Männliche und weibliche Doppelstudenten: ein Härtetest für Programmiersprachen

Christian Heinlein

Studiengang Informatik
Fakultät Elektronik und Informatik
Hochschule Aalen − Technik und Wirtschaft
vorname.nachname@htw-aalen.de

Abstract. Vererbung und Subtyp-Polymorphie sind zwei wesentliche Konzepte, die objektorientierte Programmiersprachen von klassischen prozeduralen Sprachen unterscheiden. Mit Hilfe einfacher Vererbung, die von allen objektorientierten Sprachen angeboten wird, lassen sich z. B. Männer, Frauen und Studenten jeweils als Untertypen eines allgemeineren, eventuell abstrakten Obertyps Person definieren. Um männliche und weibliche Studenten möglichst einfach und direkt definieren zu können, ohne bereits vorhandene Definitionen wiederholen zu müssen, benötigt man mehrfache Vererbung, die jedoch nur von wenigen Sprachen (z. B. C++, Eiffel und CLOS) unterstützt wird und in der Praxis zu zahlreichen Komplikationen führt (z. B. Namenskonflikte, Unterscheidung von virtuellen und nicht-virtuellen Basisklassen etc.). Um schließlich Klassen definieren zu können, die die Eigenschaften anderer Klassen mehrfach besitzen − wie z. B. Doppelstudenten, die quasi zweimal Student in unterschiedlichen Studiengängen sind −, benötigt man wiederholte Vererbung, die von keiner gängigen Sprache direkt unterstützt wird, aber z. B. durch künstliche Hilfsklassen in C++ oder durch Umbenennen von Features in Eiffel erreicht werden kann. Spätestens beim Versuch, mehrfache und wiederholte Vererbung miteinander zu kombinieren − um beispielsweise männliche und weibliche Doppelstudenten zu modellieren −, stößt man jedoch an die Grenzen gängiger Programmiersprachen.
Mit Hilfe sogenannter offener Typen und bidirektionaler Relationen, einem Kernkonzept meiner Forschungsarbeit über „verbesserte prozedurale Programmiersprachen“, lassen sich nicht nur mehrfache und wiederholte Vererbung einfacher handhaben als mit heutigen objektorientierten Sprachen, sie erlauben darüber hinaus auch deren beliebige Kombination, d. h. die erwähnten männlichen und weiblichen Doppelstudenten lassen sich ohne Probleme definieren. Der Schlüssel zu dieser erhöhten Flexibilität liegt in der Kombination der üblicherweise inkompatiblen Konzepte von Vererbung und Aggregation zu einem einzigen einheitlichen Konzept.

1 Einfache, mehrfache und wiederholte Vererbung in C++

Abbildung 1 zeigt eine Vererbungshierarchie mit einer Wurzelklasse `Person`, direkten Unterklassen `Mann`, `Frau` und `Stud` (Student/in) sowie indirekten Unterklassen `MnlStud` (männlicher Student), `WblStud` (weiblicher Student) und `DplStud` (Doppelstudent). Ein Doppelstudent sei hierbei ein Student, der in zwei Studiengängen

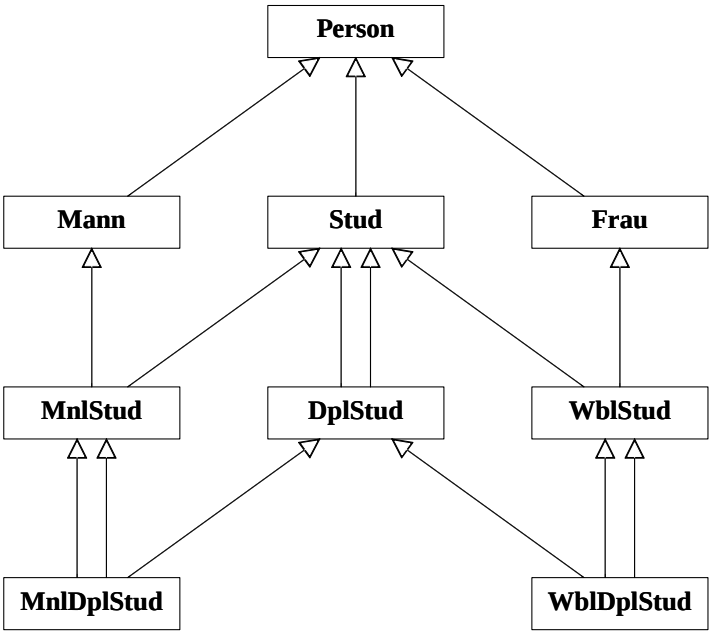

Abbildung 1: Beispiel einer Vererbungshierarchie

(z. B. Mathematik und Informatik) gleichzeitig eingeschrieben ist und dementsprechend zwei unabhängige `Stud`-Teile (mit Daten wie Studiengang, Matrikelnummer, bis jetzt erworbenen Credit points etc.) besitzen soll. Außerdem soll ein `DplStud`-Objekt zweifach polymorph als `Stud`-Objekt verwendbar sein, um beispielsweise als Mathematikstudent in eine Liste aller Mathematikstudenten und als Informatikstudent in eine Liste aller Informatikstudenten eingetragen werden zu können.

Abbildung 2 skizziert C++-Code zur Implementierung der bis jetzt beschriebenen Klassen sowie die polymorphe Verwendung eines Doppelstudenten in seinen beiden Studentenrollen. Zur Implementierung der Klassen ist folgendes anzumerken:

- Das Schlüsselwort `struct` ist äquivalent zu `class`, mit dem einzigen Unterschied, dass alle Bestandteile einer so definierten Klasse automatisch öffentlich sind.
- Die konkreten Daten und Operationen der Klassen sind für die weiteren Betrachtungen irrelevant.
- Damit Objekte der Klassen `MnlStud` und `WblStud` jeweils nur *ein* `Person`-Teilobjekt enthalten, obwohl sie indirekt jeweils zweimal von `Person` abgeleitet sind, muss `Person` eine *virtuelle* Basisklasse von `Mann`, `Frau` und `Stud` sein [4].
- Um eine Klasse wie `DplStud` zu definieren, die zweimal von derselben Basisklasse `Stud` abgeleitet werden soll, benötigt man künstliche Zwischenklassen `StudTeil1` und `StudTeil2` [4].

```
// Personen.
struct Person { ..... };

// Männer, Frauen und Studenten als spezielle Personen.
struct Mann : virtual Person { ..... };
struct Frau : virtual Person { ..... };
struct Stud : virtual Person { ..... };

// Männliche und weibliche Studenten
// als Kombination von Mann/Frau und Stud.
struct MnlStud : Mann, Stud { ..... };
struct WblStud : Frau, Stud { ..... };

// Doppelstudenten.
struct StudTeil1 : Stud {};
struct StudTeil2 : Stud {};
struct DplStud : StudTeil1, StudTeil2 { ..... };

// Ein Doppelstudent, z. B. ein Mathe-Info-Student.
DplStud* ds = new DplStud(...);

// Polymorphe Verwendung des ersten Stud-Teils.
Stud* s1 = (StudTeil1*)ds;
list<Stud*> mathe_studs; mathe_studs.push_back(s1);

// Polymorphe Verwendung des zweiten Stud-Teils.
Stud* s2 = (StudTeil2*)ds;
list<Stud*> info_studs; info_studs.push_back(s2);
```

Abbildung 2: C++-Code zur teilweisen Implementierung der Vererbungshierarchie

Die ebenfalls in Abb. 1 dargestellten Klassen `MnlDplStud` und `WblDplStud` sollen männliche bzw. weibliche Doppelstudenten repräsentieren, die polymorph sowohl als normale Doppelstudenten als auch zweifach als männliche bzw. weibliche Studenten verwendbar sein sollen, damit beispielsweise ein männlicher Mathe-Info-Student einerseits wie ein gewöhnlicher Mathe-Info-Student (und damit auch als gewöhnlicher Mathe- und als gewöhnlicher Info-Student) und andererseits sowohl als männlicher Mathe-Student (und auf diese Weise wiederum auch als normaler Mathe-Student) als auch als männlicher Info-Student (und damit auch wiederum als normaler Info-Student) verwendet werden kann. Trotzdem soll ein männlicher Doppelstudent, wie in Abb. 3 gezeigt, insgesamt natürlich nur zwei verschiedene `Stud`-Teilobjekte besitzen. (Diese Abbildung sollte am besten dreidimensional betrachtet werden: die „obere“ Ebene `Person – Stud|Stud – DplStud` beschreibt einen gewöhnlichen Doppelstudenten mit einem Person- und zwei Studententeilen, während die „untere“ Ebene `Mann – MnlStud|MnlStud – MnlDplStud` die zugehörigen männlichen Spezialisierungen enthält.)

Damit ein männlicher Doppelstudent einerseits als gewöhnlicher Doppelstudent und andererseits zweifach als männlicher Student verwendbar ist, muss die Klasse

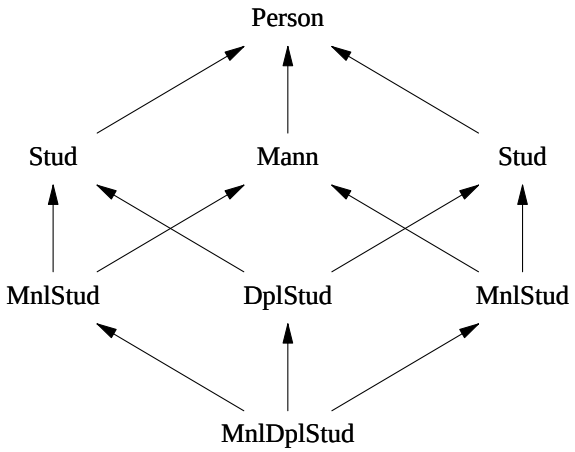

Abbildung 3: Teilobjekte eines `MnlDplStud`-Objekts

`MnlDplStud` sowohl von `DplStud` als auch zweifach (via Hilfsklassen `MnlStudTeil1` und `MnlStudTeil2`) von `MnlStud` abgeleitet werden:

```
struct MnlStudTeil1 : MnlStud {};
struct MnlStudTeil2 : MnlStud {};
struct MnlDplStud : DplStud, MnlStudTeil1, MnlStudTeil2 {};
```

Damit enthält ein Objekt dieser Klasse jedoch insgesamt vier unabhängige `Stud`-Teilobjekte! Um dies zu vermeiden, könnte man versuchen, einzelne nicht-virtuelle Unterklassenbeziehungen durch virtuelle zu ersetzen. Doch egal welche Kombination von virtuellen und nicht-virtuellen Basisklassen man wählt, man erreicht nie die in Abb. 3 dargestellte Struktur von `MnlDplStud`-Objekten, d. h. dieses Vererbungsproblem scheint in C++ unlösbar zu sein.

Andere Sprachen mit Mehrfachvererbung (konkret wurden CLOS und Eiffel untersucht) scheitern zum Teil bereits an der adäquaten Modellierung von Doppelstudenten.

2 Vererbung mit offenen Typen und bidirektionalen Relationen

Typen und Attribute. *Offene Typen* [3, 2] sind ein alternatives Datenmodell für prozedurale und objektorientierte Programmiersprachen, das im Rahmen des Projekts APPLEs (Advanced Procedural Programming Language Elements) [1] an der Universität Ulm entwickelt wurde. Ihr Grundprinzip besteht darin, dass Typ- und Attributdeklarationen syntaktisch voneinander getrennt werden. Dadurch ist es möglich, die Attribute eines Typs inkrementell zu definieren, d. h. je nach Bedarf schrittweise zu einem vorhandenen Typ hinzuzufügen. Alle so definierten Attribute eines Typs sind optional, d. h. ein konkretes Objekt des Typs muss nicht notwendigerweise Werte für alle Attribute des Typs besitzen. Wenn man auf ein nicht vorhandenes Attribut zugreift, er-

hält man als Ergebnis einen wohldefinierten Nullwert, der die Abwesenheit eines echten Werts anzeigt. Die Objekte offener Typen besitzen Referenzsemantik, ähnlich wie Objekte in Java, es gibt eine automatische Speicherbereinigung, und offene Typen sind statisch typsicher.

Abbildung 4 zeigt die Definition eines offenen Typs `Person` mit zwei einwertigen Attributen `name` und `vorname` (jeweils vom Typ `string`) und einem benutzerdefinierten Konstruktor zur Erzeugung und Initialisierung eines `Person`-Objekts sowie eine schematische Darstellung des so erzeugten Objekts `p`.

```
typename Person;                // Offener Typ.
Person -> string name;          // Einwertiges Attribut.
Person -> string vorname;       // Dto.

// Konstruktor erzeugt Person-Objekt und initialisiert
// Attribut vorname mit v und Attribut name mit n.
Person (string v, string n) {   // Konstruktor.
  return Person(@vorname, v)(@name, n);
}

// Aufruf des Konstruktors und Verwendung des Objekts p.
p = Person("Christian", "Heinlein");
print(p@vorname, p@name);
```

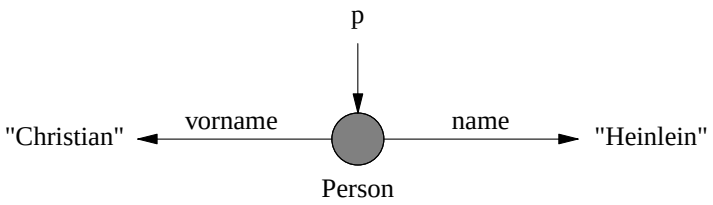

Abbildung 4: Offener Typ mit Attributen und Konstruktor

Bidirektionale Relationen. Neben ein- und mehrwertigen Attributen, bieten offene Typen auch eine direkte Unterstützung für *bidirektionale Relationen*, d. h. Paare von Attributen, die zueinander inverse Abbildungen darstellen. Das besondere hierbei ist, dass bei einer Änderung einer Richtung einer Relation die Gegenrichtung automatisch mitgeändert wird. Abbildung 5 zeigt neben zwei weiteren offenen Typen `Auto` und `Motor` zwei unterschiedliche Arten von Relationen sowie eine exemplarische Beziehungsstruktur zwischen Objekten. (Bei einer anonymen Relation fehlen die Namen der beiden Attribute; in diesem Fall können die jeweiligen Typnamen als „Rollennamen“ verwendet werden.) Wie am Anfang dieses Abschnitts erwähnt, sind Attribute und damit auch Relationen grundsätzlich optional, d. h. ein Auto muss nicht notwendigerweise mit einem Motor in Beziehung stehen und umgekehrt.

```
typename Auto;
Person besitzer <->> Auto autos;        // 1:N-Relation.

typename Motor;
Auto <-> Motor;                  // Anonyme 1:1-Relation.
```

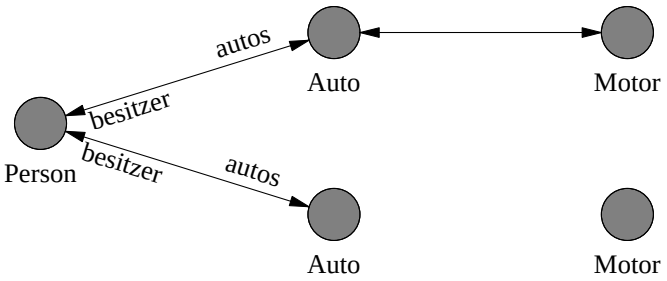

Abbildung 5: Bidirektionale Relationen

Einfache „Vererbung“. Interessanterweise lassen sich mit bidirektionalen Relationen auch Vererbungsbeziehungen modellieren, ohne dass man hierfür spezielle Vererbungsmechanismen benötigt. Die einzige notwendige Erweiterung sind sog. *automatische* Relationen, die vom Compiler bei Bedarf automatisch angewandt werden, um ein Objekt des einen Typs in ein Objekt des anderen Typs umzuwandeln (ähnlich wie in objektorientierten Sprachen ein Objekt eines Untertyps bei Bedarf automatisch in ein Objekt eines Obertyps umgewandelt wird).

Abbildung 6 zeigt, wie sich auf diese Weise `Stud` quasi als Untertyp von `Person` definieren lässt und wie ein Student schrittweise aus einem `Person`- und einem `Stud`-Teilobjekt zusammengesetzt werden kann. Abbildung 7 zeigt zwei benutzerdefinierte Konstruktoren, die diese Objektkonstruktion kapseln, einen Aufruf des zweiten Konstruktors sowie typische „objektorientierte“ Verwendungen eines `Stud`-Objekts. (Die Zweckmäßigkeit des ersten Konstruktors wird später ersichtlich.)

Mehrfache Vererbung. Abbildung 8 zeigt am Beispiel `MnlStud`, wie sich mehrfache Vererbung mit bidirektionalen Relationen modellieren lässt. Die verschiedenen Konstruktoren des Typs und die zugehörigen Abbildungen zeigen, dass man je nach Bedarf sowohl nicht-virtuelle als auch virtuelle Basisklassen nachbilden kann, wobei zur Implementierung des letzten Konstruktors die Konstruktion von `Mann` und `Stud` „aus existierenden Teilobjekten“ (erster Konstruktor in Abb. 7) verwendet wird.

Wiederholte Vererbung. Die Abbildungen 9 und 10 zeigen die Definition der Typen `DplStud` und `MnlDplStud` sowie die zugehörigen Konstruktoren. Im Gegensatz zur C++-Lösung fällt auf, dass keinerlei künstliche Hilfsklassen benötigt werden und dass sich männliche Doppelstudenten problemlos definieren lassen.

```
typename Stud;          // Offener Typ.
Stud -> string fach;    // Attribut.
Stud -> integer matr;   // Attribut.
Stud <->! Person;       // Automat. 1:1-Relationen zu Person.

// Person- und Stud-Objekt erzeugen und miteinander verbinden.
Person p = Person(@vname, "Christian")(@name, "Heinlein");
Stud s = Stud(@fach, "Info")(@matr, 123456);
s(@Person, p);
```

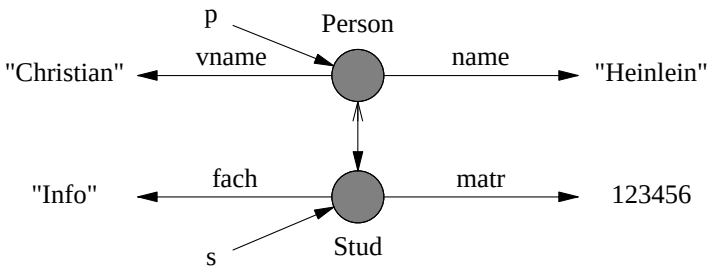

Abbildung 6: Einfache Vererbung mit bidirektionalen Relationen

```
// Konstruktion eines Studenten aus einem exist. Person-Obj.
Stud (Person p, string f, integer m) {
  return Stud(@Person, p)(@fach, f)(@matr, m);
}

// Konstruktion eines Studenten aus atomaren Werten.
Stud (string v, string n, string f, string m) {
  // Obigen Konstruktor aufrufen.
  return Stud(Person(v, n), f, m);
}

// Aufruf des zweiten Konstr. und Verwendung des Objekts s.
Stud s = Stud("Christian", "Heinlein", "Info", 123456);
print(s@fach, s@matr);

// Polymorphe Verwendung von Stud s als Person p.
print(s@name);          // Entspricht: print(s@Person@name)
Person p = s;           // Entspricht: Person p = s@Person;

// Dynamischer Typtest: Ist p ein Student?
if (p@Stud) {
  print(p@Stud@matr); // Expliziter "Downcast" Person -> Stud.
}
```

Abbildung 7: Konstruktion und Verwendung eines `Stud`-Objekts

```
typename Mann;
Mann -> boolean bart;
Mann <->! Person;   // Automatische 1:1-Beziehung zu Person.

// Konstruktoren analog zu Stud.
Mann (Person p, boolean b) { ..... }
Mann (string v, string n, boolean b) { ..... }

typename MnlStud;
MnlStud <->! Mann;  // Automatische 1:1-Beziehung zu Mann.
MnlStud <->! Stud;  // Automatische 1:1-Beziehung zu Stud.

// Konstruktion aus existierenden Teilobjekten.
MnlStud (Mann m, Stud s) {
  return MnlStud(@Mann, m)(@Stud, s);
}

// Konstr. aus atomaren Werten ("schizophren", Abb. links).
MnlStud (string v1, string n1, boolean b,
         string v2, string n2, string f, integer m) {
  Mann m = Mann(v1, n1, b);
  Stud s = Stud(v2, n2, f, m);
  return MnlStud(m, s); // Ersten Konstruktor aufrufen.
}

// Konstruktion aus atomaren Werten (normal, Abb. rechts).
MnlStud (string v, string n, boolean b, string f, integer m)
{
  Person p = Person(v, n);
  Mann m = Mann(p, b);
  Stud s = Stud(p, f, m);
  return MnlStud(m, s); // Ersten Konstruktor aufrufen.
}
```

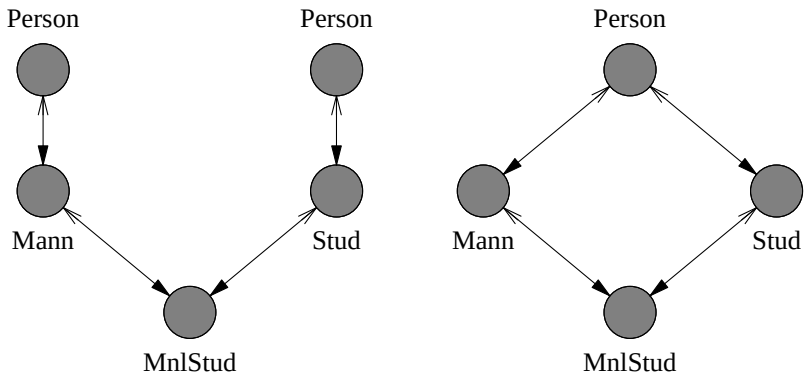

Abbildung 8: Mehrfache Vererbung mit bidirektionalen Relationen

```
typename DplStud;
DplStud <-> Stud StudTeil1; Stud StudTeil1 <->! Person;
DplStud <-> Stud StudTeil2; Stud StudTeil1 <->! Person;

// Konstruktion aus existierenden Teilobjekten.
DplStud (Stud s1, Stud 2) {
  return DplStud(@StudTeil1, s1)(@StudTeil2, s2);
}

// Konstruktion aus atomaren Werten.
DplStud (string v, string n, string f1, integer m1,
                             string f2, integer m2) {
  Stud s1 = Stud(Person(), f1, m1);
  Stud s2 = Stud(Person(), f2, m2);
  Person p = Person(v, n)(@StudTeil1, s1)(@StudTeil2, s2);
  return DplStud(s1, s2); // Obigen Konstruktor aufrufen.
}
```

Abbildung 9: Doppelstudenten

3 Zusammenfassung

Anhand des Beispiels männlicher (und weiblicher) Doppelstudenten wurde gezeigt, dass sich bestimmte Vererbungsprobleme mit gängigen Programmiersprachen (höchstwahrscheinlich) nicht lösen lassen. Offene Typen und bidirektionale Relationen hingegen bieten die notwendige Flexibilität, um derartige Probleme zu lösen, da Objekte von „Untertypen“ nach Belieben aus Objekten der „Obertypen“ zusammengesetzt werden können.

Referenzen

[1] C. Heinlein: “APPLE: Advanced Procedural Programming Language Elements.” In: W. Goerigk (ed.): *Programmiersprachen und Rechenkonzepte* (21. Workshop der GI-Fachgruppe; Bad Honnef, Mai 2004). Bericht Nr. 0410, Institut für Informatik, Christian-Albrechts-Universität zu Kiel, January 2005, 59−66.

[2] C. Heinlein: “Open Types and Bidirectional Relationships as an Alternative to Classes and Inheritance.” In: M. Hanus, F. Huch (eds.): *Programmiersprachen und Rechenkonzepte* (22. Workshop der GI-Fachgruppe 2.1.4; Bad Honnef, Mai 2005). Bericht Nr. 0513, Institut für Informatik, Christian-Albrechts-Universität zu Kiel, October 2005, 30−39.

[3] C. Heinlein: “Open Types and Bidirectional Relationships as an Alternative to Classes and Inheritance.” *Journal of Object Technology* 6 (3) March/April 2007, 101−151, http://www.jot.fm/issues/issue_2007_03/article3.

[4] B. Stroustrup: *The C++ Programming Language* (Special Edition). Addison-Wesley, Reading, MA, 2000.

```
typename MnlDplStud;
MnlDplStud <–>! DplStud;
MnlDplStud <–> MnlStud MnlStudTeil1;
                            MnlStud MnlStudTeil1 <–>! Mann;
MnlDplStud <–> MnlStud MnlStudTeil2;
                            MnlStud MnlStudTeil2 <–>! Mann;

// Konstruktion aus existierenden Teilobjekten.
MnlDplStud (DplStud ds, MnlStud ms1, MnlStud ms2) {
  return MnlStud(@DplStud, ds)
   (@MnlStudTeil1, ms1)(@MnlStudTeil2, ms2);
}

// Konstruktion aus atomaren Werten.
MnlDplStud (string v, string n, boolean b,
            string f1, integer m1, string f2, integer m2) {
  DplStud ds = DplStud(v, n, f1, m1, f2, m2);
  MnlStud ms1 = MnlStud(Mann(), ds@StudTeil1);
  MnlStud ms2 = MnlStud(Mann(), ds@StudTeil2);
  Mann m =
   Mann(ds, b)(@MnlStudTeil1, ms1)(@MnlStudTeil2, ms2);
  return MnlDplStud(ds, ms1, ms2); // Ersten Konstr. aufrufen.
}
```

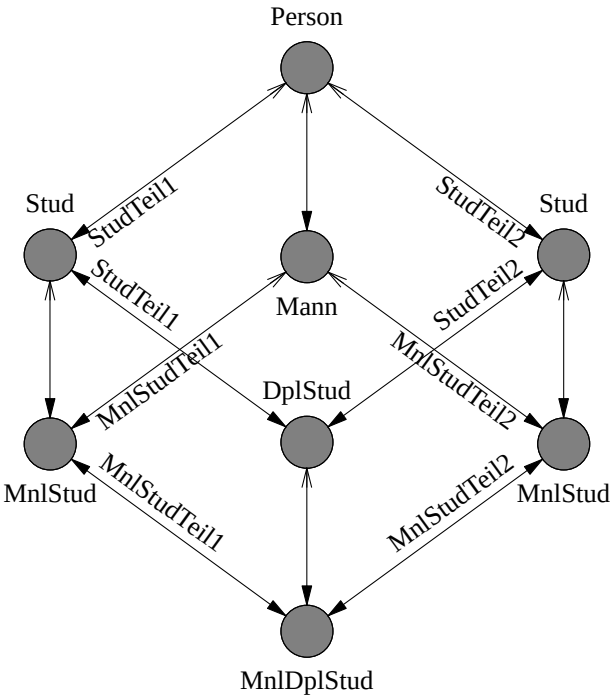

Abbildung 10: Männliche Doppelstudenten