21

Von der UML nach C++

Dieses Kapitel behandelt die folgenden Themen:

- Vererbung
- Interfaces
- Assoziationen
- Multiplizität
- Aggregation
- Komposition

Die Unified Modeling Language (UML) ist eine weit verbreitete grafische Beschreibungssprache für Klassen, Objekte, Zustände, Abläufe und noch mehr. Sie wird vornehmlich in der Phase der Analyse und des Softwareentwurfs eingesetzt. Auf die UML-Grundlagen wird hier nicht eingegangen; dafür gibt es gute Bücher wie [Oe]. Hier geht es darum, die wichtigsten UML-Elemente aus Klassendiagrammen in C++-Konstruktionen, die der Bedeutung des Diagramms möglichst gut entsprechen, umzusetzen. Die vorgestellten C++-Konstruktionen sind Muster, die als Vorlage dienen können. Diese Muster sind nicht einzigartig, sondern nur Empfehlungen, wie man die Umsetzung gestalten kann. Im Einzelfall kann eine Variation sinnvoll sein.

21.1 Vererbung

Über Vererbung ist in diesem Buch schon einiges gesagt worden, das hier nicht wiederholt werden muss, siehe Kapitel 7 ab Seite 257 oder kurz im Glossar Seite 957. Die Abbildung 21.1 zeigt das zugehörige UML-Diagramm.

Abbildung 21.1: Vererbung

In vielen Darstellungen wird die Oberklasse oberhalb der abgeleiteten Unterklasse dargestellt; in der UML ist aber nur der Pfeil mit dem Dreieck entscheidend, nicht die relative Lage. In C++ wird Vererbung syntaktisch durch »`: public`« ausgedrückt:

```
class Unterklasse : public Oberklasse {
  // ... Rest weggelassen
};
```

21.2 Interface anbieten und nutzen

Interface anbieten

Die Abbildung 21.2 zeigt das zugehörige UML-Diagramm. Die Klasse Anbieter implementiert das Interface Schnittstelle-X. Bei der Vererbung stellt die abgeleitete Klasse die Schnittstellen der Oberklasse zur Verfügung. Insofern gibt es eine Ähnlichkeit, auch gekennzeichnet durch die gestrichelte Linie im Vergleich zum vorherigen Diagramm.

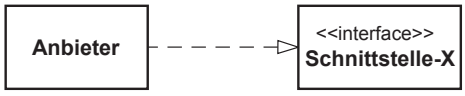

Abbildung 21.2: Interface-Anbieter

Die Ähnlichkeit wird in der Umsetzung nach C++ abgebildet: `Anbieter` wird von `SchnittstelleX`[1] abgeleitet. Um klarzustellen, dass um ein Interface geht, sollte `SchnittstelleX` abstrakt sein. Das Datenobjekt `d` wird nicht als `const`-Referenz übergeben, weil `service()` damit auch die Ergebnisse an den Aufrufer übermittelt.

[1] Die UML erlaubt Bindestriche in Namen, C++ nicht.

```
class SchnittstelleX {
  public:
    virtual void service(Daten& d) = 0; // abstrakte Klasse, vgl. Seite 275 f.
};

class Anbieter : public SchnittstelleX {
  public:
    void service(Daten& d) {
      // ... Implementation der Schnittstelle
    }
};
```

Interface nutzen

Bei der Nutzung des Interfaces bedient sich der Nutzer einer entsprechenden Methode des Anbieters. Die Abbildung 21.3 zeigt das zugehörige UML-Diagramm.

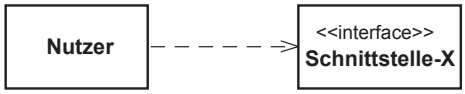

Abbildung 21.3: Interface-Nutzer

Ein `Nutzer` muss ein `Anbieter`-Objekt kennen, damit der Service genutzt werden kann. Aus diesem Grund wird in der folgenden Klasse bereits dem Konstruktor von `Nutzer` ein `Anbieter`-Objekt übergeben, und zwar per Referenz, nicht per Zeiger. Der Grund: Zeiger können NULL sein, aber undefinierte Referenzen gibt es nicht.

```
class Nutzer {
  public:
    Nutzer(Anbieter& a)
    : anbieter(a) {
      daten = ...
    }
    void nutzen() {
       anbieter.service(daten);
    }
  private:
    Daten daten;
    Anbieter& anbieter;
};
```

Nun kann man sich fragen, warum die Referenz oben nicht als `const` übergeben wird. Das kann je nach Anwendungsfall sinnvoll sein oder auch nicht. Es hängt davon ab, ob sich der Zustand des `Anbieter`-Objekts durch den Aufruf der Funktion `service(daten)` ändert. Wenn ja, zum Beispiel durch interne Protokollierung der Aufrufe, entfällt `const`.

21.3 Assoziation

Eine Assoziation sagt zunächt einmal nur aus, dass zwei Klassen in einer Beziehung (mit Ausnahme der Vererbung) stehen. Die Art der Beziehung und zu wie vielen Objekten sie aufgebaut wird, kann variieren. In der Regel gelten Assoziationen während der Lebensdauer der beteiligten Objekte. Nur kurzzeitige Verbindungen werden meistens nicht notiert. Ein Beispiel für eine kurzzeitige Verbindung ist der Aufruf `anbieter.service(daten);` oben: `anbieter` kennt durch die Parameterübergabe das Objekt `daten`, wird aber vermutlich die Verbindung nach Ablauf der Funktion lösen.

Einfache gerichtete Assoziation

Das UML-Diagramm einer einfachen gerichteten Assoziation sehen Sie in Abbildung 21.4.

Abbildung 21.4: Gerichtete Assoziation

Mit »gerichtet« ist gemeint, dass die Umkehrung nicht gilt, wie zum Beispiel die Beziehung »ist Vater von«. Falls zwar Klasse1 die Klasse2 kennt, aber nicht umgekehrt, wird dies durch ein kleines Kreuz bei Klasse1 vermerkt. Es kann natürlich sein, dass eine Beziehung zwischen zwei Objekten *derselben* Klasse besteht. Im UML-Diagramm führt dann der von einer Klasse ausgehende Pfeil auf dieselbe Klasse zurück. In C++ wird eine einfache gerichtete Assoziation durch ein Attribut `zeigerAufKlasse2` realisiert:

```
class Klasse1 {
  public:
    Klasse1() {
    : zeigerAufKlasse2(0) {
    }
    void setKlasse2(Klasse2* ptr2) {
      zeigerAufKlasse2 = ptr2;
    }
  private:
    Klasse2* zeigerAufKlasse2;
};
```

Ein Zeiger ist hier besser als eine Referenz geeignet, weil es sein kann, dass das Kennenlernen erst nach dem Konstruktoraufruf geschieht.

Gerichtete Assoziation mit Multiplizität

Die Multiplizität, auch Kardinalität genannt, gibt an, zu wie vielen Objekten eine Verbindung aufgebaut werden kann. In Abbildung 21.5 bedeutet die 1, dass jedes Objekt der Klasse2 zu genau einem Objekt der Klasse1 gehört. Das Sternchen * bei Klasse2 besagt,

dass einem Objekt der Klasse1 beliebig viele (auch 0) Objekte der Klasse2 zugeordnet sind.

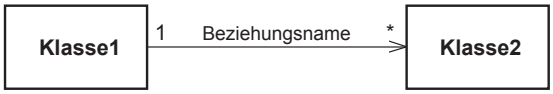

Abbildung 21.5: Gerichtete Assoziation mit Multiplizitäten

Im folgenden C++-Beispiel entspricht `Zeitung` der Klasse1 und `Abonnent` der Klasse2. Eine Zeitung kennt N Abonnenten. Ein Abonnent hat diese Zeitung abonniert oder nicht. Im letzeren Fall ist er entweder Abonnent einer anderen Zeitung oder kein Abonnent, allenfalls ein potentieller. Um die Multiplizität auszudrücken, bietet sich ein `vector` an:

```
class Abonnent; // Vorwärtsdeklaration

class Zeitung {
   public:
      void addAbonnent(Abonnent* a) {
         abonnenten.push_back(a);
      }
   private:
     std::vector<Abonnent*> abonnenten;
};

class Abonnent {
   public:
      void abonniere(const Zeitung& z) {
         meineZeitung = &z;
      }
   private:
     Zeitung* meineZeitung;
};
```

Im Beispiel sind beide Klassen voneinander abhängig. Die Vorwärtsdeklaration ist notwendig, damit der Compiler beim Compilieren der Klasse `Zeitung` die dort angesprochene Klasse `Abonnent` kennt.

Mehr zur Auflösung von gegenseitigen Abhängigkeiten lesen Sie in Abschnitt 4.8.

Einfache ungerichtete Assoziation

Eine ungerichtete Assoziation wirkt in beiden Richtungen und heißt deswegen auch bidirektionale Assoziation. Die Abbildung 21.6 zeigt das UML-Diagramm.

Wenn zwei sich kennenlernen, kann das mit einer ungerichteten Assoziation modelliert werden. Zur Abwechslung sei die Umsetzung in C++ nicht mit zwei, sondern nur mit einer Klasse (namens `Person`) gezeigt. Das heißt, die Klasse hat eine Beziehung zu sich selbst, siehe Abbildung 21.7. Solche Assoziationen werden auch rekursiv genannt und dienen zur Darstellung der Beziehung verschiedener Objekte derselben Klasse.

Abbildung 21.6: Ungerichtete Assoziation

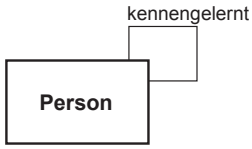

Abbildung 21.7: Rekursive Assoziation

Die Umsetzung in C++ wird am Beispiel von Personen gezeigt, die sich gegenseitig kennenlernen. Ein Aufruf `A.lerntkennen(B);` impliziert, dass `B` auch `A` kennenlernt:

Listing 21.1: Assoziation: Personen lernen sich kennen

```
// cppbuch/k21/main.cpp
#include "Person.h"

int main() {
  Person mabuse("Dr. Mabuse");
  Person klicko("Witwe Klicko");
  Person holle("Frau Holle");
  mabuse.lerntkennen(klicko);
  holle.lerntkennen(klicko);
  // ...
}
```

Die entscheidende Methode der Klasse `Person` ist `lerntkennen(Person& p)` (siehe unten). Beim Eintrag in die Menge der Bekannten wird festgestellt, ob der Eintrag vorher schon vorhanden war. Wenn nicht, wird er auch auf der Gegenseite vorgenommen.

Listing 21.2: Klasse Person

```
// cppbuch/k21/Person.h
#ifndef PERSON_H_
#define PERSON_H_
#include<iostream>
#include<set>
#include<string>

class Person {
public:
  Person(const std::string& name) :
    name_(name) {
  }
```

```
  virtual ~Person() {
  }

  const std::string& getName() const {
     return name_;
  }

  void lerntkennen(Person& p) {
    bool nichtvorhanden = bekannte.insert(p.getName()).second;
     if (nichtvorhanden) { // falls unbekannt, auch bei p eintragen
        p.lerntkennen(*this);
     }
  }

  void bekannteZeigen() const {
     std::cout << "Die Bekannten von " << getName() << " sind:" << std::endl;
     for (auto iter = bekannte.begin(); iter != bekannte.end(); ++iter) {
        std::cout << *iter << std::endl;
     }
  }
private:
  std::string name_;
  std::set<std::string> bekannte;
};
#endif
```

21.3.1 Aggregation

Die »Teil-Ganzes«-Beziehung (englisch *part of*) wird auch *Aggregation* genannt. Sie besagt, dass ein Objekt aus mehreren Teilen besteht (die wiederum aus Teilen bestehen können). Die Abbildung 21.8 zeigt das UML-Diagramm. Die Struktur entspricht der gerichteten Assoziation, sodass deren Umsetzung in C++ hier Anwendung finden kann. Ein Teil kann für sich allein bestehen, also auch vom Ganzen gelöst werden. Letzteres geschieht in C++ durch Nullsetzen des entsprechenden Zeigers.

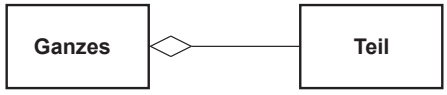

Abbildung 21.8: Aggregation

21.3.2 Komposition

Die Komposition ist eine spezielle Art der Aggregation, bei der die Existenz der Teile vom Ganzen abhängt. Damit ist gemeint, dass die Teile zusammen mit dem Ganzen erzeugt und auch wieder vernichtet werden. Ein Teil ist somit stets genau einem Ganzen zugeordnet; die Multiplizität kann also nur 1 sein. Die Abbildung 21.9 zeigt das UML-Diagramm.

Abbildung 21.9: Komposition

Es empfiehlt sich, bei der Umsetzung in C++ Werte statt Zeiger zu nehmen. Dann ist gewährleistet, dass die Lebensdauer der Teile an das Ganze gebunden ist:

```
class Ganzes {
public:
   Ganzes(int datenFuerTeil1, int datenFuerTeil2)
   : erstesTeil(datenFuerTeil1), zweitesTeil(datenFuerTeil2) {
     // ...
   }
   // ...
private:
   Teil erstesTeil;
   Teil zweitesTeil;
};
```