

# Familienpolymorphismus

Michael Schnell

22. Januar 2006

## Einleitung

Familien

Traditioneller Polymorphismus

Familienpolymorphismus

## Graph mit traditionellem Polymorphismus

Implementation in Java

Implementation in Java mit Generics

## Graph mit Familienpolymorphismus

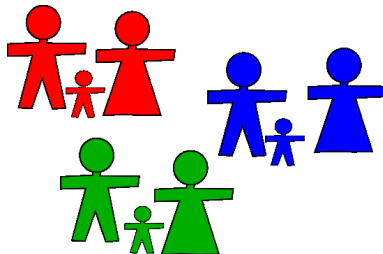
gbeta

Implementation in gbeta

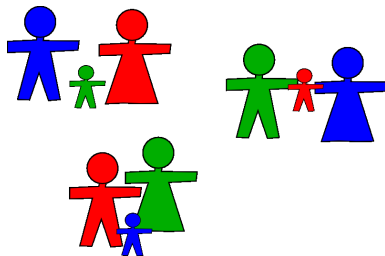
Datenstruktur, Graph erweitern

# Familien

bestehen aus einem Vater, einer Mutter und einem Kind



# Familien?



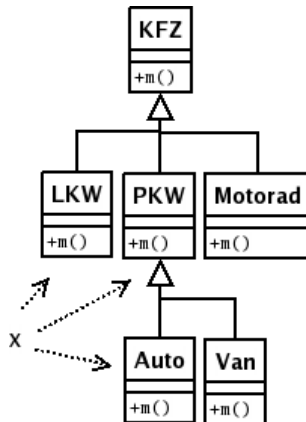
- ▶ „Ein Vater“, „eine Mutter“ und „ein Kind“ gehören nicht unbedingt zur selben Familie!

# Klassenfamilien

- ▶ Menge von Klassen (Familienmitglieder)
- ▶ Stehen zueinander in Relation
- ▶ Mischen soll verhindert werden

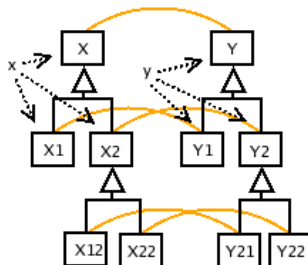
# Vererbung, Polymorphismus, späte Bindung, ein Objekt

- ▶ Polymorphe Referenz  $x$
- ▶  $x.m()$  Methode  $m$  von  $x$
- ▶ Späte Bindung: Das richtige  $m$  wird aufgerufen
- ▶ Statische Typprüfung:  $m$  existiert
- ▶ " $x.m()$ " für alle  $x$  und  $m$
- ▶ Anzahl der Klassen und Methoden unbegrenzt



# Vererbung, Polymorphismus, späte Bindung, zwei Objekte

- ▶ `x.m(y);`
- ▶ `X x; Y y;`
- ▶ Zur Laufzeit referenzieren
  - ▶ `x` Instanz von `X`
  - ▶ `y` Instanz von `Y`
 oder jeweils von Subklassen
- ▶ Keine Subtyppaarungen zur Compilezeit festlegbar.



# Traditioneller Polymorphismus

- ▶ Klassenfamilien nicht explizit
- ▶ Ableiten nur von einzelnen Klassen, nicht von Familien
- ▶ Kann Typsicherheit und Flexibilität mit Familien nicht gleichzeitig bieten

# Familienpolymorphismus

- ▶ Generalisierte Version des Polymorphismus
- ▶ Familien explizit definierbar
  - ▶ Eine Menge von Klassen bildet eine Familie
  - ▶ Mitglieder haben festgelegte Relationen untereinander
- ▶ Ableiten von ganzen Familien möglich
- ▶ Bietet Typsicherheit und Flexibilität mit Familien

# Beispiel Familien: Graph, OnOffGraph

- ▶ Graph: Menge von Knoten und Kanten. Kanten verbinden die Knoten zu einem Graphen.
- ▶ OnOffGraph: wie ein Graph, aber vorhandene Kanten können ausgeschaltet werden.
- ▶ Gewichtete Graphen
- ▶ Beschriftete Graphen
- ▶ Gerichtete Graphen
- ▶ ...

# Graph

```
class Node {  
    boolean touches(Edge e) {  
        return (this == e.n1) || (this==e.n2);  
    }  
}  
  
class Edge { Node n1, n2; }
```

# OnOffGraph

```
class OnOffNode extends Node {
    boolean touches(Edge e) {
        return ((OnOffEdge)e).enabled?
            super.touches(e) : false;
    }
}

class OnOffEdge extends Edge {
    boolean enabled;
    OnOffEdge() {this.enabled = false; }
}
```

# Hauptprogramm

```
public class Main {
    static void build(Node n, Edge e, boolean b){
        e.n1 = e.n2 = n;
        if(b == n.touches(e)) System.out.println("OK");
        else System.out.println("not OK");
    }
    public static void main(String[] args){
        build(new Node(), new Edge(), true);
        build(new OnOffNode(), new OnOffEdge(), false);
        build(new OnOffNode(), new Edge(), true);
        // ClassCastException!
        build(new Node(), new OnOffEdge(), true); // "Works"
    }
}
```

# Eigenschaften

- ▶ Flexibilität ist gegeben
  - ▶ `build` arbeitet mit den Familien `Graph`, `OnOffGraph`
  - ▶ auch mit neuen statisch nicht bekannten Familien
- ▶ Typsicherheit ist nicht gegeben
  - ▶ Typüberprüfung zur Compilezeit nicht vollständig
  - ▶ Familien mischen möglich (3.,4. Aufruf von `build`)
  - ▶ keine Compilezeitfehler
  - ▶ Exception zur Laufzeit durch dynamischen Cast
  - ▶ Vierter Aufruf verursacht evtl. erst viel später Laufzeitfehler
- ▶ Innere Klassen würden die Eigenschaften nicht ändern
- ▶ `build` wiederverwendbar, aber nicht typsicher

# Graph

```
class NodeF<N extends NodeF<N,E>, E extends EdgeF<N,E>> {
    public boolean touches(E e){
        return (this == e.n1) || (this == e.n2);
    }
}
class EdgeF<N extends NodeF<N,E>, E extends EdgeF<N,E>> {
    public N n1, n2;
}
class Node extends NodeF<Node, Edge> {}
class Edge extends EdgeF<Node, Edge> {}
```

# OnOffGraph, Node

```
class OnOffNodeF<ON extends OnOffNodeF<ON,OE>,
                OE extends OnOffEdgeF<ON,OE>>
    extends NodeF<ON, OE> {
    public boolean touches(OE e){
        return e.enabled? super.touches(e) : false;
    }
}

class OnOffNode extends OnOffNodeF<OnOffNode, OnOffEdge> {
```

# OnOffGraph, Edge

```
class OnOffEdgeF<ON extends OnOffNodeF<ON,OE>,
                OE extends OnOffEdgeF<ON,OE>>
    extends EdgeF<ON,OE> {
    boolean enabled;
    public OnOffEdgeF(){
        enabled = false;
    }
}

class OnOffEdge extends OnOffEdgeF<OnOffNode, OnOffEdge> {
```

# Main, build

```
public class GenericsMain {
    static void build(Node n, Edge e, boolean b){
        e.n1 = e.n2 = n;
        if(b == n.touches(e)) System.out.println("OK");
        else System.err.println("nicht OK");
    }
    static void build(OnOffNode n, OnOffEdge e, boolean b){
        e.n1 = e.n2 = n;
        if(b == n.touches(e)) System.out.println("OK");
        else System.err.println("nicht OK");
    }
}
```

# Main, main

```
public static void main(String[] args){
    build(new Node(), new Edge(), true);
    build(new OnOffNode(), new OnOffEdge(), false);

    // type errors:
    // build(new OnOffNode(), new Edge(), false);
    // build(new Node(), new OnOffEdge(), false);
}
}
```

# Eigenschaften

- ▶ Typsicherheit ist gegeben.
  - ▶ Parameter der `build`-Methoden nicht direkt verwandt
  - ▶ Vermischen von Familien verursacht Typfehler zur Compilezeit
- ▶ Flexibilität ist nicht gegeben.
  - ▶ Je Familie eine eigene `build`-Methode
- ▶ Typsicherheit auf Kosten von Wiederverwendbarkeit erreicht.

# parametrisierte build-Methode

```
static <N extends NodeF<N,E>, E extends EdgeF<N,E>>  
    void build(N n, E e, boolean b) {  
    e.n1 = e.n2 = n;  
    if(b == n.touches(e)) System.out.println("OK");  
    else System.err.println("nicht OK");  
}
```

# Eigenschaften der parametrisierten build-Methode

- ▶ Typsicherheit ist weiterhin gegeben.
- ▶ Wiederverwendbarkeit scheint gegeben.
- ▶ Problem Erweiterbarkeit
  - ▶ Alle Methoden parametrisieren
  - ▶ Über alle Familienmitglieder parametrisieren
  - ▶ Auch wenn sie nicht gebraucht werden!
  - ▶ Hinzufügen neuer Familienmitglieder
- ▶ Flexibilität also nicht gegeben.

# Statische Typsicherheit oder Flexibilität

Zwang zur Wahl zwischen

- ▶ Statische Typsicherheit
- ▶ Flexibilität
  - ▶ Code wiederverwenden
  - ▶ Familien erweitern

# Klassenfamilien

## Neue Konzepte nötig

- ▶ Familien explizit
- ▶ Klassen als Attribute von Objekten
  - ▶ (Familien-)Objekt ist Repository für eine Menge von Klassen
- ▶ Späte Bindung für diese Klassenattribute
  - ▶ Mehrere Familien durch einfaches Ableiten möglich.
- ▶ Relationen zwischen den Mitgliedern statisch bekannt.
- ▶ Methodenimplementation: Exakte Familie statisch zu kennen nicht nötig.
- ▶ gbeta

## gbeta

- ▶ Generalisierte Version von BETA
- ▶ Klassen, Methoden und mehr sind alles Pattern
- ▶ (# foo: (# #) #)

## Beispiel, Pattern als Klassen

- ▶ Dynamische Referenz mit ^
- ▶ Statische Referenz mit @

```
(#  
  foo: (# b: @boolean #);  
  bar: (#  
    fo: ^foo;  
    c: @foo  
  #);  
  f: @foo  
#)
```

## Beispiel, Pattern als Methode

- ▶ enter Parameter
- ▶ do imperative Anweisungen
- ▶ exit „Rückgabewert“

```
quad: (#  
  r: @real;  
  enter r  
  exit r*r  
#);
```

- ▶ (4)->quad->r
- ▶ exit this(Patternname) [] für Klassen nötig
- ▶ „new“-Operator implizit

## virtuelle Pattern

```

(#
  P:      (# Q:<(# i:@integer
            do 'Hallo\n' -> stdio   #) #);
  P2 : P (# Q:: <!(# r:@real
            do 'Welt\n' -> stdio   #) #);

  p: ^ P;
  pq: @p.Q;
do
  &P [] -> p [];
  p.Q;
  &P2 [] -> p [];
  p.Q;
#)

```

# Graph

```

Graph: (#
  Node: <(#
    touches: <(#
      e: ^Edge; b: @boolean;
      enter e[]
      do (this(Node)=e.n1) or (this(Node)=e.n2) -> b
      exit b
    #);
    exit this(Node) []
  #);
  Edge: <(#
    n1: ^Node; n2: ^Node;
    exit this(Edge) []
  #);#);

```

# OnOffGraph

```
OnOffGraph: Graph (#
  Node:: <(#
    touches:: <!(#
      do (if e.enabled then INNER if)
    #)
  #);
Edge:: <(#
  enabled: @boolean;
#);
#);
```

## build

```
build: (#
  g: < @Graph;
  n: ^ g.Node;
  e: ^ g.Edge;
  b: @ boolean;
  enter (n[], e[], b)
  do
    n->e.n1[]->e.n2[];
    (if (e->n.touches)=b then 'OK'->putline
      else 'Nicht OK!' -> putline if)
#);
```

# Hauptpattern

```
(#  
  putline: (# enter stdio do '\n' -> stdio; #);  
  Graph: (# ... #);  
  OnOffGraph: Graph (# ... #);  
  build: (# ... #);  
  
  g1 : @Graph;  
  g2 : @OnOffGraph;  
do  
  (g1.Node, g1.Edge, true) -> build(#g::@g1#);  
  (g2.Node, g2.Edge, false) -> build(#g::@g2#);  
#)
```

# Eigenschaften

- ▶ Familien explizit definiert (Graph, OnOffGraph)
- ▶ Jede Instanz hat zwei Attribute: Node, Edge
- ▶ `g1.Node` und `g1.Edge` sind aus der selbem Familie
- ▶ `g1.Node` und `g2.Node` haben **nicht** den gleichen Typ
  - ▶ außer: `g1` und `g2` sind statisch das selbe Objekt
- ▶ Nicht nötig die exakten Klassen statisch zu kennen
- ▶ Relationen zwischen beteiligten Klassen durch das Familienobjekt gegeben

# Beispiel einer Datenstruktur

```
NodesAndEdges: (#  
  g: < @ Graph;  
  nodes: @list(# element::g.Node #);  
  edges: @list(# element::g.Edge #);  
#)
```

- ▶ Datenstruktur, enthält
  - ▶ Liste von Knoten
  - ▶ Liste von Kantender selben Familie (Familienobjekt g)

## Datenstruktur benutzen

```
myGraph: @LabelledGraph;  
myNodesAndEdges: @NodesAndEdges(# g::@myGraph #);  
  
listBuild: (#  
  ne: < @NodesAndEdges;  
  n: ^ ne.g.Node; e: ^ ne.g.Edge;  
do ne.nodes.head-> n[];  
  ne.edges.head-> e[];  
  (n,e,true) -> build(#g::@ne.g#)  
#)
```

- ▶ Eindeutiges Familienobjekt (ne.g)
- ▶ Datenstruktur kann ohne statische Abhängigkeiten durch eine Aufrufskette weitergereicht werden

# Neues Familienmitglied

```
Graph: (#  
  Node: <(# ... #);  
  Edge: <(# ... #);  
  NuE: <(#  
    n: ^Node;  
    e: ^Edge;  
    exit this(NuE) []  
  #)  
#);
```

- ▶ build muss nicht geändert werden!

# Typsystem

Bis vor kurzem




- ▶ Kein formales Typsystem
- ▶ Kein Korrektheitsbeweis

Erst seit 2006 (POPL'06)

# Zusammenfassend

- ▶ Familienpolymorphismus erlaubt Familien von Klassen
- ▶ Typsicher: verhindert das Mischen von Klassen aus verschiedenen Familien
- ▶ Flexibilität:
  - ▶ Wiederverwendbarkeit: Keine statische Abhängigkeiten zu exakten Klassen
  - ▶ Erweiterbarkeit: Familien erweitern, ohne bestehenden Code zu verändern
- ▶ Einzige Voraussetzung: Familienobjekt muss mit den Instanzen der Familienmitglieder übergeben werden

# Literatur

-  Erik Ernst (2001). Family Polymorphism In *Jorgen Lindskov Knudsen, editor, Proceedings ECOOP 2001, number 2072 in LNCS, pages 303-326, Heidelberg, Germany, 2001. Springer-Verlag.*
-  Compiler, Tutorial:  
<http://www.daimi.au.dk/~eernst/gbeta/>
-  Erik Ernst, Klaus Ostermann, and William R. Cook. *A Virtual Class Calculus*. Extended version of the POPL'06 paper with the same title.