CitiusTech

Design Principles

October 2015

Contents

Review of Concepts (1/2)

- Cohesion and Coupling
  - **Cohesion** refers to the degree to which the elements of a module belong together.
  - Cohesion is a measure of how strongly-related or focused the responsibilities of a single module are.
  - If the methods that serve the given class tend to be similar in many aspects, then the class is said to have high cohesion.
  - In a highly-cohesive system, code readability and the likelihood of reuse is increased, while complexity is kept manageable.
  - **Coupling** is usually contrasted with cohesion. Low coupling often correlates with high cohesion, and vice versa. Coupling can be "low" (also "loose" and "weak") or "high" (also "tight" and "strong").

Low Cohesion - Bad

```
class MyFuncs {
            public void InitPrinter() {  … };
            public double CalcInterest() { … };
            public Date GetDate() { … };
}
```

High Cohesion - Good

```
class AirPlane {
            public double speed, altitude;
            public void TakeOff() { … }
            public void Fly() { … }
            public void Land() { … }
}
```

Tightly coupled code - Bad

```
public void DoSomething()  {
    // Go get some configuration
    // Get connection object
    // Get command statement
    // Apply business logic
     }
```

Loose coupled code – Good

```
public void DoSomething()…  (in MyApp project)
public void GetConnection()… (in DataAccess project)
public void ExecuteCommand()… (in DataAccess project)
public void ApplyBL()… (in domain tier / BL project)
```

Review of Concepts (2/2)

- Inheritance, Composition and Aggregation
  - Inheritance is a way to reuse code of existing objects, or to establish a subtype from an existing object, or both. Inheritance gives us an 'is-a' relationship
    - Cube ‘is a’ Shape, Square ’is a’ Shape
  - Composition is a way to combine simple objects or data types into more complex ones. Composition gives us a 'part-of' relationship
    - Steering wheel, Seat, Gearbox and Engine are ‘part of’ an automobile
  - Aggregation differs from ordinary composition in that it does not imply ownership. Aggregation gives us a 'has-a' relationship
    - Person ‘has a’ address or person ‘has a’ Bank account number

Composition

```
public class Engine
{
 . . .
}

public class Car
{
    Engine e = new Engine();
    .......
}
```

Aggregation

```
public class Address { … }

public class Person {
     private Address address;
     public Person(Address address) {
         this.address = address;
     }
}
Person would then be used as follows:
Address address = new Address();
Person person = new Person(address);
```

Contents

- Review of Concepts
- **What Are Design Principles?**
  - **Single Responsibility Principle**
  - **Open-Close Principle**
  - **Liskov's Substitution Principle**
  - **Interface Segregation Principle**
  - **Dependency Inversion Principle**

What Are Design Principles?

- OO design principles are set of guidelines that helps us to avoid having a bad design
- Three important characteristics of a bad design that should be avoided:
  - Rigidity - It is hard to change because every change affects too many other parts of the system
  - Fragility - When you make a change, unexpected parts of the system break
  - Immobility - It is hard to reuse in another application because it cannot be disentangled from the current application
- SOLID design principles:
  - Single Responsibility Principle
  - Open-close
  - Liskov's Substitution Principle
  - Interface Segregation Principle
  - Dependency Inversion Principle

Single Responsibility Principle (1/3)

Single Responsibility Principle (2/3)

- **Reason for use**
  - If we have 2 reasons to change for a class, we have to split the functionality in two classes. Each class will handle only one responsibility and on future if we need to make one change we are going to make it in the class which handle it
- **Intent**
  - A class should have only one reason to change

Play Video

Single Responsibility Principle (3/3)

BAD PRACTICE

```
// single responsibility principle

interface IEmail {
            public void setSender(String sender);
            public void setReceiver(String
receiver);
            public void setContent(String content);
}

class Email implements IEmail {
            public void setSender(String sender) {
// set sender; }
            public void setReceiver(String
receiver) {// set receiver; }
            public void setContent(String content)
{// set content; }
}
```

GOOD PRACTICE

```
// single responsibility principle
interface IEmail {
           public void setSender(String sender);
            public void setReceiver(String receiver);
            public void setContent(IContent content);
}
interface Content {
             public String getAsString(); // used for
serialization
}
class Email implements IEmail {
             public void setSender(String sender) {//
set sender; }
             public void setReceiver(String receiver)
{// set receiver; }
             public void setContent(IContent content)
{// set content; }
}
```

If we keep only one class, each change for a responsibility might affect the other one:

- Adding a new protocol will create the need to add code for parsing and serializing the content for each type of field
- Adding a new content type (like html) make us to add code for each protocol implemented

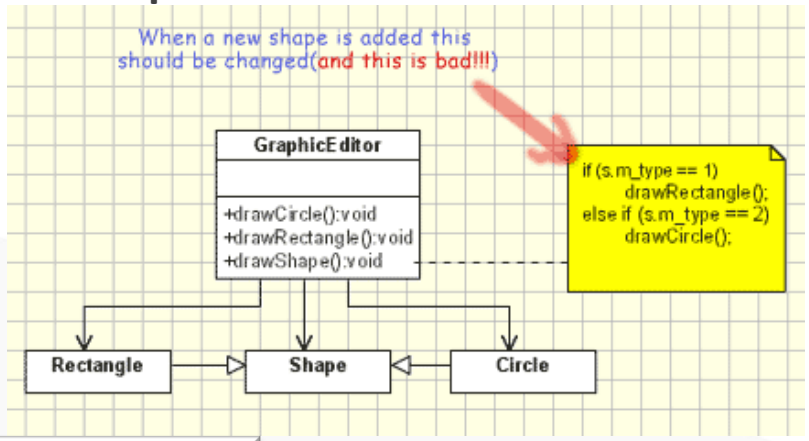
We can create a new interface and class called IContent and Content to split the responsibilities. Having only one responsibility for each class give us a more flexible design:

- adding a new protocol causes changes only in the Email class
- adding a new type of content supported causes changes only in Content class

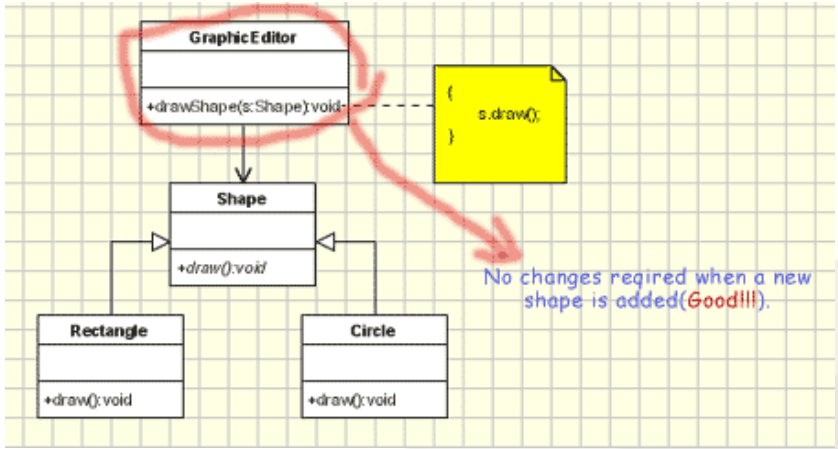
Open-Close Principle (1/3)

Open-Close Principle (2/3)

- **Reason for use**
  - Usually, many changes are involved when a new functionality is added to an existing unit tested application. Design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code.
- **Intent**
- Software entities like classes, modules and functions should be **open for extension** but **closed for modifications**.
- **Example**



Play Video



Open-Close Principle (3/3)

BAD PRACTICE

```
// Open-Close Principle
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;
               }
}
```

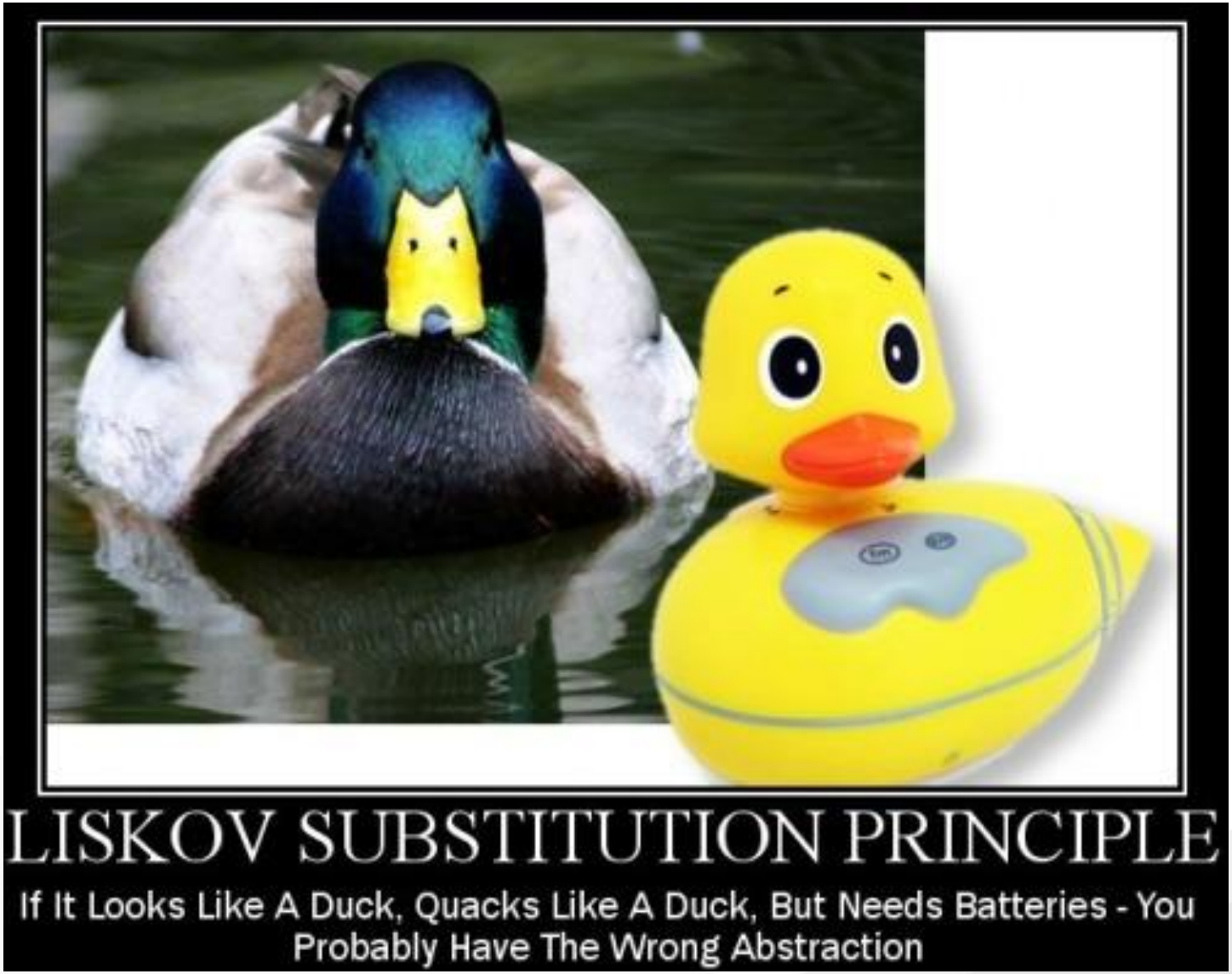
GOOD PRACTICE

```
// Open-Close Principle
class GraphicEditor {
          public void drawShape(Shape s) {
                        s.draw();
           }
}
class Shape  {
            abstract void draw();
}
class Rectangle extends Shape {
                  public void draw() {
                            // draw the rectangle
                   }
}
```

Using the Open-Close Principle the problems from the previous design are avoided, because GraphicEditor is not changed when a new shape class is added:

- no unit testing required
- no need to understand the source code from GraphicEditor
- since the drawing code is moved to the concrete shape classes, it's a reduced risk to affect old functionality when new functionality is added

Liskov's Substitution Principle (LSP) (1/3)



Liskov's Substitution Principle (LSP) (2/3)

- **Reason for use**
  - We must make sure that the new derived classes just extend without replacing the functionality of old classes. Otherwise the new classes can produce undesired effects when they are used in existing program modules.
  - If a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module.
- **Intent**
  - Derived types must be completely substitutable for their base types.

If

for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms T, the behavior of P is unchanged when o1 is substituted for o2

then

S is a subtype of T."

Play Video

Liskov's Substitution Principle (LSP) (3/3)

BAD PRACTICE

```
// Liskov's Substitution Principle
class Rectangle  {
  protected int m_width;
  protected int m_height;
  public void setWidth(int width) { m_width = width; }
  public void setHeight(int height) {  m_height =
height; }
  public int getWidth() {  return m_width; }
  public int getHeight()  { return m_height; }
  public int getArea(){  return m_width * m_height; }
}

class Square extends Rectangle
{
   public void setWidth(int width){
      m_width = width;
      m_height = width; }
  public void setHeight(int height){
      m_width = height;
      m_height = height; }
}

// <Some Factory> returns “new Square()” object
Rectangle r = <Some Factory>.getNewRectangle();
r.setWidth(5);
r.setHeight(10);
r.getArea(); // returns 100
```

FIXED ISSUE

```
// Liskov's Substitution Principle
public class Square
 {
     private Rectangle r;
     public void SetWidth (double x) {
           r.SetWidth (x);
           r.SetHeight (x);
     }
     public void SetHeight (double x) {
           r.SetWidth (x);
           r.SetHeight (x);
     }
     public void GetWidth () { return r.GetWidth; }
     public void GetHeight() { return r. GetHeight; }
     public void GetArea () { return r. GetArea; }
}

Or

You can create IShape interface and Rectangle and
Square shape can implement IShape interface to fix
this issue.
```

Interface Segregation Principle (ISP) (1/3)

Interface Segregation Principle (ISP) (2/3)

- **Reason for use**
  - When we design an application we should take care how we are going to make abstract a module which contains several sub-modules
  - Considering the module implemented by a class, we can have an abstraction of the system done in an interface. But if we want to extend our application adding another module that contains only some of the sub-modules of the original system, we are forced to implement the full interface and to write some dummy methods
  - Such an interface is named fat interface or polluted interface
  - Having an interface pollution is not a good solution and might induce inappropriate behavior in the system
- **Intent**
  - Clients should not be forced to depend upon interfaces that they don't use

Play Video

Interface Segregation Principle (ISP) (3/3)

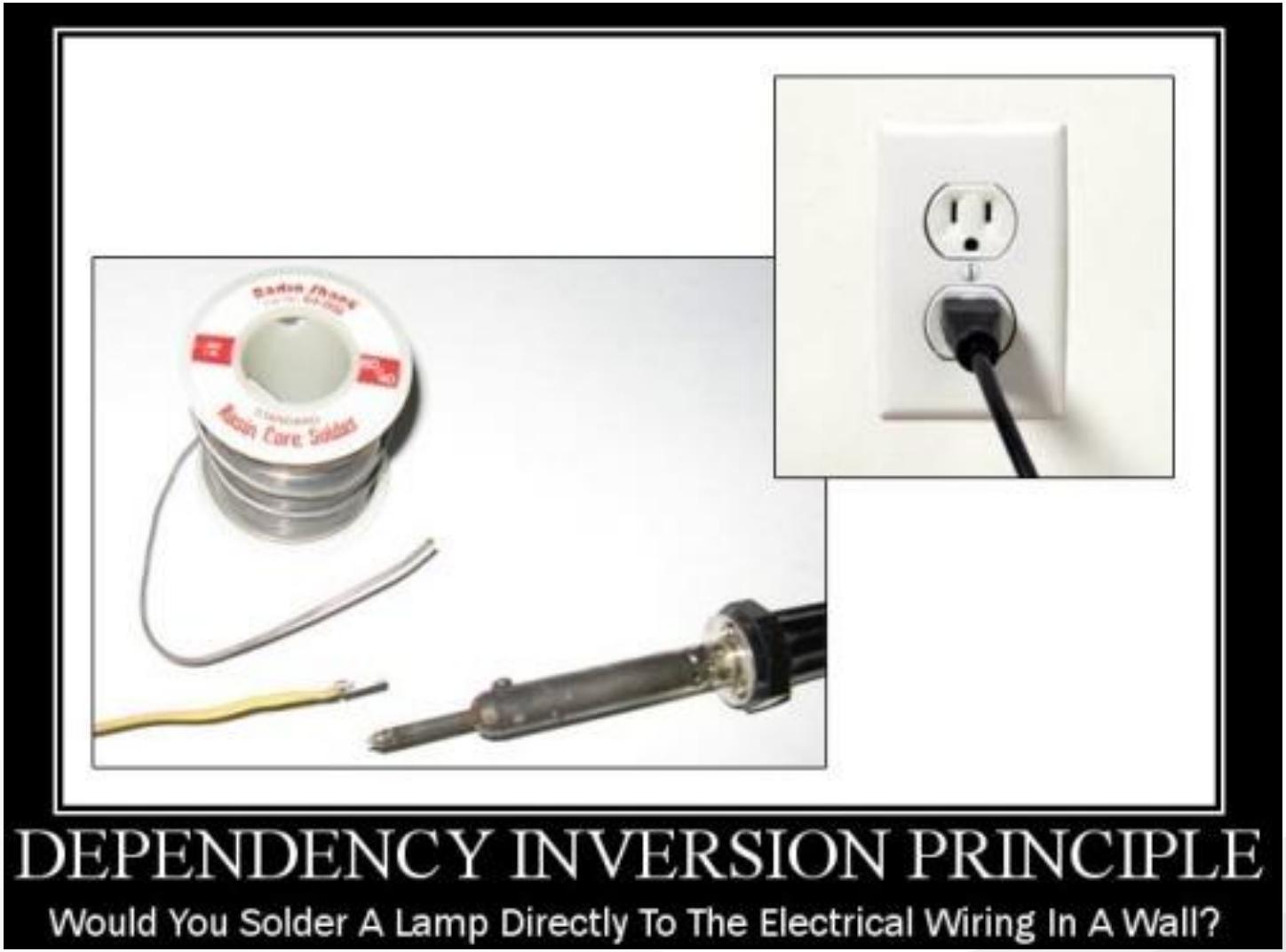
BAD PRACTICE

```
// interface segregation principle -
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();   }
}
```

GOOD PRACTICE

```
// interface segregation principle -
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 }
}

class Manager {
    Workable worker;
    public void setWorker(Workable w) { worker=w;   }
   public void manage() {   worker.work();   }
}
```

Dependency Inversion Principle (1/3)



Dependency Inversion Principle (2/3)

- **Reason for use**
  - Low level classes which implement basic and primary operations and high level classes which encapsulate complex logic and rely on the low level classes
  - In a bad design the high level class uses directly the low level classes
  - In order to avoid such problems we can introduce an abstraction layer between the high level classes and low level classes
  - According to this principle the way of designing a class structure is to start from high level modules to the low level modules:
    High Level Classes --> Abstraction Layer --> Low Level Classes
- **Intent**
  - 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

Play Video

Dependency Inversion Principle (3/3)

BAD PRACTICE

```
// Dependency Inversion Principle
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  }
}
```

GOOD PRACTICE

```
// Dependency Inversion Principle
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();
   }
}
```

THANK YOU