – Panzer

In a galaxy far far away, a humanoid race called – The Sabatons, lived peacefully and happily. The key to their peaceful life, was the annual Panzer competition, in which they combined forces into teams and assembled strong vehicles with which they fight in a deadly fray. By some unknown reasons, Reltih, the Sabatons’ Deity, has decided to give you the power of automating the Panzer competition in a software program. So, let’s go!

### Overview

Panzer is a competition between vehicles, which can be assembled with parts and upgraded. The competition needs to be automated with a software program. Reltih tried to write some code, but got lazy. As a god, he is allowed to be lazy, so you must pick up his code, and finish what he started. You must use his code, or he will get angry. We don’t want that…

### 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. The Vanguard modifies its properties in the following way:

* Weight – **increases** its given weight by **100%**.
* Attack – **decreases** its given attack by **25%**.
* Defense – **increases** its given defense by **50%**.
* HitPoints – **increases** its given hitPoints by **75%**.

##### Revenger

A jet-like aerial vehicle. The Revenger modifies its properties in the following way:

* Price – **increases** its given price by **50%**.
* Attack – **increases** its given attack by **150%**.
* Defense – **decreases** its given defense by **50%**.
* HitPoints – **decreases** its given hitPoints by **50%**.

The **modification** of the **properties** **works only** for the **Vehicle**’s **own properties**! Any **external elements** that **affect** its **total stats**, **should NOT** be **affected** by these **modifications**.

#### 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:

* 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 is given to you in the skeleton. You can check more info about it in the Skeleton section.

#### BattleOperator

The BattleOperator is given to you in the skeleton. You can check more info about it in the Skeleton section.

### 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.

**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.

### Skeleton

In this section you will be given information about the Skeleton, or the code that has been given to you.

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.

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.

#### Interfaces & Others

You will be given the **interfaces** for the Vehicle and Part entities. You should use them when you are implementing your entities.

You will **also be given** an **interface** for the Assembler class, but you will be given the **class itself** too.

You will **also be given** an **interface** for the BattleOperator class, but you will be given the **class itself** too.

Read the documentation of the interfaces to gain basic knowledge of the behavior they define.

#### 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**.

### 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.
  + 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: 500.000  Total Price: 800.000  Attack: 1200  Defense: 675  HitPoints: 3500  Parts: Cannon-KA2  Revenger - AKU  Total Weight: 1200.000  Total Price: 2500.000  Attack: 2500  Defense: 1250  HitPoints: 500  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  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 | Created Revenger Vehicle - Destroyer-2U  Created Revenger Vehicle - Falcon-303  Created Vanguard Vehicle - Rhino-CE  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: 250000.000  Attack: 33750  Defense: 2500  HitPoints: 7500  Parts: Cannon-X, Cannon-Y  Remaining Vehicles: Destroyer-2U, Falcon-303  Defeated Vehicles: Rhino-CE  Currently Used Parts: 2 |

### Tasks

#### Task 1: High Quality Structure

##### Refactor the given Skeleton code and use it.

Reltih tried to write some code before you, but he got lazy pretty quickly… But he somehow managed to write the Assembler and BattleOperator classes. His work, however, is not that trustworthy, so you might have to give it an eye or two, for potential **functionality bugs** and things that **do NOT follow** the **good practices** of **Object-Oriented Programming**.

Refactor anything, which will **improve** the **code quality**, in your opinion. Be careful **NOT** to **break the code** or one of the **rules** specified in the **Skeleton** **section**.

##### High Quality Code.

Achieve good separation of concerns using abstractions and interfaces to decouple classes, while reusing code through inheritance and polymorphism. Your classes should have strong cohesion - have single responsibility and loose coupling - know about as few other classes as possible.

##### Reflection.

Since the Assembler class does not reveal much, you will probably have to use a some reflection for the business logic of the Vehicles.

For this task, submit only the “panzer” folder.

#### Task 2: Correct business logic.

The given code provides some functionality, but it does not cover the entire task. Implement the rest of the business logic, using the given code, and implement everything following the requirements specification. Check your solutions in the Judge system.

For this task, submit the whole “src” folder.

#### Task 3: Unit Testing.

Test the Assembler class’s methods for potential bugs. Extensive testing might require you to have some of the core logic implemented, in order to cover all cases.

When testing, use **ONLY THE CLASSES, PROVIDED** by the **SKELETON**.

For this task submit the **folder** you have put your **tests** into.

**NOTE**: You are **NOT ALLOWED** to submit **non-test classes** for this task.