Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

SOLID

- OO DESIGN PRINCIPLES

Andreas Enbohm, Capgemini

© 2004 Capgemini - All rights reserved

Agenda

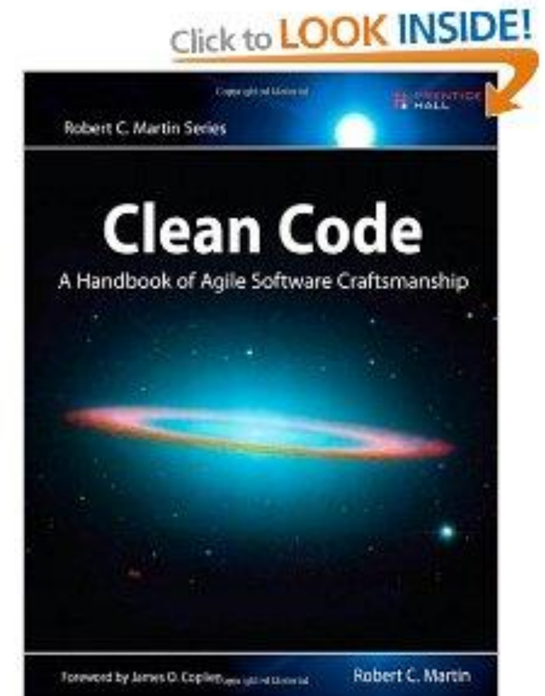
- What is SOLID Design Principles?
- Code Examples
- Q&A

© 2009 Capgemini - All rights reserved

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

SOLID

- Introduced by Robert C. Martins (”Uncle Bob”)
- Agile Manifesto
- Author of several books, e.g. ”Clean Code”



© 2009 Capgemini - All rights reserved
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

SOLID

- SOLID
  - **S**ingle Responsibility Principle
  - **O**pen Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inverison Principle
- Code becomes more *Testably* (remember TDD is not only about testing, more important its about Design)
- Apply ’smart’
  - don’t do stuff ’just because of’
  - very importad to see the context of the program/code when applying SOLID
  - Joel On Software advise – use with common sense!

© 2009 Capgemini - All rights reserved
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

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*
- My translation: A class should concentrate on doing one thing and one thing only

© 2009 Capgemini - All rights reserved
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

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

© 2009 Capgemini - All rights reserved
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

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();
}
```


© 2009 Capgemini - All rights reserved
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

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*
- My translation: Change a class' behavior using inheritance and composition

© 2009 Capgemini - All rights reserved
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

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;
         }
 }
```


© 2009 Capgemini - All rights reserved
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Open Closed Principle – a Few Problems….

- Impossible to add a new `Shape` without modifying `GraphEditor`
- Important to understand `GraphEditor` to add a new `Shape`
- Tight coupling between `GraphEditor` and `Shape`
- Difficult to test a specific `Shape` without involving `GraphEditor`
- `If-Else-/Case` should be avoided

© 2009 Capgemini - All rights reserved
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

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
      }
  }
  ```


© 2009 Capgemini - All rights reserved
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING
  ```

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*
- My translation: Subclasses should behave nicely when used in place of their base class

© 2009 Capgemini - All rights reserved

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;
         }
 }
```


© 2009 Capgemini - All rights reserved

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

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.
}
}
```


© 2009 Capgemini - All rights reserved
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

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*
- My translation: Keep interfaces small

© 2009 Capgemini - All rights reserved
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Interface Segregation Principle

- Don’t force classes so implement methods they can’t (Swing/Java)
- Don’t pollute interfaces with a lot of methods
- Avoid ’fat’ interfaces

© 2009 Capgemini - All rights reserved

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

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};
}
```


© 2009 Capgemini - All rights reserved
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Interface Segregation Principle

- Solution
  - split into two interfaces

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

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


© 2009 Capgemini - All rights reserved

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

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
- My translation: Use lots of interfaces and abstractions

© 2009 Capgemini - All rights reserved
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

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(…)
        }
}
```


© 2009 Capgemini - All rights reserved
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Dependency Inversion Principle

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

- Now its possible to change the finder to be a XmEmployeeFinder, DBEmployeeFinder, FlatFileEmployeeFinder, MockEmployeeFinder….

© 2009 Capgemini - All rights reserved
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Q&A

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

http://www.oodesign.com

© 2009 Capgemini - All rights reserved

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING