Programming 1

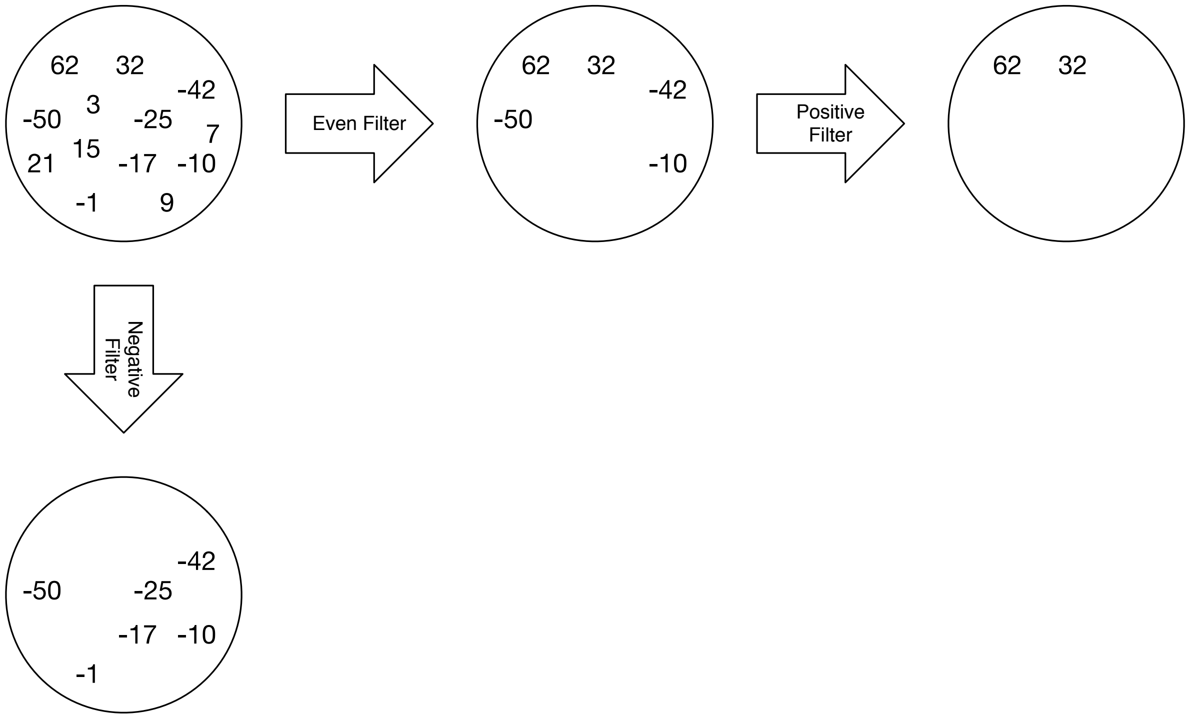
Christian Grévisse (`christian.grevisse@uni.lu`)
Gilles Neyens (`gilles.neyens@uni.lu`)

Lab 7 – Polymorphism

2017–11–21

Exercise 1 – Filters

You want to implement different filters which can be used to filter numbers meeting a certain criterion out of a given set of integers. The example in the following figure points out the required functionality.



It should be possible to apply a filter to a given set of integers in order to filter out a certain subsets. 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.

Greater Filter Filters out all numbers greater than or equal to a given minimum value.

Less Filter Filters out all numbers less than a given maximum value.

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.

Interval Filter Filters out all numbers in a given interval.

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.

Exercise 2 – Vending Machine
A vending machine has a set of buttons. Each specific kind of button is responsible for returning a specific product. For instance, a Coke button is responsible for returning a Coke, whereas a Bounty button is responsible for returning a Bounty. Every product has a price, but may have some product-specific properties, too. Every button provides a method which returns the corresponding product.

Design a hierarchy with a set of specific products and corresponding buttons. Make sure the hierarchy can be arbitrarily extended with other products and buttons.

Finally, implement a vending machine which holds a set of buttons (which can be hard-coded). The machine also holds a cart of a certain capacity. Each time a button is clicked, the corresponding product is added to the cart. When finishing the order, the total amount due is shown. Write a main method to test your implementation.

Exercise 3 – Game
A game is composed of rooms. Each room contains some enemies, a door and a next room that can be accessed by going through the door. There exists 2 types of rooms for the moment (a standard room and a boss room), 2 types of enemies (wolf and boss) and 2 types of doors (normal door and boss door). Doors should have a method isDoorOpen which returns true if the door is open and the player can pass to the next room. Normal doors are always open but boss doors are only open if the boss in the same room is dead. A standard room contains a random number of wolves (maximum 5) and have a normal door while boss rooms contain a single boss and a boss door. Enemies should have an attribute health, should implement the toString method and should have a method kill which will set the health of the enemy to 0 and print out that it has been killed. Bosses have an additional attribute name.

Design and implement a hierarchy for the rooms, enemies and doors so that the game could be extended easily with other types of doors and enemies. If the design is well done the rooms could even get extended with other attributes like items without the need to change much of the existing code.

Finally, implement a main program to test the game. To simplify things just let the computer play the game. Create a few rooms, connect them to each other and just let the computer randomly choose whether to move to the next room or to kill the enemies in the current room. He can not move to another room if the condition to open the door is not satisfied. If there is no next room the game will stop and the computer is victorious.

Christian Grévisse (christian.grevisse@uni.lu)
Gilles Neyens (gilles.neyens@uni.lu)

Exercise 4 – Bank Accounts

A normal bank account has an attribute balance and 2 methods deposit and withdraw. One can not withdraw more money than the account contains. An overdraft account has an additional attribute allowedOverdraft which represents the limit below zero that the balance of the account is allowed to reach, meaning that the balance of an overdraft account can go below zero when withdrawing money but not more than the allowedOverdraft.

Design and implement a class hierarchy that makes sense in this situation.