

Software Engineering 1



Anna Prenzel



Einführung

Ziele der heutigen Veranstaltung:

∞ Wissen über Entwurfsmuster
vertiefen



Wiederholung

- ⌘ Was versteht man in der Programmierung unter dem Prinzip der Delegation?
- ⌘ Welche Entwurfsmuster nutzen Delegation?
- ⌘ Was ist der Unterschied zwischen diesen Patterns:
 - Simple Factory
 - Factory Method
 - Abstract Factory
- ⌘ Erklären Sie die Implementierung von Singletons!



Wiederholung

Welche Entwurfsmuster nutzen Delegation?

Fazit: Fast jedes Entwurfsmuster nutzt Delegation.

➡ Entwurfsmuster befassen sich u.a. mit der geschickten Aufteilung von Verantwortlichkeiten zwischen mehreren Klassen.



Wiederholung

Singleton

-instance: Singleton

-Singleton()

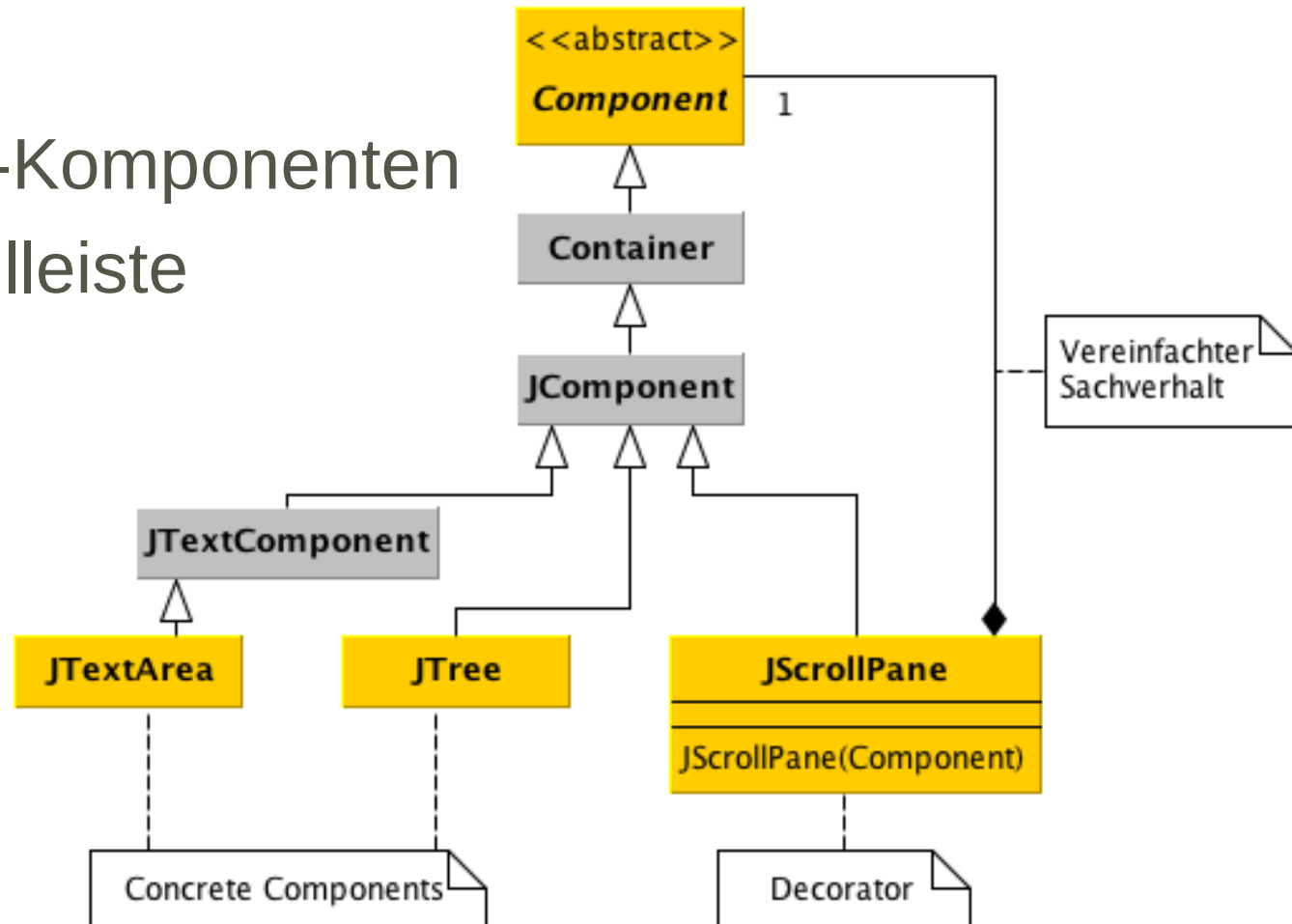
+getInstance(): Singleton



Patterns - typische Beispiele

Decorator

Beliebige GUI-Komponenten
mit einer Scrollleiste
versehen





Patterns - typische Beispiele

Decorator

```
JTextArea textComponent = new JTextArea(12,20);
```

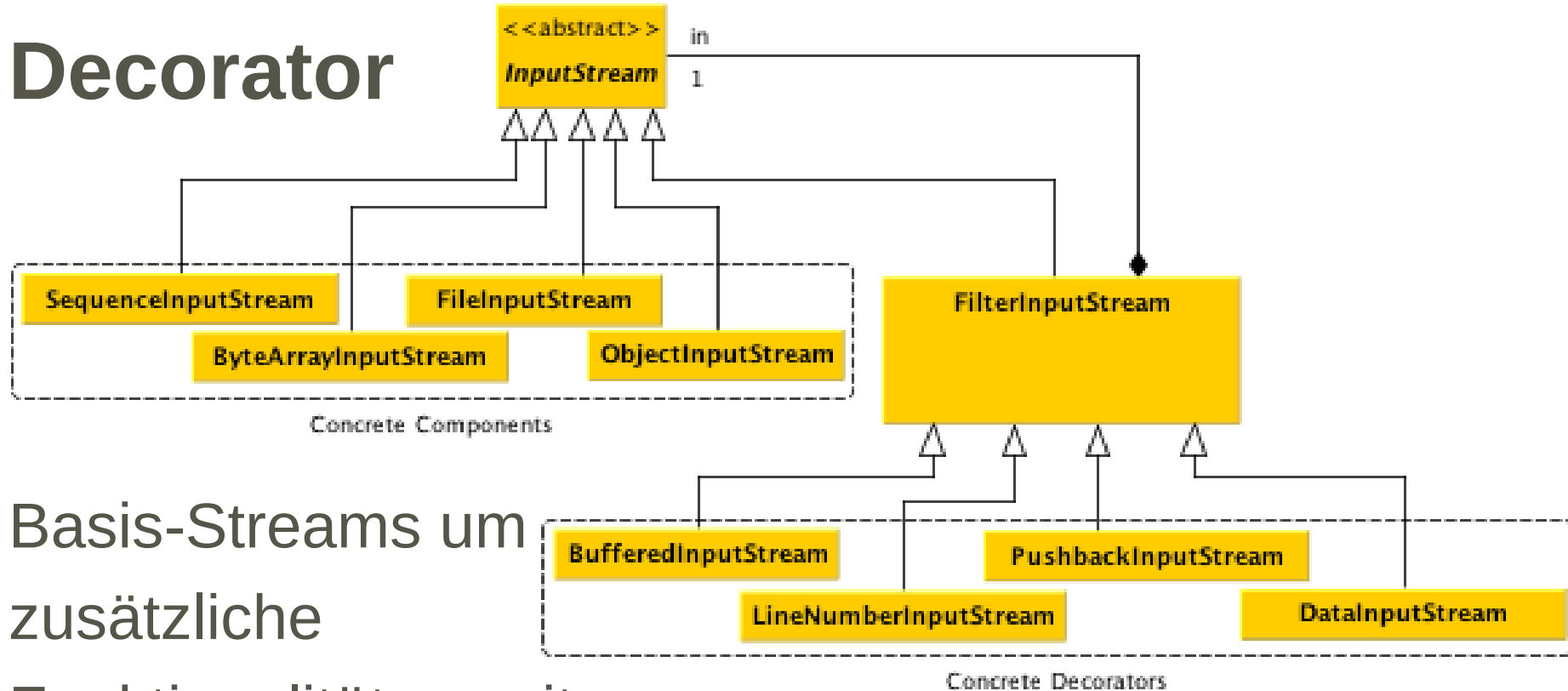
```
JScrollPane scrollComponent =  
    new JScrollPane(textComponent);
```

```
this.add(scrollComponent);
```



Patterns - typische Beispiele

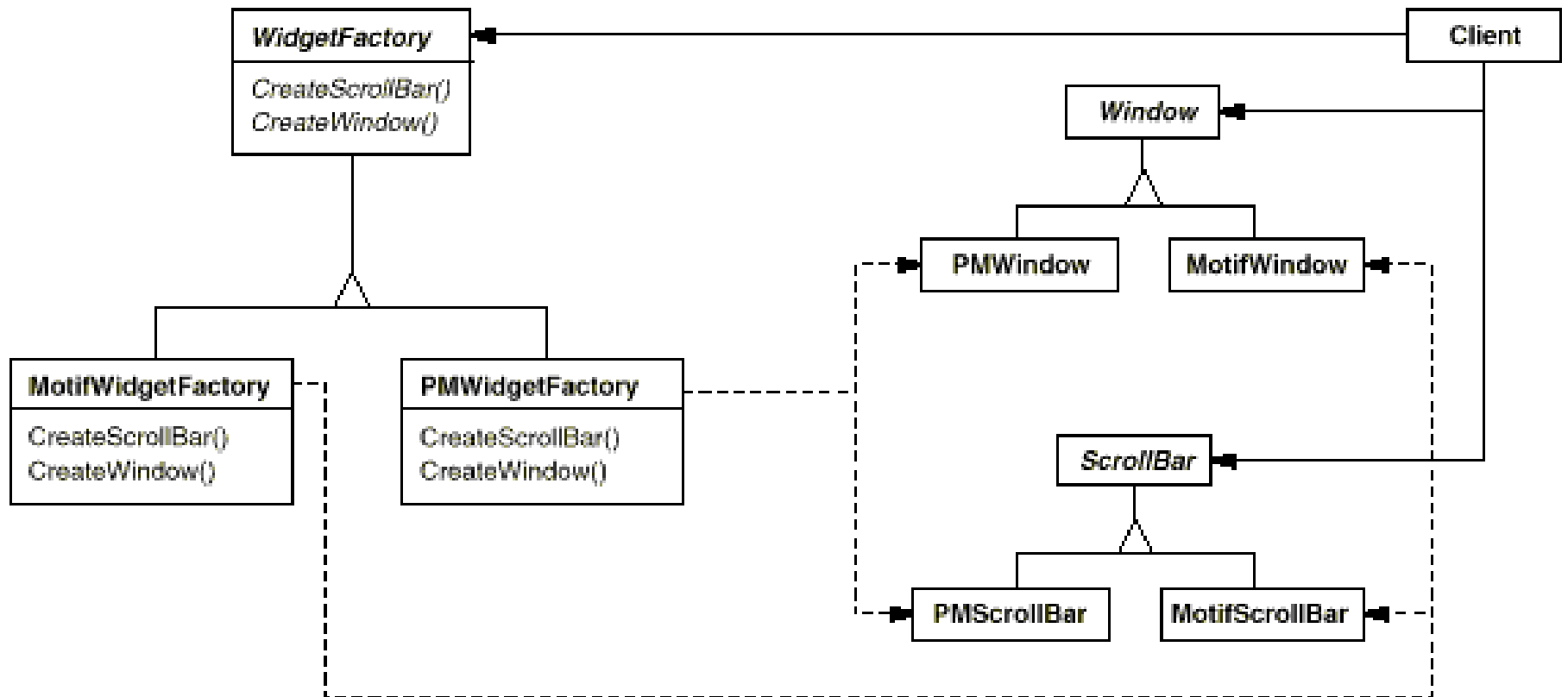
Decorator



Basis-Streams um
zusätzliche
Funktionalität erweitern

Abstract Factory

eine Familie von GUI-Steuerelementen mit einem bestimmten Look-and-Feel erzeugen





Arten von Patterns

GRASP = General Responsibility Assignment Software Patterns

1. Information Expert
2. Creator
3. Controller
4. Low Coupling





Arten von Patterns

GRASP = General Re
Assignment Software

1. Information Expert

Wie weise ich
Objekten
Verantwortlich
keiten zu?

2. Creator

Wer erstellt
ein Objekt
A?

Welches erste
Objekt nimmt eine
Nachricht aus der
UI-Schicht
entgegen?

3. Controller

Wie kann man
den Einfluss von
Änderungen
reduzieren?

4. Low Coupling



Arten von Patterns

GRASP (Erweitert)

- 5. High Cohesion
- 6. Polymorphismus
- 7. Pure Fabrication
- 8. Indirection
- 9. Protected Variations



Arten von Patterns

Information Expert

Weisen Sie eine Verantwortlichkeit der Klasse zu, die über die Informationen verfügt, die benötigt werden, um die Verantwortlichkeit zu erfüllen.

GRASP (Craig Larman)

„Es ist möglich, die detaillierten Prinzipien und Denkprozesse zu benennen und zu erklären, die erforderlich sind, um den grundlegenden Objektentwurf, nämlich die Zuweisung von Verantwortlichkeiten an Objekte, zu erfassen. →

GRASP (Craig Larman)

Die GRASP-Prinzipien oder Patterns sind eine Lernhilfe. Sie sollen Ihnen helfen, das Wesen des Objektentwurfes zu verstehen und beim Entwurf bestimmte Denkprozesse methodisch, rational und erklärbar anzuwenden.“



Arten von Patterns

GoF = Gang of Four

23 Patterns

- ✎ Adapter
- ✎ Decorator
- ✎ Factory
- ✎ Singleton
- ✎ ...

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch



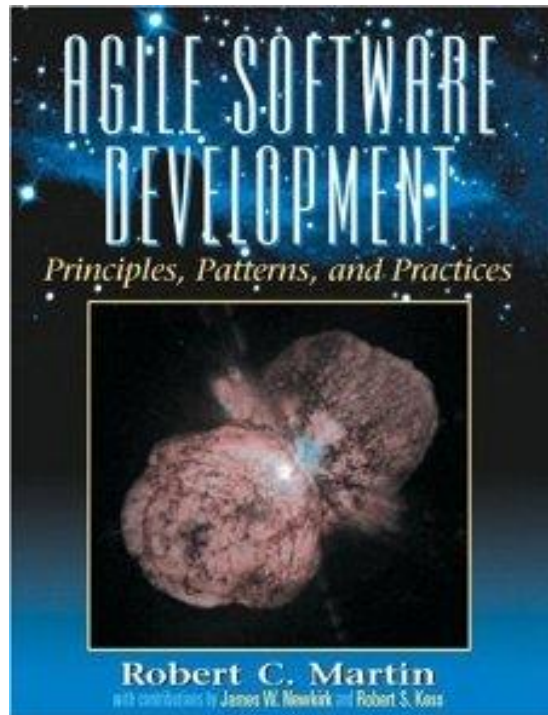
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



Weitere Prinzipien

Weitere Design-Prinzipien

Robert Martin





Weitere Prinzipien

SOLID (Robert Martin)

- ⌘ **Single Responsibility Principle**
- ⌘ **Open/Closed Principle**
- ⌘ **Liskov Substitution Principle**
- ⌘ **Interface Segregation Principle**
- ⌘ **Dependency Inversion Principle**



Weitere Prinzipien

SOLID (Robert Martin)

Ziele

Vermeidung von

- ∞ Starrheit (Rigidity)
- ∞ Zerbrechlichkeit (Fragility)
- ∞ Unbeweglichkeit (Immobility)



Weitere Prinzipien

SOLID (Robert Martin)

Starrheit

- ∞ Software ist schwer zu ändern.
- ∞ Jede Änderung zieht eine Reihe weiterer erforderlicher Änderungen nach sich.
- ∞ Folge: es wird vermieden, die Software zu ändern, solange es geht.



Weitere Prinzipien

SOLID (Robert Martin)

Zerbrechlichkeit

- ☞ Eine Änderung der Software an Stelle X führt dazu, dass an Stelle Y irgendetwas nicht mehr richtig funktioniert.
- ☞ Jeder Fix löst nicht nur ein Problem, sondern führt eine ganze Reihe neuer Probleme ein.



Weitere Prinzipien

SOLID (Robert Martin)

Unbeweglichkeit

- ⌘ Komponenten der Software können schwer wiederverwendet werden.
- ⌘ Es ist zu aufwändig und riskant, die gewünschte Komponente von den ungewünschten Komponenten zu trennen, die mit ihr verbunden sind.
- ⌘ Es ist einfacher, die Komponente selbst neu zu schreiben.

Open Closed Prinzip

Eine wiederverwendbare Klasse sollte offen sein für Erweiterungen, aber geschlossen für Änderungen.

(Bertrand Meyer, 1988)

- D.h. der Code der Klasse muss nicht geändert werden, um dem System ein neues Feature hinzuzufügen.
- Mögliche Umsetzungen dieses Prinzips:
 - Subklasse bilden und Methode überschreiben.
 - Bestehende Klassen durch Delegation wiederverwenden.
- Frameworks sind Beispiele für die Anwendung dieses Prinzips:
 - Um Frameworks benutzen zu können, müssen Basisklassen erweitert, Interfaces implementiert werden.
 - Frameworks sind i.R. 3th-party Software und als solche nicht änderbar (ausser bei Open Source).

Quelle: Franz-Josef Elmer, Universität Basel

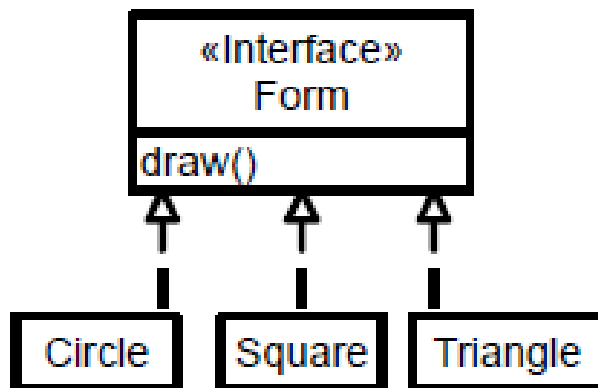
Open Closed Prinzip - Beispiel

- Folgendes Codefragment verletzt dieses Prinzip:

```
public void draw(Form form) {  
    if (form.type == CIRCLE) {  
        drawCircle(form);  
    } else if (form.type == SQUARE) {  
        drawSquare(form);  
    }  
}
```

- Die Methode draw() soll eine geometrisch Form zeichnen.
- Falls das Programm um eine neue Form (z.B. Dreieck) erweitert wird, muss diese Methode angepasst werden.

- Lösung durch Polymorphismus:



```
public void draw(Form form) {  
    form.draw();  
}
```

Quelle: Franz-Josef Elmer, Universität Basel

Open Closed Prinzip – Beispiel 2

```
public double Preis() {  
    const decimal StammkundenRabatt = 0.95m;  
    switch(kundenart) {  
        case Kundenart.Einmalkunde:  
            return menge * einzelpreis;  
        case Kundenart.Stammkunde:  
            return menge * einzelpreis * StammkundenRabatt;  
        default:  
            throw new ArgumentOutOfRangeException();  
    }  
}
```

- ☞ Klasse muss modifiziert werden, wenn weitere Art der Preisberechnung erforderlich wird
- ☞ möglicherweise wird dabei der bestehende Code „beschädigt“

Open Closed Prinzip – Lösung



Strategy
Pattern
(GoF)

```
public interface IPreisRechner {  
    double Preis(int menge, double einzelpreis)  
}  
  
public class Einmalkunde : IPreisRechner {  
    public double Preis(int menge, double einzelpreis) {  
        return menge * einzelpreis;  
    }  
}  
  
public class Stammkunde : IPreisRechner {  
    const decimal StammkundenRabatt = 0.95m;  
  
    public double Preis(int menge, double einzelpreis) {  
        return menge * einzelpreis * StammkundenRabatt;  
    }  
}
```

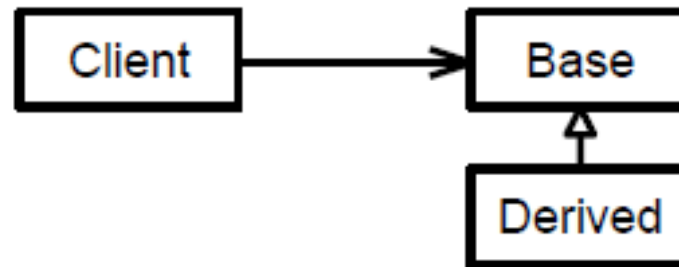
```
private IPreisRechner preisRechner;  
  
public double Preis() {  
    return preisRechner.Preis(menge, einzelpreis);  
}
```

Liskovsches Substitutionsprinzip

Instanzen von einer Klasse können durch Instanzen ihrer Subklasse ersetzt werden.

(Barbara Liskov, 1988)

– Beispiel:



- D.h. statt mit einem Objekt der Klasse Base arbeitet Client genauso gut mit einem Objekt der Subklasse Derived.
- Das ist nicht selbstverständlich, da sich die abgeleitete Klasse anders verhalten kann als die Basisklasse.

Quelle: Franz-Josef Elmer, Universität Basel

Liskovsches Substitutionsprinzip

Klasse:

```
public class Ellipse {  
    private Point focus1;  
    private Point focus2;  
    public void setFoci(Point f1,  
                        Point f2) {  
        focus1 = f1;  
        focus2 = f2;  
    }  
    public Point getFocus1() {  
        return focus1;  
    }  
    public Point getFocus2() {  
        return focus2;  
    }  
}
```

Subklasse:

```
public class Circle extends Ellipse {  
    public void setFoci(Point f1,  
                        Point f2) {  
        focus1 = f1;  
        focus2 = f1;  
    }  
}
```

Folgender Clientcode wirft ein AssertionError:

```
public void test(Ellipse e) {  
    Point p1 = new Point(1, 0);  
    Point p2 = new Point(-1, 0);  
    e.setFoci(p1, p2);  
    assert e.getFocus2().equals(p2);  
}
```

Quelle: Franz-Josef Elmer, Universität Basel

Liskovsches Substitutionsprinzip

- Die Verletzung des Liskovschen Substitutionsprinzips zeigt sich erst zur Laufzeit.
- Folgende Regeln garantieren die Einhaltung des Liskovschen Substitutionsprinzips:
 1. Die Vorbedingungen an die überschriebenen Methoden sind nicht strenger als die für die Basisklasse.
Aussagelogisch: *precondition class* $\Rightarrow$ *precondition subclass*
 2. Die Nachbedingung an die überschriebenen Methoden sind nicht schwächer als die für die Basisklasse.
Aussagelogisch: *postcondition subclass* $\Rightarrow$ *postcondition class*

Quelle: Franz-Josef Elmer, Universität Basel

Liskovsches Substitutionsprinzip

Weitere Merkmale

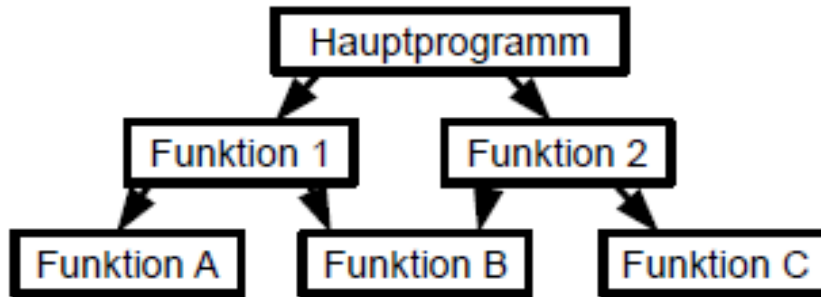
- ⌘ Wenn der Basistyp bei der Ausführung einer Methode keine Exception wirft, darf dies der Subtyp auch nicht tun
- ⌘ Wenn der Basistyp einen bestimmten Wertebereich für eine Methode erlaubt, darf der Subtyp diesen nicht einschränken

Dependency Inversion

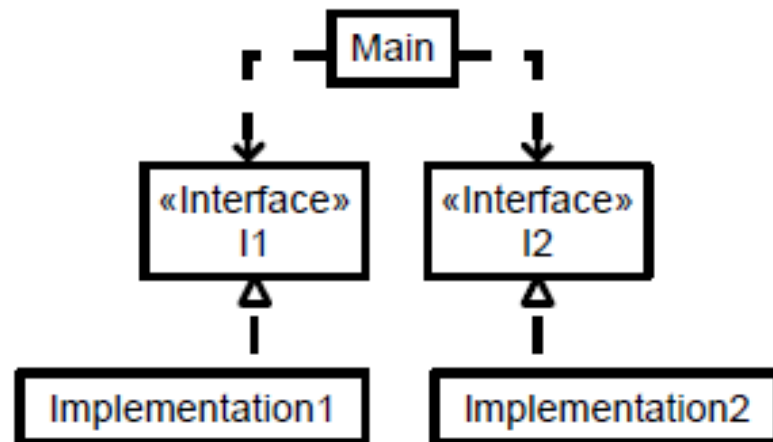
Abstraktionen sollen nicht von Details abhängen, sondern umgekehrt.

(Robert C. Martin, 1995)

- Typische Abhängigkeiten in prozeduralen Systeme:



- Umkehrung der Abhängigkeiten:



Quelle: Franz-Josef Elmer, Universität Basel

Dependency Inversion

Vorher

```
public class Lampe {  
  
    private boolean leuchtet = false;  
    public void anschalten() { leuchtet = true; }  
    public void ausschalten() { leuchtet = false; } }  
  
public class Schalter {  
    private Lampe lampe;  
    private boolean gedrueckt;  
    public Schalter(Lampe lampe) { this.lampe = lampe; }  
  
    public void drueckeSchalter() { gedrueckt = !gedrueckt;  
        if(gedrueckt) {  
            lampe.anschalten();  
        } else { lampe.ausschalten(); } } }
```

Dependency Inversion

Nachher

```
public class Lampe implements SchalterKlient{
```

```
    private boolean leuchtet = false;  
    public void anschalten() { leuchtet = true; }  
    public void ausschalten() { leuchtet = false; } }
```

```
public class Schalter {  
    private SchalterKlient klient;  
    private boolean gedrueckt;  
    public Schalter(SchalterKlient klient;) {  
        this.klient = klient; }  
  
    public void drueckeSchalter() { gedrueckt = !gedrueckt;  
        if(gedrueckt) {  
            klient.anschalten();  
        } else { klient.ausschalten(); } }  
}
```

Dependency Inversion

Nachher

```
public interface SchalterKlient{  
  
    public void anschalten();  
    public void ausschalten();  
}
```