Generierung von Software aus Spezifikationen

Prof. Dr. Uwe Kastens

WS 2002/2003

Ziele

Die Teilnehmer sollen lernen

- **Generatoren** für spezielle Software-Aufgaben **anzuwenden**,
- anwendungsspezifische **Sprachen zu entwerfen**,
- Programm**generatoren dafür herzustellen**,
- das **Werkzeugsystem Eli** für solche Aufgaben einzusetzen.

Sie sollen ein **selbstdefiniertes Anwendungsprojekt** durchführen können.

Inhalt (aktualisiert)

1. Einführung
   Wiederverwendung und Generatoren
2. Strukturierte Texte generieren
3. **Bäume aufbauen**
4. **Berechnungen in Bäumen**
5. Namen und Eigenschaften zuordnen
6. Sprachentwurf
7. Projekt: Strukturgenerator
8. Individuelle Projekte

Phase 1: Vorlesung, praktisches Tutorium und selbständige Vor- und Nacharbeit eng verzahnt

Phase 2: selbständig Projekte bearbeiten.
In den Vorlesungsstunden beraten wir zu Problemen und tauschen Erfahrungen aus

Referenzen

- U. Kastens: Generierung von Software aus Spezifikationen
  Elektronisches Skript, WS 2002/2003
  http://www.upb.de/cs/ag-kastens/gss
- Eli Online Documentation
  http://www.uni-paderborn.de/project-hp/eli.html
- U. Kastens: Praktikum Sprachimplementierung mit Werkzeugen
  Elektronisches Skript, WS 1999/2000
  http://www.upb.de/cs/ag-kastens/eliprakt/
- Artikel zum Einsatz von Eli
  http://www.uni-paderborn.de/fachbereich/AG/agkastens/eli/bib/eli_bibD.html
  http://www.uni-paderborn.de/fachbereich/AG/agkastens/literatur/literatur.eli.html
  http://www.uni-paderborn.de/fachbereich/AG/agkastens/forschung/visuell.html

Artikel zu Wiederverwendung:

- Ch. W. Kruger: Software Reuse, ACM Computing Surveys, 24(2), 1992
- R. Prieto-Diaz: Status Report: Software reusability, IEEE Software, 10(3), 1993

1.1 Wiederverwendung (Reuse) Was wird wiederverwendet?

Produkt	**Wiederverwendung von ...**
Bibliothek mit Funktionen	Implementierung
Modul, Komponente	Code
generischer Modul	geplante Varianten von Code
Software Architektur	Entwurf
Framework (Programmgerüst)	Entwurf und Code
Entwurfsmuster	Strategien zur Konstruktion
Generator	Wissen, wie man Lösungen aus Beschreibungen konstruiert
Konstruktionsprozess	Wissen, wie man Werkzeuge anwendet und kombiniert

Ch. W. Kruger: Software Reuse, ACM Computing Surveys, 24(2), 1992

R. Prieto-Diaz: Status Report: Software reusability, IEEE Software, 10(3), 1993

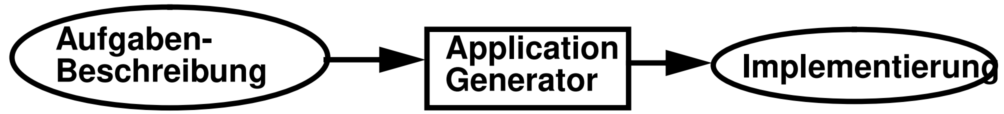
Wo wird Wiederverwendung praktiziert?

Anwendungsgebiet bestimmt das Potenzial der Wiederverwendung

breites, allgemeines Gebiet	**enge** Wiederverwendung (horizontal)
enges Gebiet, **gut verstandene** Begriffe	**mächtige** Wiederverwendung (vertikal) Generatoren, Frameworks

Anwendungsspezifische Sprachen
(Domain specific languages, DSL)

Anwendungsgebiete außerhalb der Informatik
- Robotorsteuerung
- Börsenhandel
- mechanische Steuerungssysteme
- Musiknoten



Gebiete der Software -Technik
- Datenbank-Reports
- Bedienungsschnittstellen
- Repräsentation von Datenstrukturen (XML)

Sprachimplementierung als Anwendungsgebiet
- Scanner spezifiziert durch reguläre Ausdrücke
- Parser spezifiziert durch kontextfreie Grammatiken
- Sprachimplementierung spezifiziert für *Eli*

Wie wird Wiederverwendung organisiert?

Wie	Produkte	Konsequenzen
ad hoc	• Code kopieren und modifizieren • inkrementelles Adaptieren durch oo Unterklassen	• keine Kosten a priori • sehr problematisch für die Wartung
geplant	• oo Bibliotheken, Frameworks • Spezialisierung von Klassen	• hohe Kosten a priori • wirksame Wiederverwendung
automatisch	• Generatoren, Entwicklungsumgebungen	• hohe Kosten a priori • wirksame Wiederverwendung • große kognitive Distanz

Wer bereitet Wiederverwendung vor, wer nutzt sie?

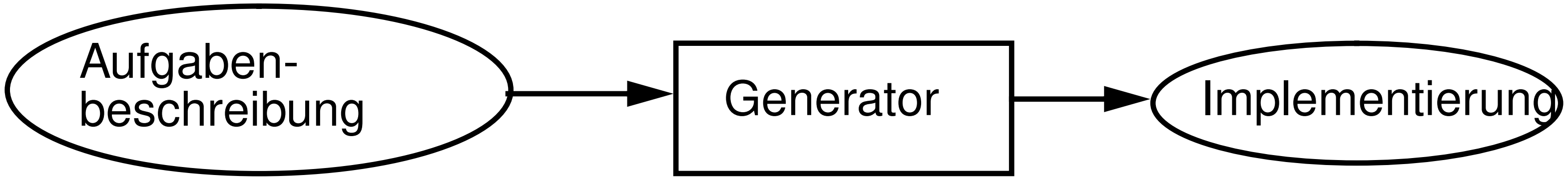
Bereitstellung und Wiederverwendung auf demselben Erfahrungsstand:

- dieselben Personen, Gruppen, Berufe
- Anbieter nimmt an, dass der Benutzer die Produkte genauso versteht
- Beispiele: Wiederverwendung von Code, Entwurfsmuster

Experte bietet Produkte an zum wiederverwenden durch Amateure:

- Wiederverwendung überspannt eine große **kognitive Distanz**
- **Expertenwissen** wird Nicht-Experten verfügbar gemacht
- erfordert, dass das Anwendungsgebiet **vollständig verstanden ist**
- erfordert Begriffe und **Konzepte auf hohem Niveau**
- Beispiele: Generatoren, Frameworks, Entwicklungsumgebungen

1.2. Das Generator-Prinzip



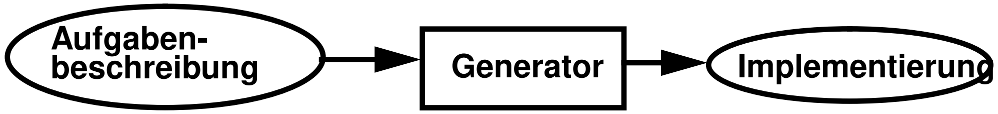
Application generator: die wirksamste Wiederverwendungs-Methode
[Ch. W. Kruger: Software Reuse]

spezielles Anwendungsgebiet eng	vollständig verstanden Implementierungen automatisch erzeugt
Abstraktionen auf hohem Niveau	abgebildet auf ausführbare software
Benutzer versteht **Abstraktionen** des Anwendungsgebietes	**Experte** versteht **Implementierungsmethoden**

große kognitive Distanz
Generator macht Expertenwissen verfügbar

Beispiele: Datenbank-Report-Generator
Bedienungsschnittstellen-Generator
Parser-Generator

Spezifikationssprachen für Generatoren



Anwendungsspezifische Sprachen
(Domain specific languages, DSL)

Anwendungsgebiete außerhalb der Informatik
- Robotorsteuerung
- Börsenhandel
- mechanische Steuerungssysteme
- Musiknoten

Gebiete der Software -Technik
- Datenbank-Reports
- Bedienungsschnittstellen
- Repräsentation von Datenstrukturen (XML)

Sprachimplementierung als Anwendungsgebiet
- Scanner spezifiziert durch reguläre Ausdrücke
- Parser spezifiziert durch kontextfreie Grammatiken
- Sprachimplementierung spezifiziert für *Eli*

Generator: **übersetzt Spezifikationssprache**
in ein ausführbares **Programm und/oder Daten**
benutzt anwendungsspezifische
Methods und Techniken

Beispiel: Wrapper-Klassen generieren

```
int;
PairPtr; "Pair.h"
float;
```

Spezifikation

generierter C++ Modul

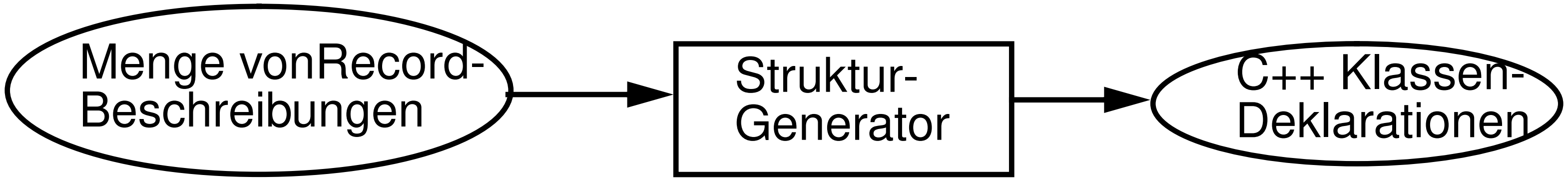
```
#ifndef WRAPPER_H
#define WRAPPER_H

#include „Pair.h“

#define noKind      0
#define intKind 1
#define PairPtrKind 2
#define floatKind 3

class intWrapper;
class PairPtrWrapper;
class floatWrapper;

class Object {
public:
  class WrapperExcept {};
  int getKind () { return kind; }
  int getintValue ();
  PairPtr getPairPtrValue ();
  float getfloatValue ();
protected:
  int kind;
};
```

```
class intWrapper : public Object {
private:
   int v;
public:
   intWrapper (int value)
       { kind = intKind; v = value; }
   int getValue () { return v; }
};

class PairPtrWrapper : public Object {
private:
   PairPtr v;
public:
   PairPtrWrapper (PairPtr value)
       { kind = PairPtrKind; v = value; }
   PairPtr getValue () { return v; }
};

class floatWrapper : public Object {
private:
   float v;
public:
   floatWrapper (float value)
       { kind = floatKind; v = value; }
   float getValue () { return v; }
};

#endif
```

```
#include „example.h“

int Object::getintValue () {
   if (kind == intKind)
       return ((intWrapper*)this)->getValue();
   else
       throw WrapperExcept();
}

PairPtr Object::getPairPtrValue () {
   if (kind == PairPtrKind)
     return ((PairPtrWrapper*)this)->getValue();
   else
       throw WrapperExcept();
}

float Object::getfloatalue () {
   if (kind == floatKind)
       return ((floatWrapper*)this)->getValue();
   else
       throw WrapperExcept();
}
```

Beispiel: Struktur-Generator

Generator um Record-Strukturen zu implementieren,
zu benutzen als **Hilfsmittel bei der Software-Konstruktion**




```
Customer ( addr:      Address;
           account:   int; )

Address (  name:  String;
           zip:   int;
           city:  String; )

import String from "util.h"
```

```
#include "util.h"

typedef class Customer_Cl *Customer;
typedef class Address_Cl *Address;

class Customer_Cl {
   private:
      Address addr_fld;
      int account_fld;
   public:
      Customer_Cl
         (Address addr, int account)
      {  addr_fld=addr;
         account_fld=account; }
...
};
```

Aufgabenzerlegung für Implementierung von Spezifikationssprachen

Structuring	**Lexical analysis**	**Scanning** **Conversion**
	Syntactic analysis	**Parsing** **Tree construction**
Translation	**Semantic analysis**	**Name analysis** **Property analysis**
	Transformation	**Data mapping** **Action mapping**

[W. M. Waite, L. R. Carter: Compiler Construction, Harper Collins College Publisher, 1993]

Entspricht der Aufgabenzerlegung für

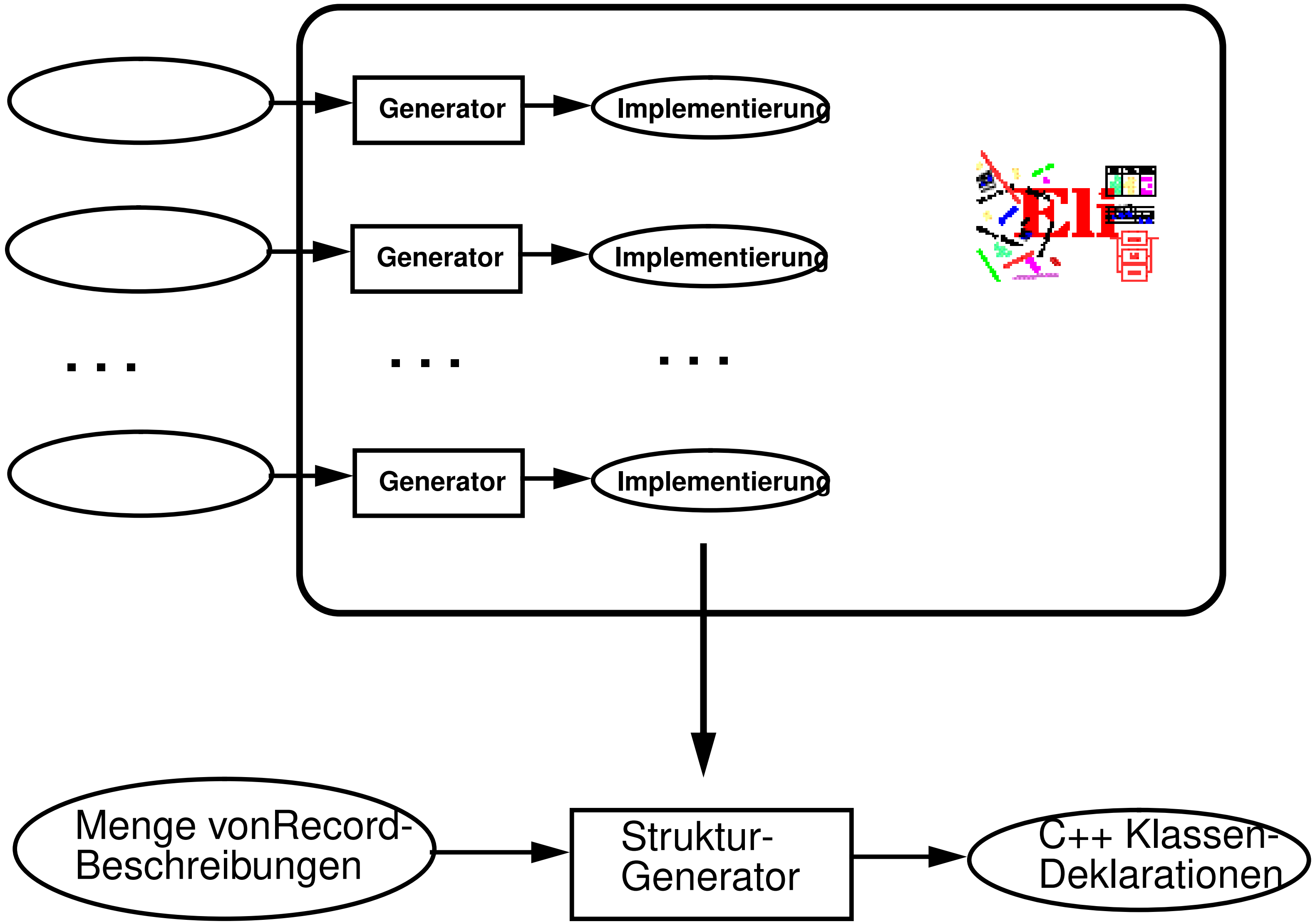
Frontend von Programmiersprach-**Übersetzer** (ohne Code-Erzeugung)
Source-to-Source Übersetzung

Aufgabenzerlegung für den Struktur-Generator

Structuring	Lexical analysis	Symbole der Berschreibung erkennen Namen speichern und codieren
	Syntactic analysis	Struktur der Beschreibung erkennen Struktur als Baum repräsentieren
Translation	Semantic analysis	Namen an Strukturen und Felder binden Eigenschaften speichern und prüfen
	Transformation	Klassen-Deklarationen generieren mit Konstructoren und Zugriffsmethoden

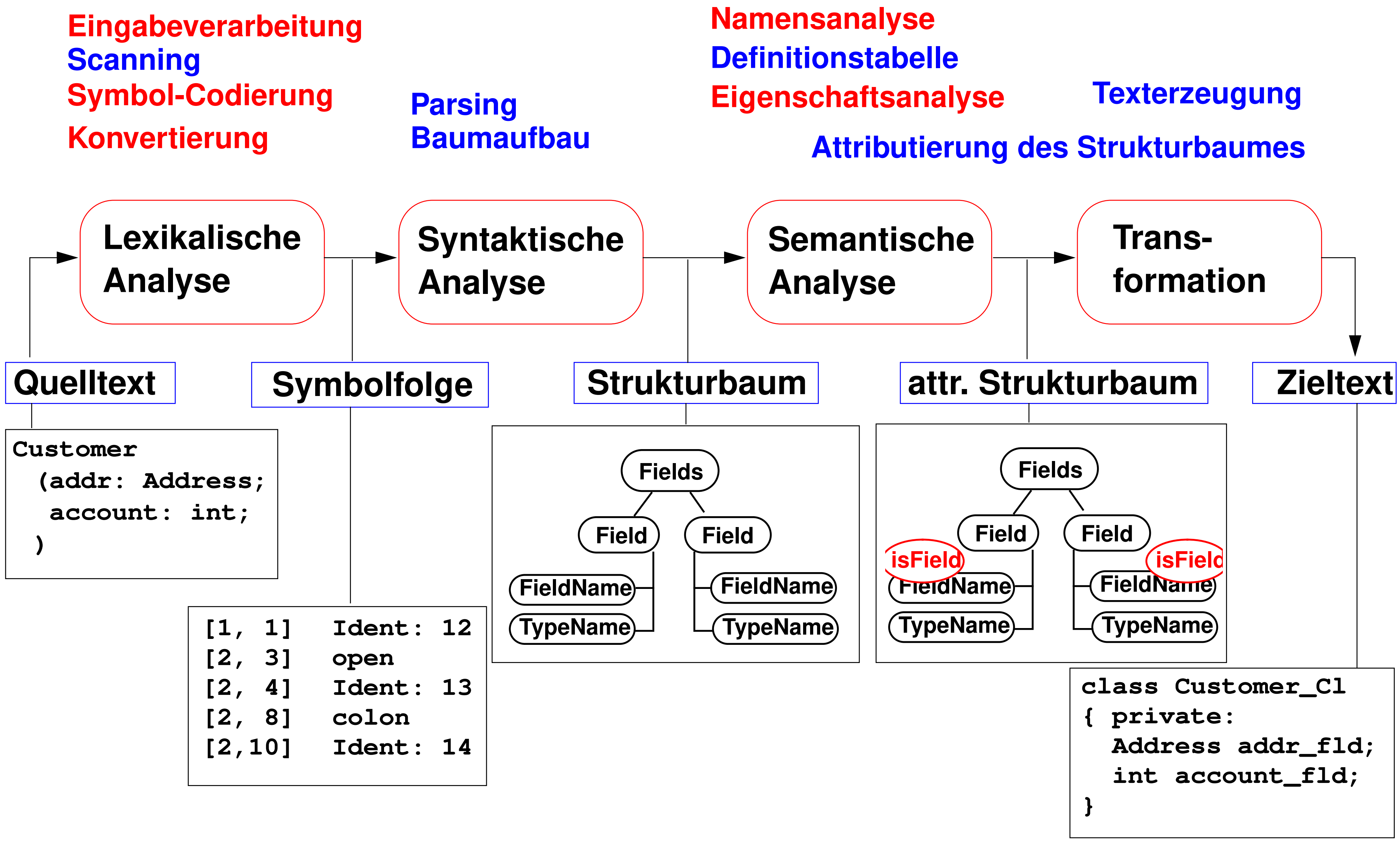
```
Customer ( addr:      Address;
           account:   int; )

Address (  name:  String;
           zip:   int;
           city:  String; )

import String from "util.h"
```

Eli generiert Struktur-Generator

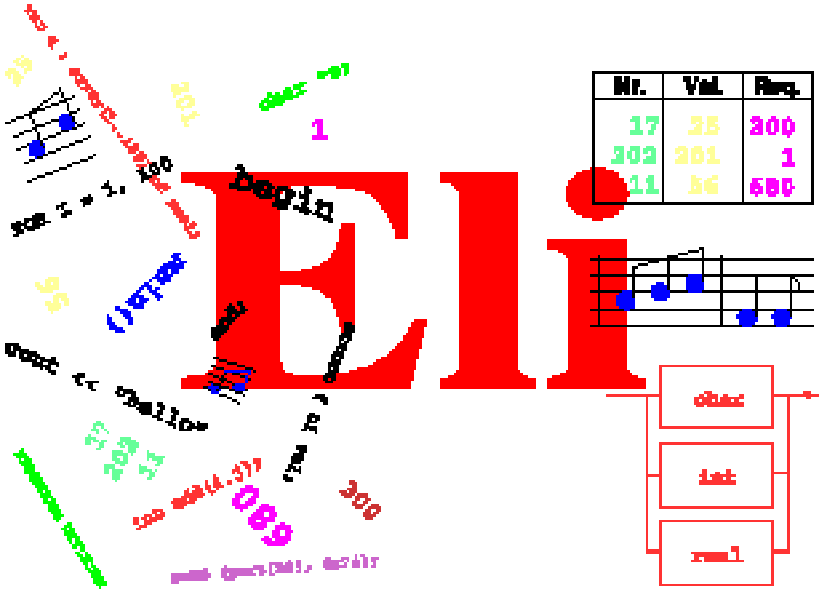


Generator-Architektur gemäß Aufgabenzerlegung

Spezialisierte Werkzeuge erledigen spezielle Aufgaben für die Herstellung des Produktes:



Das Eli System



- **Werkzeugsystem zur Sprachimplementierung**
- Einsetzbar für jede Art von Sprachen:
  Anwendungs-Spezifikationssprachen,
  Programmiersprachen
- **state-of-the-art Compiler Technik**
- Basiert auf der (vollständigen)
  Aufgabenzerlegung (siehe GSS-1.12)
- **Automatischer Herstellungsprozess**

- In vielen **praktischen Projekten** weltweit benutzt
- Entwickelt, erweitert und gewartet seit 1989 von
  William M. Waite (University of Colorado at Boulder),
  Uwe Kastens (University of Paderborn), and
  Antony M. Sloane (Macquarie University, Sydney)
- **Frei verfügbar** via Internet:
  http://www.uni-paderborn.de/project-hp/eli.html