Vorlesung Fortgeschrittene Softwaretechnik

Wintersemester 2024/25

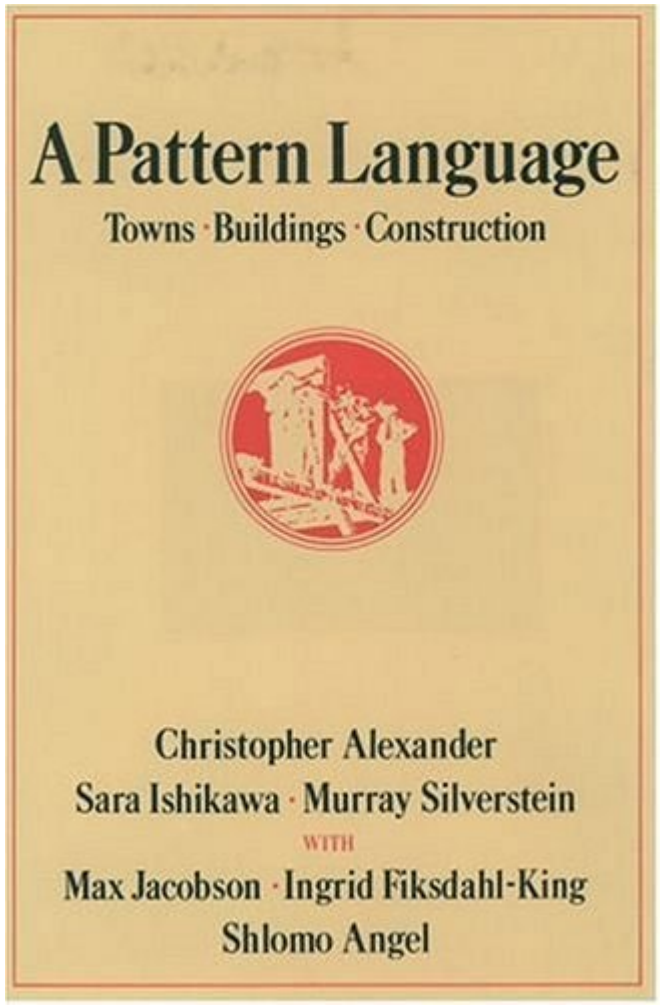
Prof. Dr. Stephan Diehl

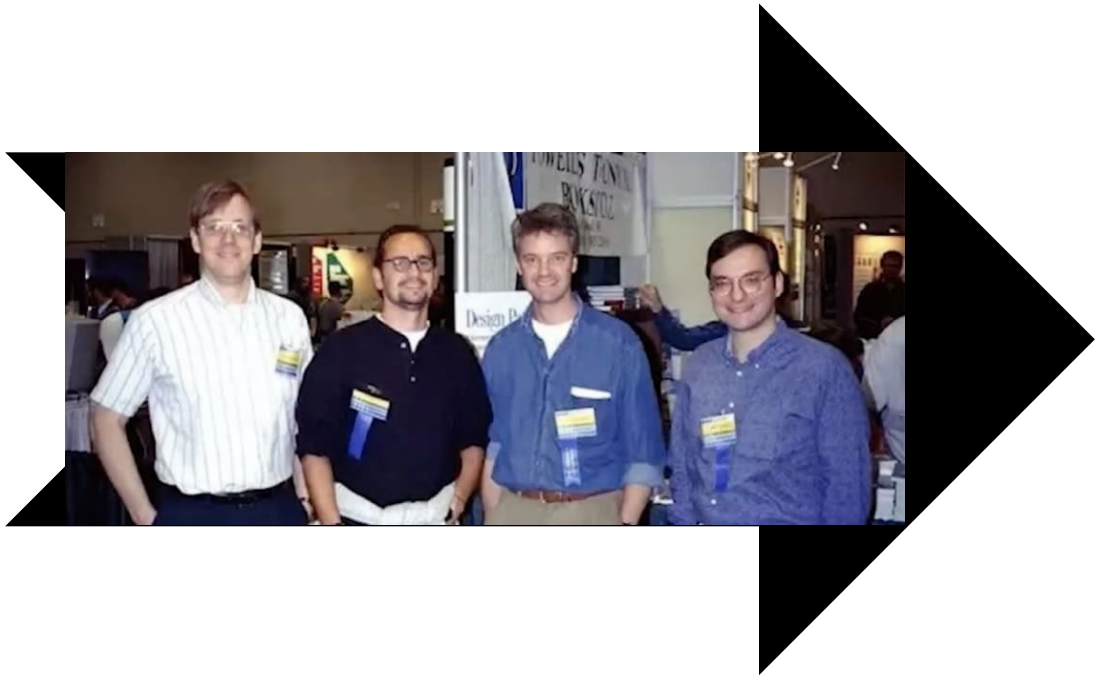
Informatik

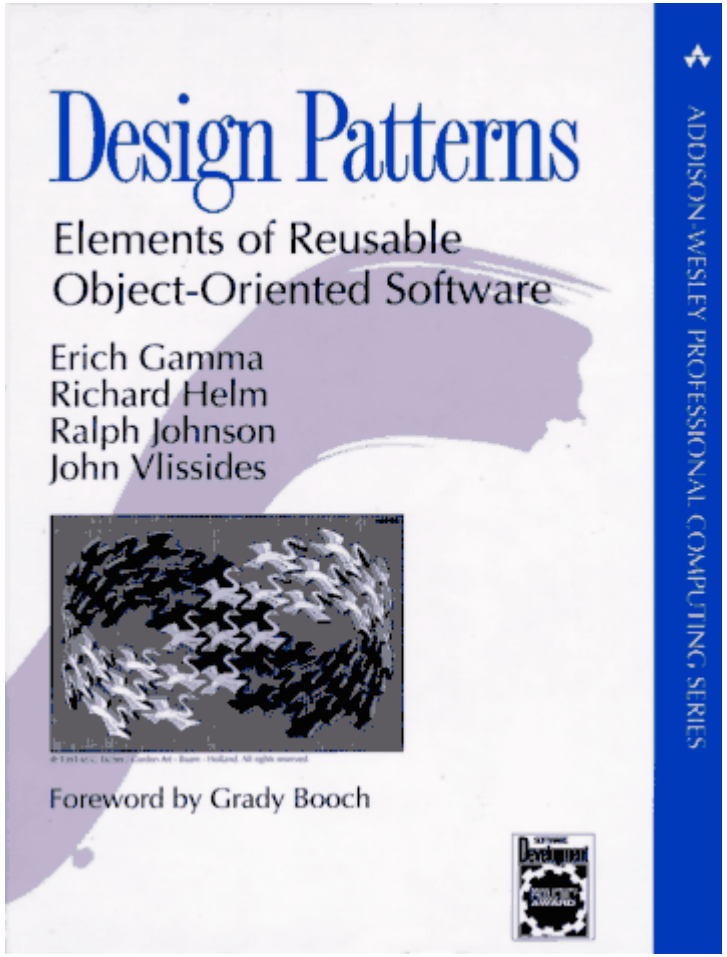
Universität Trier

Heute

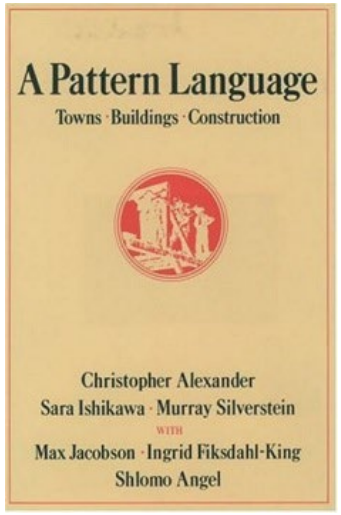
- Antipattern
  - „Gegenteil von Designpattern“
- Bad Smells
  - Schlechte Strukturen auf Code-Ebene


A Pattern Language
Towns · Buildings · Construction
Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

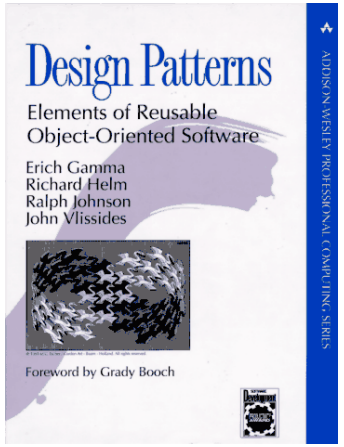



Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Entwurfsmuster (Design Patterns)



- Ursprüngliche Idee des Architekten Christopher Alexander (1977, Architekturmuster für Gebäude und Städtebau)



- Übertragen ins Software-Engineering:
  - "Design-Patterns - Elements of Reusable Object-Oriented Software„ von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides (GoF, "Gang of Four"), Addison-Wesley, 1995

- Entwurfsmuster = Designvorschlag für den Entwurf objektorientierter Softwaresysteme

Entwurfsmuster

„descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.“

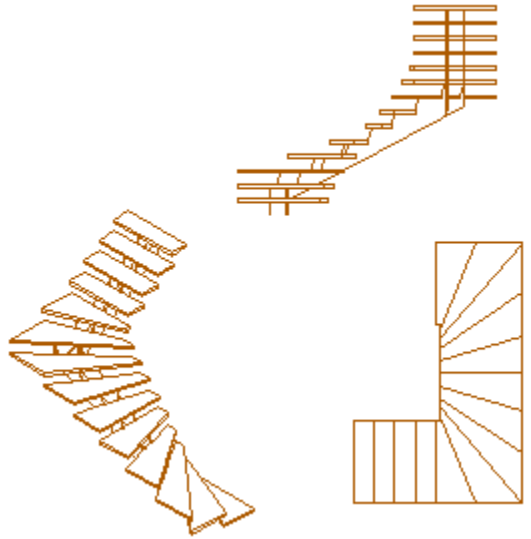
Entwurfsmuster

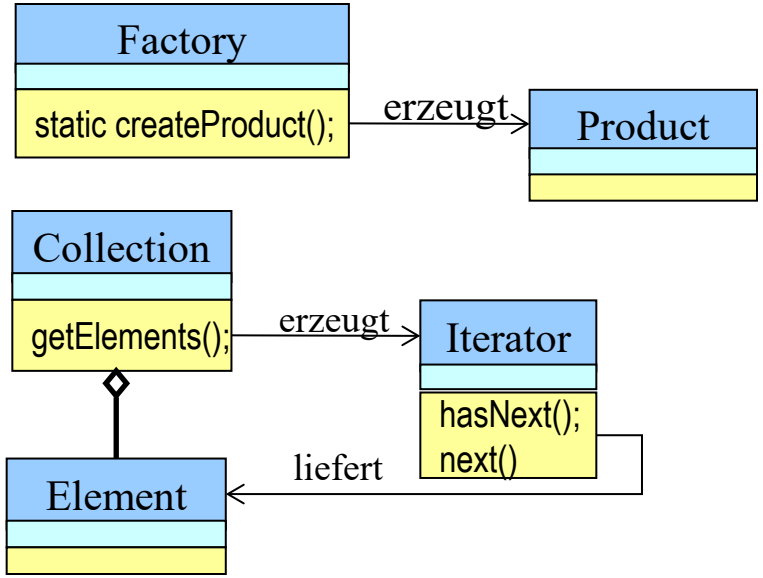
- Lösung auf hoher Abstraktionsebene für ein konkretes Entwurfsproblem
- beschreibt Zusammenwirken von Klassen, Objekten und Methoden.
- Meist mehrere Algorithmen und/oder Datenstrukturen beteiligt.

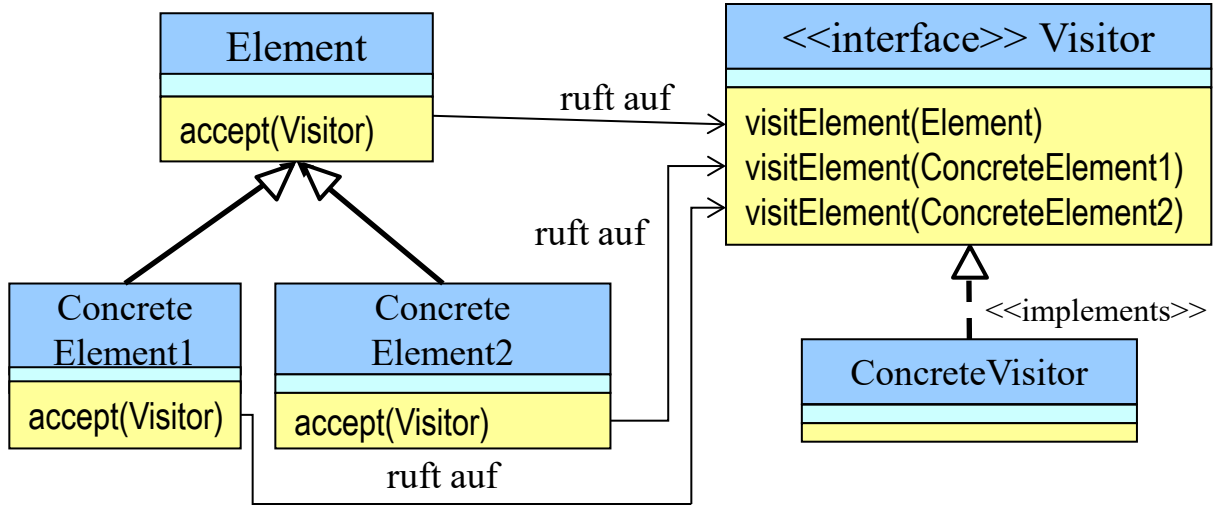
- Wichtigster Verdienst
  - standardisierte Namen für wiederkehrende Softwaredesigns
  - erleichtern Kommunikation in Entwicklungsteams

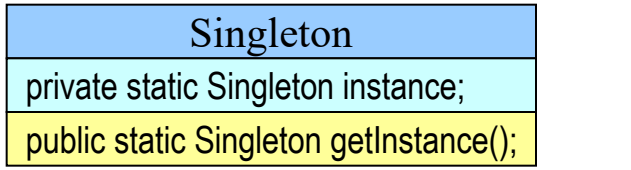
Entwurfsmuster aus dem Design-Pattern Buch

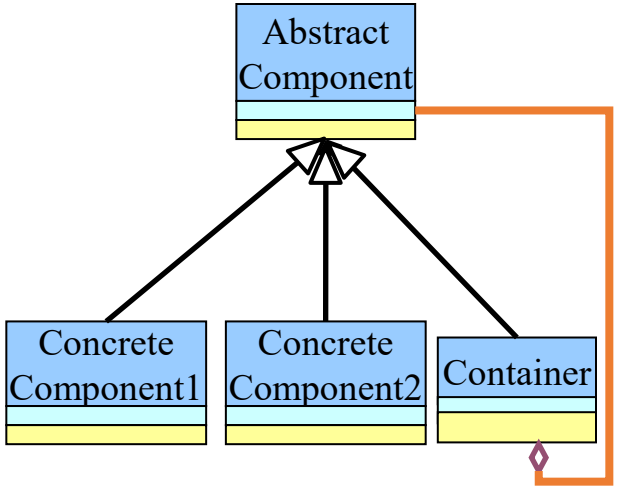
- **Abstract Factory**
- **Adapter**
- **Bridge**
- **Builder**
- **Chain of Responsibility**
- **Command**
- **Composite**
- **Decorator**
- **Facade**
- **Factory Method**
- **Flyweight**
- **Interpreter**
- **Iterator**
- **Mediator**
- **Memento**
- **Observer**
- **Prototype**
- **Proxy**
- **Singleton**
- **State**
- **Strategy**
- **Template Method**
- **Visitor**

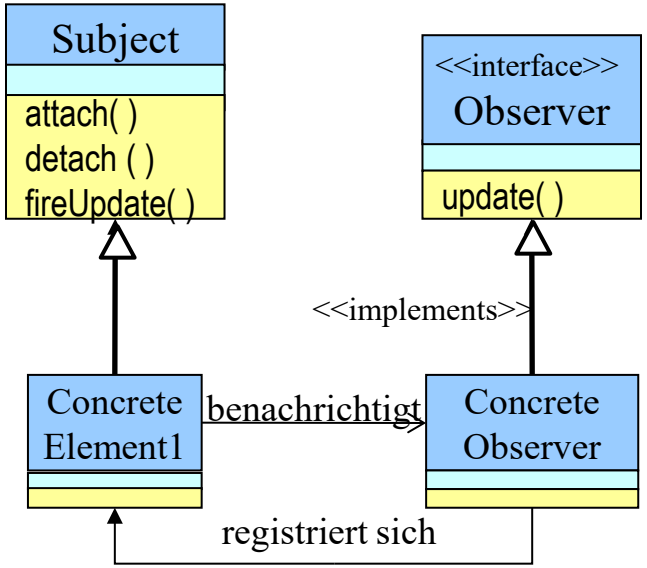


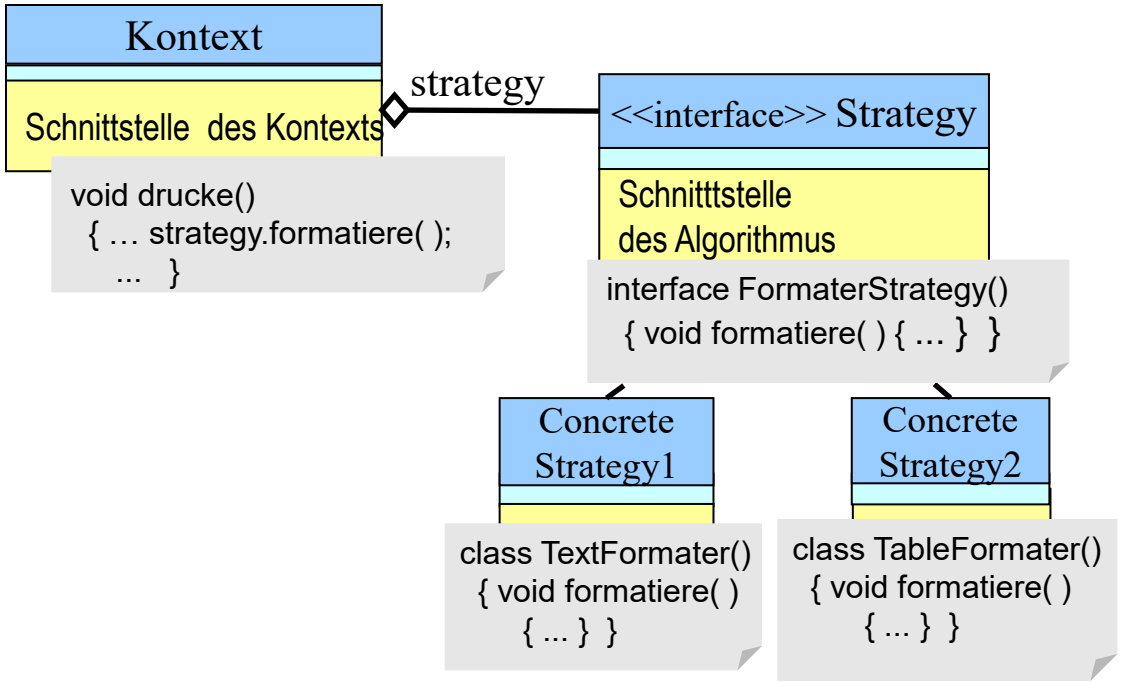

Factory
static createProduct();
erzeugt
Product
Collection
getElements();
erzeugt
Iterator
hasNext();
next()
liefert
Element


Element
accept(Visitor)
ruft auf
<<interface>> Visitor
visitElement(Element)
visitElement(ConcreteElement1)
visitElement(ConcreteElement2)
ruft auf
<<implements>>
ConcreteVisitor
Concrete Element1
accept(Visitor)
Concrete Element2
accept(Visitor)
ruft auf


Singleton
private static Singleton instance;
public static Singleton getInstance();

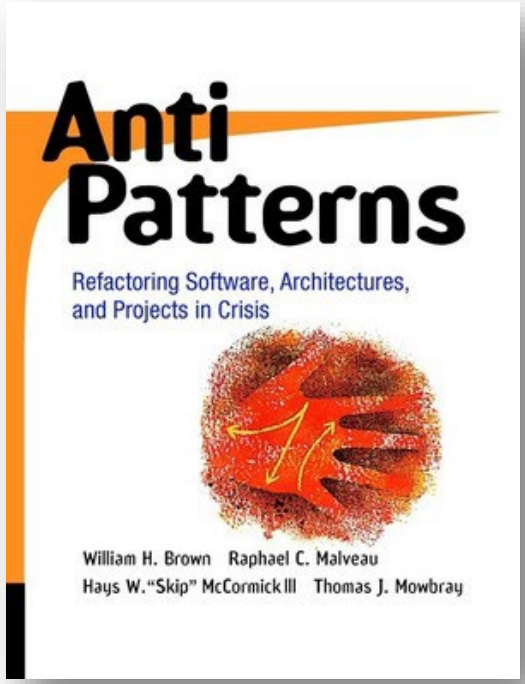

Abstract Component
Concrete Component1
Concrete Component2
Container


Subject
attach( )
detach ( )
fireUpdate( )
<<interface>>
Observer
update( )
<<implements>>
Concrete Element1
benachrichtigt
Concrete Observer
registriert sich


Kontext
Schnittstelle des Kontexts
strategy
<<interface>> Strategy
Schnitttstelle
des Algorithmus
void drucke()
{ … strategy.formatiere( );
... }
interface FormaterStrategy()
{ void formatiere( ) { … } }
Concrete Strategy1
Concrete Strategy2
class TextFormater()
{ void formatiere( )
{ ... } }
class TableFormater()
{ void formatiere( )
{ ... } }

Antipatterns

- Muster für schlechtes Softwaredesign
  - auf Architekturebene
- erste Sammlung: Anti Patterns (Brown et al.,1998)



Ziel von „Patterns“ besteht darin, „Best Practices“ festzuhalten und wieder zu verwenden.

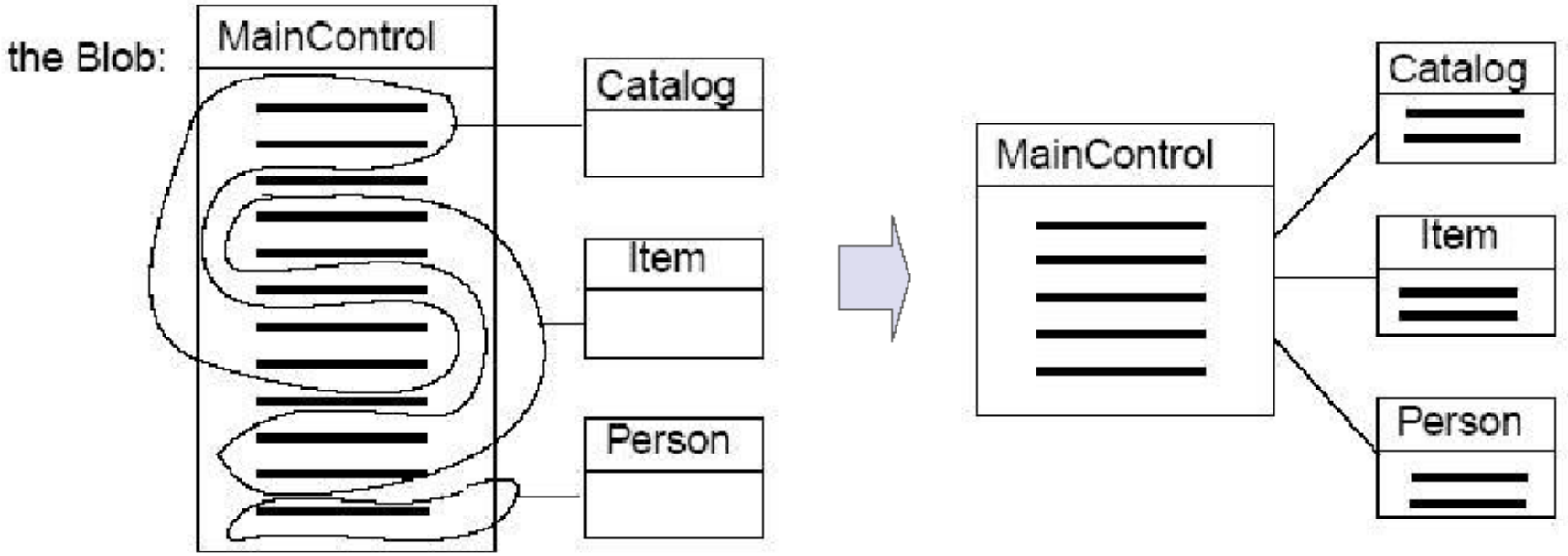
Antipattern: *The Blob*

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

Symptome: Große Klasse mit vielen Methoden und Attributen
Geringe Kohärenz.

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

Refactoring: **Zerlegen**; Operationen und zugehörige Daten **zusammenfassen**.



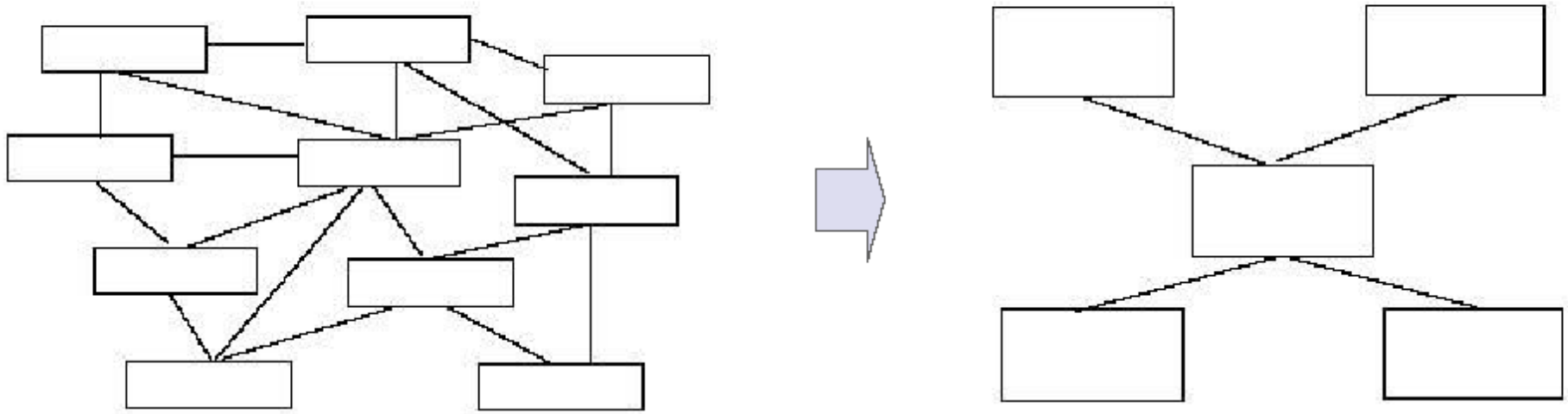
Antipattern: *Poltergeist*

Beschreibung: **zu stark verfeinert (schwache Abstraktionen)**
sehr viele Klassen mit Beziehungen dazwischen,
unnötige Klassen mit **winzigen Aufgaben**,
unnötig komplexe Struktur.

Symptome: s.o.

Konsequenzen: schwer zu verstehen (Unnötige Klassen mit Verbindungen)
schwer zu ändern (Verbindungen)

Refactoring: kleine Aufgaben zusammenfassen



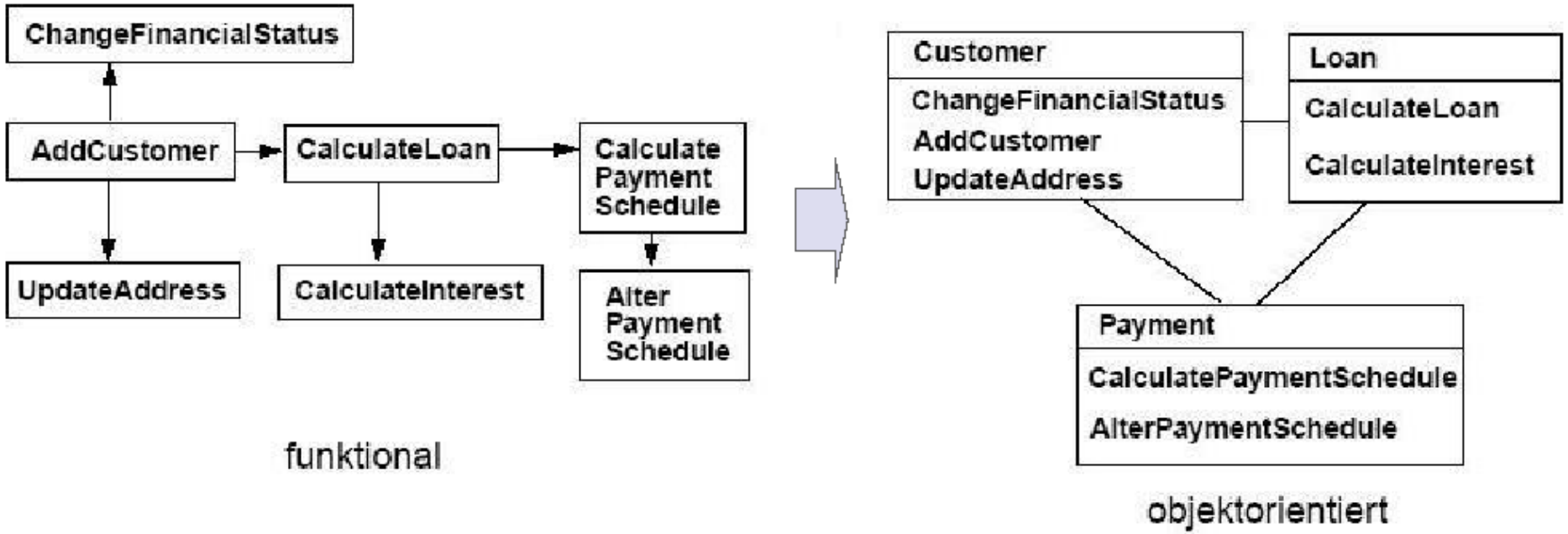
Antipattern: *Funktionale Zerlegung*

Beschreibung: Abfolge von Funktionen (schrittweise verfeinert)
Programmiertechnik aus prozeduralen Sprachen
auf OO-Sprache übertragen → **jede Funktion in eine Klasse**

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

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

Refactoring: Mit OO-Konzepten neu entwerfen.



Bad Smells (Liste nach Fowler)

Bad Smell = schlechte Struktur auf Code-Ebene

1. Duplizierter Code
2. Lange Methode
3. Große Klasse
4. Lange Parameterliste
5. Divergierende Änderungen
6. Schrotkugeln herausoperieren
7. Neid (Feature Envy)
8. Datenklumpen
9. Neigung zu elementaren Typen
10. Switch-Anweisungen

12. Parallel Kurzbeschreibung
13. Spekulative Allgemeinheit
14. Temporäre Felder
15. Nachrichtenketten
16. Vermittler
17. Unangebrachte Intimität
18. Alternative Klassen mit verschiedenen Schnittstellen
19. Unvollständige Bibliotheksklasse
20. Datenklassen
21. Ausgeschlagenes Erbe
22. Kommentare

Neid vs Unangebrachte Intimität

https://stackoverflow.com/questions/23311833/what-is-the-difference-between-inappropriate-intimacy-and-feature-envy

- “Inappropriate Intimacy means compromising the other class's encapsulation, such as by **directly accessing instance variables** that aren't meant to be directly accessed. Very bad. Fix the grabby class to only use public features of the compromised class and, if possible, change the compromised class so that other classes can't get at its private features.
- Feature Envy is when a method **uses more public features** of another class than it does of its own. Not as bad, because (assuming the other class's public features are safe to use) it won't lead to bugs. But it does lead to design entangling between the two classes. Fix by adding higher-level (better abstracted) public features to the envied class, or moving methods from the envious class to the envied class, so that the envious class has less methods to call.”

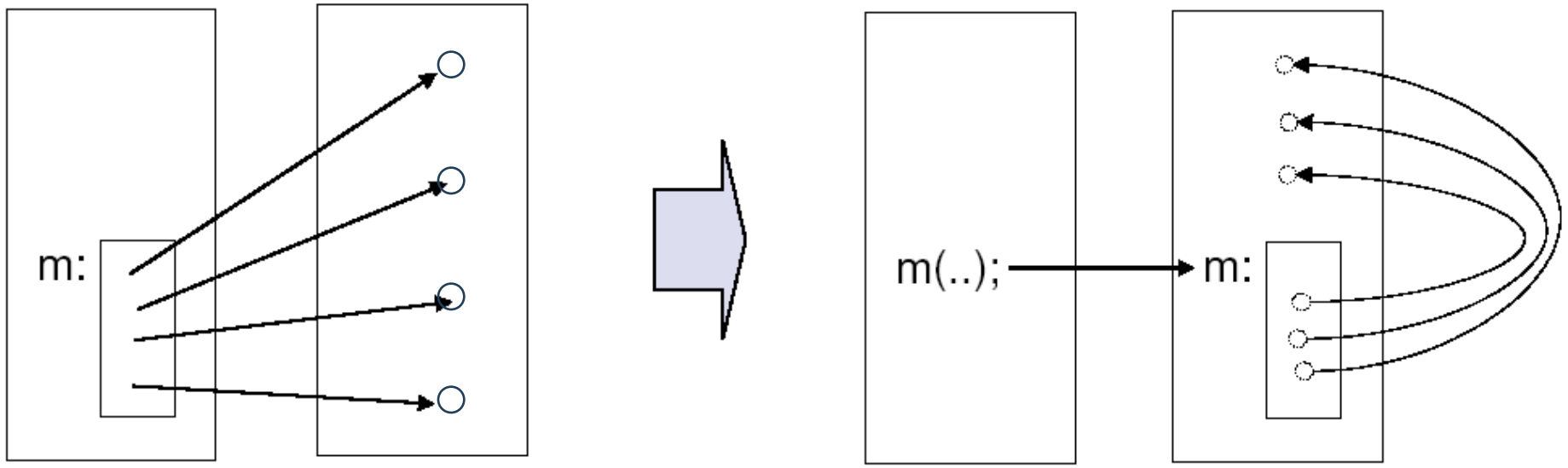
Bad Smell: *Unange-brachte Intimität*

Bad Smell:

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

Refactoring:

Methode verschieben



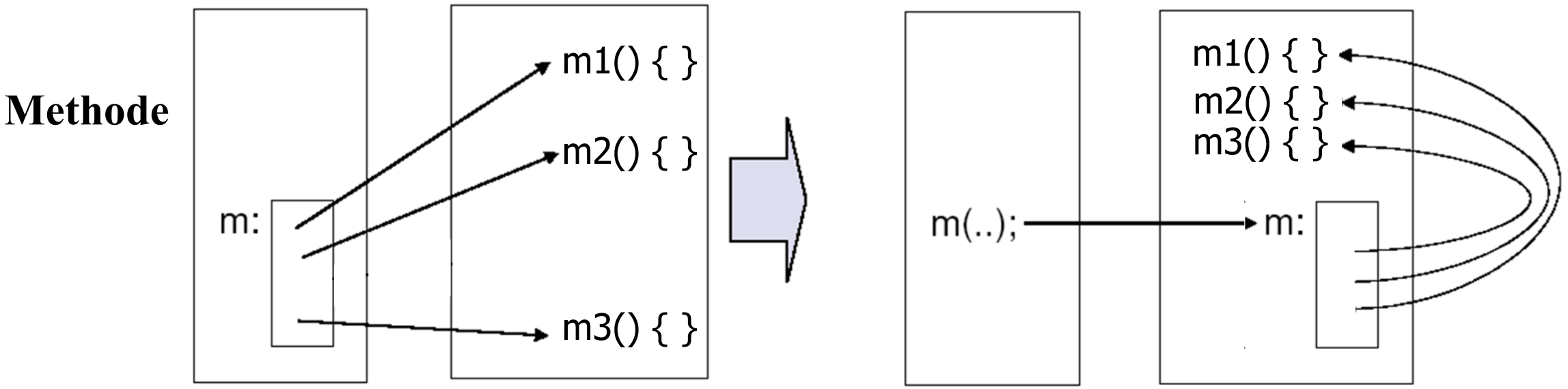
Bad Smell: *Feature Neid*

Bad Smell:

Methode befasst sich mehr mit den öffentlichen Daten und Methoden einer anderen Klasse als mit den eigenen.

Refactoring:

Methode verschieben



Bad Smell: *Duplizierter Code*

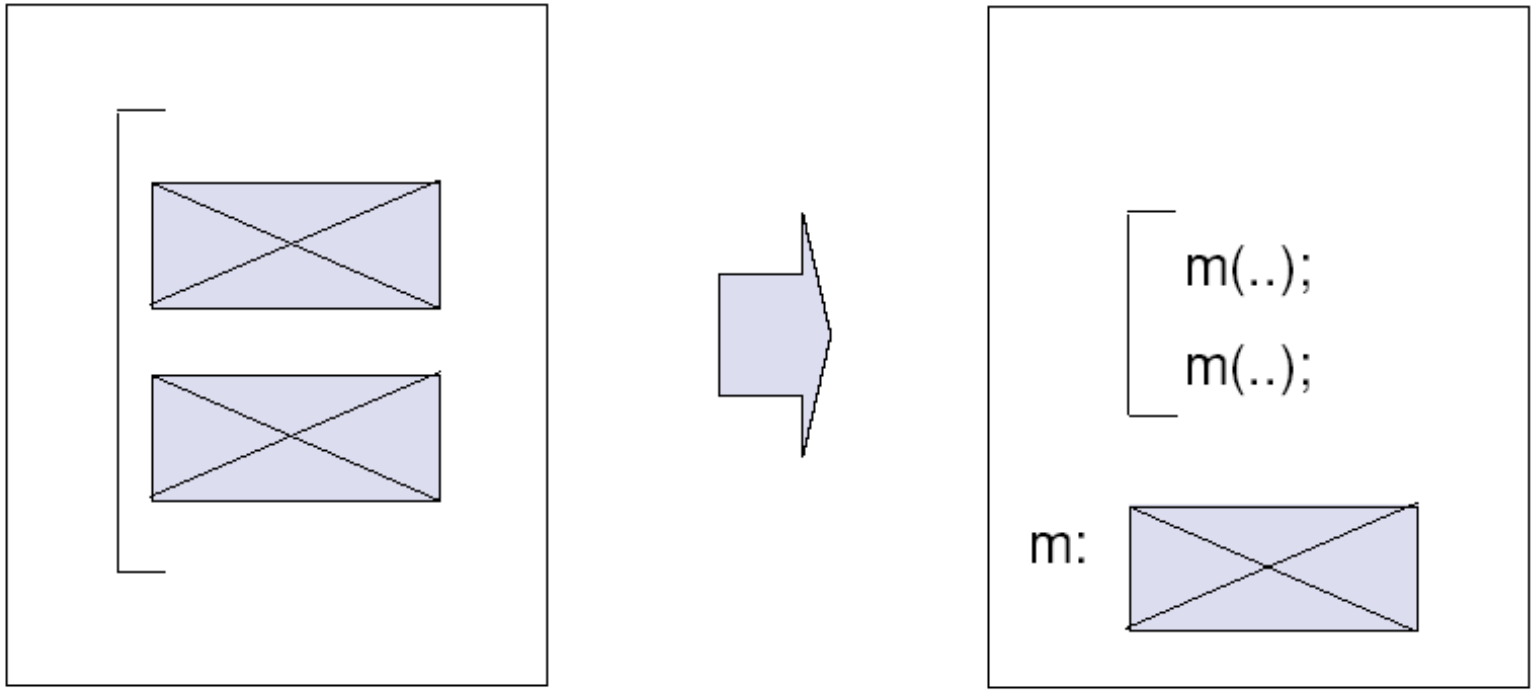
Bad Smell:

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

Problem: Änderungen müssen wiederholt werden

Refactoring:

Struktur in Methode verpacken



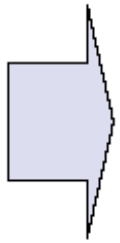
Bad Smell: *Switch-Anweisungen*

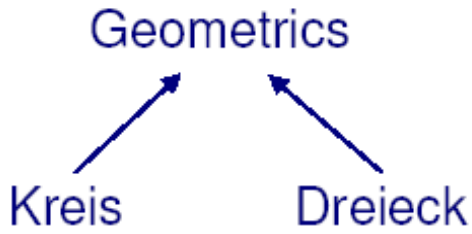
Bad Smell:
In **mehreren** switch-Anweisungen wird über denselben Wertebereich verzweigt; Typschlüssel; Änderungen betreffen alle Auftreten; nicht objektorientiert

Refactoring:
Typschlüssel durch Unterklassen ersetzen, Bedingten Ausdruck durch Polymorphismus ersetzen

```
float Fläche (..)
        switch (figur)
                case Kreis: …
                case Dreieck: …

float Umfang (..)
        switch (figur)
                case Kreis: …
                case Dreieck: …
```



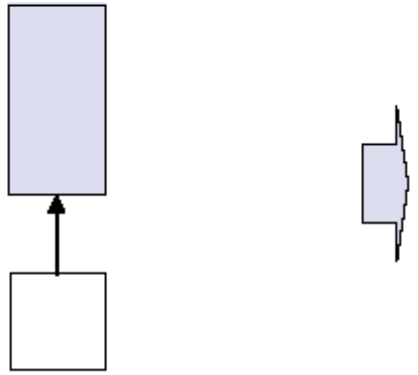



```
Geometrics(..)
    float fläche();
    float Umfang();
```

Bad Smell: *Ausge-schlagenes Erbe*

Bad Smell (Typ A):

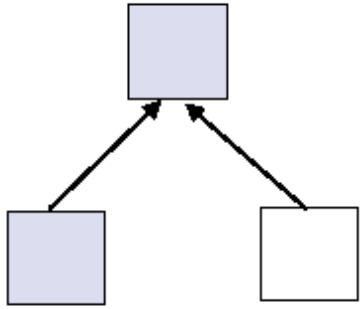
Unterklasse verwendet kaum Methoden und Variablen aus der Oberklasse



Refactoring (Typ A):

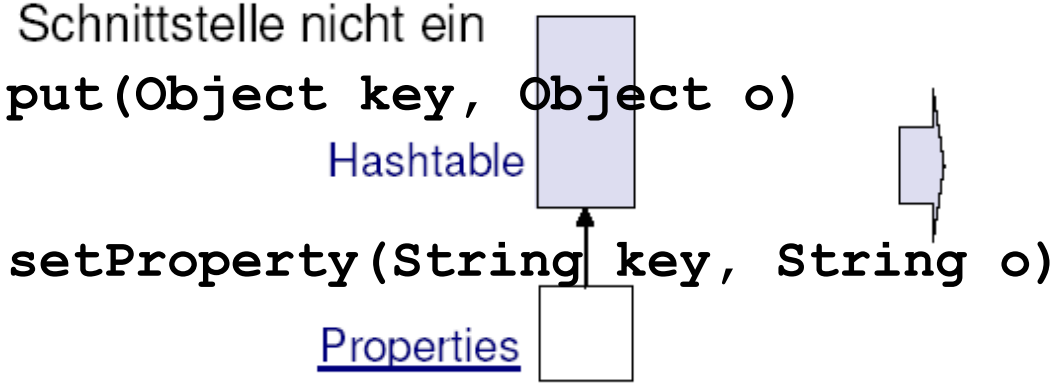
Oberklasse teilen

nicht genutztes nach unten verschieben



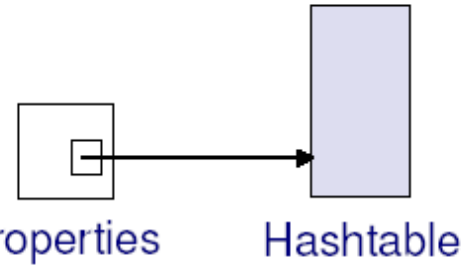
Bad Smell (Typ B):

Unterklasse verwendet Methoden und Var. der Oberklasse, hält aber deren Schnittstelle nicht ein



Refactoring (Typ B):

Vererbung durch Delegation ersetzen



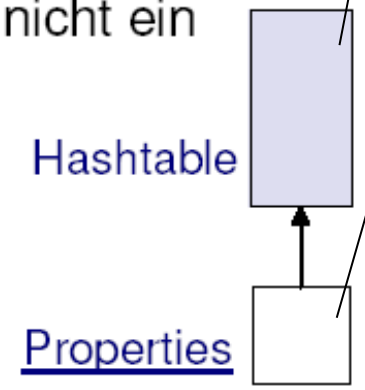
Bad Smell: *Ausge-schlagenes Erbe*

Erlaubt beliebige Objekte als Schlüssel und Wert

Schlüssel und Wert müssen Strings sein

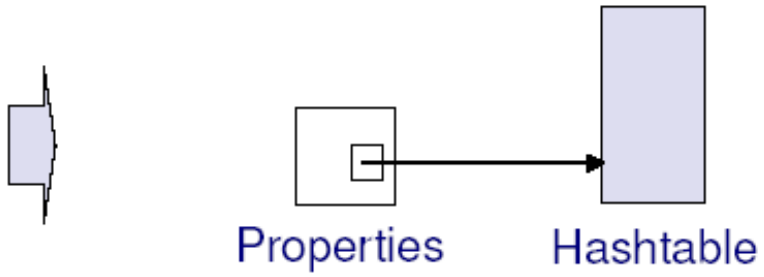
Bad Smell (Typ B):

Unterklasse verwendet Methoden und Var. der Oberklasse, hält aber deren Schnittstelle nicht ein



Refactoring (Typ B):

Vererbung durch Delegation ersetzen

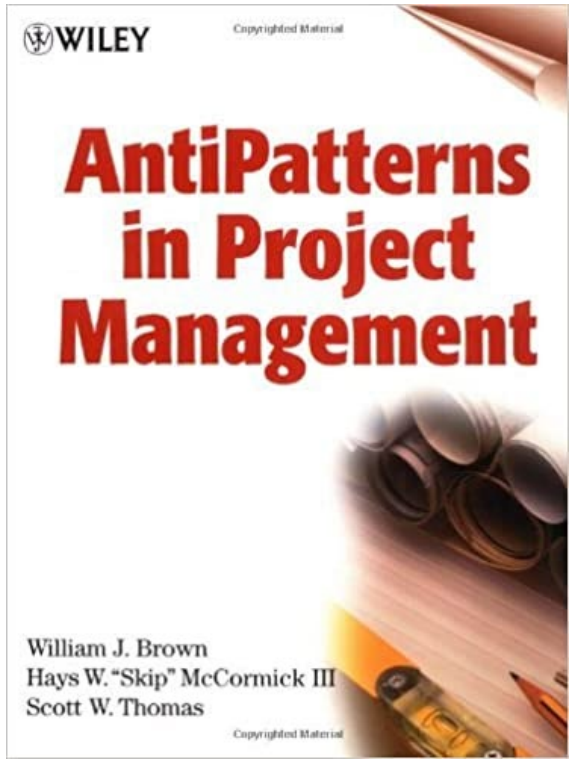


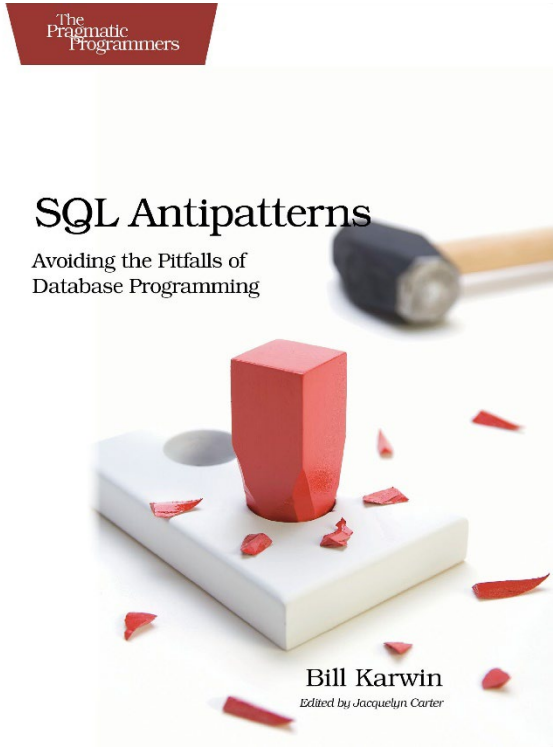
Ausgeschlagenes Erbe in java.util

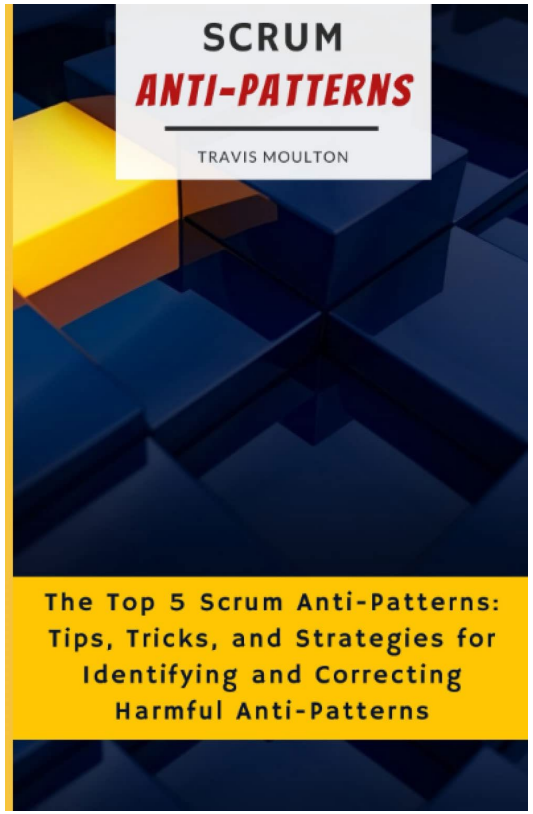
- Each key and its corresponding value in the property list is a string:
  - `public class Properties extends Hashtable<Object,Object>`
- Search for the property with the specified key in this property list:
  - `String getProperty(String key)`
- Beispiel:
  - `Properties p = new Properties();`
  - `p.setProperty("dings","bums");`
  - `System.out.println(p.getProperty("dings"));`
  - `p.put("dings", new Integer(1));`
  - `System.out.println(p.getProperty("dings"));`
  - `System.out.println(p.get("dings"));`

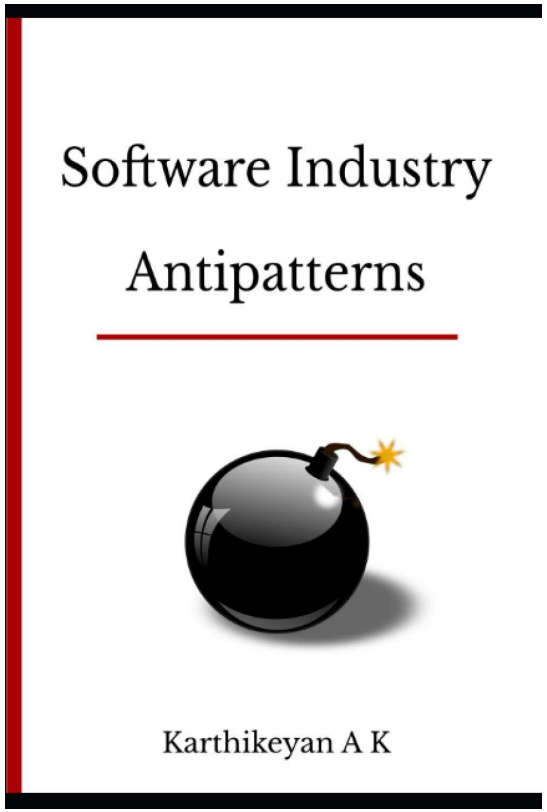
Ausgabe:
bums
null
1

AntiPatterns in anderen Bereichen der Softwareentwicklung









Projektmanagement-Antipattern: *Email is dangerous*

Beschreibung:

- E-Mail hat starke Bedeutung in betrieblicher Kommunikation
- Spektrum von Emails: von Scherzen bis geheime Daten
- E-Mail sind unverschlüsselt und sind nicht gesichert gegen Fälschung.

Konsequenzen

- Vertrauliche Nachricht gelangt zu unerwünschtem Empfänger (Boss, Konkurrenz)
- E-Mail kann permanent gespeichert und später gegen jemanden verwendet werden.
- E-Mail kann falsch interpretiert werden, viel stärker als bei persönlichem oder telefonischem Kontakt.
- Eine E-Mail kann an sehr viele Leute gleichzeitig weitergeleitet werden.

Lösung

- Generell E-Mail im betrieblichen Umfeld vermeiden. Emails nicht für folgende Themen verwenden:
  - Kritik
  - Vertrauliche Daten
  - Politisch Inkorrekte Themen
  - Strafbare Äußerungen
- Ausnahmen möglich, wenn die E-Mails verschlüsselt und signiert werden.

Übung:
https://github.com/paulkr/Flappy-Bird

Untersuchen Sie den Quellkode der „Flappy Bird“-Implementierung systematisch auf die in der Vorlesung (Folien) vorgestellten Anti-Patterns und Bad Smells:

- Anti-Patterns: The Blob, Poltergeist, Funktionale Zerlegung
- Bad Smells: Duplizierter Code, Unangebrachte Intimität, Switch-Anweisungen, Ausgeschlagenes Erbe

Pro Anti-Pattern und Bad Smell müssen Sie nur ein Beispiel mit kurzer Erläuterung angeben. Konnten Sie kein Beispiel finden, vermerken Sie dies entsprechend.