Technical University of Moldova
Inginerical department S.A.

# Report

**№**: 4

OOp

**Subject:** Inheritance and composition

Author: Terman Emil FAF161
Prof: M. Kulev

Chisinau 2017

# 1 Objectives

- study of inheritance, advantages and disadvantages
- study of composition
- study of inheritance patterns

# 2 Main notions of theory and used methods

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.

The idea of inheritance implements the is a relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

# 3 Task

1. Să se creeze o ierarhie a claselor joc – joc sportiv – volei. Determinaţi constructorii, destructorul, operatorul de atribuire şi alte funcţii necesare.
2. Să se creeze class rate, care conţine rază. Determinaţi constructorii şi metodele de acces. Creaţi clasa automobil, care conţine roţi şi un câmp care reprezintă firma producătoare. Creaţi o clasă derivată autocamion care se deosebeşte prin tonaj. Determinaţi constructorii, destructorul şi alte funcţii necesare.

# 4 Data analysis

## 4.1 Ex00

- *VolleyBall* inherits from *SportGame* and this class inherits from *Game*.
- The *Game* has its function `void play(void) const;` overriden by each of his superclasses, each time differently.

## 4.2 Ex01

- The class *Car* is composed of a vector of wheels because there can be vehicles with different number of wheels.
- Since the class *Lorry* inherits from *Car*, the *_wheels* and *_mark* fields are *protected* so that *Lorry* can have access to these fields.
- Starting with the *Car*, we have a composition of 2 different classes: *Wheel* and *Car*.

# 5 The actual code

## 5.1 Ex00

**CPP 1**: Game.hpp

```cpp
#ifndef GAME_HPP
# define GAME_HPP

# include <iostream>
# include <string>

class Game {
public:
    Game(void);
    Game(int playersCount);

    int     getPlayerCount(void) const;
    void    setPlayerCount(int newPlayerCount);

    void    play(void) const;

    Game & operator = (Game const & target);
    friend std::ostream & operator << (std::ostream & o, Game const & target);

protected:
    int     _playerCount;
};

#endif
```

**CPP 2**: SportGame.hpp

```cpp
#ifndef SPORTGAME_HPP
# define SPORTGAME_HPP

# include "Game.hpp"

class SportGame : public Game {
public:
    SportGame(void);
    SportGame(int playersCount, bool inOpenField);

    bool    isInOpenField(void) const;
    void    play(void) const;

    friend std::ostream & operator << (std::ostream & o, SportGame const & g);

private:
    bool const  _inOpenField;
};

#endif
```

**CPP 3**: VolleyBall.hpp

```cpp
#ifndef VOLLEYBALL_HPP
# define VOLLEYBALL_HPP

# include "SportGame.hpp"
# include <stdexcept>

class VolleyBall: public SportGame {
public:
    static int const defaultNbPlayers;
    static int const minNbPlayers;

    class InvalidNbOfPlayers: public std::exception {
    public:
        virtual char const * what() const throw() {
            return "Invalid␣Number␣of␣Players.";
        }
    };

    VolleyBall(void);
    VolleyBall(int nbOfPlayers);

    void play(void) const;

    friend std::ostream & operator << (std::ostream & o, VolleyBall const & t);
};

#endif
```

**CPP 4**: VolleyBall.cpp

```cpp
#include "VolleyBall.hpp"

int const VolleyBall::defaultNbPlayers = 12;
int const VolleyBall::minNbPlayers = 2;

VolleyBall::VolleyBall(void) : SportGame(defaultNbPlayers, true) {}

VolleyBall::VolleyBall(int nbOfPlayers) : SportGame(nbOfPlayers, true) {
    if (_playerCount < minNbPlayers)
        throw InvalidNbOfPlayers();
}

void VolleyBall::play(void) const {
    std::cout << "We're␣playing␣Volley␣Ball!" << std::endl;
}

std::ostream & operator << (std::ostream & o, VolleyBall const & t) {
    o << "Volley␣Ball:␣{players:␣" << t._playerCount << ";␣"
        << "is␣in␣open␣field:␣" << t.isInOpenField() << "}";
    return o;
}
```

## 5.2 Ex01

**CPP 5**: Car.hpp

```cpp
#ifndef CAR_HPP
# define CAR_HPP

# include "Wheel.hpp"
# include <vector>

class Car {
public:
    Car(size_t wheelCount, float wheelR, std::string const & mark);

    size_t wheelCount(void) const;
    float wheelsR(void) const;

    friend std::ostream & operator << (std::ostream & o, Car const & t);

protected:
    std::vector<Wheel>  _wheels;
    std::string const   _mark;
};

#endif
```

**CPP 6**: Lorry.hpp

```cpp
#ifndef LORRY_HPP
# define LORRY_HPP

# include "Car.hpp"

class Lorry: public Car {
public:
    Lorry(float mass, size_t wheelN, float wheelR, std::string const & mark);
    friend std::ostream & operator << (std::ostream & o, Lorry const & t);

private:
    float const _mass;
};

#endif
```

**CPP 7**: Wheel.hpp

```cpp
#ifndef WHEEL_HPP
# define WHEEL_HPP

# include <iostream>
# include <string>
# include <stdexcept>

class Wheel {
public:
    class InvalidR: public std::exception {
    public:
        virtual char const * what() const throw();
    };

    Wheel(float r);
    Wheel(Wheel const & t);

    float getR(void) const;

    friend std::ostream & operator << (std::ostream & o, Wheel const & t);

private:
    float const _r;
};

#endif
```

# 6 Analysis of the results and conclusions

- inheritance is a very good way of reusing the code and it also helps to keep an intuitive structure.

- alongside *public* and *private*, in this laboratory work we discovered a new keyword: "*protected*". It makes the fields withing this section accessible to superclasses but private to any other class.

- it's possible to override the function of the inherited class and we also have the option to keep either keep or override the previous method definition.