Design Principles

Agenda

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

Bad / Good Design

Bad / Good Design

- Exercise

Woran erkennt man schlechtes Design?

- der Sourcecode ist wenig verständlich, unter anderem verursacht durch ***ungünstige Namensgebung*** oder eine ***fehlende Dokumentation*** komplizierterer Stellen.

- Es existiert ***unnötige Komplexität*** oder ein extrem flexibles Design mit hohem Variantenreichtum, obwohl diese Extras (teilweise) nicht benötigt werden.

- Erweiterungen oder ***Modifikationen lassen sich nur schwierig durchführen*** und besitzen z.T. *große Auswirkungen* auch in anderen Teilen des Sourcecodes

- Das zu lösende Problem kann nicht anhand der Implementierung abgeleitet werden, 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.

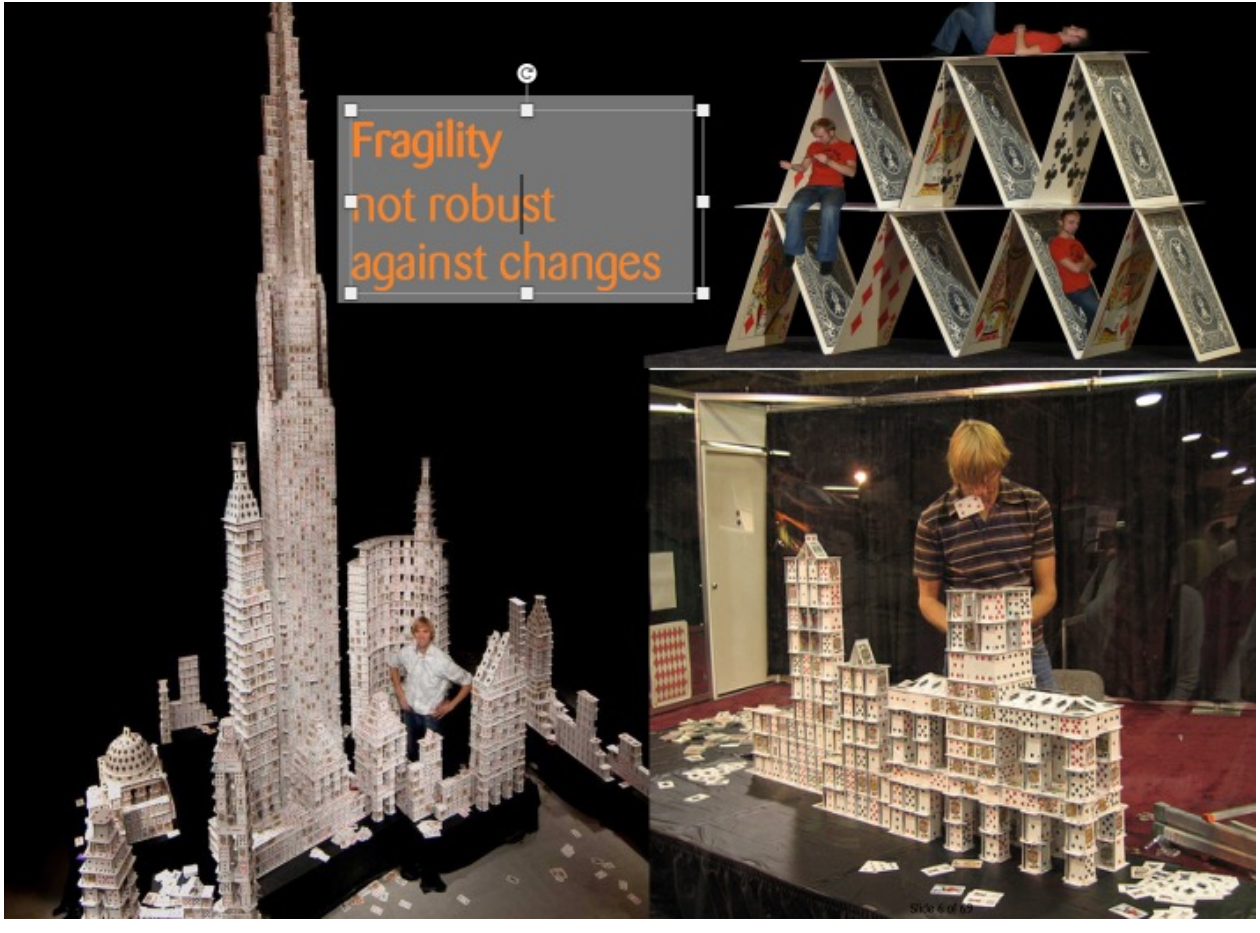
Design Smells (by Robert C. Martin)

- 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

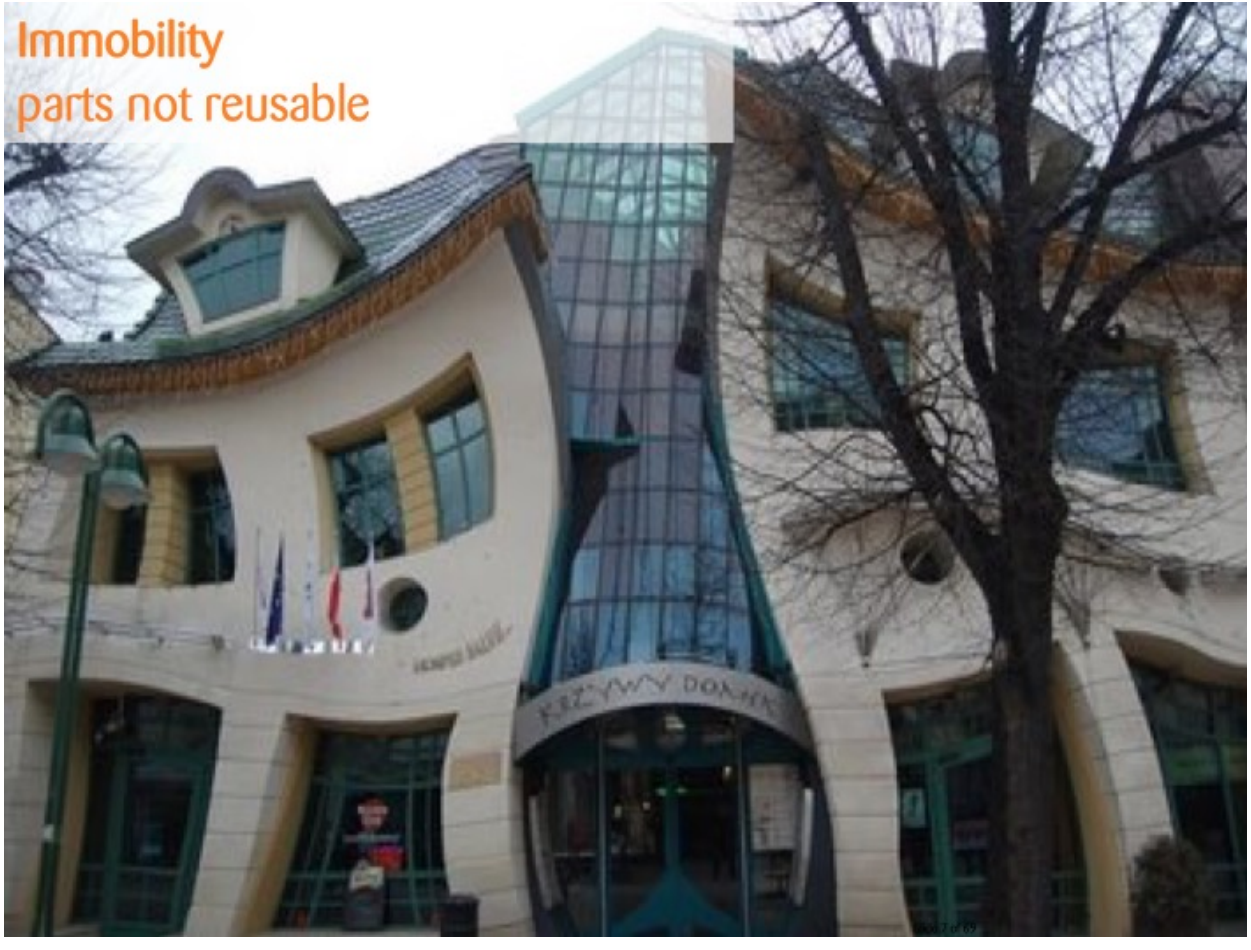
Design Smells (by Robert C. Martin)

- Rigidity: difficult to change, chain of changes

Design Smells (by Robert C. Martin)

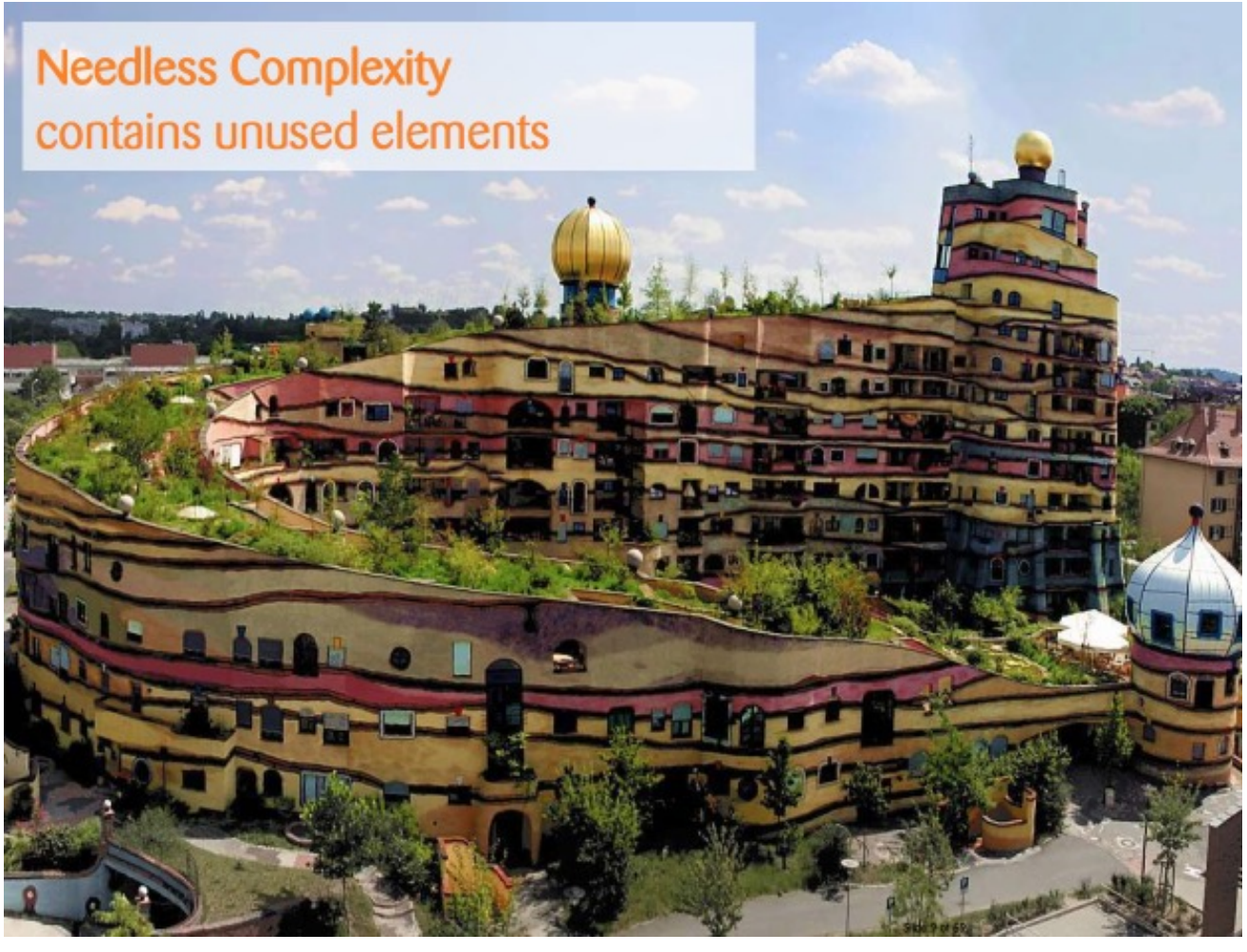


Design Smells (by Robert C. Martin)



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

- ✓ Es entsteht ein eher lose gekoppeltes System mit einer guten Modularisierung.
- ✓ Testbarkeit der einzelnen Teile unabhängig voneinander. Auch die Anzahl der benötigten Tests reduziert sich, da Aufrufe nur über die öffentliche Schnittstelle erfolgen und Zustandsänderungen besser kontrolliert werden können.
- ✓ Details der Implementierung können ohne Rückwirkungen auf Nutzer geändert werden.

Bewertung: Geheimnisprinzip nach Parnas

- ✓ 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

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»

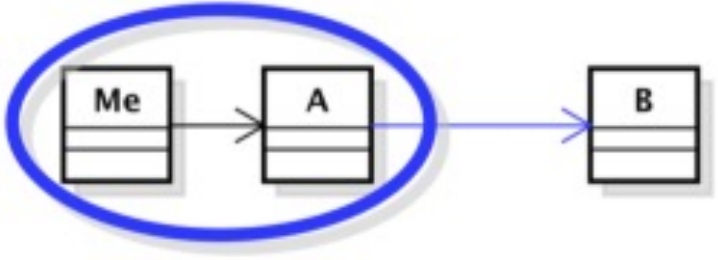
Law of Demeter

- *»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.



Bewertung: Law of Demeter

- ✓ es entsteht ein eher lose gekoppeltes System mit wenigen Abhängigkeiten, was die Wiederverwendbarkeit erleichtert.
- ✓ einzelne Klassen leichter änderbar und weniger Ausbreitung von Folgeänderungen im System
- ✓ Leichtere Testbarkeit aufgrund geringerer Anzahl an Abhängigkeiten
- X Mehr Methoden im Interface der eigenen Klasse, weil einige Methoden eventuell nun nicht mehr über Indirektion angesprochen werden können.

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

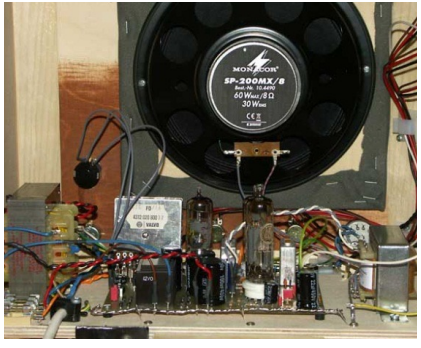
OPEN CLOSED PRINCIPLE

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

Open Closed Principle

Modules conforming to OCP …

… are open for extension:
behaviour of module can be extended to meet new requirements.

… are closed for modification:
extending the module does not result in changes to the source or binary code of the module.

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 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 Objectives

- Use inheritance the right way
- Know the possible traps of Inheritance and Polymorphism

More than IS-A

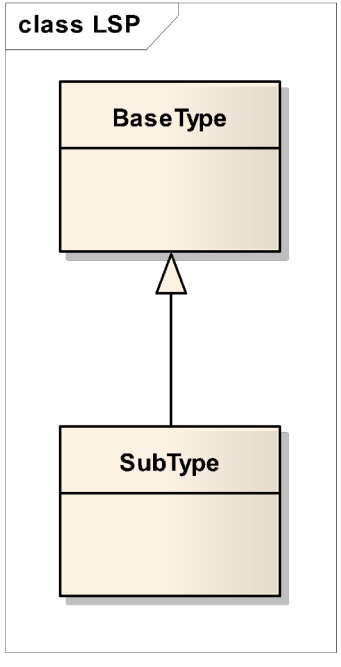
- Know about "Design by Contract" to help making your design and code less fragile
- Take care of important Preconditions, Postconditions and Invariants in your Design
- Apply this knowledge in your designs, documentations and unit tests

Design by Contract

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

Pay attention when you inherit …

Your Subclass has to ensure that the following holds:

The "Contract" of the Superclass must still be fulfilled.

Contract:

What users expect of the operation …

- Given Preconditions are fulfilled before execution
- The Postconditions must be fulfilled after execution

```
// Precondition: Enough money on my account
public void transferMoney(Account other, int amount);
// Postcondition: money moved to other account
```

Contracts

Which letterbox would you trust??

→ Contracts belong to the interface of a class

→ Document any pre- and postconditions.

Precondition	Post until 18:00, Porto CHF 1.-- (A-Post)	What the Client needs to ensure
Postcondition	Delivery next day until 12:00	What the Supplier needs to ensure

Contract of a Class

Precondition: What users must ensure before calling an operation

Postcondition: The least that users can expect to be fulfilled after execution
(when precondition was fulfilled)

Invariant: Special condition on the state of an object that is always true
(before and after each public operation execution)

Belongs to the interface of a class:

- either express it as conditions (if supported by language)
- or document it in text style at least

Inheriting a Contract

Your Subclass has to ensure that the following holds:

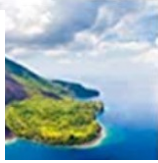
The "Contract" of the Superclass must still be fulfilled.

What happens with the Contract in the Subclass?

- It is inherited too
- Preconditions may be relaxed (but not restricted!)
- Postconditions may be restricted (but not relaxed!)

In other words … you are allowed to:

- Expect less from the user (caller) but not more
- Provide more than the user expects but not less



```
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 ...

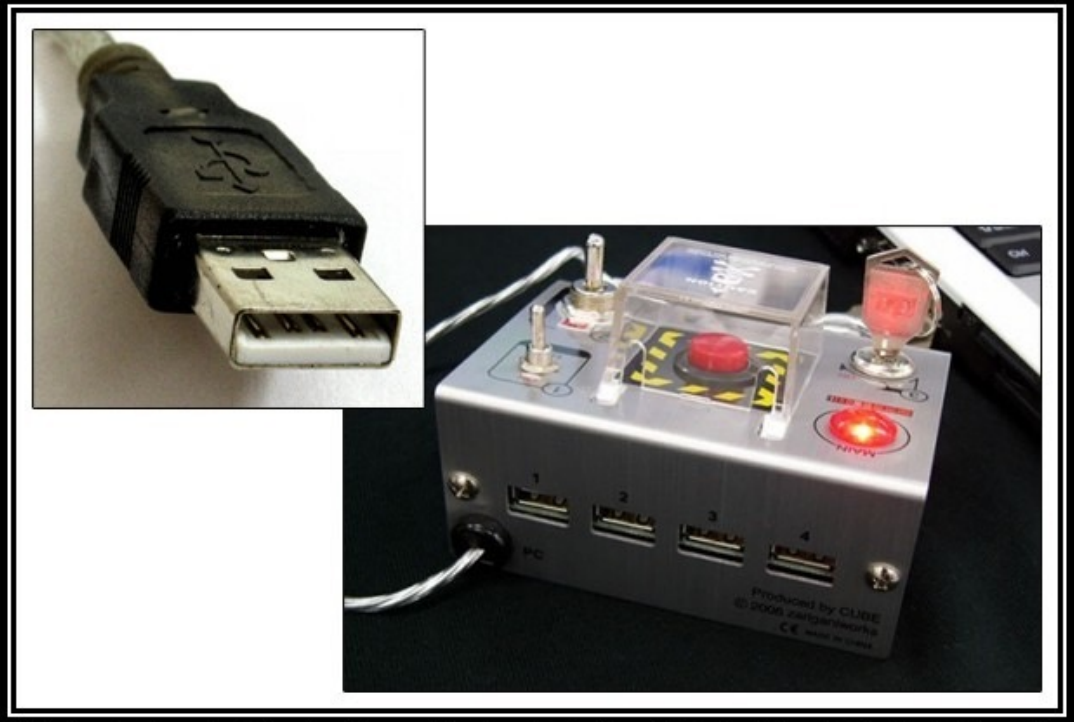
Law of Demeter

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/

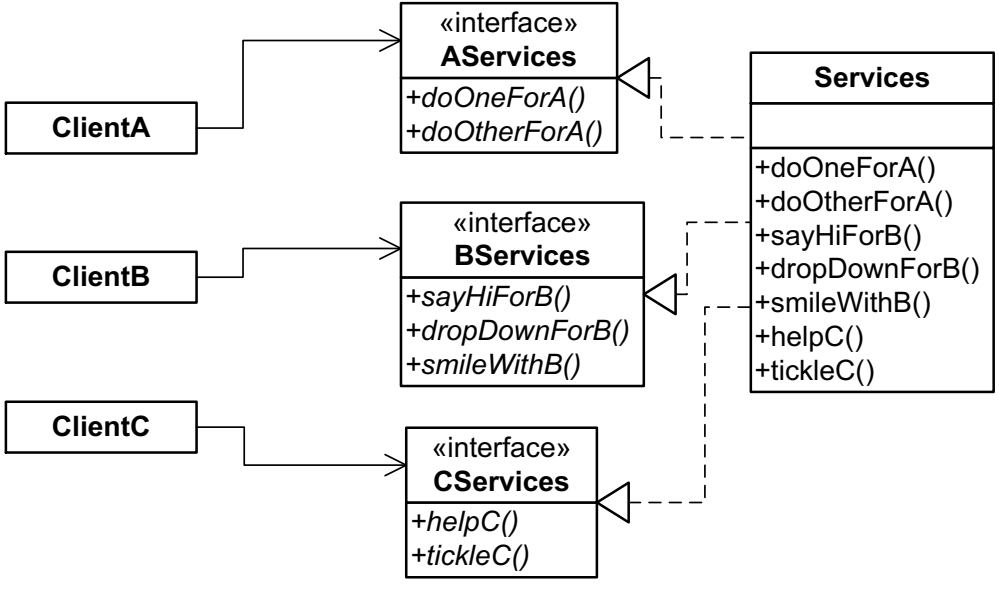
I S P – Interface Segregation Principle

- ***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/

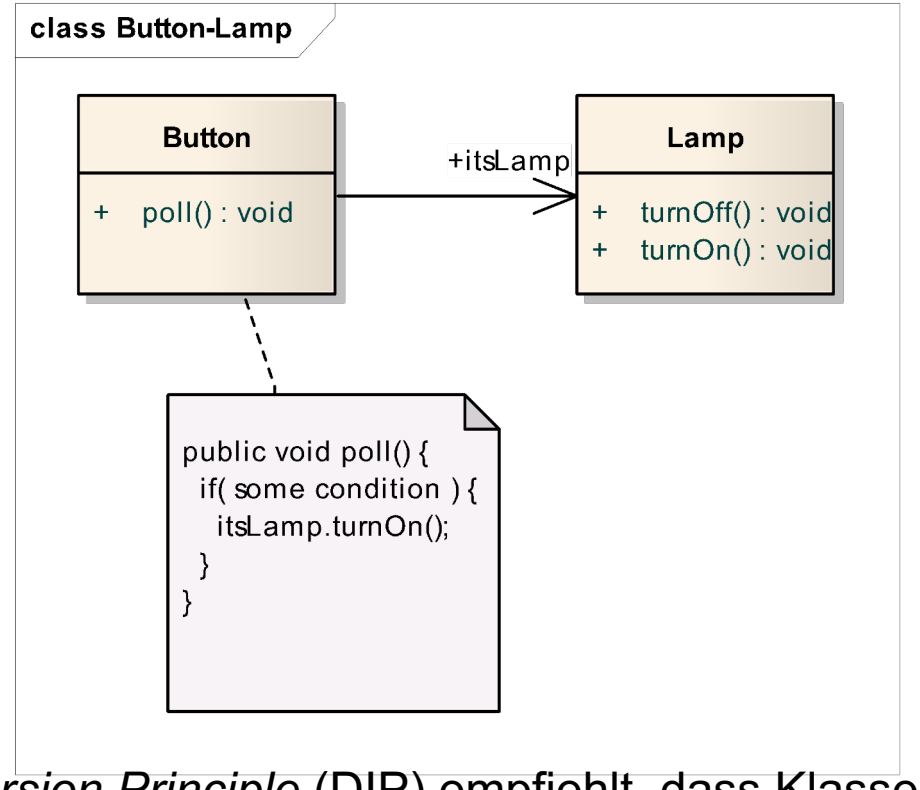
Dependency Inversion Principle

- “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?

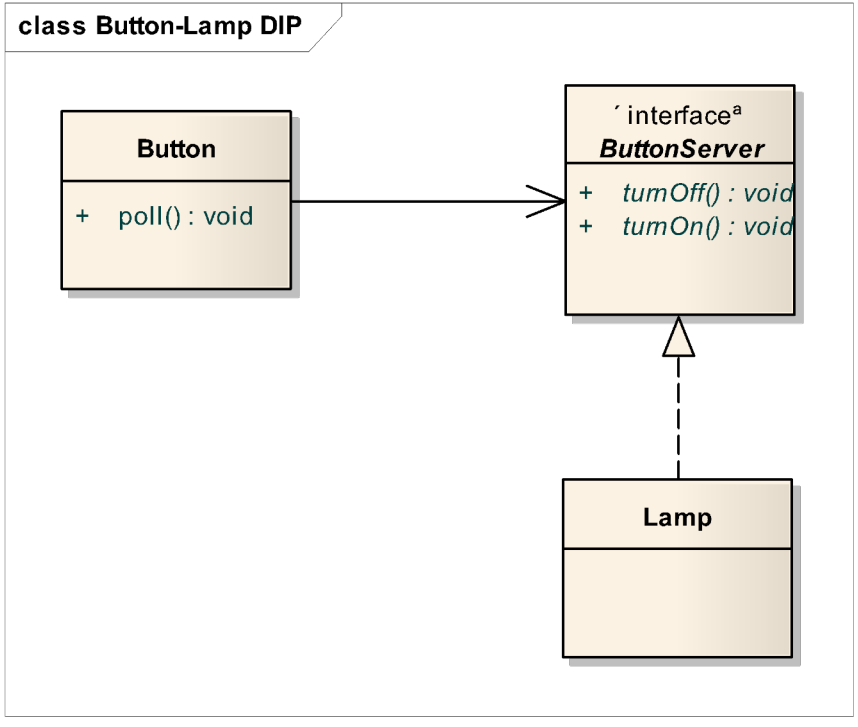
- **Verwende möglichst Interfaces (bzw. abstrakte Klassen), um (konkrete) Klassen voneinander zu entkoppeln**

DIP What is wrong here?

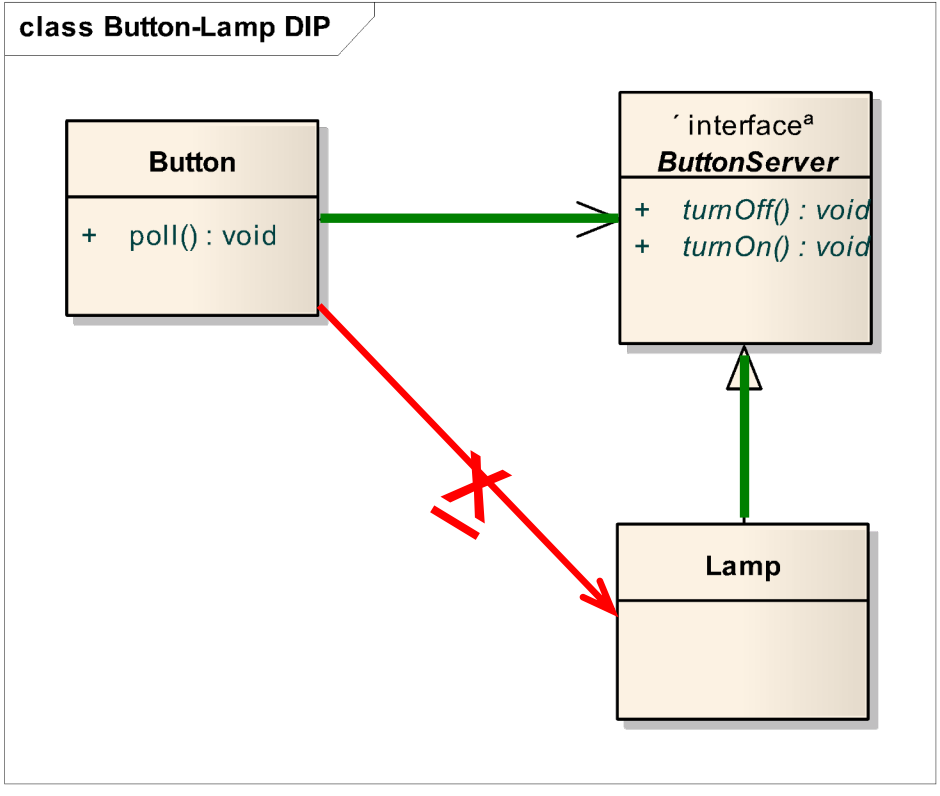


Das *Dependency Inversion Principle* (DIP) empfiehlt, dass Klassen möglichst unabhängig von anderen konkreten Klassen sein sollen, indem eine Abhängigkeit von einer konkreten Klasse durch eine Referenz auf deren Schnittstelle aufgelöst wird.

Introducing Button “Server” Interface



DIP Inverting the dependency



Button depends no more on Lamp

Instead Lamp has a dependency on the abstraction (=ButtonServer).

```
public class PizzaService
{
   private final Discount discount;
   private final CustomerDAO customerDAO;

   private final Map<Long, Receipt> customerToReceipt = new HashMap<>();

   public PizzaService()
   {
      // Direkte Abhängigkeiten
      discount = new Discount();
      customerDAO = new CustomerDAO();
   }

   public void orderPizza(final long customerId, final Pizza pizza)
   {
      final Customer customer = customerDAO.findById(customerId);
      customerToReceipt.putIfAbsent(customerId, new Receipt(customer));

      final Receipt receipt = customerToReceipt.get(customerId);
      final double price = discount.apply(pizza);
      receipt.addEntry(pizza, price);
   }

    ...

}
```

Analyse Was an Abhängigkeiten ungünstig ist, wollen wir nun ein wenig genauer betrachten: Die eben vorgestellte Klasse `PizzaService` besitzt verschiedene Abhängigkeiten. Das ist im Besonderen ungünstig, wenn wir die Funktionalität der Klassen testen wollen. Das wäre aus folgenden zwei Gründen schwierig:

1. Es besteht eine Fixierung auf die Datenquelle `CustomerDAO`, und diese Klasse liest die Kundendaten aus einer Datenbank. Dadurch können wir keine Tests mit einem eng umrissenen und vordefinierten Testdatenbestand vornehmen. Außerdem sollten wir tunlichst keine Testdaten in den sensiblen Kundendatenbestand einspielen, um Missverständnisse, Fehlbestellungen oder sonstige Fallstricke zu vermeiden.
2. Die Auswirkungen verschiedener Rabattaktionen, etwa ein Einführungsangebot oder eine Happy Hour, sind schwierig zu testen, da momentan fix der durch die Klasse `Discount` realisierte Rabatt zur Preisreduktion genutzt wird.

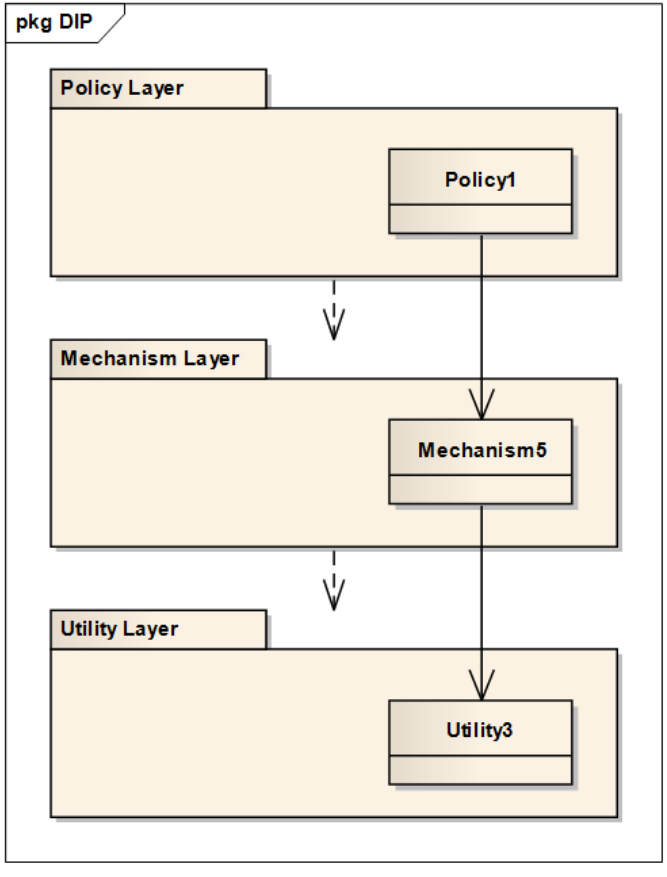
```
public class PizzaService
{
   private final ICustomerRepository customerRepository;

   private Map<Long, Receipt> customerToReceipt = new HashMap<>();

   // Konstruktor-Injektion
   public PizzaService(final ICustomerRepository customerRepository)
   {
      this.customerRepository = customerRepository;
   }

   // Method-Injektion
   public void orderPizza(final long customerId, final Pizza pizza,
                          final IDiscountStrategy discountStrategy)
   {
      final Customer customer = customerRepository.findById(customerId);
      customerToReceipt.putIfAbsent(customerId, new Receipt(customer));

      final Receipt receipt = customerToReceipt.get(customerId);
      final double price = discountStrategy.apply(pizza);
      receipt.addEntry(pizza, price);
   }

   ...

}
```

Dependency Inversion Principle

„All well structured architectures have clearly defined layers, with each layer providing some coherent set of services through a well-defined and controlled interface.“

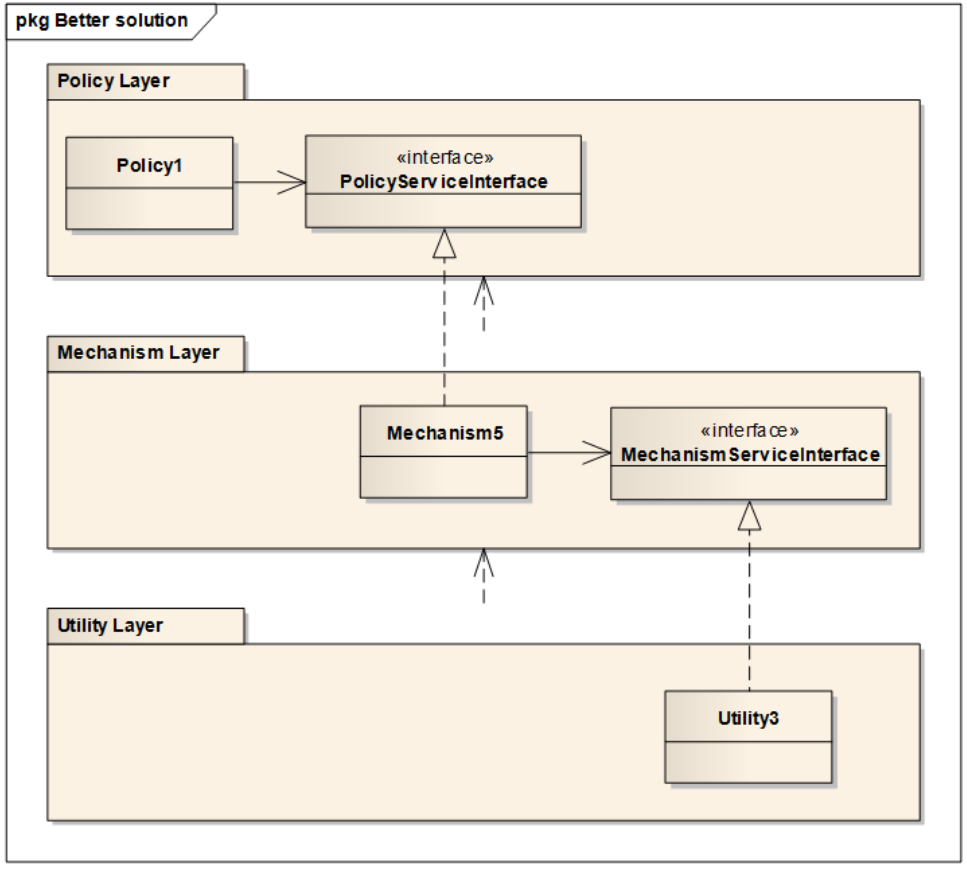
Grady Booch, 1996

Simple view of that quote



Dependency Inversion Principle

Service Provider view of that quote (SPI / API)



Design Principles

Immutability

Immutability

- **Immutable objects** sind Instanzen von Klassen, die sich nach ihrer Erzeugung nicht mehr verändern

- **Kochrezept**
  1. Die Klasse muss `final` sein, um Subklassen zu verhindern
  2. Attribute müssen `final` sein, idealerweise `private`, sodass Veränderungen unterbunden werden
  3. Kein `set()`-Methoden
  4. Lesezugriff über `get()`-Methoden
  5. Initialisierung aller Attribute im Konstruktor
  6. Immer gültiger Zustand, wenn einmalige Zustandsprüfung vor der Konstruktion erfolgt

Beispiel für Immutability

```
class ImmutablePerson
{
    private final String name;
    private final LocalDate dateOfBirth;

    public ImmutablePerson(final String name, final LocalDate dateOfBirth)
    {
        this.name = name;
        this.dateOfBirth = dateOfBirth;
    }

    public String getName()
    {
        return name;
    }

    public LocalDate getDateOfBirth()
    {
        return dateOfBirth;
    }
}
```

Beispiel für Immutability

1. - 5. wie zuvor
6. *Immer* **gültiger Zustand**, wenn einmalige Zustandsprüfung bei der Konstruktion erfolgt

```
public ImmutablePerson(final String name, final LocalDate dateOfBirth)
{
    if (name == null || name.isEmpty())
        throw new IllegalArgumentException("name must not be null or empty!");

    if (dateOfBirth.isAfter(LocalDate.now()))
        throw new IllegalArgumentException("birthday must not be in the future");

    this.name = name;
    this.dateOfBirth = dateOfBirth;
}
```

Design Principles

Stimmt das wirklich?

Kochrezept für Immutability, letzte Prise

- **Bei Referenzvariablen muss man ganz besonders aufpassen!**

```
public class ImmutableSurprise
{
    private final int id;
    private final int[] values;
    private final List<String> additionalValues;

    public ImmutableSurprise(final int id, final int[] values,
                             final List<String> additionalValues)
    {
        // Sicherheitsprüfungen ...
        this.id = id;
        this.values = values;
        this.additionalValues = additionalValues;
    }

    // getXyz() ...
```

Kochrezept für Immutability, letzte Prise

- **Probieren wir es einmal aus ...**

```
public static void main(final String[] args)
{
    List<String> values = new ArrayList<>(List.of("Tim", "Tom", "Mike"));
    ImmutableSurprise is = new ImmutableSurprise(42,
                                                 new int[] { 1, 2, 3},
                                                 values);


    System.out.println(is);

    is.getValues()[1] = -123;
    is.getAdditionalValues().add("UNEXPECTED");

    System.out.println(is);
}
```

Kochrezept für Immutability, letzte Prise

- **Alle Attribute sind final und es gibt keine set()-Methoden, aber die Klasse ist nicht immutable!**
- Durch Javas Referenzsemantik können wir Referenzen auf die Objekte erhalten und deren Zustand modifizieren:

```
System.out.println(is);

is.getValues()[1] = -123;
is.getAdditionalValues().add("UNEXPECTED");

System.out.println(is);
```

```
ImmutableSurprise [id=42, values=[1, 2, 3],
                   additionalValues=[Tim, Tom, Mike]]
ImmutableSurprise [id=42, values=[1, -123, 3],
                   additionalValues=[Tim, Tom, Mike, UNEXPECTED]]
```

Kochrezept für Immutability, letzte Prise

- **Bei Referenzvariablen muss man ganz besonders aufpassen! Das erfordert folgende Aktionen:**

- **IN**
  => Bei der Konstruktion muss man die Inhalte der Referenzvariablen kopieren (clone(), Copy-Konstruktor, deep copy)
- **OUT**
  => Bei der Rückgabe von Inhalten veränderlicher Typen muss man Kopien erstellen

- **MODIFY**
  => Methoden, die zuvor interne Daten geändert haben, müssen neue Immutable-Objekte erzeugen

Immutability Take aways

- **Vorteile von Immutable Objects**
  - Gut verständlich, leicht nachvollziehbar
  - Ideal für Value Objects, die einen Wert und keine Enitity repräsentieren*
  - Ebenfalls ideal für Data Transfer Objects
  - Gut als Key für Maps zu nutzen, da unveränderlich
  - Frei von Seiteneffekten, damit frei von negativen Überraschungen
  - Thread-Safe bei Design
  - Keine Probleme im Kontext von Multithreading

* Ein Wert kann mehrere Attribute haben (Money, Gewicht, ...), besitzt keine Identität.

Immutability Take aways

- **Nachteile von Immutable Objects**
  - Änderungen erfordern neue Objekte und führen zu mehr Speicherbedarf
  - Unpassend für häufig modifizierte Daten
  - Leicht schlechtere Performance (beachtenswert für High-Performance-Apps/Games, allerdings nahezu irrelevant in Kombination mit REST+ DB)

* Ein Wert kann mehrere Attribute haben (Money, Gewicht, ...), besitzt keine Identität.

Übungen

Vielen Dank für die Teilnahme und happy coding!