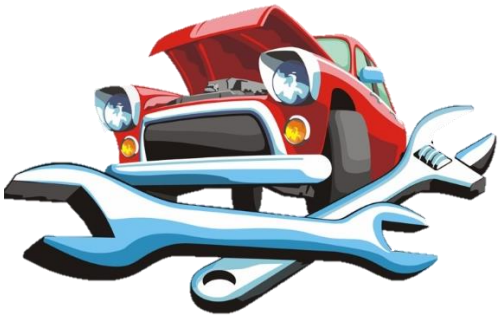
Exam Preparation – 31 July 2023



Link: https://judge.softuni.org/Contests/3930/Java-OOP-Retake-Exam-18-April-2023

1. Overview

You work in a car shop, but you're behind the scenes - you make them. Your task is to create the classes needed for the application and implement the logic. The application must have support for **`Worker`**, **`Vehicle`**, **`Shop`** and **`Tool`**. The project will consist of **model classes** and a **controller class**, which manages the **interaction** between the **workers, vehicles, shops** and **tools**.

2. Setup

- Upload **only the vehicleShop** package in every task **except Unit Tests.**
- **Do not modify the interfaces or their packages.**
- Use **strong cohesion** and **loose coupling.**
- **Use inheritance and the provided interfaces wherever possible**:
  - This includes **constructors**, **method parameters,** and **returns types.**
- **Do not** violate your **interface implementations** by adding **more public methods** 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 interfaces, and you have to implement their functionality in the **correct classes**.

There are **4** types of entities in the application: **`Worker`, `Vehicle`, `Shop`, and `Tool`**.

There should also have been a **`WorkerRepository`**, as well as **`VehicleRepository`**.

BaseWorker

`BaseWorker` is a **base class** or any **type of `Worker`** and it **should not be able be instantiated**.

Data

- **`name – String`**
  - If the name **is null or whitespace,** throw an **`IllegalArgumentException`** with a message: **`"Worker name cannot be null or empty."`**
  - All names will be unique.

Note: Use this check when writing the **setter**: **`if (name == null || name.equals("")){…}`**.

- **`strength – int`**

- The strength of a worker.
- If the **strength** is below **0,** throw an **`IllegalArgumentException`** with a message: **`"Cannot create a Worker with negative strength."`**.

- **`tools – Collection<Tool>`**
  - A collection of a worker's tools.

Constructor

A **`BaseWorker`** should take the following values upon initialization:

```
(String name, int strength)
```

Behavior

`void working()`

The **`working()`** method decreases workers' strength by 10.

- A worker's strength should **not** drop **below 0** (If the strength becomes less than 0, set it to 0).

`void addTool(Tool tool)`

This method **adds** a **tool** to the worker's **collection** of tools.

`boolean canWork()`

This method returns:

- **`true`** - if the current strength of the worker is **greater** than **0**
- **`false`** - otherwise

Child Classes

There are two types of **`BaseWorker`**:

FirstShift

Initial **strength** units: **100.**

The constructor should take the following values upon initialization:

```
(String name)
```

SecondShift

Initial **strength** units: **70.**

The method **`working()`** **decreases** the workers' strength by additional **5 units**.

The constructor should take the following values upon initialization:

```
(String name)
```

ToolImpl

The **`ToolImpl`** is a class that represents the tool, which a **`Workers`** uses to make **`Vehicle`**.

It should be able to be **instantiated**.

Data

- **`power – int`**
  - The power of a tool.
  - If the **initial** power is below **0,** throw an **`IllegalArgumentException`** with a message:

`"Cannot create a Tool with negative power."`.

Constructor

A **`ToolImpl`** should take the following values upon initialization:

```
(int power)
```

Behavior

`void decreasesPower()`

The **`decreasesPower()`** method **decreases** the tool's **power** by **5**.

- A tool's power should **not** drop **below 0.** (If the power becomes less than 0, set it to 0).

`boolean isUnfit()`

This method returns:

- **`true`** – when **power** becomes equal to **0**
- **`false`** - otherwise.

VehicleImpl

This is the class that holds information about the **`Vehicle`** that a **`Worker`** is working on.

It should be able to be **instantiated**.

Data

- **`name - String`**
  - If the name **is null or whitespace,** throw an **`IllegalArgumentException`** with a message: **`"Vehicle name cannot be null or empty."`**.

Note: Use this check when writing the **setter**: **`if (name == null || name.equals("")){…}`**.

- **`strengthRequired – int`**
  - The **strength** required to make the vehicle.
  - If the **initial** strength is below **0,** throw an **`IllegalArgumentException`** with a message: **`"Cannot create a Vehicle requiring negative strength."`**.

Constructor

A **`VehicleImpl`** should take the following values upon initialization:

```
(String name, int strengthRequired)
```

Behavior

`void making()`

The **`making()`** **decreases** the required strength of the vehicle by **5 units**.

- A vehicle's required strength should **not** drop **below 0**.
  - If the strength becomes less than 0, set it to 0.

`boolean reached()`

The **`reached()`** method returns **true** if the **`strengthRequired`** reaches **0**.

ShopImpl

Create a **`ShopImpl`** class. The **`ShopImpl`** class holds the main action, which is the **`make`** method.

Behavior

`void make(Vehicle vehicle, Worker worker)`

Here is how the **`make`** method works:

- The worker starts making the vehicle. This is only possible if the worker has strength and a tool that isn't broken.
- Keep working **until** the vehicle is **done** or the worker has **strength** (and **tools** to use).
- If at some point the **power** of the current tool **reaches** or **drops below 0**, meaning it is **broken**, then the worker should take the **next tool** from its collection, if it has **any left**.

WorkerRepository

The worker repository is a repository for the worker working at Vehicle's Shop.

Data

- **`workers`** – **a collection of workers**

Behavior

`void add(Worker worker)`

- **Adds** a worker to the collection.
- There will be no workers of the same name.

`boolean remove(Worker worker)`

- **Removes** a worker from the collection.
- Returns **`true`** if the deletion was **successful.**

`Worker findByName(String name)`

- Returns a **worker** with that **name** if such exists. If it **doesn't exist** - return **null**.

`Collection<Worker> getWorkers()`

- Returns a collection of workers (**unmodifiable**).

VehicleRepository

The vehicle repository is a repository for vehicles that await to be made.

Data

- **`vehicles`** – **a collection of vehicles.**

Behavior

`void add(Vehicle vehicle)`

- **Adds** a vehicle to be making.
- There will be no vehicle of the same name.

`boolean remove(Vehicle vehicle)`

- **Removes** a vehicle from the collection.

- Returns **`true`** if the deletion was **successful.**

`Vehicle findByName(String name)`

- Returns a **vehicle** with that **name** if such exists.
- It is guaranteed that the vehicle **exists** in the collection.

`Collection<Vehicle> getWorkers()`

- Returns a collection of workers (**unmodifiable**).

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 `ControllerImpl` class SHOULD NOT handle exceptions! The tests are designed to expect exceptions, not messages!

The first interface is **`Controller`**. You must create a **`ControllerImpl`** class, which implements the interface and implements all its methods. The constructor of **`ControllerImpl`** does not take any arguments. 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.

AddWorker Command

Parameters

- **`type - String`**
- **`workerName - String`**

Functionality

Creates a worker with the given name of the given type.

If the worker is invalid (the type is not **FirstShift** or **SecondShift**), throw an **`IllegalArgumentException`** with the message:

`"Worker type doesn't exist."`

The method should **return** the following message if the **worker** has been **added** to the repository:

`"Successfully added {workerType} with name {workerName}."`

AddVehicle Command

Parameters

- **`vehicleName - String`**
- **`strengthRequired - int`**

Functionality

Creates a **vehicle** with the provided **name** and **required strength** and adds it to the corresponding repository.

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

- `"Successfully added Vehicle: {vehicleName}."`

AddToolToWorker Command

Parameters

- `workerName – String`
- `power – int`

Functionality

Creates a tool with the given power and adds it to the collection of the worker.

If the worker doesn't exist in the worker repository, throw an **`IllegalArgumentException`** with the message:

`"The worker doesn't exist. You cannot add a tool."`

The method should **return** the following message if the **tool** has been **added** to the worker:

`"Successfully added tool with power {toolPower} to worker {workerName}."`

MakingVehicle Command

Parameters

- `vehicleName – String`

Functionality

When the **`making`** command is called, the action happens.

You should start making the given vehicle, by assigning workers which are almost ready:

- The workers that you should select are the ones with strength **above** 70 units.
- The suitable ones start working on the given vehicle.
- If no **workers are ready**, throw **`IllegalArgumentException`** with the following message:
  `"There is no worker ready to start making."`
- After the work is done, you must return the following message, reporting whether the vehicle is done and how many total tools were unfit in the process:

`"Vehicle {vehicleName} is {done/not done}. {countBrokenTools} tool/s have been unfit while working on it!"`

Note: The **name** of the **vehicle** you receive will always be a **valid** one.

Statistics Command

Functionality

Returns information about **making vehicles** and **workers**:

`"{countMadeVehicle} vehicles are ready!`

`Info for workers:`

`Name: {workerName1}, Strength: {workerStrength1}`

`Tools: {countTools} fit left"`

…

`"Name: {workerNameN}, Strength: {workerStrengthN}`

`Tools: {countTools} fit left"`

Exit Command

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 **Engine** 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:

- `AddWorker {workerType} {workerName}`
- `AddVehicle {vehicleName} {strengthRequired}`
- `AddToolToWorker {workerName} {power}`
- `MakingVehicle {vehicleName}`
- `Statistics`
- `Exit`

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

`AddWorker SecondShift Sara`
`AddWorker FirstShift Maria`
`AddWorker InvalidShift Sam`
`AddToolToWorker Sara 10`
`AddToolToWorker Maria 20`
`AddToolToWorker Maria 30`
`AddToolToWorker Maria 20`
`AddToolToWorker Maria 40`
`AddVehicle Truck 20`
`AddVehicle Car 20`
`AddVehicle Train 30`
`MakingVehicle Truck`
`MakingVehicle Car`
`MakingVehicle Train`
`Statistics`
`Exit`

Output

`Successfully added SecondShift with name Sara.`
`Successfully added FirstShift with name Maria.`
`Worker type doesn't exist.`
`Successfully added tool with power 10 to worker Sara.`
`Successfully added tool with power 20 to worker Maria.`
`Successfully added tool with power 30 to worker Maria.`
`Successfully added tool with power 20 to worker Maria.`
`Successfully added tool with power 40 to worker Maria.`
`Successfully added Vehicle: Truck.`

```
Successfully added Vehicle: Car.
Successfully added Vehicle: Train.
Vehicle Truck is done. 1 tool/s have been unfit while working on it.
There is no worker ready to start making.
There is no worker ready to start making.
1 vehicles are ready!
Info for workers:
Name: Sara, Strength: 70
Tools: 1 fit left
Name: Maria, Strength: 60
Tools: 3 fit left
```

Input

```
AddWorker SecondShift Maria
AddWorker FirstShift Anna
AddWorker SecondShift Nicol
AddWorker FirstShift Antonio
AddToolToWorker Maria 40
AddToolToWorker Anna 50
AddToolToWorker Maria 50
AddToolToWorker Nicol 50
AddToolToWorker Antonio 40
AddToolToWorker Antonio 30
AddVehicle Train 50
AddVehicle Tractor 20
AddVehicle AutomaticCar 30
MakingVehicle Train
MakingVehicle Tractor
MakingVehicle AutomaticCar
Statistics
Exit
```

Output

```
Successfully added SecondShift with name Maria.
Successfully added FirstShift with name Anna.
Successfully added SecondShift with name Nicol.
Successfully added FirstShift with name Antonio.
Successfully added tool with power 40 to worker Maria.
Successfully added tool with power 50 to worker Anna.
Successfully added tool with power 50 to worker Maria.
Successfully added tool with power 50 to worker Nicol.
Successfully added tool with power 40 to worker Antonio.
Successfully added tool with power 30 to worker Antonio.
Successfully added Vehicle: Train.
Successfully added Vehicle: Tractor.
Successfully added Vehicle: AutomaticCar.
Vehicle Train is done. 1 tool/s have been unfit while working on it.
Vehicle Tractor is done. 0 tool/s have been unfit while working on it.
There is no worker ready to start making.
2 vehicles are ready!
Info for workers:
Name: Maria, Strength: 70
Tools: 2 fit left
Name: Anna, Strength: 0
Tools: 0 fit left
Name: Nicol, Strength: 70
Tools: 1 fit left
Name: Antonio, Strength: 60
```

```
Tools: 2 fit left
```

Task 3: Unit Tests (100 points)

You will receive a skeleton with three classes inside – **`Main`**, **`Car`**, and **`CarShop`**. **`CarShop`** class will have some methods, fields, and constructors. Cover the whole class with the unit test to make sure that the class is working as intended. In Judge, you upload **.zip** to **gifts (**with **`CarShopTests`** inside**)** from the **skeleton**.