

Designing classes In a OOP program

How to write classes in a way that they are easily understandable, maintainable and reusable

Software changes

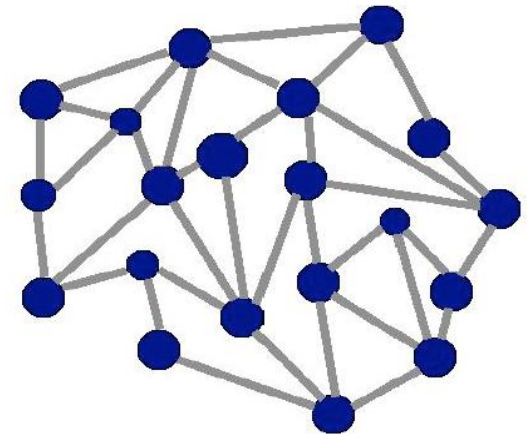
- Software is not like a novel that is written once and then remains unchanged.
- Software is extended, corrected, maintained, ported, adapted...
- The work is done by different people over time (often decades).

Качество на кода

2 важни концепции за качество на кода:
Кохезия и свързаност

- Силна кохезия (Strong Cohesion)
- Слаба свързаност (Loose Coupling)

Coupling



Coupling

- Coupling refers to links between separate units of a program.
- If two classes depend closely on many details of each other, we say they are *tightly coupled*.
- We aim for *loose coupling*.

Loose coupling

Loose coupling makes it possible to:

- understand one class without reading others;
- change one class without affecting others.
- Thus: improves maintainability.

Cohesion

- Cohesion refers to the the number and diversity of tasks that a single unit is responsible for.
- If each unit is responsible for one single logical task, we say it has *high cohesion*.
- Cohesion applies to classes and methods.
- We aim for high cohesion.

High cohesion

High cohesion makes it easier to:

- understand what a class or method does;
- use descriptive names;
- reuse classes or methods.

Cohesion

- Of methods: a method should be responsible for one and only one well defined task.
- Of Classes: a class should represent one single, well defined entity.

Code duplication

Code duplication

- is an indicator of bad design,
- makes maintenance harder,
- can lead to introduction of errors during maintenance.

Localizing change

- One aim of reducing coupling and responsibility-driven design is to localize change.
- When a change is needed, as few classes as possible should be affected.

Основни въпроси при проектирането

- Колко голям трябва да е един метод?
- Колко голям трябва да е един клас?
- Можете ли да отговорите в смисъла на кохезия и свързаност ???

Кохезия

- За метод: Всеки един метод трябва да е отговорен за една и само една добре дефинирана задача.
- За класове: Всеки клас трябва да описва един добре дефиниран обект в проблемната област.

Основни въпроси при проектирането

- Колко голям трябва да е един метод?
- Колко голям трябва да е един клас?
- Можете ли да отговорите в смисъла на кохезия и свързаност ???

Принципи на проектирането

- Един метод е прекалено голям, когато изпълнява повече от една задача;
- Един клас е прекалено сложен, ако описва повече от една логическа единици.

Review

- Programs are continuously changed.
- It is important to make this change possible.
- Quality of code requires much more than just performing correct at one time.
- Code must be understandable and maintainable.

Review

- Good quality code avoids duplication, displays high cohesion, low coupling.
- Coding style (commenting, naming, layout, etc.) is also important.
- There is a big difference in the amount of work required to change poorly structured and well structured code.

S. O. L. I. D.

- Принцип на едноличната отговорност - Single Responsibility Principle (SRP);
- Отворено-затворен принцип - Open Closed Principle (OCP);
- Принцип на субституцията (Лисков) - Liskov's Substitution Principle (LSP);
- Принцип за разделяне на интерфейсите - Interface Segregation Principle (ISP);
- Принцип за обръщане на зависимостта - Dependency Inversion Principle (DIP).

Software development is not a Jenga game.



Принцип на едноличната отговорност

- Един модул трябва да има само една причина да се променя.
- Този принцип гласи, че ако имаме две неща да се променят в един клас, трябва да се разделят функционалност в два класа.
- Всеки клас ще се справи само с една отговорност и за бъдеще, ако ние трябва да направим една промяна, ние ще го направим в класа, в който се отнася.

Just because you can, doesn't mean you should.



```
public class EmployeeService
{
public void SaveEmployeeInfo(Employee e)
{// To do something}
public void UpdateEmployeeInfo(int empId, Employee e)
{//To do Something

}
public Employee GetEmployeeInfo(int empID)
{// To do something
}

public void MAPEmployee(SqlDatareader reader)
{// To do something
}
}
```

ЛОШ

ДОБЪР

```
public class EmployeeService
{
public void SaveEmployeeInfo(Employee e)
{// To do something}
public void UpdateEmployeeInfo(int empId,
Employee e)
{//To do Something

}
public Employee GetEmployeeInfo(int empID)
{// To do something
}
}
public class
ExtendEmployeeService:EmployeeService
{

public void MAPEmployee(SqlDatareader
reader)
{// To do something
}
```

Отворено-затворен принцип

- Един модул трябва да бъде отворен за разширение, но затворен за модификация.
- Ако има нови изисквания към софтуера, тогава няма да модифицираме вече работещия код, а ще имплементираме нов.

Open chest surgery is not needed when putting on a coat.



```
// Open-Close Principle - Bad example
class GraphicEditor {

    public void drawShape(Shape s) {
        if (s.m_type==1)
            drawRectangle(s);
        else if (s.m_type==2)
            drawCircle(s);
    }
    public void drawCircle(Circle r) {....}
    public void drawRectangle(Rectangle r) {....}
}

```

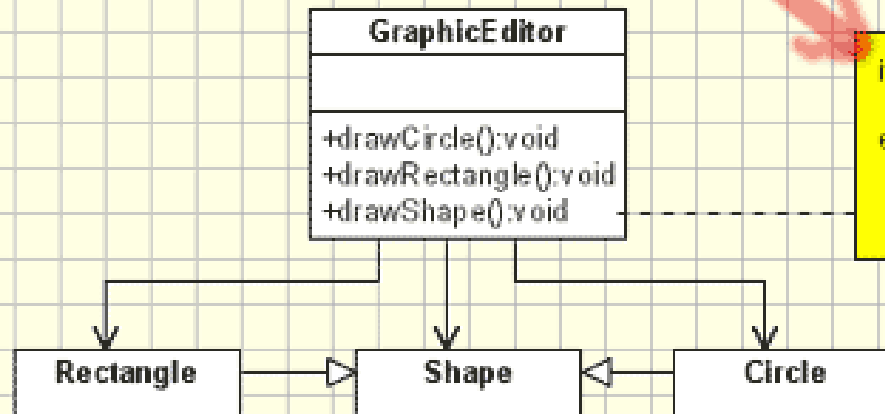
```
class Shape {
    int m_type;
}

class Rectangle extends Shape {
    Rectangle() {
        super.m_type=1;
    }
}

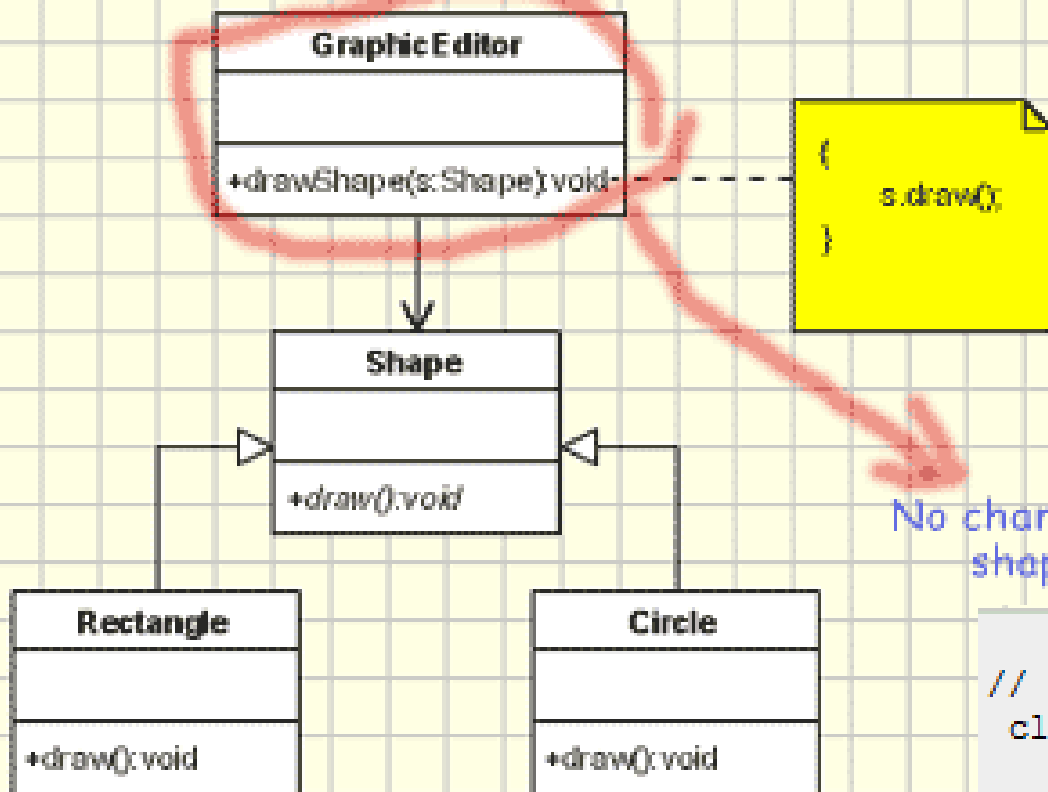
class Circle extends Shape {
    Circle() {
        super.m_type=2;
    }
}

```

When a new shape is added this should be changed (and this is bad!!!)



if (s.m_type == 1)
drawRectangle();
else if (s.m_type == 2)
drawCircle();



No changes required when a new shape is added (Good!!!).

```

// Open-Close Principle - Good example
class GraphicEditor {
    public void drawShape(Shape s) {
        s.draw();
    }
}

class Shape {
    abstract void draw();
}

class Rectangle extends Shape {
    public void draw() {
        // draw the rectangle
    }
}
  
```

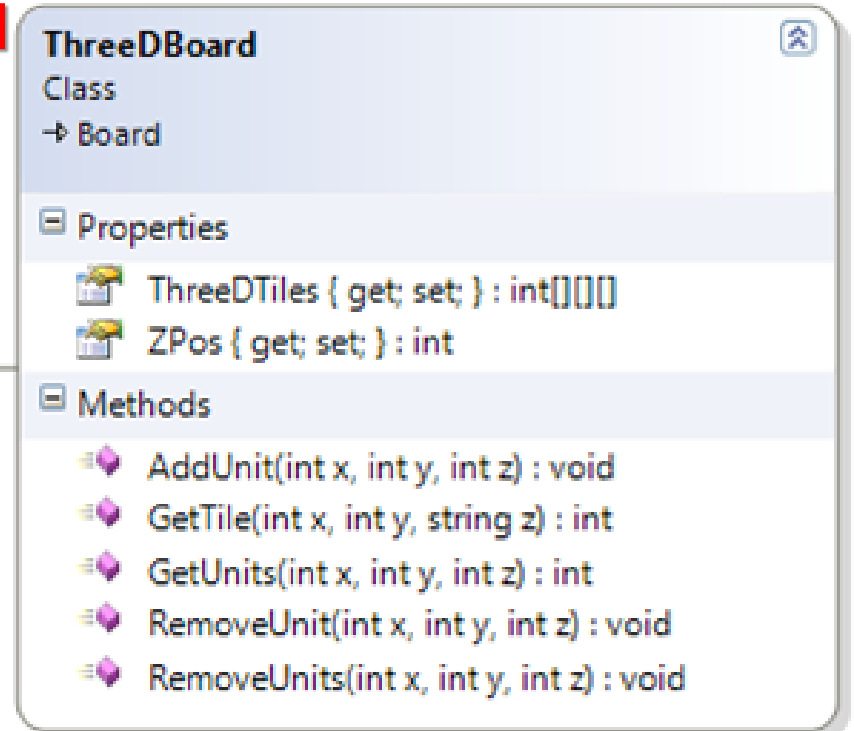
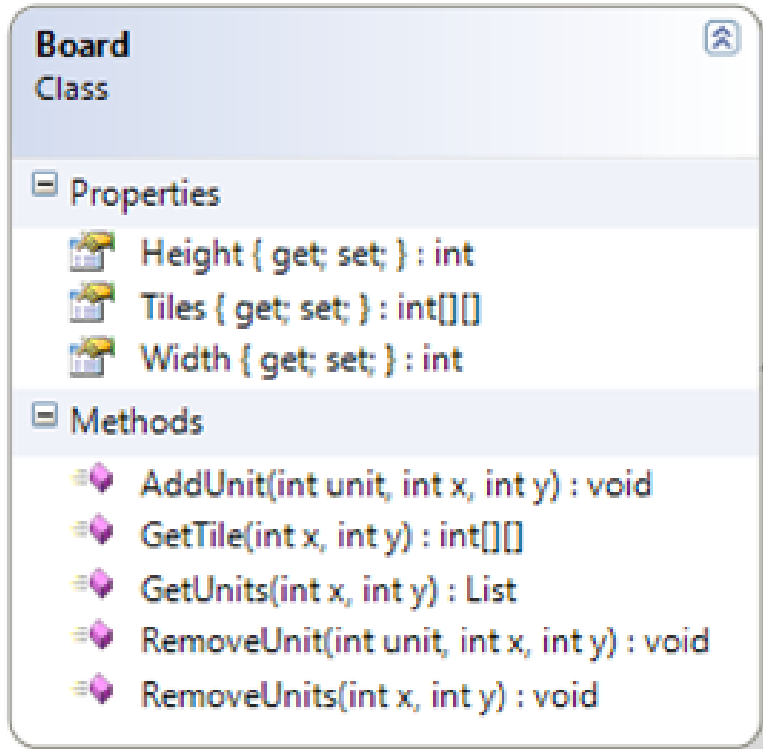
Принцип на субституцията (Лисков)

- Наследниците трябва да бъдат заместими от техните базови класове.
- Правилна йерархия на класовете.
- Методи или функции, които използват тип от базов клас, трябва да могат да работят и с обекти от наследниците без да се налага промяна.

If it looks like a duck, quacks like a duck, but needs batteries – you probably have the wrong abstraction



ЛОШ



Instead of extending Board, ThreeDBoard should be composed of Board objects. One Board object per unit of the Z axis.

Принцип за разделяне на интерфейсите

Classes should not depend on interfaces that they not use.

- The meaning of this phrase is to avoid tying a client class to a big interface if only a subset of this interface is really needed.
- Many times you see an interface which has lots of methods.
- This is a bad design choice since probably a class implementing it will infringe Single Responsibility Principle and for many other issues which arises when interfaces grow.

You want me to plug this in, where?



```

// interface segregation principle - bad example
interface IWorker {
    public void work();
    public void eat();
}

class Worker implements IWorker{
    public void work() {
        // ....working
    }
    public void eat() {
        // ..... eating in launch break
    }
}

class SuperWorker implements IWorker{
    public void work() {
        //.... working much more
    }

    public void eat() {
        //.... eating in launch break
    }
}

class Manager {
    IWorker worker;

    public void setWorker(IWorker w) {
        worker=w;
    }

    public void manage() {
        worker.work();
    }
}

```

```

// interface segregation principle - good example
interface IWorker extends Feedable, Workable {
}

interface IWorkable {
    public void work();
}

interface IFeedable{
    public void eat();
}

class Worker implements IWorkable, IFeedable{
    public void work() {
        // ....working
    }

    public void eat() {
        //.... eating in launch break
    }
}

class Robot implements IWorkable{
    public void work() {
        // ....working
    }
}

class SuperWorker implements IWorkable, IFeedable{
    public void work() {
        //.... working much more
    }

    public void eat() {
        //.... eating in launch break
    }
}

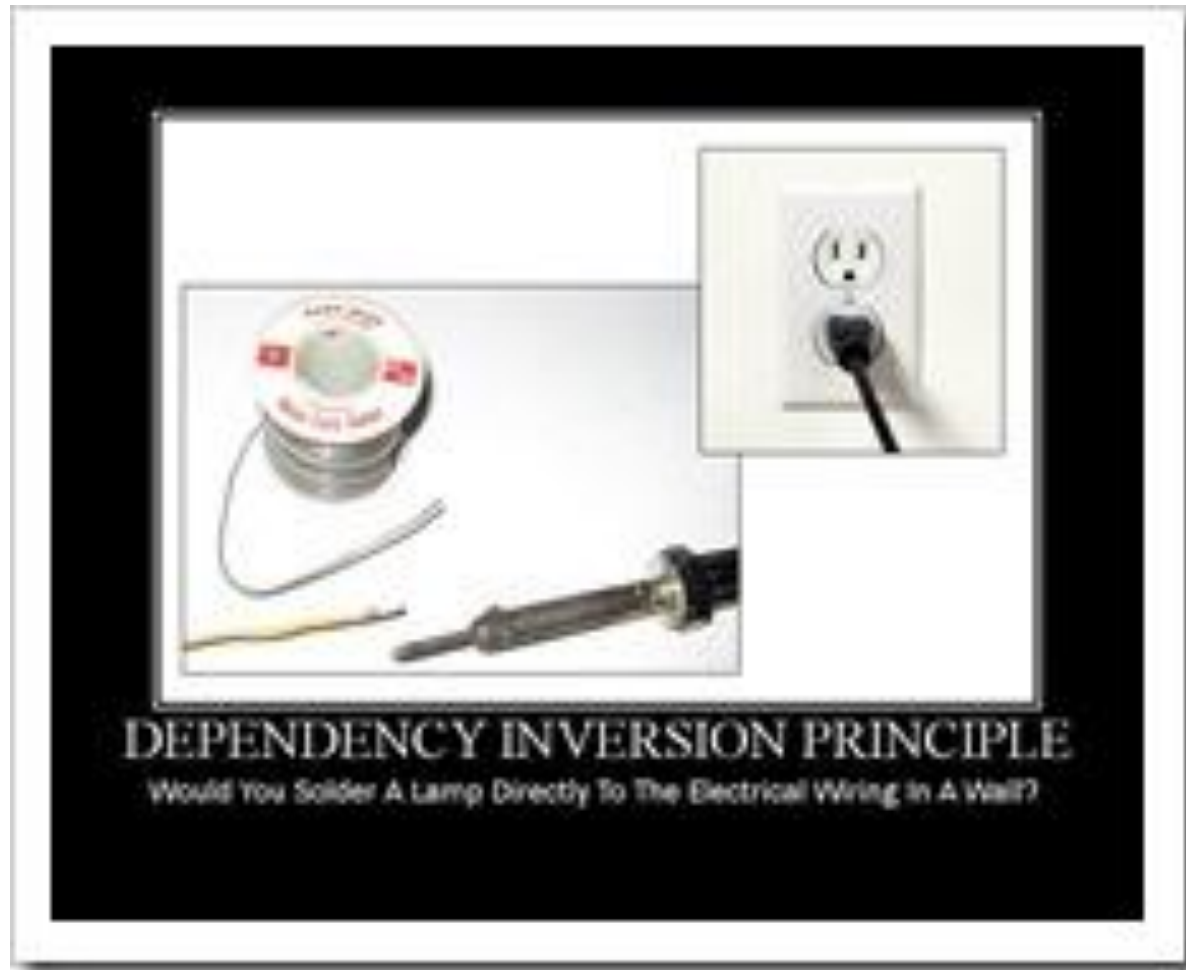
```

Принцип за обръщане на зависимостта

- Depend upon abstractions. Do not depend upon concretions.
- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

High Level Classes --> Abstraction Layer --> Low Level Classes

Would you solder a lamp directly to the electrical wiring in a wall?



```
// Dependency Inversion Principle - Bad example
class Worker {
    public void work() {
        // ....working
    }
}

class Manager {
    Worker m_worker;

    public void setWorker(Worker w) {
        m_worker=w;
    }

    public void manage() {
        m_worker.work();
    }
}

class SuperWorker {
    public void work() {
        //.... working much more
    }
}
```

```
// Dependency Inversion Principle - Good example
interface IWorker {
    public void work();
}

class Worker implements IWorker{
    public void work() {
        // ....working
    }
}

class SuperWorker implements IWorker{
    public void work() {
        //.... working much more
    }
}

class Manager {
    IWorker m_worker;

    public void setWorker(IWorker w) {
        m_worker=w;
    }

    public void manage() {
        m_worker.work();
    }
}
```