

## 5. Entwurfsfehler, Übersicht

**Hier:** Fehler beim **Entwurf objektorientierter Strukturen**,  
bei **Anwendung objektorientierter Konzepte**.

**nicht:** Fehler in **anderen Entwicklungsphasen**:  
Anforderungsanalyse, Systemmodellierung, Grobstruktur, Projektorganisation,  
Implementierung, Validierung

Versuch und Anfang einer **Sammlung typischer Fehler**

5.1 **Missbrauch von Vererbung** (Zusammenfassung aus Abschnitt 3)

5.2 **Anti Patterns** (W.J. Brown et.al)

5.3 **Üble Gerüche im Code** - Signal zum Refaktorisieren (M. Fowler)

5.4 **Unangenehme OOP-Überraschungen** (nach E. Plödereder)

nicht erwähnt: die Negation positiver Regeln (z. B. „starke Kopplung statt schwacher“)

### Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 501

**Ziele:**

Einordnung des Themas

**in der Vorlesung:**

Erläuterungen zu den vier Abschnitten

## 5.1 Missbrauch von Vererbung

siehe Abschnitt 2; Darstellung hier im Stil der AntiPattern

### 5.1.1 Verkannter Schauspieler

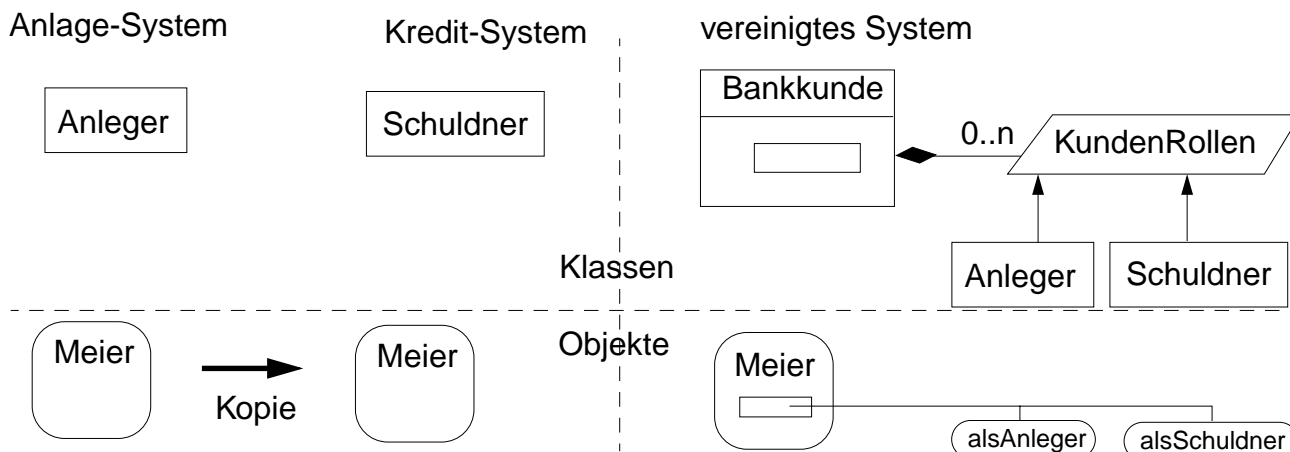
**Beschreibung:** Unterschied **Objekt und sein Verhalten** nicht modelliert

**Ursache:** evtl. wurden getrennt entworfene **Systeme zusammengelegt**

**Symptom:** ein **Objekt** soll vorübergehend oder auf Dauer seine **Klasse wechseln**.

**Abhilfe:** Objekt bleibt in allgemeiner Klasse und **spielt verschiedene Rollen**.

**Beispiel:**



© 2013 bei Prof. Dr. Uwe Kastens

## Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 502

**Ziele:**

Schema einprägen

**in der Vorlesung:**

- Erläuterungen dazu am Beispiel;
- Erinnerung an Abschnitt 2

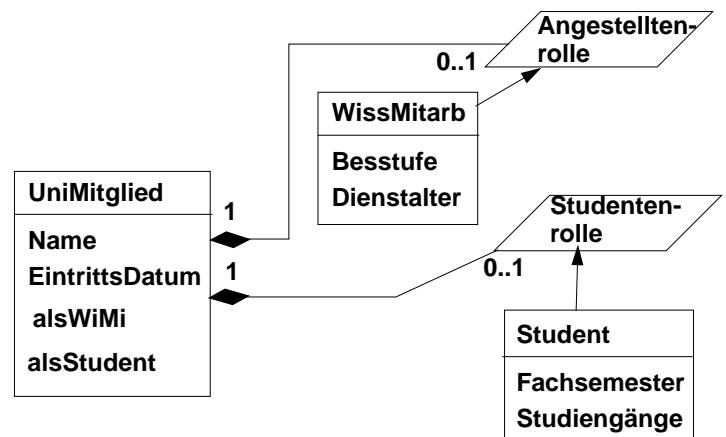
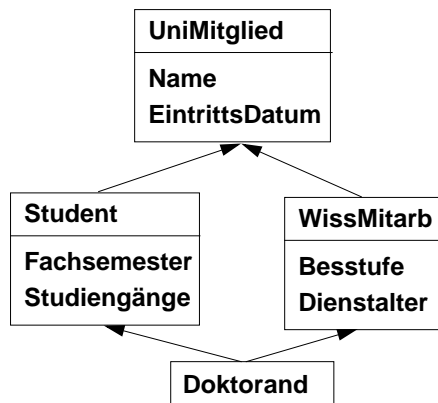
## 5.1.2 Januskopf

**Beschreibung:** Eine Klasse soll die Eigenschaften **mehrerer Oberklassen** vereinigen. Anwendung des Prinzips „**Abstraktion**“ (2.2) hat zu **überlappenden Unterklassen** geführt (2.16).

**Ursache:** evtl. **Rollenkonzept unbekannt**

**Abhilfe:** Abstraktionsebene aus der Klassenhierarchie entfernen  
**Komposition von Rollen**  
**Abstraktion der Rollen**

**Beispiel:**



**Doktorand** ist nun ein UniMitglied, das **zugleich beide Rollen** spielt: Student und WissMitarb.

## Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 503

### Ziele:

Schema einprägen

### in der Vorlesung:

- Erläuterungen dazu am Beispiel;
- Erinnerung an Abschnitt 2

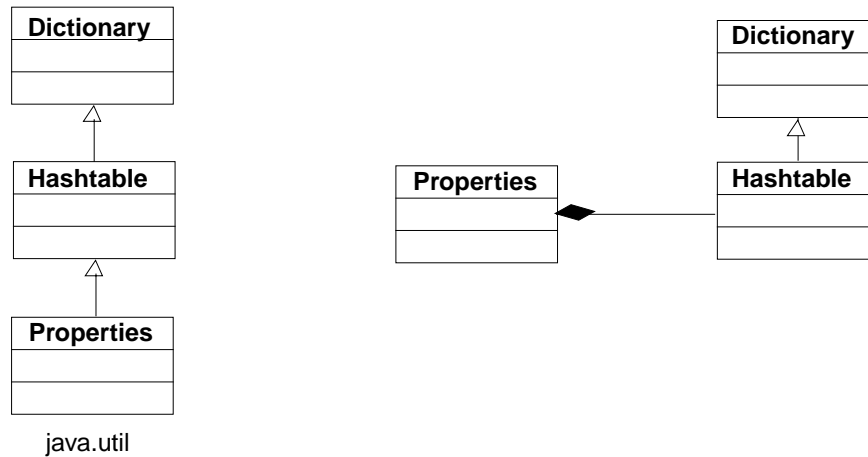
### 5.1.3 Ungeplanter Anbau

**Beschreibung:** Unterklasse nutzt die Funktionalität der Oberklasse, um ein **anderes Konzept** zu realisieren; schlechte **inkrementelle Weiterentwicklung** (Abschnitt 2.6)

**Ursache:** evtl. **Spezialisierung missverstanden**

**Abhilfe:** Funktionalität nicht durch Vererbung sondern durch **Delegation** nutzen.

**Beispiel:** siehe OOP-2.21



### Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 504

**Ziele:**

Schema einprägen

**in der Vorlesung:**

- Erläuterungen dazu am Beispiel;
- Erinnerung an Abschnitt 2

## 5.2 AntiPatterns

W.J. Brown, R.C. Malveau, H.W. McCormick III, T.J. Mowbray:  
**Anti Patterns, Refactoring Software, Architectures, and Projects in Crisis**,  
John Wiley, 1998.

**AntiPattern:** beschreibt ein **Schema von Lösungen** das häufig angewandt wird,  
aber **mehr Probleme verursacht als es löst**.

Anti Patterns für **Software-Entwicklung, Architektur, Projektorganisation**

### Darstellungsschema im Buch:

- Hintergrund
- Allgemeine Beschreibung
- Symptome und Konsequenzen
- Ausnahmen
- Verbesserte Lösung
- Varianten
- Beispiel
- Verwandte Lösungen

### Kurzform hier:

- Beschreibung
- Symptome
- Konsequenzen
- Verbesserte Lösung
- Beispiel

## Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 505

### Ziele:

Eindruck zu dem Buch bekommen

### in der Vorlesung:

- Ausrichtung und Struktur des Buches vorstellen,
- Begriff AntiPatterns als Kontrast zu Entwurfsmustern vorstellen
- AntiPatterns sind weniger konkret und technisch detailliert als Entwurfsmuster,
- Darstellungsschema erläutern

### nachlesen:

W.J. Brown, et. al.: AntiPatterns, John Wiley, 1998

## 5.2.1 The Blob

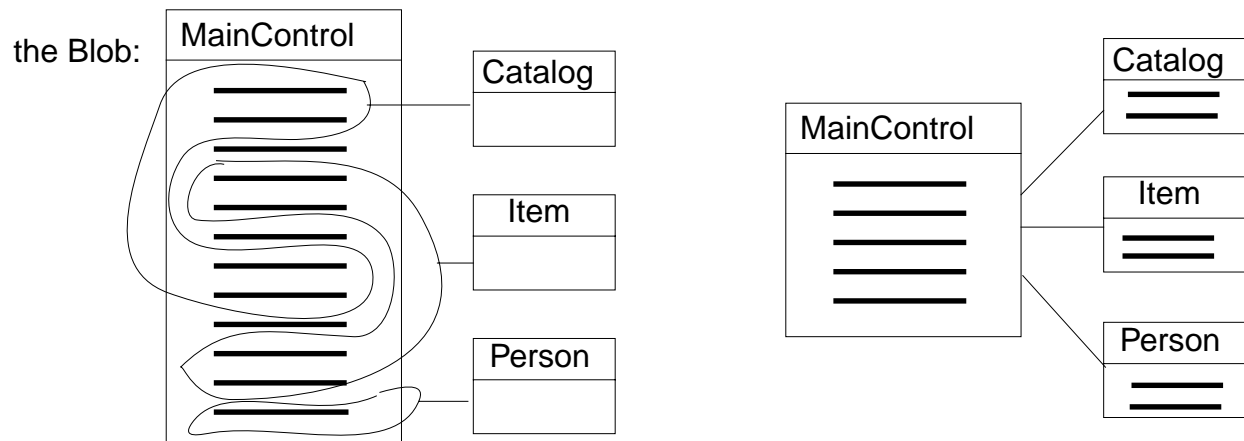
**Beschreibung:** Eine einzige **Klasse dominiert** den Ablauf.  
Darum herum sind **Datenklassen** angeordnet.  
Ablauf-orientierter Entwurf: **Daten von Operationen getrennt.**

**Symptome:** Eine **Klasse mit vielen Methoden und Attributen**,  
**geringe Kohärenz.**

**Ursache:** evtl. OO-Kenntnisse fehlen

**Konsequenzen:** **Schlecht wartbar**, änderbar; **nicht wiederverwendbar**

**Verbesserte Lösung: Zerlegen;** Operationen und zugehörige Daten **zusammenfassen.**



### Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 506

**Ziele:**

Schema einprägen

**in der Vorlesung:**

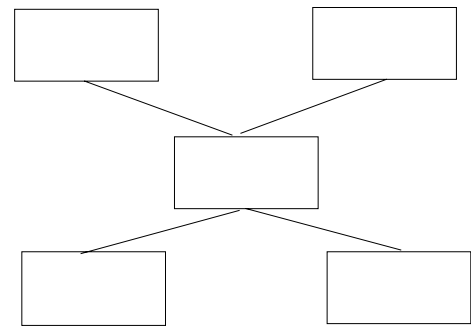
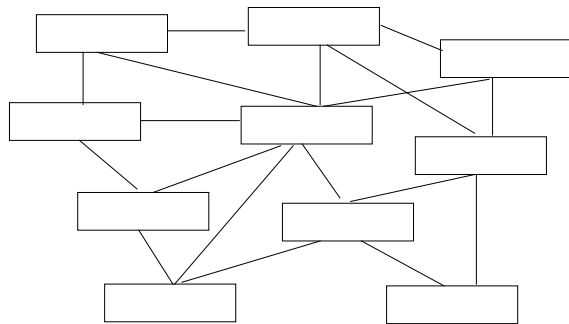
- Erläuterungen dazu am Beispiel;
- The Blob: Filmfigur, alles aufsaugendes Glibbermonster;

**nachlesen:**

W.J. Brown, et. al.: AntiPatterns, John Wiley, 1998

## 5.2.2 Poltergeister

- Beschreibung:** **Übermäßig viele Klassen** und Beziehungen dazwischen, **unnötige Klassen** mit **winzigen Aufgaben**, kurzlebige Objekte ohne Zustand; **schwache Abstraktionen**, unnötig komplexe Struktur.
- Symptome:** s.o.
- Ursache:** evtl. übertriebener Einsatz von Klassen; Anfängerfehler
- Konsequenzen:** Unnötige Klassen und Komplexität **stören das Verständnis**, **verschlechtern die Wartbarkeit**, Änderbarkeit.
- Verbesserte Lösung:** **Irrelevante Klassen eliminieren**; winzige Aufgaben zusammenfassen zu **kohärenten größeren Klassen**.



### Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 507

**Ziele:**

Schema einprägen

**in der Vorlesung:**

- Erläuterungen dazu am Beispiel;
- Poltergeister (Originalname aus dem Buch): spuken lärmend herum und stehen immer störend im Wege;

**nachlesen:**

W.J. Brown, et. al.: AntiPatterns, John Wiley, 1998

## 5.2.3 Funktionale Zerlegung

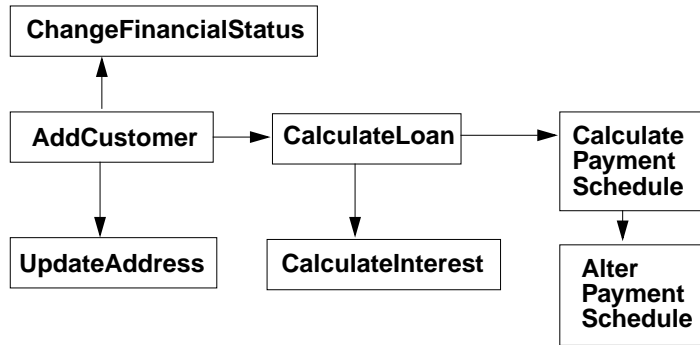
**Beschreibung:** Funktionen **schrittweise verfeinert**; eine Klasse für jede Funktion; OO-Techniken nicht angewandt.

**Symptome:** Klassen haben Funktionsnamen (z. B. CalculateInterest), nur eine Methode in jeder Klasse; **degenerierte Struktur**;

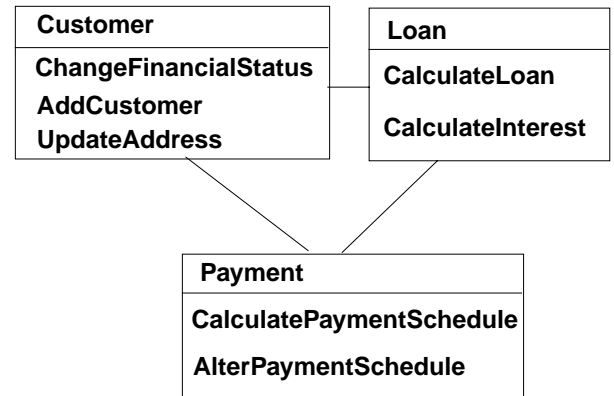
**Ursache:** evtl. OO-Kenntnisse fehlen, Programmierstil der 1970er Jahre

**Konsequenzen:** schwer zu verstehen und zu warten; keine Wiederverwendung.

**Verbesserte Lösung:** Mit OO-Konzepten neu entwerfen.



funktional



objektorientiert

### Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 508

**Ziele:**

Schema einprägen

**in der Vorlesung:**

- Erläuterungen dazu am Beispiel;

**nachlesen:**

W.J. Brown, et. al.: AntiPatterns, John Wiley, 1998



## 5.2.3 Spaghetti Code

Spaghetti Code: geprägt als Begriff zur Charakterisierung schlechter FORTRAN-Programme mit **verschlungenen Abläufen**, nur durch **Sprünge und Marken** programmiert.

- Beschreibung:** schwach strukturiertes System;  
Klassen mit **riesigen Methoden**,  
**viele Methoden ohne Parameter**.
- Symptome:** Methoden sind sehr **Ablauf-geprägt**,  
**sehr lange Methodenrumpfe**;  
**globale Variable statt Parameter**.
- Ursache:** evtl. Programmier- und OO-Kenntnisse fehlen
- Konsequenzen:** **schlecht wartbar**;  
**kein Nutzen von OO**, nicht wiederverwendbar.
- Verbesserte Lösung:** **Umstrukturieren**;  
besser: **nicht entstehen lassen**.

### Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 509

**Ziele:**

Schema einprägen

**in der Vorlesung:**

- Erläuterungen dazu am Beispiel;

**nachlesen:**

W.J. Brown, et. al.: AntiPatterns, John Wiley, 1998

## Kurzbeschreibungen einiger Anti Patterns

### 5.2.5 Lava Flow (ausgeflossen, alt und hart geworden)

**Beschreibung:** **Unkontrollierte Auswüchse und Anbauten** des Systems; veraltete, **schwierig zu entfernende Klassen**;

**Ursachen:** **verpasster Redesign**, mangelhafte Struktur

**Abhilfe:** **Redesign**

### 5.2.4 Ofenrohr-Struktur (zusammengeflickt und verrostet)

**Beschreibung:** **adhoc-Struktur**; alle Subsysteme sind **paarweise verbunden**; Verbindungen müssen **immer wieder repariert** werden.

**Abhilfe:** **Redesign**

### 5.2.6 Swiss Army Knife (universell verwendbar)

**Beschreibung:** Klasse mit **riesiger Schnittstelle**; große Zahl von Methoden, die **alle Möglichkeiten und Varianten** vorsehen sollen

**Abhilfe:** **Schnittstelle strukturieren**, reduzieren, **Schema (Profile) einführen zur Parametrisierung**

## Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 510

### Ziele:

Schlechte Struktureigenschaften einprägen

### in der Vorlesung:

- Erläuterung der AntiPatterns,
- Die Schemata sind recht allgemein.

### nachlesen:

W.J. Brown, et. al.: AntiPatterns, John Wiley, 1998

## 5.3 Üble Gerüche im Code - Signal zum Refaktorisieren

### Refaktorisieren:

**Ziel: Code verbessern**

- bessere Struktur,
- leichter verständlich,
- besser änderbar,
- besser korrigierbar

**Methode:**

Objektorientierten Code eines Software-Systems in kleinen Schritten systematisch verändern:

- Transformation aus Katalog anwenden (72 bei Fowler)
- beobachtbare Wirkung bleibt unverändert
- prüfen durch umfassende Tests (selbstprüfende Regressionstests)
- **üble Gerüche** (bad smells) als Hinweis auf schlechte Stellen im Code

**Herkunft des Refaktorisierens (Refactoring):**

Mitte 80er, Smalltalk-Community; Cunningham, Fowler, Beck, Opdyke

Martin Fowler: Refactoring, Addison-Wesley,

Kent Beck: eXtreme Programming, Addison-Wesley, 1999

## Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 510a

**Ziele:**

Grober Überblick zum Refactoring

**in der Vorlesung:**

Erläuterungen dazu

## Liste übler Gerüche (nach Fowler)

- |                                     |                                                          |
|-------------------------------------|----------------------------------------------------------|
| <b>1. Duplizierter Code</b>         | 12. Faule Klasse                                         |
| 2. Lange Methode                    | 13. Spekulative Allgemeinheit                            |
| 3. Große Klasse                     | 14. Temporäre Felder                                     |
| 4. Lange Parameterliste             | 15. Nachrichtenketten                                    |
| 5. Divergierende Änderungen         | 16. Vermittler                                           |
| 6. Schrotkugeln herausoperieren     | <b>17. Unangebrachte Intimität</b>                       |
| 7. Neid                             | 18. Alternative Klassen mit verschiedenen Schnittstellen |
| 8. Datenklumpen                     | 19. Unvollständige Bibliotheksklasse                     |
| 9. Neigung zu elementaren Typen     | <b>20. Datenklassen</b>                                  |
| <b>10. Switch-Anweisungen</b>       | <b>21. Ausgeschlagenes Erbe</b>                          |
| 11. Parallele Vererbungshierarchien | <b>22. Kommentare</b>                                    |

### Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 510b

**Ziele:**

Eindruck von der Art der Symptome bekommen

**in der Vorlesung:**

Sammlung erläutern und mit Anti-Patterns vergleichen

## Beispiele für üble Gerüche im Code (1)

### 1. Duplizierter Code:

An mehreren Stellen tritt die gleiche (ähnliche) Codestruktur auf.

Refaktorisierungen zur Abhilfe:

Methode extrahieren, Klasse extrahieren

### 10. switch-Anweisung

In mehreren switch-Anweisungen wird über denselben Wertebereich verzweigt; Typschlüssel; Änderungen betreffen alle Auftreten; nicht objektorientiert

Refaktorisierungen zur Abhilfe:

Methode extrahieren, Methode verschieben, Typschlüssel durch Unterklassen ersetzen, Bedingten Ausdruck durch Polymorphismus ersetzen

### 17. Unangebrachte Intimität:

Methode befasst sich zu intensiv mit den Daten und Methoden einer anderen Klasse

Refaktorisierungen zur Abhilfe:

Methode verschieben, Feld verschieben, Klasse extrahieren, Delegation verbergen

### 20. Datenklassen:

Klasse hat nichts außer Feldern und get- und set-Methoden dafür; wird von anderen Klassen aus manipuliert

Refaktorisierungen zur Abhilfe:

Methode verschieben, Methode extrahieren, set-Methode entfernen, Methode verbergen

## Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 510c

### Ziele:

"Geruch" und Abhilfe zuordnen

### in der Vorlesung:

Erläuterungen zu den Beispielen

- Symptome,
- Transformationen aus dem Refaktorisierungskatalog,

## Beispiele für üble Gerüche im Code (2)

### 21. Ausgeschlagenes Erbe:

a. Unterklasse verwendet kaum Methoden und Felder aus der Oberklasse

Refaktorisierungen zur Abhilfe:

Neue Geschwisterklasse bilden, Methode/Feld nach unten schieben

b. Unterklasse verwendet Methoden und Felder der Oberklasse, hält aber deren Schnittstelle nicht ein (Ersetzbarkeit verletzt, inkrementelle Weiterentwicklung)

Refaktorisierungen zur Abhilfe:

Vererbung durch Delegation ersetzen

### 22. Kommentare:

Kommentare als Deodorant gegen übel riechenden Code; signalisieren häufig schwer verständlichen Code

Refaktorisierungen zur Abhilfe:

Methode extrahieren, Methode umbenennen, Zusicherung einführen

Gute Kommentare an gutem Code begründen z. B. Entwurfsentscheidungen - hilfreich für spätere Wartung

## Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 510d

### Ziele:

"Geruch" und Abhilfe zuordnen

### in der Vorlesung:

Erläuterungen zu den Beispielen

- Symptome,
- Transformationen aus dem Refaktorisierungskatalog,
- Rolle von Kommentaren

## 5.4 Unangenehme OOP-Überraschungen

Für die **Software-Wartung** wünscht man sich, dass folgendes nicht eintreten kann:

**Zufügen einer Deklaration in einem Modul**  
ändert die **Wirkung** von Funktionen eines **anderen Moduls** -  
**ohne Warnung** durch den Übersetzer.

Man müsste sonst bei **Weiterentwicklungen und beim Testen**  
**immer das ganze System** betrachten.

**In nicht-OO-Sprachen weitgehend erfüllt.**

**In objektorientierte Sprachen:**

dynamische Methodenbindung und Überschreiben  
erzeugen den **o.g. Effekt - beabsichtigt.**

Die folgenden Beispiele zu o.g.Effekt stammen aus einem Vortrag über OOP & Maintenance von E. Plödereder, Universität Stuttgart.

### Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 511

**Ziele:**

Sicht der Software-Wartung

**in der Vorlesung:**

- Erläuterungen zu der Forderung

## Versehentlich überschrieben (1)

```
class C1
{
    void doA () {... if (...) this.doB(); ...}
    void doB () {...}
}

class C2 extends C1
{
    // doA geerbt
    // doB geerbt, dann überschrieben
    void doB () {... this.doA(); ...}
}
```

Nach Einfügen von `doB` in `C2` wird in `doA` nicht mehr `C1.doB` sondern `C2.doB` aufgerufen.

Der Aufruf von `C2.doB` terminiert dann u. U. nicht.

Nach einem Beispiel aus einem Vortrag über OOP & Maintenance von E. Plödereder, Universität Stuttgart.

## Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 512

### Ziele:

Problem erkennen

### in der Vorlesung:

Erläuterungen dazu



## Versehentlich überschrieben (2)

```
class C1
{ public void doA () {...}
}
```

```
class C2 extends C1
{
}
```

```
class C3 extends C2
{
}
```

----- Ende der Bibliothek

```
class C4 extends C2
{ public void doC () {...}
}
```

Nach einem Beispiel aus einem Vortrag über OOP & Maintenance von E. Plödereder, Universität Stuttgart.

## Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 513

### Ziele:

Problem der Bibliothekswartung

### in der Vorlesung:

- Erläuterungen dazu.
- Auf der nächsten Folie wird eine Änderung der Bibliothek gezeigt, die Probleme durch Überschreiben verursacht.

## Vorsehentlich überschrieben (2)

```
class C1
{ public void doA () {...}
}
```

```
class C2 extends C1
{
    protected void doC () {...}
}
```

Ergänzung der Bibliothek

```
class C3 extends C2
{
    void foo (C2 v) { ... v.doC(); ...}
}
```

Ergänzung der Bibliothek  
Aufruf führt in die

----- Ende der Bibliothek

schon vorher existierende

```
class C4 extends C2
{ public void doC () {...}
}
```

Benutzermethode  
C4.doC

Programmierer sollten explizit machen können, ob Überschreiben gemeint ist.  
Der Übersetzer dann prüfen und melden „**accidental overriding**“.

Nach einem Beispiel aus einem Vortrag über OOP & Maintenance von E. Plödereder, Universität Stuttgart.

## Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 513a

### Ziele:

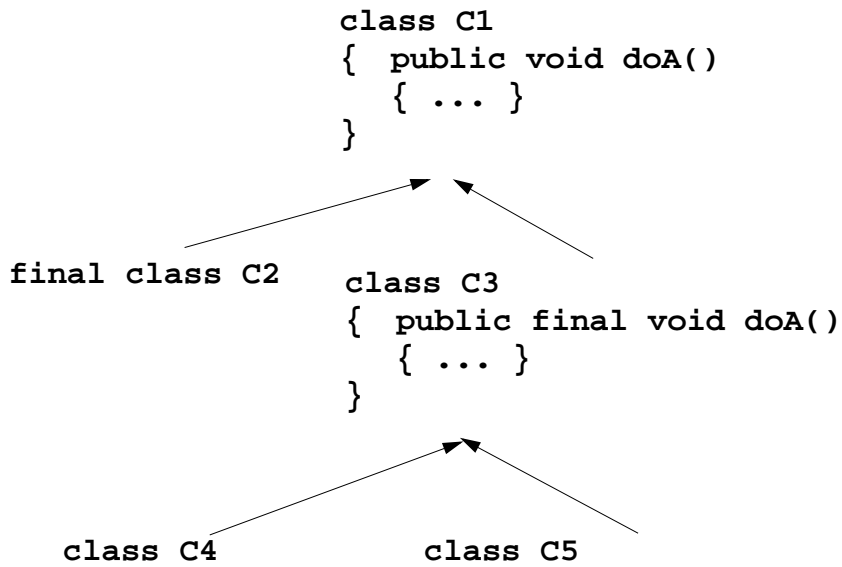
Problem der Bibliothekswartung

### in der Vorlesung:

- Erläuterungen dazu.
- Man müsste alle von der Bibliothek exportierten Namen kennen. Sie dürften sich nicht ändern.
- Wortsymbol für Überschreiben löst das Problem.

## final umgangen

In der Klassenhierarchie C1, C2, C3 sollte weiteres Überschreiben von doA verhindert werden:



Nach einem Beispiel aus einem Vortrag über OOP & Maintenance von E. Plödereder, Universität Stuttgart.

## Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 514

### Ziele:

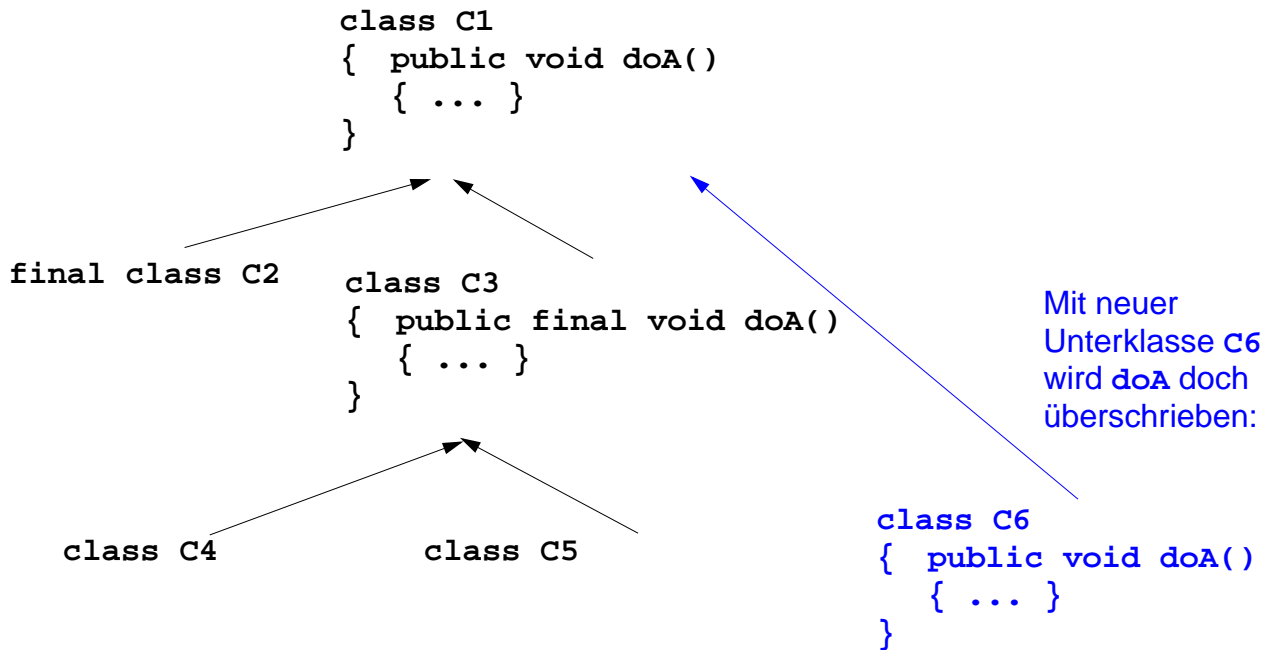
Problem erkennen

### in der Vorlesung:

- Erläuterungen dazu.
- Auf der nächsten Folie wird gezeigt, wie die Methode doch überschrieben werden kann.

## final umgangen

In der Klassenhierarchie C1, C2, C3 sollte weiteres Überschreiben von doA verhindert werden:



Nach einem Beispiel aus einem Vortrag über OOP & Maintenance von E. Plödereder, Universität Stuttgart.

## Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 514a

### Ziele:

Problem erkennen

### in der Vorlesung:

- Erläuterungen dazu.
- Man kann eine Klassenhierarchie so nicht gegen Überschreiben schützen.

# Überschreiben - Überladen

## vorher:

```
class C1
{ void doM (C1 x, T1 B);
  {...}
}

class C2 extends C1 {...}

class C3 extends C2
{ void doM (C1 x, T1 B);
  {...}
}

class C4 extends C3 {...}
```

C3.doM überschreibt C1.doM

## nachher:

```
class C1
{ void doM (C1 x, T2 B);
  {...}
}

class C2 extends C1 {...}

class C3 extends C2
{ void doM (C1 x, T1 B);
  {...}
}

class C4 extends C3 {...}
```

C3.doM überlädt C1.doM

Nach einem Beispiel aus einem Vortrag über OOP & Maintenance von E. Plödereder, Universität Stuttgart.

## Vorlesung Objektorientierte Programmierung WS 2012/2013 / Folie 515

### Ziele:

Problem erkennen

### in der Vorlesung:

- Erläuterungen dazu.
- In Sprachen, die Unter- oder Oberklassen als Parametertypen in der überschreibenden Signatur erlauben, ist das Problem noch schwieriger zu erkennen.