Design Principles

Agenda

- Good / Bad Design
  - What is bad design
  - Indicators of bad design
  - Design Smells
  - Why code rots
- Design Principles
  - Law Of Demeter
  - SOLI(D)

Bad / Good Design

Woran erkennt man schlechtes Design?

- der Sourcecode wenig verständlich ist, unter anderem verursacht durch ***ungünstige Namensgebung*** oder eine ***fehlende Dokumentation*** komplizierterer Stellen.
- ***unnötige Komplexität*** oder ein extrem flexibles Design mit hohem Variantenreichtum existiert, obwohl diese Extras (teilweise) nicht benötigt werden.
- Erweiterungen oder ***Modifikationen lassen sich nur schwierig durchfuhren*** und besitzen zum Teil *große Auswirkungen* auch in anderen Teilen des Sourcecodes
- das zu losende Problem anhand der Implementierung nicht abgeleitet werden kann, etwa weil es an semantischer Strukturierung fehlt oder ***kein klares Layout des Sourcecodes*** vorliegt.

Woran erkennt man schlechtes Design?

- es existieren ***nur wenige Tests und Testbarkeit kaum gegeben*** ist, z. B. weil *kaum für sich testbare Klassen existieren* und Objekte stark voneinander abhängen, sodass lediglich schwierig testbare Objekthaufen vorhanden sind.

- **diverse Fehler bereits bekannt** sind und wahrscheinlich noch viele weitere versteckt lauern. Manchmal ist das System so weit verrottet, dass nur noch ein Aufräumkraftakt oder aber ein vollständiges Redesign helfen.

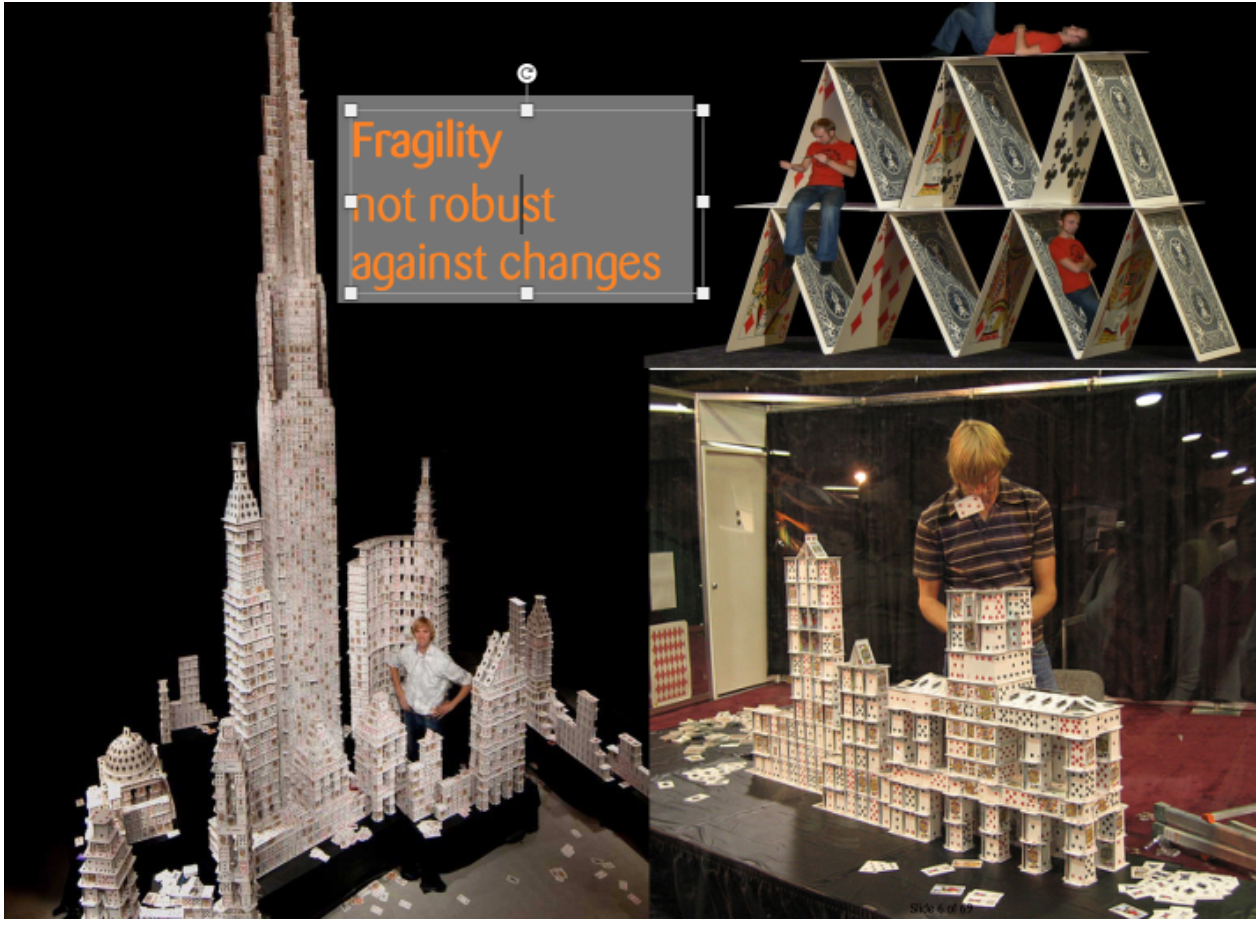
Design Smells (by Robert C. Martin)

- **Rigidity:** difficult to change, chain of changes
- **Fragility:** not robust against changes
- **Immobility:** parts not reusable
- **Viscosity:** easy to do the wrong thing
- **Needless Complexity:** contains unused elements
- **Needless Repetition:** no reuse, copy & paste
- **Opacity:** difficult to understand

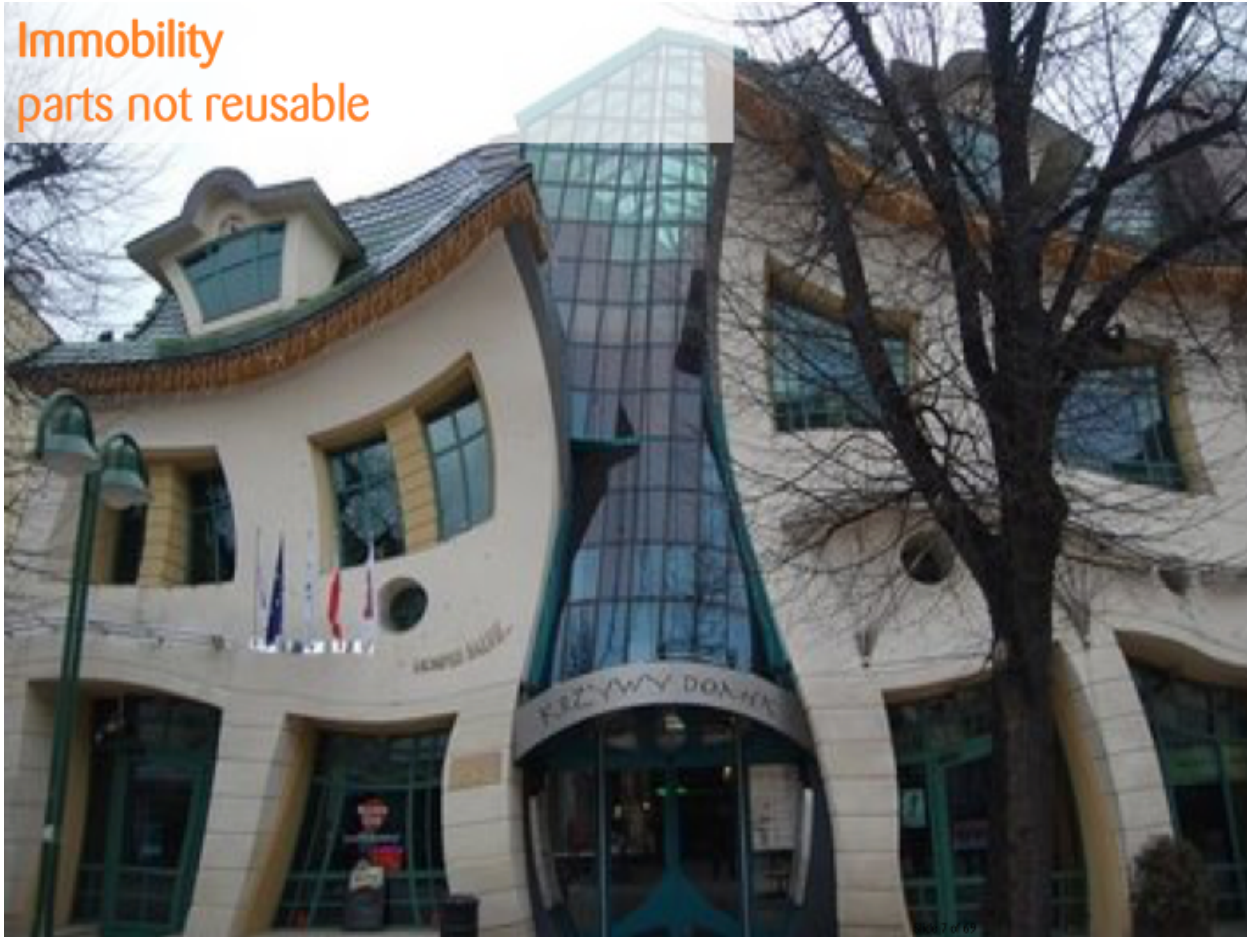
Design Smells (by Robert C. Martin)

- **Rigidity: difficult to change, chain of changes**

Design Smells (by Robert C. Martin)

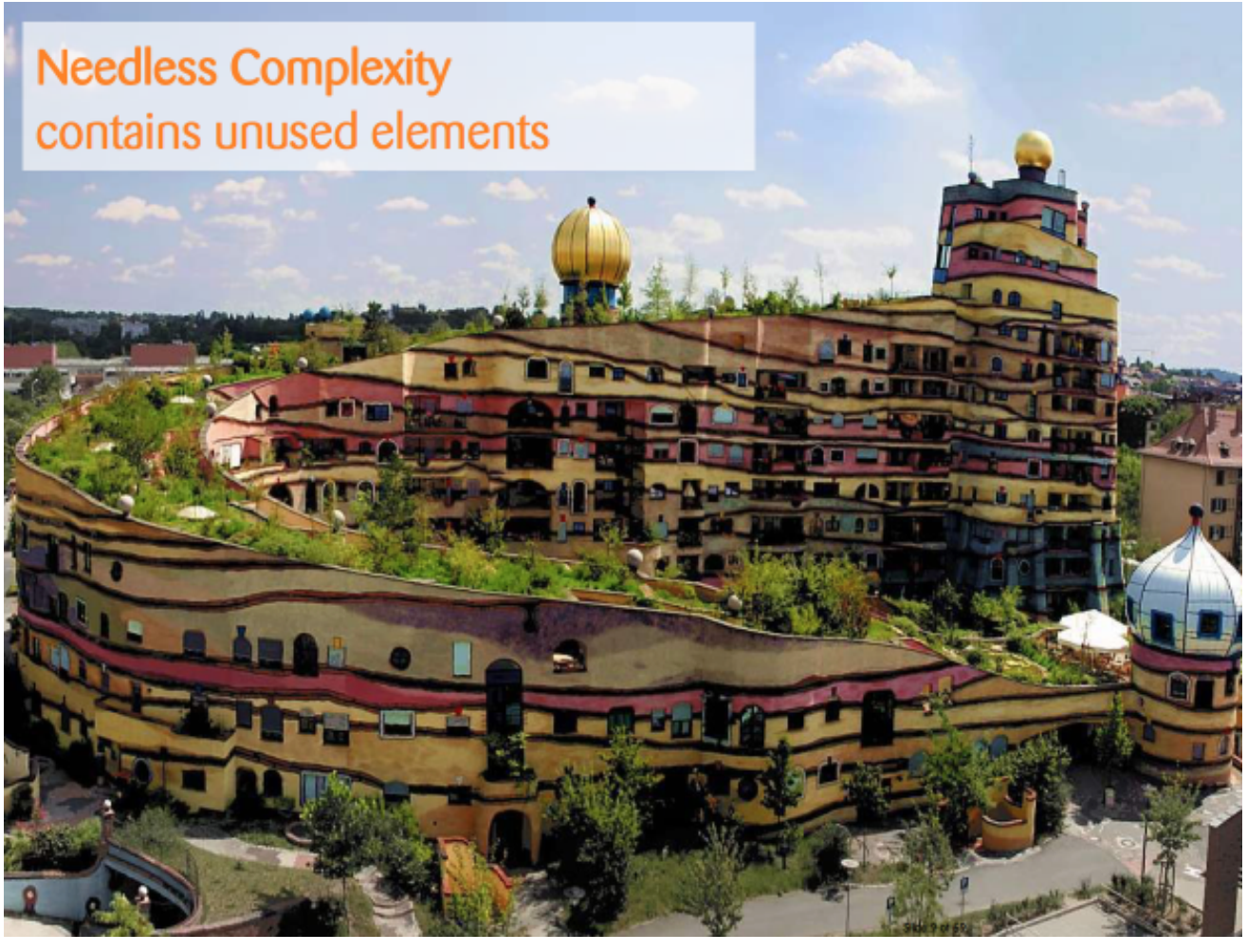


Design Smells (by Robert C. Martin)



Design Smells (by Robert C. Martin)

Design Smells (by Robert C. Martin)



Needless Repetition
no reuse, copy & paste

Opacity
difficult to understand

Why is Software rotting?

- Code rapidly rots in presence of change
- Todays (agile) Software Development processes require many code changes.
- As in your household / flat: if you don't clean up often it will get messier and messier
- Example: Implement a class „Copier" for copying keyboard input to printer

Why is Software rotting?

- First, simple (a bit stupid) prototype implementation

```
public class Copier {
    public void copy() {
        int ch = Keyboard.read();
        while (ch != -1) {
            Printer.write(ch);
            ch = Keyboard.read();
        }
    }
}
```

Why is Software rotting?

- First, simple (a bit stupid) prototype implementation

```
public class Copier {
    public void copy() {
        int ch = Keyboard.read();
        while (ch != -1) {
            Printer.write(ch);
            ch = Keyboard.read();
        }
    }
}
```

- NEW REQUIREMENT: Extend to be able to write alternatively to Console

Why is Software rotting?

- Fast modification

```
public class Copier
{
    public boolean writeConsole = false;

    public void copy()
    {
        int ch = Keyboard.read();
        while (ch != -1)
        {
            if (writeConsole)
                Console.write(ch);
            else
                Printer.write(ch);

            ch = Keyboard.read();
        }
    }
}
```

Why is Software rotting?

- NEW REQUIREMENT: Extend to be able to read from file
- NEW REQUIREMENT: Extend to be able to write alternatively to file
- …
- => a lot more special cases and your software gets bigger and messier and unmaintainable with every step

How to avoid rotting Software Design

- What can we do against it?
- How to avoid that software starts to rot?

Why is Software rotting?

- **Iterative MERCILESS Refactoring**
  - Not in an extra refactoring iteration
  - Not at the end of each iteration
  - Not every Friday
    - Whenever the Software has to change!
    - Continuously during development!

Design is a process, not an initial step!

Design Principles

Design Principles

- Geheimnisprinzip nach Parnas
- Law of Demeter
- SOLID-Prinzipien
- Immutability

Design Principles

Geheimnisprinzip nach Parnas

Geheimnisprinzip nach Parnas

- **Keine Details** – Klienten einer Komponente zu deren Nutzung keine Kenntnis über die internen Details besitzen (müssen).
- **Fokus** – nur diejenigen Bestandteile einer Komponente nach außen zugänglich zu machen, die für andere Komponenten zur Zusammenarbeit wirklich relevant sind.
- **Abstraktion** – Business-Methoden liefern Verhalten nach außen, nicht das innere nach außen stülpen

Bewertung: Geheimnisprinzip nach Parnas

- ✓ Details der Implementierung können ohne Rückwirkungen auf Nutzer geändert werden.
- ✓ Bessere Verständlichkeit und Nachvollziehbarkeit. Allein basierend auf der öffentlichen Schnittstelle sollte die Funktionalität ermittelbar sein.
- ✓ neue Subklassen lassen sich leichter implementieren, da in diesen gewöhnlich nur wenige Anpassungen notwendig sind
- X minimaler Mehraufwand zur Implementierung von Zugriffsmethoden zur Datenkapselung und für deren Aufruf erforderlich

Design Principles

Law of Demeter

Law of Demeter

- **Ziel** – **Kopplung** auf ein verständliches und wartbares Maß **reduzieren**.
- **Verstoß** – mehrere/viele Methodenaufrufe wie folgt mithilfe der **.-Notation** miteinander verknüpfen:

```
getPreferencesService().getDimension(MAIN_WINDOW_ID).setWidth(700);
getPreferencesService().getColorScheme(OCEAN).getTextColor().setColor(BLUE);

if (getCommandProcessor().getPool().getSize() >= MAX_POOL_SIZE)
{
    // warning
}
else
{
    // process command
}
```

Law of Demeter

- **diverse Annahmen** – etwa darüber, dass die Komponenten alle zugreifbar und korrekt initialisiert sind.
- **Fragilität / Folgeänderungen** – Änderungen an den Details genutzter Klassen führen nahezu zwangsläufig auch zu Änderungen in der eigenen Klasse

```
getPreferencesService().getDimension(MAIN_WINDOW_ID).setWidth(700);
getPreferencesService().getColorScheme(OCEAN).getTextColor().setColor(BLUE);

if (getCommandProcessor().getPool().getSize() >= MAX_POOL_SIZE)
{
    // warning
}
else
{
    // process command
}
```

Law of Demeter

- **Daher sind .-Verkettungen potenziell „böse“**

```
car.getOwner().getAddress().getStreet();
```

- **Extraktion einzelner Objekte macht es nicht besser**

```
Owner owner = car.getOwner();
Address ownerAddress = owner.getAddress();
Street ownerStreet = ownerAddress.getStreet();
```

- **Idee: Kontrolle über die beteiligten Objekte**

Law of Demeter

- »*Don’t talk to strangers*«

- «Don't call us, we'll call you»

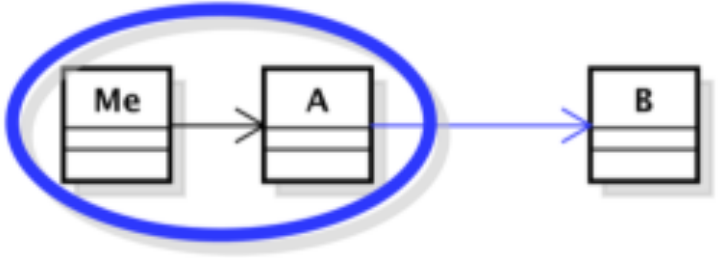
Law of Demeter

- *»Don't talk to strangers« ...*

```
myObject.Never().Talks().To().Strangers();
```

- Namensbrücke: »***Only talk to your immediate friends.***«

- Wie erreicht man das? Regeln zur Gestaltung von Aufrufen:
  1. Methoden der eigenen Klasse,
  2. Methoden von Objekten, die als Parameter übergeben werden,
  3. Methoden von Objekten, die das eigene Objekt selbst erzeugt, oder
  4. Methoden assoziierter Klassen.



Law of Demeter

Was ist mit Fluent Interfaces?

Verstoßen die nicht gegen das Law of Demeter?

Law of Demeter – Fluent Interfaces

- **NEIN!**
- **Sie besitzen eine komplett andere Designausrichtung**
- **Vor allem operieren auf ein und demselben Objekt (somit durch Regel 4 erlaubt)**

```
Report report = new ReportBuilder()
        .withHeader("Law of Demeter Report")
        .withBorder(2, Color.BLACK)
        .titled("Fluent Interfaces are fine with Law of Demeter")
        .writtenBy("Michael Inden")
        .outputAs(OutputType.PDF)
        .build();
```

SOLID

- **S** R P – Single Responsibility Principle
- **O** C P – Open Closed Principle
- **L** S P – Liskov Substitution Principle
- **I** S P – Interface Segregation Principle
- **(D** I P – Dependency Inversion Principle)

Quelle: http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

S R P – Single Responsiblity Principle

- *genau eine klar definierte Aufgabe erfüllen*
- ***wer viele Dinge auf einmal tut, dem gelingen selten alle gut.***
- hohe Kohäsion, also einen hohen Zusammenhalt
- Orthogonalität = Funktionalitäten ohne (größere) Nebenwirkungen einfach miteinander kombinieren

- **Indiz für Verstoß: dass es schwerfällt, prägnante Namen für eine Methode oder eine Klasse zu finden oder dass dieser Name mehrere Verben oder Nomen enthält.**
- schiere **Methodenlänge** ein recht guter Indikator

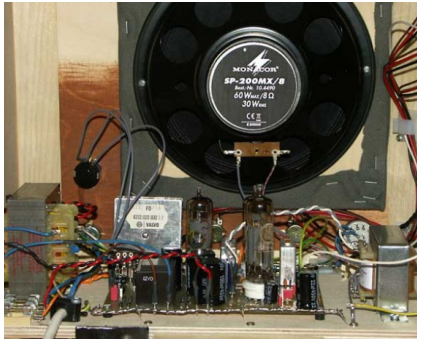
OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat

Quelle: http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

Open Closed Principle

Open for Modifications



Closed for Extensions

Open for Extensions. Closed for Modifications

O C P – Open Closed Principle

- leichte Erweiterbarkeit
- korrekte Kapselung sowie Trennung von Zuständigkeiten.
- *sollte sich eine Klasse nach ihrer Fertigstellung nur noch dann ändern müssen, wenn komplett neue Anforderungen oder Funktionalitäten zu integrieren sind oder aber Fehler korrigiert werden müssen*.

Open Closed Principle - Example

The following code does not conform to OCP:

```
public class Drawer {
  public void drawAll(List<Shape> shapes) {
     for (Shape shape : shapes) {
          if (shape instanceof Circle) {
                  drawCircle((Circle) shape);
          }
          if (shape instanceof Square) {
                  drawSquare((Square) shape);
          }
     }
  }
  ...
```

Open Closed Principle - Example

The following code does **conform** to OCP:

```
public class Drawer {
  public void drawAll(List<Shape> shapes) {
     for (Shape shape : shapes) {
          shape.draw()
     }
  }
  ...
```

O C P – Open Closed Principle

- Beispiel **Spieleapplikation** mit verschiedenen **Bonuselemente**, wie Extraleben oder Zusatzausrüstungen, als Anreiz
- Aufgabe: ein neues Level gestalten und dort neue Arten von Bonuselementen zu integrieren.
- Wunsch: möglichst einfach und erweiterbar, idealerweise nur neue Klassen für die neuen, speziellen Bonuselemente erstellen
- Folgt das Basisdesign bereits dem OCP: Dann besitzen alle Bonuselemente ein ***gemeinsames Interface*** oder eine (abstrakte) Basisklasse
- Die restliche Applikation ist kaum oder im besten Fall gar nicht von Änderungen bzw. Erweiterungen der Bonuselemente betroffen

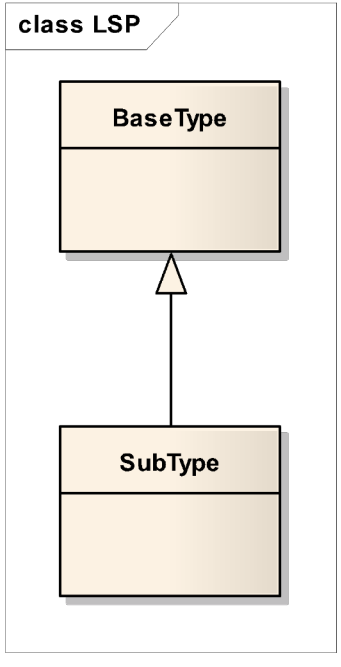
Liskov Substitution Principle

Quelle: http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

Liskov Substitution Principle

"Subtypes must be substitutable for their base type"

(Barbara Liskov, 1988)



Wherever an object of BaseType is used,

an instance of SubType could be used instead.

```
BaseType b = new SubType();
b.doSomething();
```

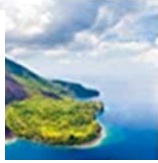
A simple example of LSP Violation

```
public class B {

   public String getName() {
      return "Base";
   }

}
```

```
public class E extends B {

   public String getName() {
      throw new UnsupportedOperationException();
   }

}
```

```
public class SomeClient {

   @Override

   public String lowerName(B b) {
      return b.getName().toLowerCase();

   }

}
```

SomeClient makes an assumption about B that does not hold for E



```
class CalculationException extends Exception
{
    // ...
}
class SpecialCalculationException extends CalculationException
{
    // ...
}

class BaseFigure
{
    Number calcArea() throws CalculationException
    {
        return new Double(getWidth() * getHeight());
    }
}

class Polygon extends BaseFigure
{
    // Speziellere Rückgabe (Double extends Number) und speziellere Exception
    Double calcArea() throws SpecialCalculationException
    {
        // Bewusst, um gleich ein Problem zu zeigen
        return null;
    }
}
```

L S P – Liskov Substituion Principle

```
class Client
{
    void doSomethingWithFigure(final BaseFigure figure)
    {
        final Number result = figure.calcArea();
        // NullPointerException für Polygon
        final double area = result.doubleValue();
        // ...
    }
}
```

Conclusion / Pragmatics

- Inheritance is more than just an "is a" relationship
- Inheritance means "substitutable“
- Therefore you have to think hard if an inheritance-relation in your design is appropriate

- Document the expectations of clients:
  Specify the contract: Pre- & Post-conditions, Invariants
  (most important: contracts in interface classes!)

Design Principles

Spezialfall trotz „is-a“

Liskov Substituion Principle – Beispiel

```
public class Rectangle
{
    private int height;
    private int width;

    public Rectangle(int height, int width)
    {
        this.height = height;
        this.width = width;
    }

    public int computeArea()
    {
        return this.height * this.width;
    }

    public void setHeight(int height) { this.height = height; }
    public void setWidth(int width) { this.width = width; }

    public int getHeight() { return this.height; }
    public int getWidth()  { return this.width;  }
}
```

L S P – Liskov Substituion Principle

```
public class Square extends Rectangle
{
    public Square(int sideLength)
    {
        super(sideLength, sideLength);
    }

    protected void setSideLength(final int sideLength)
    {
        // gleiche Seitenlänge sicherstellen
        super.setHeight(sideLength);
        super.setWidth(sideLength);
    }

    // HINWEISE und PROBLEME SPÄTER
    @Override
    public void setHeight(int height) { setSideLength(height); }
    @Override
    public void setWidth(int width) { setSideLength(width); }
}
```

L S P – Liskov Substituion Principle

```
public class RectangleTest
{
    @Test
    void testAreaCalulaction()
    {
        Rectangle rect = new Rectangle(7, 6);

        rect.setWidth(5);
        rect.setHeight(10);


        assertEquals(50, rect.computeArea());
    }
}
```

Für Rectangle funktioniert das gut, aber laut LSP sollte man auch Square nutzen können

L S P – Liskov Substituion Principle

```
public class SquareTest
{
    @Test
    void testAreaCalulaction()
    {
        Rectangle rect = new Square(7);

        rect.setWidth(5);
        rect.setHeight(10);

        assertEquals(50, rect.computeArea());

        // Quadrat entweder 25 oder 100 je nach Reihenfolge ...
    }
}
```

- Für Square ist die Reihenfolge entscheidend, in jedem Fall passt das für Rectangle berechnete Ergebnis jedoch nicht!
- eigentlich würde man die set()-Methoden gerne verbieten ...

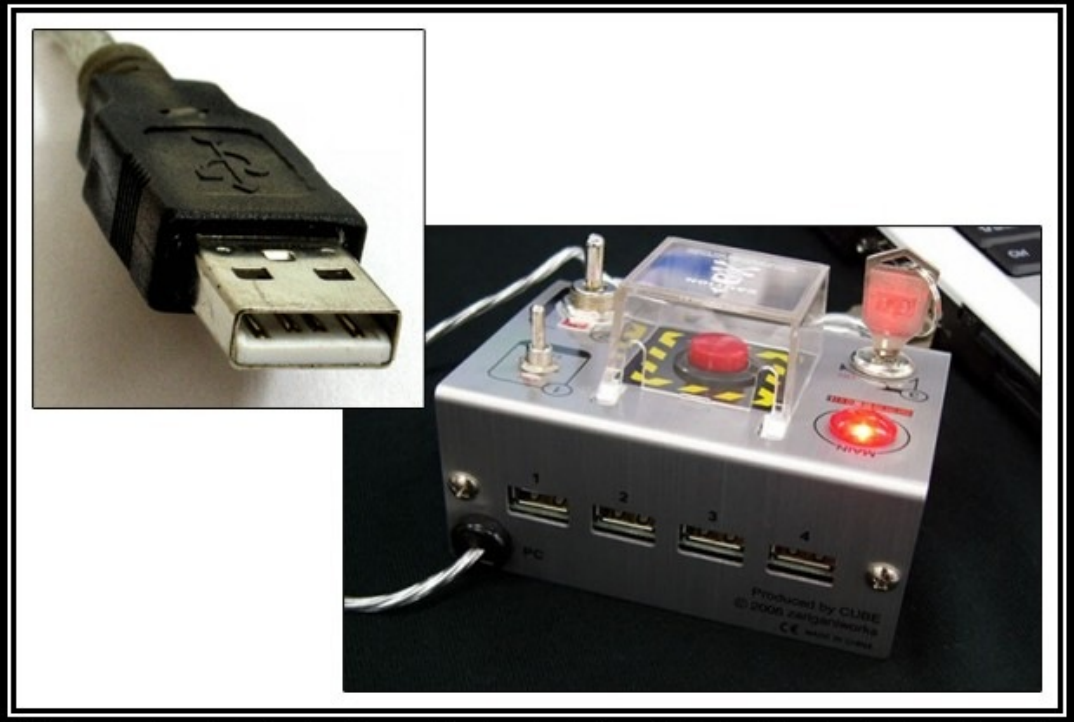
Law of Demeter

Alles klar soweit? Schauen wir mal!

L S P – Liskov Substituion Principle

```
Rectangle rect1 = new Rectangle(7, 6);
rect1.setWidth(5)
rect1.setHeight(5)


Rectangle rect2 = new Rectangle(7, 7);
```

Hmmm ... man kann also aus einem Rechteck semantisch ein Quadrat machen, aber es hat den Typ `Rectangle` .... Da kommt noch etwas Arbeit auf uns zu ;-)



INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

Quelle: http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

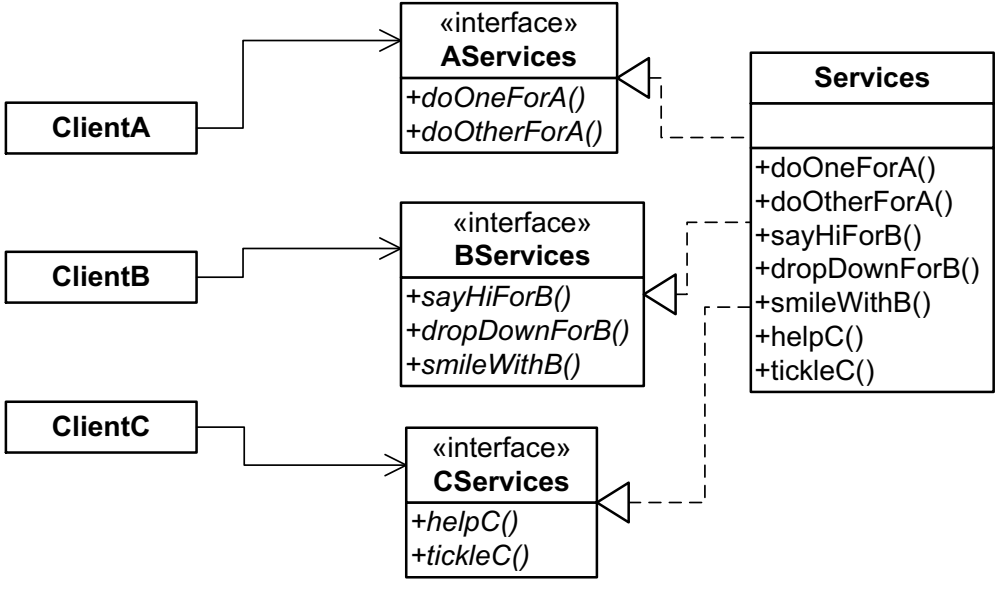
I S P – Interface Segregation Principle

- ***Erstelle möglichst spezifische, auf die jeweilige Aufgabe oder auf ihn als Klienten zugeschnittene Schnittstelle***
- Oftmals sieht man in der Praxis eher zu breite oder zu unspezifische Interfaces, die folglich fast immer auch Funktionalität anbieten, die ein Klient nicht benötigt, etwa wie folgt:

```
interface IUniversalFileCustomerAndPizzaService
{
    void scanDisk(final Drive drive) throws IOException;
    boolean rename(final File fileToRename,
                   final String newName) throws IOException;

    Customer findCustomerByName(final String name);
    Iterable<Customer> getAllCustomers(final FilterCondition filterCondition);

    boolean orderPizza(final long customerId, final Pizza pizza);
}
```

Interface Segregation Principle

Several client specific interfaces are better than one single, general interface



not only one interface for all clients

one interface per kind of client

I S P – Interface Segregation Principle

```
interface IFileService
{
    void scanDisk(final Drive drive) throws IOException;
    boolean rename(final File fileToRename,
                   final String newName) throws IOException;
}

interface ICustomerService
{
    Customer findCustomerByName(final String name);
    Iterable<Customer> getAllCustomers(final FilterCondition filterCondition);
}

interface IPizzaService
{
    boolean orderPizza(final long customerId, final Pizza pizza);
}
```

- der Entwurf einer gelungenen Schnittstelle ist gar nicht so leicht
- Es gilt, die »**richtige**« **Granularität** zu finden.
- Das benötigt etwas Erfahrung, Fingerspitzengefühl und auch ein wenig Ausprobieren – insbesondere auch eine Betrachtung aus Sicht möglicher Nutzer.

DEPENDENCY INVERSION PRINCIPLE

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

Quelle: http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

Dependency Inversion Principle

- “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”

- Hääääääääää?

- **Verwende möglichst Interfaces (bzw. abstrakte Klassen), um (konkrete) Klassen voneinander zu entkoppeln**

Vielen Dank für die Teilnahme und happy coding!