Vorlesung

Graphentechnik

SS 2014

Albert Zündorf

Termine:

Vorlesungen / Übungen: Freitags 14:00 - 15:30 Uhr Raum: -1418

Gliederung

- Vorlesung:
  1. Einführung:
  2. Graphen
  3. Graphersetungssysteme (Story Diagramme)
  4. Beweisverfahren
  5. Anwendungsbeispiele
- Übung:
  6. Fujaba, . . .
  7. Prototypen
- Unterlagen:
  - Folienkopien
  - Literatur: siehe WWW-Seite

Über Home-Page: http://www.se.eecs.uni-kassel.de/se

Der Graphentechnikansatz „Story Driven Modeling“ (Systematischer Umgang mit Graphgrammatiken)

Bsp. 1: Requirements Analysis	**⇦ GUI, Use Cases**
1. Story Boarding	⇦ Story Boards
2. Design	⇦ Graph Schema / Class Diagrams
3. Method Specifications	⇦ Graph Rewrite Rules / Story Charts, Story Diagrams
4. Validation	⇦ Executable Specification
5. Prototyping	⇦ Java-Code + GUI + Framework
6. Efficiency	⇦ Indexes, Parameterization, "more imperative code"

Graphen Modelle

(Theorie und Implementierung)

Motivation:

Knoten	<=>	**Objekt**
Kante	<#>	**Zeiger**
bidirektional		unidirektional
referentielle Integrität		??? (evtl. Garbage Collection)
formal		?
Graphersetzungsregeln		new, delete, :=

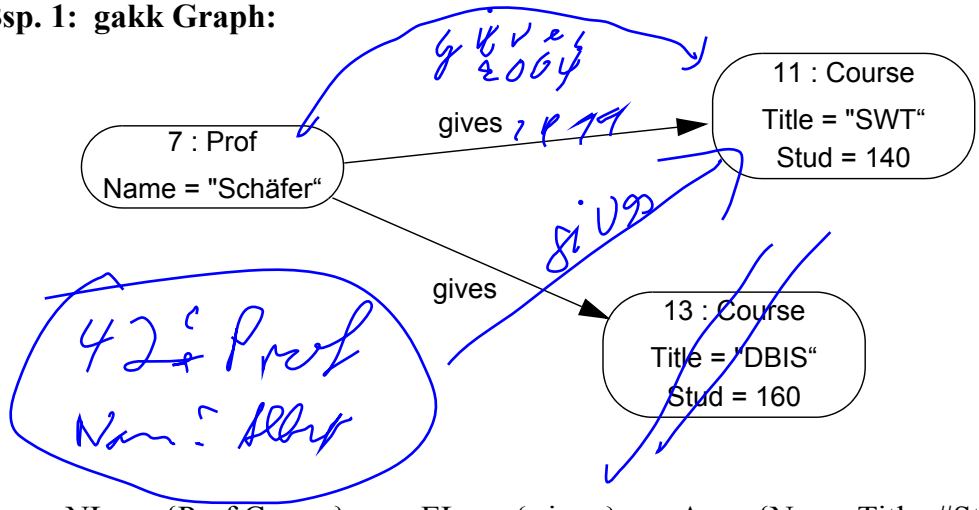
Theorie:

Def. 1: gerichtete, attributierte, knoten- und kantenmarkierte (gakk) Graphen

G := (NL, EL, A, N, E, l, av) mit

NL	endliche Menge von Knotenmarkierungen / -typen / -labeln
EL	endliche Menge von Kantenmarkierungen / -typen / -labeln
A	endliche Menge von Attribut(nam)en
N	endliche Menge von Knoten(bezeichnern)
E	$\subseteq N \times EL \times N$
l	$N \rightarrow NL$
av	$N \times A \rightarrow \{true, false\} \cup N \cup CHAR^* \cup ... \cup P(N) \cup P(CHAR^*) \cup P(...)$

Bsp. 1: gakk Graph:



NL = {Prof, Course} EL = {gives} A = {Name, Title, #Stud}

N = {7, 11, 13} E = {(7, gives, 11), (7, gives, 13)}

l = {7 → Prof, 11 → Course, 13 → Course} also l(7) → Prof

av = {(7, Name) → "Schäfer", (11, Title) → "SWT", (11, #Stud) → 140,

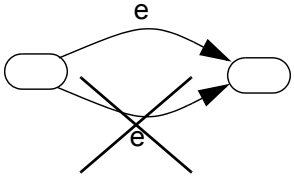
(13, Title) → "DBIS", (13, #Stud) → 160}

ACHTUNG:

$E \subseteq N \times EL \times N$

$\Rightarrow$

- Kanten sind **KEINE** eigenständigen Objekte
  - keine 2 parallelen Kanten gleichen Typs
  - keine Kantenattribute
  - nicht direkt adressierbar
  - (existenzabhängig)



- Kanten sind nicht geordnet: keine Listen, Bäume, Hashtabellen (müssen explizit mitmodelliert werden)

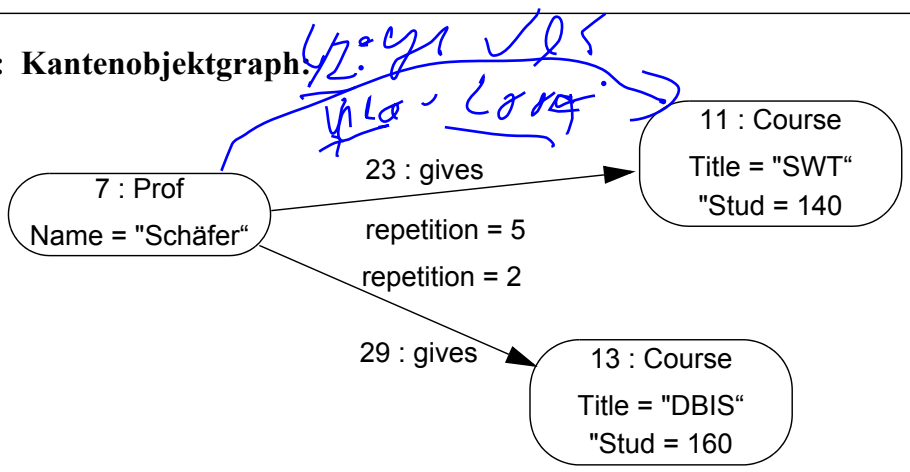
Alternative Graphdefinitionen:

Def. 2: Kantenobjektgraphen

G := (NL, EL, A, N, E, **s, t, nl, el**, av) mit

NL	endliche Menge von Knotenmarkierungen / -typen / -labeln	
EL	endliche Menge von Kantenmarkierungen / -typen / -labeln	
A	endliche Menge von Attribut(nam)en	
N	endliche Menge von Knoten(bezeichnern)	
E	**endliche Menge von Kanten(bezeichnern)**	
s	$E \rightarrow N$	**liefert Source einer Kante (total)**
t	$E \rightarrow N$	**liefert Target einer Kante (total)**
nl	$N \rightarrow NL$	
el	$E \rightarrow EL$	
av	$\langle N \cup E \rangle \times A \rightarrow \{true, false\} \cup N \cup CHAR^* \cup ... \cup P(N) \cup P(CHAR^*) \cup P(...)$	

Bsp. 2: Kantenobjektgraph:



NL = {Prof, Course} EL = {gives} A = {Name, Title, #Stud, repetition}

N = {7, 11, 13} E = {23, 29} s = {23 → 7, 29 → 7}

t={23 → 11, 29 → 13}

nl = {7 → Prof, 11 → Course, 13 → Course} el = {23 → gives, 29 → gives}

av = {(7, Name) → "Schäfer", (11, Title) → "SWT, (11, #Stud) → 140,

(13, Title) → "DBIS", (13, #Stud) → 160,

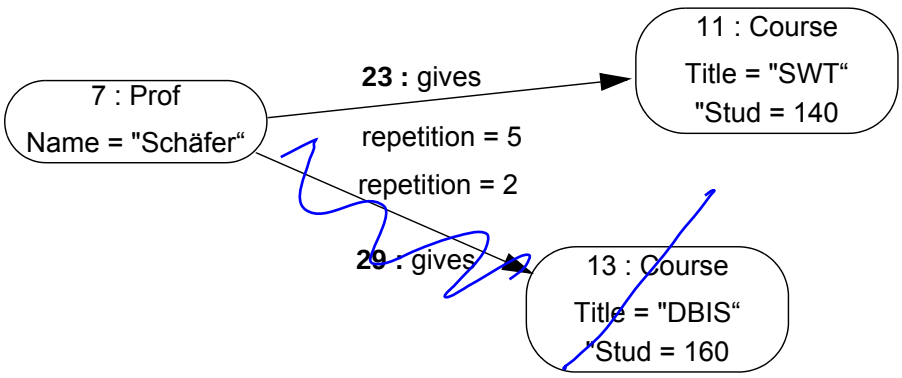
(23, repetition) → 5, (29, repetition) → 2}

Def. 3: Hypergraphen (nur noch Kanten)

G := (EL, A, **E**, **s**, el, av) mit

EL	endliche Menge von Kantenmarkierungen / -typen / -labeln
A	endliche Menge von Attribut(nam)en
E	**endliche Menge von Kanten(bezeichnern)**
s	$E \rightarrow P(E)$ **(liefert Menge von Sourcen einer Kante)**
el	$E \rightarrow EL$
av	$E \times A \rightarrow \{true, false\} \cup N \cup CHAR^* \cup ... \cup P(N) \cup P(CHAR^*) \cup P(...)$

Bsp. 3:Hypergraph:



EL = {gives, Prof, Course} A = {Name, Title, #Stud, repetition}

E = {7, 11, 13, 23, 29}

s = {7 → ∅, 11 → ∅, 13 → ∅, 23 → {7, 11}, 29 → {7, 13}}

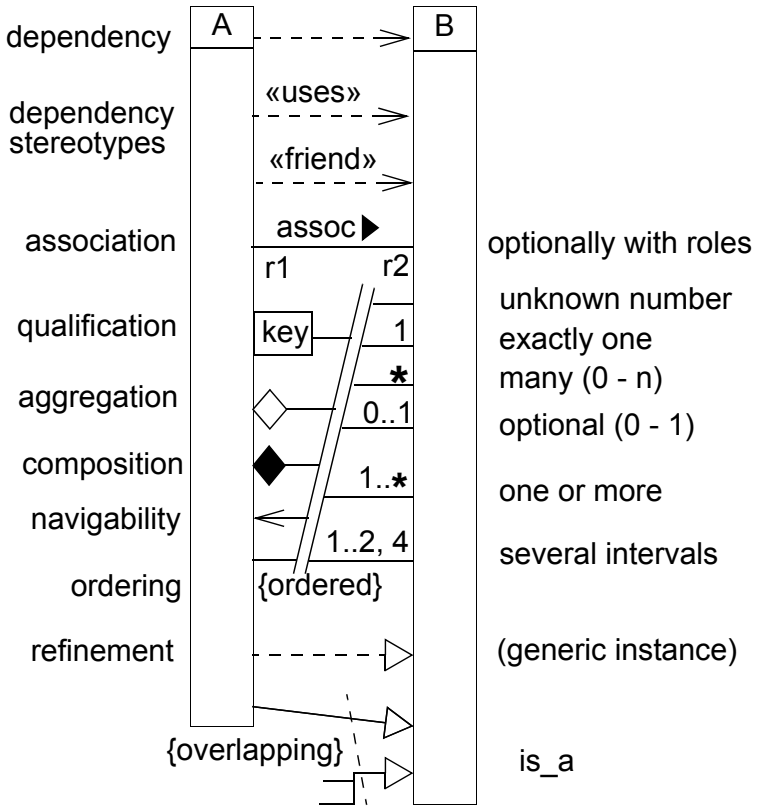
el = {7 → Prof, 11 → Course, 13 → Course, 23 → gives, 29 → gives}

av = {(7, Name) → "Schäfer", (11, Title) → "SWT, (11, #Stud) → 140,

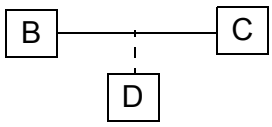
(13, Title) → "DBIS", (13, #Stud) → 160,

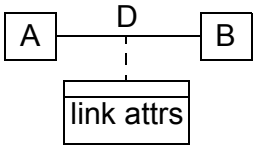
(23, repetition) → 5, (29, repetition) → 2}

Ziel hier: Formalisierung des OO-Datenmodells / von UML-(Klassendiagrammen)

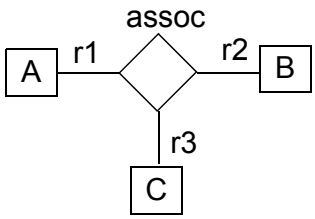


association as class:





ternary (n-ary) association:



Def. 4: Objektorientierte Graphen (erweiterte gakk-Graphen)

G := (SI, Ext) mit

SI := (NL, EL, A, IsAs, Assocs, Attrs) mit (Schema Info)

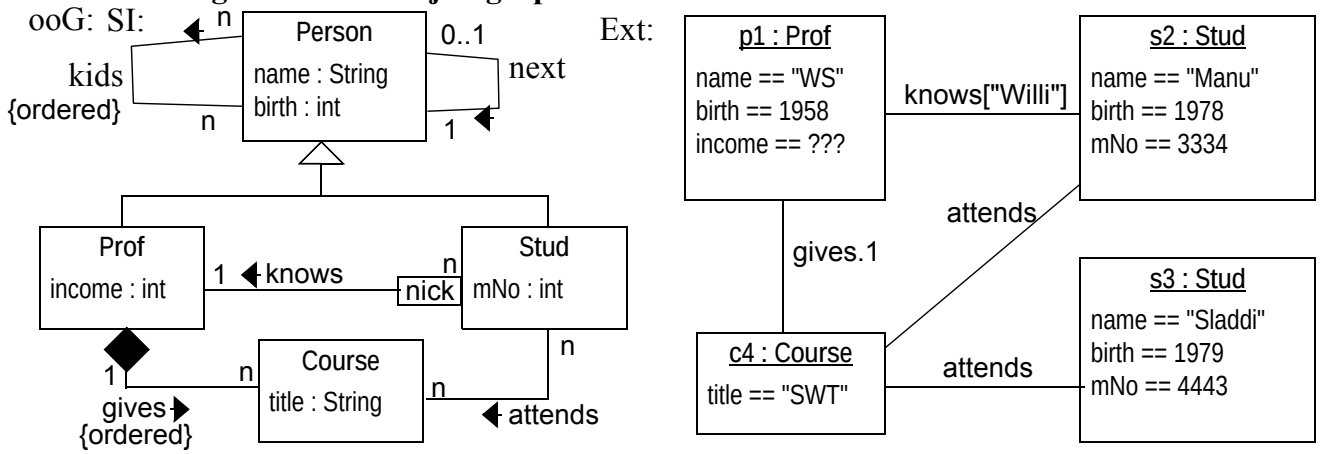
- NL endliche Menge von Knotenmarkierungen / -typen / -labeln
- EL endliche Menge von Kantenmarkierungen / -typen / -labeln
- A endliche Menge von Attribut(nam)en
- **IsAs** $\subseteq$ **Rel (desc $\in$ NL, anch $\in$ NL)** **mit Rel := "Relation über"**
- **Assocs** $\subseteq$ **Rel (el $\in$ EL, srcNl $\in$ NL, srcCard $\in$ MultiInfo := {one, many}**
  assocType $\in$ P (AssocTypes := {qualified, aggregation, ordered})
  tgtNl $\in$ NL, tgtCard $\in$ MultiInfo) **mit Rel := "Relation über"**
- **Attrs** $\subseteq$ **Func (A$\rightarrow$ {Boolean, Integer, String, Float, P(Integer), ...})**

Ext := (N, E, nl, av)

- N endliche Menge von Knoten(bezeichnern)
- E Rel (src $\in$ N, el $\in$ EL, **i** $\in \varepsilon \cup$ ***R*, q** $\in \varepsilon \cup$ **AttrValues**, tgt $\in$ N)
- l Func ((N) $\rightarrow$ NL)
- av Func ((N, A) $\rightarrow$ AttrValues := {true, false} $\cup N \cup R \cup$ *char** $\cup N^* \cup$ ...)

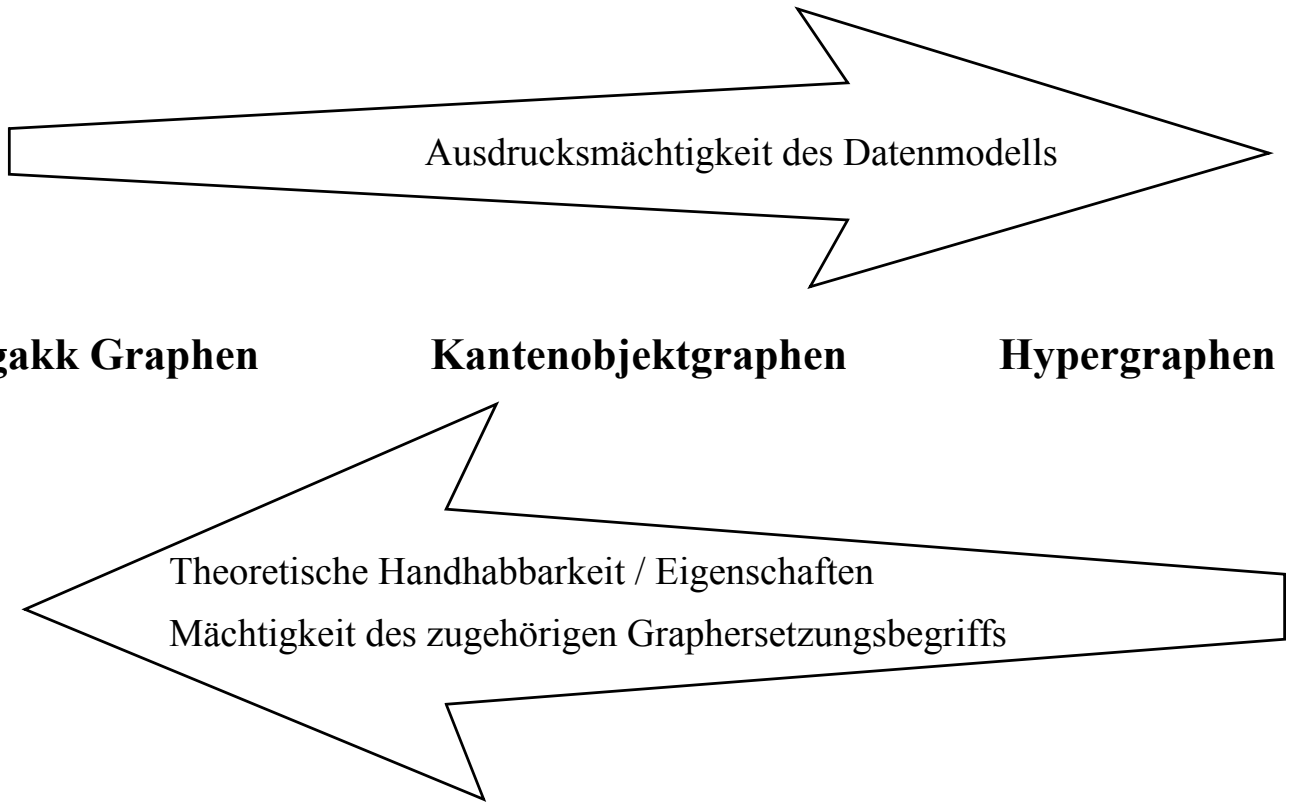
Extension hält Typschema ein.

Bsp. 4: Klassendiagramm und Objektgraph



ooG = (SI, Ext); SI = (NL, EL, A, IsAs, Assocs, Attrs); NL = {Person, Prof, Stud, Course};
EL = {knows, gives, attends, next, kids}; A = {name, birth, income, title, mNo}
IsAs = {(Prof, Person), (Stud, Person), (Prof, Prof), (Stud, Stud), (Person,Person), (Course,Course)};
Assocs = { (attends, Stud,many, {}, Course, many), (knows, Stud, many, {qualified}, Prof, one),
(gives, Prof, one, {aggregation, ordered}, Course, many),
(next, Person, one, {}, Person, one), (kids, Person, many, {ordered}, Person, many)}
Attrs = {(Person.name) → String, (Person.birth) → int, (Prof.income) → int, Stud.mNo → int, Course.title → String}

Ext = (N, E, nl, av)
N = {p1, s2, s3, c4};
E = {(s2, attends, ε, c4), (s3, attends, ε, c4), (p1, gives, 1, c4), (s2, knows, "Willi", p1)};
nl = {(p1)→Prof, (s2)→Stud, (s3)→Stud, (c4)→Course}; av={(p1,Prof.name)→"WS", (p1,Prof.birth)→1958, ...}



Wir: OO-Graphen, das Datenmodell von FUJABA

Implementierung von (gakk) Graphen:

- Matritzen

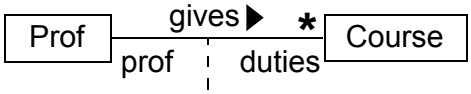
OId	...	7	...	11	...	13
...						
7				gives		gives
...						
11						
...						
13						

OId	Name	Title	#Stud
...			
7	"Schäfer"		
...			
11		"SWT"	140
...		"DBIS"	160
13			

- Pro: Graphalgorithmen per Matrixmultiplikation, Einsatz von Großrechnern
- Contra:
  - statische Graphgröße, hoher (quadratischer) Platzbedarf => Sparse Arrays
  - Problemklasse mit Matrixmultiplikationslösungen sehr speziell, häufig "hart"

Für die Graphentechnik (Softwaretechnik) unbedeutend

- Nachbarschaftslisten und Rückzeiger, Design Pattern (FUJABA):




```
----------file Prof.java----------
import com.objectspace.jgl.*;
...
public class Prof { ...
   private OrderdSet duties = new OrderedSet ();
   public void addToDuties (Course elem) {
      if (elem != null && !this.hasInDuties (elem)) {
         this.duties.add (elem);
         elem.setProf (this);
      }
   }
   public void removeFromDuties (Course elem) {
      if (this.hasInDuties (elem)) {
         this.duties.remove (elem);
         elem.setProf (null);
      }
   }
   ...
```

```
----------file Course.java----------
import com.objectspace.jgl.*;
...
public class Course { ...
   private Prof prof = null;
   public void setProf (Prof prof) {
      if (this.prof != prof) { // newPartner
         if (this.prof != null) { // inform old partner
            Prof oldProf = this.prof;
            this.prof = null;
            oldProf.removeFromDuties (this);
         } // if
         this.prof = prof;
         if (prof != null) { // inform new partner
            prof.addToDuties (this);
         } // if
      } // if
   }
   ...
```

```
------file AssocTest.java----------
. . .
class AssocTest {
public static void main ( . . . ) {
    Course c1;
    Course c2;
    Prof p1;
    ...
    p1 = new ...;
    p2 = new ...;
    c1 = new ...;
    c2 = new ...;
    ...
    p1.addToDuties (c1);
    p1.addToDuties (c2);
    p2.addToDuties (c2);
    ...
    c1.putProf (null);
    ...
    p1.removeYou ();
    ...
}
```

```
----------trace for p1.addToDuties (c2) ----------

1. main: p1.addToDuties (c2)
    2. p1.addToDuties: ! hasInDuties (c2) ==> true
    3. p1.addToDuties: this.duties.add (c2)
    4. p1.addToDuties: c2.setProf (p1)
        5. c2.setProf: this.prof != p1 ==> true
        6. c2.setProf: this.prof != null ==> false
        7. c2.setProf: this.prof = p1
        8. c2.setProf: p1 != null ==> true
        9. c2.setProf: p1.addToDuties (c2)
            10. p1.addToDuties': ! hasInDuties (c2) ==> false
            11. p1.addToDuties': return
        12. c2.setProf: return
    13. return
14. main: . . .
```



```
// Prof.java

    public boolean hasInCourses (Course elem) {
        return this.courses.get (elem) != null;
    }
    public Enumeration elementsOfCourses () {
        return this.courses.elements ();
    }
    ...
    public void removeYou () {
        Enumeration enum = elementsOfCourses ();
        while (enum.hasMoreElements ()) {
            Course elem =
                    (Course) enum.nextElement ();
            removeFromDuties (elem);
        }
        . . .
    }
}
```

```
// Course.java

    public Prof getProf () {
        return prof;
    }
    public void removeYou () {
        setProf (null);
    }
    . . .
}
```

```
// qualified Assocs
class Stud { ...
   private TreeMap knows;
   public addToKnows (String key, Prof obj) {
        . . .
   } ... }
```

```
// ordered assocs
class Prof { ...
   private LinkedList gives;
   ...
   // Aggregation
   public void removeYou () {
        ...
        Enumeration enum = elementsOfCourses ();
        while (enum.hasMoreElements ()) {
             Course  obj =
                  (Course) enum.nextElement ();
             obj.removeYou ();
        }
   }
...
}
```


Design Pattern:

- Theorie: OO-Graphen
- Implementierung
  - Expliziter Graph fehlt (simuliert per Einstiegsknoten)
  - 1 zu 1 -Beziehung als Zeigerpaar
  - zu n -Beziehung mit generischen Mengen, Listen, Maps
  - beliebig viele Objektarten
  - beliebige Attributierung
- Schnittstelle
  - Information-Hiding
  - komfortable
  - referentielle Integrität
  - Typsicherheit
- Manipulationsoperationen: elementar
- Graphentechnik:
  - ~~manuelle Pattern-Instantiierung (Automatisierung wünschenswert)~~
  - automatische Instantiierung per Fujaba Environment
  - geringer Overhead
  - Pattern verbesserbar

- Leda: Bibliothek für **Kantenobjektgraphen** (Mehlhorn)
  - Graph ist Liste von Knoten plus Liste von Kanten
  - nur je ein Knotentyp, Kantentyp, Knotenattribut, Kantenattribut

```
----------file LEDA/graph.h------

class graph {
private
   obj_list V;
   obj_list E; . . .

public
   graph();
   virtual ~graph();

   list<edge> all_edges()     const;
   list<node> all_nodes()     const;

   node source(edge e)    const;
   node target(edge e)    const;
   list<edge> adj_edges(node v) const;
   list<edge> in_edges(node v) const;

   node new_node();
   edge new_edge(node v, node w);
   void del_node(node v);
   void del_edge(edge e);

   void sort_nodes(int (*cmp)(const node&, const node&));
   void sort_edges(int (*cmp)(const edge&, const edge&));

   list<edge> insert_reverse_edges();

   void write(ostream& O = cout) const;
   int  read(istream& I = cin);
   . . . }

template<class vtype, class etype> class GRAPH : public graph {
public
   vtype  inf(node v)    const;
   etype  inf(edge e)    const;
   void   assign(node v, vtype x);
   void   assign(edge e, etype x);
   node   new_node(vtype x);
   edge   new_edge(node v, node w, etype a);
```



```
----------file main.cc------

   . . .
main ( . . . ) {
   GRAPH<ProfCourse *, el> g;
   ProfCourse * c1;
   ProfCourse * c2;
   ProfCourse * p1;

   node<ProfCourse *> nc1, nc2, np1;
   ...
   p1 = new ...;
   c1 = new ...;
   c2 = new ...;
   ...
   nc1 = g.new_node ( c1 );
   nc2 = g.new_node ( c2 );
   np1 = g.new_node ( p1 );
   ...
   g.new_edge ( np1, nc1, gives );
   g.new_edge ( np1, nc2, gives );
   ...
```

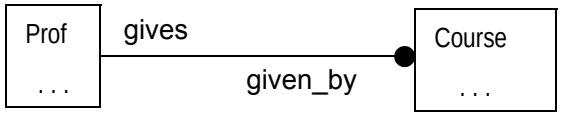
```
edge< el > e;

e = connecting ( np1, nc1, gives );
        // this operation is not supported by LEDA;

g.del_edge ( e );

...

g.del_node ( np1 );
```

Leda:

- Theorie:
  - |NL| = 1
  - |EL| = 1
  - |A| = 2
  - Kantenobjekte
- Implementierung
  - Liste von Knoten jeweils mit
    - Liste von auslaufenden Kanten
    - Liste von einlaufenden Kanten
    - ein generisches Attribut
  - Liste von Kanten jeweils mit
    - ein Quellknoten
    - ein Zielknoten
    - ein generisches Attribut
- Manipulationsoperationen: elementar
- Schnittstelle
  - komfortable
  - nur Frage (N, ?, N) fehlt
- Bemerkungen
  - Vielzahl von Graphalgos:
    - TopSort
    - DepthFirstSearch
    - BreadthFirstSearch
    - Planarity
  - Revert
- für Graphentechnik:
  - **eingeschränkte Markierung- und Attributierung kritisch**
  - x zu 1 -Beziehung immer Liste

- ODMG ’93: gakk Graphen
  - Methoden
  - Persistenz
  - mengenorientierte Anfrage- und Manipulationssprache:

```
Set<String> ss = select c.Title from c in Courses
                 where c.given_by->get_name ( ) == "Schäfer"
                   and c.get_#stud >= 100;
```




```
----------file Course.odmg----------
. . .
class Course {
   Ref<Prof> given_by inverse Prof::gives;
   ...
}
```

```
----------file Prof.odmg----------
. . .
class Prof{
   Set<Ref<Course>> gives inverse Course::given_by;
   ...
}
```

ODMG '93:

- Theorie: gakk Graph
- Implementierung
  - Expliziter Graph
    - Extensionen für jede Objektklasse
    - explizite Einstiegsobjekte
  - 1 zu 1 -Beziehung als Zeigerpaar
  - zu n -Beziehung mit generischer Collection
  - beliebig viele Objektarten
  - beliebige Attributierung
- Schnittstelle
  - Information-Hiding
  - komfortable
  - referentielle Integrität
  - Typsicherheit
- Manipulationsoperationen: elementar
- Anfrageoperationen: mengenorientiert
- Datenbankfunktionalität
  - Persistenz
  - Nebenläufige Zugriffe
  - Client/Server-Konzept
  - heterogene Plattformen
  - Standardisierung
- Graphentechnik:
  - prima, aber
  - Persistenz oft Overhead
  - Baukasten für Einzelaspekte der Datenbankfunktionalität wäre toll

Hausaufgabe 1:

~~Wir planen eine Biergartensimulationssystem mit folgenden Eigenschaften:~~

- ~~der Biergarten besteht im wesentlichen aus ´ner Reihe von Tischen, der Theke, den Gästen und der Bedienung~~
- ~~zu trinken gibts Bier (1 Euro) und Wasser (1.70 Euro) (erst mal nur aus Flaschen)~~
- ~~die Bedienung läuft die Tische an, liefert Getränke aus (muß direkt bezahlt werden) und nimmt Bestellungen entgegen~~
- ~~an der Theke gibt die Bedienung das Geld ab und nimmt neue Getränke aus dem Vorrat~~
- ~~was nicht mehr da ist kann man nicht mehr bestellen~~
- ~~Leergut wird zunächst vernachlässigt~~

~~AUFGABE:~~

- ~~Malt (per Hand) (ein paar) Story-Board-Diagramme, die die oben beschriebenen Vorgänge modellieren. Versucht die finanziellen Sachen in Attributen mitzumodellieren.~~

Abgabe: _____