Тема 8 : Структуриране на класова йерархия.

В ООП, клас се определя като шаблон, който определя вида и поведение на даден обект. Един клас може да бъде деклариран и от характеристиките на други класове. Примерно: и кола, и камион, са два примера за превозно средство. Друг пример е това как и буква и цифра са пример за символ, който може да бъде изрисуван на екрана. Използвайки терминология на програмен език, втория пример може да бъде обяснен по следния начин:

- Класът Цифра е детски клас (дете) на класа Символ (на английски – child class, използва се също subclass или derived class);

- Класът Символ е прекия родителски клас (родител) на класа Цифра (на английски – parent class, superclass);

- Класът Цифра *наследява* класът Символ;

Думата наследява изказва специфично и характерно поведение на класовете при ООП. Когато се каже че едно дете наследява от родителския си клас се разбира, че то може да използва полета (променливи) и методи(функции), които принадлежат на родителя, стига да има разрешение за достъп до тях (public или protected).

Като синтаксис, в C++, наследяването изглежда по следния начин:

```
class child_class_name : access_mode parent_class_name
{
      . . .
};
```

, където **child_class_name** е името на детето, което ще наследява, **parent_class_name** е името на родителския клас, от който се наследява, а **access_mode** е начина по който ще бъде наследен класа, който може да бъде един от 3 вида: **private**, **protected**, **public**. Разликата между тях е следната:

- при **public** наследяване, всички наследени public полета и методи ще останат public в детето, както всички protected полета и методи ще останат protected в детето.

- при **protected** наследяване, всички наследени public и protected полета и методи ще се превърнат в protected в детето.

- при **private** наследяване, всички наследени public и protected полета и методи ще се превърнат в private в детето.

Изразено с C++ код, наследяването ще изглежда по следния начин:

```
class A
{
      public:
          int x;
      protected:
          int y;
      private:
          int z;
};

class B : public A
{
    // x е public
    // y е protected
    // z не е достижима от B
};

class C : protected A
{
    // x е protected
    // y е protected
    // z не е достижима от C
};

class D : private A    // 'private' е по подразбиране за класовете
{
    // x е private
    // y е private
    // z не е достижима от D
};
```

Целта на наследяването е да се разшири поведението на даден клас. Различни деца могат по различен начин да надграждат поведението на класа. Колкото повече деца се създадат за определен родител, толкова по-специализирано става поведението на детето. Децата също така могат и да променят наследено поведение, примерно, ако съществува родителски клас с метод draw(), които изрисува на екрана нещо, но при викане на детето, трябва да се рисува друго, то тогава детето може да презапише поведението на метода, така че да постигне собствените си цели.
Благодарение на това се получава т.нар в ООП *полиморфизъм*(polymorphism) – една и съща инструкция изпратена на различни обекти води до различно поведение, което е зависимо от характера на обекта, получаващ инструкциите.

Наследяването може да бъде следните видове:

1. Единично наследяване (Single Inheritance):

   Това е при клас, който наследява само от един клас, т.е. детето има само един родител. Изразено чрез синтаксис изглежда:

   ```
   class child_class_name : access_mode parent_class_name
   {
      . . .
   };
   ```

, където **child_class_name** е името на детето, което ще наследява, **parent_class_name** е името на родителския клас, от който се наследява, а **access_mode** е начина по който ще бъде наследен класа.

Изразено с пример в C++ изглежда:

```
class Vehicle  //родителски клас
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

class Car: public Vehicle //дете
{
   public:
    Car ()
    {
      cout << "This is Car is also a Vehicle" << endl;
    }

};
```

2. Множествено наследяване (Multiple Inheritance)::

Това е клас, който наследява от повече от един клас, т.е. детето има повече от един родител. Изразено чрез синтаксис изглежда:

```
class child_class_name : access_mode parent _name1, access_mode parent
_name2
{

   . . .

};
```

, където **child_class_name** е името на детето, което ще наследява, **parent _name1** и **parent _name2** са имената на родителските класове, от които се наследява, разделени със запетая, и задължително всеки от тях трябва да притежава **access_mode**, начина по който ще бъде наследен класа.

Изразено с пример в C++ изглежда:

```
class Vehicle //родител 1
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
```

```
class FourWheels //родител 2
{
  public:
    FourWheels ()
    {
      cout << "These are Four Wheels" << endl;
    }
};


class Car: public Vehicle, public FourWheels  //дете
{
   public:
    Car ()
    {
      cout << "This is Car is also a Vehicle with Four Wheels" << endl;
    }

};
```

3. Наследяване на слоеве (Multilevel Inheritance):

Това е клас, който наследява от клас, който вече е дете. Изразено чрез синтаксис изглежда точно като единично наследяване. Изразено чрез код изглежда:

```
class Vehicle //родител 1
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

class FourWheels: public Vehicle //дете 1
{
  public:
    FourWheels ()
    {
      cout << "This Vehicle has Four Wheels" << endl;
    }
};


class Car: public FourWheels  //дете 2
{
   public:
    Car ()
    {
      cout << "This is Car is also a Vehicle with Four Wheels" << endl;
    }

};
```

4. Наследяване чрез йерархия (Hierarchical Inheritance):

При този вид наследяване един родител може да има няколко деца. Изразено чрез синтаксис изглежда точно като единично наследяване. Изразено чрез код изглежда:

```
class Vehicle //родител
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

class Bus: public Vehicle //дете 1
{
  public:
    Bus ()
    {
      cout << " This is Bus is also a Vehicle " << endl;
    }
};


class Car: public Vehicle //дете 2
{
   public:
    Car ()
    {
      cout << "This is Car is also a Vehicle" << endl;
    }

};
```

5. Смесено (виртуално) наследяване (Hybrid (Virtual) Inheritance):

Този вид наследяване се имплементира чрез комбиниране на повече от един вид за наследяване, примерно комбиниране на Наследяване чрез Йерархия и Множествено Наследяване. Изразено чрез код би изглеждало:

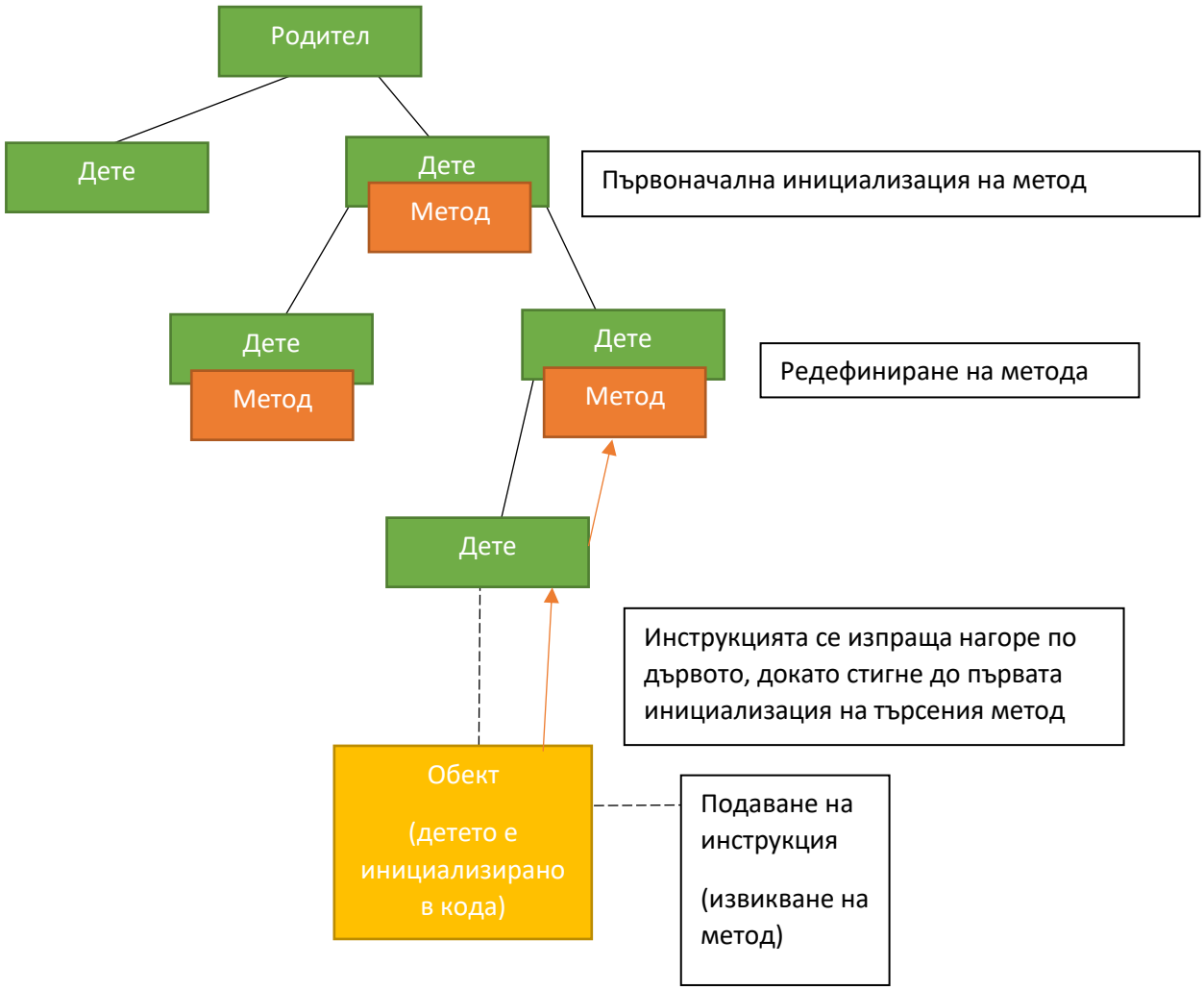
```
class Vehicle //родител
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

class Public_transport //родител
{
  public:
    Public_transport ()
    {
      cout << "This is a member of Public Transport" << endl;
    }
};
```

```
class Bus: public Vehicle, public Public_transport //дете 1
{
  public:
    Bus ()
    {
      cout << " This is Bus is also a Vehicle from Public Transport " <<
endl;
    }
};


class Car: public Vehicle //дете 2
{
   public:
    Car ()
    {
      cout << "This is Car is also a Vehicle" << endl;
    }

};
```

Подобно на таксономия, класицификацията на видовете, класова йерархия означава реализация на различни класове и връзките между тях: наследяване, разширяване, добавка на интерфейс и абстракция на друг клас.

Класовата йерархия няма ограничение за дълбочина, тоест може да съдържа неограничен брой елемента (класове). Полетата и методите се наследяват надолу по слоевете и могат да бъдат редефинирани ако се наложи, според изискванията на детето. Когато е изпратен метод към класова йерархия, тя се приема от класа, който е получил инструкцията, и се изпраща нагоре по дървото, докато стигна до търсения метод. Наименованието за това явление е **upcasting** (букв. преведено: хвърляне нагоре).

Графично изразено може да изглежда по следния начин:



Пример, изразен чрез код би изглеждал така:

```
class Parent
{
...
	public:
		void doSomething()
		{
			cout << "I am the parent!" << endl;
		}
};

class Child1: public Parent
{
	...
};
```

```
class Child2: public Parent
{
...
      public:
            void doSomething()
            {
                  cout << "I am the child!" << endl;
            }
};

int main()
{
      Child2 object;
      object.doSomething();
      return 0;
}
```

Резултатът, който ще бъде изведен е:

```
I am the child!
```

, тъй като сме редефинирали метода doSomething() в детето *Child2* на класа *Parent*, и това е първата инстанция на метода, която е открита.

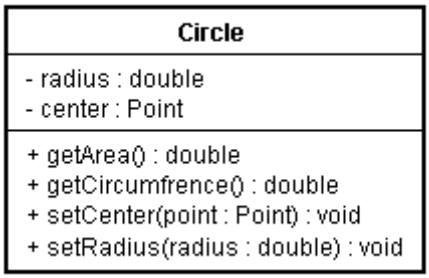
Теми 9 и 10. Оценка на качество на кода. Свързаност и структурираност на код || Качество на код: дублиращи се фрагменти. Целево-ориентиран проект.

За писане на качествен код е нужно да се спазват принципите за свързаност и еднородност, за премахване на дублиращи се фрагменти и за практическо коментиране. Добра практика е също да се умее да се визуализира програмата посредством UML класова диаграма.

Unified Modeling Language (UML) класова диаграма представлява графично изобразяване на взаимоотношенията между класовете в дадена програма. Нека за пример да притежаваме следния клас:

```
class Circle {
  private:
    double radius;
    Point center;
  public:
    void setRadius(double radius);
    void setCenter(Point center);
    double getArea();
    double getCircumfrence();
};
```

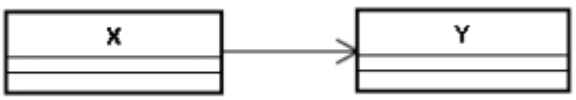
Визуално представяне на този клас в класова диаграма изглежда по следния начин:



При изобразяване на класове се използват следните стандарти: модификаторите за достъп се отбелязват с ‘+’ (**public**), ‘-‘ (**private**) и ‘#’ (**protected**). Статичните членове на класа са подчертани, а виртуалните функции се изписват с курсив. Целта на класовата диаграма е да се изобразят взаимотношенията между класовете, следвателно за тази цел са създадени следните връзки между класовете: Асоциация, Зависимост, Композиция, Съвкупност и Наследяване.

1. Асоциация (Клас X „познава“ клас Y)

   Един клас знае за съществуването на друг, като най-често съдържа показател към втория клас. Изобразява се по следния начин:



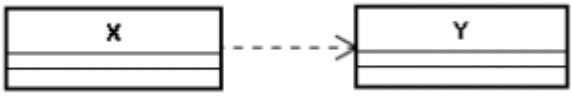
   Изразено с код, може да изглежда така:

```
Class X {
  Y *y_ptr; // показател към клас Y
```

```
  X(Y *y) : y_ptr(y) {} /*конструктора приема като аргумент показател
към клас Y */
  void SetY(Y *y) { y_ptr = y;   }
  void f()        { y_ptr->Foo();}
  . . .
  };
```

2. Зависимост (Клас X „използва“ клас Y)

Един клас зависи от друг, ако независимия клас е локална променлива в зависимия клас или класа се подава като аргумент на метод. Изобразява се по следния начин:



Изразено с код, може да изглежда така:

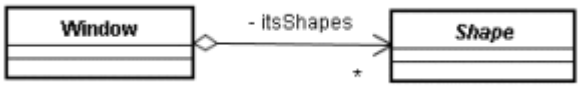
```
class X
{
 ...
   void f1(Y y)  {…;  y.Foo();       }
   void f2(Y *y) {…;  y->Foo();      }
   void f3(Y &y) {…;  y.Foo();       }
   void f4()     {Y y; y.Foo();  …   }
 ...
};
```

3. Съвкупност (Клас X „притежава“ клас Y)

При съвкупност един клас съдържа други класове. Той действа като един вид контейнер на други класове. Важното тука е, че класовете, които контейнера притежава, не зависят от жизнения цикъл на самия контейнер – с други думи казано, когато класът контейнер е унищожен, неговото съдържание остава. Пример с код може да се даде:

```
class Window
{
 public:
  //...
 private:
   vector<Shape> itsShapes;
}
```

Тука притежаваме клас *Window*, който в себе си съдържа *vector* от елементи тип “клас *Shape*”. С други думи казано, *Window* е контейнера, а вектора *itsShapes* е съдържанието на контейнера – формите, които може да притежава един прозорец. Изобразява се по следния начин:



4. Композиция (Клас X „притежава“ клас Y)

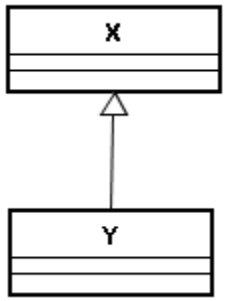
Композиция е по-развита форма на съвкупността. Разликата е, че при унищожаване на класът контейнер, неговото съдържание също се унищожава. Пример с код може да се даде:

```
class House
{
private:
      ...
     Room bedroom;
....
};
```

За да се разбере разликата между композиция и съвкупност, най-лесно може да се обясни по следния начин: при композицията има пряка връзка между контейнера и съдържанието му. Пример за съвкупност може да е Група (контейнер) и Студент (съдържание). Когато се премахне групата, студента си остава студент (няма пряка зависимост между двете). Пример за композиция може да се даде Къща (контейнер) и Спалня (съдържание). Няма как да съществува спалня без къща (съществува пряка зависимост).

5. Наследяване (Клас Y наследява клас X)

Наследяването в класовата диаграма няма разлика с вече изученото. Визуално се представя по следния начин:



Писането на качествен код визира връзка между свързаност (**Coupling**) и еднородност (**Cohesion**). Еднородност определя какво може да прави един клас. Нисък коефицент на еднородност означава че класа е широко приложим и може да се използва за много действия. Висок коефицент на еднородност притежава клас, който е много специализиран в действеията си. Пример за нисък коефицент на еднородност може да се даде:

```
class Staff
{
      private:
            string email;
      public:
            void checkEmail();
            void sendEmail();
            bool emailValidate();
            void PrintLetter();
};
```

За висок може да се даде:

```
class Staff
{
      private:
            double salary;
            string emailAddress;
      public:
            void setSalary(newSalary);
            double getSalary();
            void setEmailAddr(newEmail);
            string getEmailAddr();
};
```

Под ‘свързаност’ се визира взаимоотношенията между класовете, т.е. колко зависими са един от друг и до колко са свързани. За класове с нисък коефицент на свързаност промяната на нещо значително в един клас няма да има влияние върху другите класове. При висок коефицент на свързност такава промяна може да създаде големи проблеми в структурата на програмата, тъй като класовете са много зависими един от друг, следвателно промяна може да означава да се направи преобмисляне на логиката на цялата програма. Една качествена програма притежава **висок коефицент на еднородност** и **нисък коефицент на свързаност**.

При нарастване на програмата, определени идеи ще бъдат повторени по няколко пъти. Примерно: ако правите игра, ще ви е нужен код, който да изрисува различни графични елементи на екрана (да речем кораб или коршум от оръдие). Преди да е възможно да се нарисува кораба е нужно познание за това как се рисува пиксел (единна точка, съдържаща цвят, разположена чрез Декартова координатна система върху екрана). Също така ще трябва и код, които да използва пиксели за да се нарисуват елементите на играта (кораба, т.н). Този код ще бъде използван много често в играта - всеки път когато един елемент се движи, той ще трябва да бъде пренарисуван. Ако този код е сложен всеки път когато се налага да се нарисува кораба, програмата ще се напълни с фрагмент (откъс от код), който се дублира множество пъти. Това дублиране добавя ненужна сложнота на кода на програмата, и това превръща програмата в много трудна за разбиране. Препоръчително е да съществува стандарт за извършване на такива дублиращи се фрагменти, вместо да се оставя кода да повтаря един и същи откъс на много места. Под стандарт се разбира кода, който постоянно се повтаря, да бъде сложен в отделна функция. Това е нужно, защото, ако примерно трябва да се смени цвета на кораба, е много по-лесно да се извърши тази промяна във функция, така променяйки се при всяко викане, сравенено със смяна навсякъде, където кода се дублира. Отново пример с код: ако кораб се рисува като се прави кръг, който се попълва с цвят, е много по-лесно да се обединят двете действия във фунцкия.

сложно:

```
...
        circle(10, 10, 5);
        fillCircle(10, 10, Red);
...
```

разбираемо:

```
...
        dispayShip(10, 10, 5, Red);
...

void dispayShip(int x, int y, int size, string color)
{
    circle(x, y, size);
    fillCircle(x, y, color);
}
```

При отделяне във функция е много важно действието на функцията да стане ясно от самото име. Колкото по-голяма става програмата, толкова повече време ще се прекарва в преглеждане на код, от колкото в писането му. Поради тази причина е нужно действията на функциите да са ясни от името, улеснявайки много работата при поправяне на грешки в кода.

Голяма част от качествения код са коментарите. Доброто коментиране в програмата е много по-сложно действие, от колкото звучи. Добър коментар означава следното: да може веднага да отговори на въпроси, които читателя може да има. Предполага се че читателя има опит с програмирането, заради това коментари от този тип:

```
//деклариране на променлива i и задаване на стойност на променливата 5
int i = 5;
```

са ненужни. Въпросите, които читателя си задава, най-често са от тип: "Това е много странен начин за решаване на този казус. Защо е направен така?" или "Какви са приемливите стойности, които може да приеме тази фунцкия? Какви са им приложенията?". Пример за приемливо коментиране, които може да бъде написано е следното:

```
/* Проверка дали подадения int number е четно число. Ако number е четно,
връща се 1. Иначе се връща 0. При подаване на отрицателно число, фунцкията
връща 0.*/
int isEven(int number)
{
    if(number > 0)
    {
        if(number%2 == 0)
        {
            return 1;
        }else
        {
            return 0;
        }
    }else
    {
        return 0;
    }
}
```

От коментара става ясно веднага какво е действието на функцията, какъв аргумент е валиден, какви стойности се връщат и какво се случва при подаване на невалиден аргумент. Читателя може да разбере как действа фунцкията без да гледа имплементацията ѝ, което е много важно при редактиране на много сложен и/или дълъг код.