Programming 1

Aryobarzan Atashpendar (aryobarzan.atashpendar@uni.lu)

Lab 9 – Abstract Classes, Polymorphism

Starcraft © *Blizzard Ent.*

Exercise 1 – You Must Construct Additional Pylons

In the video game *Starcraft*, there are different races the player can choose from, with each race featuring their own unique units to build an army out of.

A basic **Unit** has the following properties:

- *health* and *maxHealth*, with *health* tracking the current health of the **Unit**. The *maxHealth* cannot be changed once initialized. The *health* starts at the maximum when creating a **Unit**.
- *damage*, which cannot be changed once initialized.
- *name*.
- *health*, *maxHealth* and *damage* cannot be negative.

However, a basic **Unit** cannot be created. Instead, each race has their own set of available units. The first race is Protoss, who can have **ProtossUnit**s:

- **ProtossUnit** has two extra properties: *shield* and *maxShield*, which are similar to *health* and *maxHealth*.
- The Protoss race can create two specific types of **ProtossUnit**:
  1° **ZealotUnit**: 75 *maxHealth*, 2 *damage*, 25 *shield*.
  2° **ArchonUnit**: 50 *maxHealth*, 5 *damage*, 150 *shield*.

The second race is Zerg, who can have **ZergUnit**s:

- **ZergUnit** has no extra properties.
- The Zerg race can create two specific types of **ZergUnit**:
  1° **ZerglingUnit**: 25 *maxHealth*, 1 *damage*
  2° **HydraliskUnit**: 50 *maxHealth*, 3 *damage*

A player keeps track of their units through an Army, which can contain a specific number of Units:

- If the player has picked the Protoss race, they own a **ProtossArmy**, which can contain up to 10 **ProtossUnits**.
- If the player has picked the Zerg race, they own a **ZergArmy**, which can contain up to 20 **ZergUnits**.
- To add a **Unit**, **Army** features a function **add(Unit unit)**, which should add the given *unit* to the **Army**'s array of *units* if there is sufficient space. Ensure a **ProtossArmy** only accepts **ProtossUnits** and a **ZergArmy** only accepts **ZergUnits**.

Finally, a **Unit** can attack another **Unit** through its function **attack(Unit unit)**:

- Both the attacker and target unit must be alive ($health > 0$).
- A **Unit** cannot attack itself.
- If both constraints are satisfied, the attack results in the target unit's *health* being reduced by an amount given by the attacker's *damage*.
    - **ProtossUnits** behave slightly differently: before their *health* takes a hit, their *shield* is reduced instead. Once a **ProtossUnit**'s *shield* has reached 0, their *health* can also be reduced by attacks.

Exercise 2 – Dungeons & Dragons

In the tabletop game Dungeons and Dragons, players get to pick a specific **Adventurer** they want to play:

- All **Adventurers** have a specific *constitution*, *wisdom* and *dexterity* (integer values), which cannot be changed once initialized.
- **Adventurers** also have a property *isAlive*, which indicates whether a given **Adventurer** is alive or not.

Players can pick from 3 different **Adventurers**:

1° **ClericAdventurer**: 20 *wisdom*, 1 *constitution*, 5 *dexterity*.

2° **FighterAdventurer**: 1 *wisdom*, 20 *constitution*, 5 *dexterity*.

3° **DungeonMaster**: 9999 *wisdom*, 9999 *constitution*, 9999 *dexterity*.

- Only a single **DungeonMaster** can exist.

Each specific **Adventurer** has their own type of **Action**:

- **Actions** have a function **void execute(Adventurer target)**.
    - **ClericAdventurer** has a **ReviveAction**, which should set the target's *isAlive* to true.
    - **FighterAdventurer** has a **MurderAction**, which should set the target's *isAlive* to false.

- However, **ClericAdventurer** and **FighterAdventurer** cannot directly use their own action. Instead, they request the **DungeonMaster** to perform the **Action** for them, who checks some conditions before performing the **Action**:
  - An **Adventurer** cannot target themself for their own **Action**.
  - An **Adventurer** has to be alive to request their **Action** to be performed.
  - The **DungeonMaster** can never be the target of an **Action** and the **DungeonMaster** has no **Action** of their own.
- Think of how you could model this behavior and the different **Actions**.

Exercise 3 – Filters

Implement different filters which can be used to filter numbers meeting a certain criterion out of a given set of integers. The example in Figure 1 points out the required functionality.

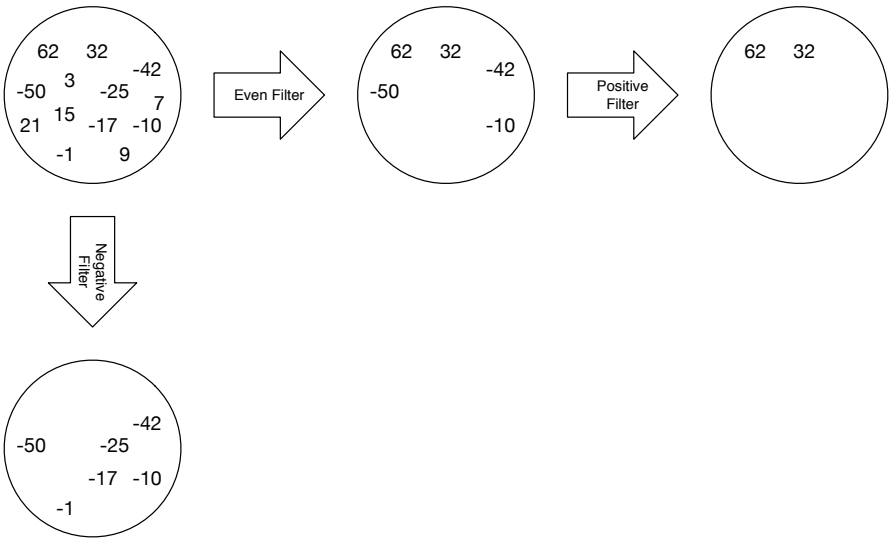

Figure 1: Example of different filters being applied to a set of numbers

It should be possible to apply a filter to a given set of integers in order to filter out a certain subset. Moreover the sequential composition of filters needs to be possible. In the example the *Even Filter* filters out all even numbers of the set and afterwards the *Positive Filter* filters all positive numbers out of the resulting subset. Applying the *Negative Filter* to the initial set filters out all negative numbers.

The following types of filters should be implemented:

- **Even Filter** Filters out all even numbers.
- **Odd Filter** Filters out all odd numbers.
- **Negative Filter** Filters out all negative numbers.
- **Positive Filter** Filters out all positive numbers including zero.
- **Divisible-by Filter** Filters out all numbers divisible without remainder by a given integer value.

Think carefully about a good design of a class hierarchy that fulfills the requirements mentioned above. Some questions that need to be addressed when designing an appropriate hierarchy are:

- Which class do you actually need?
- Which classes can be abstract?
- How can the set of integers be realized?
- Should a filter modify an existing integer set or create a new one?
- Where to implement a method to apply a given filter to a given set?
- Should a filter be tied to a specific set or should sets and filters be more loosely coupled?

Make sure that your hierarchy allows the seamless replacement of filters and that the extension by additional filter types is easily possible.

After carefully designing a class hierarchy implement your approach. Think about what are the advantages and disadvantages of your solution. It is even recommended to implement different approaches since this could be pretty helpful in comparing the different designs. Moreover implement a main function to test your classes.