SOLID Design Principle

MFEC Dev Day #2

MAHASAK PIJITTUM

Original slide: SOLID OO DESIGN PRINCIPLES by Andreas Enbohm, Capgemini


programmer
source
traffic
management
erp
programming
content
best
page
cost
approach
overview
problem
maintenance
few
control
developing
architecture
marketing
product
css
team
time
quality
spent
term
client
information
process
website
crm
development
coding
model
implementation
different
use
experience
several
plan
solution
application
usability
design
web2.0
project
planned

What is software design ?

source code

UML diagram

Software design process

Why do we need a good design ?

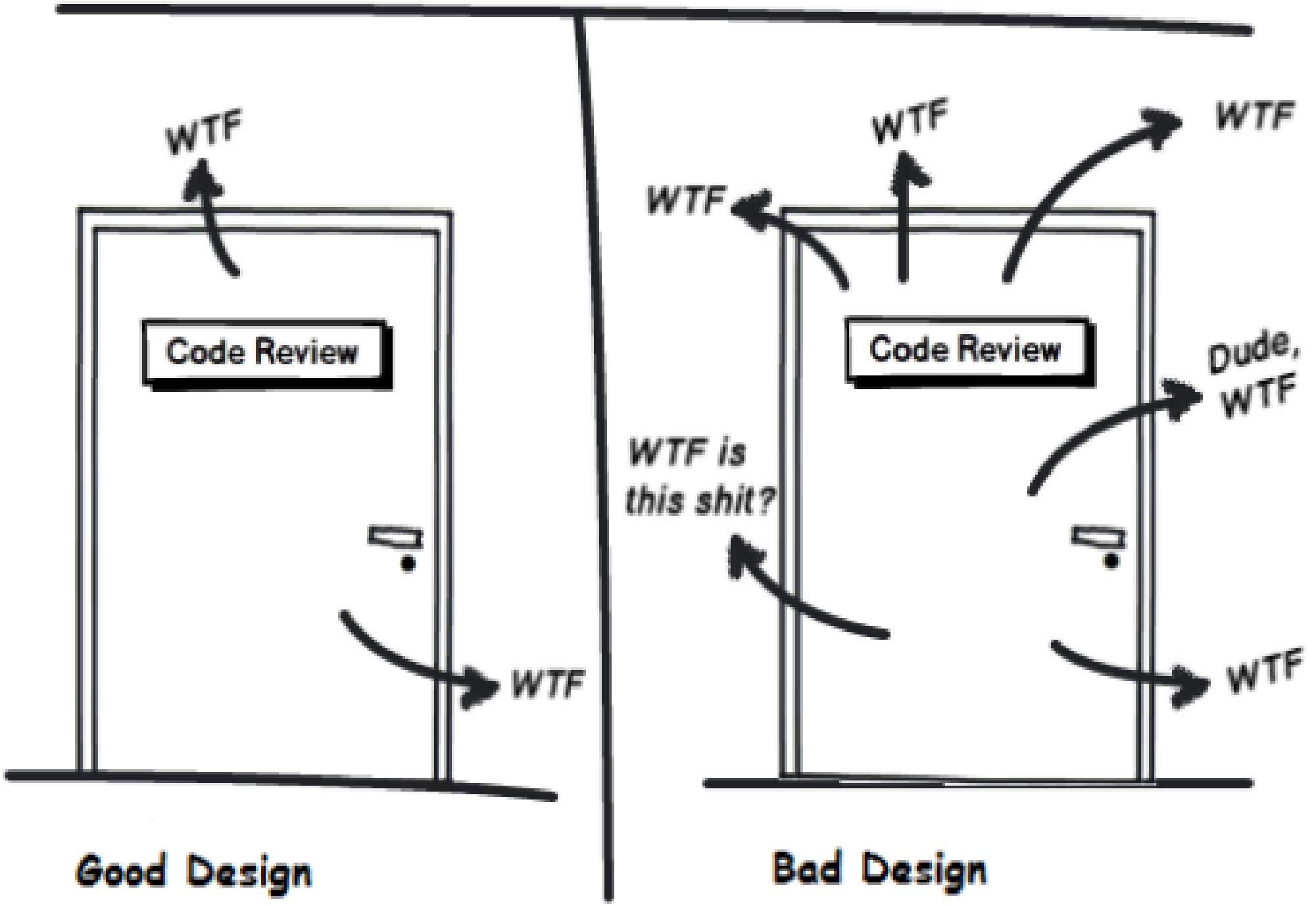
deliver fast

deal with complexity

manage change easily

How to identify a bad design ?


สัตว์
!!
สัตว์
!!


WTF
Code Review
WTF
Good Design
WTF
WTF
WTF
WTF
Code Review
Dude,
WTF
WTF is
this shit?
WTF
Bad Design

Rigidity

Fragility

Immobility

Viscosity

What are the characteristics of a good design ?

High Cohesion Low Coupling

How to achieve a good design ?

practices

principles

patterns

- What is SOLID Design Principles?
- Code Examples
- Q&A

Single Responsibility Principle

Open Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inverison Principle

Single Responsibility Principle

There should never be more than one reason for a class to change."
– Robert Martin, SRP paper linked from [The Principles of OOD]

Single Responsibility Principle

- Two resposibilities

```
interface Modem {
  public void dial(String pno);
  public void hangup();

  public void send(char c);
  public char recv();
}
```

Connection Management + Data Communication

Single Responsibility Principle

- Separate into two interfaces

```
interface DataChannel {
 public void send(char c);
  public char recv();
}
```

```
interface Connection {
  public void dial(String phn);
   public char hangup();
}
```

Open Closed Principle

"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification." – Robert Martin paraphrasing Bertrand Meyer, OCP paper linked from [The Principles of OOD]

Open Closed Principle

```
// 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;
                }
  }
```

Open Closed Principle – Improved

```
// 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
        }
}
```

Liskov Substitution Principle

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it." – Robert Martin, LSP paper linked from [The Principles of OOD]

Liskov Substitution Principle

```
// Violation of Liskov's Substitution Principle
  class Rectangle
  {
            int m_width;
            int m_height;

            public void setWidth(int width){
                        m_width = width;
            }

            public void setHeight(int h){
                        m_height = ht;
            }


            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;
            }
  }
```

Liskov Substitution Principle

```
class LspTest
  {
  private static Rectangle getNewRectangle()
  {
                // it can be an object returned by some factory ...
                return new Square();
  }

  public static void main (String args[])
  {
                Rectangle r = LspTest.getNewRectangle();
                r.setWidth(5);
                r.setHeight(10);
// user knows that r it's a rectangle. It assumes that he's able to set the width and height as
  for the base class

                System.out.println(r.getArea());
                // now he's surprised to see that the area is 100 instead of 50.
  }
  }
```

Interface Segregation Principle

"Clients should not be forced to depend upon interfaces that they do not use." – Robert Martin, ISP paper linked from [The Principles of OOD]

Interface Segregation Principle

```
//bad example (polluted interface)
interface Worker {
 void work();
 void eat();
}
```

```
ManWorker implements Worker {
        void work() {…};
        void eat() {30 min break;};
}
```

```
RobotWorker implements Worker {
        void work() {…};
        void eat() {//Not Appliciable
                for a RobotWorker};
}
```

Interface Segregation Principle

Solution
- split into two interfaces

```
interface Workable {
	public void work();
}

interface Feedable{
	public void eat();
}
```

Dependency Inversion Principle

"A. High level modules should not depend upon low level modules. Both should depend upon abstractions.
B. Abstractions should not depend upon details. Details should depend upon abstractions." – Robert Martin, DIP paper linked from [The Principles of OOD]

Dependency Inversion Principle

```
//DIP - bad example
public class EmployeeService {
        private EmployeeFinder emFinder //concrete class, not abstract. Can access a SQL DB for instance
        public Employee findEmployee(…) {
                emFinder.findEmployee(…)
        }
}
```

Dependency Inversion Principle

```
//DIP - fixed
public class EmployeeService {
        private IEmployeeFinder emFinder //depends on an abstraction, no an implementation
        public Employee findEmployee(…) {
                emFinder.findEmployee(…)
        }
}
```