universitätbonn
ROOTS

Typfluss

Dennis Molter

molter@cs.uni-bonn.de

Beispiel: Typgeneralisierung

```
package mini;

class A {
  void a() {}
}
class B extends A
  void b() {}
}
class C {
  B m() {
    B x = new B();
  x.a();
    return x;
  }
  void n() {
    B y = m();
    y.b();
  }
}
```

Genügt Typ A?

Nein!
x muss die Methode
b() kennen!

Beispiel: Typspezialisierung

```
package demo;
class A {
}
class A1 extends A {
}
class A2 extends A {
}
class B {
  void foo() {
    bar(new A1());
  }
  void bar(Object o) {
    A a = (A)o;
  }
}
```

Ist der Cast nicht zu allgemein?

Ja! A1 reicht!

Gliederung

1. **was ist Typfluss?**
   1. Definition
   2. Analogie zu Typfluss
   3. Analogie in Bezug auf CFA / DFA
2. historische Ansätze
3. Tip, Kiezun, Bäumer
   1. allgemeine Vorgehensweise
   2. Typrestriktionen
   3. Beispiel 1: Extract Interface
   4. Beispiel 2: Custom Library „Refactoring“

was ist Typfluss?

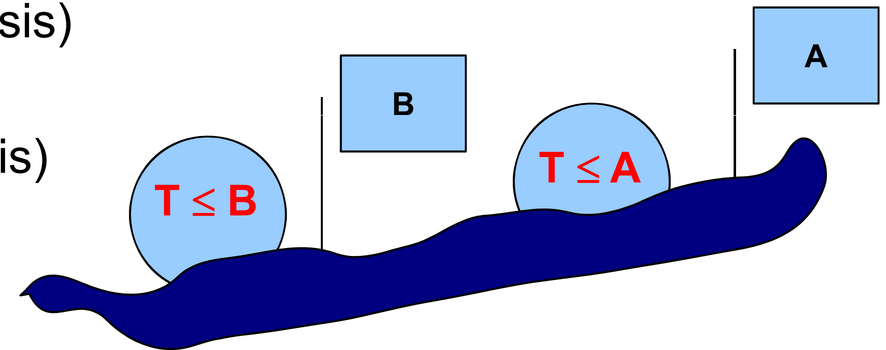
- Begriff „type-flow“ als solcher nicht gebräuchlich
  - „type-based flow analysis“
  - ...?

Definition

- Obergebiet: statische Programmanalyse
  - Compileroptimierung
- Teilgebiet und analytisches Hilfsmittel: Flussanalyse
  - siehe vorherige Vorträge zu CFA und DFA
- Fundament: Typsysteme
  - Halbordnung durch Subtypenbeziehung

Analogie zu Typfluss

- Flussanalyse
  - es fliesst eine „Substanz“ durch ein „System“
  - wohin muss sie fliessen? (must-analysis)
    - ⇨ Typgeneralisierung
  - wohin kann sie fliessen? (may-analysis)
    - ⇨ Typspezialisierung
  - Constraints!



- „System“
  - das Programm im statischen Zustand

- „Substanz“
  - in diesem Fall: Typen
  - Constraints durch Ordnung in der Typhierarchie

Analogie in Bezug auf CFA / DFA

- **CFA**
  - C = Control
  - „Kontrollfluss“
  - Kontrolle ist „Substanz“
- **DFA**
  - D = Data
  - „Datenfluss“
  - Daten sind „Substanz“
  - must-analysis: Available Expressions
  - may- analysis: Reaching Definitions

TFA besitzt nahe Verwandtschaft zu DFA

may-analysis basiert auf must-analysis!

Gliederung (Abschnitt 2)

1. was ist Typfluss?
   1. Definition
   2. Analogie zu Typfluss
   3. Analogie in Bezug auf CFA / DFA
2. **historische Ansätze**
3. Tip, Kiezun, Bäumer
   1. allgemeine Vorgehensweise
   2. Typrestriktionen
   3. Beispiel 1: Extract Interface
   4. Beispiel 2: Custom Library „Refactoring“

historische Ansätze

- Opdyke
  - Diplomarbeit zu Refactoring
  - aus den „Kindertagen“
  - fundamentale Aussagen

"After a refactoring, the type of each expression assigned to a variable must be an instance of the variable's defined type, or [..] one of its subtypes. [..]"

"the type of a variable can be changed by a refactoring, as long as each operation referenced on the variable is defined equivalently for its new type"

-- Opdyke, Refactoring object-oriented frameworks (1992)

historische Ansätze (2)

$$class = \mathtt{class}\ X\ \{dec_j\{exp_j\};\ type_i\ y_i;\ X(type'_1\ z_1, \ldots, type'_k\ x_k)\{exp\};\ |\ i \in M,\ j \in N\}$$

$$classType_1 = \mathtt{template}\langle|\rangle\mathtt{class}(X)\{\} \qquad TVars_0 = \{\} \qquad Classes \cup \{X \mapsto classType_1\}, TVars_{i-1} \vdash type_i \rightsquigarrow ptype_i \textbf{ with } TVars_i \text{ for } i \in M$$

$$TVars'_0 = TVars_m \qquad Classes \cup \{X \mapsto classType_1\}, TVars'_{j-1} \vdash dec_j \rightsquigarrow pdec_j \textbf{ with } (TVars'_j, Cons'_j) \text{ for } j \in N$$

$$TVars''_0 = TVars'_n \qquad Classes \cup \{X \mapsto classType_1\}, TVars''_{h-1} \vdash type'_h \rightsquigarrow ptype'_h \textbf{ with } TVars''_h \text{ for } h = 1, \ldots, k$$

$$classType_2 = \mathtt{template}\langle|\rangle\mathtt{class}(X)\{pdec_j\{\};\ ptype_i\ y_i;\ X(ptype'_1\ z_1, \ldots, ptype'_k\ z_k)\{\};\ |\ i \in M,\ j \in N\}$$

$$Classes \cup \{X \mapsto classType_2\}, InhHier, TVars'''_{j-1}, \{(this : X\langle\rangle)\} \vdash dec_j\{exp_j\} \rightsquigarrow pdec_j\{pexp_j\} \textbf{ with } (TVars'''_j, Cons'''_j) \text{ for } j \in N$$

Con

$$\frac{\text{call } E \equiv E_0.m(E_1, \cdots, E_n) \text{ to method } M \qquad RootDefs(M) = \{M_1, \cdots, M_k\},\ E'_i \equiv \text{Param}(M, i),\ 1 \leq i \leq n}{\begin{array}{lr} [E] = [M] & (4) \\ [E_i] \leq [E'_i] & (5) \\ [E_0] \leq Dcl(M_1) \text{ or } \cdots \text{ or } [E_0] \leq Dcl(M_k) & (6) \end{array}}$$

BASE)

- Tip, Kiezun, Bäumer (2003)
  - zugänglichere formale Darstellung
  - ebenfalls Java
  - Fokus dieses Vortrags!

Gliederung (Abschnitt 3)

1. was ist Typfluss?
   1. Definition
   2. Analogie zu Typfluss
   3. Analogie in Bezug auf CFA / DFA
2. historische Ansätze
3. **Tip, Kiezun, Bäumer**
   1. allgemeine Vorgehensweise
   2. Typrestriktionen
   3. Beispiel 1: Extract Interface
   4. Beispiel 2: Custom Library „Refactoring“

Tip, Kiezun, Bäumer

- allgemeine Vorgehensweise
- Constraints (Notation, Syntax, Generation)
- Extract Interface
  - Typgeneralisierung
- Custom Library „Refactoring“
  - Typspezialisierung
  - entgegen dem OOP-Prinzip => schwieriger zu motivieren!

Zur Erinnerung:

- **wohin muss sie fliessen? (must-analysis)**
  - ⇨ **Typgeneralisierung**
- **wohin kann sie fliessen? (may-analysis)**
  - ⇨ **Typspezialisierung**

Vorgehensweise

1. Generation von Typ-Constraints aus <u>typkorrektem(!)</u> Programmcode
2. Bilden von Ungleichungsketten durch Kombination der Constraints
3. Herleitung neuer Constraints und Wiederholung von 2. bis keine neuen Ergebnisse (Fixpunkt)
4. finale Constraints aus Ungleichungsketten ablesen

```
1    package mini;
2
3    class A {
4        void p() {}
5    }
6    class B extends A {
7        void q() {}
8    }
9
10   class C {
11       B m() {
12           B x = new B();     [new B()] ≤ [x]
13           x.p();             [x] ≤ A
14           return x; }        [x] ≤ [C.m()]
15       void n() {
16           B y = m();         [C.m()] ≤ [y]
17           y.q(); }           [y] ≤ B
18   }
```

Zuweisungen, Methodenaufrufe und Rückgabeanweisungen stellen Bedingungen an die Typen [E] der beteiligten Ausdrücke E:

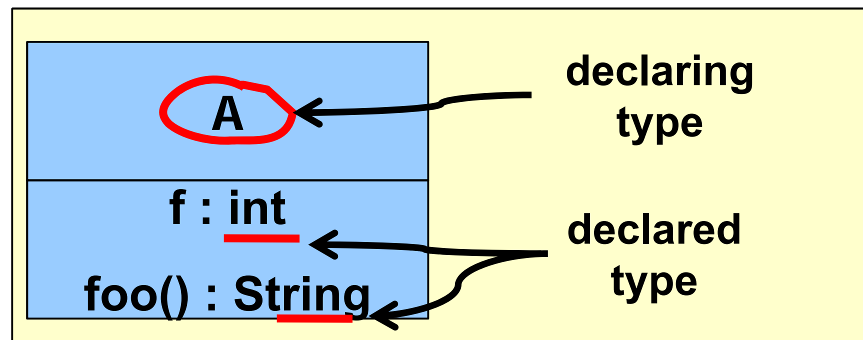
$$[x] \leq [C.m()] \leq [y] \leq B$$

praktische Anwendung

- Eclipse-Refactoring
  - **Extract Interface**
  - Pull Up Members
  - Generalize Declared Type
  - Infer Generic Type Arguments

Typrestriktionen (Notation)

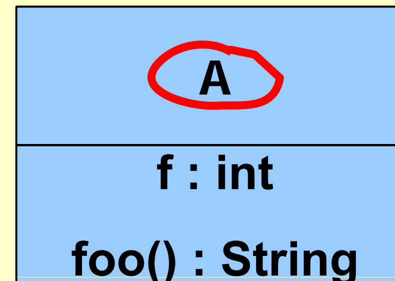
[E]	Typ des Ausdrucks E	declared type
[M]	deklarierter (Rückgabe-)Typ der Methode M	
[F]	deklarierter Typ des Feldes F	
Param(M,i)	deklarierter Typ des i - ten formalen Parameters der Methode M	
Decl(M)	Typ, der die Methode M enthält	declaring type
Decl(F)	Typ, der das Feld F enthält	



Typrestriktionen (Syntax)

$[E] = [E']$	Typen der Ausdrücke E und E' müssen identisch sein
$[E] < [E']$	Typ des Ausdrucks E muss ein echter Subtyp des Ausdrucks E' sein
$[E] \leq [E']$	entweder $[E] = [E']$ oder $[E] < [E']$

$[E] \equiv T$ Typ des Ausdrucks E ist definiert als T



Decl(A.f) ≡ A

Decl(A.foo()) ≡ A

[A.f] ≡ int

[A.foo()] ≡ String

$[E] \leq [E_1] \vee \ldots \vee [E] \leq [E_k]$

mindestens ein Constraint muss erfüllt sein (Disjunktion)

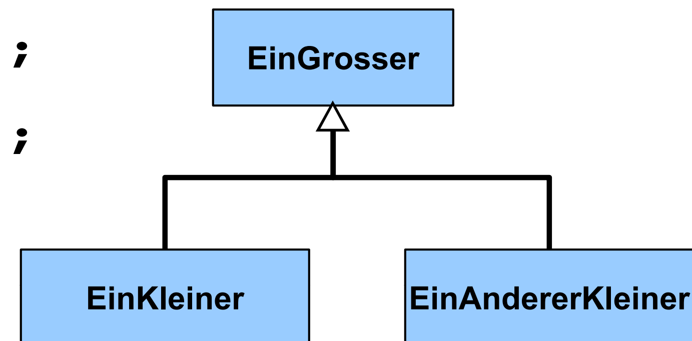
universitätbonn ROOTS

Typrestriktionen (Generation)

das Grundprinzip zur Generation von Constraints:

Zuweisung E1 = E2 ⟶ [E2] ≤ [E1]

```
EinGrosser derGrosse = new EinGrosser();
EinKleiner derKleine = new EinKleiner();
EinAndererKleiner derAndereKleine
 = new EinAndererKleiner();
```



`derGrosse = derKleine;`	[derKleine] ≤ [derGrosse]	ok!
`derKleine = (EinKleiner)derGrosse;`	EinKleiner ≤ [derKleine]	ok!
`derKleine = derAndereKleine;`	~~[derAndereKleine] ≤ [derKleine]~~	keine Chance!

Typrestriktionen (Generation)

Zuweisung $E_1 = E_2$	$[E_2] \leq [E_1]$
return E in Methode M	$[E] \leq [M]$
Deklaration T o	$[o] \equiv T$
Methode M in Klasse C	$Decl(M) \equiv C$
this in Methode M	$[this] \equiv Decl(M)$
Zugriff E.f auf Feld F	$[E.f] \equiv [F]$ $[E] \leq Decl(F)$
Aufruf $E.m(E_1 \vee \ldots \vee E_n)$ der Methode M (innerhalb der gleichen Klasse)	$[E.m(E_1 \vee \ldots \vee E_n)] \equiv [M]$ $[E_i] \leq [Param(M,i)]$ $[E] \leq Decl(M)$

Aufruf „virtueller“ Methode

M ist virtuell = M ist kein Konstruktor, nicht private und nicht static

<u>Mehrfachvererbung möglich!</u>

$[E] \leq Decl(M_1)$ v … v $[E] \leq Decl(M_k)$ mit $RootDefs(M) = \{M_1$ v … v $M_k\}$

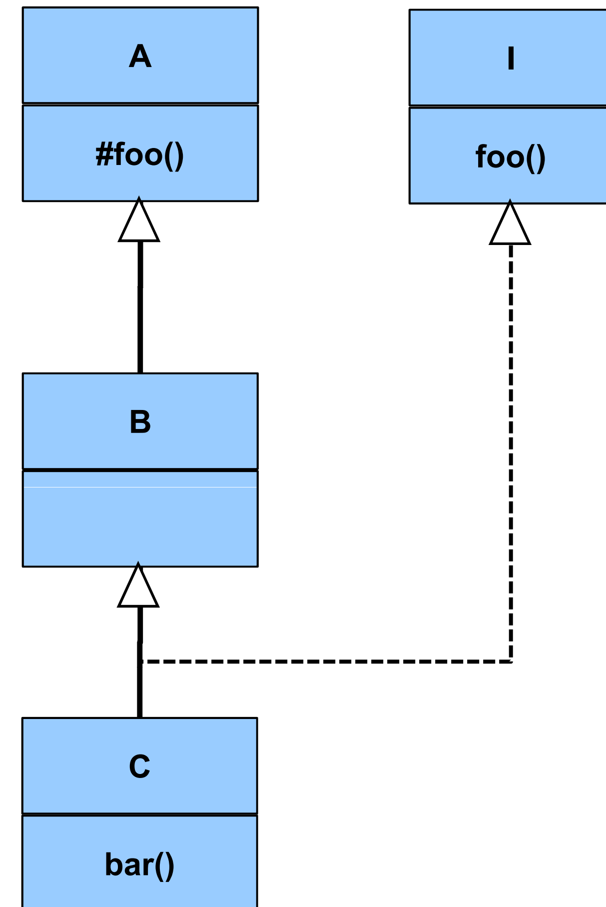
RootDefs(M) =

{ M’ | M überschreibt M’, und es ex. kein M’’ ≠ M’, so dass M’ dieses M’’ überschreibt }

sprich: die allgemeinste(n) Deklaration(en) einer nicht-lokalen Methode

Beispiel: RootDefs

```
class C {
  void bar() {
    foo();
  }
}
```



Beispiel: RootDefs (2)

```
class C {
  void bar() {
    this.foo();
  }
}
```

this in Methode M

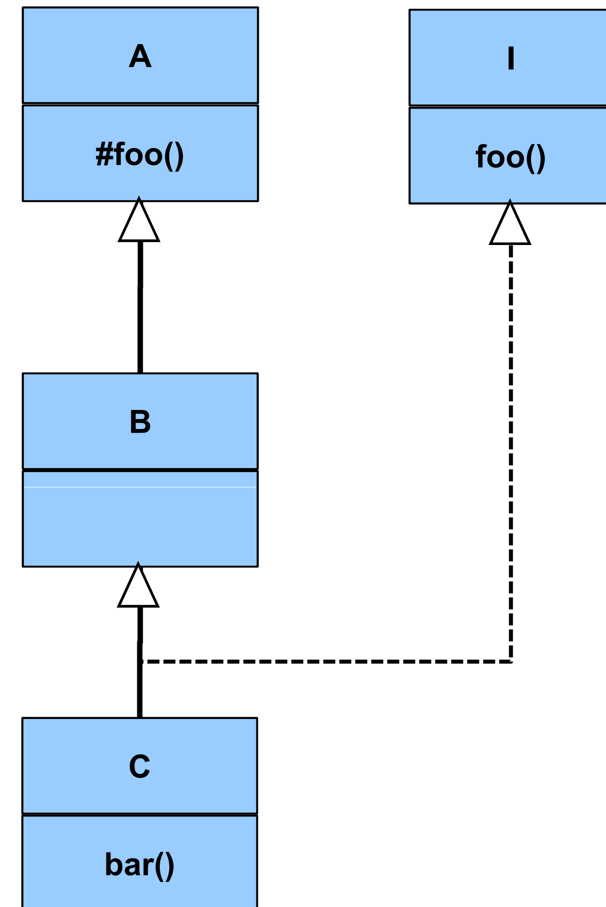
[this] ≡ Decl(M)

[this] ≡ Decl(C.bar()) ≡ C

C ≤ Decl(C.foo()) v C ≤ Decl(I.foo())

mit RootDefs(C.foo()) = {A, I}

Aufruf virtueller Methode



Überschreiben / Überdecken

falls Methode M' eine Methode M überschreibt:

[Param(M',i)] = [Param(M,i)]

[M'] ≤ [M] ***(Gleichheit vor Java 5.0)***

Decl(M') < Decl(M)

falls Feld F' ein Feld F überdeckt:

Decl(F') < Decl(F)

Casts

für einen Cast C(E) mit Klassen C und [E]:

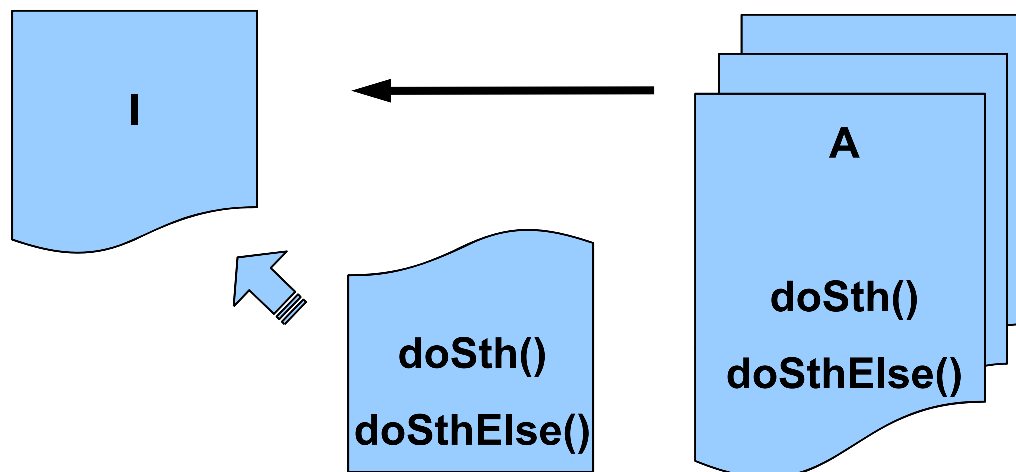
[(C)E] ≡ C

[E] ≤ [(C)E] oder [(C)E] ≤ [E]

- Unterscheidung Upcast/Downcast
  - **Upcast: [E] ≤ C   Downcast: C ≤ [E]**

Extract Interface

- Fowler's „Refactoring“
  - create an empty interface
  - declare the common operations in the interface
  - declare the relevant class(es) as implementing the interface



  - adjust client type declarations to use the interface

dabei hilft Typfluss-Analyse!

variable Constraints

- Refactoring-spezifisch!
- Aufteilung aller Constraints in 2 Mengen
  - $TC_{\textbf{fixed}}$
  - $TC_{\textbf{var}}$
    - ⇨ Festlegung vor Refactoring

$TC_{\textbf{fixed}}$ muss nach Programmtransformation unverändert sein!

Update-Algorithmus

- Input
  - die Menge **D** aller Deklarationselemente, die in Betracht gezogen werden
- Output
  - eine Menge **M** nicht ersetzbarer Deklarationselemente (zunächst leer)

1. finde **d** aus **D**, die aufgrund von Restriktionen nicht ersetzt werden können und füge jene **d** zu **M** hinzu
2. für alle **x** aus **M**, finde Constraints **[y] ≤ [x], [y] = [x], or [y] < [x]** und füge **y** zu **M** hinzu
3. iteriere über 2. bis die Menge **M** sich nicht mehr ändert (Fixpunkt erreicht)

Extract Interface Example 1/5

$TC_{var} = \{[o] \equiv List\}$

```
interface Bag {
  public Iterator iterator();
  public List add(Comparable e);
  public List addAll(List v0);
}

class List implements Bag {
  int size = 0; Comparable[] elems = new Comparable[10];
  public Iterator iterator(){ return new ListIterator(this); }
  public List add(Comparable e) {
    if (this.size + 1 == this.elems.length) {
      Comparable[] newElems = new Comparable[2 * this.size];
      System.arraycopy(this.elems, 0, newElems, 0, this.size);
      this.elems = newElems;
    }
    this.elems[this.size++] = e; return this;
  }
  public List addAll(List v1) {
    java.util.Iterator i = v1.iterator();
    for (; i.hasNext(); this.add((Comparable)i.next()));
    return this;
  }
  public void sort() { /* insertion sort */ }
}
```

```
interface Bag {
  public Iterator iterator(...
  public List add(Comparable e);
  public List addAll(List v0);
}

class List implements Bag {
  int size = 0; Comparable[] elems =
  public Iterator iterator(){ return
  public List add(Comparable e) {
    if (this.size + 1 == this.elems.length) {
      Compar                                   ];
      System                                   ze);
      this.e
    }
    this.ele
  }
  public Lis
    java.uti
    for (; i
    return t
  }
  public voi
}
```

[E] ≤ Decl(M) (Call)

[v1] ≤ Decl(Bag.iterator()) ≡ Bag

[v0] ≤ [v1]

[E_i] ≤ [Param(M,i)] (Parameterübergabe)

d(..)) ≡ List

(1.

l)

d(..)]

(return in M)

3. [List.add(..)] ≡ [Bag.add(..)] ≡ Bag

Das Gleiche für addAll

Extract Interface Example 3/5

```
class ListIterator implements java.util.Iterator {
  private int count = 0; private List v2;
  ListIterator(List v3){ v2 = v3; }
  public boolean hasNext(){ return this.count < this.v2.size; }
  public Object next(){ return this.v2.elems[this.count++]; }
}

public class Client {
  public static void main(String[] args) {
    List v4 = createList(); populate(v4); update(v4);
    sortList(v4); print(v4);
  }
  static List createList(){ return new List(); }
  static void populate(List v5){ v5.add("foo").add("bar"); }
  static void update(List v6) {
    List v7 = new List().add("zap").add("baz"); v6.addAll(v7);
  }
  static void sortList(List v8){ v8.sort(); }
  static void print(List v9) {
    for (Iterator iter = v9.iterator(); iter.hasNext();)
      System.out.println("Object: " + iter.next());
  }
}
```

Extract Interface Example 4/5

```
class ListIterator implements java.util.Iterator {
  private int count = 0; private List v2;
  ListIterator(List v3){ v2 = v3; }
  public boolean hasNext(){ return this.count < this.v2.size; }
  public Object next(){ return this.v2.elems[this.count++]; }
}

public class Client {
  public static void main(String[] args) {
    List v4 = createList(); populate(v4); update(v4);
    sortList(v4); print(v4);
  }
  static List createList(){ return new List(); }
  static void populate(List v5){ v5.add("foo").add("bar"); }
  static void update(List v6) {
    List v7 = new List().add("zap").add("baz"); v6.addAll(v7);
  }
  static void sortList(List v8){ v8.sort(); }
  static void print(List v9) {
    for (Iterator iter = v9.iterator(); iter.hasNext();)
      System.out.println("Object: " + iter.next());
  }
}
```

[v2] ≤ Decl(List.size) ≡ List

[v3] ≤ [v2]

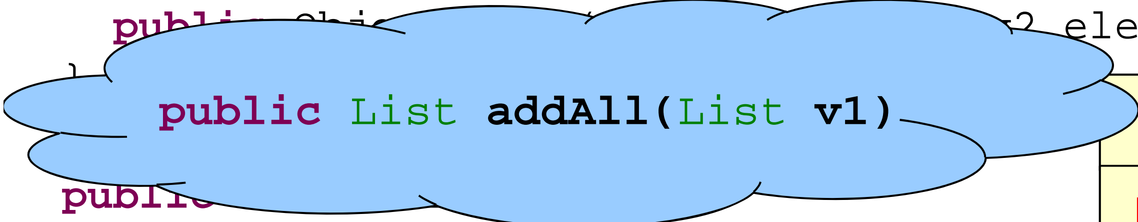
[v8] ≤ Decl(List.sort()) ≡ List

[v4] ≤ [v8]

[createList()] ≤ [v4]

Extract Interface Example 5/5

```
class ListIterator implements java.util.Iterator {
  private int count = 0; private List v2;
  ListIterator(List v3){ v2 = v3; }
  public boolean hasNext(){ return this.count < this.v2.size; }
  publ                                 2.elems[this.count++]; }
}
publ
  public static void main(String[] args) {
    List v4 = createList(); populate(v4); update(v4);
    sortList(v4); print(v4);
  }
  static List createList(){ return new List(); }
  static void populate(List v5){ v5.add("foo").add("bar"); }
  static void update(List v6) {
    List v7 = new List().add("zap").add("baz"); v6.addAll(v7);
  }
  static void sortList(List v8){ v8.sort(); }
  static void print(List v9) {
    for (Iterator iter = v9.iterator(); iter.hasNext();)
      System.out.println("Object: " + iter.next());
  }
}
```



[v5] ≤ Decl(Bag.add(..)) ≡ Bag

[v6] ≤ Decl(Bag.addAll(..)) ≡ Bag

[v7] ≤ [v1] ≤ Bag

[v9] ≤ Decl(iterator()) ≡ Bag

Anwendung des Update-Algorithmus

1. [v2] ≤ List und [v8] ≤ List => M = {v2, v8}
2. Zwei Iterationen:
   1. [v3] ≤ [v2], [v4] ≤ [v8] => M = {v2, v3, v4, v8}
   2. [createList()] ≤ [v4] => M = {v2, v3, v4, v8, createList()}

finale Ungleichungsketten für alle nicht ersetzbaren Deklarationselemente:

[v3] ≤ [v2] ≤ List

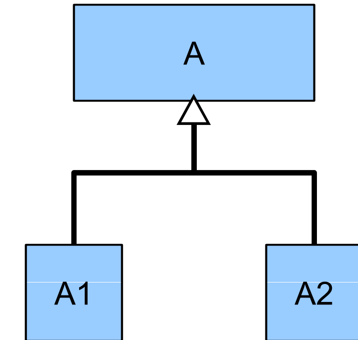
[createList()] ≤ [v4] ≤ [v8] ≤ List

Custom Library „Refactoring“

- Typspezialisierung
- kein Refactoring aus dem Fowler-Katalog
- Motivation
  - eigene Klassen (Bibliotheken) verwenden
    - ⇨ spezieller (untypischer) Anwendungsbereich
    - ⇨ neue Klassen sind Subklassen der gleichen Superklasse wie jene, die angepasst werden soll
  - **nicht** vererbt von Originalklassen
    - ⇨ nicht verwendete Methoden „zuhause lassen“
    - ⇨ weniger Ballast
    - ⇨ weniger Fehlerquellen
    - ⇨ aber auch: weniger Wiederverwendbarkeit

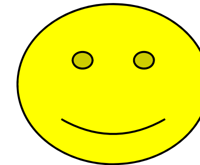
Kriterien

- **statische Typen** (Deklaration) => minimale Restriktion
  - linke Seite von Zuweisungen
  - Einhalten von Typkorrektheit




```
A1 a1 = new A1();
A2 a2 = new A2();
      ...
   a1 = a2;
```

```
A a1 = new A1();
A a2 = new A2();
      ...
   a1 = a2;
```

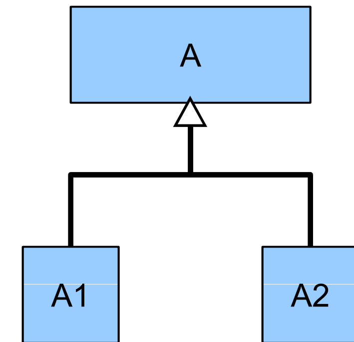


- **„dynamische“ Typen** (Allokation) => maximale Restriktion
  - rechte Seite von Zuweisungen
  - Spezialisierung!

- Custom Library verwenden wann immer möglich

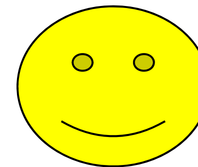
Kriterien (2)

- Casts
  - maximale Restriktion
  - Verhalten bewahren (Cast erfolgreich / nicht erfolgreich)




```
A a = new A2();
A1 a1 = new A1();
...
A2 a2 = (A2)a1;
```

```
A a = new A2();
A1 a1 = new A1();
...
A2 a2 = (A2)a;
```

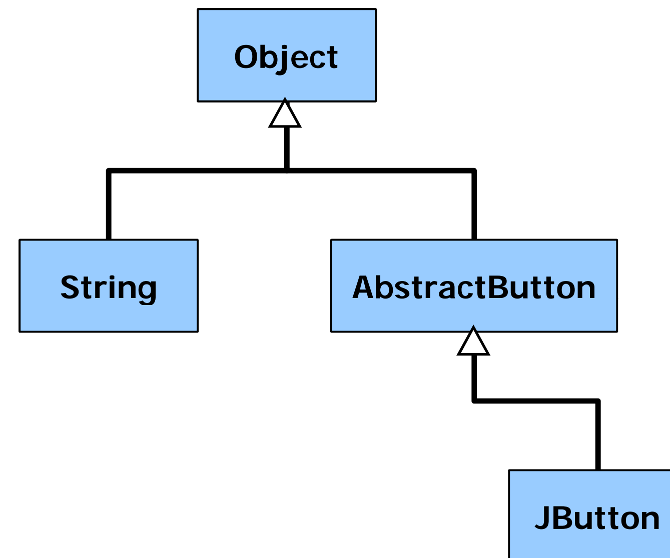


universitätbonn ROOTS

Customize Library

```
class Example {
  void foo() {
    JToolbar jt = new JToolbar();
    JButton jb1 = new JButton();
    PropertyChangeListener pcl
       = jt.createActionChangeListener(jb1);
    JButton jb2 = new JButton();
    JButton jb3 = new JButton();
    jb2.doClick();
    bar(jb3);
    bar("CRASH");
    jb3 = jb2;
    // ...
  }

  @Override
    bar(Object o) {
      JButton jb = (JButton)o;
      jb.updateUI();
  }
  // ...
}
```

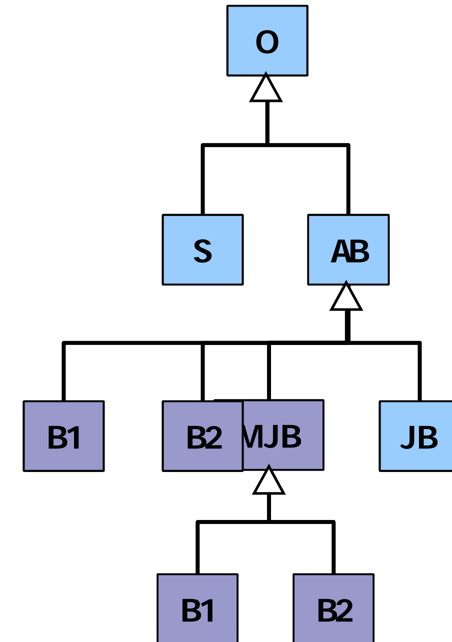


Customize Library (3)

```
class Example {
  void foo() {
    JToolbar jt = new JToolbar();
    JB jb1 = new JB();
    PropertyChangeListener pcl
       = jt.createActionChangeListener(jb1);
    MJB jb2 = new B1();
    MJB jb3 = new B2();
    jb2.doClick();
    bar(jb3);
    bar("CRASH");
    jb3 = jb2;
    // ...
  }

  @Override
    bar(O o) {
      JB jb = (JB)o;
      jb.updateUI();
  }
  // ...
}
```

Zuweisungsfehler!



Customize Library (4)

```
     class Example {
       void foo() {
         JToolbar jt = new JToolbar();
1)       S1 jb1 = new D1();
2)       PropertyChangeListener pcl
            = jt.createActionChangeListener(jb1);
3)       S2 jb2 = new D2();
4)       S3 jb3 = new D3();
5)       jb2.doClick();
6)       bar(jb3);
7)       bar("CRASH");
8)       jb3 = jb2;
         // ...
       }

       @Override
         bar(S4 o) {
9)         S5 jb = (C1)o;
10)        jb.updateUI();
       }
       // ...
     }
```

1) $D1 \leq S1$

2) $S1 \leq JB$

3) $D2 \leq S2$

4) $D3 \leq S3$

5) $S2 \leq AB$

6) $S3 \leq S4$

7) $S \leq S4$

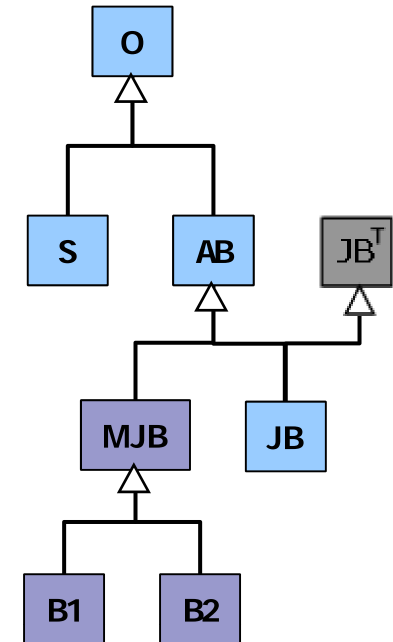
8) $S2 \leq S3$

9a) $C1 \leq S4$ v ~~$S4 \leq C1$~~

9b) $C1 \leq S5$

10) ~~$S5 \leq JB$ v $S5 \leq MJB$~~

$S5 \leq JB^T$



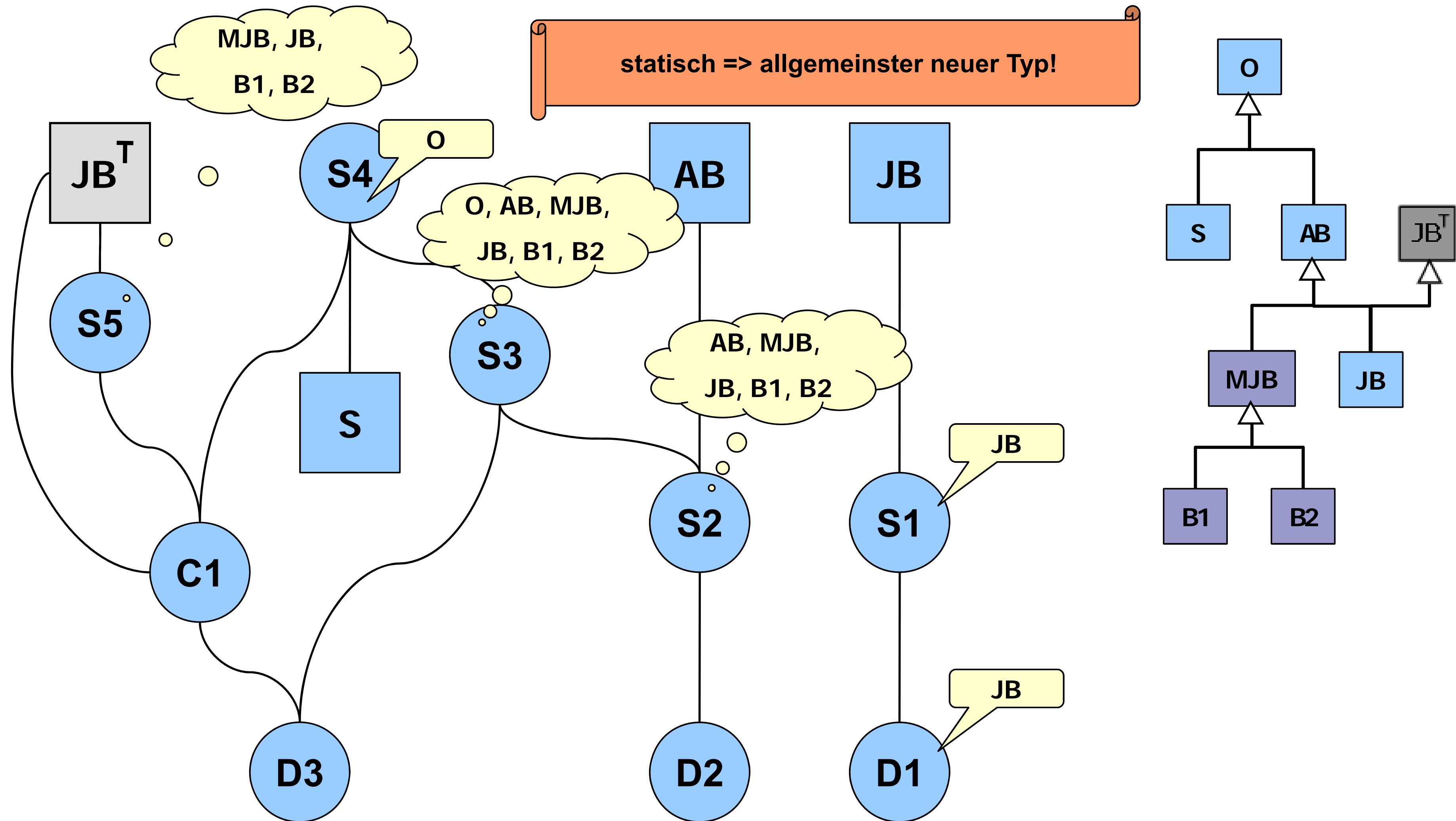
Cast-Verhalten von zuvor

$D3 \leq C1$

$S \;)(\; C1$

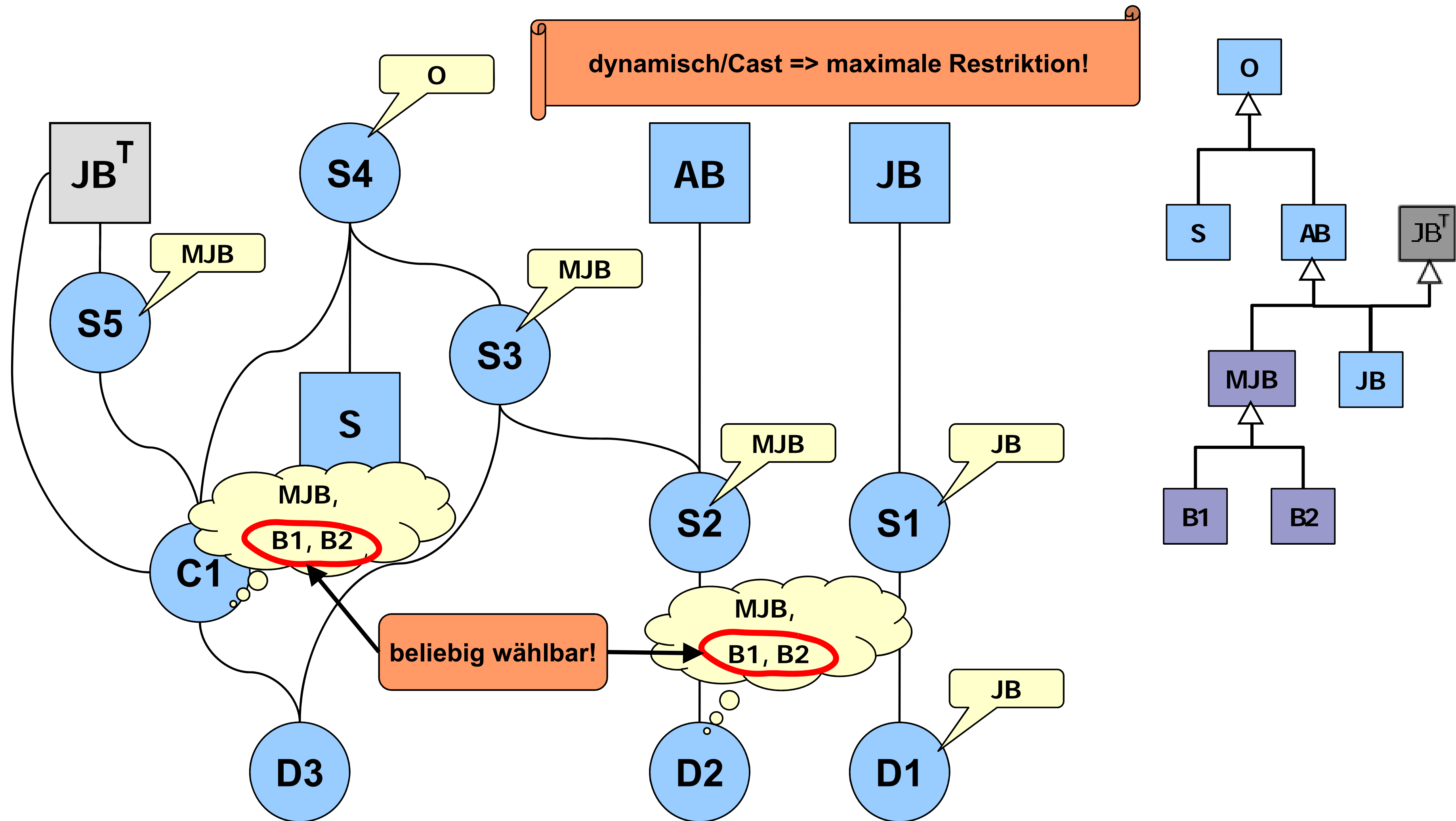
$c1 \leq JB^T$

Auflösen durch Constraints



universitätbonn ROOTS

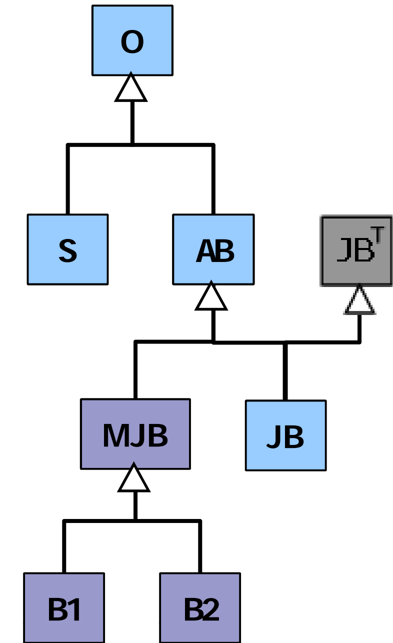
Auflösen durch Constraints (2)



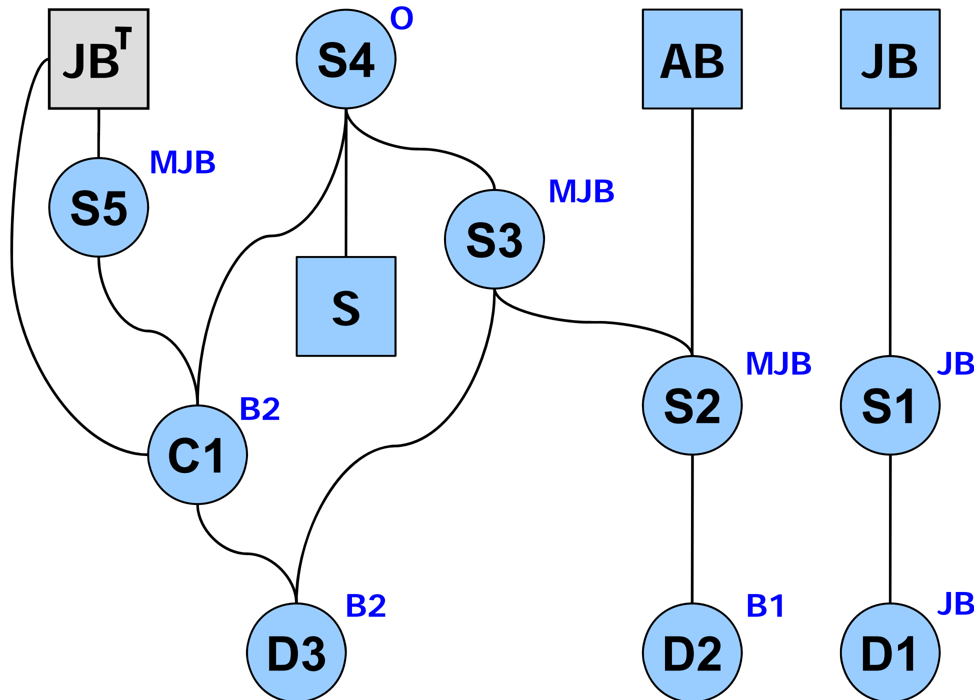
universitätbonn ROOTS

Auflösen durch Constraints (3)

JB^T
O
S4
AB
JB
MJB
S5
MJB
S3
S
MJB
JB
B2
S2
S1
C1
B1
JB
B2
D3
D2
D1



modifizierter Code




```
class Example {
  void foo() {
    ...
    JB jb1 = new JB();
    ...
    MJB jb2 = new B1();
    MJB jb3 = new B2();
    jb2.doClick();
    bar(jb3);
    bar("CRASH");
    jb3 = jb2;
    // ...
  }

  @Override
    bar(O o) {
      MJB jb = (B2)o;
      jb.updateUI();
  }
  // ...
}
```

Zusammenfassung

- Definition und Einordnung von Typfluss
  - Flussanalyse basierend auf Typsystemen
- kurze Zeitlinie
  - Opdyke, (Seguin), Duggan, Tip/Kiezun/Bäumer
- Tip, Kiezun, Bäumer
  - vereinfachte Definitionen und Grundaufbau des Formalismus
  - Beispiel: Extract Interface
  - Beispiel: Customize Library

Literatur

- [DUG99] *Modular type-based reverse engineering of parameterized types in Java code*, ACM SIGPLAN Notices, 34(10), pp. 97-113, October 1999.
- [FOW99] *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, August 1999.
- [OPD92] Opdyke, *Refactoring object-oriented frameworks*, 1992
- [SEG00] Seguin, C. *Refactoring tool challenges in a strongly typed language*. In OOPSLA'00 Companion (October 2000), pp. 101–102.
- [SKR07] D. Speicher, G. Kniesel, T. Rho. *JTransformer – Eine logikbasierte Infrastruktur zur Codeanalyse,* Institut für Informatik III, Universität Bonn: http://www.iai.uni-bonn.de/~gk/papers/speicherKniesel-wsr2007.pdf
- [TKB03] F. Tip, A. Kiezun, D. Bäumer. *Refactoring for generalization using type constraints*, ACM SIGPLAN Notices, 38(11), pp. 13-26, November 2003.
- [TIP04] F. Tip, *Applications of Type Constraints in Software Engineering Tools,* IBM T.J. Watson Research Center: http://www.brics.dk/MC/04/TypeConstraints/