CHAIR IV
Software & Systems Engineering

TECHNISCHE UNIVERSITÄT MÜNCHEN

Vorlesung Software-Wartung
4.6. Programm-Transformation

Dr. Markus Pizka
Technische Universität München
Institut für Informatik
pizka@in.tum.de

Programm-Transformation

TECHNISCHE UNIVERSITÄT MÜNCHEN

Programm P: strukturierte Einheit mit Verhalten (Semantik)

- Struktur erlaubt Umformung von P in P´
- Semantik erlaubt
  - Vergleich von P und P´
  - Beurteilung der Zulässigkeit der Transformation
- Semantik hier: von außen beobachtbares und inneres Verhalten

Programm-Transformation ist das Überführen eines Programms P in ein anderes Programm P´

$$t: Q \rightarrow Z;\ t(P) = P'$$

$P \in Q$ (Quellsprache), $P' \in Z$ (Zielsprache)

Bezeichnungen

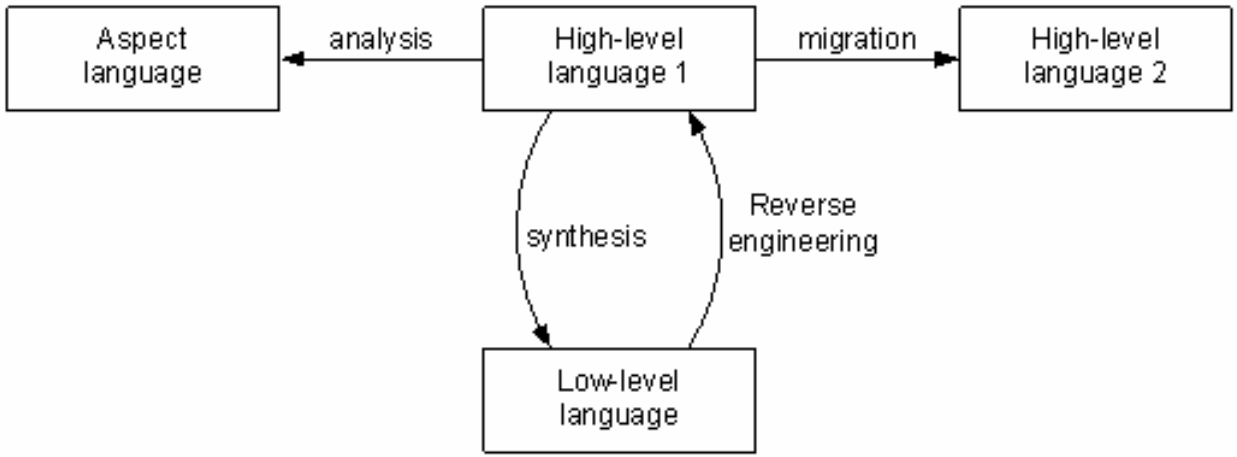
- weitere Bezeichnungen:
  - Meta programming
  - Software generation
  - Generative programming
  - Program synthesis
  - Program refinement
  - Program calculation
  - Refactoring

- charakteristisches Merkmal Q = Z ?

  Q ≠ Z: Übersetzung (Translation)

  Q = Z: Umformulierung (Rephrasing)

Übersetzung



sei A(L) Abstraktionsniveau der Sprache L

- Synthese: A(Q) > A(Z); z.B. Java-Übersetzer
- Migration: A(Q) = A(Z); z.B. C++ → Java
- Reverse Engineering: A(Q) < A(Z)
- Programm-Analyse: $Z \subset Q$ (d.h. Konzentration auf einen Aspekt)

Umformulierung

- Ziel:
  - Gleiches in anderen Worten ausdrücken
  - Änderungen eines bestimmten Aspekts ➔ Änderung der Semantik
- Varianten:
  - Programm-Normalisierung ~ Überführung in einheitliche Form
  - Programm-Optimierung (Zeit-/Platzbedarf), z.B.:
    - Falten von Konstanten, 3 + 5 ==> 8
    - Eliminieren von nicht erreichbarem Code
  - Refactoring ~ Verbesserung des Designs
  - Verschleierung ~ Lesbarkeit vermindern
  - Reflektion ~ zusätzliche Berechnung Programm-Eigenschaft
    z.B. Anfertigung eines Ausführungsprotokolls
  - Renovierung ~ Fehlerbehebung, Restrukturierung

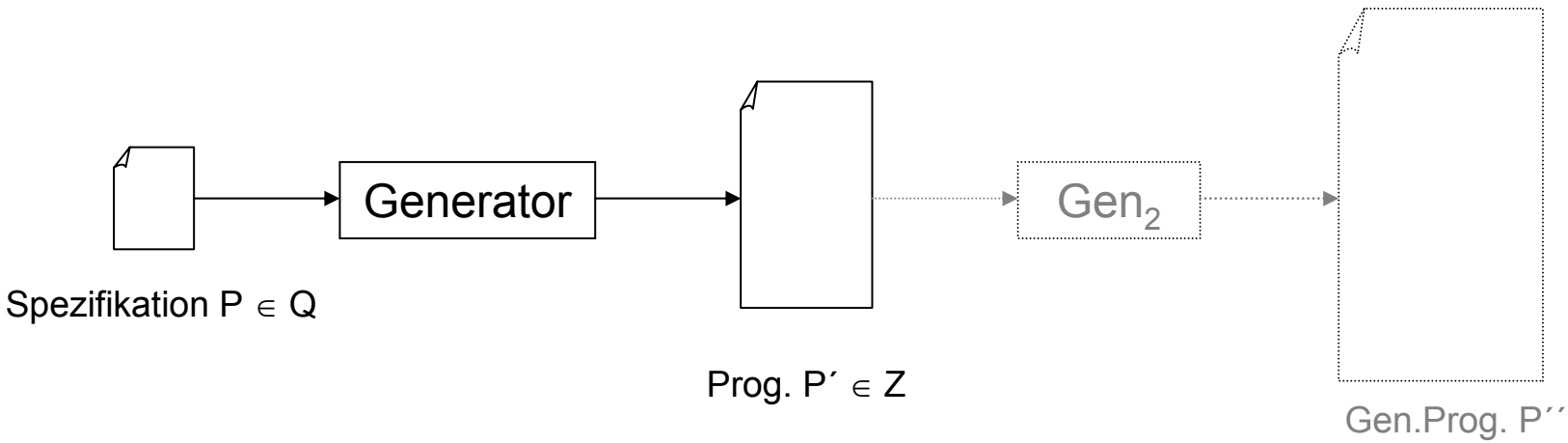
… von weiterem Interesse

1. Synthese von Programmen durch Verfeinerung
   - herausragende Chance zur Reduzierung des Wartungsaufwands
   - seit vielen Jahren in der Forschung und Praxis
   - aktuell: UML, MDA

2. Refactoring
   - populär
   - werkzeuggestützt
   - Fehleinschätzungen

Programm-Synthese

- Idee: Generierung von Programmen aus Spezifikation auf höherer Abstraktionsordnung

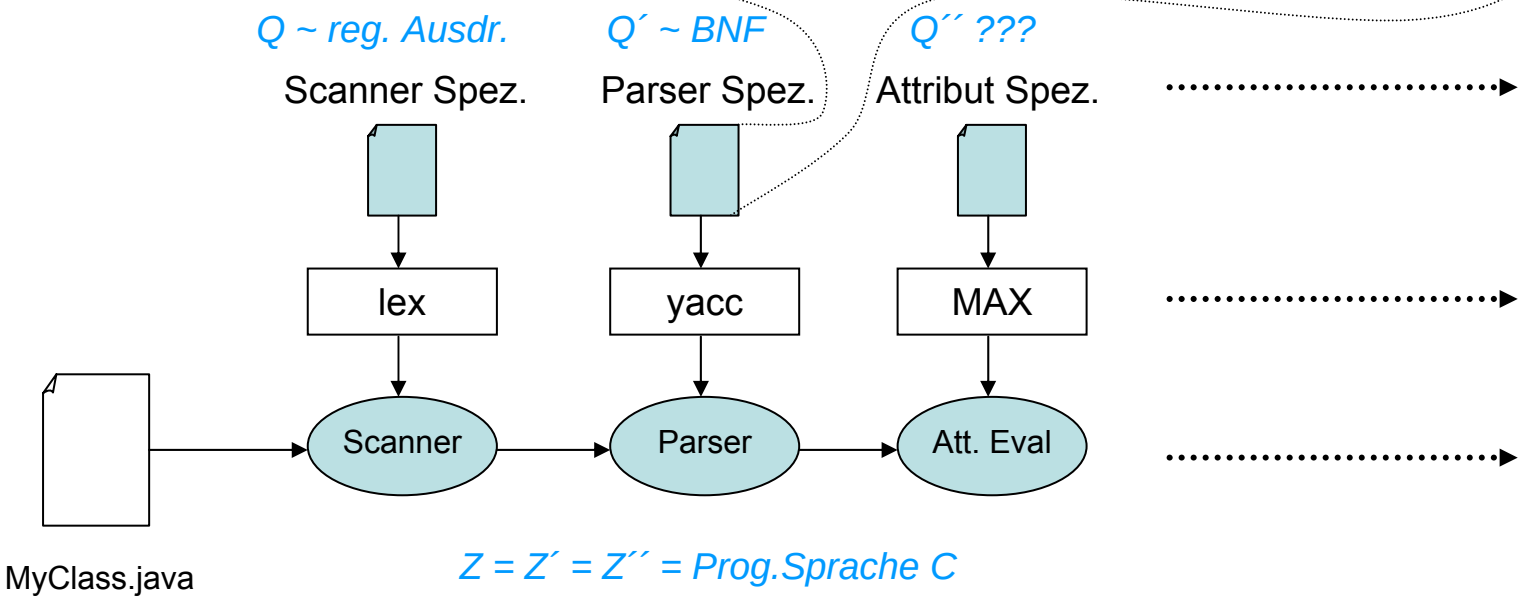


Chancen:

- Qualität: Verhinderung von Fehlern bei manueller Verfeinerung
- Produktivität: falls |P| < |P´| (==> auch reduzierter W-Aufwand)

Erfolgsgeschichten (1) - Übersetzerbau

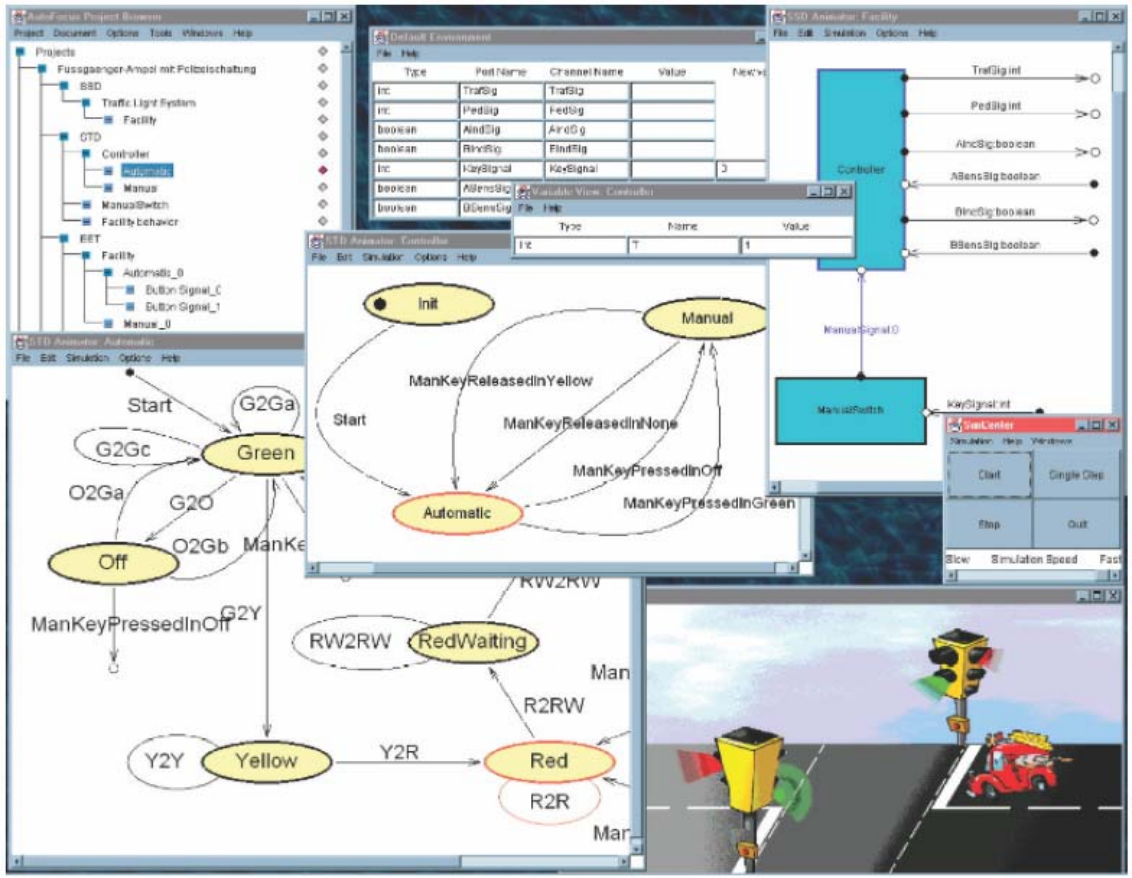
```
for_while:   /*empty */              { $$ = MAX_Empty(); }
           | FOR IDENTIFIER IN range_part
             { $$ = MAX_For( MAX_DeclId( $2.val, MAX_ptoe(filename), $2.ln ), $4 ); }
           | WHILE condition     { $$ = MAX_While( $2 ); };
if_stmt: IF condition THEN statement_part
           elsif_part
           else_part
         END IF { $$ = MAX_If( MAX_appfront(MAX_IfRule( $2, $4), $5), $6 ); };
```



primäre Motivation: Reduzierung des Aufwands

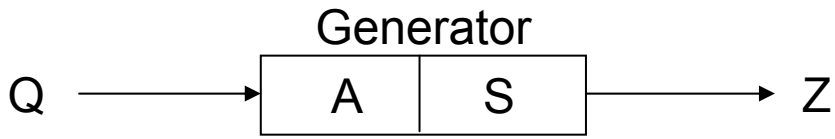
Erfolgsgeschichten (2) – eingebettete Systeme

Autofocus: Code-Generierung aus formalen System-Modellen



Motivation: Korrektheit.

Generatoren



- Analyse
  - Einlesen des Quellprogramms
  - Informationsgewinnung

    Bsp: Daten-/Kontrollstrukturen, benötigte Infrastruktur, …
- Synthese

  wesentliche Leistung des Generators
  - Treffen optimierter Realisierungsentscheidungen (Verfeinerung)

    Bsp: Register oder Keller, Verkettete Liste oder Hashing, …
  - Produktion des Zielprogrammes
- Leistung abh. von:
  - Wortschatz der Quellsprache Q (Frage: Einschränkungen?)
  - Analyse + Synthese

Chancen / Risiken aus Sicht der Wartung

- Chancen
  - Voraussetzung |P| < |P´|; P ∈ Q, P´ ∈ Z, t : Q → Z, t(P)= P´

    ~ erfordert A(Q) > A(Z)
  - höhere Produktivität (geringerer Umfang) + Qualität (A.-Niveau)

- Schwierigkeiten
  - Automatisierung limitiert (s. Berechenbarkeit + Komplexität)
  - Q mit wachsendem A.-Niveau domänenspezifisch
    - keine schlanke high-level Universalsprache
    - Kenntnis umfangreicher Quellsprache(n)
  - Abhängigkeit von Generatoren erfordert stabile Q
  - Generierte Programme i.A. nicht sinnvoll manuell bearbeitbar
    - Vollständigkeit und Durchgängigkeit der Transformation
    - keine manuellen Eingriffe in t(P)!

Ausprägungen

- CIP (Computer-Aided, Intuituon-Guided, Programming)
  - „The Munich Project CIP“; F.L. Bauer, TUM, 1984
  - Breitband-Sprache mit
    - Kern: Logik- und Funktionale Programmierung
    - applikative und prozedurale Elemente
    - formale Transformationen
- Programme aus Beweisen (TUM: Isabelle, LMU: MINLOG)

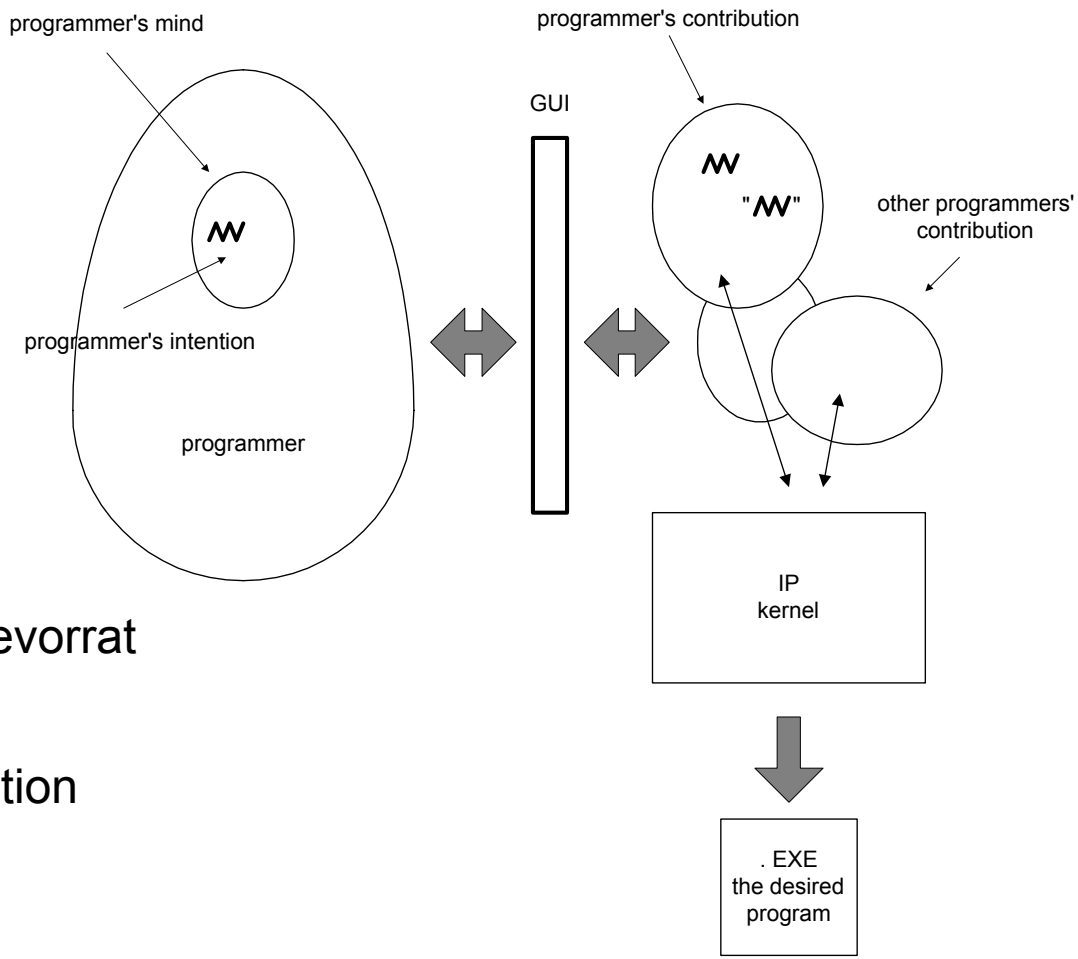
$$\textbf{theorem}\ division\colon \exists r\ q.\ a = Suc\ b * q + r \wedge r \leq b$$

Beweis (Theorembeweiser)

$$\begin{array}{l} division \equiv \\ \lambda x\ xa. \\ \quad nat\text{-}rec\ (0,\ 0) \\ \quad\ (\lambda n.\ prod\text{-}case \\ \qquad\qquad (\lambda x\ y.\ case\ nat\text{-}eq\text{-}dec\ x\ xa\ of\ Left \Rightarrow (0,\ Suc\ y) \\ \qquad\qquad\qquad\qquad \mid Right \Rightarrow (Suc\ x,\ y))) \\ \quad\ x \end{array}$$

Ausprägungen - Intentional Programming

„The death of computer languages, the birth of Intentional Programming", Charles Simonyi, 1995, Microsoft



- Konzeptsprache
- erweiterbarer Konzeptevorrat
- GUI Repräsentationen
- automatische Komposition

Ausprägungen

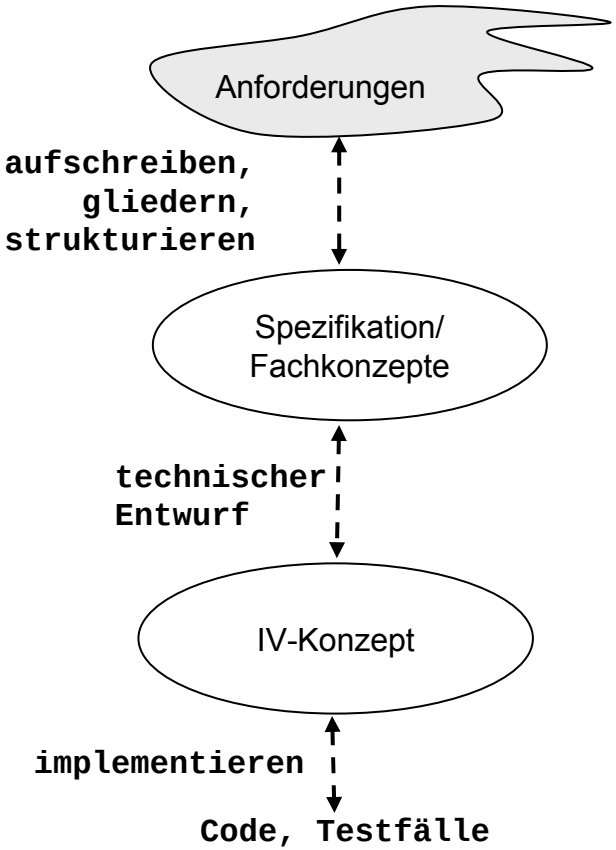
- Makrosubstitution, z.B. C Präprozessor
  - `#define min(X, Y)  ((X) < (Y) ? (X) : (Y))`
  - `#define COMMAND(NAME) { #NAME, NAME ## _command }`

```
struct command commands[ ] = {
        COMMAND (quit),
        COMMAND (help), ... };
```

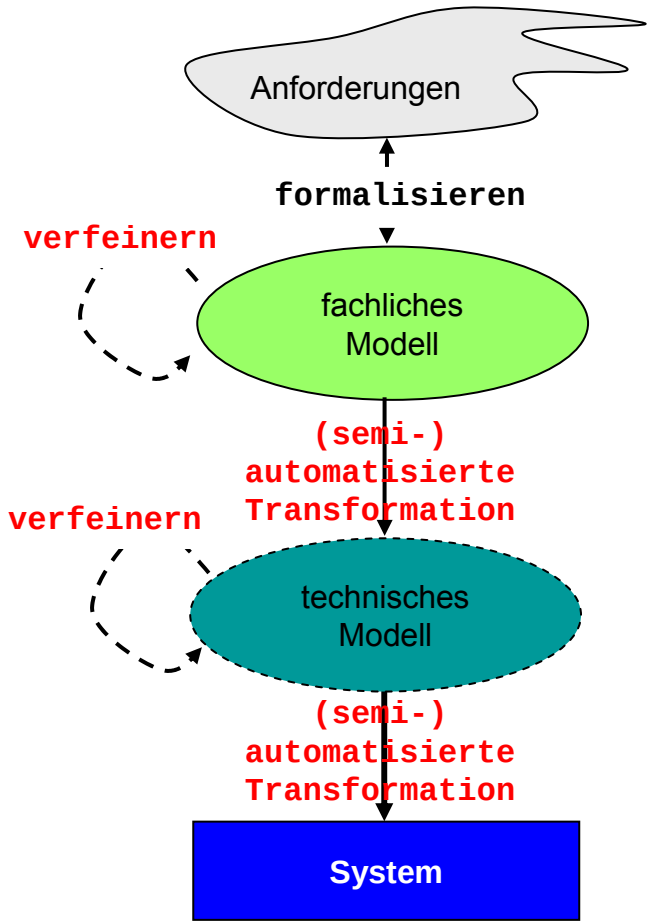
- C++ Template Programmierung
  - `template <class T> T max(T a, T b) { return a > b ? a : b ; }`
- Domänenspezifische Sprachen
  - Bsp.: SQL, GUI-Sprachen, partielle Differentialgleichungen, …
  - Domänen-Analyse → Domänen-Modell → Sprache → Generator

Modellierung und Generierung

Dokument-basiert

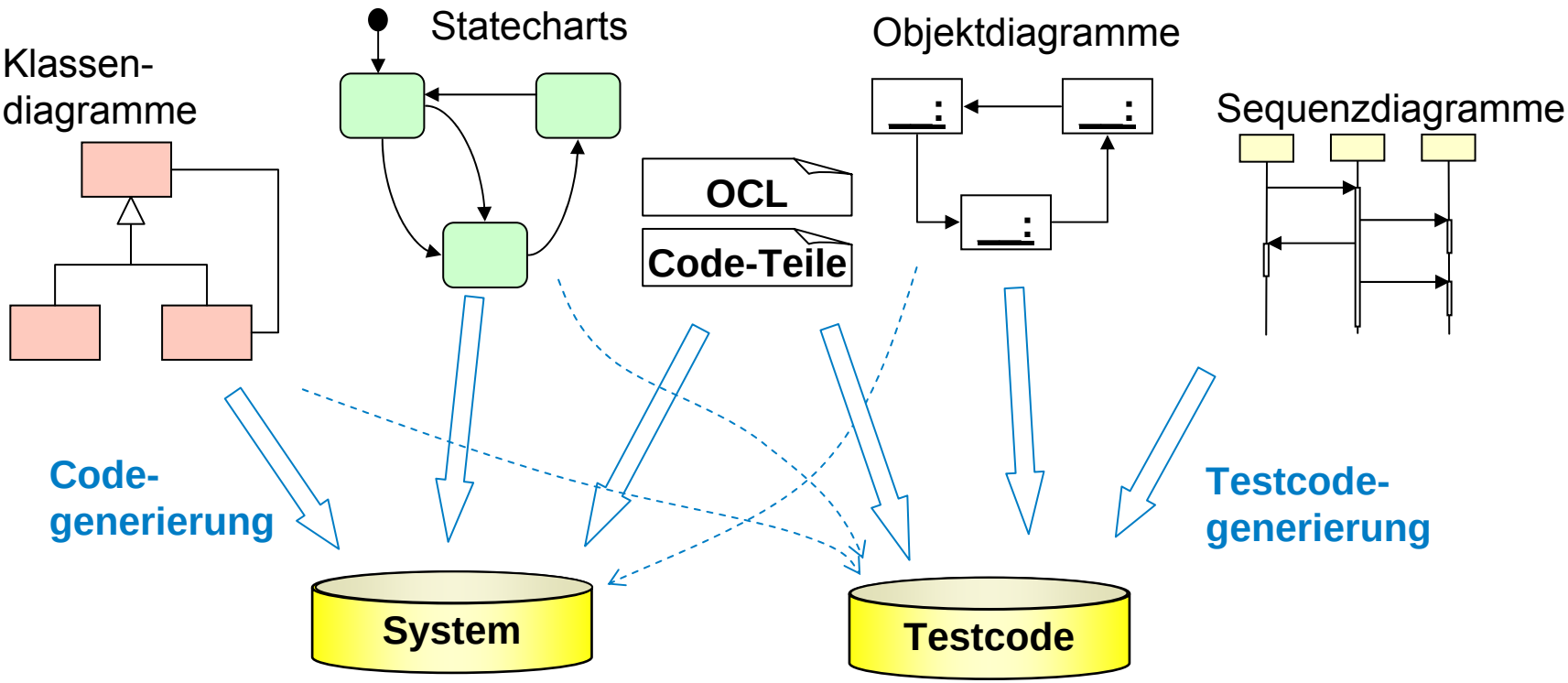


Modell-basiert



Formale Modelle Voraussetzung für Generierung!

Beispiel UML



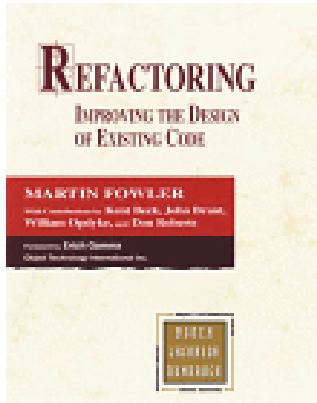
(Quelle: B. Rumpe, 2003)

- Bruchstellen
- mangelnde Systematik der Verfeinerung
- Abhängigkeit von (volatilen) Generatoren

Refactoring

- William Opdyke, 1992
  - Ph.D. Thesis “Refactoring OO Frameworks”
  - Smalltalk
- Martin Fowler, 1999
  - Buch “Improving the Design of Existing Code”
  - Vorstellung:
    - disciplined technique for restructuring an existing body of code
    - altering its internal structure without changing its external behavior
    - series of small transformations … significant restructuring
    - since each refactoring is small, it's less likely to go wrong
    - many … automated

    ~ ohnehin notwendige Dinge automatisieren
  - Kontext:
    - Agile Software-Entwicklungsprozesse (eXtreme Programming)
    - häufige Anwendung bei Bedarf



Refactoring Katalog (www.refactoring.com)

TECHNISCHE UNIVERSITÄT MÜNCHEN

- Add Parameter
- Change Bidirectional Association to Unidirectional
- Change Reference to Value
- Change Unidirectional Association to Bidirectional
- Change Value to Reference
- Collapse Hierarchy
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Convert Dynamic to Static Construction
- Convert Static to Dynamic Construction
- Decompose Conditional
- Duplicate Observed Data
- Eliminate Inter-Entity Bean Communication Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Class / Interface / Method / Package
- Extract Subclass / Superclass
- Form Template Method
- Hide Delegate
- Hide Method
- Hide presentation tier-specific details from the business tier Inline Class
- Inline Method / Temp
- Introduce A Controller / Assertion
- Introduce Business Delegate
- Introduce Explaining Variable
- Introduce Foreign Method
- Introduce Local Extension
- Introduce Null Object
- Introduce Parameter Object
- Introduce Synchronizer Token
- Localize Disparate Logic
- Merge Session Beans
- Move Business Logic to Session
- Move Class
- Move Field
- Move Method
- Parameterize Method
- Preserve Whole Object
- Pull Up Constructor Body
- Pull Up Field
- Pull Up Method
- Push Down Field
- Push Down Method
- Reduce Scope of Variable

uvm., 07/2004, 93 Refactorings

Refactoring im Detail (1)

automatisiert

Substitute Algorithm

„You want to replace an algorithm with one that is clearer.“

```
String foundPerson(String[] people){
        for (int i = 0; i < people.length; i++) {
                if (people[i].equals ("Don")){
                        return "Don";
                }
                if (people[i].equals ("John")){
                        return "John";
                }
                if (people[i].equals ("Kent")){
                        return "Kent";
                }
        }
        return "";
}
```

```
String foundPerson(String[] people){
        List candidates = Arrays.asList(new String[] {"Don", "John", "Kent"});
        for (int i=0; i<people.length; i++)
                if (candidates.contains(people[i]))
                        return people[i];
        return "";
}
```

Refactoring im Detail (2)

Wrap entities with session

Entity beans from the business tier are exposed to clients in another tier:

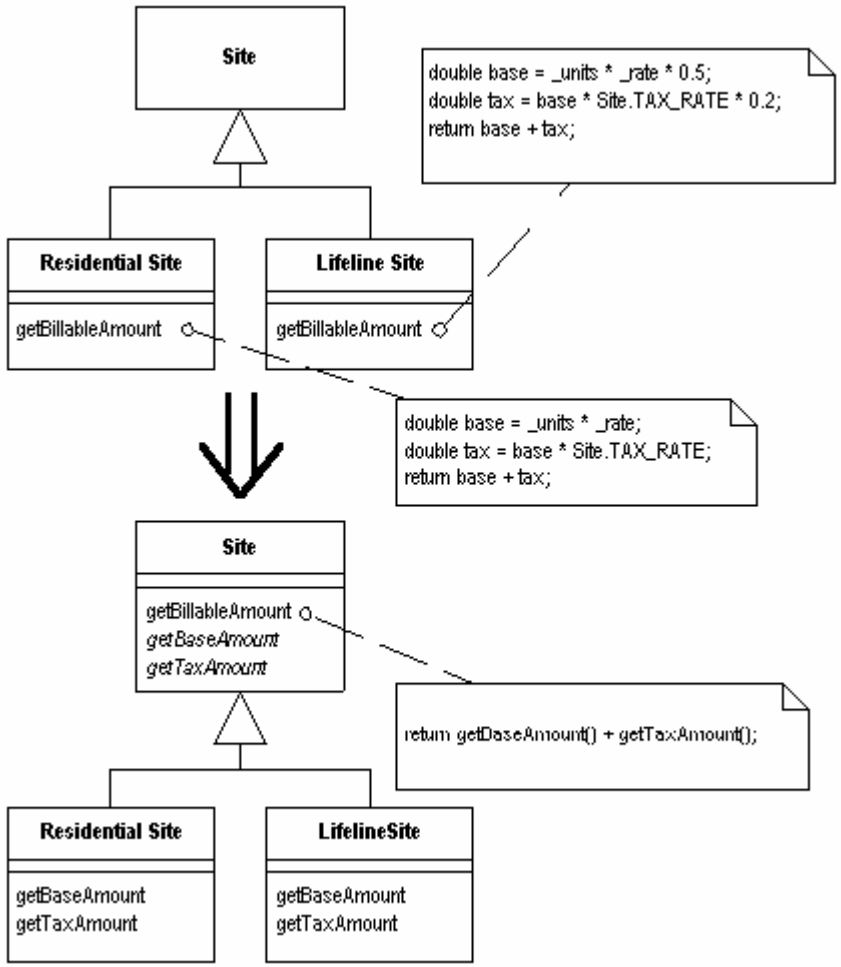
Use a Session Facade to encapsulate the entity beans

Form Template Method

2 methods in subclasses similar steps in same order, yet steps different:

Get steps into methods with the same signature, so that the original methods become the same. Then you can pull them up.

simple, small?



Refactoring im Detail (3)

Extract_Method

- lange Anweisungslisten
  ~ Strukturmangel
- Lösung:
  Extraktion in Methode
- Werkzeugunterstützt
  (z.B. Eclipse)

```
      int i = 0, j = 0;
      while (true) {
->        System.out.println(s);
->        i++;
      }
```

```
    private int foo(String s, int i) {
        System.out.println(s);
        i++;
        return i;
    }

    while (true) {
       i = foo(s, i);
    }
```

Refactoring im Detail (4)

- Extract_Method (2)

```
        while (true) {
->          System.out.println(s);
->          i++;
->          j++;
        }
```

- automatische Extraktion nicht möglich!
- Typ von i und j muss geändert werden, z.B. int[ ]

  … alle Operationen auf i und j in müssten in semantisch äquivalente Op. des neue Typs transformiert werden.
- i.A. nicht entscheidbar ~ Halteproblem

==> selbst elementare Refactorings nicht automatisierbar!

Refactoring Defizite

- Refactoring:

  syntaktische Transformation
  primär Verschieben von Text Fragmenten

- Defizite in Programmen:

  in der Regel semantische Schwächen

  z.B. inadäquate Algorithmen & Datenstrukturen

```
a = 0;
while (c > 0) {
    a = a + b;
    c--;
}
```

$\Rightarrow$

```
a = b * c;
```

kein automatisierbares Refactoring!

+ mangelnde Systematik, Orthogonalität, …