Ministry of Education of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Computer Science

Object-Oriented Programming

Laboratory work #1

The four principles of OOP

Authors:
Daniel Surdu

Supervisor:
Anastasia Șerșun

Chișinău - 2018

1 Work purpose:

The study and implementation of the four principles of object-oriented programming.

2 Introduction

In this laboratory work, I designed a siple 2D sega style batman fighting. It's kind of funny because your are the batman that fight bad red batmen. In this program I have a main function and 2 classes: main_character and the enemy of the batman. The idea of game is to get the highest score 'killing' enemies and you die if at least one of the enemy succeed in hitting you, in this case GAME OVER.

3 Task implementation

3.1 Abstraction

Data abstraction and encapuslation are closely tied together, because a simple definition of data abstraction is the development of classes, objects, types in terms of their interfaces and functionality, instead of their implementation details. An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer. In short, abstraction is a mechanism of extracting the essential elements for the creation of a system, without its implementation details. The focus is only on what is to be done instead of how it should be done.

In the following example, I showed the Movement class. It is a virtual class that contains the function change_position(int x) that will be used in the enemy class and character class which will have different implimentation in both cases.

```
1 class Movement
2 {
3 public:
4     virtual int change_position(int x){}
5 };
```

3.2 Encapsulation

Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition. Encapsulation is the hiding of data implementation by restricting access to accessors and mutators. It's main benefit is that it can reduce system complexity.

An accessor is a method that is used to ask an object about itself, i.e. any public method that gives information about the state of the object. A mutator is a public method that is used to modify the state of an object, while hiding the implementation of exactly how the data gets modified. Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state.

For example, in this example I have public private and protected variables. For private and protected variables exist getters like int getWidth() and int getHeight() and their values are set in onther functions and user don't have acces directly to these variables. The pPosition variable is protected because it is needed in the enemy class.

```
1  class character : public Movement
2  {
3  public:
4      bool stop_left = false;
5      bool stop_right = false;
6      bool stop = false;
7
8      bool hit = false;
9
10     character();
11     ~character();
12     bool loadFromFile(std::string path);
13
14     void free();
15
16     void render(int x, int y, SDL_Rect* clip, SDL_Point* center,
           SDL_RendererFlip flip);
17     int getWidth();
18     int getHeight();
19     int get_position();
20     void set_position(int x);
21     int change_position(int x);
22     void set_initial_position();
23
24 private:
25     SDL_Texture* mTexture;
26
27     int mWidth;
28     int mHeight;
29 protected:
30     int pPosition;
31 };
```

3.3 Inheritance

Inheritance allows us to define a class in terms of another class. Inheritance is a way to reuse code of existing objects, or to establish a subtype from an existing object, or both, depending upon programming language support. Classes can inherit attributes and behavior from pre-existing classes called base classes. The resulting classes are known as derived classes, subclasses or child classes. The relationships of classes through inheritance gives rise to a hierarchy.

The superclass establishes a common interface and foundational functionality, which specialized subclasses can inherit, modify, and supplement. The software inherited by a subclass is considered reused in the subclass. In some cases, a subclass may customize or redefine a method inherited from the superclass. A superclass method which can be redefined in this way is called a virtual method.

In this case I have multiple inheritance. The class enemy inherit public functions of the character and Movement and do not have access to change directly the private variables of the character class.

```
1  class enemy : public character, public Movement
2  {
3  public:
4      SDL_RendererFlip flip;
5      int id;
6      int enemy_waiting_hit;
7      bool wait;
8
9      enemy()
10     {
11         wait = false;
12         int enemy_waiting_hit = 0;
13         flip = SDL_FLIP_HORIZONTAL;
14         pPosition = 810;
15     };
16
17     int change_position(int x);
18     int get_initial_position();
19     void set_initial_position(int i);
20 private:
21     int initial_position;
22 };
```

3.4 Polymorphism

Polymorphism manifests itself by having multiple methods all with the same name, but slightly different functionality. There are 2 basic types of polymorphism. Overridding, also called run-time polymorphism. For method overloading, the compiler determines which method will be executed, and this decision is made when the code gets compiled. Overloading, which is referred to as compile-time polymorphism. Method will be used for method overriding is determined at runtime based on the dynamic type of an object. Below, I presented the different implementations of the change_position() method in the character class and the enemy class, which present the principle of polymorphism.

```
1  int enemy::change_position(int x)
2  {
3      if (initial_position == 0 && !stop)
4      {
5          if (pPosition < -80)
6          {
7              return pPosition = 810;
8          }
9          else
10         {
11             return pPosition -= x;
12         }
13     }
14     else if (!stop)
15     {
16         if (pPosition > 815)
17         {
18             return pPosition == -80;
19         }
20         else
21         {
22             return pPosition += x;
23         }
24     }
25     return pPosition;
26 }
```

```
1  int character::change_position(int x)
2  {
3      return pPosition += x;
4  }
```

Conclusion

During this laboratory work I learned how to use the four principles of OOP. I created a very siple game in C++ which made me think in OO and made me gain experience in working with all OOP principles.

References

1 OOP Principles, https://anampiu.github.io/blog/OOP-principles/

2 Major Principles of OOP, http://codebetter.com/raymondlewallen/2005/07/19/4-major-principles-of-object-oriented-programming/

3 SDL2 tutorials, http://lazyfoo.net/tutorials/SDL/