## BarracksWars - A New Factory

You are given a small console based project called Barracks (the code for it is included in the provided skeleton).

The general functionality the project has is adding new units to its repository and printing a report with statistics about the units currently in the repository. First let's go over the original task the project was made for:

### Input

The input consists of commands each on a separate line. Commands that execute the functionality are:

* **add** <**Archer**/**Swordsman**/**Pikeman/{…}**> - adds a unit to the repository.
* **report** - prints a lexicological ordered statistic about the units in the repository.
* **fight** - ends the input.

### Output

Each command except **fight** should print output on the console.

* **add** should print: "<**Archer**/**Swordsman**/**Pikeman**/**{…}> added!**"
* **report** should print all the info in the repository in the format: "<**UnitType**> **->** <**UnitQuantity**>", sorted by UnitType

### Constraints

* Input will consist of no more than **1000** lines
* **report** command will never be given before any valid add command was provided

### Your task

1) You have to **study the code of the project and figure out how it works**. However, there are parts of it that are not implemented (left with TODOs). You must implement the functionality of the **createUnit** method in the **UnitFactoryImpl** class so that it creates a unit based on the unit type received as parameter. Implement it in such a way that whenever you add a new unit it will be creatable without the need to change anything in the **UnitFactoryImpl** class (psst - use reflection). You can use the approach called Simple Factory.

2) Add two new unit classes (there will be tests that require them) - **Horseman** with 50 health and 10 attack and **Gunner** with 20 health and 20 attack.

If you do everything correctly for this problem, you should write code only in the **factories** and **units** packages.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| add Swordsman  add Archer  add Pikeman  report  add Pikeman  add Pikeman  report  fight | Swordsman added!  Archer added!  Pikeman added!  Archer -> 1  Pikeman -> 1  Swordsman -> 1  Pikeman added!  Pikeman added!  Archer -> 1  Pikeman -> 3  Swordsman -> 1 |
| add Pikeman  add Pikeman  add Gunner  add Horseman  add Archer  add Gunner  add Gunner  add Horseman  report  fight | Pikeman added!  Pikeman added!  Gunner added!  Horseman added!  Archer added!  Gunner added!  Gunner added!  Horseman added!  Archer -> 1  Gunner -> 3  Horseman -> 2  Pikeman -> 2 |

## \* BarracksWars - The Commands Strike Back

As you might have noticed commands in the project from Problem 3 are implemented via a switch case with method calls in the **Engine** class. Although this approach works it is flawed when you add a new command because you have to add a new case for it. In some projects you might not have access to the engine and this would not work. Imagine this project will be outsourced and the outsourcing firm will not have access to the engine. Make it so whenever they want to add a new command they won't have to change anything in the **Engine.**

To do so employ the design pattern called [**Command Pattern**](https://en.wikipedia.org/wiki/Command_pattern). We've done this in the **BashSoft** **Lab** and you can look there for tips too. Use the provided **Executable** interface as a frame for the command classes. Put the new command classes in the provided **commands** package inside **core**. You can also make a Command Interpreter to decouple that functionality from the Engine. Here is how the base (abstract) command should look like:

Notice how all commands that extend this one will have both a Repository and a UnitFactory although not all of them need these. Leave it like this for this problem, because for the reflection to work we need all constructors to accept the same parameters. We will see how to go around this issue in problem 5.

Once you've implemented the pattern add a new command. It will have the following syntax:

* **retire** <**UnitType**> - All it has to do is remove a unit of the provided type from the repository.
  + If there are no such units currently in the repository print: "**No such units in repository.**"
  + If there is such a unit currently in the repository, print: "**<UnitType> retired!**"

To implement this command, you will also have to implement a corresponding method in the **UnitRepository**.

If you do everything correctly for this problem, you should write/refactor code only in the **core** and **data** packages.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| retire Archer  add Pikeman  add Pikeman  add Gunner  add Horseman  add Archer  add Gunner  add Gunner  add Horseman  report  retire Gunner  retire Archer  report  retire Swordsman  retire Archer  fight | No such units in repository.  Pikeman added!  Pikeman added!  Gunner added!  Horseman added!  Archer added!  Gunner added!  Gunner added!  Horseman added!  Archer -> 1  Gunner -> 3  Horseman -> 2  Pikeman -> 2  Gunner retired!  Archer retired!  Archer -> 0  Gunner -> 2  Horseman -> 2  Pikeman -> 2  No such units in repository.  No such units in repository. |