Object-Oriented Analysis & Design (OAD)

Object-Oriented Design Principles

https://youtu.be/d0GjujVOM3g

Alexander Felfernig

Institute for Software Technology

Inffeldgasse 16b/2

Motivation [MAR2000]

Graz University of Technology

- Every programmer is a designer.
- Design begins as an easy to understand concept in the minds of designers.
- The resulting applications often make it to the first release.
- But then, the software starts to rot!
- Maintainability/Extensibility of the code becomes increasingly an issue.

Symptoms of Rotting Design

Graz University of Technology

- **Rigidity**: even simple changes are uneasy and cause a cascade of further changes. A planned 2-day change makes it into a 2-weeks marathon.
- **Fragility**: tendency of software to break in many places due to change operations.
- **Immobility**: rewriting appears easier as reuse.
- **Viscosity**: design-preserving methods are harder to apply than just applying hacks.
- **Immediate cause**: changing requirements!

Working Example

```
class GraphicalObject {
        public double Surface() { …
        }
}

class Circle extends GraphicalObject{
        double radius;

        public Circle(int r) {
        radius = r;
        }
```

```
    public double Surface() {
        return(this.radius*this.radius*Math.PI);
    }
}

class Square extends GraphicalObject{
    double length;

    public Square(double l) {length = l;}

    public double Surface() {
        return(this.length*this.length);
    }
}
```

Working Example

```
class SurfaceCalculator{
    GraphicalObject objectlist[];

        public SurfaceCalculator(GraphicalObject l[])
        {objectlist = l;}

        public double TotalSurface() {
        double surface = 0.0;
        for (GraphicalObject object : objectlist) {
            surface = surface + object.Surface();
        }
        return(surface);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Circle object1 = new Circle(12);
        Square object2 = new Square(24);
        GraphicalObject objectlist[] =
                            new GraphicalObject[2];
        objectlist[0] = object1;
        objectlist[1] = object2;
        SurfaceCalculator surfacecalculator =
              new SurfaceCalculator(objectlist);
        double surfaceval =
              surfacecalculator.TotalSurface();
    }
}
```

5 OO Design Principles

Graz University of Technology

- **S** - Single-responsibility principle (SR)
- **O** - Open-closed principle (OC)
- **L** - Liskov substitution principle (LS)
- **I** - Interface segregation principle (IS)
- **D** - Dependency Inversion Principle (DI)

An evaluation of these principles is reported, for example, in [SH2015].

Single Responsibility (SR)

Graz University of Technology

- A class should have one and only one responsibility, i.e, a class should have only one task/job.
- In our example: formatting and output of calculated surface is not a task of the class **SurfaceCalculator**!
- Calculation and output are two different tasks which contradicts the single responsibility principle (see next slide …).

Example: SR

Graz University of Technology

```
class SurfaceCalculator{
   GraphicalObject objectlist[];

   public SurfaceCalculator(GraphicalObject l[])
   {objectlist = l;}

   public double TotalSurface() {        task 1
      double surface = 0.0;
      for (GraphicalObject object : objectlist) {
         surface = surface + object.Surface();}
      return(surface);}

   public void PrintTotalSurface() {     task 2
      double surface;
      surface = TotalSurface();
      java.lang.System.out.print(surface);}
}
```

→ <u>two tasks</u>, i.e., not single responsibility!

```
class SurfaceOutput {
   double surface;
   public SurfaceOutput(double s) {
      surface = s;}

   public void printhtml() { … }

   public void printtext() { … }
}
class SurfaceCalculator{
   GraphicalObject objectlist[];

   public SurfaceCalculator(… l[])       task 1
   {objectlist = l;}

   public double TotalSurface() {
      …}
```

→ <u>one task</u>, i.e., single responsibility!

Single Responsibility and Cohesion

- Measuring the extent to which methods and data of a class belong to a common concept.
- Thus, class elements cooperate to achieve one common function (task).
- Low cohesion classes violate the criteria of single responsibility.
- Cohesion class c: $coh(c) = \frac{\sum_{i=1}^{n} \#methods(attribute\ i)}{n}$

Open Closed Principle (OC)

- Objects or entities should be open for extension, but closed for modification.
- As a consequence of this principle, <u>a class should be extendable without modification</u>!
- In our example, we should be able to add more shapes without the need of modifying the **TotalSurface** method (see next slide).

Example: OC

Graz University of Technology

```
class SurfaceCalculator{
    GraphicalObject objectlist[];
    public SurfaceCalculator(GraphicalObject l[])
    {objectlist = l;}
    public double TotalSurface() {
        double surface = 0.0;        modification need!
        for (GraphicalObject object : objectlist) {
            if(object instanceof Circle){
             surface=surface+
             (((Circle)object).getradius() *
             ((Circle)object).getradius() * Math.PI);}
            else{
             surface=surface + …}
        }
        return(surface);
    }
}
```

→ **SurfaceCalculator** has to be modified for new graphical classes!

```
class SurfaceCalculator{
   GraphicalObject objectlist[];
   public SurfaceCalculator(GraphicalObject l[])
   {objectlist = l;}
   public double TotalSurface() {
        double surface = 0.0; no modification need!
        for (GraphicalObject object : objectlist) {
            surface = surface + object.Surface();}
        return(surface);}
   }
class Circle extends GraphicalObject{
    public double radius; …
    public double Surface() {
        return(this.radius*this.radius*Math.PI);}
}
```

→ No modification of **SurfaceCalculator**, new classes responsible for **Surface**!

Liskov Substitution Principle (LS)

Graz University of Technology

- Each instance of a derived class B should behave as expected even it assumed to be an instance of the base class A.
- In our example, Circle (B) could be interpreted as subclass of Ellipse (A).
- However, depending on the implementation, the substitution principle is not taken into account (see next slide).

Example: LS

Graz University of Technology

```
class Ellipse extends GraphicalObject{
    int xscale; int yscale;

    public Ellipse(int r, int s) {xscale=r; yscale=s;}
    public int getxscale() {return(xscale);}
    public int getyscale() {return(yscale);}
    public double Surface() {
    return(this.getxscale()*
          this.getyscale()*Math.PI);}
}
class Circle extends Ellipse{
    public Circle(int r, int s) {super(r,s)}
    public double Surface() {
      return(this.getxscale()*
      this.getxscale()*Math.PI);}
}
```

→ Behavior of Surface differs!

```
class Circle extends GraphicalObject{
    public double radius;

    public Circle(int r) {
        radius = r;
    }
    public double getradius() {
        return(radius);
    }
    public double Surface() {
        return(this.radius*this.radius*Math.PI);
    }
}
```

→ Circle not a subclass of Ellipse!

Interface Segregation Principle (IS)

Graz University of Technology

- Clients (of interfaces) should not be forced to depend on methods they do not use.
- In our example, we could assume that different clients use either a statistics service (counting) or a print service.
- The two aspects should be covered by different interfaces (see next slide).

Example: IS

Graz University of Technology

interface IGraphicalObjects

{ int **Count**();

void **PrintHTML**();

void **PrintText**(); ...

}

class CIGraphicalObjects implements IGraphicalObjects

{public int **Count**(){…};

public void **PrintHTML**(){…};

public void **PrintText**(){…}; …

}

→ Assumption: all clients will use all of the provided methods!

interface ICount

{int **Count**(); … }

interface IPrint

{ void **PrintHTML**();

void **PrintText**(); ... }

class CICount implements ICount

{public int **Count**(){…}; ... }

class CIPrint implements IPrint

{ public void **PrintHTML**(){…};

public void **PrintText**(){….}; …

}

→ Client-specific interfaces, e.g., one client only want to have printing!

Dependency Inversion Principle (DI)

- Entities should depend on abstractions but not specializations.
- High-level entities should not rely on low-level entities.
- The idea of dependency inversion is decoupling; see the example on the next slide.

Example: DI

Graz University of Technology

```
class LightBulb {
    public void turnOn() {
        System.out.println("on...");}
    public void turnOff() {
        System.out.println("off...");}}

class ElectricPowerSwitch {
    public LightBulb lightBulb; public boolean on;
    public ElectricPowerSwitch(LightBulb lightBulb) {
        this.lightBulb = lightBulb; this.on = false;}
    public boolean isOn() {return this.on;}
    public void press(){boolean checkOn = isOn();
        if (checkOn) {lightBulb.turnOff();
            this.on = false;
        } else {lightBulb.turnOn(); this.on = true;}
    }
}
```

→ ElectricPowerSwitch depends on LightBulb!

```
interface Switch {boolean isOn();
                  void press();}
interface Switchable {void turnOn();
                      void turnOff();}

class ElectricPowerSwitch implements Switch {
    public Switchable c; public boolean on;
    public ElectricPowerSwitch(Switchable c) {
        this.c = c; this.on = false;}
    public boolean isOn() {return this.on;}
    public void press(){boolean checkOn = isOn();
        if (checkOn) {c.turnOff(); this.on = false;}
        else {c.turnOn(); this.on = true;}}}

class LightBulb implements Switchable {
    public void turnOn() {System.out.println("on...");}
    public void turnOff() {System.out.println("off...");}
}
```

→ More flexibility, e.g., „switching“ can be used in other contexts!

Example from: springframework.guru/principles-of-object-oriented-design

Further Principles: Don‘t Repeat Yourself (DRY)

```
class Animal {
    public void eatfood() {…}
}

class Cat extends Animal {
    public void meow() {…}
}

class Dog extends Animal {
    public void bark() {…}
}
```

- Same functionalities across different classes: couple these into a common parent class or an interface.
- In our simple example: both dogs and cats eat food, i.e., this aspect can be assigned to **Animal**.

Further Principles: Keep it Stupid & Simple (KISS)

- Keep the code simple and readable for humans.
- If a class handles more than one aspect, think about class splitting.
- Unreadable and long methods are extremely hard to maintain.

Further Principles

Graz University of Technology

- Reuse = criteria for grouping classes into packages; packages are also the unit of reuse (**Reuse Equivalence Principle** - **REP**)
- Classes that change together, belong together (**Common Closure Principle** – **CCP**)
- Classes that are not reused together, should not be grouped together (**Common Reuse Principle** - **CRP**)
- REP and CRP makes it easier for re-users, CCP makes it easier for maintainers!

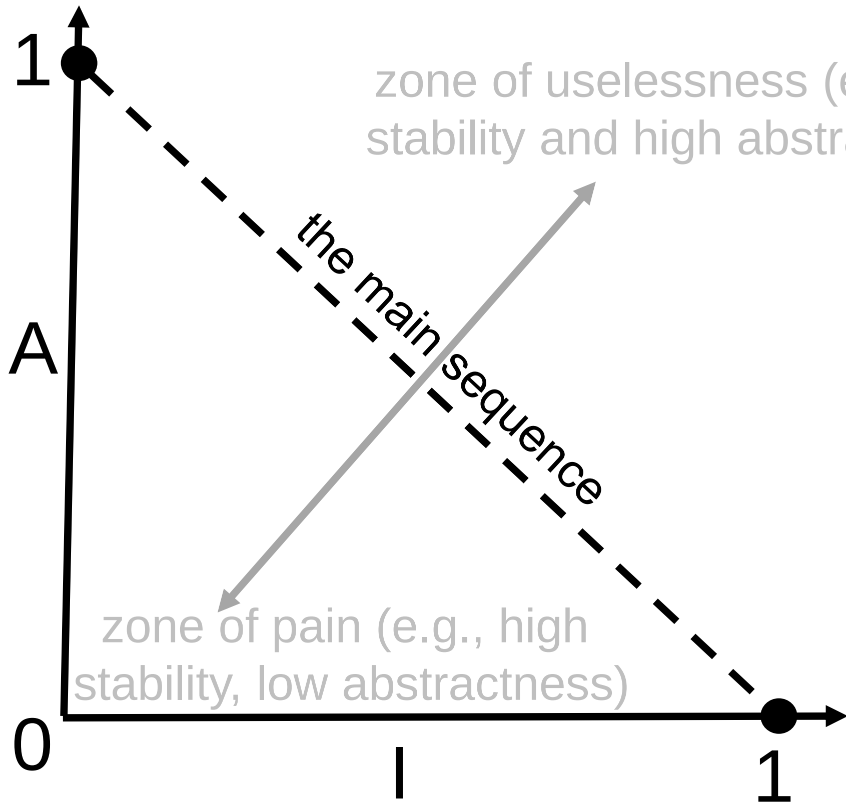
Further Principles

- Dependencies between packages must not form circles (**Acyclic Dependencies Principle** – **ADP**). This could trigger additional test efforts. Solution: inclusion of a new package.
- Depend in the direction of stability (**Stable Dependencies Principle** – **SDP**). High number of incoming dependencies ~ high „stability“
- Stable packages should be abstract packages (**Stable Abstractions Principle** – **SAP**); these are not easy to change but easy to extend!

Package Stability Metrics

- **Ca**: #classes outside the package p that depend upon classes inside the package ("**incoming**")
- **Ce**: #classes outside the package p that classes inside the package depend upon ("**outgoing**")
- (Assumed) **Instability** [0..1]: $\boldsymbol{I}(\boldsymbol{p}) = \frac{Ce}{Ca+Ce}$
- **Nc**: #classes in the package
- **Na**: #abstract classes in the package
- (Assumed) **Abstractness** [0..1]: $\mathbf{A}(\boldsymbol{p}) = \frac{Na}{Nc}$

„I vs A Graph“



Concrete packages should be instable while abstract packages should be stable!

OO Metrics Suite [CK1994]

- Concrete complexity measures
- Relationships to object-oriented design principles
- Basic rules for object-oriented programming styles
- Six Metrics (see the next slide …)

OO Metrics Suite

Graz University of Technology

- **(Weighted) Methods Per Class (WMC)**: time efforts per class, impacts on children
- **Depth of Inheritance Tree (DIT)**: the deeper a class, the more methods inherited, +complexity
- **Number of Children (NOC)**: the greater NOC, the greater the likelihood of improper abstraction
- **Coupling between Object Classes (CBO)**: detrimental to modular design, prevents reuse!
- **Response for a Class (RFC)**: response set is a set of methods that could be activated
- **Lack of Cohesion in Methods (LCOM)**: low cohesion indicates that a class should be split

References

- [MAR2000] R. Martin. Design Principles and Design Patterns, pp. 1-34, 2000. Link to paper: https://bit.ly/2yLYufQ
- [SH2015] H. Singh and S. Hassan. Effect of SOLID Design Principles on Quality of Software: An Empirical Assessment, Int. Journal of Scientific & Engineering Research, 6(4):1321-1324, 2015.
- [CK1994] S. Chidamber and C. Kemerer. A metrics suite for object oriented design, IEEE Transactions on Software Engineering, 28:476-492, 1994.

Graz University of Technology

Thanks!

ase.ist.tugraz.at
www.felfernig.eu