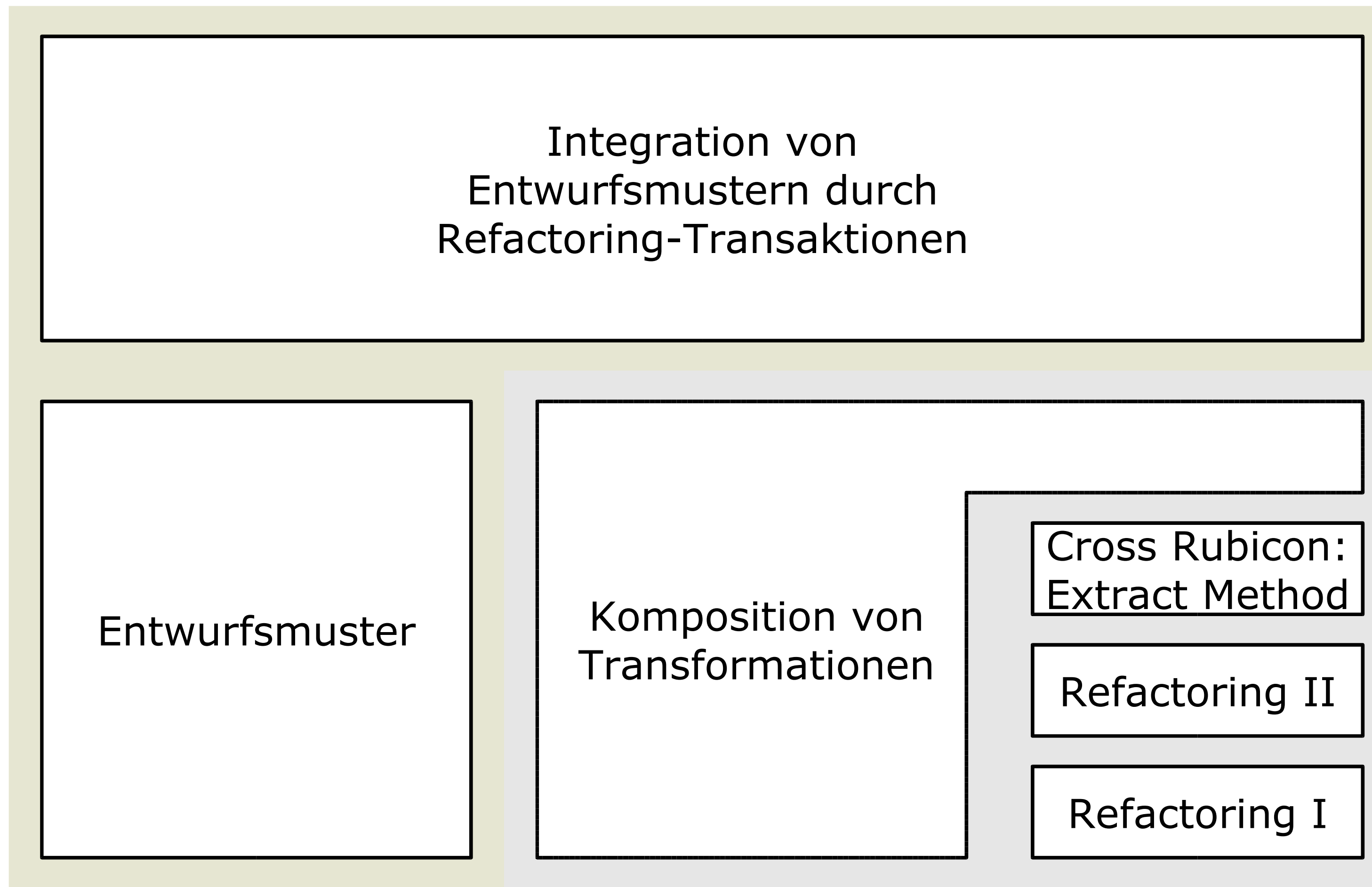
Integration von Entwurfsmustern durch Refactoring-Transaktionen

Seminar: Refactoring in eXtreme Programming

Alexander Bruder
anib@uni-paderborn.de

Einordnung

Vortrag im Kontext des Seminars



Motivation (technisch/wissenschaftlich)

Wir wollen:

- Entwurfsmustern verwenden

Wir brauchen:

- verhaltenskonservierende Reorganisation
- Automatisierte Transformationen

Wir bekommen:

- Entwicklung auf Entwurfsebene
- Vereinfachung von Softwareerweiterungen

Motivation (ökonomisch)

Unser Problem:

- Stichwort *Softwarekrise*

Wir wollen:

- Kosten reduzieren durch schnellere Entwicklung
- hohe Qualität abliefern

Die Lösung:

- Bekanntes und Erprobtes wiederverwenden
- Automatisierung

Inhalt

1. Entwurfsmuster
2. Refactoring-Transaktionen
3. Ansätze
4. Die Precursor-Methode
5. Factory Method integrieren
6. Fehlgeschlagene Integration (Facade)
7. Unterstützung durch Werkzeuge
8. Schlussfolgerung

1

Entwurfsmuster (Design Patterns)

erprobtes Wissen katalogisieren und wiederverwendbar machen

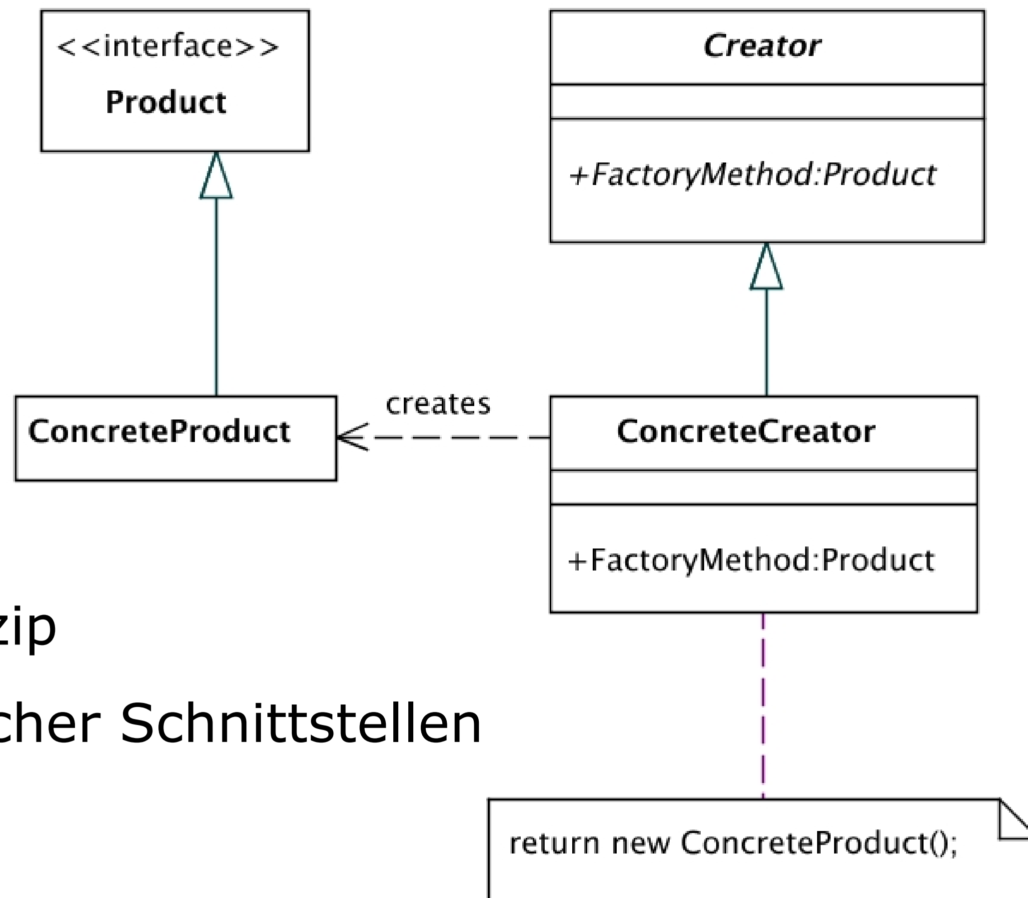
Standard ist Katalog von Gamma *et al* (GOF)

Entwurfsmusterarten:

- erzeugende Muster (Creational Patterns)
- Strukturmuster (Structural Patterns)
- Verhaltensmuster (Behavioral Patterns)

1 Entwurfsmuster (Design Patterns)

Beispiel: Factory Method



- Produkt-Erzeuger-Prinzip
- Bereitstellung einheitlicher Schnittstellen

2 Refactoring-Transaktionen

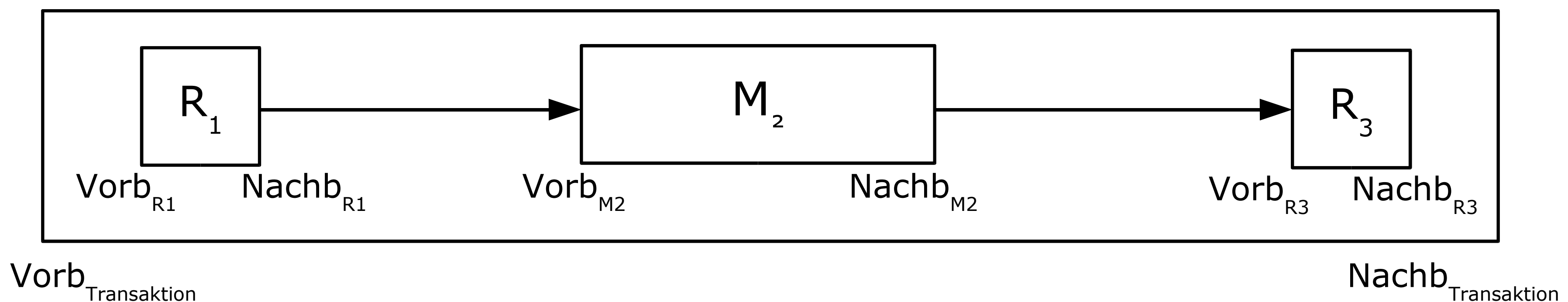
Entwurfsmuster existieren auf Entwurfsebene

-> Transformation benötigt mehrere Refactorings

basiert auf Sequenz von Operationen

müssen Vor- und Nachbedingungen einhalten

Bedingungen sind in Prädikatenlogik formuliert



2

Refactoring-Transaktionen Definition

Folge von Quellcodetransformationen als Einheit (atomar)

nur vor und nach Transaktion in zulässigen Zuständen

zu erfüllende Eigenschaften:

1. Atomarität
2. Konsistenz
3. Isolation
4. Dauerhaftigkeit

3 Ansätze

Ansätze

- Green Field Situation
- Antipatterns
- Precursor

Unterschiede

- Ausgangssituation
- Automatisierungsgrad

3 Green Field Situation

zu transformierende Komponenten stehen in keinem Zusammenhang

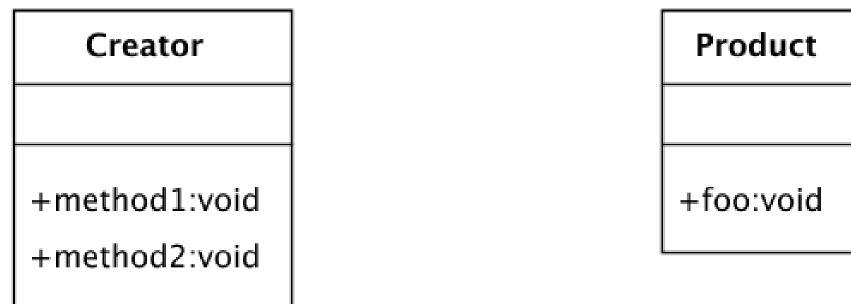
=> keine Abhängigkeiten

Programmanalyse entfällt zum Teil

Transformation kann fast immer automatisiert werden

Nachteil:

Realitätsfremde Ausgangssituation



3 Antipatterns

Annahme:

- Entwickler hat keine Kenntnis von Entwurfsmustern
- „unglückliche“ Lösung

Bisherige Realisierung ist:

- nicht korrekt
- schlecht
- kompliziert
- Entwurfsmuster an falscher Stelle

Transformation von „Schlecht“ in „Gut“ ist nicht generisch lösbar.

4 Die Precursor-Methode

Modell nach Mel Ó Cinnéide

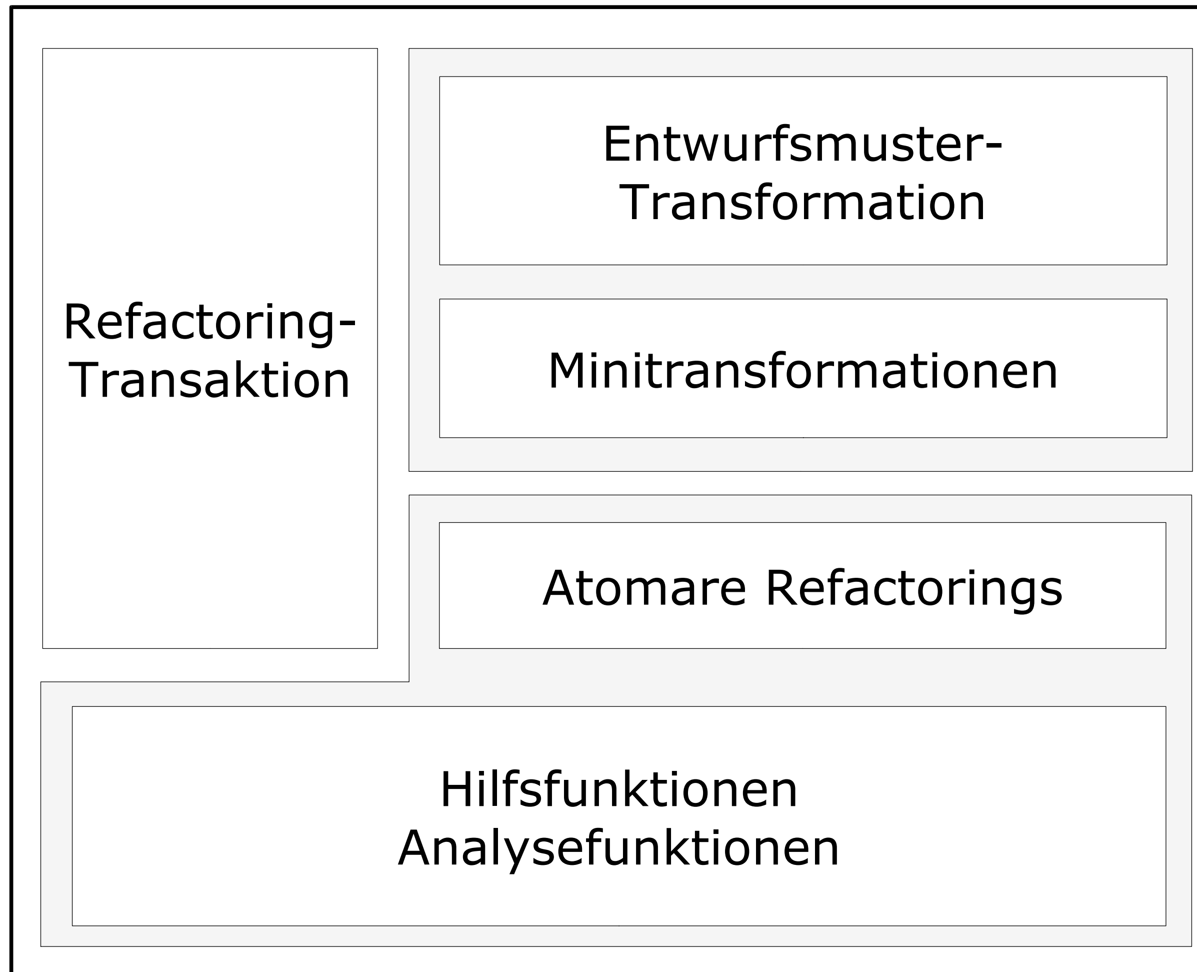
Precursor = Vorläufer, Vorgänger, Vorbote

Precursor

- Zentrales Element im Modell
- Intention hinter Entwurfsmuster
- Mittelweg zwischen *Green Field Situation* und *Antipatterns*
- Erkennung über Analysefunktionen

4 Aufbau des Modells

4-Schichten-Architektur



4

Analysefunktionen

- werden in Vor- und Nachbedingungen benutzt
- sind Prädikate der Prädikatenlogik erster Stufe
- im Modell als Operationen vorhanden
- extrahieren Eigenschaften des Quellcodes

Beispiel:

Boolean **isClass**(Class *c*)

4 Hilfsfunktionen

- Besitzen Vor- und Nachbedingungen
- Komplexer als Analysefunktionen

Beispiel:

Interface **abstractClass**(Class *c*, String *newName*)

Vorbedingung:

isClass(c)

Nachbedingung:

isInterface'=isInterface[inf/true]

nameOf'=nameOf[inf/newName]

equalInterface'=equalInterface[(c,inf)/true]

4 Atomare Refactorings

Grundtransformationen mit Vor- und Nachbedingungen

Beispiel:

void **addGetMethod**(Class *concrete*, String *fieldName*)

Vorbedingung:

$$\text{isClass}(\text{concrete}) \wedge \text{classOf}(\text{fieldName}) = \text{concrete}$$

$$\forall m : \text{Method}, \text{declares}(\text{concrete}, m) \wedge \text{nameOf}(m) \neq \text{"get"} + \text{fieldName}$$

Nachbedingung:

$$\exists m : \text{Method so dass}$$

$$\text{classOf}' = \text{classOf}[m/\text{concrete}]$$

$$\text{nameOf}' = \text{nameOf}[m/\text{"get"} + \text{fieldName}]$$

$$\text{returnsObject}' = \text{returnsObject}[m/\text{fieldName}]$$

4 Minipatterns/Minitransformation

Minipatterns:

wiederkehrende Muster in den Entwurfsmustern

Unterteilung erfolgt für Wiederverwendungszwecke

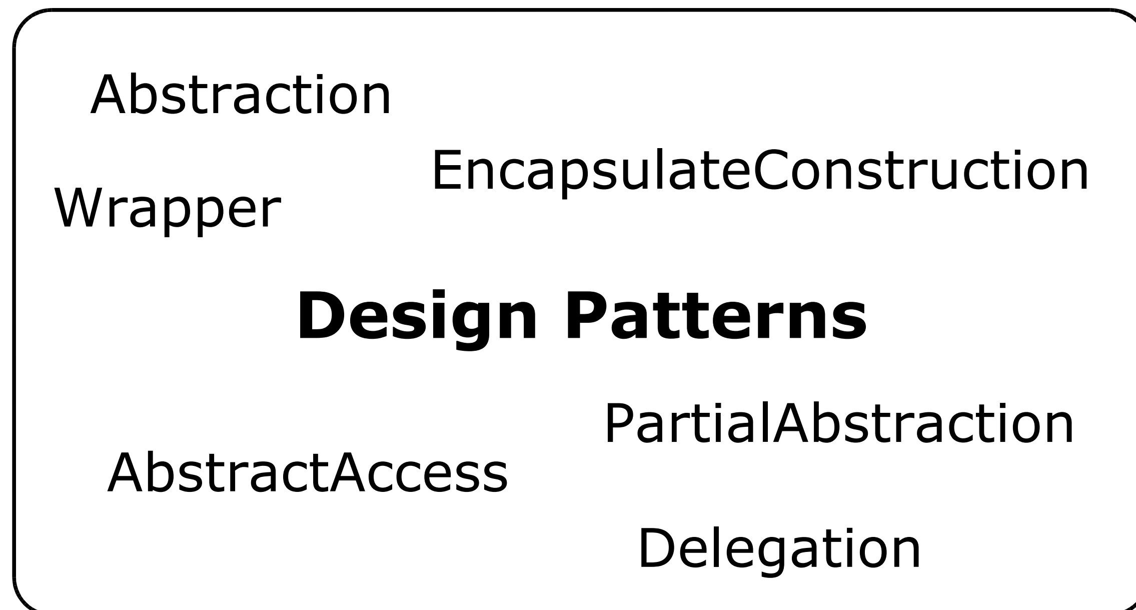
-> Schaffung einer Bibliothek

Minitransformation:

Algorithmus zur

Transformation

eines Minipatterns



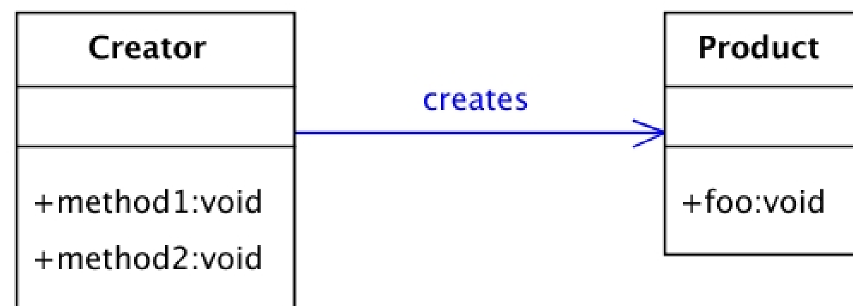
5 Factory Method integrieren

Precursor für Factory Method:

Hersteller muss Instanz von Produkt erstellen.

Erkennung über Analysefunktion

creates(creator, product)



5 Notwendige Minipatterns

1. *Abstraction*:

   Schnittstelle für Produkte

2. *EncapsulateConstruction*:

   Konstruktion in Methode kapseln und Objektreferenzen ersetzen

3. *AbstractAccess*:

   Produkt durch Interface ersetzen

4. *PartialAbstraction*:

   Hersteller abstrahieren

5 Abstraction

Abstrakter Zugriff auf *Product*

Zugriff über Schnittstelle

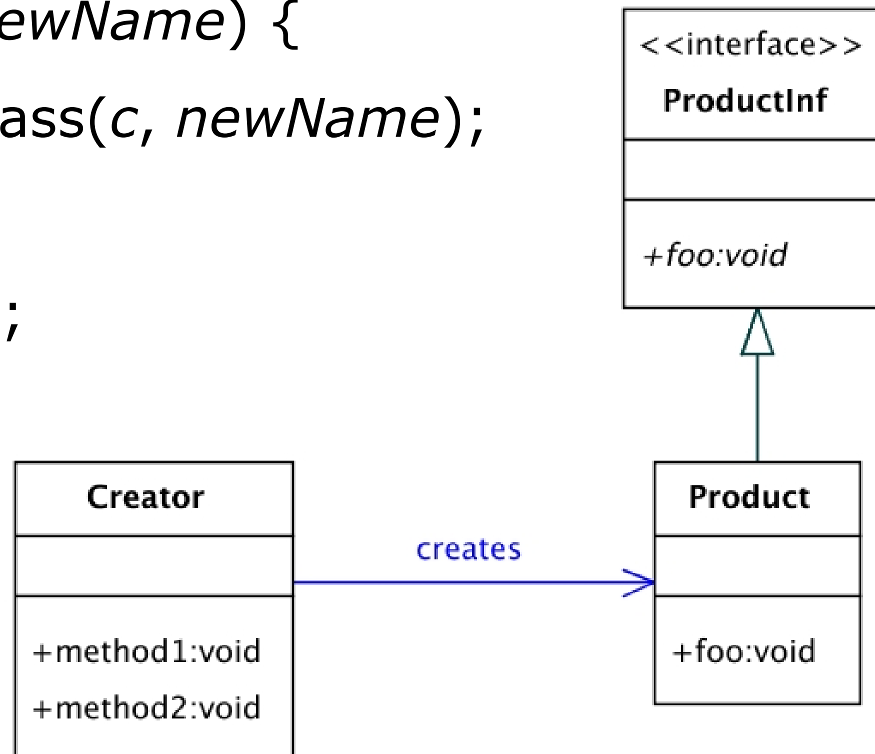
Abstraction(Class *c*, String *newName*) {

  Interface inf = abstractClass(*c*, *newName*);

  addInterface(inf);

  addImplementLink(*c*, inf);

}



5

Bedingungen für Abstraction

Vorbedingung

isClass(c)

$\neg$isClass(newName) $\wedge$ $\neg$isInterface(newName)

Nachbedingung

isInterface' = isInterface[inf/true]

equalInterface' = equalInterface[(c, inf) / true]

implementsInterface' = implementsInterface[(c, inf) / true]

nameOf' = nameOf[inf/newName]

5 EncapsulateConstruction

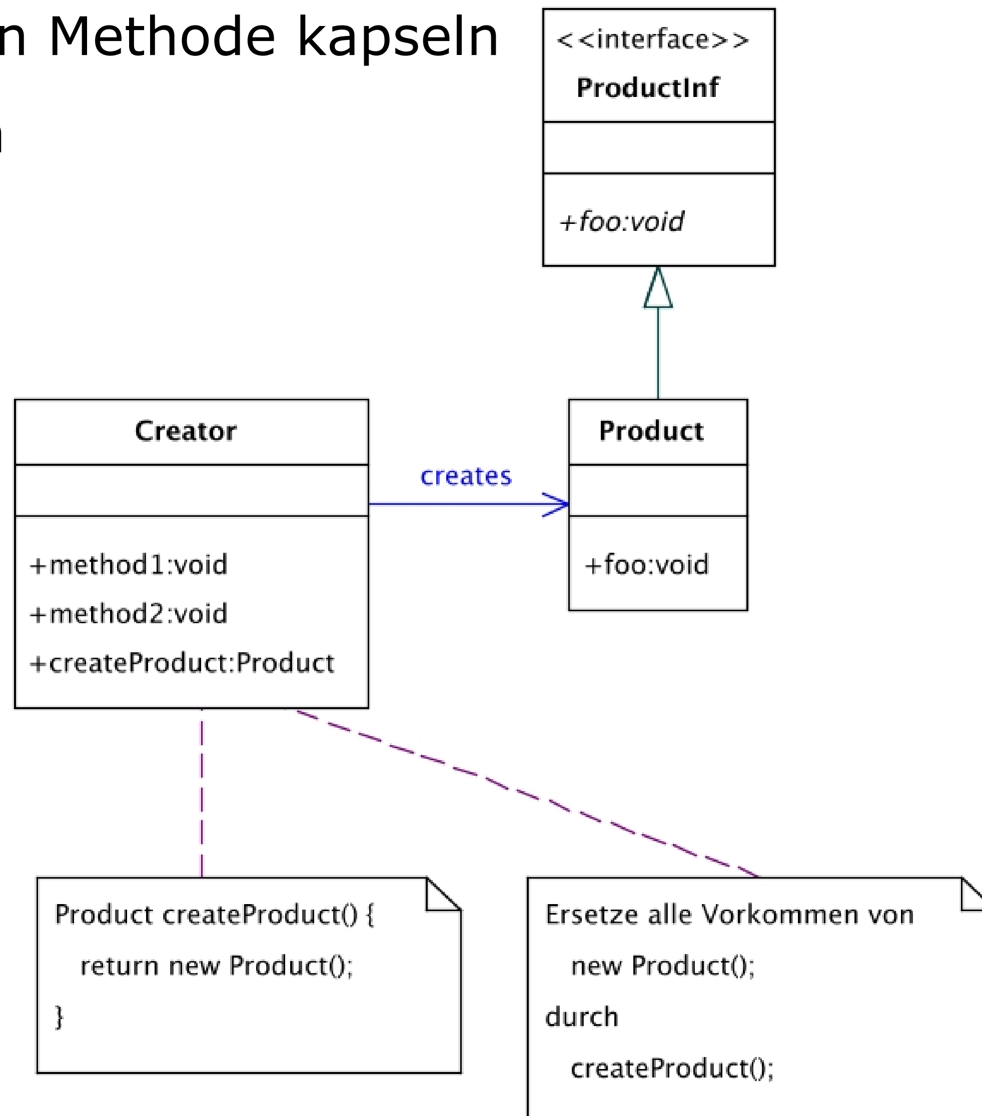
Erzeugung von Produkten in Methode kapseln

Objektreferenzen anpassen

alle Vorkommen von

new Product()

ersetzen



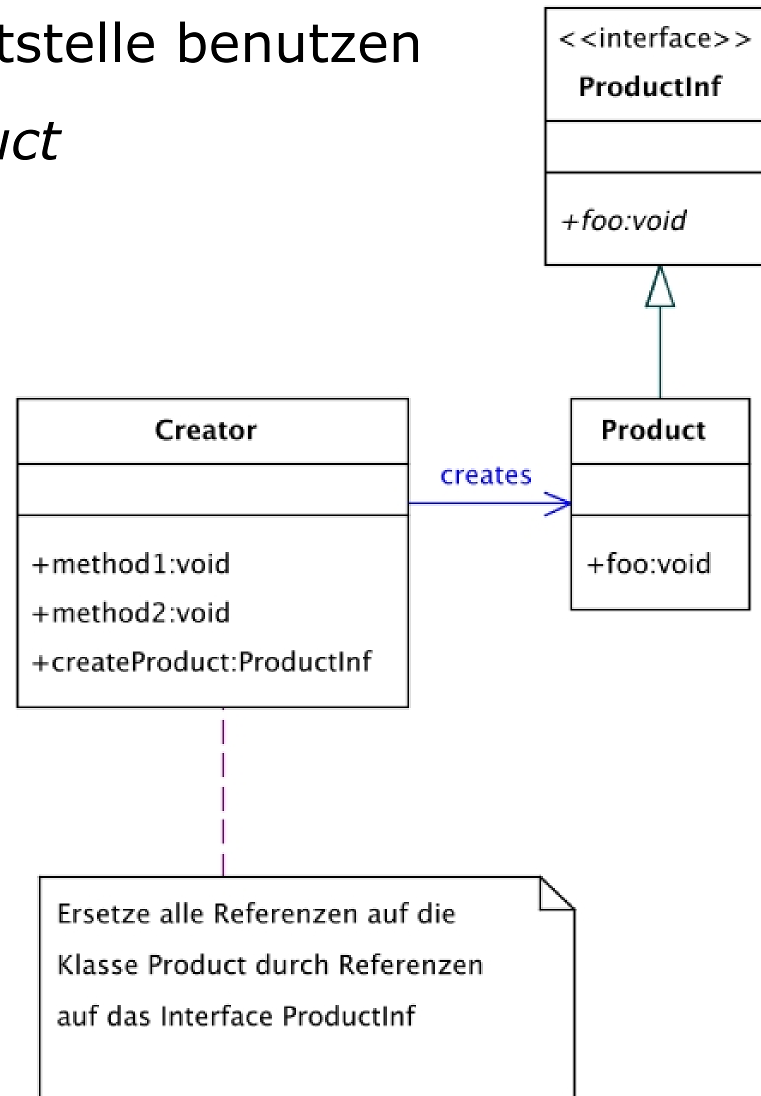
5 EncapsulateConstruction Transformation

```
EncapsulateConstruction(Class creator, Class product, String
 createP) {
    ForAll c:Constructor, classOf(c) = product {
        Method m = makeAbstract(c, createP);
        addMethod(creator, m);
    }
    ForAll e:ObjectCreationExprn, classCreated(e) = product ∧
        containingClass(e) = creator ∧
        nameOf(containingMethod(e)) ≠ createP {
        replaceObjectCreationWithMethInvocation(e, createP);
    }
}
```

5 AbstractAccess

Creator soll Produktschnittstelle benutzen

-> Entkopplung von *Product*



5 AbstractAccess Transformation

AbstractAccess(Class *context*, Class *concrete*, Interface *inf*,
SetOfString *skipMethods*) {

ForAll o:ObjectRef, typeOf(o) = *concrete*, containingClass(o) = *context*, nameOf(containingMethod(o)) ∍ *skipMethods* {

replaceClassWithInterface(o, *inf*);

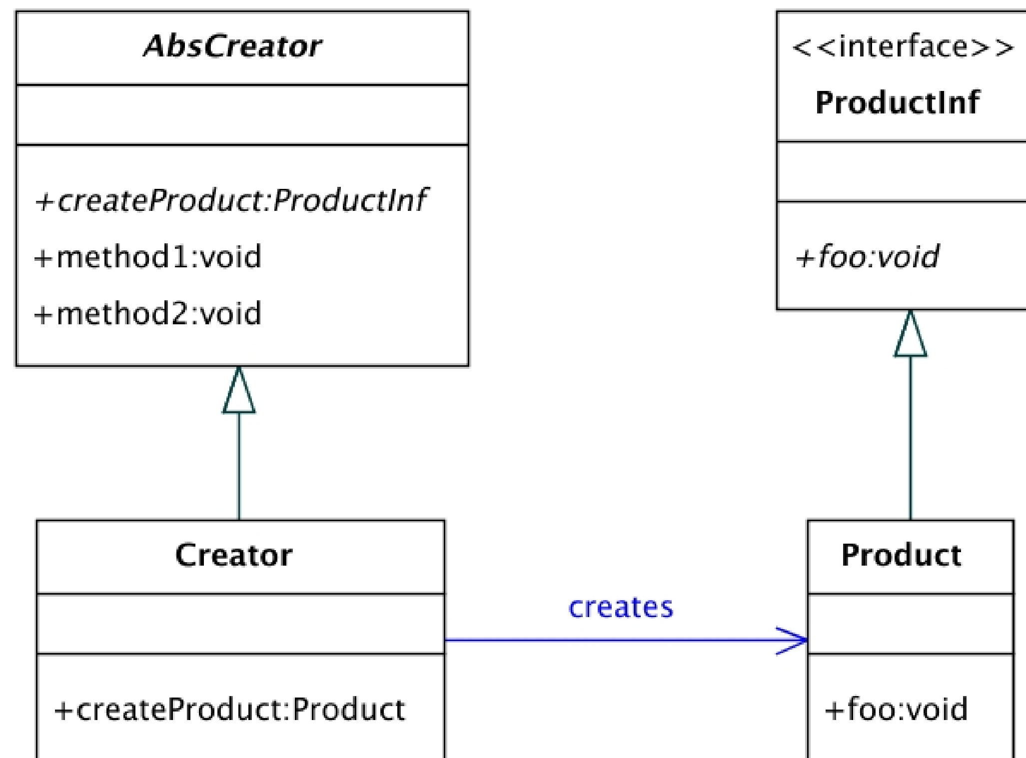
}

}

5

PartialAbstraction

Erzeugung einer abstrakten Oberklasse für Creator



5 PartialAbstraction Transformation

PartialAbstraction(Class *concrete*, String *newName*,
SetOfString *abstractMethods*) {

Class abstract = createEmptyClass(*newName*);
addClass(abstract, superclass(*concrete*), *concrete*);

ForAll m:Method, m Є *concrete*, nameOf(m) Є
abstractMethods {
Method absMethod = abstractMethod(m);
addMethod(*abstract*, absMethod);
}
ForAll m:Method, m Є *concrete*, nameOf(m) Э
abstractMethods {
pullUpMethod(m);
}

}

5 Vollständige Transformation

applyFactoryMethod(Class *creator*, Class *product*, String *productInf*, String *absCreator*, String *createProduct*) {

Abstraction(*product*, *productInf*);

EncapsulateConstruction(*creator*, *product*, *createProduct*);

AbstractAccess(*creator*, *product*, *productInf*, *createProduct*);

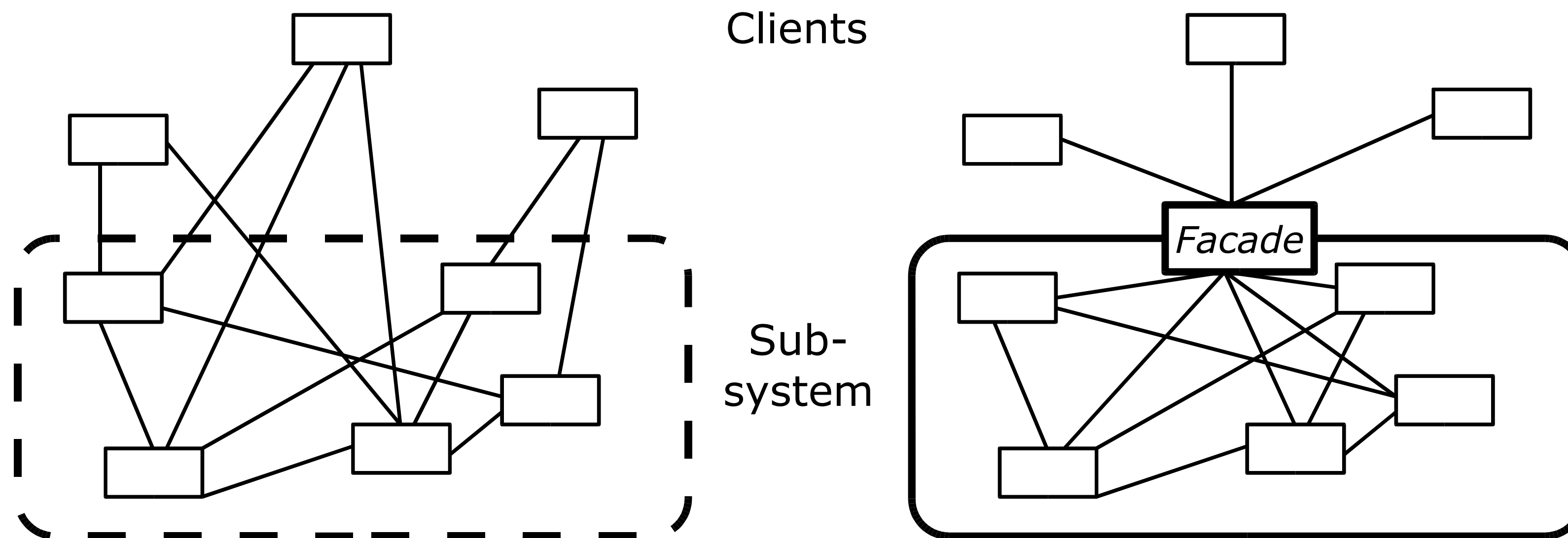
PartialAbstraction(*creator*, *absCreator*, *createProduct*);

}

6 Fehlgeschlagene Transformation

Beispiel: *Facade*-Entwurfsmuster

- Bündelung von Zugriffen auf ein Subsystem durch ein Interface
- praktisches Beispiel: Compilersystem



6 Fehlgeschlagene Transformation

Precursor:

Eine Menge von Klassen interagiert mit einer anderen Menge von Klassen

Problem:

- gemeinsamer Nenner für das Interface muss gefunden werden
- Die Abhängigkeiten zwischen Client und Subsystem können aber sehr komplex sein

=> Problem kann in der Praxis nicht generisch gelöst werden

7 Unterstützung durch Werkzeuge

Praktischer Einsatz der automatischen Integration

Together 6 (http://www.togethersoft.com)

- ist ein Modellierungswerkzeug
- Arbeit auf Entwurfsebene mit UML
- Unterstützt eine gewisse Anzahl von Entwurfsmustern

als Beispiel wird wieder *Factory Method* benutzt

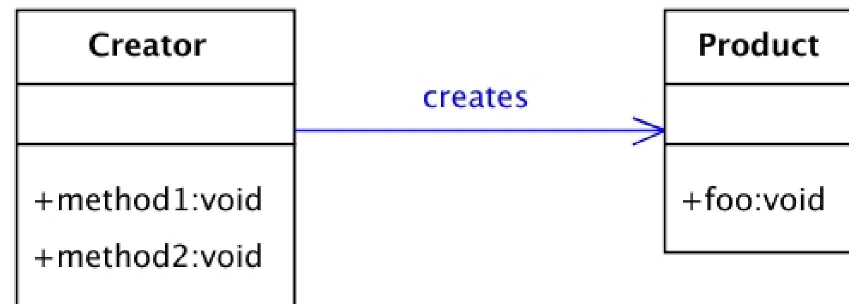
es soll ein Vergleich mit der beschriebenen Methode angestellt werden

7

Beispieltransformation

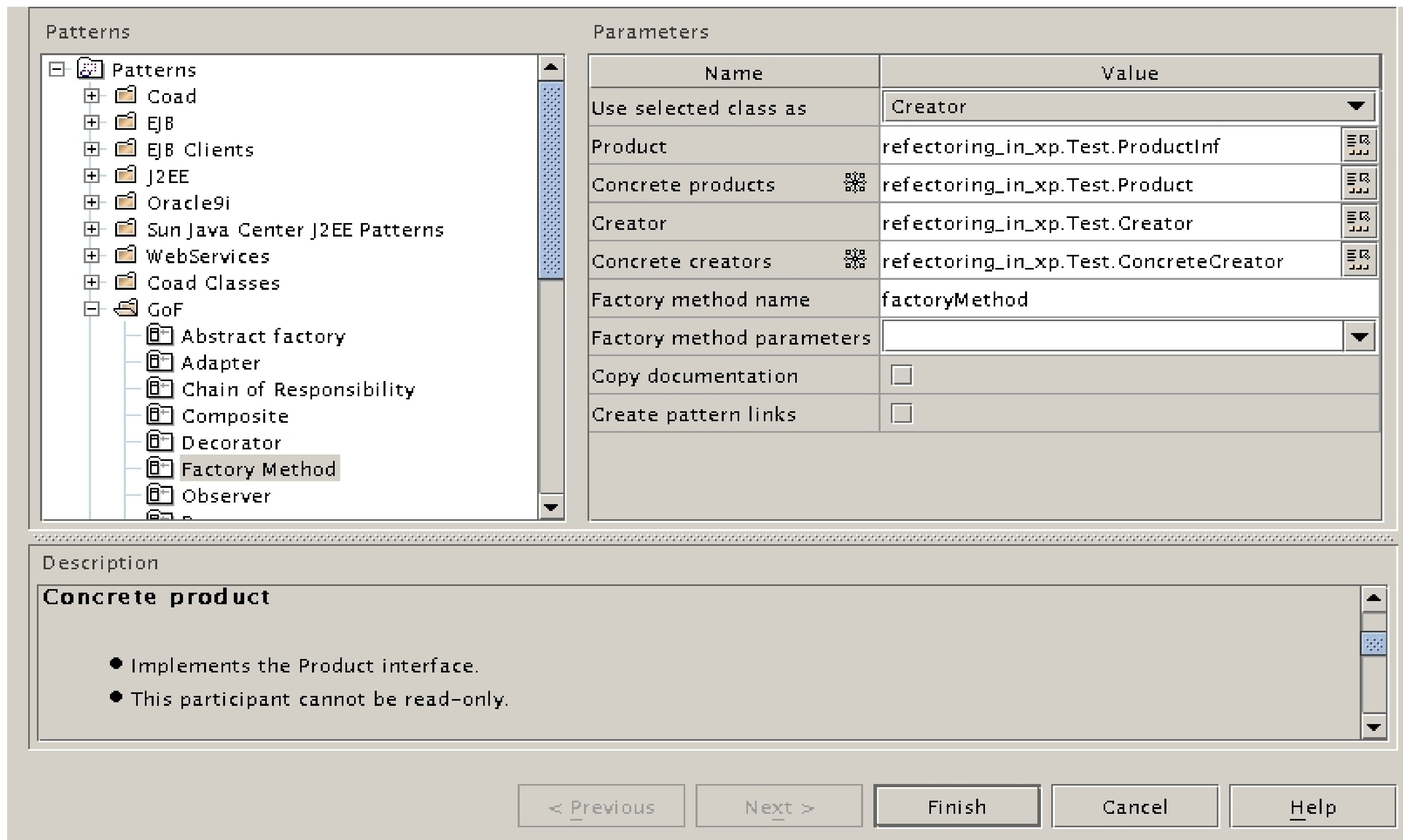
Ausgangspunkt ist Precursor

```
public class Creator {
    public void method1() {
        Product p = new Product();
    }

    public void method2() {}
}

public class Product {
    public void foo() {}
}
```



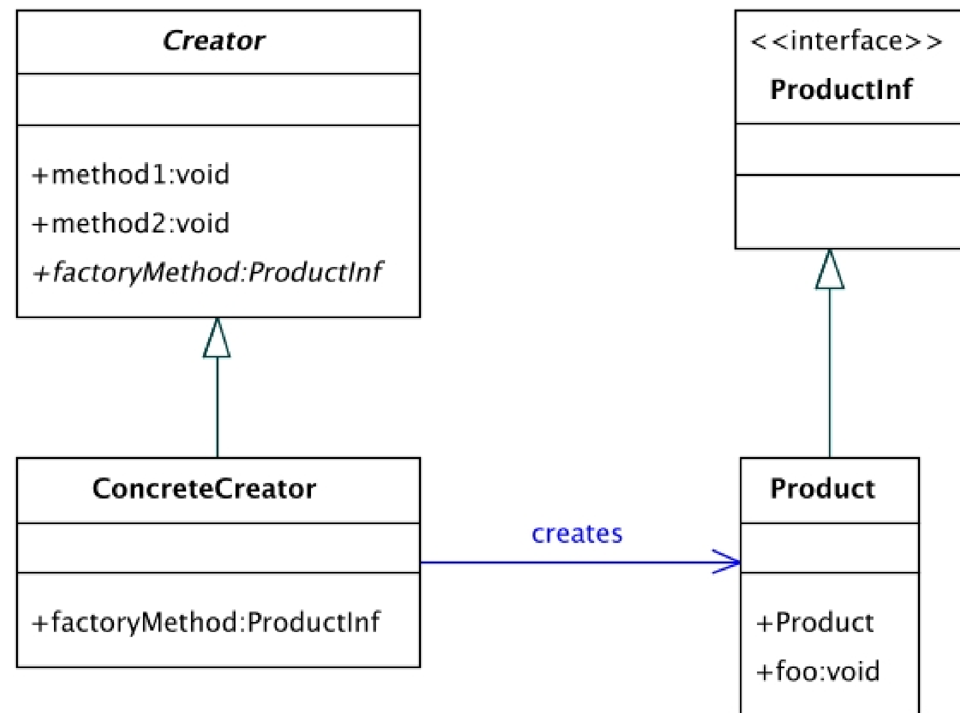
7

Transformationsdialog



7 Ergebnis

```
abstract public class Creator {
    public abstract ProductInf factoryMehod();

    public void method1() {
        Product p = new Product();
    }

    public void method2() {}
}
```



Korrekt:

```
Product p = factoryMethod();
```

8 Schlussfolgerung

Modell funktioniert

Precursor ist Schlüssel

Ergebnisse für Gamma *et al* Katalog:

- 11 vollständig transformierbar
- 6 teilweise transformierbar
- 6 nicht transformierbar

Einschränkung:

- Modell untersucht nur Struktur
- kann Verhalten nicht erkennen

Literatur

Gunter Saake, Andreas Heuer. *Datenbanken: Konzepte und Sprachen 2., aktualisierte und erweiterte Auflage.* Mitp, 2000.

Mel Ó Cinnéide. *Automated Application of Design Patterns: A Refactoring Approach.* PhD thesis, University of Dublin, Trinity Coolege, 2001.

Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

zeroCode build. *Software Lifecycle Costs.* ZeroCode-build, Mai 2002.