11. Проектиране на обектно-структуриран код

Софтуерът не е нещо което се прави само веднъж и остава така за винаги, а той постояно се допълва, променя, приспособява и това доста често става от различни хора. Това е причината защо един код трябва да бъде обектно-структуриран и добре описан, за да може всеки който го погледне лесно да разбере за какво става въпрос.

Две неща са доста важни за квалитета на кода:

-Свързаноста

-Еднородноста

Свързаноста претставлява връзка помежду отделните единици на една програма. Ако два класа са свързани помежду си с повече общи характеристики тогава става дума за плътна свързаност. В един добре структуриран код свързаноста трябва да се избягва възможно повече. Това се осъществява чрез наследяването и абстракцията. Това би овозможило лесно разбиране и промяна на класовете без да има влияние нвърху други класове и лесно подържане на кода.

Еднородноста се отнася на броят и еднообразието на задачите за които една единица е отговорна, като ако една единица е отговорна за само една логическа задача, става дума за висока еднородност. Еднородноста се прилага върху класовете и методите в един код. Това овозможава лесно разбиране на това какво прави един клас или метод, като те използват описателни имена.

По точно казано:

-Един метод трябва да бъде одговорен за една добре описана задача

-Един клас трябва да описва един логически обект

За да можем да съставим един добре структуриран код можем да си поставим няколко вупроса като на пример:

-Каде трябва да добавим метод (кой клас).

-Дали един метод твърде голям? (Дали извършва повече от една логична задача)

-Кой трябва да достъпва данните на един клас?

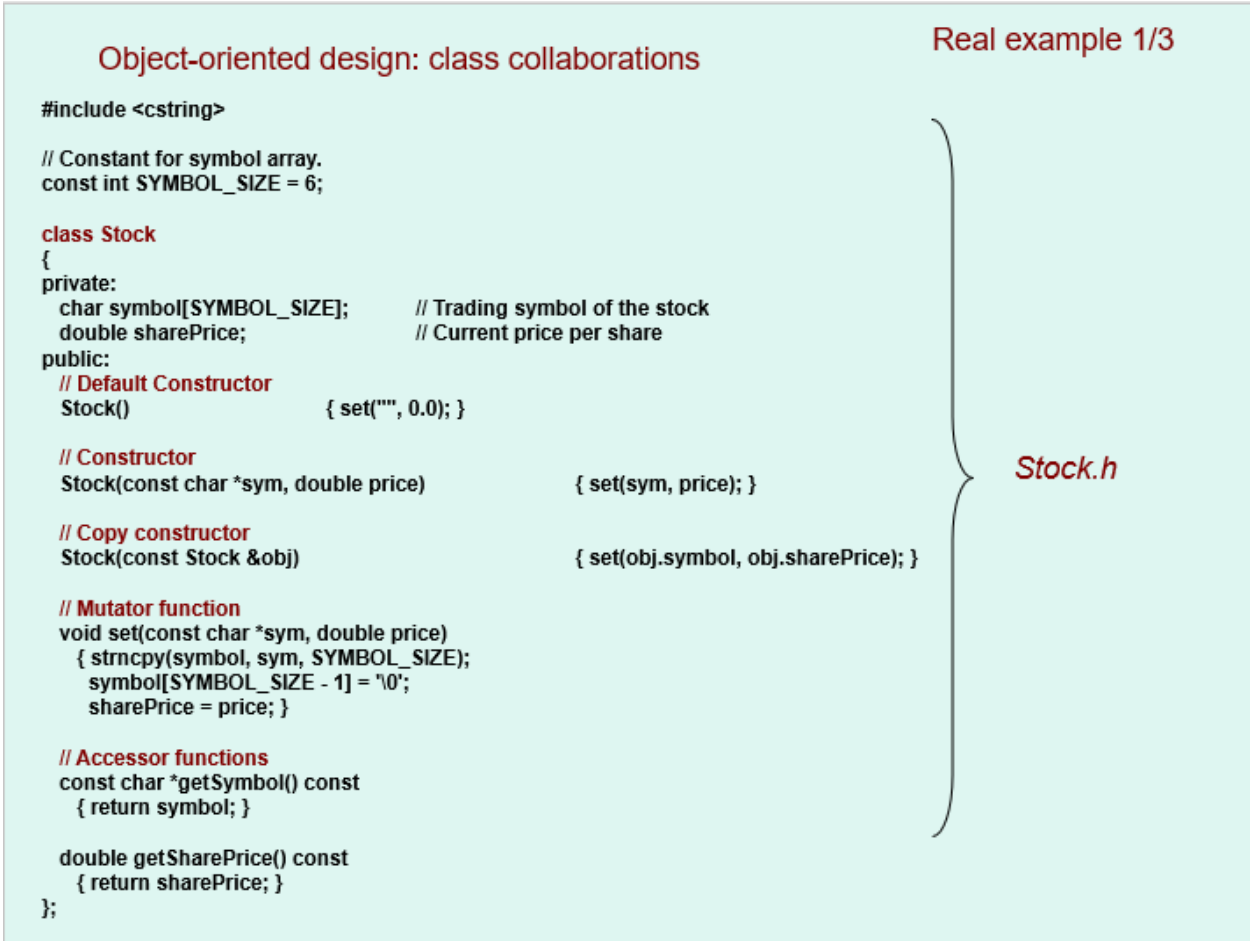
-Дали един клас е твърде сложен? (Дали описва повече от един логичен обект)

-Трябва ли да добавим коментар?

-Трябва ли да направим повече различни тествания на самия код?

Също така от голямо значение е **локализиране на промяната.** Това означава при една промяна да се повлияе върху възможно по малко класове.

По-долу в примера е даден един добре структуриран и разбираем код.



Object-oriented design: class collaborations

```
#include "Stock.h"

class StockPurchase
{
private:
   Stock stock;          // The stock that was purchased. So, include the stock object
   int shares;           // The number of shares
public:
   // The default constructor sets shares to 0. The stock
   // object is initialized by its default constructor.
   StockPurchase()
     { shares = 0; }

   // Constructor
   StockPurchase(const Stock &stockObject, int numShares)
     { stock = stockObject;
       shares = numShares; }

   // Accessor function
   double getCost() const
     {  return shares * stock.getSharePrice(); }
};
```

Stockpurchase.h

Object-oriented design: class collaborations

```
// Stock trader program
#include "Stock.h"
#include "StockPurchase.h"
using namespace std;

int main()
{
   int sharesToBuy;                                   // Number of shares to buy

   // Create a Stock object for the company stock. The trading symbol is XYZ and the stock is currently
   // priced at $9.62 per share.
   Stock xyzCompany("XYZ", 9.62);

   // Display the symbol and current share price.
   cout << setprecision(2) << fixed << showpoint;
   cout << "XYZ Company's trading symbol is " << xyzCompany.getSymbol() << endl;
   cout << "The stock is currently $" << xyzCompany.getSharePrice() << " per share.\n";

   // Get the number of shares to purchase.
   cout << "How many shares do you want to buy? ";
   cin >> sharesToBuy;

   // Create a StockPurchase object for the transaction.
   StockPurchase buy(xyzCompany, sharesToBuy);

   // Display the cost of the transaction.
   cout << "The cost of the transaction is $"
        << buy.getCost() << endl;
   return 0;
}
```

Program output:

```
XYZ Company's trading symbol is XYZ
The stock is currently $9.62 per share.
How many shares do you want to buy? 100[Enter]
The cost of the transaction is $962.00
```

12. Класове и обекти: разделяне на декларация и дефиниция. Създаване и унищожаване на обекти. Структура и обект.

13.Конструктори и деструктори. Видове конструктори (виртуални и статични). Методи на клас.

Класове

Класът е шаблон с полета и методи по който се създава обект. Класът е тип, дефиниран от потребителя.

Дефиниция

• Дефиницията на класа се състои от две части:

Заглавна част, която се състои от ключовата дума class и идентификатор, обозначаващ името на класа.

Тяло на класа, което е затворено във фигурни скоби.

• Дефиницията на класа трябва да бъде последвана от точка и запетая или от списък от дефиниции на променливи.

```
1. class Point { /* ... */ };
2. class Rectangle { /* ... */ } r1 ,r2;
```

В тялото на класа се дефинира списъкът от членове на класа и нивото на достъп до тях.

• Класът са смята за дефиниран когато достигнем края на тялото на класа. Когато класът е дефиниран всички негови членове са известни.В следния случай е деклариран и дефиниран класут Point.

```
1. class Point {
2. double x _;
3. double y_;
4. public:
5. void set _ x(double x);
6. void set _y(double y);
7. double get_x(void) { return x_ ;}
8. double get_y(void) { return y_ ;}
9. };
```

Декларация

• Възможно е да се декларира клас без да се дефинира. В следният пример е деклариран класът Point, без той да е дефиниран.

```
class Point;
```

• Когато класът е деклариран, но не е дефиниран, възможната му употреба е силно ограничена.

Обекти

Декларация

• При деклариране на променлива от типа на даден клас се декларира обект (инстанция) от класа.

```
1. Point p;
2. Point * ptr =&p;
```

Инициализация

Задаване на конкретни стойности на полетата на обекта.

```
1. p. set_x (42.0);
2. ptr->set_x (42.0);
```

Структура. Сравнение с клас.

Структурата претставлява почти едно и също нещо като класът, обаче те се различават по доста неща.

Структура

-Примитивен тип. Подаването на параметри става чрез копие (промените се отразяват само на копието) (C#)

-Не подържат наследяването (C#)

-Има само полета(C#)

-Достъпа до полетата по подразбиране е public (разлики в C++)

-Обектите и се създават в стековата памет(C#)

Клас

-Референтен тип. Подаването на параметри е чрез референция (промените се отразяват върху оригинала) (C#)

-Подържат наследяването(C#)

-Имат и полета и методи(C#)

-Достапа по подразбиране е private (разлики в C++)

-Обектите му се създават в heap паметта(C#)

Конструктори и деструктори

Създаването на обект става чрез **конструктор**.

-Конструкторът има същото име като класа

-Няма тип на връщан резултат и не връща резултат

-Може да бъде презапишан т.е. могат да се създадат повече от един конструктори

Видове конструктори

Конструктор по подразбиране – конструктор без тяло и параметри. Не е нужно да се декларира и дефинира конструктор по подразбиране. Него компилатора сам го създава. Той би изглеждал така:

Point();

При създаване на друг конструктор, конструктора по подразбиране не е вече познат за компилатора и той трябва също така да бъде създаден, като в тялото му се задават стойности по подразбиране на полетата на класа.

```
1. Point(){
2.     x=5;
```

```
3.     y=5;
4.  };
```

Параметризиран конструктор – конструктор с параметри и тяло който овозможава един обек да бъде инициализиран при създаването му. Той се използва за инициализиране на различни обекти с различни данни.

```
1.  Point::Point (int corX, int corY){
2.  x=corX;
3.  y=corY;
4.  };
```

Copy конструктор – конструктор който инициализира даден обект възползвайки се от друг опект от същия тип подаден като параметар.

```
1.  Point(const Point &p2) {
2.   x = p2.x;
3.   y = p2.y;
4.  }
```

Конструктор за преобразуване (conversion) – конструктор който може да бъде извикан с единичен аргумент.

```
1.  class Test
2.  {
3.   private:
4.     int x;
5.   public:
6.     Test(int i) {x = i;}
7.     void show() { cout<<" x = "<<x<<endl; }
8.  };
9.
10.int main()
11.{
12. Test t(20);
13. t.show();
14. t = 30; // conversion constructor is called here.
```

```
15. t.show();
16. getchar();
17. return 0;
18.}
```

Explicite конструктор – конструктор дефиниран с думата **explicite** който не може да бъде извикан само с единичен аргумент.

```
1.  class A
2.  {  public:
3.     explicit A(int);
4.  };
```

Унищожаването (изтриването) на обект става чрез **деструктор.**

-Обикновено се използва в края на кода когато програмата трябва да приключи.

-В един клас не може да имаме повече от един деструктор.

-Декларирането му става по същия начин като и деструткора с разлика че в началото има знак ‘~’.

-Няма параметри.

-Освобождава паметта заемана от обекта

~Point();

Методи на клас

Методите или член-функциите реализират множеството от операции, които могат да се извършват върху обектите от даден клас. Идеологията на обектно-ориентираното програмиране налага правилото, че всички операции върху член-променливите трябва да се извършват от член-функциите на класа. Методите, както и полетата на един клас, имат модификатор за достъп и връщан резултат. В следния пример е дадена декларация и дефиниция на метод.

```
1. class Point {
2.  //...
3.  void set_x( double x);
4.  };
```

```
1. void Point :: get_x(){
2. return x;
3. }
```

Метода както се вижда в примера може да има или да няма параметри.

Методите се извикват по следния начин:

```
1. Point p1;
2.  p1.get_x();
```