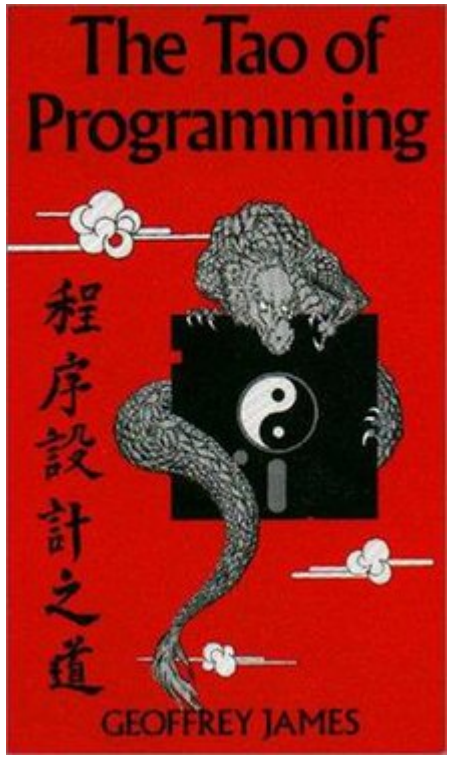
Objektorientierte Programmierung

Hochschule Bochum
WS 19/20
Dr.-Ing. Darius Malysiak

V. OOP Software Engineering

Tao of Programming („The Way of Programming“):

- Äußerst humorvoll geschriebenes Buch im Stil taoistischer Texte.
- 1987 von Geoffrey James veröffentlicht.
- Kurzfassung unter http://www.mit.edu/~xela/tao.html verfügbar.
- Behandelt Programmierung im Allgemeinen (keine Objektorientierung).



https://upload.wikimedia.org/wikipedia/en/thumb/d/d2/The_Tao_of_Programming.jpg/220px-The_Tao_of_Programming.jpg

Thus spake the Master Programmer:
"When a program is being tested, it is too late to make design changes."

Thus spake the Master Programmer:
"A well-written program is its own Heaven; a poorly-written program is its own Hell."

Thus spake the Master Programmer:
"Though a program be but three lines long, someday it will have to be maintained."

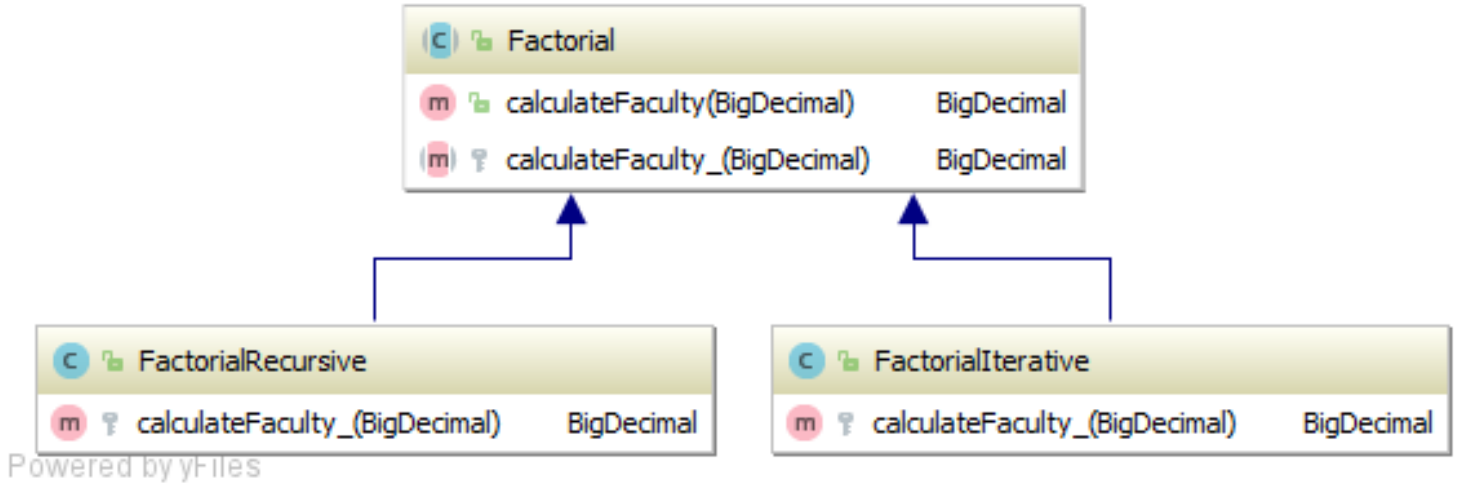
V. OOP Software Engineering

Klassen von Anti-Patterns:

Vorzeitige Optimierung (Algorithmik):

> *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*
> *-- Donald Knuth*

Zunächst eine schnell realisierbare Implementierung vornehmen. Erst bei Bedarf über weitere Variationen bzw. Verbesserungen nachdenken.



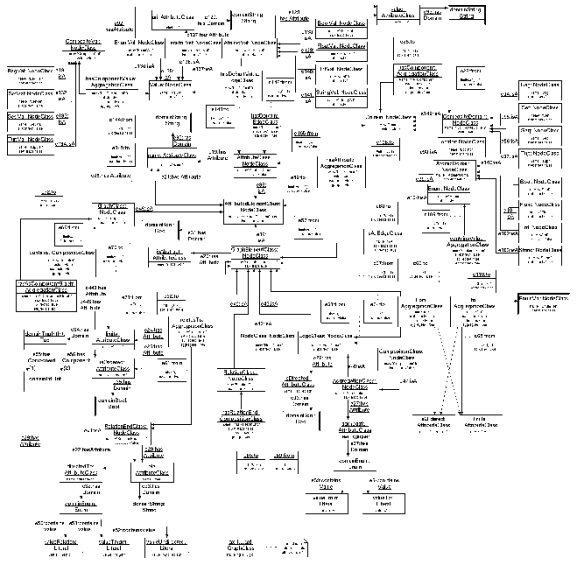
V. OOP Software Engineering

Klassen von Anti-Patterns:

Bikeshedding / Overengineering (Architektur):

> *Every once in a while we'd interrupt that to discuss the typography and the color of the cover. And after each discussion, we were asked to vote. I thought it would be most efficient to vote for the same color we had decided on in the meeting before, but it turned out I was always in the minority! We finally chose red. (It came out blue.)*
> *--Richard Feynman*

Overengineering vermeiden! Das konkrete Problem soll unter Verwendung von guten Programmierprinzipien gelöst werden. Hypothetische oder subjektive Probleme sind maximal sekundär.



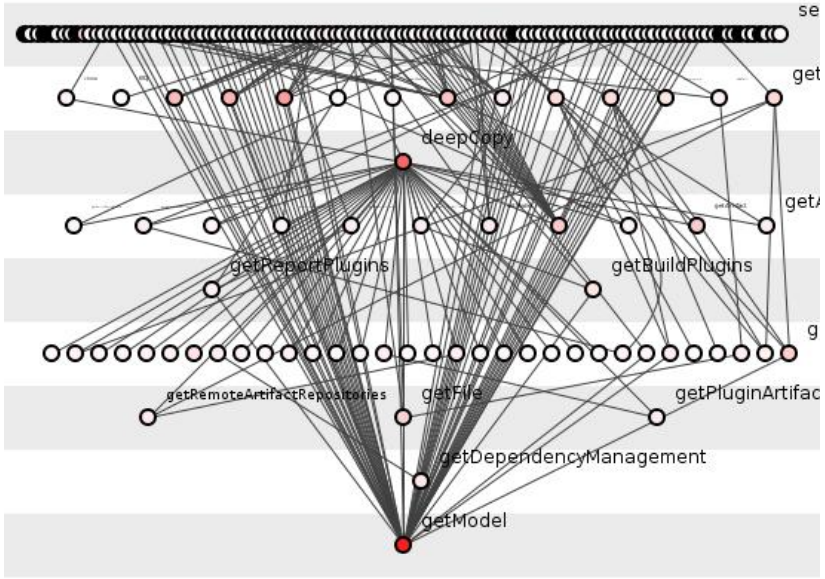
http://www.gupro.de/GXL/examples/metaschema/graph/gxl-1.0.png

V. OOP Software Engineering

Klassen von Anti-Patterns:

God Classes (Abhängigkeiten):

> *Simple is better than complex.*
> *-- Tim Peters, The Zen of Python*



http://edmundkirwan.com/image/godobject/inside-god-object.jpg

Klassen sollten feingranular hinsichtlich ihrer Funktionalität bzw. Aufgabe sein!

V. OOP Software Engineering

Klassen von Anti-Patterns:

> *Explicit is better than implicit.*
> *-- Tim Peters, The Zen of Python*

Konstanten (Ausprägungen):

Ausprägungen im Sourcecode sind nicht falsch!!

```
public class Constants {
    @Autowired
    IConstantArray varArgsInjected;

    @Autowired
    ConstantSpellChecker spellChecker;

    public Constants()
    {
        spellChecker.check(varArgsInjected);
    }
}
```

V. OOP Software Engineering

API Design:

Def.: **Application Programming Interface (API)**

Eine wohldefinierte Schnittstelle, welche die bereitgestellte Funktionalität eines Moduls, einer Softwarekomponente oder einer Applikation spezifiziert. Konsumenten der Funktionalität sind i.A. weitere Module, Softwarekomponenten oder Applikationen.

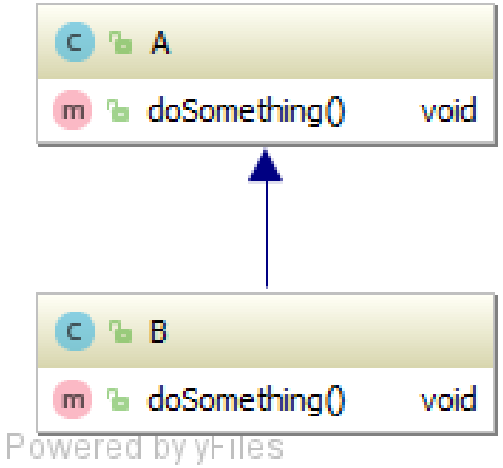
V. OOP Software Engineering

API Design:

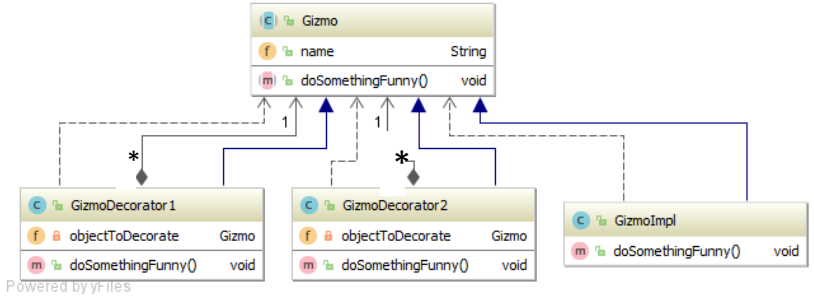
Def.: **Open Closed Principle (OCP)**

Module oder Softwarekomponenten sollen offen für Erweiterungen aber geschlossen für Veränderungen sein.

Bsp:



Vererbung



Design Patterns (hier Decorator)

V. OOP Software Engineering

API Design:

Def.: **Single Responsibility Principle (SRP)**

Module oder Softwarekomponenten sollen für genau einen funktionalen Aspekt der Softwarekomponente verantwortlich sein.

Bsp:

```
public class Minion extends Employee {
    public double calculatePay(){…};
    public void saveEmployeeToDatabase(){…}
    public int reportWorkHours(){…}
}
```

Berechnet das Gehalt.

Persistiert das Objekt in der Datenbank.

Berichtet wie lange der Mitarbeiter gearbeitet hat.

Was ist hier katastrophal falsch?

V. OOP Software Engineering

API Design:

```
public class Minion extends Employee {
    public double calculatePay(){…};
    public void saveEmployeeToDatabase(){…}
    public String reportWorkHours(){…}
}
```

Was ist hier katastrophal falsch?

1. Ein Mitarbeiter sollte nicht selbst seine Finanzen im Sinn der Buchhaltung machen. ?Betrug durch Kindklassen?
2. Eine Entity sollte nicht selbst die Funktionalität zum Persistieren definieren. -> Trennung von Daten und Funktionalität!
3. Ein Mitarbeiter sollte nicht selbst seine Stunden im frei wählbaren Format berichten. -> Format-Chaos.

Hier sind 3 gänzlich verschiedene Aspekte aufgeführt, welche jeweils allein genug Rechtfertigung für Refactoring bringen.

V. OOP Software Engineering

API Design:

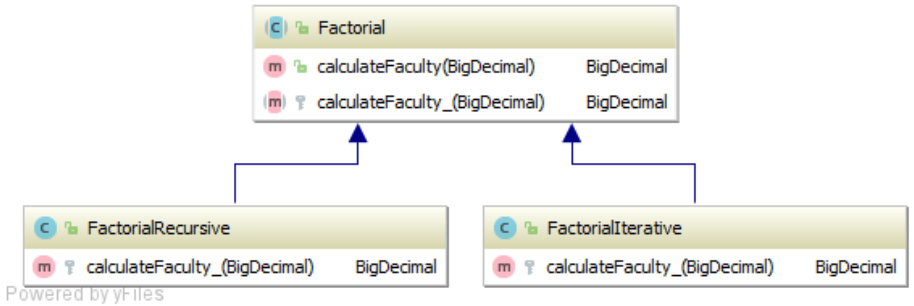
Def.: **Don‘t repeat yourself (DRY)**

Module oder Softwarekomponenten sollen Redundanzen im Sourcecode vermeiden.

Bsp:

```
@RequiredArgsConstructor
@Getter
@Setter
@EqualsAndHashCode
@Entity
public class Person {
    @NonNull
    String name;
    @EqualsAndHashCode.Exclude Integer age = 10;
}
```

Annotation Syntax Sugar



Design Patterns (hier Template Method Pattern)

V. OOP Software Engineering

API Design:

Def.: **Interface Segregation Principle (ISP)**

Schnittstellen sollen hinsichtlich der Methoden überschaubar bleiben, eine feine Granularität bzgl. technischen und fachlichen Anforderungen sollte angestrebt werden.

Bsp:

Jeweils eine Methode pro Interface

```
public class SubMinion implements PayCalculation, PersistEntity, ReportTime {
    public double calculatePay(){ return 0;};
    public void saveEmployeeToDatabase(){}
    public String reportWorkHours(){return "";}
}
```

Hierdurch ist hinsichtlich eines Refactorings zur Einhaltung des SRPs keine Klassenrestrukturierung nötig da bei zuvor eingehaltenem OCP mit z.B. Decorater Patterns nur Schnittstellen als Parameter erwartet werden!

V. OOP Software Engineering

API Design:

Def.: **Dependency Inversion Principle (DIP)**

1. Softwarekomponenten sollen von gemeinsamen abstrakten Komponenten abhängen anstatt hierarchisch voneinander.
2. Abstrakte Komponenten sollen nicht von Ausprägungen abhängen.

Bsp:

```
public class Customer {
    Transaction t;
    void doSomething(){…}
}
```

triggert

```
public class Transaction {
    public void performTransaction(){…};
}
```

Customer benötigt eine Transaktion und verwendet diese in einer sehr komplexen ‚doSomething' Methode.

Was passiert wenn Customer nun mit einer verbesserten Transaction Klasse umgehen soll? -> Refactoring (Logik und Tests) von ‚doSomething'.

V. OOP Software Engineering

API Design:

Bsp:

```
public class Customer {
    ITransaction t;
    void doSomething(){…}
}
```

triggert

```
public class Transaction implements ITransaction{
    public void performTransaction(){…};
}
```

Hierdurch ist die aus fachlicher Sicht hierarchisch höhergelegene Klasse ‚Customer' unabhängig von der konkreten Implementierung der Transaction.

Bem.: Dies setzt ein klare Definition der Schnittstelle voraus!!

1.

2.

V. OOP Software Engineering

API Design:

Def.: **Application Programming Interface (API)**

Eine wohldefinierte Schnittstelle, welche die bereitgestellte Funktionalität eines Moduls, einer Softwarekomponente oder einer Applikation spezifiziert. Konsumenten der Funktionalität sind i.A. weitere Module, Softwarekomponenten oder Applikationen.

S O L I D

Bem.: In Java ist das Liskovsche Substitutionsprinzip durch Vererbung ausgeprägt. Hierdurch kann Java die SOLID Kriterien teilweise implizit erfüllen, i.A. sind diese jedoch unabhängig von der Programmiersprache oder Objektorientierung.