Lecture:

Designing classes

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).

OOA, OOD, and OOP

Object-oriented methods may be applied to different phases in the software life-cycle:

e.g., analysis, design, implementation, etc.

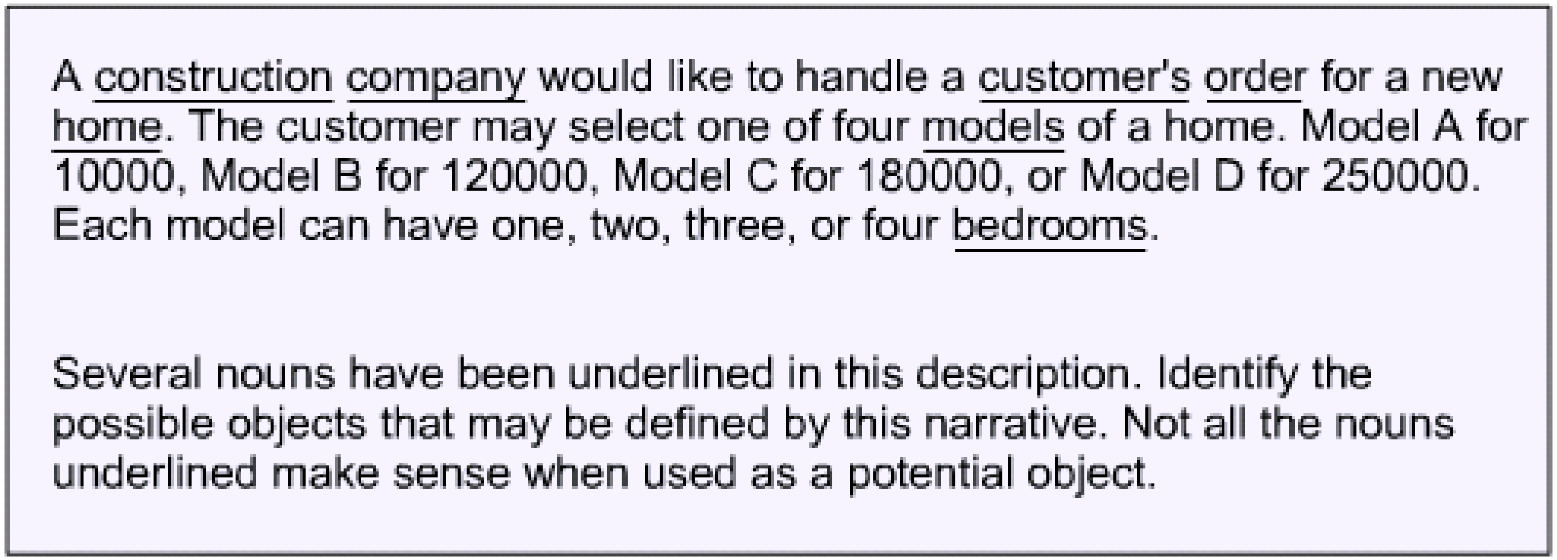
- **OO analysis (OOA)** is a process of discovery

Where a development team models and under-stands the requirements of the system

- **OO design (OOD)** is a process of invention and adaptation

Where the development team creates the abstractions and mechanisms necessary to meet the system's behavioral requirements determined during analysis

Designing classes



A construction company would like to handle a customer's order for a new home. The customer may select one of four models of a home. Model A for 10000, Model B for 120000, Model C for 180000, or Model D for 250000. Each model can have one, two, three, or four bedrooms.

Several nouns have been underlined in this description. Identify the possible objects that may be defined by this narrative. Not all the nouns underlined make sense when used as a potential object.

Abstraction and modularization

- **Abstraction** is the ability to ignore details of parts to focus attention on a higher level of a problem.
- **Modularization** is the proce dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.

Divide-and-conquer

The Role of Abstraction

- "We (humans) have developed an exceptionally powerful technique for dealing with complexity. We abstract from it. Unable to master the entirety of a complex object, we choose to ignore its inessential details, dealing instead with the generalized, idealized model of the object." (Wulf)

Role of Decomposition

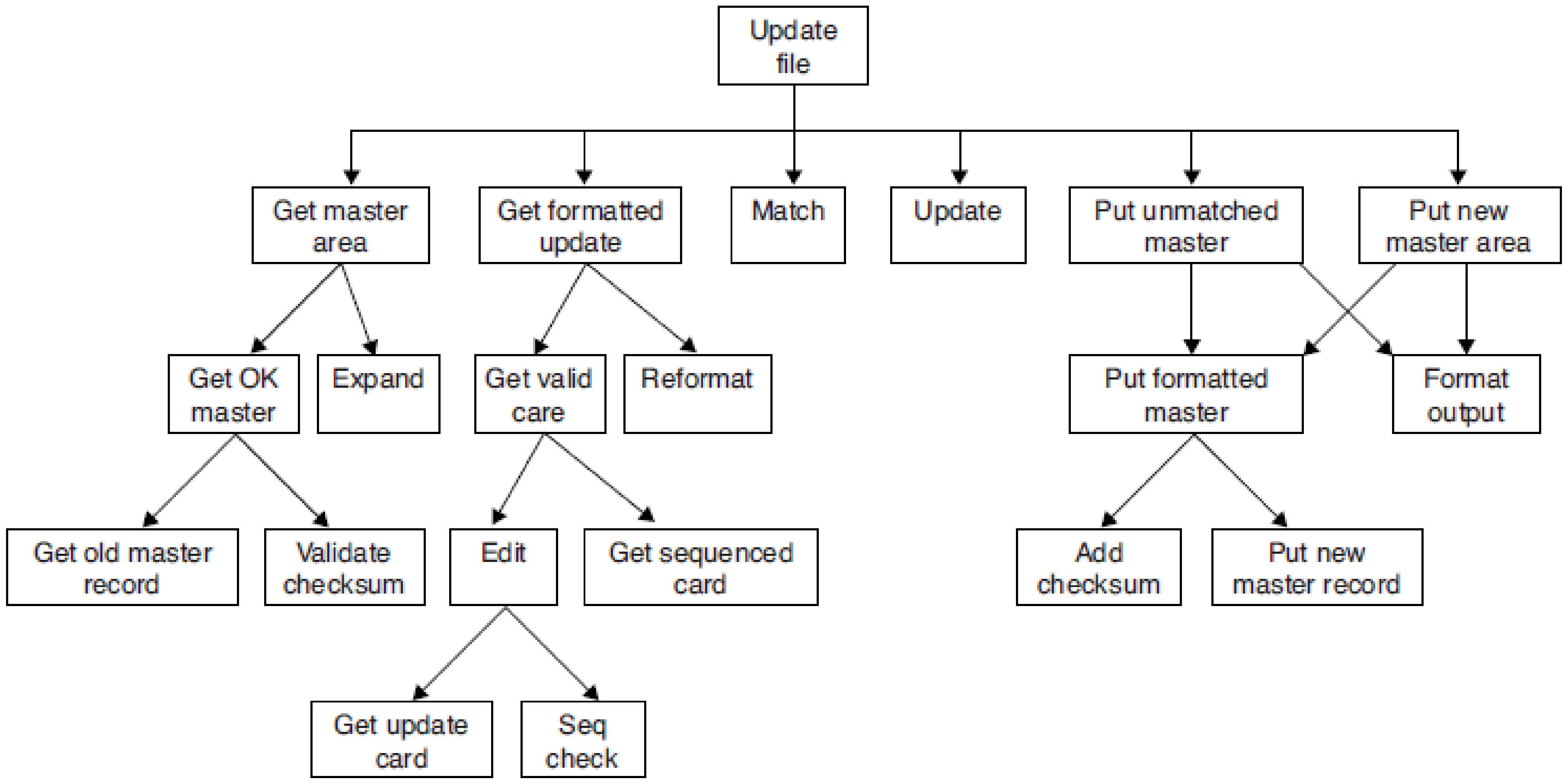
- When designing a complex software system, it is essential to decompose it into smaller and smaller parts, each of which we may then refine independently.

- divide et impera (divide and rule)

Algorithmic Decomposition

- Decomposition as a simple matter of algorithmic decomposition, wherein each module in the system denotes a major step in some overall process.

Example

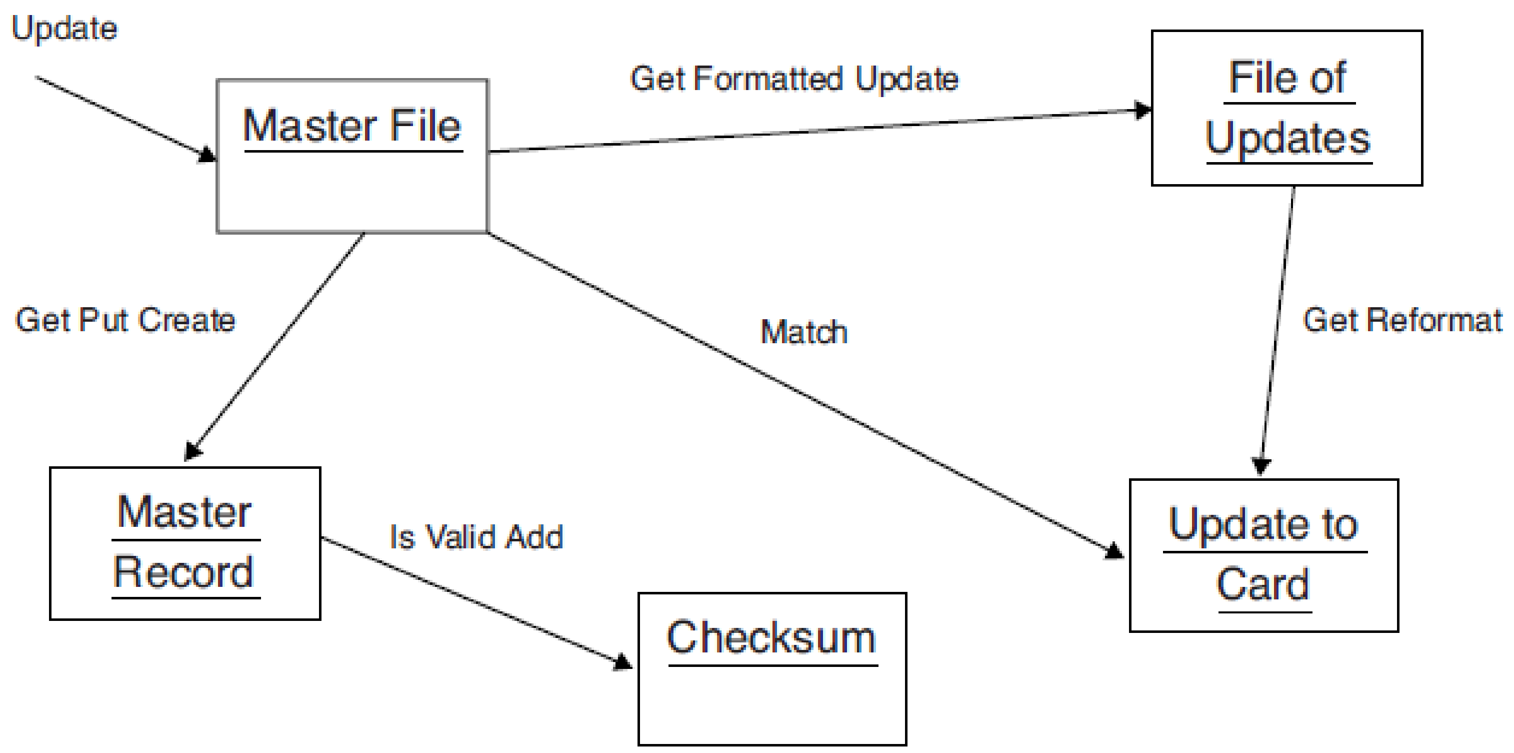

Update file
Get master area
Get formatted update
Match
Update
Put unmatched master
Put new master area
Get OK master
Expand
Get valid care
Reformat
Put formatted master
Format output
Get old master record
Validate checksum
Edit
Get sequenced card
Add checksum
Put new master record
Get update card
Seq check

Object-Oriented Decomposition

- Decomposed the system according to the key abstractions in the problem domain.

- We view the world as a set of autonomous agents that collaborate to perform some higher-level behavior.

Example


Update
Master File
Get Formatted Update
File of Updates
Get Put Create
Match
Get Reformat
Master Record
Is Valid Add
Checksum
Update to Card

Good OO design

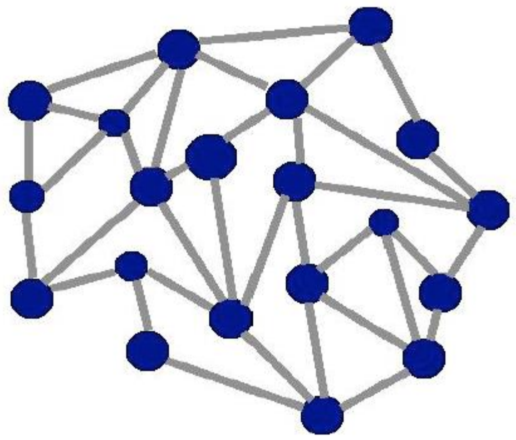
- The most important promise of OO is improve modularization
  - that is, enhance maintainability
  - while still supporting functional decomposition (which is good for cohesion)
  - while also promoting re-use
- Good OO = high cohesion + low coupling

Качество на кода

2 важни концепции за качество на кода: Кохезия и свързаност

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

Coupling



Couplin

- 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 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 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.

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)).

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 направи в класа, в който се отнася.

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

SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

single responsibility principle - bad example

```
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
}
}
```

single responsibility principle - good example

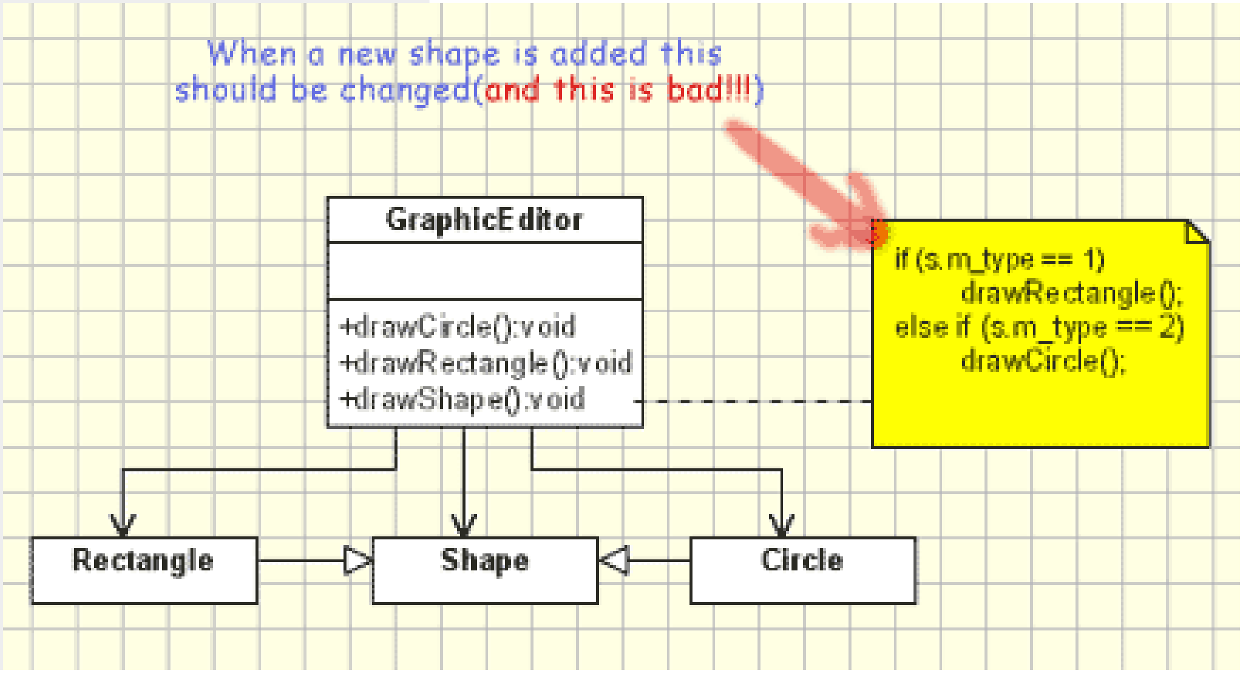
```
public class EmployeeService
{
        public void SaveEmployeeInfo(Employee e)
        {// To do something}
        public void UpdateEmployeeInfo(int empId, Employee e)
        {//To do Something }
        public Employee GetEmployeeInfo(int empID)
        {// To do something }
}

public class ExtendEmployeeService extend EmployeeService
{
        public void MAPEmployee(SqlDatareader reader)
        {// To do something}
}
```

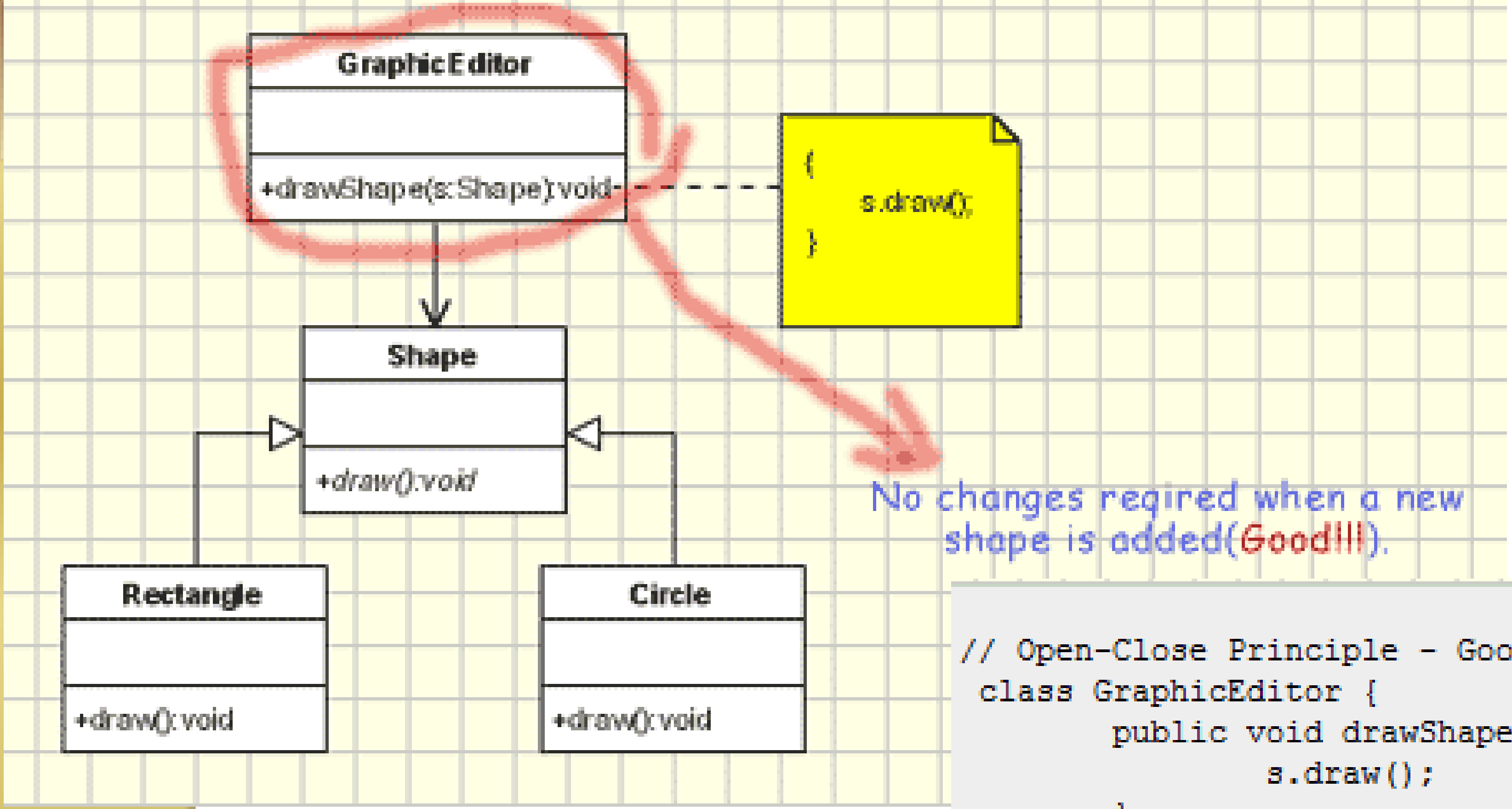
Отворено-затворен принцип

- 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!!!)
GraphicEditor
+drawCircle():void
+drawRectangle():void
+drawShape():void
if (s.m_type == 1) drawRectangle(); else if (s.m_type == 2) drawCircle();
Rectangle
Shape
Circle



```
// 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)

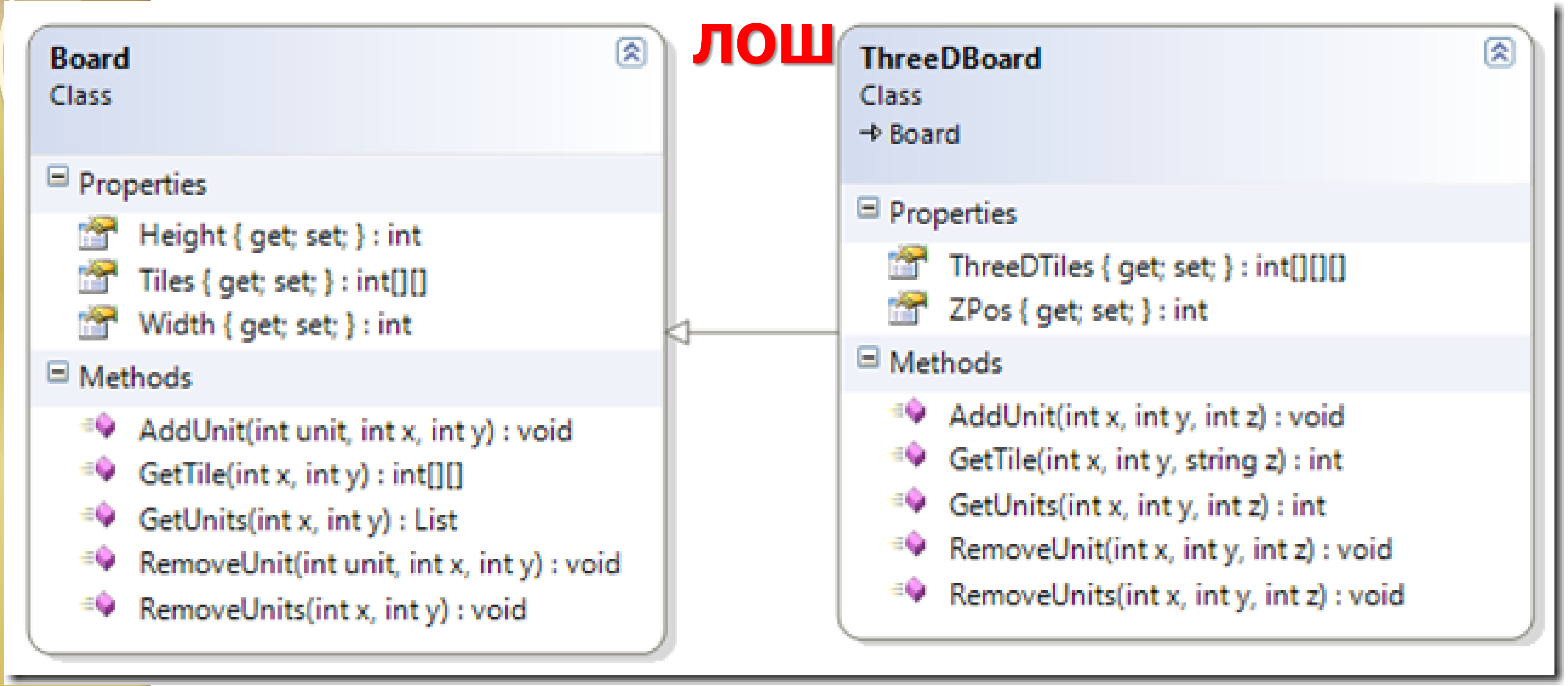
- Наследниците трябва да бъдат заместим от техните базови класове.

- 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

LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction



Board
Class

Properties
- Height { get; set; } : int
- Tiles { get; set; } : int[][]
- Width { get; set; } : int

Methods
- AddUnit(int unit, int x, int y) : void
- GetTile(int x, int y) : int[][]
- GetUnits(int x, int y) : List
- RemoveUnit(int unit, int x, int y) : void
- RemoveUnits(int x, int y) : void

ThreeDBoard
Class
→ Board

Properties
- ThreeDTiles { get; set; } : int[][][]
- ZPos { get; set; } : int

Methods
- AddUnit(int x, int y, int z) : void
- GetTile(int x, int y, string z) : int
- GetUnits(int x, int y, int z) : int
- RemoveUnit(int x, int y, int z) : void
- RemoveUnits(int x, int y, int z) : void

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

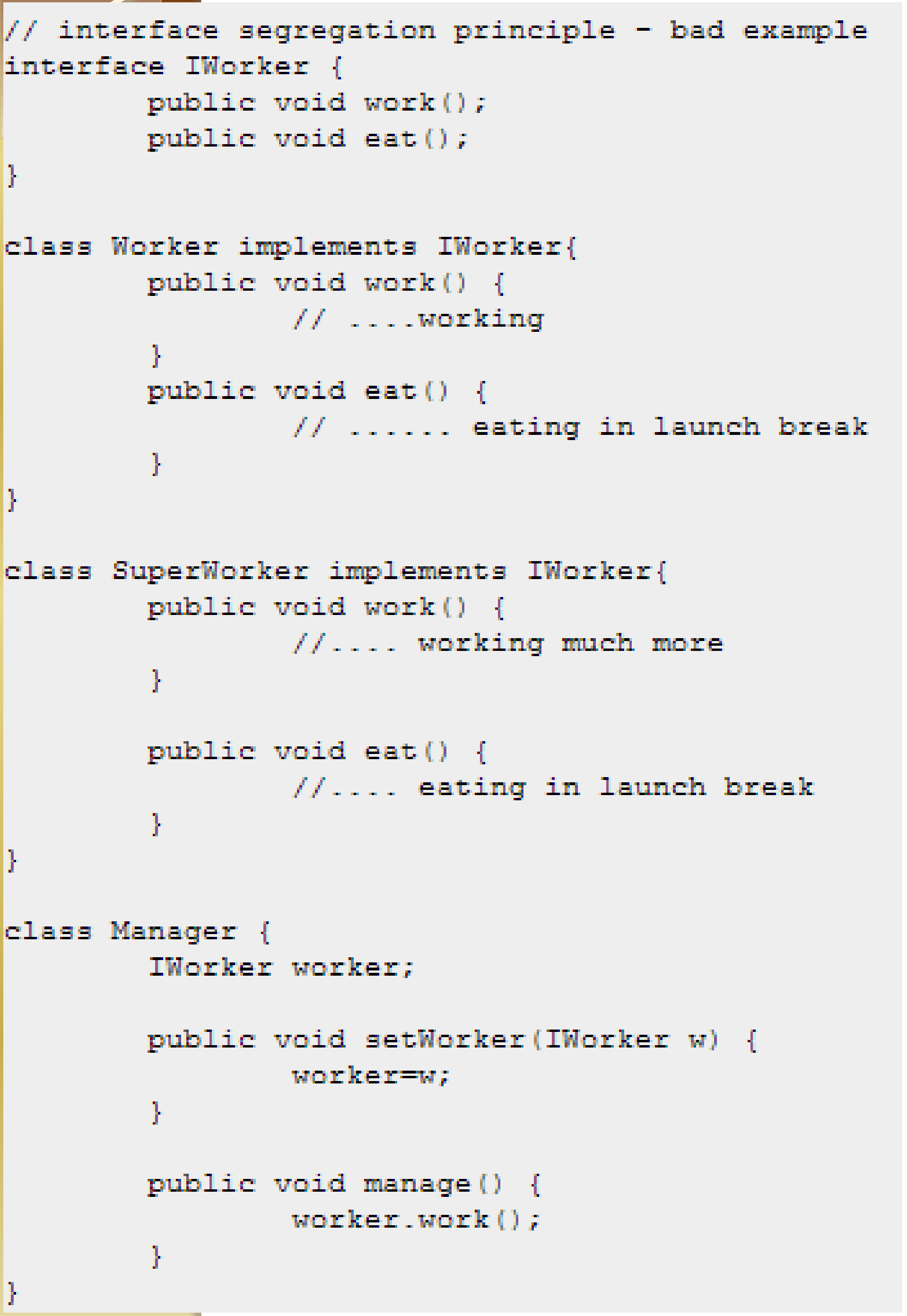
Interface Segregation Principle (ISP)

Classes should not depend on interfaces that they not use.

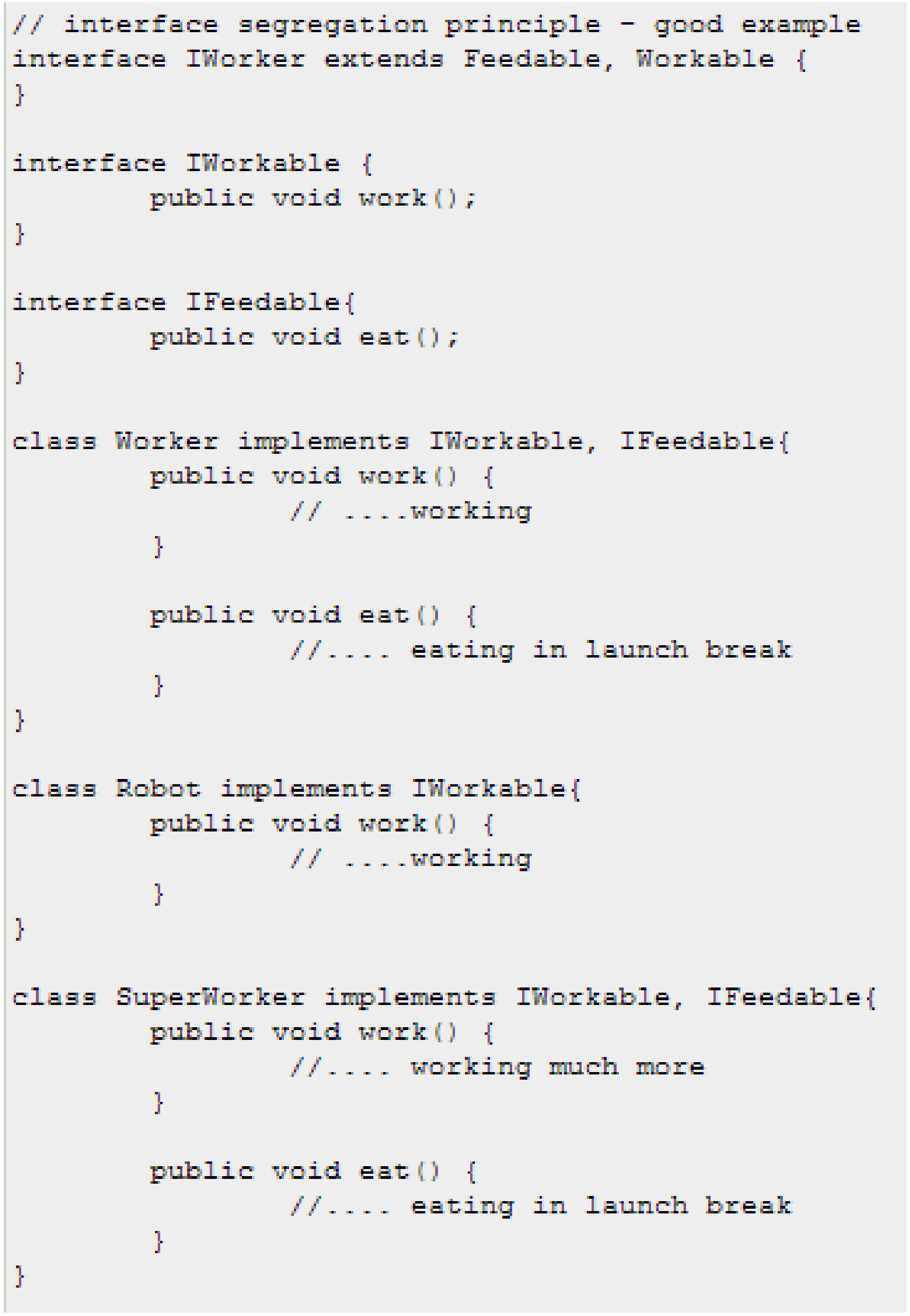
- The meaning of this phrase is to avoid tying a client class to a big interface if only a subset of this interface is really needed.
- Many times you see an interface which has lots of methods.
- This is a bad design choice since probably a class implementing it will infringe Single Responsibility Principle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

You want me to plug this in, where?

INTERFACE SEGREGATION PRINCIPLE
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
        }
}
```

Dependency Inversion Principle (DIP)

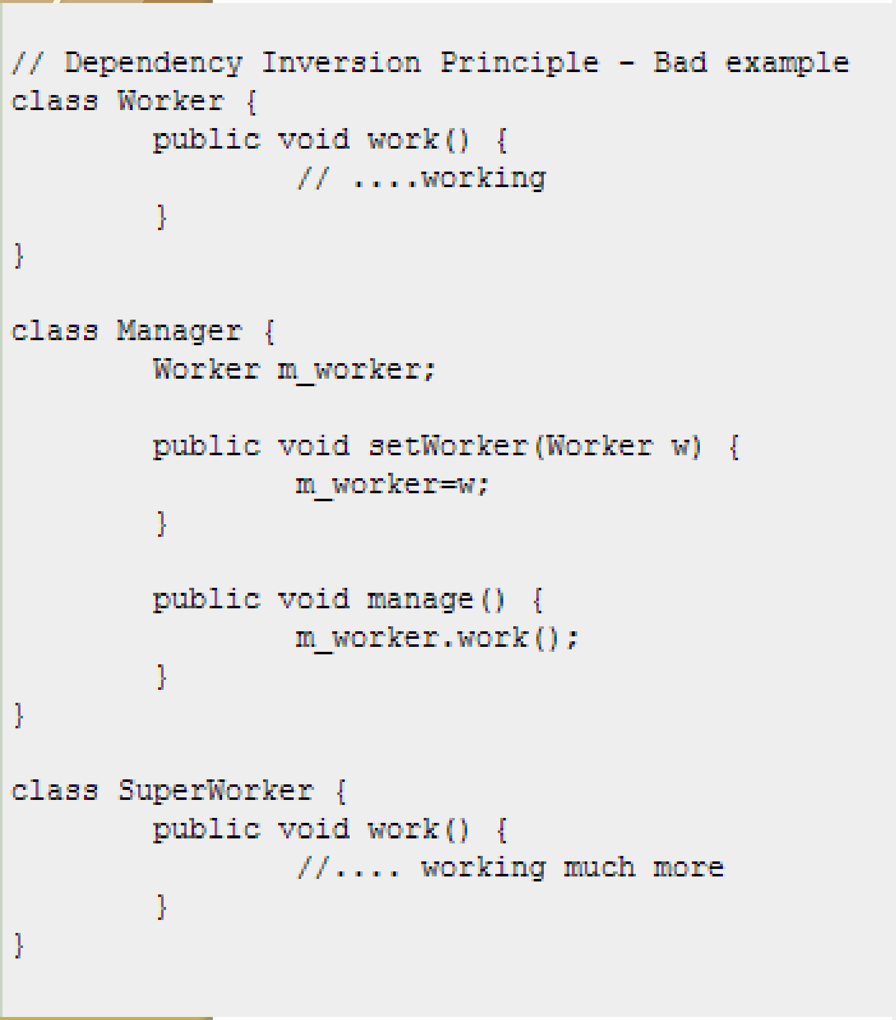
- Depend upon abstractions. Do not depend upon concretions.

- 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.

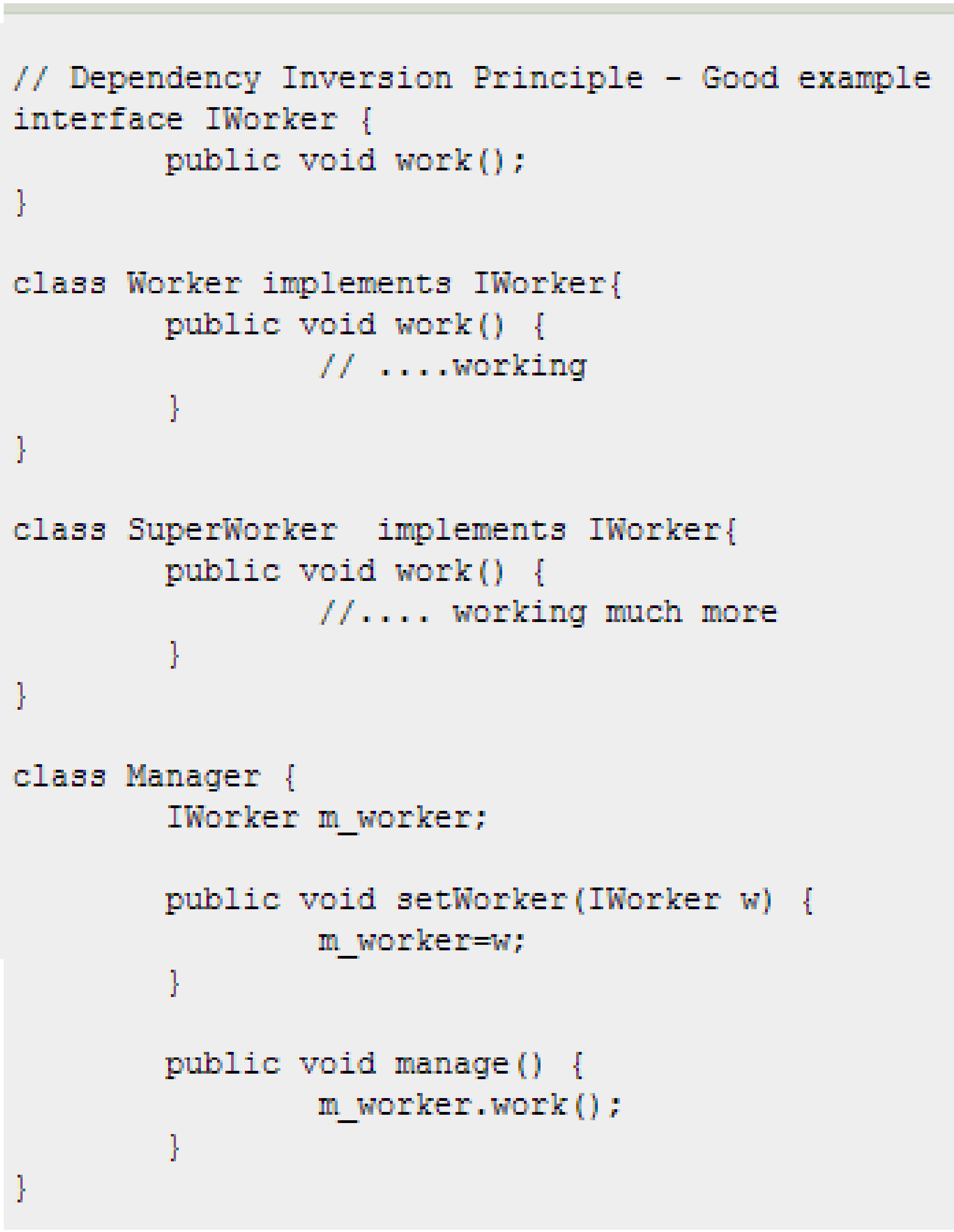
High Level Classes --> Abstraction Layer --> Low Level Classes

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

DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?



```java
// 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
        }
}
```



```java
// 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();
        }
}
```

model-view-controller (MVC)

- In object-oriented programming development, model-view-controller (MVC) is the name of a methodology or design pattern for successfully and efficiently relating the user interface to underlying data models.

model-view-controller (MVC)

- A *Model* , which represents the underlying, logical structure of data in a software application and the high-level class associated with it. This object model does not contain any information about the user interface.
- A *View* , which is a collection of classes representing the elements in the user interface (all of the things the user can see and respond to on the screen, such as buttons, display boxes, and so forth)
- A *Controller* , which represents the classes connecting the model and the view, and is used to communicate between classes in the model and view.