Technical University of Moldova
Inginerical department S.A.

Report

№: 6

OOP

Subject: Polymorfism and virtual functions

Author: Terman Emil FAF161
Prof: M. Kulev

Chisinau 2017

1 Objectives

- studierea polimorfismului.
- studierea principiilor legăturii întîrziate.
- studierea funcţiilor virtuale.
- polimorfismul ad-hoc.
- realizarea funcţiilor virtuale.
- studierea claselor abstracte.

2 Main notions of theory and used methods

In programming languages, ad hoc polymorphism is a kind of polymorphism in which polymorphic functions can be applied to arguments of different types, because a polymorphic function can denote a number of distinct and potentially heterogeneous implementations depending on the type of argument(s) to which it is applied. It is also known as function overloading or operator overloading. The term ad hoc in this context is not intended to be pejorative; it refers simply to the fact that this type of polymorphism is not a fundamental feature of the type system. This is in contrast to parametric polymorphism, in which polymorphic functions are written without mention of any specific type, and can thus apply a single abstract implementation to any number of types in a transparent way.

In programming languages and type theory, polymorphism is the provision of a single interface to entities of different types.A polymorphic type is one whose operations can also be applied to values of some other type, or types. There are several fundamentally different kinds of polymorphism:

- Ad hoc polymorphism: when a function denotes different and potentially heterogeneous implementations depending on a limited range of individually specified types and combinations. Ad hoc polymorphism is supported in many languages using function overloading.
- Parametric polymorphism: when code is written without mention of any specific type and thus can be used transparently with any number of new types. In the object-oriented programming community, this is often known as generics or generic programming. In the functional programming community, this is often shortened to polymorphism.
- Subtyping (also called subtype polymorphism or inclusion polymorphism): when a name denotes instances of many different classes related by some common superclass.[3] In the object-oriented programming community, this is often referred to as simply Inheritance.

3 Task

Creaţi clasa abstractă de bază Worker cu funcţia virtuală calcularea salariului. Creaţi clasele derivate StateWorker, HourlyWorker şi CommissionWorker, în care funcţia dată este redefinită. În funcţia main determinaţi masivul de pointeri la clasa abstractă, cărei i se atribuie adresele obiectelor claselor derivate.

4 Data analysis

- *Worker* is an abstract class, with a virtual, nonimplemented function *Salary()*.
- the other 3 classes inherit from *Worker*, implementing the *Salary* function.

5 The actual code

CPP 1: Worker.hpp

```cpp
#ifndef WORKER_HPP
# define WORKER_HPP

# include <string>

class Worker
{
public:
    std::string Name() const;
    float TimeWorked() const;

    void SetTimeWorked(float newTime);

    Worker(std::string name);

    virtual float Salary() const = 0;

private:
    std::string const _name;
    float _timeWorked;
};

#endif
```

CPP 2: StateWorker.hpp

```cpp
#ifndef STATEWORKER_HPP
# define STATEWORKER_HPP

# include <string>
# include "Worker.hpp"

class StateWorker : public Worker
{
public:
    StateWorker(std::string name);
    float Salary() const;
};

#endif
```

CPP 3: HourlyWorker.hpp

```cpp
#ifndef HOURLYWORKER_HPP
# define HOURLYWORKER_HPP

# include <string>
# include "Worker.hpp"

class HourlyWorker : public Worker
{
public:
    HourlyWorker(std::string name);
    float Salary() const;
};

#endif
```

CPP 4: CommissionWorker.hpp

```cpp
#ifndef COMMISSIONWORKER_HPP
# define COMMISSIONWORKER_HPP

# include <string>
# include "Worker.hpp"

class CommissionWorker : public Worker
{
public:
    CommissionWorker(std::string name);
    float Salary() const;
};

#endif
```

CPP 5: StateWorker.cpp

```cpp
#include "StateWorker.hpp"

StateWorker::StateWorker(std::string name) : Worker(name)
{
}

float StateWorker::Salary() const
{
    return TimeWorked() * 42 * 1;
}
```

CPP 6: main.cpp

```cpp
#include "Worker.hpp"
#include "StateWorker.hpp"
#include "HourlyWorker.hpp"
#include "CommissionWorker.hpp"

#include <iostream>
#include <vector>
#include <stdlib.h>

int main()
{
    std::vector<Worker*> workers;

    workers.push_back(new StateWorker("StateWorker_Kek1"));
    workers.push_back(new HourlyWorker("HourlyWorker_Kek2"));
    workers.push_back(new HourlyWorker("CommissionWorker_Kek3"));

    srand(0);

    for (auto const & w : workers)
    {
        w->SetTimeWorked(rand() % 100);
        std::cout << w->Name() << ":␣" << w->Salary() << std::endl;
    }
    return 0;
}
```

6 Analysis of the results and conclusions

– an Interface represents a class purly made out of virtual methods.

– an Abstract class is a class with at least one pure virtual method, that is, a method with = *0* at the end.

– an Abstract class cannot be instantiated, only another class which inherits from it and implements the virtual methods can be instantiated.

– using a virtual function, we benefit from the polymorphism. If we don't use virtual and we try to override a simple method in the superclass, and then try to call the method from the base class, then the base class' method will be called, whereas if using virtual, we would've recived the superclass' overriden method instead.