André Willms

Einstieg in Visual C++ 2008

Galileo Press

Auf einen Blick

Inhalt

Es ist bemerkenswert, dass nur vielleicht 10 % aller Programmierer Programme ohne Verwendung von Flussdiagrammen erfolgreich schreiben können. Unglücklicherweise glauben aber 90 %, dass sie der Gruppe dieser 10 % angehören.
– Rodnay Zaks

11 Vererbung

Abgesehen von der Tatsache, dass Vererbung ein wesentliches Abstraktionsprinzip der Objektorientierten Programmierung darstellt, dient sie hauptsächlich einem einzigen Zweck: der Erweiterung bestehender Funktionalitäten.

Angenommen, die Klasse `Becher` aus Kapitel 10 soll um die Fähigkeit erweitert werden, einen Aufdruck zu besitzen (wie z. B. »Weltbester C++-Programmierer«). Sie könnten diese Eigenschaft problemlos in die Klasse `Becher` einbauen.

Dann tritt der Nächste mit einem Wunsch an Sie heran: Die Klasse `Becher` soll verschiedene »Stoffe« in unterschiedlicher Menge aufnehmen können (z. B. Kaffee, Milch und Zucker). Auch das bauen Sie in die Klasse ein. Nun hat der zukünftige Mischer aber auch noch den vorhin implementierten Aufdruck dabei, obwohl er ihn vielleicht nicht benötigt.

Sie können dieses Beispiel weiterspinnen, bis Sie eine riesige, monolithische Klasse erhalten, die zwar fast alles kann, wovon der Anwender aber immer nur maximal 10 % benötigt. Dieses Problem wird mit Vererbung umgangen.

Ein anderes Problem ist der Wunsch, eine Klasse zu erweitern, deren Quellcode Ihnen nicht zur Verfügung steht. Auch das ist mit Vererbung lösbar.

11.1 Das Wesen der Vererbung

Die in fast allen Fällen verwendete Vererbung ist die öffentliche Vererbung, die ein »ist ein(e)«-Verhältnis zwischen Klassen zum Ausdruck bringt.[1]

1 C++ kennt noch zwei andere Arten der Vererbung. Da vom .NET Framework aber nur die öffentliche Vererbung unterstützt wird, wollen wir uns hier auf sie beschränken. Weitere Informationen finden Sie in [Willms05].

Als einfaches Studienbeispiel soll die `Becher`-Klasse, wie im vorigen Abschnitt angesprochen, mit der Möglichkeit erweitert werden, einen Aufdruck zu bekommen. Wir können sagen: Ein Becher mit Aufdruck ist ein Becher, der zusätzlich noch einen Aufdruck besitzt. Der Kern dieser Aussage lautet: Ein Becher mit Aufdruck ist ein Becher. Das ist auch logisch, denn ein Becher mit Aufdruck kann all das, was auch ein normaler Becher kann, er hat eben nur noch einen Aufdruck.

Und genau das macht die Vererbung. Die Klasse, an die vererbt wird (auch »Subklasse« oder »abgeleitete Klasse« genannt), erbt alle Eigenschaften der vererbenden Klasse (auch Basisklasse oder Superklasse genannt) und kann diese nach Bedarf erweitern.

Bei der Vererbung stehen die öffentlichen Elemente der Basisklasse als öffentliche Elemente der Subklasse zur Verfügung. Die geschützten Elemente der Basisklasse sind als geschützte Elemente der Subklasse vorhanden.

Nur: Die privaten Elemente der Basisklasse werden zwar auch an die Subklasse vererbt, sie sind aber von der Subklasse aus nicht ansprechbar. Abbildung 11.1 stellt den Zusammenhang grafisch dar.

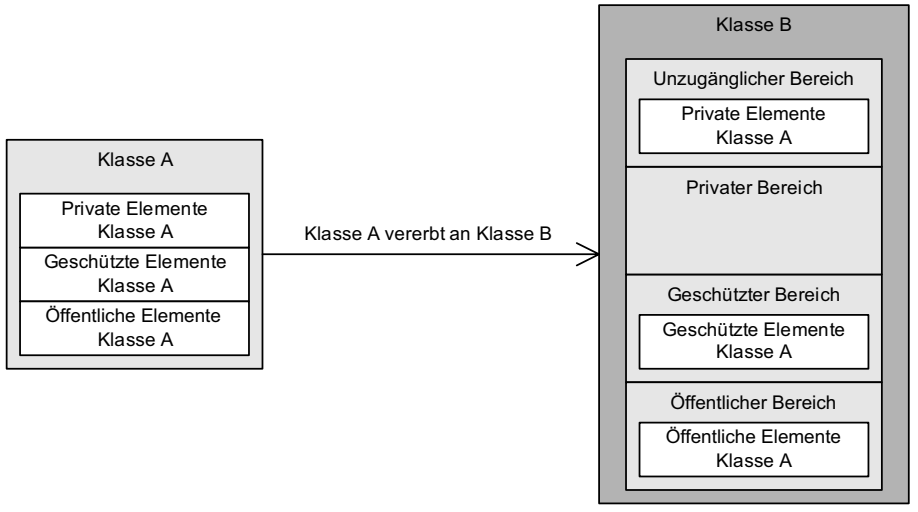

Abbildung 11.1 Das Prinzip der öffentlichen Vererbung

Es ist vielleicht etwas gewöhnungsbedürftig, dass im obigen Fall Klasse B Elemente von Klasse A geerbt hat, auf die sie nicht zugreifen kann. Aber andererseits ist dies eine logische Folge aus der Forderung, dass auf private Elemente einer Klasse nur die Klasse selbst zugreifen kann.

11.2 Die Syntax der Vererbung

Syntaktisch ist die Vererbung schnell umgesetzt. Die neue Klasse wird zunächst, wie in Abschnitt 10.1, »Definition einer Klasse«, beschrieben, mit leerem Anweisungsblock angelegt.

```
class BecherMitAufdruck
{
}
```

Hinter dem Klassennamen wird, durch einen Doppelpunkt getrennt, der Name der Basisklasse (in unserem Fall Becher) angegeben:

```
class BecherMitAufdruck : public Becher
{
}
```

Listing 11.1 Die Syntax der Vererbung

Das Schlüsselwort `public` vor dem Basisklassennamen leitet die gewünschte öffentliche Vererbung ein.

Sie erinnern sich, dass die Klasse `Becher` im Namensbereich `Getraenke` steht. Grundsätzlich ist es sinnvoll, die Klasse `BecherMitAufdruck` ebenfalls in diesem Namensbereich unterzubringen:

```
namespace Getraenke
  class BecherMitAufdruck : public Becher {
  }
}
```

Listing 11.2 Die Klasse »Becher/MitAufdruck« im Namensbereich Getraenke

Sollte es aus logischer Sicht keinen Sinn ergeben, die Klasse `BecherMitAufdruck` in den Namensbereich `Getraenke` zu setzen, dann muss auf die Klasse `Becher` mit expliziter Angabe ihres Namensbereichs verwiesen werden. Es ist dabei unerheblich, ob `BecherMitAufdruck` in einem anderen oder in gar keinem Namensbereich steht:

```
class BecherMitAufdruck : public Getraenke::Becher
{
}
```

Listing 11.3 Angabe des Namensbereichs der Basisklasse

Im weiteren Verlauf soll der Ansatz aus Listing 11.2 weiterverfolgt werden.

11.2.1 Vererbung mit Visual C++

Die Entwicklungsumgebung unterstützt auch die Vererbung. Wenn eine Klasse, wie in Abschnitt 10.1.1 beschrieben, mit dem Klassenassistenten der Entwicklungsumgebung erstellt wird, erscheint irgendwann das in Abbildung 11.2 gezeigte Fenster.

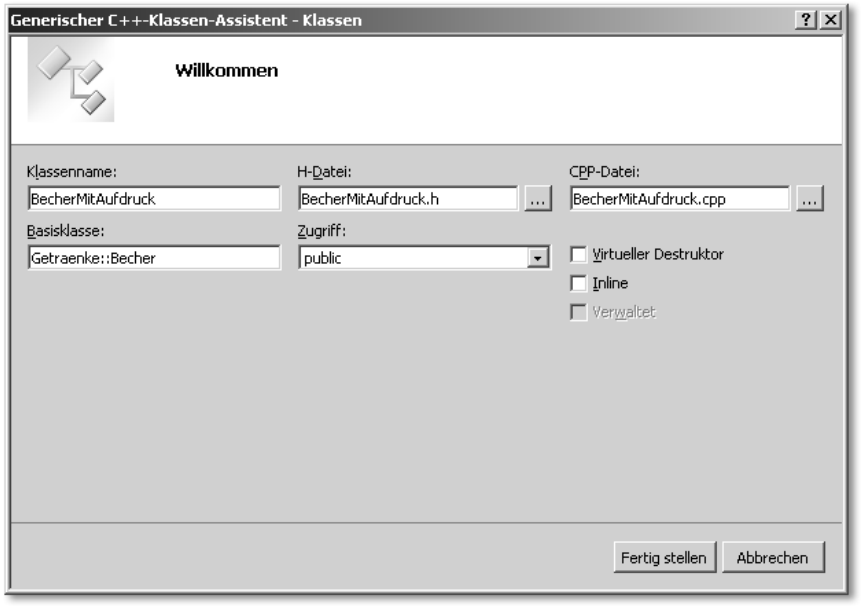

Abbildung 11.2 Die Vererbung im Klassenassistenten

Unter dem Punkt **Basisklasse** wird der Name der Basisklasse eingetragen. Der Assistent erstellt die Datei immer im globalen Namensbereich, deswegen muss der Namensbereich der Basisklasse auf jeden Fall angegeben werden.

Bei Zugriff ist `public` bereits eingetragen und muss nicht mehr verändert werden. Nach abschließendem Klick auf **Fertig stellen** erzeugt der Assistent die Header- und Quellcode-Datei für die Klasse. Die Header-Datei sieht so aus:

```
#pragma once
#include "becher.h"

class BecherMitAufdruck :
  public Getraenke::Becher
{
public:
  BecherMitAufdruck(void);
public:
```

```
  ~BecherMitAufdruck(void);
};
```

Listing 11.4 Vom Assistenten erzeugte »BecherMitAufdruck.h«

Der Assistent hat automatisch die Datei »becher.h« eingebunden, denn zum Ableiten muss dem Compiler die Definition der Basisklasse bekannt sein.

Auch besitzt die Klasse wieder einen Destruktor, dessen Bedeutung in Abschnitt 12.3, »Destruktoren«, besprochen wird und erst einmal entfernt werden kann. Auch das `void` in der Parameterliste des Konstruktors ist unnötig und kann gelöscht werden. Und der gewünschte Namensbereich muss noch manuell hinzugefügt werden. Nach entsprechender kosmetischer Aufbereitung sieht die Datei so aus:

```
#pragma once
#include "becher.h"

namespace Getraenke {
  class BecherMitAufdruck : public Becher {
  public:
    BecherMitAufdruck();
  };
}
```

Listing 11.5 Die aufbereitete »BecherMitAufdruck.h«

Die Datei »BecherMitAufdruck.cpp« muss dann noch entsprechend angepasst werden.

11.3 Konstruktoren

Völlig nebensächlich, ob Sie mit dem Klassenassistenten gearbeitet oder die Klassen »zu Fuß« erstellt haben, werden Sie die bittere Enttäuschung eines Fehlers während der Kompilation erfahren.

Und zwar beschwert sich der Compiler über einen fehlenden Standardkonstruktor in der Klasse `Becher`.

Ein guter Zeitpunkt also, ein wenig die Konstruktion von Objekten abgeleiteter Klassen zu beleuchten. Wir können der Lösung bereits mit einigen einfachen Fragen erstaunlich nahekommen.

Wie wird in einer Klasse ein Objekt erzeugt? Über den Konstruktor. Wie konstruiert ein Konstruktor ein Objekt? Indem er – entweder im Anweisungsblock oder in der Elementinitialisierungsliste – die Attribute des Objekts initialisiert – meist mithilfe übergebener Argumente. Hier tritt schon der erste Fauxpas zutage: Unser Konstruktor besitzt noch keine sinnvollen Parameter.

Diese kleine formale Unschärfe einmal kurz beiseite geschoben, tritt ein neues Problem auf: Wie soll der Konstruktor von `BecherMitAufdruck` die Objektattribute initialisieren? Einfach durch Zuweisung oder über die Elementinitialisierungsliste? Eher nicht, denn wie Abbildung 11.1 schön darstellt, können die Elemente der Subklasse – und dazu gehört der Konstruktor von `BecherMitAufdruck` – nicht auf die privaten Elemente der Basisklasse zugreifen.

Es wird also etwas benötigt, das auf der einen Seite Zugriff auf die privaten Elemente der Basisklasse hat – es kommt also nur ein Element der Basisklasse infrage – und in der Lage ist, die Basisklassenattribute zu initialisieren.

Einmal kurz über den Quellcode von Becher meditiert findet sich der geeignete Kandidat: Der Konstruktor von `Becher`. Er ist öffentlich und daher von der Basisklasse aus ansprechbar und er weiß, wie die Attribute zu initialisieren sind. Wir brauchen also nur vom Subklassenkonstruktor aus den Basisklassenkonstruktor aufzurufen.

Dummerweise benötigt der Basisklassenkonstruktor zur Initialisierung der Attribute drei Argumente. Damit ihm diese Argumente übergeben werden können, muss der Subklassenkonstruktor diese vom Anwender einfordern.

Die grobe Richtung steht also fest: Der Basisklassenkonstruktor benötigt drei Argumente, die der Subklassenkonstruktor über eigene Parameter beschaffen muss. Betrachten wir dazu kurz die Datei »BecherMitAufdruck.cpp«, die den aktualisierten Konstruktor enthält:

```
#include "BecherMitAufdruck.h"

using namespace std;

namespace Getraenke {

  BecherMitAufdruck::BecherMitAufdruck(string i,
                                       int fa,
                                       float fu) {
  }
}
```

Listing 11.6 Die Datei »BecherMitAufdruck.cpp«

Die Konstruktordeklaration in der Klassendefinition muss ebenfalls mit den Parametern versehen werden.

Kompilieren lässt sich das Programm aber immer noch nicht, denn der Basisklassenkonstruktor wurde noch nicht aufgerufen. Dafür gilt:

> Der Basisklassenkonstruktor wird in der Elementinitialisierungsliste des Subklassenkonstruktors aufgerufen.

Der endgültige Konstruktor sieht so aus:

```
BecherMitAufdruck::BecherMitAufdruck(string i,
                                     int fa,
                                     float fu)
: Becher(i, fa, fu) {

}
```

Listing 11.7 Aufruf des Basisklassenkonstruktors

Und wenn Sie sich fragen, wie der ursprüngliche Fehler entstanden ist und was er bedeutet:

> In C++ *muss* der Subklassenkonstruktor einen Basisklassenkonstruktor aufrufen.

Welcher Konstruktor aufgerufen wird – es können durch Überladung mehrere existieren –, spielt keine Rolle. Diese Regel ist so wichtig, dass selbst dann ein Basisklassenkonstruktor aufgerufen wird, wenn der Programmierer keinen expliziten Aufruf angegeben hat:

> Besitzt der Subklassenkonstruktor keinen expliziten Aufruf eines Basisklassenkonstruktors, dann wird der Standardkonstruktor der Basisklasse aufgerufen.

Als Standardkonstruktor wird der Konstruktor ohne Parameter bezeichnet. Zu Beginn wurde also automatisch dieser Standardkonstruktor von `Becher` aufgerufen. Da wir einen solchen aber nicht programmiert haben, kam die Fehlermeldung »kein geeigneter Standardkonstruktor verfügbar«.

Nun endlich kann ein Objekt der neuen Klasse erstellt werden:

```
BecherMitAufdruck b("Milch", 300, 90);
```

Die Klasse `BecherMitAufdruck` hat durch die Vererbung alle Fähigkeiten der Basisklasse `Becher` geerbt. Ohne weiter entwicklerisch aktiv zu werden, kann ein

`BecherMitAufdruck`-Objekt daher über die geerbte Methode `ausgabe` ausgegeben werden:

```
b.ausgabe();
```

Alle anderen in `Becher` implementierten Methoden sind ebenfalls verfügbar.

11.4 Erweitern durch Vererbung

Der bisher betriebene Aufwand für die Klasse `BecherMitAufdruck` hat uns zu dem Punkt gebracht, dass sie exakt die gleichen Fähigkeiten besitzt wie ihre Basisklasse `Becher`. Das klingt nicht nach einem großartigen Gewinn, aber: Wir haben über die Vererbung die Funktionalitäten von `Becher` übernommen, ohne die Klasse `Becher` verändern zu müssen. Diese Technik funktioniert daher auch bei Klassen, auf deren Quellcode wir keinen Zugriff haben.[2]

Die Klasse `BecherMitAufdruck` kann jetzt allerdings nach Belieben erweitert werden und soll im weiteren Verlauf ihrem Namen alle Ehre machen.

Der Becheraufdruck soll in der Klasse als Text gespeichert werden. Wir benötigen dazu in `BecherMitAufdruck` ein entsprechendes Attribut. Damit dieses Attribut initialisiert werden kann, muss der Konstruktor um einen Parameter erweitert werden:

```
class BecherMitAufdruck : public Becher {
  std::string aufdruck;
public:
  BecherMitAufdruck(std::string i, int fa, float fu,
                    std::string auf);
};
```

Listing 11.8 Die Klasse »BecherMitAufdruck« mit neuem Attribut

Das neue Attribut steht direkt am Klassenanfang und besitzt deshalb implizit privates Zugriffsrecht. Im obigen Beispiel wurde der zusätzliche Konstruktorparameter an die Parameterliste angehängt. Die Parameterreihenfolge ist allerdings beliebig und kann nach Bedarf verändert werden. Die Parameterlisten von Deklaration und Definition müssen aber übereinstimmen.

Die Definition des Konstruktors sieht so aus:

```
BecherMitAufdruck::BecherMitAufdruck(string i,
                                     int fa,
```

2 In diese Rubrik fallen später die Klassen der .NET-Bibliothek.

```
                                    float fu,
                                    string auf)
: Becher(i, fa, fu), aufdruck(auf) {
}
```

Listing 11.9 Der neue Konstruktor von »BecherMitAufdruck«

Weitere Funktionalität kann auf diese Weise nach Belieben hinzugefügt werden. Diese Zugriffsmethode für den Aufdruck wäre beispielsweise nicht schlecht:

```
std::string getAufdruck() const {
  return(aufdruck);
}
```

Listing 11.10 Die Zugriffsmethode »getAufdruck«

Oben ist nur die Definition der Methode zu sehen. Handelt es sich bei dieser Definition um eine externe oder um eine Inline-Definition?

Wäre es eine externe Definition, müsste angegeben werden, zu welcher Klasse die Methode gehört (`BecherMitAufdruck::getAufdruck`), insofern muss es sich um eine Methodendefinition innerhalb der Klassendefinition handeln.

Falls Sie nicht mehr genau wissen, welche Auswirkungen das `const` hinter dem Methodenkopf hat, dann schlagen Sie schnell in Abschnitt 10.6, »Konstanzwahrende Methoden«, nach.

11.5 Methoden überschreiben

Sie wissen von den vorigen Abschnitten, dass in der Klasse `BecherMitAufdruck` durch die Vererbungsbeziehung zu `Becher` deren Methode `ausgabe` geerbt wurde und aufgerufen werden kann.

Nun wäre es für die Klasse `BecherMitAufdruck` aber schön, wenn bei der Ausgabe auch der Becheraufdruck ausgegeben würde. Aber wie geht das?

Die bestehende `ausgabe`-Methode entsprechend abzuändern, ist aus einem einfachen Grund nicht möglich: Die Methode gehört zu `Becher` und dort existiert das auszugebende Attribut `aufdruck` noch nicht.

Die Konsequenz daraus führt zu einer neuen Methode für `BecherMitAufdruck`. Eine Methode mit demselben Namen ist auf den ersten Blick nicht möglich, denn die Parameterliste würde sich nicht von der geerbten Methode unterscheiden und ein Überladen unmöglich machen.

Eine Methode mit anderem Namen zu implementieren ist auch keine saubere Lösung, denn es würde weiterhin die geerbte `ausgabe`-Methode zur Verfügung stehen, die den Aufdruck nicht mit ausgibt.

Die Lösung ist erschreckend simpel: Wenn wir in `BecherMitAufdruck` eine Methode `ausgabe` implementieren, dann ist das kein Überladen. Zur Erinnerung[3]: Überladen ist ein Name nur dann, wenn er mehrfach im selben Bezugsrahmen definiert ist. Die geerbte Methode gehört zum Bezugsrahmen `Becher`, die neue Methode wird zum Bezugsrahmen `BecherMitAufdruck` gehören. Unterschiedliche Bezugsrahmen, daher kein Überladen, vielmehr:

> Wird in einer abgeleiteten Klasse eine Methode mit gleichem Namen und Parameterliste einer geerbten Methode definiert, wird die geerbte Methode mit der neuen Methode überschrieben.

Praktisch heißt das: Wir können in `BecherMitAufdruck` eine neue `ausgabe`-Methode definieren und haben damit einfach die alte überschrieben:

```
void BecherMitAufdruck::ausgabe() const {
  cout << "Becher mit " << inhalt <<
          " und Aufdruck \"" << aufdruck <<
          "\"" << endl;
}
```

Listing 11.11 Erster Versuch einer ausgabe-Methode

Fertig? Mitnichten! Die Methode wird bei der Kompilation einen Fehler melden. Der Grund müsste Ihnen bereits bekannt sein, wird aber im folgenden Abschnitt noch genauer besprochen.

11.6 Geschützte Attribute

Das Problem bei der Methode aus Listing 11.11 liegt im Zugriff auf das Attribut `inhalt`. Es handelt sich hierbei um ein privates Attribut von `Becher`, weshalb eine Methode von `BecherMitAufdruck` keinen Zugriff darauf hat.

Es gibt nun zwei Möglichkeiten. Die erste besteht darin, das Attribut mit geschütztem Zugriffsrecht zu versehen. Wie in Abschnitt 10.3, »Zugriffsrechte«, bereits beschrieben wurde, erlaubt das geschützte Zugriffsrecht auch abgeleiteten Klassen den Zugriff. Eigentlich genau das, was wir brauchen:

3 Abschnitt 10.7, »Überladen von Methoden«

```
class Becher {
public:
  typedef float AbsWertTyp;
  Becher(std::string i, int fa, float fu);
  void ausgabe() const;
  bool reichtKapazitaet(int ml) const;
  bool reichtKapazitaet(const Becher* b) const;
  static AbsWertTyp berechneAbsolutwert(float gw, float ps);

protected:
  std::string inhalt;

private:
  int fassungsvermoegen;
  float fuellhoehe;
  static int maxmenge;
};
```

Listing 11.12 Die Klasse »Becher« mit geschütztem inhalt-Attribut

Nun lässt sich das Programm problemlos kompilieren. Aber: Eine vollständige Datenkapselung ist nicht mehr gewährleistet, weil auf `inhalt` nun jede abgeleitete Klasse zugreifen kann.

Dieses Zugriffsrecht bezieht sich zwar nur auf das geerbte Attribut. Eine Methode der Klasse `BecherMitAufdruck` ist nicht in der Lage, auf das `inhalt`-Attribut eines `Becher`-Objekts zuzugreifen.

Trotzdem kann eine abgeleitete Klasse eventuell von der Basisklasse auferlegte Beschränkungen umgehen. Aus diesem Grund sieht ein sauberer Ansatz weiterhin private Attribute vor und ermöglicht den Zugriff über Methoden, die nach Bedarf dann mit geschütztem Zugriffsrecht versehen werden können. Dazu wird das Attribut `inhalt` wieder privat und die Klasse um eine öffentliche Methode `getInhalt` erweitert:

```
class Becher {
public:
  typedef float AbsWertTyp;
  Becher(std::string i, int fa, float fu);
  void ausgabe() const;
  bool reichtKapazitaet(int ml) const;
  bool reichtKapazitaet(const Becher* b) const;
  static AbsWertTyp berechneAbsolutwert(float gw, float ps);
  std::string getInhalt() const {
    return(inhalt);
```

```
  }

private:
  std::string inhalt;
  int fassungsvermoegen;
  float fuellhoehe;
  static int maxmenge;
};
```

Listing 11.13 Die Klasse »Becher« mit getInhalt

Die `ausgabe`-Methode von `BecherMitAufdruck` muss nun nur noch entsprechend angepasst werden:

```
void BecherMitAufdruck::ausgabe() const {
  cout << "Becher mit " << getInhalt() <<
          " und Aufdruck \"" << aufdruck <<
          "\"" << endl;
}
```

Listing 11.14 Die endgültige Fassung von »BecherMitAufdruck::ausgabe«

11.7 Polymorphie

Wir wissen mittlerweile, dass die öffentliche Vererbung eine »ist ein(e)«-Beziehung darstellt. In den letzten Abschnitten haben wir deshalb über die Vererbung zum Ausdruck gebracht, dass ein Becher mit Aufdruck ein Becher ist. Im Umkehrschluss muss es deshalb möglich sein, einen Becher mit Aufdruck als gewöhnlichen Becher zu behandeln, denn er ist ja ein Becher. Dieser Sachverhalt wird in der Objektorientierten Programmierung Polymorphie genannt:

> Überall dort, wo ein Objekt der Basisklasse erwartet wird, kann auch ein Objekt einer abgeleiteten Klasse verwendet werden.

Nehmen wir im einfachsten Fall einen Zeiger vom Typ `Becher`:

```
Becher *bptr;
```

Weil ein Becher mit Aufdruck ein Becher ist, kann die Adresse eines `BecherMitAufdruck`-Objekts einem `Becher`-Zeiger zugewiesen werden:

```
BecherMitAufdruck b("Milch", 300, 90, "Meine Privattasse");
bptr = &b;
```

Über diesen Zeiger können dann die `Becher`-Methoden des Objekts aufgerufen werden:

```
cout << bptr->getFuellmenge() << endl;
```

Polymorphie funktioniert nur mit Zeigern und Referenzen. Über den `Becher`-Zeiger können allerdings keine speziellen Methoden der Klasse `BecherMitAufdruck` aufgerufen werden, denn der Zeiger »weiß« schließlich nicht, dass das `Becher`-Objekt, auf das er zeigt, in Wirklichkeit ein `BecherMitAufdruck`-Objekt ist.

Praktischer Einsatz der Polymorphie könnte eine `maximum`-Funktion sein, die zwei `Becher`-Objekte übergeben bekommt und den Becher mit mehr Inhalt zurückliefert:

```
Becher* maximum(Becher* b1, Becher* b2) {
  if(b1->getFuellmenge()>=b2->getFuellmenge())
    return(b1);
  else
    return(b2);
}
```

Listing 11.15 Eine maximum-Funktion für »Becher«

Die Funktion ist so geschrieben, dass bei gleicher Füllmenge das erste Objekt zurückgegeben wird, genau so verhalten sich auch die Funktionen der C++-Standardbibliothek.

Wegen der Polymorphie können `maximum` beliebige Becher übergeben werden, solange deren Klassen von `Becher` abgeleitet sind. Diese Art der Programmierung bietet ein enormes Maß an Wiederverwendbarkeit, denn es kann Programmcode geschrieben werden, der mit Objekten arbeitet, an die der Programmierer bis dato nicht einmal gedacht hat.

11.8 Virtuelle Methoden

Im Rahmen der Polymorphie können Effekte auftreten, die vielleicht nicht auf Anhieb klar sind. Nehmen wir folgendes Beispiel:

```
BecherMitAufdruck b("Milch", 300, 90, "Privattasse");
Becher *bptr = &b;
bptr->ausgabe();
```

Was wird ausgegeben? Oder programmtechnisch gefragt, welche `ausgabe`-Methode wird aufgerufen, die von `Becher` oder die von `BecherMitAufdruck`?

Es gibt zwei Argumente:

- Die `ausgabe`-Methode von `BecherMitAufdruck` wird aufgerufen, weil das Objekt ein `BecherMitAufdruck`-Objekt ist.
- Die `ausgabe`-Methode von `Becher` wird aufgerufen, weil der Zeiger vom Typ `Becher*` ist.

Im Normalfall werden Datentypen zur Kompilationszeit geprüft. Dies wird *statische Typüberprüfung* genannt. Und zur Kompilationszeit zeigt `bptr` auf ein `Becher`-Objekt, weil `bptr` vom Typ `Becher*` ist. Es spielt dabei keine Rolle, dass dem Zeiger während des Programmlaufs ein `BecherMitAufdruck`-Objekt zugewiesen wird.

Es wird also die `ausgabe`-Methode von `Becher` aufgerufen. Obwohl programmtechnisch nachvollziehbar, ist das Verhalten nicht unbedingt erwünscht. Denn nur, weil der Zugriff auf das Objekt über einen Zeiger vom Typ `Becher*` stattfindet, sollte trotzdem die richtige `ausgabe`-Methode aufgerufen werden.

Um die richtige `ausgabe`-Methode aufzurufen, muss das Programm den tatsächlichen Typ des Objekts, auf das `bptr` zeigt, zur Laufzeit prüfen. Dies wird *dynamische Typüberprüfung* genannt.

Aktiviert wird die dynamische Typüberprüfung für eine Methode, indem vor der Deklaration in der Basisklasse das Schlüsselwort **`virtual`** geschrieben wird. Daher wird eine solche Methode in C++ auch virtuelle Methode genannt.

Für die dynamische Typüberprüfung spielt es keine Rolle, wie weit der tatsächliche Typ in der Klassenhierarchie von der Basisklasse entfernt ist. Sollte beispielsweise von der Klasse `BecherMitAufdruck` eine Klasse `BecherMitAufdruckUndBild` abgeleitet werden, würde über den `Becher`-Zeiger, wenn er auf ein solches Objekt zeigt, die `ausgabe`-Methode von `BecherMitAufdruckUndBild` aufgerufen – falls die Methode in der Klasse existiert. Es muss sich aber um eine tatsächliche Überschreibung handeln; die Methode in der Subklasse muss genau so heißen wie die überschriebene Methode und in Parameterliste und Rückgabetyp komplett übereinstimmen.

> Virtuelle Methoden gewährleisten, dass sich das Verhalten eines Objekts nicht ändert, wenn über einen Basisklassenzeiger darauf zugegriffen wird.

Dynamische Typüberprüfung funktioniert nur mit Methoden. Ein Grund mehr, auf den direkten Attributzugriff zu verzichten.

11.9 UML

Um im weiteren Verlauf des Buches eine möglichst klare und einfache Darstellungsmöglichkeit für Klassen und deren Beziehungen einsetzen zu können, die darüber hinaus auch noch weit verbreitet ist, wollen wir diesen Abschnitt dem Klassendiagramm der UML[4] widmen. Die UML definiert einen Satz an Diagrammen, mit denen dynamische und statische Eigenschaften von Methoden und Klassen dargestellt werden können. Die bisher verwendeten Diagramme zur Darstellung des Programmflusses in diesem Buch sind Aktivitätsdiagramme der UML.

Um den statischen Aufbau von Klassen und deren Beziehungen untereinander darzustellen, wird das UML-Klassendiagramm eingesetzt. In Abbildung 11.3 ist die aktuelle Becher-Hierarchie als UML-Klassendiagramm dargestellt.

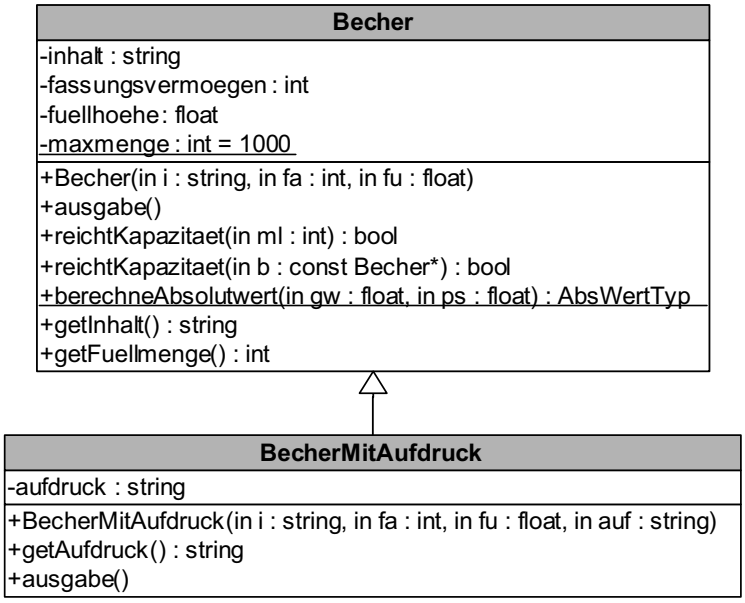

Abbildung 11.3 Die »Becher«-Hierarchie im UML-Klassendiagramm

Im Klassendiagramm wird eine Klasse durch ein in drei Bereiche aufgeteiltes Rechteck repräsentiert. Der obige Bereich beinhaltet den Klassennamen, darunter stehen die Attribute und im untersten Abschnitt die Methoden. Vor den Attributen und Methoden wird mit einem Zeichen das Zugriffsrecht des Elements angezeigt. Dabei steht

4 Abkürzung für »Unified Modeling Language«.

- `-` für privates Zugriffsrecht
- `#` für geschütztes Zugriffsrecht
- `+` für öffentliches Zugriffsrecht

Hinter Attributen steht, durch einen Doppelpunkt getrennt, deren Typ und dahinter optional mit Gleichheitszeichen der Initialisierungswert. Der Initialisierungswert wird nur angegeben, wenn er allgemeingültig ist, was nur bei statischen Elementen der Fall ist.

Statische Elemente sind unterstrichen.

Besitzt eine Methode einen Rückgabetyp, dann steht er, durch Doppelpunkt getrennt, hinter der Parameterliste. Ein Parameter besteht aus seinem Namen und seinem dahinter mit Doppelpunkt folgenden Datentyp.

Optional kann vor dem Parameternamen noch die Übergaberichtung spezifiziert werden. Hier gibt es drei Möglichkeiten:

- `In` – der Parameter dient nur zur Wertübergabe an die Methode. Für ihn muss beim Aufruf ein gültiges Argument angegeben werden. Entspricht der Wert- oder Adressübergabe in C++.
- `inout` – für diesen Parameter muss beim Aufruf ein gültiges Argument angegeben werden. Änderungen am Parameter innerhalb der Methode wirken sich jedoch auf das beim Aufruf übergebene Argument aus. Entspricht in C++ der Übergabe als Referenz.
- `Out` – das an die Methode übergebene Argument dient nur dazu, innerhalb der Methode beschrieben zu werden und muss beim Aufruf keinen gültigen Wert besitzen. Es wird innerhalb der Methode beschrieben und vom Aufrufer ausgelesen. Diese Variante wird von C++ nicht direkt unterstützt, kann aber mit Referenzen simuliert werden.

Die Vererbung wird durch den Generalisierungspfeil zum Ausdruck gebracht, der immer von der Subklasse zur Basisklasse weist.

Im weiteren Verlauf werden jeweils bei Bedarf noch einige Darstellungselemente hinzukommen.

11.10 Schnittstellen

Wir können nun für die Klasse `Becher` eine Bibliothek an Funktionen programmieren[5] und wissen, dass jede Subklasse von `Becher`, egal wie tief sie in der Hierarchie liegen mag, mit dieser Bibliothek arbeiten kann. Und mit den virtuellen

Methoden ist gewährleistet, dass sich das Verhalten der Subklassenobjekte nicht ändert, wenn über einen Basisklassenzeiger auf sie zugegriffen wird.

Da für die Erweiterung der Klassenhierarchie keine Änderungen an der Bibliothek vorgenommen werden müssen, ist die auf `Becher` basierende Klassenhierarchie beliebig erweiterbar. Auch die Funktionsbibliothek kann um Funktionen erweitert werden, ohne dass die tatsächlichen Subklassen von `Becher` bekannt sein müssen.

Die Vererbung erlaubt also ein hohes Maß an Wiederverwendbarkeit, wie es ohne sie nicht möglich wäre.

Der bisherige Ansatz hat nur noch einen Schönheitsfehler: Jede Klasse, die mit der bestehenden Funktionsbibliothek zusammenarbeiten will, muss von der Klasse `Becher` oder einer ihrer Subklassen abgeleitet werden. Was aber, wenn die Objekte der neuen Klasse gar keine Becher sind?

Das Problem entstand dadurch, dass die Funktionsbibliothek für eine konkrete Klasse programmiert wurde. Geschickter wäre es, wenn die Bibliothek nicht von einer konkreten Klasse, sondern lediglich von einer bestimmten Fähigkeit oder Funktionalität abhängig wäre.

Nehmen wir als einfaches Beispiel die Funktionalität der Ausgabe. Wir wollen Funktionen schreiben, die von den zu bearbeitenden Objekten lediglich die Fähigkeit erwarten, ausgegeben werden zu können. Dies soll über eine Methode `ausgabe` geschehen. Wir könnten denselben Fehler wie zuvor begehen und die Funktionen mit Zeigern des Typs `Becher*` versehen, denn `Becher` besitzt eine `ausgabe`-Methode. Wollten wir dann aber vielleicht eine geometrische Form ausgeben, dann müsste diese von `Becher` erben, womit die merkwürdige Aussage »Eine geometrische Form ist ein Becher« getätigt würde, nur um die Funktionsbibliothek nutzen zu können.

Um solch unrealistische Beziehungen zu vermeiden, muss die Basisklasse weiter abstrahiert werden. Wenn die Funktionsbibliothek nur eine `ausgabe`-Methode benötigt, dann sollte die oberste Basisklasse auch nur diese `ausgabe`-Methode besitzen.

Wir werden diese Klasse `IAusgabe`[6] nennen und als Inline-Klasse – also ohne Quellcodedatei – erstellen. Die Header-Datei sieht so aus:

5 Später unter .NET wird es keine Funktionen mehr geben. Dort wird eine solche Bibliothek durch statische Methoden einer Klasse realisiert.

6 Das vorangestellte »I« steht für »Interface«, auf Deutsch Schnittstelle.

```
#pragma once
#include <iostream>
class IAusgabe {
public:
  virtual void ausgabe() const {
    std::cout << "Basis-Ausgabe" << std::endl;
  }
};
```

Listing 11.16 Die Basisklasse »IAusgabe«

Die Methode `ausgabe` muss als virtuell deklariert werden, damit eine Überschreibung in der Subklasse korrekt aufgerufen wird. Die Klasse steht nicht im Namensbereich `Getraenke`, weil sie auch als Basisklasse anderer Hierarchien dienen kann.

Als primitives Beispiel einer auf dieser Basisklasse fußenden Bibliothek soll die folgende Funktion dienen:

```
void simpleAusgabe(IAusgabe* obj) {
  obj->ausgabe();
}
```

Listing 11.17 Die Funktion »simpleAusgabe«

Zugegeben, diese Funktion ist nicht würdig, in einer praxisrelevanten Bibliothek aufgenommen zu werden, sie demonstriert aber auf einfache Weise, worauf es ankommt. Auch eine komplexe Funktion wird nicht anders auf das Objekt zugreifen.

Noch kann die Klasse `Becher` aber nicht von der neuen Bibliothek bearbeitet werden, denn sie hat `IAusgabe` noch nicht als Basisklasse. Das ist schnell nachgeholt:

```
#pragma once
#include "IAusgabe.h"
#include <string>

namespace Getraenke {

  class Becher : public IAusgabe {
    // Klasseninhalt
  };
}
```

Listing 11.18 »Becher« als Subklasse von »IAusgabe«

Alles bestens. Aber immer noch hat der Ansatz drei Schönheitsfehler, die eine gemeinsame Ursache teilen. Betrachten wir `IAusgabe` etwas genauer. Sie besitzt eine Methode `ausgabe`, auf die die Klassenbibliothek zugreift und die von den Subklassen mit ihren spezifischen Methoden überschrieben wird. Die Methode in `IAusgabe` wird aber nie etwas Sinnvolles ausgeben können, weil sie nicht wissen kann, welche Klassen alle von ihr ableiten. Trotzdem mussten wir die Methode mit einem Anweisungsblock ausstatten, der einen sinnlosen Text ausgibt. Gut, die Ausgabe des Textes hat theatralische Gründe, denn der Anweisungsblock hätte auch leer bleiben können. Eine leere Methode ist aber auch kein sehr viel sauberer Ansatz.

Aber was wirklich verrückt ist: Obwohl die Klasse nichts macht, kann von ihr ein Objekt erzeugt und an eine der Bibliotheksfunktionen übergeben werden:

```
IAusgabe o;
simpleAusgabe(&o);
```

Und noch schlimmer: Da eine von ihr abgeleitete Klasse die unsinnige `ausgabe`-Methode erbt, ist die Subklasse nicht gezwungen, eine eigene `ausgabe`-Methode zu implementieren:

```
class Teddybaer : public IAusgabe {
  std::string fellfarbe;
public:
  Teddybaer(std::string ff)
    : fellfarbe(ff)
  {}
};
```

Listing 11.19 Die Klasse »Teddybaer«

Wenn jetzt jemand einen Teddybären mit der gewünschten Fellfarbe konstruiert, dann wird er bei der Ausgabe vermutlich mit einer Erwähnung eben jener Farbe rechnen, aber Pustekuchen:

```
Teddybaer baer("schwarz");
baer.ausgabe(); // Aufruf von IAusgabe::ausgabe
```

All das sind Auffälligkeiten im Verhalten der Klassenhierarchie, die im Optimalfall vermieden werden sollten.

11.10.1 Rein virtuelle Methoden

Es gibt in der Objektorientierten Programmierung glücklicherweise die Möglichkeit, die Existenz einer Methode einzufordern, ohne sie selbst implementieren zu müssen. Solche Methoden werden *abstrakte Methoden genannt*. C++ bezeichnet

sie auch als *rein virtuelle Methoden*. Deklariert werden sie durch ein `=0` hinter dem Methodenkopf. Schauen wir uns das einmal praktisch an der Klasse `IAusgabe` an:

```
class IAusgabe {
public:
  virtual void ausgabe() const =0;
};
```

Listing 11.20 Die Klasse »IAusgabe« mit rein virtueller Methode

Es wird auffallen, dass es sich bei der rein virtuellen Methode nur noch um eine Deklaration handelt. Eine Definition – also ausführbarer Programmcode – ist nicht mehr notwendig. Doch die abstrakte Methode hat weitreichende Konsequenzen:

> Eine Klasse mit abstrakten Methoden ist eine abstrakte Klasse. Von abstrakten Klassen kann kein Objekt erzeugt werden.

Demnach kann `IAusgabe` nicht mehr instanziiert werden:

```
IAusgabe o; // Fehler
```

Eine abstrakte Methode wird erst dann konkret, wenn sie in einer Subklasse mit einer konkreten Methode – also einer Methode mit Definition – überschrieben wurde.

Dies ist bisher in der Klasse `Teddybaer` noch nicht geschehen. Die Klasse hat die abstrakte Methode von `IAusgabe` geerbt und ist damit ebenfalls eine abstrakte Klasse. Um einen Teddybären erzeugen zu können, muss `ausgabe` überschrieben werden:

```
class Teddybaer : public IAusgabe {
  std::string fellfarbe;
public:
  Teddybaer(std::string ff)
    : fellfarbe(ff)
  {}
  void ausgabe() const {
    std::cout << "Baer mit Fellfarbe " << fellfarbe
              << std::endl;
  }
};
```

Listing 11.21 Die Klasse »Teddybaer« mit eigener »ausgabe«-Methode

Nun kann auch wieder ein `Teddybaer`-Objekt erzeugt werden. Auf diese Weise wird jeder Programmierer, der von `IAusgabe` ableitet, gezwungen, eine `ausgabe`-Methode zu programmieren. Leider kann er nicht gezwungen werden, eine sinnvolle Methode zu implementieren, aber dass er eine implementieren muss, ist viel wert. Denn die auf `ausgabe` zurückgreifende Bibliothek hat die Gewissheit, dass immer eine `ausgabe`-Methode vorhanden ist. Denn eine Klasse ohne `ausgabe`-Methode ist abstrakt; von ihr kann kein Objekt erzeugt werden.

Eine abstrakte Klasse muss dabei nicht zwangsläufig von der direkt folgenden Subklasse implementiert werden. Im Gegenteil, eine Subklasse kann noch weitere abstrakte Methoden hinzufügen, wie `IGeometrischeFigur` zeigt:

```
namespace Geometrie {
  class IGeometrischeFigur : public IAusgabe {
  public:
    virtual double umfang() const =0;
    virtual double flaeche() const =0;
  };
}
```

Listing 11.22 Die Klasse »IGeometrischeFigur«

Die Klasse `IGeometrischeFigur` könnte die Basisklasse einer anderen Bibliothek sein, die mit Flächen und Umfängen arbeitet. Weil sie von `IAusgabe` abgeleitet ist, können ihre Subklassen zusätzlich noch die Ausgabebibliothek verwenden, die bisher nur aus der Methode `simpleAusgabe` besteht. Um `IGeometrischeFigur` von anderen Klassen abzugrenzen, wurde sie in einen eigenen Namensbereich `Geometrie` gesetzt. Eine Subklasse von `IGeometrischeFigur` muss die abstrakten Methoden von `IGeometrischeFigur` und `IAusgabe` implementieren, damit Objekte von ihr erzeugt werden können. Als Beispiel sei hier die Klasse `Rechteck` vorgestellt:

```
namespace Geometrie {
  class Rechteck : public IGeometrischeFigur {
    double x,y,b,h;
  public:
    Rechteck(double x, double y, double b, double h)
      : x(x), y(y), b(b), h(h)
    {}
    void ausgabe() const {
      std::cout << "Rechteck mit Breite " << b
        << " und Hoehe " << h << std::endl;
    }
    double umfang() const {
      return(2*b+2*h);
```

```
    }
    double flaeche() const {
      return(b*h);
    }
  };
}
```

Listing 11.23 Die Klasse »Rechteck«

Etwas merkwürdig mag der Konstruktor erscheinen, weil die Parameter denselben Namen wie die Attribute haben. Obwohl es problemlos funktioniert, sollte es in der Praxis vermieden werden. Da Sie in Ihrer C++-Karriere aber aller Wahrscheinlichkeit nach auch Code anderer Personen in die Finger bekommen werden, kann es nicht schaden, in loser Folge einige Unarten zu demonstrieren, damit Sie gewappnet sind. Deswegen gleich eine Frage: Welches `h` wird in der Ausgabe des Konstruktors unten ausgegeben?

```
Rechteck(double x, double y, double b, double h)
  : x(x), y(y), b(b), h(h) {
    std::cout << h << std::endl;
}
```

In diesem Fall spielt es natürlich keine Rolle, welches `h` ausgegeben wird, weil beide den gleichen Wert haben. Es gibt aber Situationen, in denen die Frage entscheidend ist.

Die Frage kann mit einer anderen Frage beantwortet werden: Welches `h` ist lokaler? Eindeutig der Parameter. Deshalb wird in der Ausgabe auch der Inhalt des Parameters ausgegeben. Aber wie kann das Attribut trotzdem angesprochen werden, obwohl es vom Parameter verdeckt ist? Wir verwenden einfach den Objektzeiger `this`[7]:

```
Rechteck(double x, double y, double b, double h)
  : x(x), y(y), b(b), h(h) {
    std::cout << h << std::endl;       // Parameter
    std::cout << this->h << std::endl; // Attribut
}
```

Schauen wir uns abschließend noch in Abbildung 11.4 die aktuelle Klassenhierarchie an:

7 Abschnitt 10.4.2.

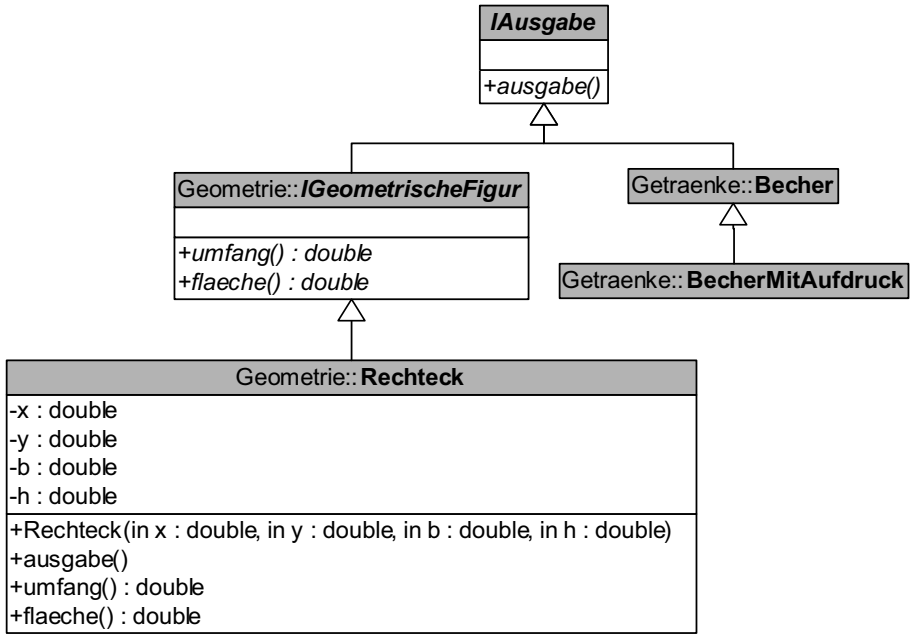

Abbildung 11.4 Die aktuelle Klassenhierarchie

Die Klassen `Becher` und `BecherMitAufdruck` sind ohne ihre Attribute und Methoden dargestellt, weil sich an ihnen nichts geändert hat. Wo vorhanden, sind die Namensbereiche der Klassen mit angegeben.

Im Diagramm ist zu erkennen, dass abstrakte Klassen und Methoden kursiv dargestellt werden.

11.10.2 Rein abstrakte Klassen

Klassen, die als Elemente nur abstrakte Methoden besitzen, werden als *rein abstrakte Klassen* bezeichnet. Sie entsprechen in Standard-C++ den Schnittstellen, die nicht direkt von C++ unterstützt werden, unter .NET (Abschnitt 18.5, »Schnittstellen«) aber eine wichtige Rolle spielen.

Grundsätzlich gilt als Regel – von der es natürlich auch Ausnahmen gibt –, dass ganz oben in der Klassenhierarchie immer eine Schnittstelle bzw. eine abstrakte Klasse stehen sollte.

11.11 Downcasts

Betrachten wir folgenden Codeschnipsel:

```
BecherMitAufdruck b("Tee", 200, 95, "Top-Becher");
Becher * bptr=&b;
```

Wir wissen, dass wegen der Polymorphie die Adresse des `BecherMitAufdruck`-Objekts in einem Zeiger des Typs `Becher*` gespeichert werden kann. Über diesen Zeiger können alle Methoden von `Becher` aufgerufen werden. Sollte es sich um virtuelle Methoden handeln, dann wird die Methode des tatsächlichen Typs aufgerufen, wie in Abschnitt 11.8, »Virtuelle Methoden«, gezeigt wurde.

Über den `Becher`-Zeiger können aber keine Methoden aufgerufen werden, die in der Subklasse neu angelegt wurden, wie in diesem Fall `getAufdruck`.

Manchmal ist aber genau das notwendig. Dazu muss der ursprüngliche Typ des Objekts über eine Typumwandlung wiederhergestellt werden. Der dafür notwendige Cast nennt sich **`dynamic_cast`**:

```
BecherMitAufdruck *p =
                  dynamic_cast<BecherMitAufdruck*>(bptr);
```

Die in `bptr` gespeicherte Adresse wird in den Typ `BecherMitAufdruck*` umgewandelt und dem Zeiger `p` zugewiesen. Wir wissen an dieser Stelle, dass `bptr` tatsächlich auf ein `BecherMitAufdruck`-Objekt zeigt. Häufig wird der `dynamic_cast` aber in Situationen angewandt, bei denen keine hundertprozentige Klarheit darüber herrscht, von welchem Typ das Objekt wirklich ist. Sollte das Objekt nicht von dem Typ sein, in den umgewandelt wird, schlägt die Umwandlung fehl. Es ist in der Praxis daher meist nötig, den Erfolg der Umwandlung zu überprüfen. Sollte die Umwandlung misslingen, liefert der Cast einen Nullzeiger zurück:

```
if(p)
  cout << p->getAufdruck() << endl;
```

Der Aufruf der `BecherMitAufdruck`-Methode wird nur dann ausgeführt, wenn die Umwandlung erfolgreich war.

11.12 Zusammenfassung

Dieses Kapitel hat Ihnen den Mechanismus der Vererbung nähergebracht. Vererbung wird hauptsächlich eingesetzt, um bestehende Klassen zu erweitern.

Die bestehende Klasse wird Basisklasse genannt, die erweiterte Klasse Subklasse. Bei der hier besprochenen öffentlichen Vererbung werden die öffentlichen und

geschützten Elemente der Basisklasse zu öffentlichen und geschützten Elementen der Subklasse. Diese erbt zwar auch die privaten Elemente der Basisklasse, hat aber keinen Zugriff darauf.

Der Konstruktor der Subklasse muss einen Basisklassenkonstruktor aufrufen, andernfalls wird implizit der Standardkonstruktor der Basisklasse aufgerufen.

Geerbte Methoden können überschrieben werden. Soll auch bei einem Aufruf über einen Basisklassenzeiger die überschreibende Methode der Subklasse aufgerufen werden, muss die Basisklassenmethode als virtuell deklariert werden.

Methoden können als abstrakt (rein virtuell) deklariert werden. Eine abstrakte Methode besitzt keinen Anweisungsblock und macht ihre Klasse ebenfalls abstrakt. Von einer abstrakten Klasse können keine Objekte erzeugt werden.

Eine von einer abstrakten Klasse abgeleitete Klasse ist ebenfalls abstrakt, bis sie die die geerbten abstrakten Methoden mit eigenen konkreten Methoden überschreibt.

11.13 Übungen

1. Sie möchten ein Programm schreiben, mit dem Sie Ihre auf Musikkassetten, CDs und DVDs befindliche Musiksammlung verwalten können. Erstellen Sie entsprechende Klassen. Welche Elemente besitzen diese Klassen und wie stehen die Klassen im Zusammenhang? Berücksichtigen Sie bei Ihrem Design typische Aktivitäten bei der Arbeit mit einem Archiv, wie »Finde alle Alben eines Interpreten«, Finde das Lied mit dem Titel XYZ« etc.
2. Sie planen, systematisch Ihre Vorfahren und Verwandten elektronisch zu erfassen und benötigen dazu ein Programm, mit dem die verwandtschaftlichen Verhältnisse datentechnisch abgebildet werden können. Überlegen Sie sich sinnvolle Klassen.

Wer weinende Erben hinterlassen will,
darf keine Lebensversicherung abschließen.
– Marcel Pagnol

18 Vererbung II

Vererbung war bereits Thema in Kapitel 11. An dieser Stelle soll lediglich besprochen werden, wie sich unter .NET die Vererbung syntaktisch verändert und in ihren Möglichkeiten erweitert hat.

Zur Demonstration sollen einfache, aber sinnlose Klassen herhalten, bei denen der Fokus auf das programmtechnische Verhalten gelegt werden kann. Begonnen wird mit der Klasse `Basis`, die später die Rolle der Basisklasse übernehmen soll:

```
ref class Basis {
  int basis;
public:
  Basis(int i)
    : basis(i)
  {}
  property int Wert {
    int get() {
      return(basis);
    }
    void set(int i) {
      basis=i;
    }
  }
  void ausgabe() {
    System::Console::WriteLine("Basis:"+basis);
  }
};
```

Listing 18.1 Die Klasse Basis

Sie besitzt als Attribut einen `int`-Wert, der über den Konstruktor initialisiert wird. Für den Zugriff steht eine Eigenschaft `Wert` und die Methode `ausgabe` zur Verfügung. Sie wissen bereits, wie eigene Klassen mit der `Console`-Klasse ausge-

geben werden können[1], um aber nicht bereits eine geerbte Methode überschreiben zu müssen (denn jede verwaltete Klasse hat `Object` als Basisklasse und erbt damit die `ToString`-Methode), wird zu Anschauungszwecken eine eigene Ausgabemethode implementiert. Ein Objekt von `Basis` ist schnell angelegt und die Methoden ausgetestet:

```
Basis^ b=gcnew Basis(20);
b->ausgabe();
Console::WriteLine(b->Wert);
```

Nun folgt eine von `Basis` abgeleitete Klasse mit dem passenden Namen `Abgeleitet`. Sie definiert eine eigene `ausgabe`-Methode und eine neue Eigenschaft Wert:

```
ref class Abgeleitet : Basis {
  int ab;
public:
  Abgeleitet(int i, int a)
    : Basis(i), ab(a)
  {}
  property int Wert {
    int get() {
      return(ab);
    }
    void set(int i) {
      ab=i;
    }
  }
  void ausgabe() {
    System::Console::WriteLine("Abgeleitet:"+ab);
  }
};
```

Listing 18.2 Die Klasse »Abgeleitet«

Ein Unterschied zu ANSI C++ tritt bereits zutage: Weil es unter .NET nur öffentliche Vererbung gibt, kann auf das Schlüsselwort `public` vor der Basisklasse verzichtet werden.

Auch von der abgeleiteten Klasse ist ein Objekt problemlos erstellt:

```
Abgeleitet^ a=gcnew Abgeleitet(20,30);
a->ausgabe();
Console::WriteLine(a->Wert);
```

Als Nächstes soll über einen Basisklassenzeiger zugegriffen werden:

1 Abschnitt 17.2, »Ausgabe«

```
b=a;
b->ausgabe();
Console::WriteLine(b->Wert);
```

Über den Basisklassenzeiger verhält sich das `Abgeleitet`-Objekt wie ein `Basis`-Objekt. Mit den grundlegenden Vererbungsmechanismen bereits vertraut, erkennen Sie die Ursache. Die Methoden der Basisklasse sind nicht virtuell.

Basisklassen sollten die potenziell für Überschreiben infrage kommenden Methoden immer als virtuell deklarieren. Das folgende Listing zeigt die Änderungen in komprimierter Darstellung:

```
ref class Basis {
  int basis;
public:
  Basis(int i) : basis(i) {}
  property int Wert {
    virtual int get() {return(basis);}
    virtual void set(int i) {basis=i;}
  }
  virtual void ausgabe() {
    System::Console::WriteLine("Basis:"+basis);
  }
};
```

Listing 18.3 Die Klasse »Basis« mit virtuellen Funktionen

Der Compiler wird sich nun beschweren, dass die Methoden der abgeleiteten Klasse keine Kennzeichnung haben, ob es sich um eine überschreibende oder eine neue Methode handelt. Um das gewünschte Verhalten zu erreichen, müssen die Methoden der Basisklasse überschrieben und die Methoden der abgeleiteten Klasse daher mit `override` versehen werden. Zusätzlich muss die Subklasse ihre Methoden explizit als virtuell deklarieren:

```
ref class Abgeleitet : Basis {
  int ab;
public:
  Abgeleitet(int i, int a) : Basis(i), ab(a) {}
  property int Wert {
    virtual int get() override {return(ab);}
    virtual void set(int i) override {ab=i;}
  }
   virtual void ausgabe() override {
    System::Console::WriteLine("Abgeleitet:"+ab);
  }
};
```

Listing 18.4 Die Klasse »Abgeleitet« mit virtuellen Methoden

18.1 override vs. new

Um den Unterschied zwischen `override` und `new` zu erklären, soll folgende Klassenhierarchie aus vier Klassen herhalten, die allesamt eine Methode `Print` besitzen, mit der ausgegeben wird, um welche Klasse es sich handelt. Die Methoden sind virtuell und überschreiben jeweils die Methode der Basisklasse, damit bei Aufrufen über Basisklassenzeiger die korrekte Methode aufgerufen wird:

```
ref class A {
public:
  virtual void Print() {Console::WriteLine("A");}
};
ref class B : public A {
public:
  virtual void Print() override {Console::WriteLine("B");}
};
ref class C : public B {
public:
  virtual void Print() override {Console::WriteLine("C");}
};
ref class D : public C {
public:
  virtual void Print() override {Console::WriteLine("D");}
};
```

Listing 18.5 Eine Klassenhierarchie

Nun werden zwei `D`-Objekte erzeugt und unterschiedlichen Basisklassenzeigern zugewiesen. Die Ergebnisse der Methodenaufrufe sind wie erwartet:

```
A ^a=gcnew D;
C ^c=gcnew D;
a->Print();   // Ausgabe D
c->Print();   // Ausgabe D
```

Jetzt wird die Methode in der Klasse `C` mit **`new`** deklariert:

```
ref class C : public B {
public:
  virtual void Print() new {Console::WriteLine("C");}
};
```

Durch `new` beginnt für die Methode `Print` in Klasse `C` eine neue Klassenhierarchie. Die dynamische Typprüfung endet für `Print` daher vor `C`, also bei Klasse `B`, beziehungsweise beginnt bei `C` neu.

Oder anders ausgedrückt: Die Methode `Print` der Klasse `B` wird in Klasse `C` nicht überschrieben, sondern die Methode `Print` in `C` ist eine komplett neue Methode.

Wenn über ein Trackinghandle des Typs `A` oder `B` jetzt die `Print`-Methode eines Typs `D` aufgerufen wird, dann wird die Print-Methode von `B` ausgeführt, da ab `C` eine neue Hierarchie beginnt. Für alle Zeiger ab Klasse `C` bleibt das Verhalten gleich:

```
A ^a=gcnew D;
C ^c=gcnew D;
a->Print();   // Ausgabe B
c->Print();   // Ausgabe D
```

Diese Eigenschaft wird zugegebenermaßen weitaus seltener eingesetzt als `override`.

18.2 Abstrakte Methoden und Klassen

Beim Klassendesign kommt es häufiger vor, dass eine Basisklasse bereits eine gewisse Funktionalität zur Verfügung stellt, zur korrekten Ausführung dieser Funktionalität aber Informationen notwendig sind, die erst in den Subklassen bereitgestellt werden.

Nehmen wir die Klasse `Mitarbeiter`. Sie kann eine Methode `getMonatsgehalt` bereitstellen, die aus dem Jahresgehalt das durchschnittliche Monatsgehalt ermittelt. Wie das Jahresgehalt ausfällt, entscheidet sich aber erst in der Subklasse, weil ein einfacher Angestellter höchstwahrscheinlich eine andere Gehaltstruktur haben wird als ein Abteilungsleiter. Werfen wir einen Blick auf einen ersten Entwurf der Klasse `Mitarbeiter`:

```
ref class Mitarbeiter {
public:
  float getMonatsgehalt() {
    return(getJahresGehalt()/12);
  }
};
```

Die Klasse wird sich so noch nicht kompilieren lassen, weil der Compiler nicht weiß, wo die Methode `getJahresgehalt` zu finden ist. Sie muss in der Klasse verfügbar gemacht werden, aber in den Subklassen durch eine spezialisiertere Variante ersetzt werden können. Die Wahl fällt daher auf eine virtuelle Methode.

Die Methode kann in `Mitarbeiter` aber keinen sinnvollen Rückgabewert liefern, also muss es eine bereits besprochene rein virtuelle Methode[2] werden, die auch

2 Abschnitt 11.10.1

abstrakte Methode genannt wird. Unter .NET kann zwar auch die aus ANSI C++ bekannte Schreibweise `=0` verwendet werden, passender ist aber das Schlüsselwort **`abstract`**.

Eine Klasse mit abstrakten Methoden wird automatisch ebenfalls abstrakt, das heißt, von ihr können keine Objekte erzeugt werden. Das macht im Falle der Klasse `Mitarbeiter` auch Sinn, denn ihr fehlt schließlich eine funktionsfähige Methode `getJahresgehalt`. Obwohl die Klasse durch die abstrakte Methode schon abstrakt ist, muss sie unter .NET explizit als abstrakt deklariert werden:

```
ref class Mitarbeiter abstract {
public:
  float getMonatsgehalt() {
    return(getJahresgehalt()/12);
  }

  virtual float getJahresgehalt() abstract;
};
```

Listing 18.6 Die Klasse »Mitarbeiter«

Eine von einer abstrakten Basisklasse abgeleitete Subklasse erbt die abstrakte Methode und ist zunächst auch abstrakt. Erst, wenn alle abstrakten Methoden überschrieben wurden, wird die Klasse konkret und kann instanziiert werden.

Unter .NET ist es auch möglich, eine Klasse ohne abstrakte Methoden als abstrakt zu deklarieren. Eine davon abgeleitete Klasse ist automatisch konkret (weil sie keine abstrakten Methoden geerbt hat).

Doch zurück zu unserer Firmenhierarchie. Um eine konkrete Klasse zu erhalten, soll die Klasse `Abteilungsleiter` abgeleitet werden, die ein Jahresgehalt und Weihnachtsgeld erhält:

```
ref class Abteilungsleiter : Mitarbeiter {
  float jahresgehalt;
  float weihnachtsgeld;
public:
  Abteilungsleiter(float jg, float wg)
    : jahresgehalt(jg), weihnachtsgeld(wg)
  {}
  virtual float getJahresgehalt() override {
    return(jahresgehalt+weihnachtsgeld);
  }
};
```

Listing 18.7 Die Klasse »Abteilungsleiter«

Von dieser Klasse kann ein Objekt erzeugt und die geerbte Methode `getMonatsgehalt` aufgerufen werden:

```
Abteilungsleiter^ p=gcnew Abteilungsleiter(4000,2000);
Console::WriteLine(p->getMonatsgehalt());
```

18.3 Versiegelte Methoden

Eine Methode ist versiegelt, wenn sie nicht mehr überschrieben werden kann.

Falls Sie zu dem Schluss kommen sollten, die Methode `getJahresgehalt` in `Abteilungsleiter` sei der Weisheit letzter Schluss und einfach undenkbar, dass jemand eine noch bessere oder sinnvollere Methode implementieren könnte, dann deklarieren Sie die Methode als **`sealed`**:

```
ref class Abteilungsleiter : Mitarbeiter {
  float jahresgehalt;
  float weihnachtsgeld;
public:
  Abteilungsleiter(float jg, float wg)
    : jahresgehalt(jg), weihnachtsgeld(wg)
  {}
  virtual float getJahresgehalt() override sealed {
    return(jahresgehalt+weihnachtsgeld);
  }
};
```

Listing 18.8 Eine versiegelte Methode in »Abteilungsleiter«

Nur virtuelle Methoden können versiegelt werden.

18.4 Versiegelte Klassen

Es ist aber auch möglich, eine komplette Klasse zu versiegeln. Von dieser Klasse kann dann nicht mehr abgeleitet werden.

In der darzustellenden Firmenhierarchie gibt es vielleicht keine Erweiterung mehr von `Abteilungsleiter` und Sie wollen eventuellen Versuchen, doch noch eine Subklasse zu erstellen, im Vorfeld bereits eine Abfuhr erteilen, dann deklarieren Sie einfach die Klasse als `sealed`:

```
ref class Abteilungsleiter sealed : Mitarbeiter {
  float jahresgehalt;
```

```
  float weihnachtsgeld;
public:
  Abteilungsleiter(float jg, float wg)
    : jahresgehalt(jg), weihnachtsgeld(wg)
  {}
  virtual float getJahresgehalt() override sealed {
    return(jahresgehalt+weihnachtsgeld);
  }
};
```

Listing 18.9 Die versiegelte Klasse »Abteilungsleiter«

Das `sealed` bei der Methode `getJahresgehalt` könnte theoretisch eingespart werden, denn wenn von einer Klasse nicht mehr abgeleitet werden kann, dann ist auch das Überschreiben einer ihrer Methoden nicht mehr möglich.

Die beiden Klassen funktionieren einwandfrei, sind aber nicht gerade .NET-mäßig programmiert. Unter .NET sollten die Zugriffsmethoden durch Eigenschaften ersetzt werden:

```
ref class Mitarbeiter abstract {
public:
  property float Monatsgehalt {
    float get() {
      return(Jahresgehalt/12);
    }
  }
  property float Jahresgehalt {
    virtual float get() abstract;
  }
};

ref class Abteilungsleiter sealed : Mitarbeiter {
  float jahresgehalt;
  float weihnachtsgeld;
public:
  Abteilungsleiter(float jg, float wg)
    : jahresgehalt(jg), weihnachtsgeld(wg)
  {}
  property float Jahresgehalt {
    virtual float get() override sealed {
      return(jahresgehalt+weihnachtsgeld);
    }
  }
};
```

Listing 18.10 »Mitarbeiter« und »Abteilungsleiter« mit Eigenschaften

Die `get`- und `set`-Funktionen der Eigenschaften können ebenso virtuell, abstrakt oder versiegelt sein wie normale Methoden. Wichtig ist nur, dass sich die Deklarationen auf die einzelnen Funktionen der Eigenschaft beziehen, nicht auf die gesamte Eigenschaft. Eine Eigenschaft könnte theoretisch eine abstrakt-virtuelle `get`-Funktion und eine versiegelte `set`-Funktion besitzen.

18.5 Schnittstellen

In Abschnitt 11.10, »Schnittstellen«, wurde das Grundprinzip von Schnittstellen bereits besprochen. Hier soll die Definition einer Schnittstelle unter .NET behandelt werden. Schnittstellen kommen immer dann zum Einsatz, wenn an eine Subklasse gestellte Anforderungen von der Basisklasse nicht geleistet werden können oder sollen. Gewissermaßen war das auch Thema des vorangegangenen Abschnitts, denn die Klasse `Mitarbeiter` hat von ihren Subklassen eine Eigenschaft `Jahresgehalt` gefordert.

In ANSI C++ ist eine Schnittstelle nichts anderes als eine Klasse, die nur aus rein virtuellen Methoden besteht. Unter .NET existiert ein eigenes Konstrukt, dass mit dem Schlüsselwort **`interface`** eingeleitet wird. Im Folgenden wird eine Schnittstelle `IAusgebbar` definiert, die eine Methode `ausgeben` fordert:

```
interface class IAusgebbar {
  void ausgeben();
};
```

Listing 18.11 Definition einer Schnittstelle

Eine Klasse, die von der Schnittstelle ableitet – im Zusammenhang mit Schnittstellen wird die Formulierung »Eine Klasse implementiert eine Schnittstelle« verwendet – ist abstrakt, bis sie die von der Schnittstelle geforderten Methoden implementiert. Die in der Schnittstelle deklarierten Methoden entsprechen in ihrem Verhalten einer abstrakten Methode.

Syntaktisch gibt es zu Klassen und deren Methoden aber einige Besonderheiten:

- Die Methoden einer Schnittstelle sind automatisch öffentlich, eine Angabe von `public` ist nicht notwendig. Das ist logisch, denn nur eine öffentliche Methode ist überschreibbar und von außen zugänglich.
- Um eine Methode sinnvoll überschreiben bzw. implementieren zu können, muss sie virtuell sein. Eine Methode ohne Definition ist darüber hinaus auch noch abstrakt. Aus diesem Grund ist eine Schnittstellenmethode immer virtuell und abstrakt, ohne dass die entsprechenden Schlüsselwörter angegeben werden müssen.

- Schnittstellen dürfen Deklarationen von Methoden, Eigenschaften und deren Funktionen sowie Ereignissen[3] enthalten.
- Eine Schnittstelle darf keine Definitionen – also Anweisungsblöcke – für Methoden, Eigenschaftsfunktionen oder Ereignisse besitzen.
- Die Definition von statischen Elementen (Attribute, Eigenschaften, Methoden, Ereignisse) ist allerdings erlaubt.
- Von einer Schnittstelle können Wert- und Verweisklassen sowie andere Schnittstellen abgeleitet werden.
- Die implementierende Methode der Subklasse darf kein Schlüsselwort `override` besitzen, da die Schnittstellenmethode nicht überschrieben, sondern implementiert wird.

Die Klasse `Ganzzahl` implementiert die Schnittstelle `IAusgebbar`:

```
ref class Ganzzahl : IAusgebbar {
  int zahl;
public:
  Ganzzahl(int i)
    :zahl(i)
  {}
  virtual void ausgeben() {
    System::Console::WriteLine(zahl);
  }
};
```

Listing 18.12 Die Klasse »Ganzzahl«

Wie oben erwähnt, ist die implementierte Methode virtuell, besitzt aber kein `override`.

18.5.1 Schnittstellen und Mehrfachvererbung

.NET unterstützt zwar keine Mehrfachvererbung, aber eine Klasse kann mehrere Schnittstellen implementieren. In den meisten Fällen lässt sich ein auf Mehrfachvererbung basierendes Klassendesign auf die Mehrfachimplementierung von Schnittstellen reduzieren, von daher treten die Vorteile einer echten Mehrfachvererbung gegenüber der damit einhergehenden Nachteile und Schwierigkeiten in den Hintergrund.

Zur Demonstration soll das obige Beispiel um eine Schnittstelle `IGanzzahlig` ergänzt werden, die `Ganzzahl` dann ebenfalls implementiert. Für Abwechslung sorgt der Einsatz einer Eigenschaft anstelle einer Methode:

3 Eigenschaften sind Thema von Abschnitt 21.2, »Ereignisse«.

```
interface class IGanzzahlig {
  property int Zahl {
    int get();
  }
};
```

Listing 18.13 Die Schnittstelle »IGanzzahlig«

Es folgt die Implementierung in der Klasse:

```
ref class Ganzzahl : IAusgebbar, IGanzzahlig {
  int zahl;
public:
  Ganzzahl(int i)
    :zahl(i)
  {}
  virtual void ausgeben() {
    System::Console::WriteLine(zahl);
  }
  property int Zahl {
    virtual int get() {
      return(zahl);
    }
  }
};
```

Listing 18.14 Mehrfachimplementierung bei »Ganzzahl«

Die zu implementierenden Schnittstellen werden einfach durch Komma getrennt im Klassenkopf aufgeführt.

18.5.2 Identische Methoden

Wenn es möglich ist, in einer Klasse mehrere Schnittstellen zu implementieren, dann kann es theoretisch auch passieren, dass zwei Schnittstellen zwei Methoden mit demselben Namen und derselben Signatur vorgeben:

```
interface class A {
  void ausgabe();
};

interface class B {
  void ausgabe();
};
```

Listing 18.15 Zwei Schnittstellen mit derselben Methode

Wenn diese beiden Methoden dieselbe Bestimmung haben – es also ausreicht, für beide Methoden eine Implementierung zu besitzen – unterscheidet sich der Mechanismus nicht von dem bisher angewandten:

```
ref class C : A, B {
public:
  virtual void ausgabe() {
    Console::WriteLine("Ausgabe");
  }
};
```

Listing 18.16 Gemeinsame Implementierung

Unerheblich, über welche Schnittstelle die Methode ausgerufen wird, das Ergebnis ist dasselbe:

```
C^ c=gcnew C;
A^ a=c;
B^ b=c;
a->ausgabe();
b->ausgabe();
```

Es ist aber durchaus möglich, dass die Methoden der beiden Schnittstellen, obwohl gleichen Namens, eine unterschiedliche Implementierung benötigen. Dazu ist eine spezielle Syntax notwendig, weil es keine zwei Methoden im selben Gültigkeitsbereich mit gleichem Namen und Signatur geben kann:

```
ref class C : A, B {
public:
  virtual void ausgabea() = A::ausgabe {
    Console::WriteLine("A");
  }
  virtual void ausgabeb() = B::ausgabe {
    Console::WriteLine("B");
  }
};
```

Listing 18.17 Individuelle Implementierung

Die Namen der Methoden in der implementierenden Klasse (hier `ausgabea` und `ausgabeb`) sind beliebig wählbar. Der Aufruf über die Schnittstellentypen bleibt gleich:

```
C^ c=gcnew C;
A^ a=c;
B^ b=c;
```

```
a->ausgabe();
b->ausgabe();
```

Müssen die Methoden über `C` aufgerufen werden, dann ist der neue Name zu verwenden:

```
c->ausgabea();
c->ausgabeb();
```

18.6 Zusammenfassung

In diesem Kapitel wurde die .NET-Vererbung unter die Lupe genommen.

Das Überschreiben von virtuellen Methoden muss mit dem Schlüsselwort `override` kenntlich gemacht werden. Zusätzlich gibt es mit `new` die Möglichkeit, eine geerbte Methode nicht zu überschreiben, sondern eine neue Hierarchie für diese Methode zu beginnen.

Schnittstellen werden unter .NET durch eine eigene Konstruktion (`interface`) unterstützt. Die in einer Schnittstelle deklarierten Methoden sind automatisch öffentlich und abstrakt.

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z