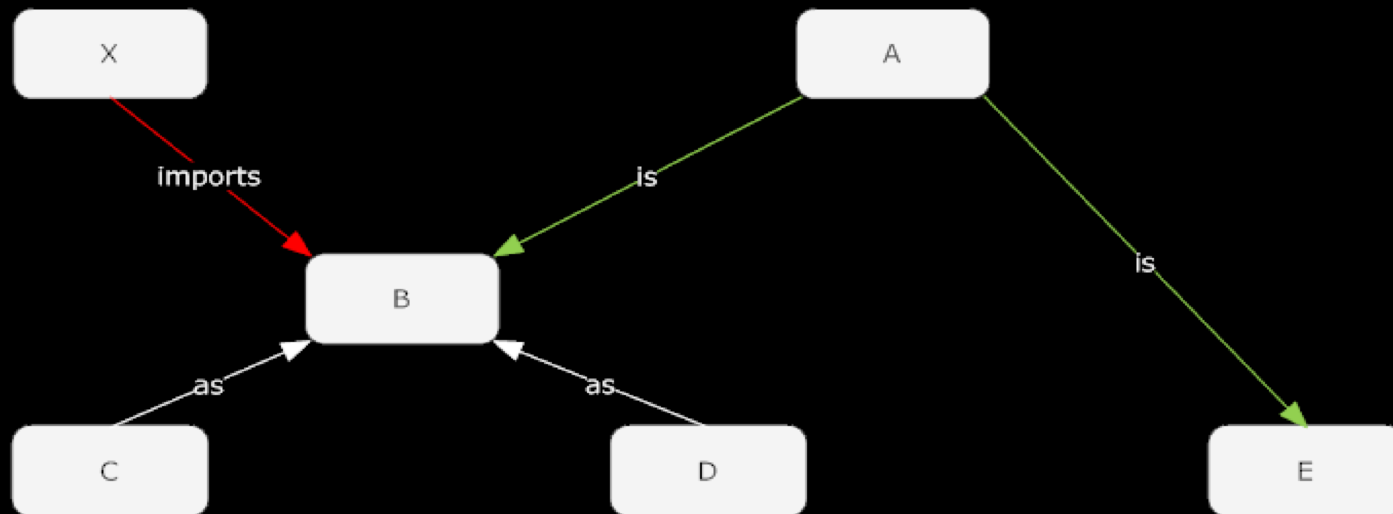
Formale Spezialisierungstechniken

am Beispiel des binären Baums.

Hybride Programmiersprachen
Daniel Krompass
Berlin, 2009

CHELSEA

Spezialisierungsarten (Typbeziehungen *erster Art*)



- *X* stellt Methoden und Eigenschaften bereit, die *B* übernimmt, ohne eine Typbeziehung einzugehen.
- *B* stellt Methoden und Eigenschaften bereit, die *C* und *D* übernehmen.
- *C* und *D* gehen mit *B* im Sinne der *Vererbung* eine herkömmliche Typbeziehung ein.
- *A* gibt Eigenschaften und Methoden vor, oder stellt diese unmittelbar zur Verfügung.
- *B* und *E* gehen mit *A* im Sinne einer *Schnittstelle* eine herkömmliche Typbeziehung ein.
- Mehrfachbeziehungen zwischen Typen (*Mehrfachvererbung*) sind möglich.

- *Innovation:* Typbeziehungen *zweiter Art* werden unmittelbar durch den Programmierer definiert.

Definition einer binären Baumstruktur

```
public static type Node is Tree, NoTree;
public static type Tree Node, Int, Node;
public static type NoTree;
```

- Jeder Basistyp identifiziert sich durch ein optionales Argument – dem Exponenten.

- Mit Konstruktoren erhalten wir:

```
public static type Tree Node, Int, Node
  public func ctor i:=Int => Tree
    return new Tree NoTree, i, NoTree;

public static type Node is Tree, NoTree;

public static type NoTree
  public func ctor => NoTree
    return new NoTree;
```

Spezialisierung

```
public static type XTree Node, XInt, Node as Tree;
```

- Das neue Argument muss die Eigenschaften des alten in Typ und Dimension erfüllen.
- Wir betrachten hierfür erneut:

```
public static type Tree Node, Int, Node
  public func ctor i:=Int => Tree
    return new Tree NoTree, i, NoTree;
```

- Problem: Der Konstruktor würde nach wie vor nur ein *Int* akzeptieren.
- Ferner würde dieser auch nicht den Typ *XTree* zurückgeben.

- Primitive Lösung: Wir überschreiben den Konstruktor.

- Schlaue Lösung: Wir parametrisieren Basis- (*Tree*) und Elementtyp (*Int*).
- Dabei definiert man «*I* entspreche *Int*» bzw. «*T* entspreche *Tree*».

CHELSEA

Typparametrisierung

```
public static type T:=Tree Node, I:=Int, Node
   public func ctor i:=I => T
      return new T NoTree, i, NoTree;

public static type Node is Tree, NoTree;

public static type NT:=NoTree
   public func ctor => NT
      return new NT;
```

- Nun stellt sich der Konstruktor von *XTree* auf *XInt* ein. Analoges gilt auch für *NoTree*.
- Allgemein lassen sich unterschiedliche Bindungsfunktionalitäten aus JAVA und C# realisieren.

- Problem: *Node* kann sich nicht auf *T* einstellen, d.h. die Spezialisierung explizit fordern.

- Die Lösung: Typunterordnung.

Unterordnung von Typdefinitionen

```
public static type T:=Tree T.Node, I:=Int, T.Node
  public func ctor i:=I => T
    return new T T.NoTree, i, T.NoTree;

  public static type Node is T, T.NoTree;

  public static type NT:=NoTree
    public func ctor => T.NT
      return new T.NT;
```

- *Node* stellt sich nun auf *T* ein.
- Die Generizität der Baumstruktur ist vollständig definiert.
- Hohe Abstraktion im Sinne der Strukturpolymorphie ist gegenwärtig essentiell.

Vergleichbarkeitseigenschaft fordern

```
public static type T:=Tree T.Node, C:=Comparable, T.Node
  public func ctor c:=C => T
    return new T T.NoTree, c, T.NoTree;

  public static type Node is T, T.NoTree;

  public static type NT:=NoTree
    public func ctor => T.NT
      return new T.NT;
```

- Die Baumstruktur ist bezüglich *Comparable* polymorph in seinen Elementen.
- Typparameter sind fortan bekannt. Symmetrische Eigenschaften der Teilbäume sind daher wie folgt definierbar.

```
public static type T:=Tree N:=T.Node, Comparable, N;
```

Spezialisierte Baumstruktur

```
public static type T:=IntTree T.Node, Int, T.Node as Tree;

public static type T:=StringTree T.Node, String, T.Node as Tree;

public static type T:=VectorTree T.Node, Vector, T.Node as Tree;
```

- Die definierten Baumstrukturen unterscheiden sich in ihrem Elementtyp.
- Die Vergleichbarkeit von Vektoren ist z.B. durch die euklidische Norm gegeben.

CHELSEA

Elemente hinzufügen

```
public static type T:=Tree l:=T.Node, (c,C):=Comparable, r:=T.Node

  public func ctor c:=C => T
    return new T T.NoTree, c, T.NoTree;

  public proc Add c':=C
    if c' <= c then l.Add c' else r.Add c';

  public static type Node is T, T.NoTree;
    def public proc Add C;

  public static type (nt,NT):=NoTree

    public func ctor => T.NT
      return new T.NT;

    public proc Add c':=C
      nt = T c';
```

Ordnungsrelation parametrisieren

```
public static type T:=Tree
O:=T.Relation, l:=T.Node, (c,C):=Comparable, r:=T.Node

  public static type Relation is C.(<=), C.(>=);

  public func ctor c:=C => T
    return new T T.NoTree, c, T.NoTree;

  public proc Add c':=C
    if c'.O c then l.Add c' else r.Add c';

  public static type Node is T, T.NoTree;
    def public proc Add C;

  public static type (nt,NT):=NoTree

    public func ctor => T.NT
      return new T.NT;

    public proc Add c':=C
      nt = T c';
```

Spezialisierte Baumstruktur

```
public static type T:=IntTree
Int.(<=), T.Node, Int, T.Node as Tree;

public static type T:=StringTree
String.(>=), T.Node, String, T.Node as Tree;

public static type T:=VectorTree
Vector.(>=), T.Node, Vector, T.Node as Tree;
```

- Die definierten Baumstrukturen unterscheiden sich in ihrem Elementtyp und in ihrer Ordnung.