

Kapitel 21

Vererbung in C++

Vererbung

Vererbung ist ein Strukturierungsprinzip, durch das eine **Ist-Ein Beziehung** zwischen Klassen modelliert wird. Wir sprechen in diesem Zusammenhang auch von Generalisierung und Spezialisierung. Die beiden Begriffe unterscheiden sich dabei eigentlich nicht, sie betrachten nur den selben Prozess von unterschiedlichen Standpunkten aus.

Bei der **Spezialisierung** entstehen neue Klassen (abgeleitete Klassen, Unterklassen, Kind-Klassen oder Children) durch die Detaillierung und Konkretisierung bestehender Klassen (Basisklassen, Oberklassen, Elternklassen oder Parents).

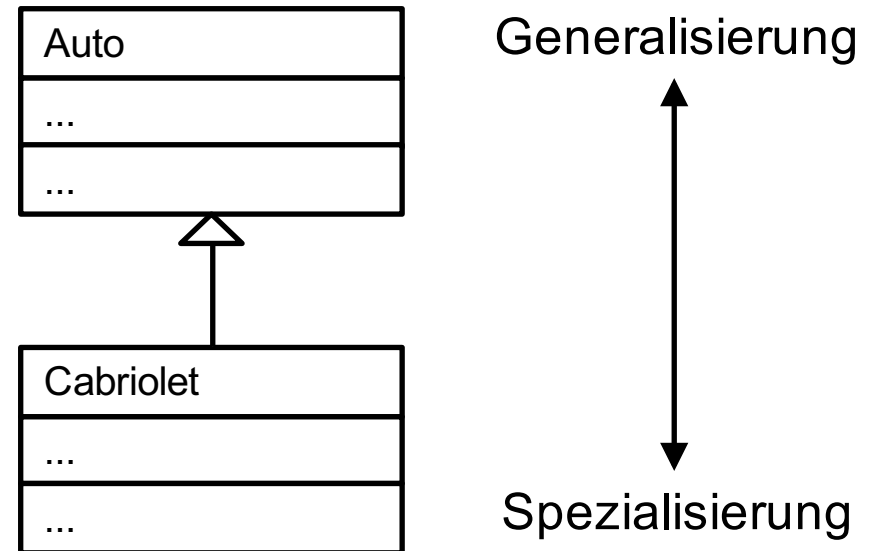
Bei der **Generalisierung** entstehen neue Klassen durch Abstraktion aus bestehenden Klassen.

Kindklassen erben die Attribute und Methoden ihrer Elternklassen und können diese verwenden. Dazu können sie bei Bedarf weitere Attribute und Methoden ausprägen oder bestehenden Methoden modifizieren.

Darstellung der Vererbung

In UML kennzeichnen wir Vererbungs-Beziehungen durch einen Pfeil:

Wir kennen diese Prinzip der generalisierten und spezialisierten Betrachtung aus unserem Alltag, wo wir es ständig verwenden. Wir spezialisieren, indem wir einen Begriff mit zusätzlichen Details anreichern. Wir generalisieren, indem wir störende Details weglassen und uns auf das Wesentliche konzentrieren.



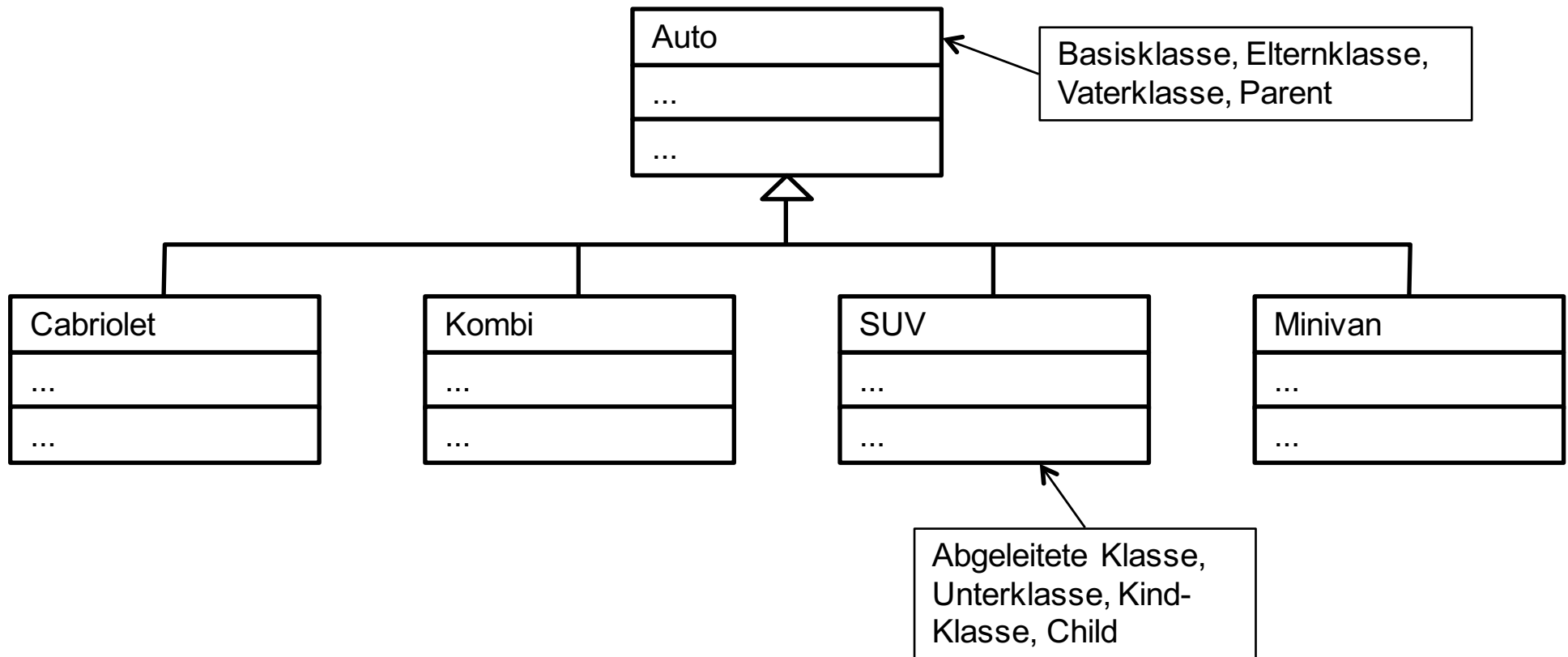
Wir haben zum Beispiel aus der Beobachtung der uns umgebenden Welt durch Generalisierung den Begriff „Auto“ entwickelt. Dazu haben wir keine genaue Checkliste bekommen, anhand derer wir feststellen könnten, wann ein Objekt ein Auto ist. Trotzdem sind wir in der Lage, so verschieden aussehende Autos wie ein Cabriolet, einen Kombi oder einen Minivan als Auto zu identifizieren, auch wenn wir das betreffende Modell noch nie zuvor gesehen haben.

Wir wissen:

- Ein Cabriolet ist ein Auto
- Ein Kombi ist ein Auto
- Ein Minivan ist ein Auto

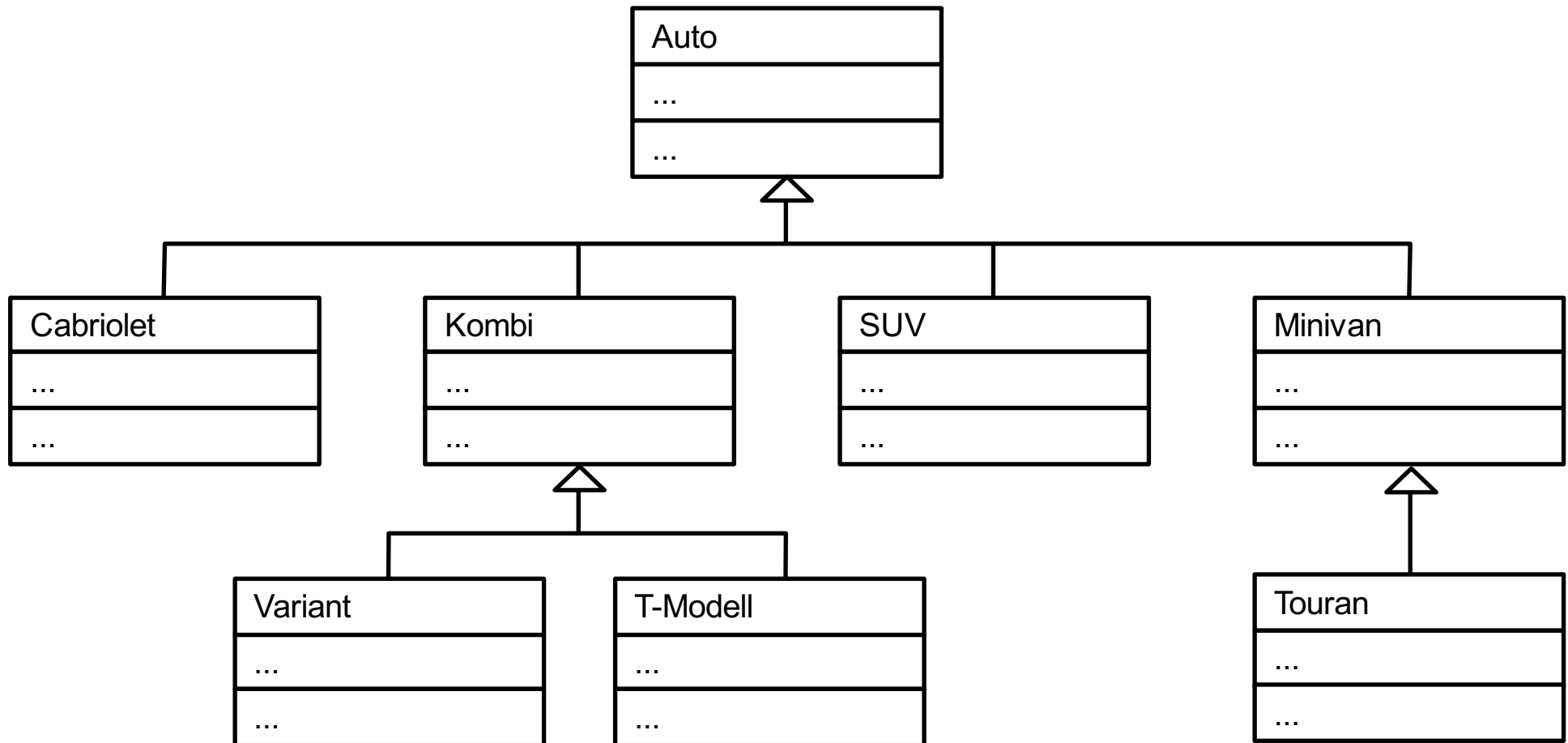
Mehrere abgeleiteten Klassen

Unser vorheriges Beispiel hat schon gezeigt, dass eine Basisklasse (hier das Auto) mehrere abgeleitete Klassen haben kann. In der UML Beschreibung werden dann die verschiedenen Pfeile zusammengefasst. Unser erweitertes Beispiel sieht dann folgendermaßen aus:



Mehrstufige Vererbung

Wie auch in der realen Welt, kann Vererbung auch über mehrere Generationen erfolgen. Es gibt Eltern, Kinder, Kindeskindern und so weiter. Bei allen Vererbungen handelt es sich um eine Ist-Ein Beziehung:



Anwendung der Generalisierung und Spezialisierung

Generalisierung und Spezialisierung erleichtern den allgemeinen Umgang mit einem behandelten Objekt.

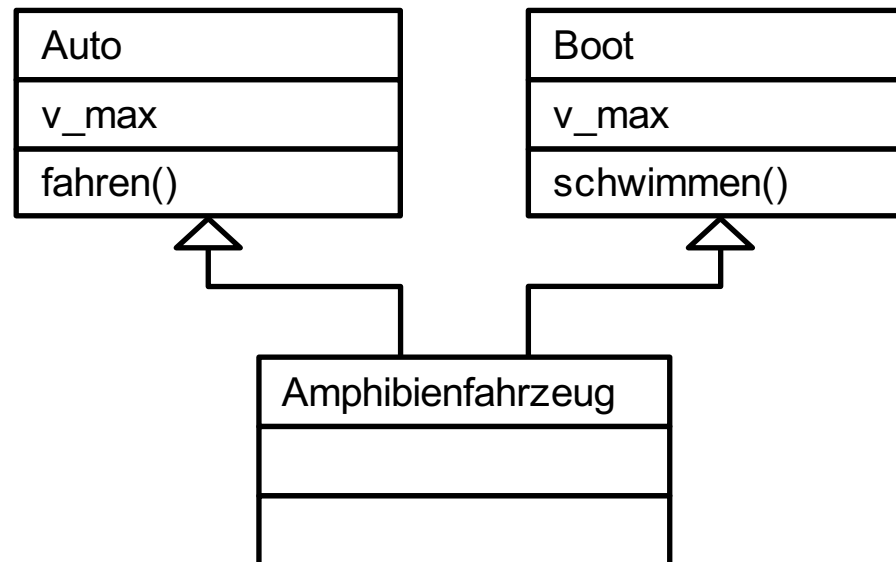
Beispielsweise wissen wir, dass eine Autowaschanlage mit ganz unterschiedlichen Autos umgehen kann. Die Autowaschanlage hat eine „Schnittstelle“ für Autos. Jedes Objekt das ein Auto ist, kann an die Waschanlage „übergeben“ werden.

Wir können alle Objekte vom Typ Auto an die Waschanlage übergeben. Wir benötigen also keine spezielle Autowaschanlagen für Kombis, denn ein Kombi ist ein Auto. Auch ein SUV kann in eine Autowaschanlage einfahren, ebenso wie ein Minivan.

Wir werden uns dieses Prinzip, dass abgeleitete Objekte eine Ist-Ein Beziehung zu ihrer Elternklasse haben, später noch zunutze machen.

Mehrfachvererbung

Eine abgeleitete Klasse kann auch mehrere Basisklassen haben. Wir sprechen dann von Mehrfachvererbung. Die Notation dafür sieht folgendermaßen aus:



Wir haben hier weiter eine Ist-Ein Beziehung. Ein Amphibienfahrzeug ist ein Auto und es ist auch gleichzeitig ein Boot. Es erbt die Methoden und Attribute seiner beiden Elternklassen und kann fahren und schwimmen.

Die Mehrfachvererbung kann in Detail durchaus verzwickelt werden und wird auch nicht von allen objektorientierten Sprachen unterstützt. So hat unser Amphibienfahrzeug beispielsweise zwei Attribute für die Maximalgeschwindigkeit `v_max` von seinen Elternklassen geerbt. Solche Konfliktsituationen müssen wir auflösen können.

Vererbung in C++

Die gezeigten Prinzipien der Abstraktion und Spezialisierung wollen wir konkret für unsere bisherigen Klassen anwenden.

Wir haben bereits unsere Textklasse `text` implementiert. Bisher können wir in unserer Klasse einen Text ablegen, den Text ausgeben und in dem Text nach einem Subtext suchen. Wir wollen diese Klasse nun als Basis für eine Vererbung nutzen. Um uns die folgenden Schritte zu erleichtern, ergänzen wir die Klasse `text` zuerst noch um einen parameterlosen Konstruktor.

```
class text
{
private:
    int len;
    char* txt;
public:
    text( char* t );
    text(); ←
    ~text();
    text( const text& s );
    text& operator=( const text& s );
    void print(){ printf( "%s", txt ); }
    int find( char* f );
};
```

```
text::text()
{
    len = strlen( "leer" );
    txt = ( char* ) malloc( len + 1 );
    strcpy( txt, "leer" );
}
```

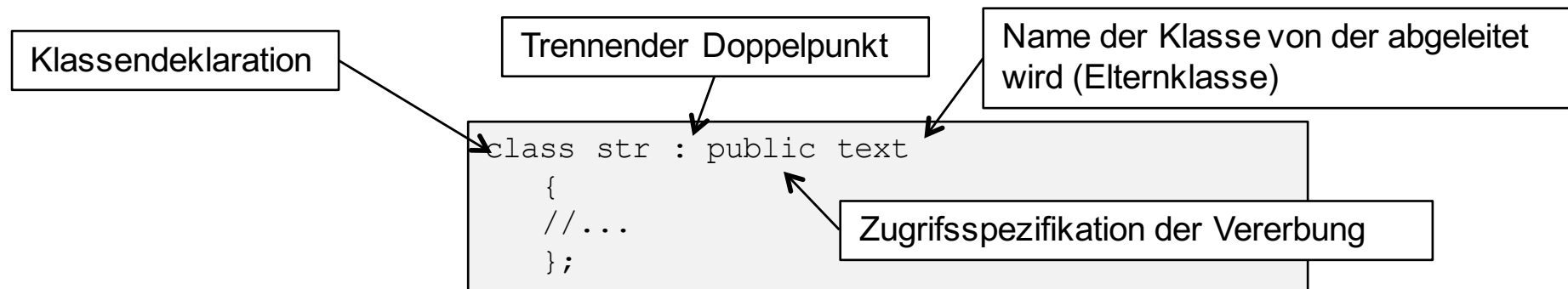
Deklaration und Implementation
des parameterlosen Konstruktors

Der parameterlose Konstruktor initialisiert die Klasse mit dem festen Text „leer“. Dies ist für Anwendung der Klasse `text` selbst wenig sinnvoll, vereinfacht aber unsere nächsten Schritte.

Ableiten der Klasse `str` von der Klasse `text`

Wir können in C++ jederzeit eine bestehende Klasse nehmen und eine weitere Klasse von ihr ableiten. Dies ist möglich, ohne in die Implementierung der bestehenden Klasse einzugreifen.

Mit der Ableitung stellen wir eine Ist-Ein-Beziehung her. Um die Klasse abzuleiten, gehen wir wie folgt vor:

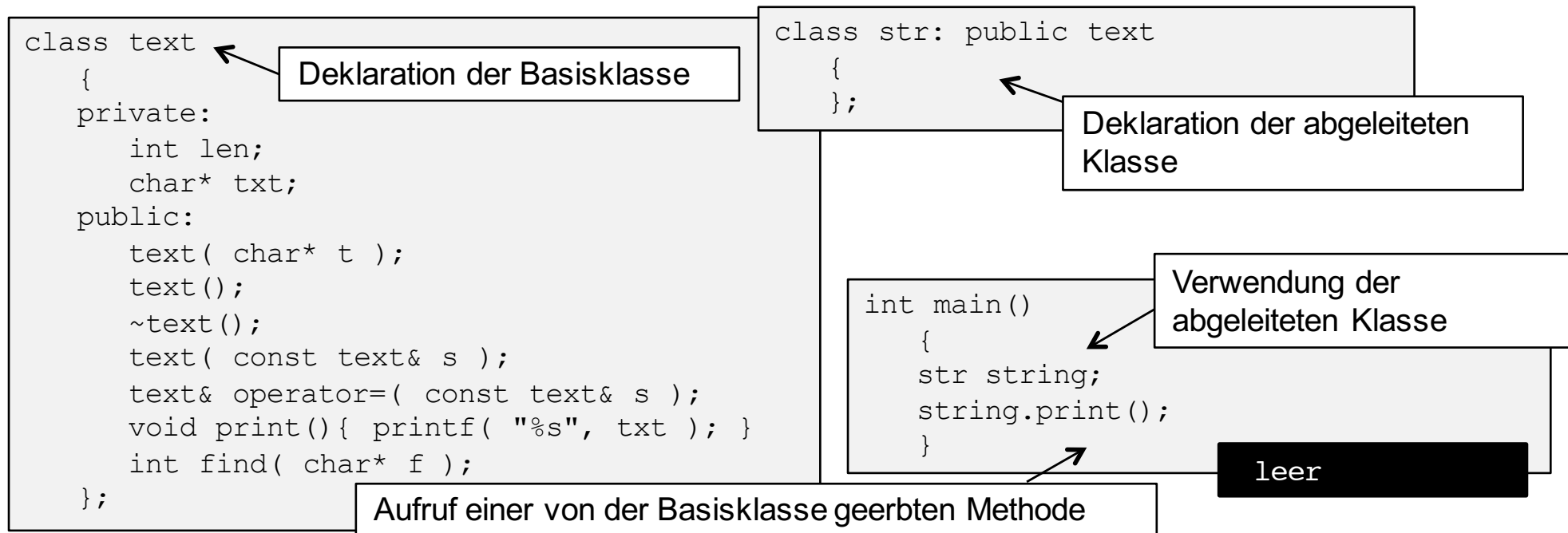


Wir fügen hinter dem Namen der abgeleiteten Klasse den Namen ihrer Basisklasse an. Dazwischen steht noch ein Doppelpunkt „:“ als Trennzeichen und eine Zugriffsspezifikation, in diesem Fall `public`. Mit dieser überwiegend verwendeten Zugriffsspezifikation werden wir uns aktuell ausschließlich beschäftigen. Es gibt aber weitere Zugriffsspezifikationen, die an dieser Stelle verwendet werden können.

Nach der obigen Deklaration ist die String-Klasse `str` als neue von `text` abgeleitete Klasse eingeführt und kann verwendet werden.

Instanziierung abgeleiteter Klassen

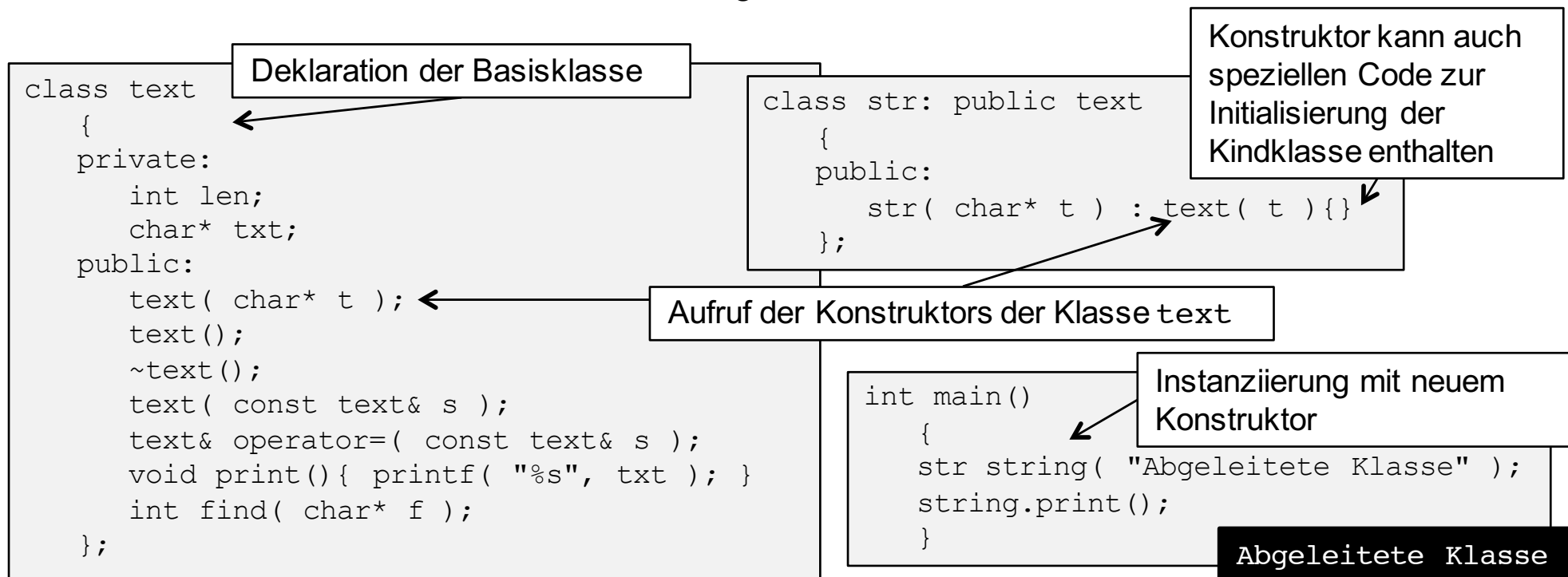
Da wir hier die Basisklasse parameterlos instanziierten können, reicht das bereits aus, um die abgeleitete Klasse zu instanziiieren. Wir haben für die abgeleitete Klasse keinen Konstruktor definiert, daher erstellt das System automatisch einen parameterlosen Konstruktor und wir können geerbte Methoden der Basisklasse über die erzeugte Instanz aufrufen.



Die Basisklasse `text` unserer Klasse `str` wird automatisch mit ihrem parameterlosen Konstruktor initialisiert. Über den Aufruf der `print` Methode greift die abgeleitete Klasse dann auf eine öffentliche Methode der Basisklasse zu und nutzt damit eine geerbte Funktionalität der Elternklasse.

Gezielte Aufrufe des Konstruktors der Basisklasse

Analog zur Instanziierung bei Aggregation ist bei Vererbung die abgeleitete Klasse Nutzer der Basisklasse und damit verpflichtet, diese zu initialisieren. Wenn wir eine abgeleitete Klasse instanziierten, müssen wir daher dafür sorgen, dass alle ihre Basisklassen korrekt instanziiert werden. Im vorherigen Beispiel wurde dies über einen parameterlosen Konstruktor erledigt. Erlaubt die Basisklasse keine parameterlose Erzeugung, muss ein passender Konstruktor der Basisklasse bei der Konstruktion der Kindklasse explizit aufgerufen werden. Der Ablauf ist dabei so, dass die Basisklassen immer vor ihren abgeleiteten Klassen instanziiert werden.

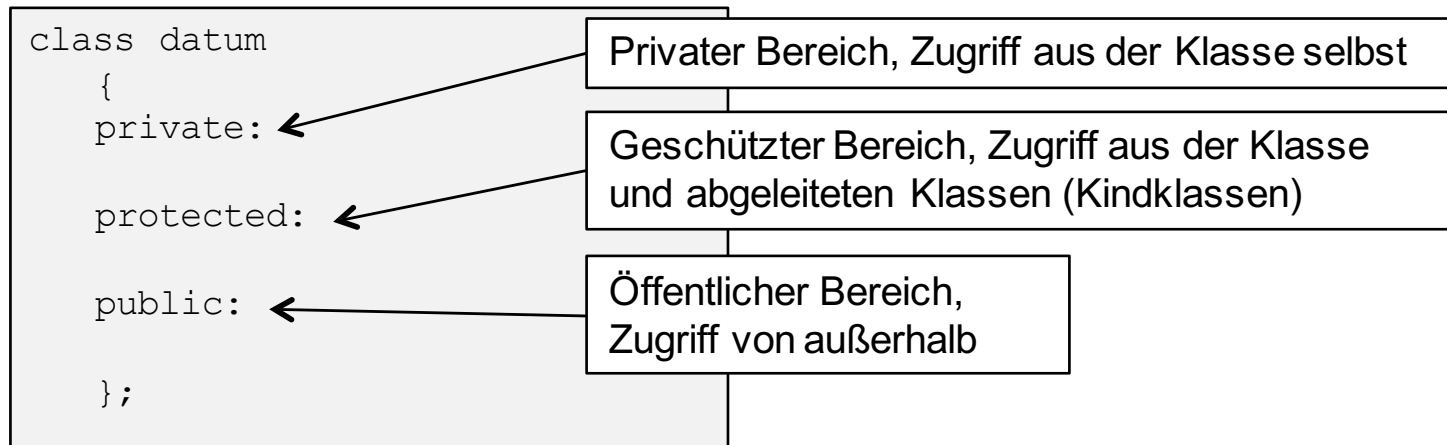


Das Vorgehen ist analog zur Konstruktion eingebetteter Objekte, allerdings wird der Konstruktor der Basisklasse unter Nennung des Namens der Basisklasse mit seiner entsprechenden Parametersignatur aufgerufen.

Der geschützte Zugriffsbereich einer Klasse

Wir haben nun bereits eine Klasse von unserer Basisklasse abgeleitet. Unsere abgeleitete Klasse hat allerdings keine besonderen Zugriffsrechte auf die Methoden und Attribute ihrer Basisklasse. Auch die Kindklasse kann nur den öffentlichen Teil ihrer Basisklasse nutzen. Diese Beschränkung ist notwendig, um den Zugriffsschutz zu erhalten, der ja nicht von außen umgangen werden soll.

Es ist allerdings sinnvoll, dass eine Klasse ihren Nachkommen gewisse Sonderrechte beim Zugriff einräumt. Hierzu bietet C++ als passendes Konzept die geschützten Member. Den geschützten Bereich „protected“ hatten wir bisher ausgeklammert. In dem Bereich protected werden die Methoden und Attribute abgelegt, auf die die abgeleiteten Klassen, also die Kinder einer Klasse, Zugriff haben dürfen.



Wir werden den geschützten Bereich direkt einsetzen, um die Möglichkeiten der Vererbung unserer Klasse text zu erweitern.

Verlagerung von Membern in den geschützten Bereich

In unserem Beispiel wollen wir die Funktionalität Klasse `str` erweitern, und ihr Zugriff auf die Attribut ihrer Basisklasse `text` ermöglichen. Wir legen dazu innerhalb der Klasse `text` einen geschützten Bereich an und verlagern die Member `len` und `txt` in diesen Bereich.

```
class text
{
protected:
    int len;
    char* txt;
public:
    text( char* t );
    text();
    ~text();
    text( const text& s );
    text& operator=( const text& s );
    void print(){ printf( "%s", txt ); }
    int find( char* f );
};
```

Geschützter Bereich, enthält jetzt die
vormals im privaten Bereich
enthaltenen Elemente der Klasse

Bereits bekannter Teil der Klasse
text im öffentlichen Bereich

Außenstehende, die die Klasse `text` verwenden, haben weiter nur über die Konstruktoren und öffentliche Methoden Zugriff auf den in der Klasse enthaltenen Text, während die zukünftigen Kinder der Klasse jetzt direkt auf die Attribute `len` und `txt` zugreifen können.

Ein privater Bereich ist nicht mehr erforderlich. Grundsätzlich könnte ein solcher Bereich aber zusätzlich vorkommen.

Erweiterung abgeleiteter Klassen

Zu einer abgeleiteten Klasse können wir Attribute und Methoden hinzufügen. Dabei können wir auf die aus der Basisklasse ererbten Attribute und Methoden zugreifen, sofern sie zugänglich sind, wie dies bei den geschützten Attributen von `text` nun der Fall ist.

Die Klasse `text` hat bisher nur einen festen Text, den sie bei Ihrer Erstellung zugewiesen bekommt. Wir wollen die Kindklasse so erweitern, dass der enthaltene Text jederzeit über eine `setText` Methode geändert werden kann.

```
class str: public text
{
public:
    str( char* t ) : text( t ) {}
    void setText( char* t );
};
```

Deklaration der neuen
Methode `setText`

Implementierung der neuen Methode

```
void str::setText( char * t )
{
    free( txt );
    len = strlen( t );
    txt = ( char * ) malloc( len + 1 );
    strcpy( txt, t );
}
```

Zugriff auf geschützte
Elemente der Basisklasse
innerhalb der Kindklasse

```
int main()
{
    str string( "Abgeleitete Klasse" );
    string.print();
    printf( "\n" );
    string.setText( "ist erweitert" );
    string.print( );
}
```

Die Implementierung erfolgt wie im
Zuweisungsoperator mit Freigabe
und Neuallokierung von Speicher

Verwendung der neuen Methode

Aufruf einer geerbten Methode zur
Ausgabe des geänderten Textes

**Abgeleitete Klasse
ist erweitert**

Überladen von Funktionen der Basisklasse

Bisher haben wir Funktionen der Basisklasse verwendet oder die abgeleitete Klasse gegenüber der Basisklasse mit erweiterter Funktionalität ausgestattet. Wir wollen jetzt eine Funktion überladen und modifizieren, die in der Basisklasse bereits vorhanden ist. Wir wählen dazu die `print` Methode. In unserer Kindklasse wollen wir hier eine von der Basisklasse abweichende Funktionalität umsetzen.

Für unsere Klasse `str`, soll die geänderte `print` Methode den ausgegebenen Text mit spitzen Doppelklammern einfassen. Die modifizierte Funktionalität verfolgt keinen besonderen Zweck, wir wollen lediglich ein beobachtbares abweichendes Verhalten erzeugen.

```
class str: public text
{
public:
    str( char* t ) : text( t ){}
    void setText( char* t );
    void print();
};
```

Deklaration der überladenen Methode `print`

```
void str::print()
{
    printf( "<<%s>>", txt );
}
```

Geänderte Ausgabe in der überladenen Funktion

```
int main()
{
    str s1( "Ueberladen" );
    s1.print();
}
```

Aufruf der Funktion. Es wird die überladene Funktion der Kindklasse gerufen

<<Ueberladen>>

Integrieren von Methoden der Basisklasse

Im vorherigen Beispiel haben wir die `print`-Methode komplett neu implementiert und die Methode der Basisklasse nicht mehr verwendet. Bei der einfachen Funktionalität ist das auch angemessen. Generell wollen wir aber vermeiden, Funktionalität der Basisklasse neu umzusetzen. Um dies zu ermöglichen, können wir auch Funktionalität der Basisklasse in eine überladene Methode integrieren und zwischen den modifizierenden Code „einpacken“. Wir rufen die Methode der Basisklasse, indem wir in der alternativen Implementierung ihren voll qualifizierten Namen angeben:

```
class str: public text
{
public:
    str( char* t ) : text( t ){}
    void setText( char* t );
    void print();
};
```

Aufruf der Methode der Basisklasse

```
void str::print()
{
    printf( "<<" );
    text::print();
    printf( ">>" );
}
```

Alternative Implementierung

Deklaration der überladenen Methode `print`

Ausgabe der umgebenden spitzen Klammern vor und nach dem Text

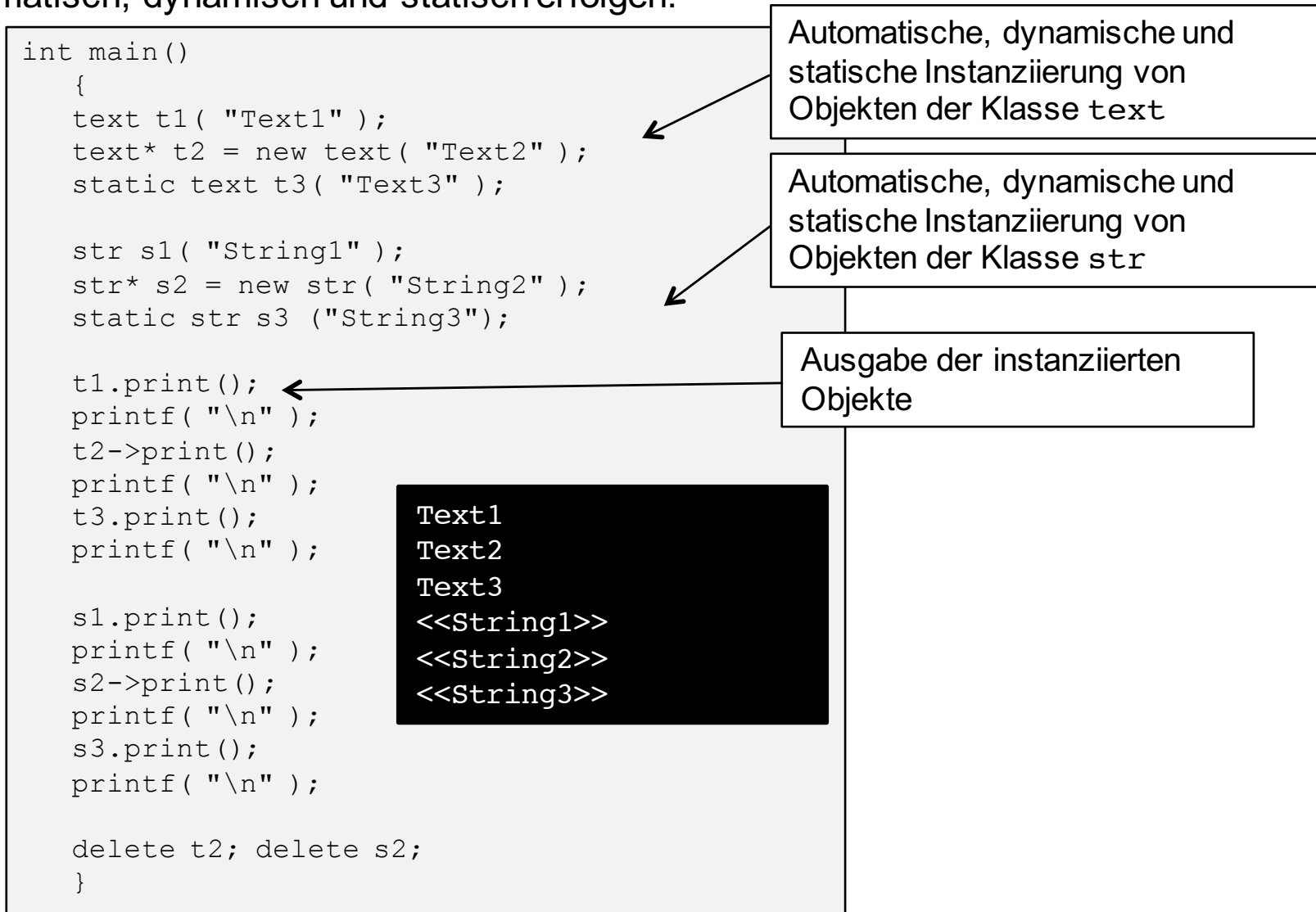
```
int main()
{
    str s1( "Ueberladen" );
    s1.print();
}
```

Aufruf der Funktion. Es wird die überladene Funktion der Kindklasse gerufen

`<<Ueberladen>>`

Unterschiedliche Instanziierungen und deren Verwendung

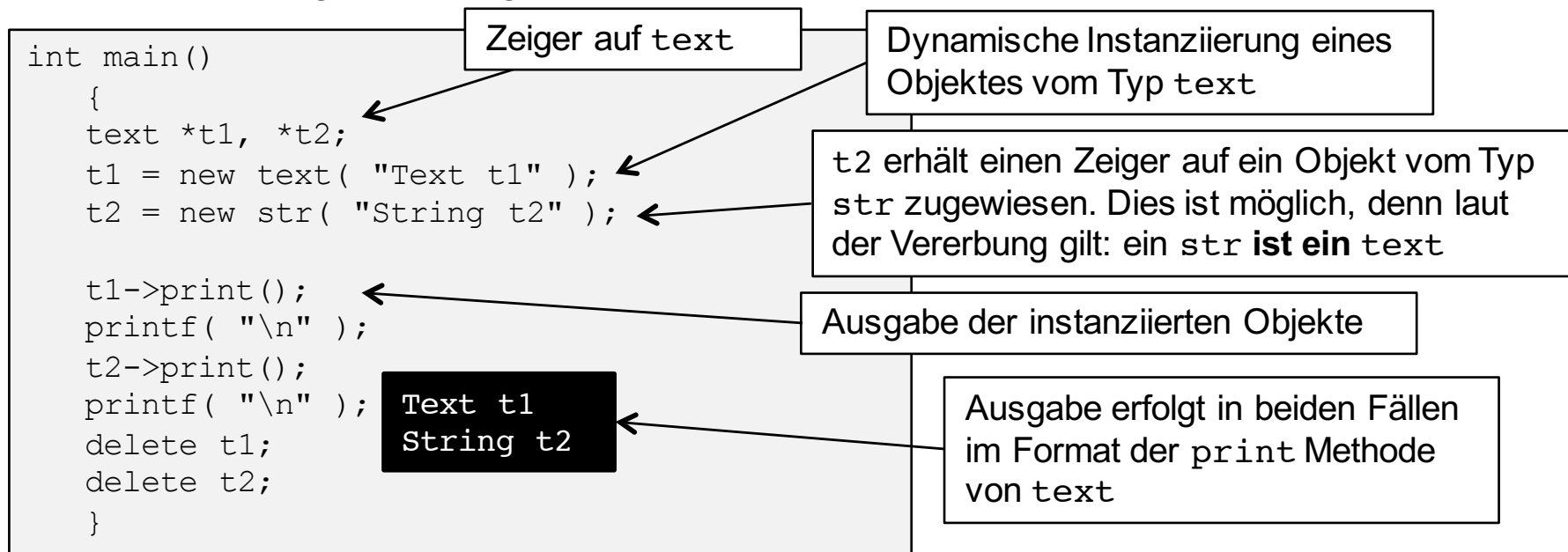
Wir können nun eine Klasse von einer Basisklasse ableiten, die Klassen instanziiieren und die Funktionalität der abgeleiteten Klasse erweitern, überladen und verwenden. Die Instanziierung kann automatisch, dynamisch und statisch erfolgen:



Aufruf von Methoden über Zeiger auf Objekte

Es ist ein Ziel, unsere Objekte auch auf einer abstrakteren Ebene verwenden zu können und uns deren Ist-Ein-Beziehung zunutze zu machen, beispielsweise indem wir ein Objekt der Klasse `str` als Objekt des Typs `text` verwenden.

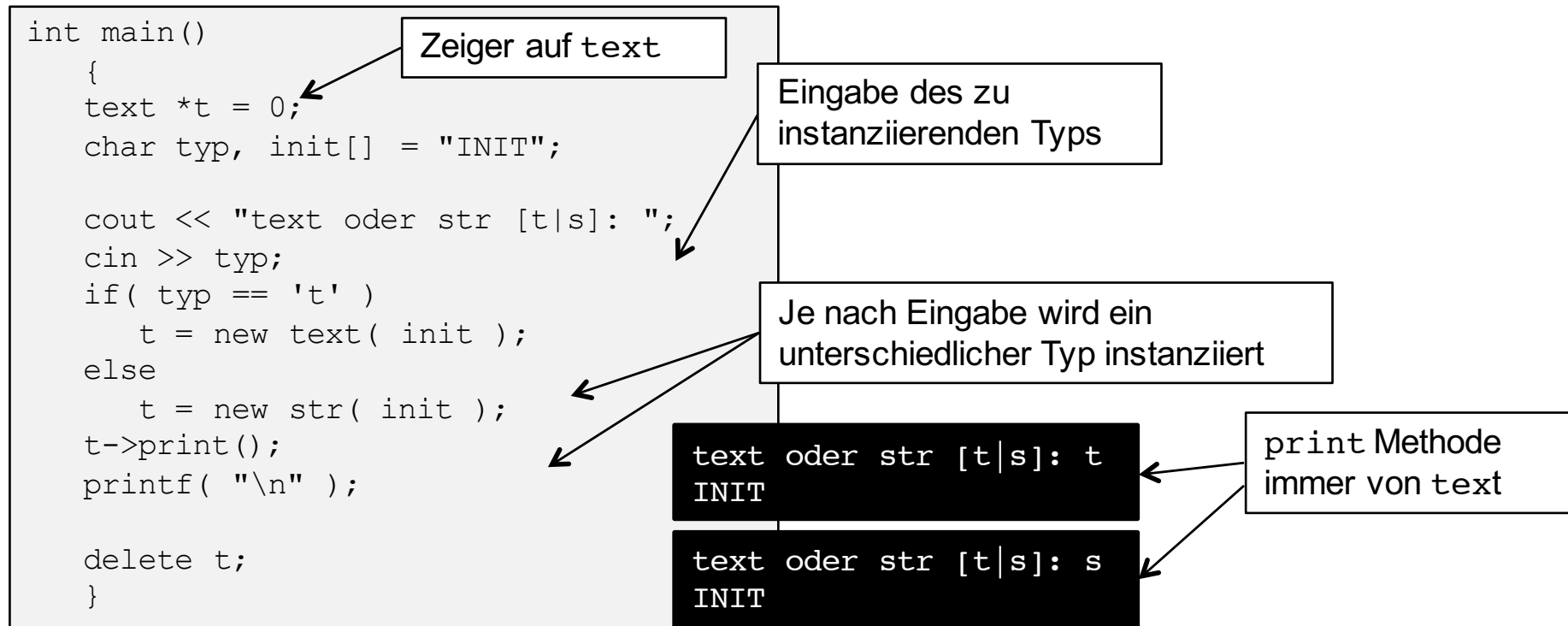
Wir können Objekte vom Typ `text` und `str` dynamisch erstellen und jeweils einem Zeiger auf die Basisklasse `text` zuweisen. Wenn wir mit diesen Zeigern die `print` Methode aufrufen, dann erhalten wir folgende Ausgabe:



Das war auch zu erwarten. Wir instanziierten für die Variable `t2` zwar ein Objekt der Klasse `str`, wir verwenden diese jedoch als `text`. Wenn wir die `print` Methode eines Objektes der Klasse `text` aufrufen, dann erfolgt die Ausgabe auch mit dessen Methode und ohne die spitzen Klammern der überladenen Ausgabe von `str`.

Dynamische Instanziierung

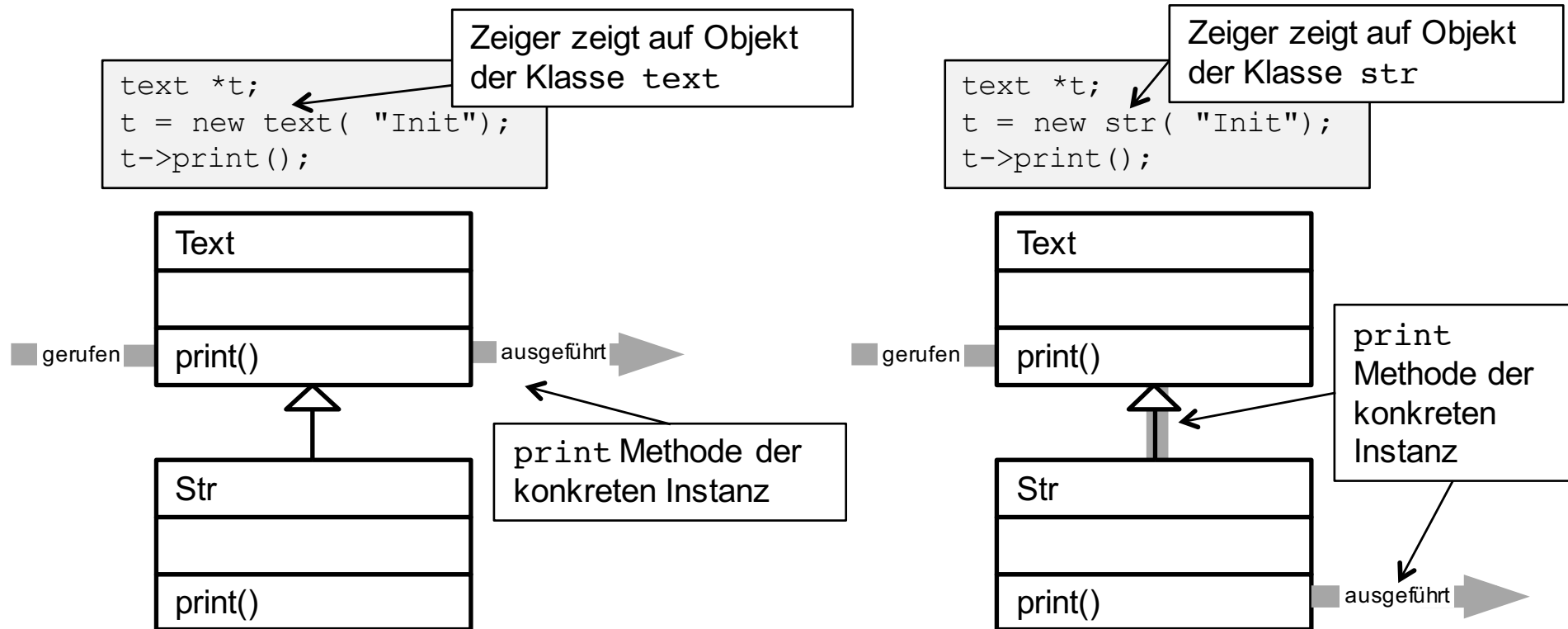
Wenn Objekte dynamisch instanziiert werden, ist zum Übersetzungszeitpunkt der genaue Typ des Objektes, auf den der Zeiger während der Ausführung zeigen wird, unter Umständen noch gar nicht bekannt.



Im obigen Beispiel wählen wir die Klasse des zu erstellenden Objekts erst nach einer Benutzereingabe aus. Wir müssen dem System daher einen Hinweis geben, dass beim Aufruf der `print` Methode zur Laufzeit geprüft wird, welcher konkreten Klasse das Element angehört. Abhängig vom Ergebnis (`text` oder `str`) muss dann die `print` Methode dieser identifizierten Klasse gerufen werden.

Prüfung des Aufrufes zur Laufzeit

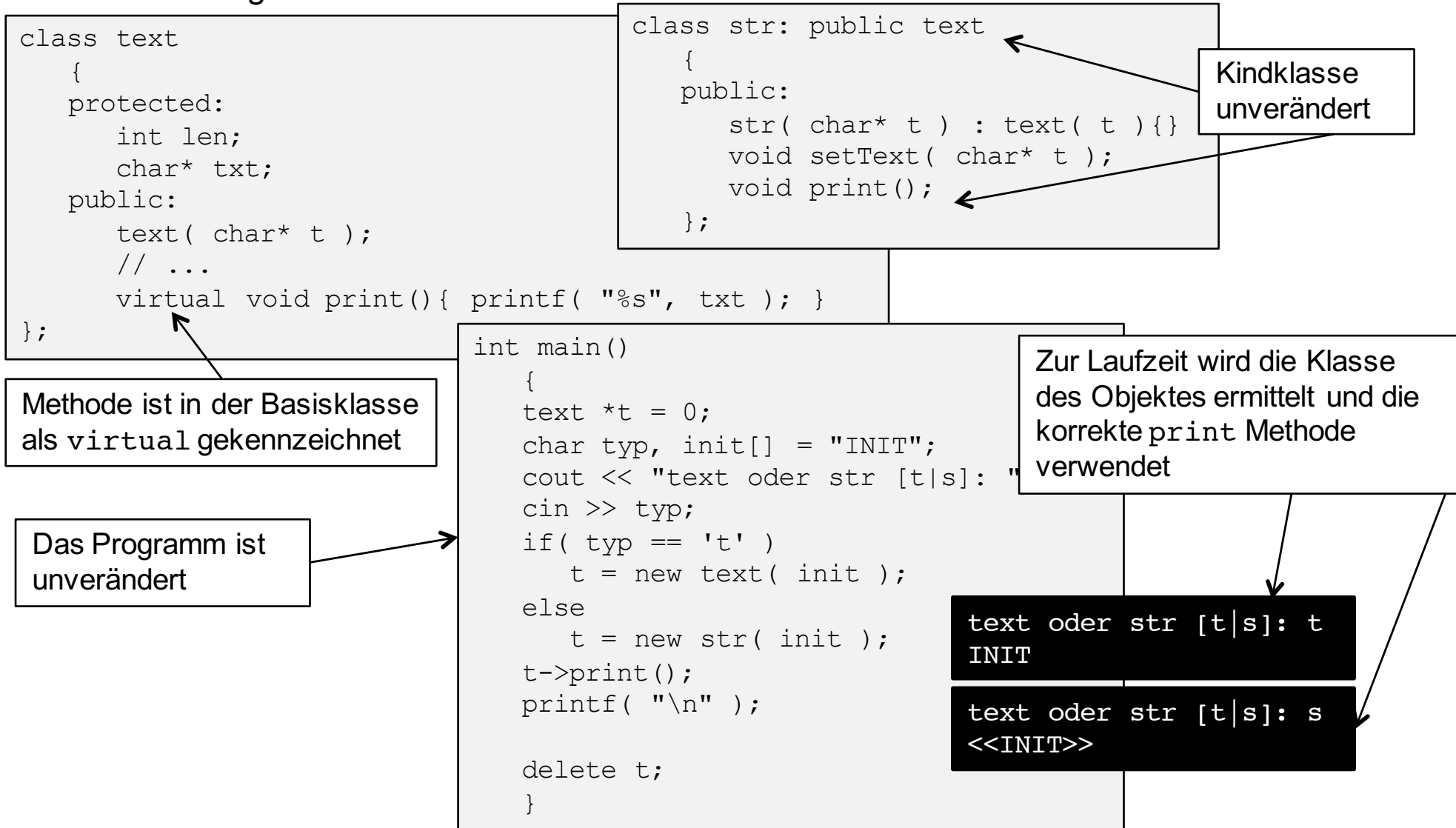
Mit einer Prüfung zur Laufzeit kann das System anhand des identifizierten Datentyps für die konkrete Instanz bei der Ausführung die richtige Methode bestimmen und aufrufen.



Die zum Umleiten des Methodenaufufes erforderlichen Mechanismen werden dadurch aktiviert, dass die entsprechenden Methoden in der Basisklasse als „virtuell“ deklariert werden. Dies ist für das Laufzeitsystem der Hinweis, bei einem Aufruf dieser Methode immer zunächst die tatsächliche Klasse festzustellen und dann die Version dieser Methode zu rufen, die zu der Klasse gehört.

Virtuelle Member Funktionen

Um eine Methode als virtuell zu deklarieren, wird der Methoden-Deklaration das Schlüsselwort **virtual** vorangestellt:



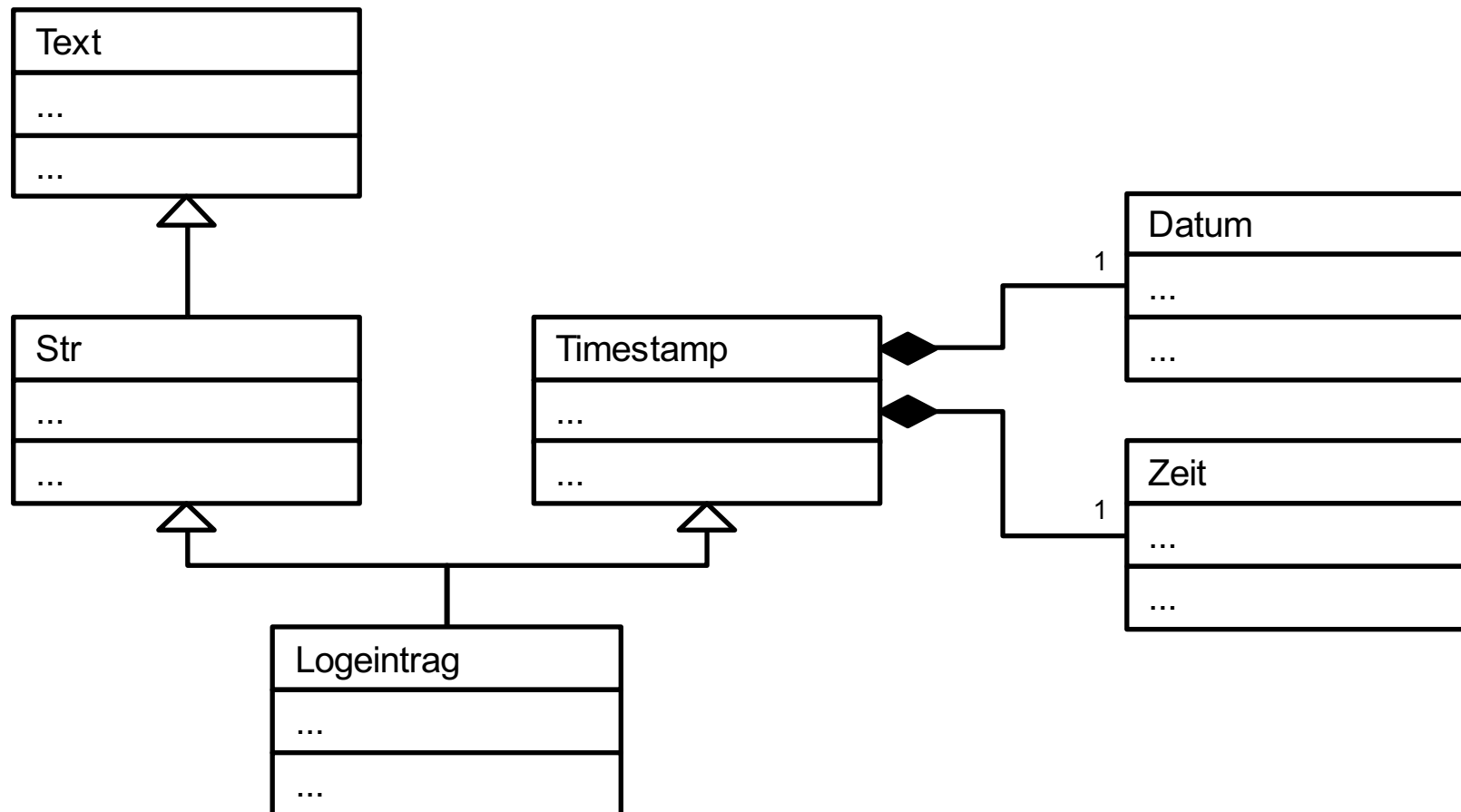
Verwendung des Schlüsselwortes `virtual`

Das Schlüsselwort `virtual` aktiviert für die gekennzeichnete Methode die dynamische Prüfung des Typs zur Laufzeit, auch „dynamisches Binden“ genannt. Diese Prüfung erfordert natürlich einen erhöhten Laufzeit- und Speicheraufwand. Die Kennzeichnung sollte daher nur dort erfolgen, wo sie wirklich benötigt wird.

Mehrfachvererbung

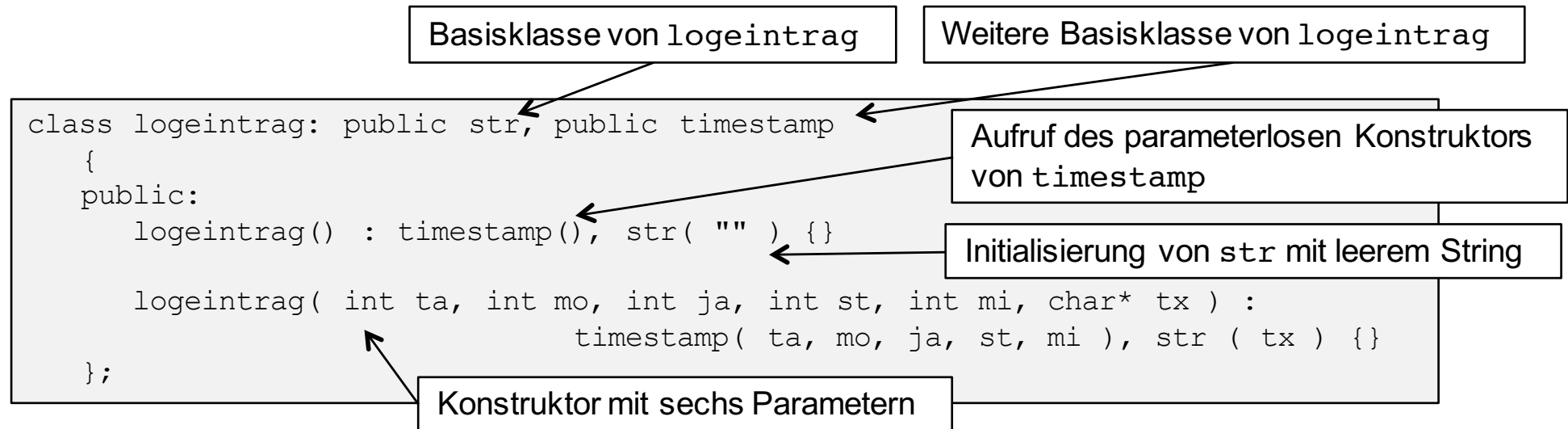
Wir haben die einfache Vererbung am Beispiel der Klasse `text` und `str` implementiert. Wir wollen jetzt noch einen Schritt weiter gehen und eine Klasse erstellen, die von zwei unterschiedlichen Klassen erbt.

Dazu erstellen wir die Klasse `logeintrag`, die einen Eintrag in einem Systemprotokoll oder Logbuch verwaltet. In der UML soll unsere Klassenhierarchie folgendermaßen aussehen:



Implementierung der Mehrfachvererbung

Unsere neue Klasse `logeintrag` erbt von den beiden bestehenden Klassen `str` und `timestamp`:



Auch bei der Mehrfachvererbung werden die Namen der Basisklassen durch den Doppelpunkt getrennt mit ihrer Zugriffsspezifikation aufgelistet.

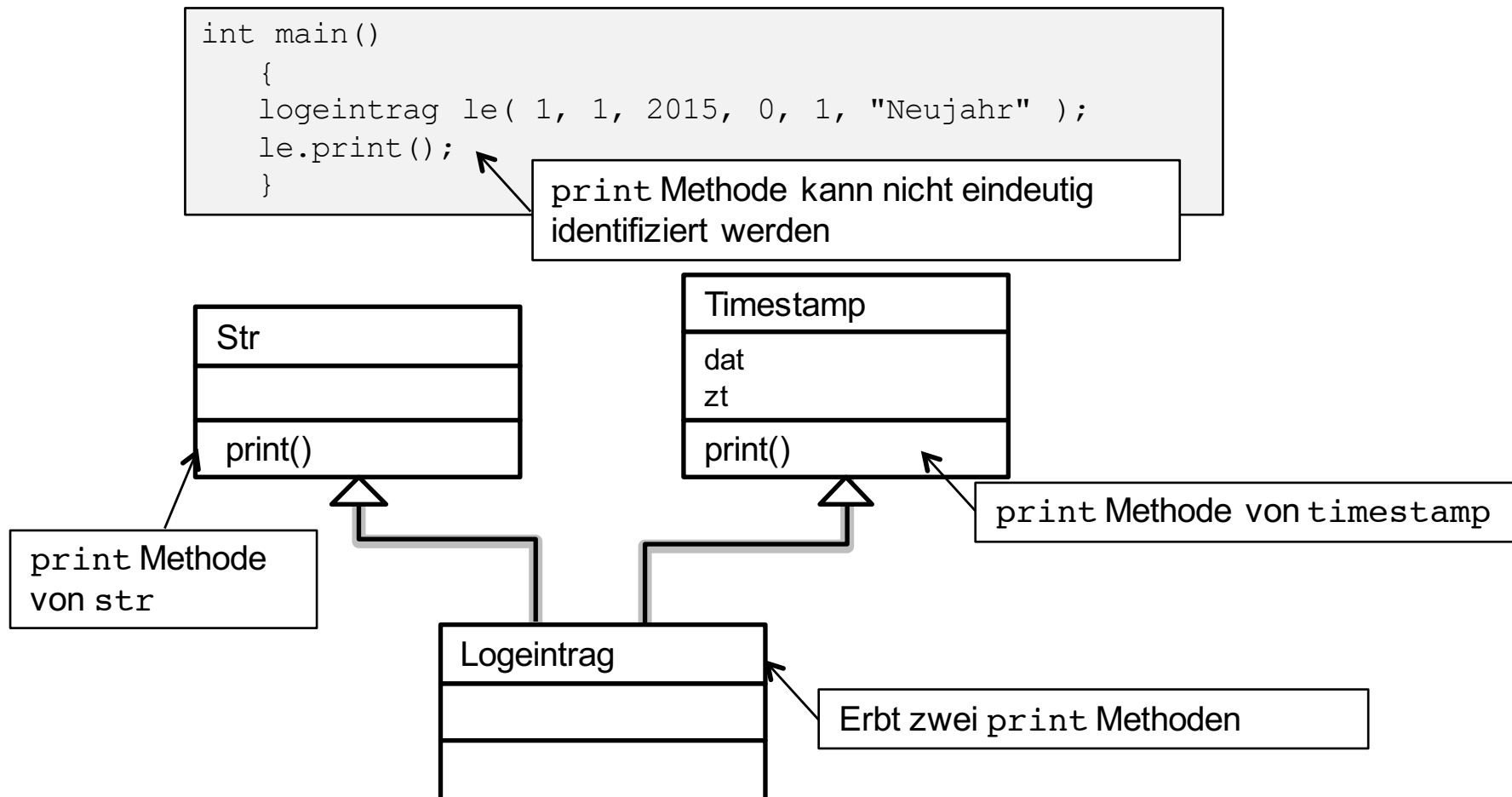
Für die Bereitstellung entsprechender Konstruktoren gilt das gleiche wie für die einfache Vererbung. Zur Initialisierung der Basisklassen haben wir der Klasse einen parameterlosen Konstruktor erstellt, der `timestamp` mit dem parameterlosen Konstruktor und `str` mit einem leeren String initialisiert.

Zusätzlich haben wir einen weiteren Konstruktor mit insgesamt sechs Parametern bereitgestellt, der beide Basisklassen unserer Klasse und damit auch deren eingebettete Klassen mit den Initialisierungswerten versorgt. Dies ist im folgenden noch einmal grafisch dargestellt.

Implementierung der Methode `print`

Unsere Klasse `logeintrag` hat nun die zwei Basisklassen und erbt alle Methoden und Attribute jeder ihrer Basisklassen. Damit erbt die Klasse `logeintrag` hier zwei Methoden mit dem Namen `print`.

Diese Mehrdeutigkeit ist vom System nicht aufzulösen. Wenn wir für die Klasse `logeintrag` eine `print` Methode verwenden wollen, müssen wir zuerst wieder Eindeutigkeit herstellen.



Deklaration der überladenen `print` Methode

Um Eindeutigkeit herzustellen, überladen wir die `print` Methode der Klasse `logeintrag`:

```
class logeintrag: public str, public timestamp
{
public:
    logeintrag() : timestamp(), str( "" ) {}
    logeintrag( int ta, int mo, int ja, int st, int mi, char* tx ) :
        timestamp( ta, mo, ja, st, mi ), str ( tx ) {}

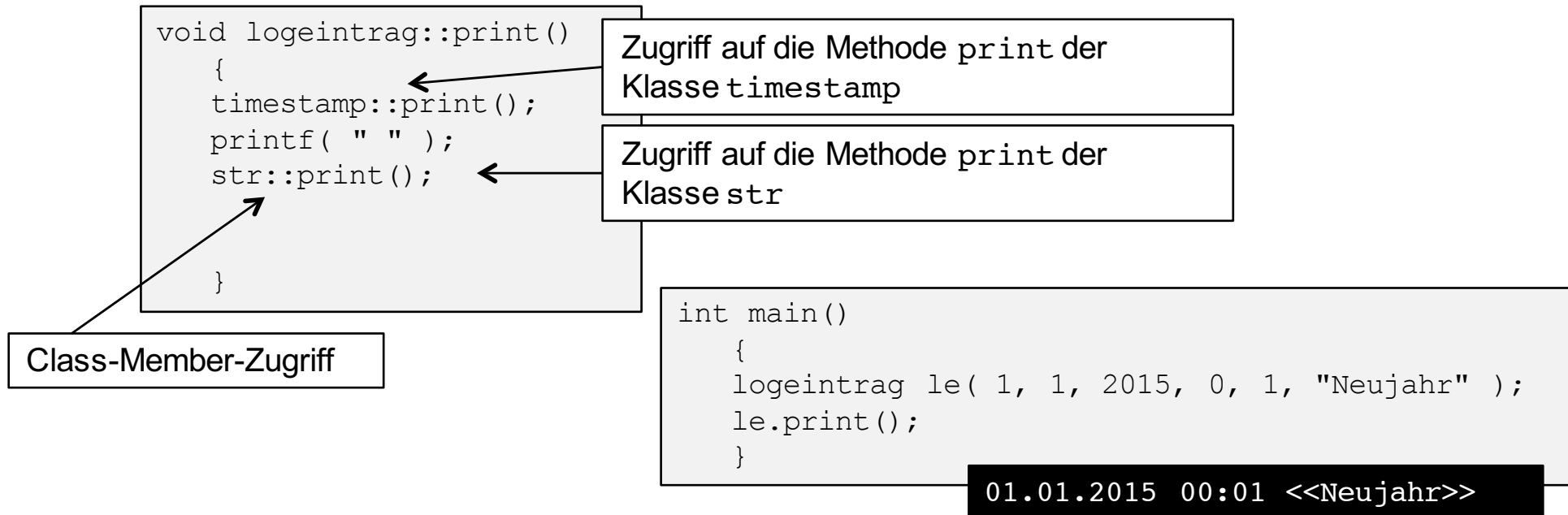
    void print();
};
```

Deklaration der überladenen `print` Methode

Mit der Deklaration einer eigenen `print` Methode ist für den Compiler wieder unzweifelhaft geklärt, welche Methode aufgerufen werden soll. Jetzt fehlt nur noch eine passende Implementierung, die auch die bereits vorhandene Funktionalität der Basisklassen verwenden soll.

Implementierung der neuen `print` Methode

Innerhalb der zu implementierenden Methode wollen wir wieder mittels „Wrapping“ auf die bereits vorhandenen Methoden der Basisklassen zugreifen:



Über den Operator „`::`“ für den „Class-Member-Zugriff“ können wir die gewünschten `print` Methoden in den jeweiligen Basisklassen eindeutig adressieren. Dazu geben wir den Namen der Klasse an, in der wir zugreifen wollen, gefolgt von dem Operator `::` und dem Namen der aufzurufenden Methode mit ihren Parametern. Damit können wir spezifisch auf die gewünschten Funktionen der Basisklassen zugreifen.

Rein virtuelle Funktionen

Mit dem Schlüsselwort `virtual` haben wir eine Möglichkeit kennengelernt, mit der wir eine Methode in einer abgeleiteten Klasse dynamisch binden können und zur Laufzeit die passende Methode ermittelt wird.

Gelegentlich benötigt man aber auch Klassen, die virtuelle Methoden enthalten, bei denen es (noch) nicht sinnvoll ist, eine Implementierung vorzunehmen. In einem solchen Fall deklarieren wir die Methode in der Basisklasse als **rein virtuelle Funktion**, und erzwingen damit die Implementierung in den Kindklassen. Rein virtuelle Funktionen haben statt einer Implementierung den Zusatz „`=0`“ in der Klassendeklaration:

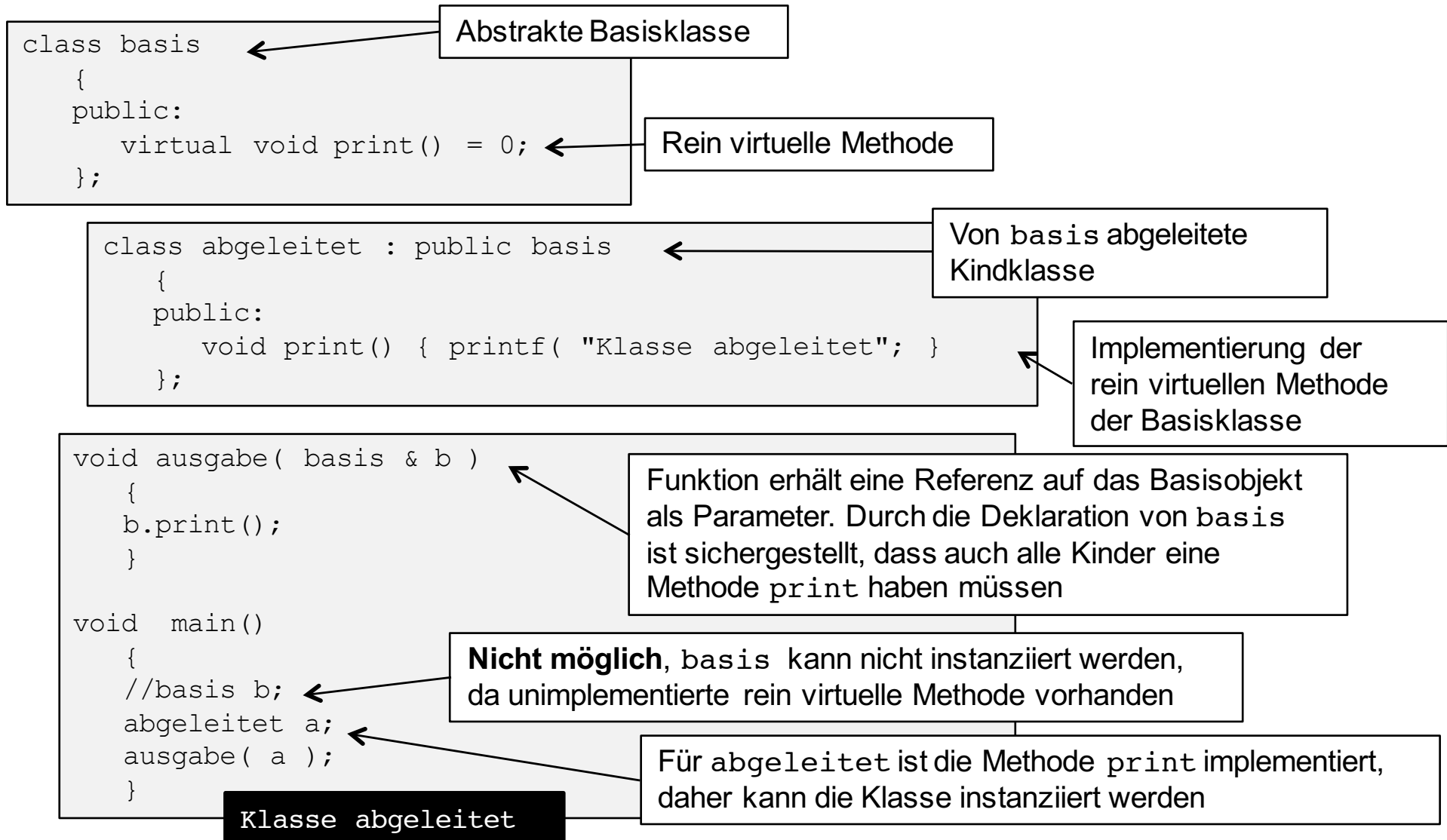
```
class basis
{
public:
    virtual void print() = 0; ← Rein virtuelle Methode
};
```

Eine Einsatzmöglichkeit liegt vor, wenn wir wissen, dass alle Kinder einer Klasse eine bestimmte Methode erhalten sollen, deren Implementierung aber erst mit der Konkretisierung der abgeleiteten Klasse erfolgen kann.

Die Deklaration einer rein virtuellen Methode hat zur Folge, dass die Klasse nicht mehr instanziiert werden kann, da die notwendige Implementierung noch „fehlt“. Wir sprechen in diesem Zusammenhang von einer abstrakten Klasse. Die Kinder oder Kindeskindern der Klasse müssen die Methode überschreiben, erst dann können sie instanziiert werden.

Instanziierung von abstrakten Klassen

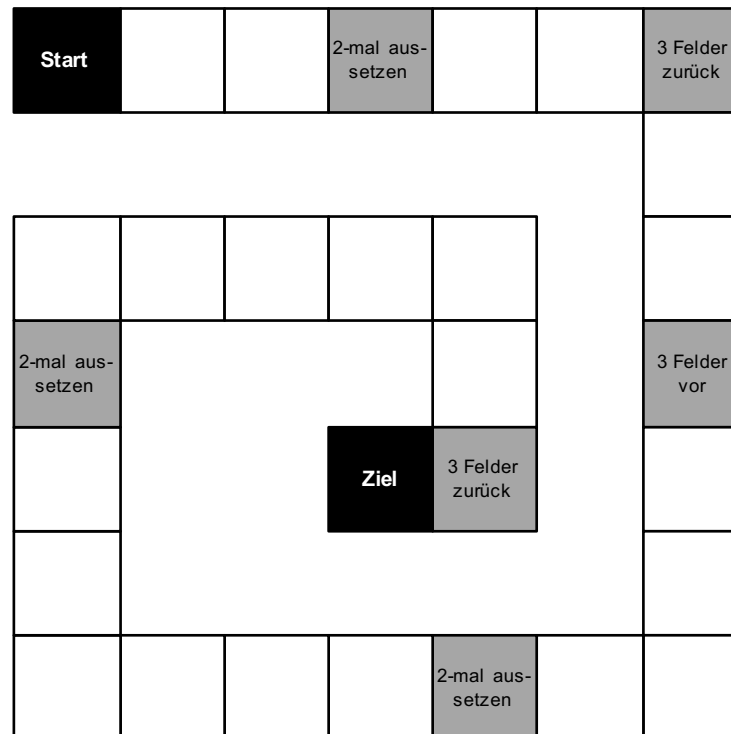
Die Kinder oder Kindeskindern einer abstrakten Klasse müssen alle rein virtuellen Methoden der Basisklasse implementieren, damit sie instanziiert werden können:



Würfelspiel

Wie wollen nun die Prinzipien der objektorientierten Programmierung und die Stärken der Vererbung in einem Beispiel einsetzen. Es handelt sich um ein Spiel, bei dem die Spieler von einem Startfeld aus ein bestimmtes Ziel erreichen müssen.

Auf dem Weg zum Zielfeld gibt es Hindernisse, die einen Spieler aufhalten oder zurückwerfen, aber auch Felder, die einen Spieler weiter voranbringen. Ein beispielhafter Spielplan könnte folgendermaßen aussehen:

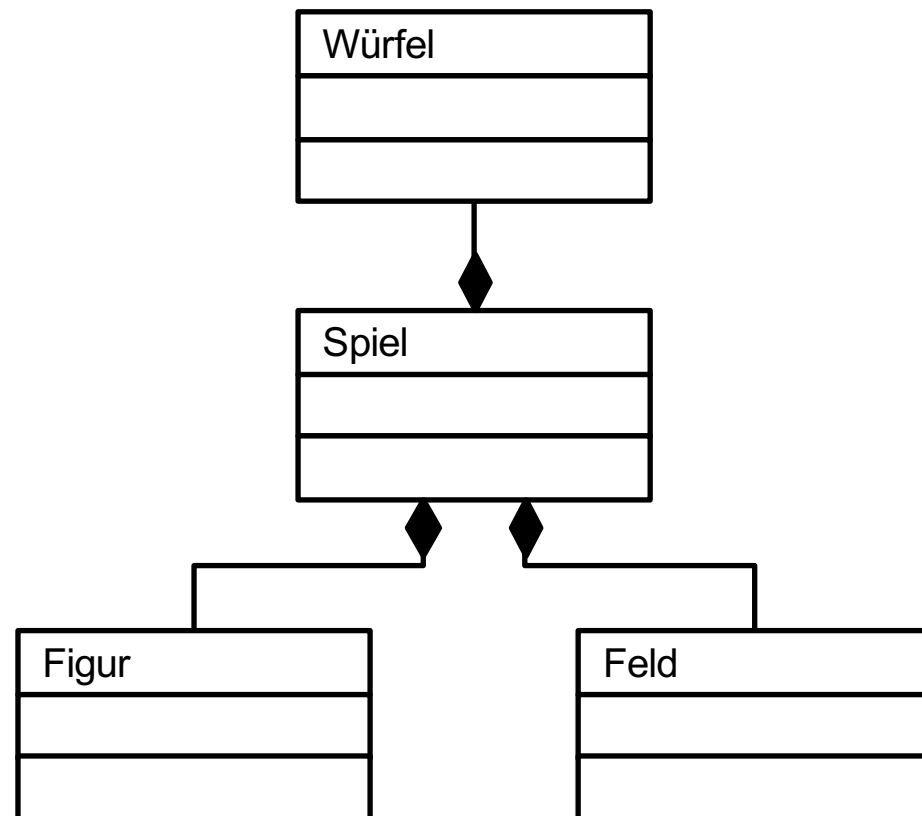


Design unseres Spiels

Wir wollen in unserem Spiel den gerade gezeigten Spielplan umsetzen. Das Programm soll aber so gestaltet sein, dass beliebige Spielpläne dieser Art erstellt werden können. Bei dem Blick auf das Spiel können wir die folgenden Elemente identifizieren:

- Den Spielplan mit einer Reihe von nacheinander angeordneten Spielfeldern
- Vier unterschiedliche Spielfiguren
- Einen Würfel

Damit haben wir bereits die wesentlichen Klassen für unser Design gefunden:



Verfeinerung unseres Designs

Wenn wir den Spielplan noch einmal genauer betrachten, dann erkennen wir schnell, dass unser Spielplan aus einzelnen Feldern besteht, bei denen unterschiedliche Typen von Feldern vorkommen. Neben den normalen Feldern gibt es ein Start- und Zielfeld sowie besondere „Ereignisfelder“.

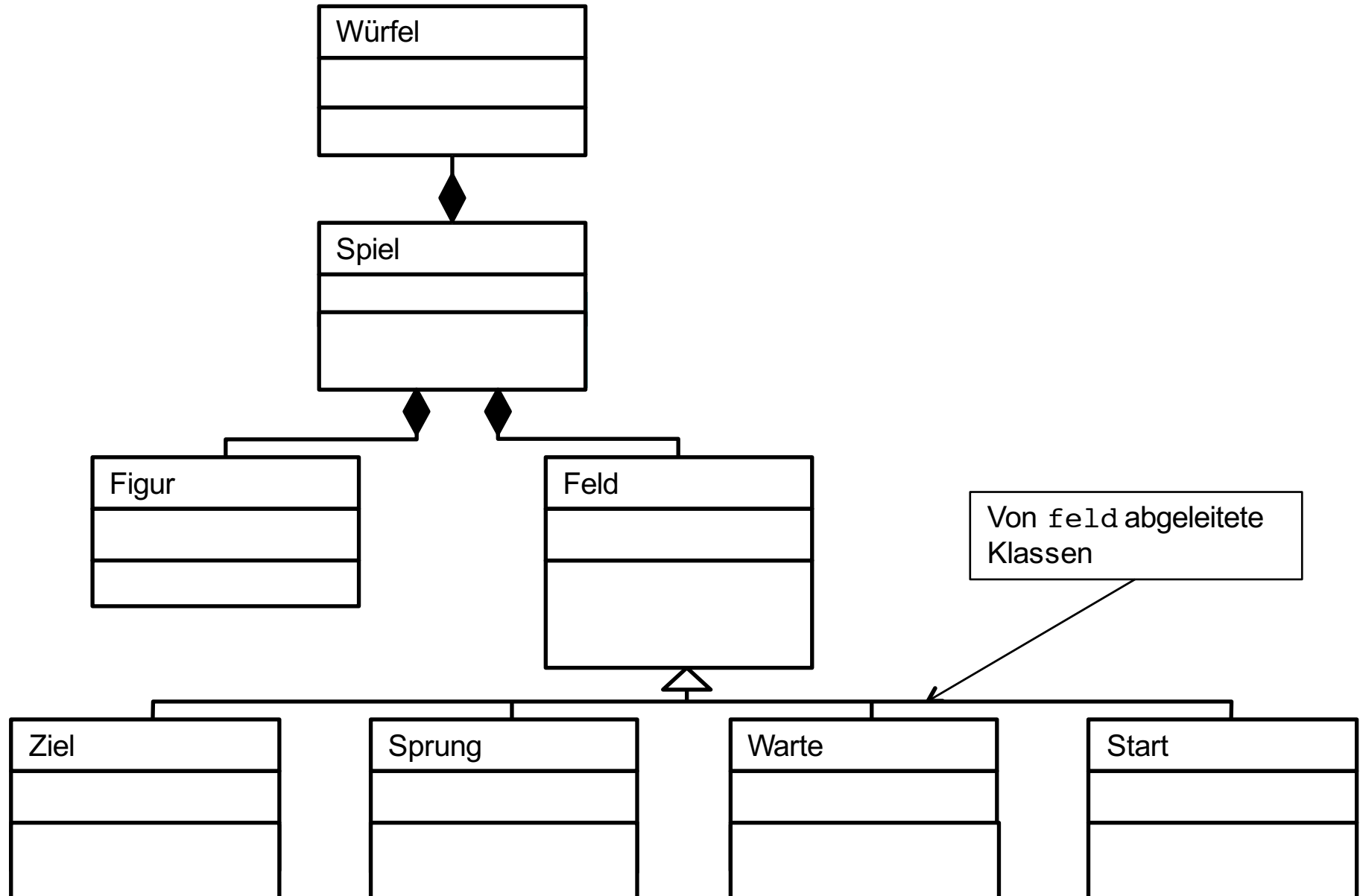
Insgesamt finden wir auf dem Spielplan die folgenden Felder wieder:

- Ein Startfeld
- Ein Zielfeld
- Sprungfelder
- Wartefelder

Jedes dieser Felder ist ein Feld, stellt aber eine Spezialisierung des gewöhnlichen Feldes dar. Wir werden diese Spezialfelder von dem gewöhnlichen Feld durch Vererbung ableiten:

Erweitertes Design

Das erweiterte Klassendesign zeigt sich dann wie folgt:



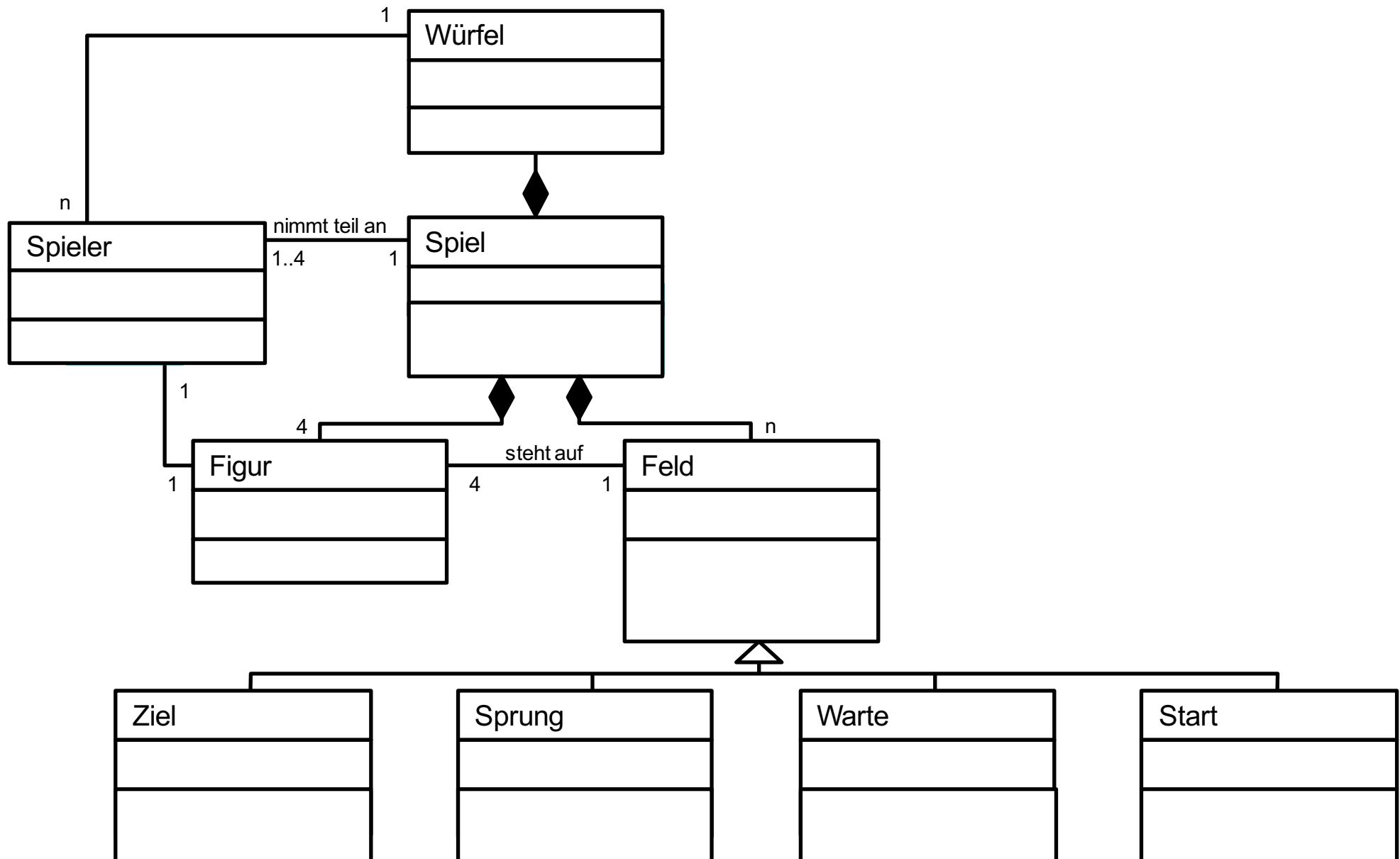
Die Spielanleitung

Der Spielanleitung entnehmen wir jetzt noch die folgenden „Anforderungen“:

- Das Spiel kann von ein bis vier Spielern gespielt werden
- Jeder Spieler bekommt eine Spielfigur und startet mit dieser vom Startfeld
- Es wird reihum gewürfelt und ein Spieler rückt immer entsprechend seiner Augenzahl vor
- Es können mehrere Figuren auf einem Feld stehen und Figuren können nicht geschlagen werden
- Wenn eine Figur auf ein Sprungfeld kommt, muss sie die dort angegebene Zahl an Schritten vor- oder zurückgehen
- Wenn eine Figur auf ein Wartefeld kommt, so muss der Spieler die dort angegebene Anzahl von Runden aussetzen
- Wenn man mit einem Wurf über das Ende des Spielplanes hinauskommt, so muss man die überzähligen Punkte wieder zurücksetzen
- Wer als erster das Zielfeld (exakt) erreicht hat, hat gewonnen

Aus der Spielanleitung können wir die Kardinalitäten ermitteln, die wir in unser Design eintragen, ebenso eine weitere wichtige Komponente, den Spieler:

Hinzufügen der Kardinalitäten und des Spielers



Das dynamische Verhalten des Spiels

Aus den vorhandenen Informationen leiten wir nun das dynamische Verhalten des Spiels ab:

Am Anfang melden sich die Spieler als Teilnehmer am Spiel an. Das Spiel hat dazu eine Methode `anmeldung`, die vom Spieler aufgerufen wird.

Bei der Anmeldung erhält der Spieler eine Figur, die mit `figurAufstellen` auf dem Startfeld platziert wird. Wenn sich genügend Spieler angemeldet haben, wird das Spiel gestartet, um eine Partie zu spielen.

Dies wird von der Methode `partie` kontrolliert. In der Partie übernimmt das Spiel die Kontrolle und fordert die Spieler nacheinander auf, einen Zug zu machen. Es sendet eine Botschaft an den betroffenen Spieler, der daraufhin seine Methode `zug` ausführt.

Bevor der Spieler aufgefordert wird, einen Zug zu machen, fragt das Spiel das Feld auf dem die Figur des Spielers steht, ob die Figur das Feld verlassen darf, oder ob sie blockiert ist.

Dazu dient die Methode `blockiert` des Feldes. Ist die Figur blockiert, wird der Spieler übersprungen. Ist ein Spieler aufgefordert zu ziehen, nimmt er den Würfel und würfelt mit dessen Methode `wurf`.

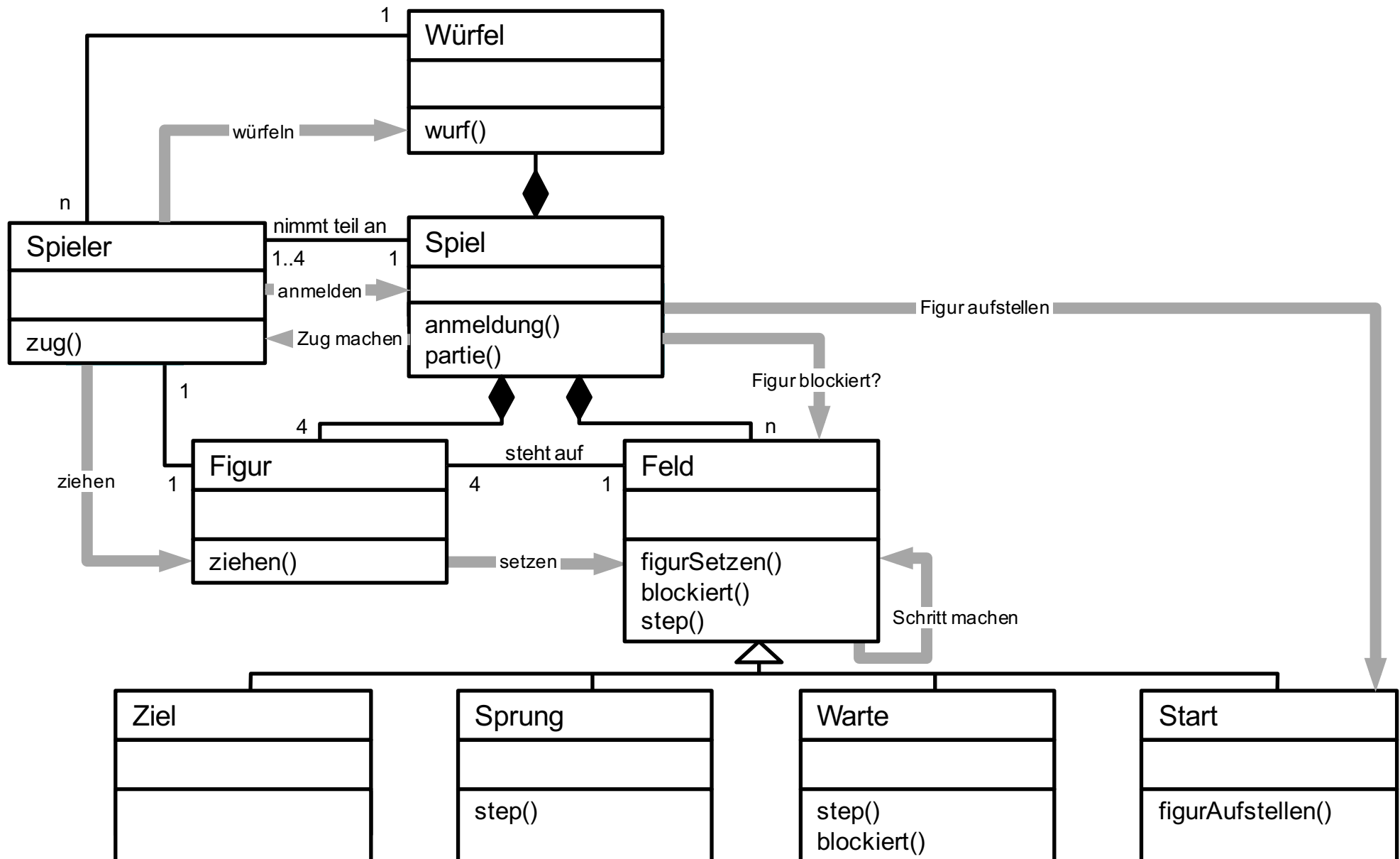
Mit der gewürfelten Zahl wendet er sich an seine Figur und fordert sie auf, die entsprechende Anzahl von Feldern weiterzuziehen. Die Figur wendet sich dazu an ihr Feld und dessen Methode `figurSetzen` mit dem Auftrag entsprechend weiterzuziehen.

Die Felder reichen die Figur dann mit der Methode `step` von Feld zu Feld weiter. Das Spiel setzt den Prozess fort, bis eine Figur das Zielfeld erreicht hat.

Im folgenden Diagramm werden die entsprechenden Botschaften noch einmal dargestellt.

Das dynamische Verhalten des Spiels im Diagramm

in der Klassenansicht stellt sich das Verhalten so dar:

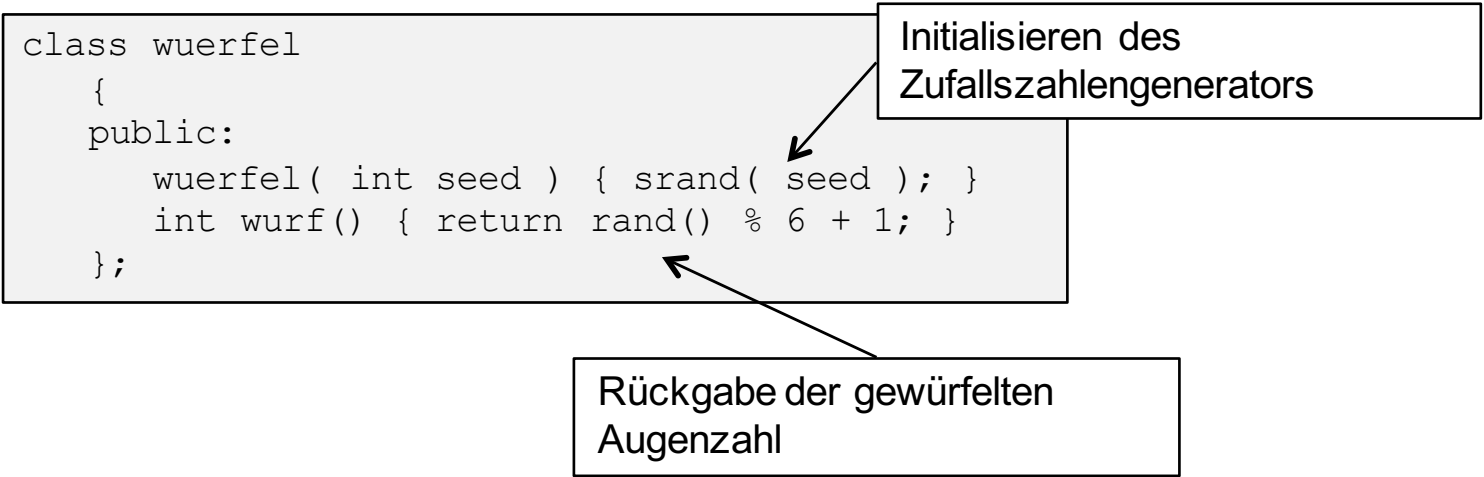


Implementierung

Die erste Klasse die wir realisieren ist der Würfel. Wir verwenden die Funktionen zur Generierung von Zufallszahlen aus der C-Runtime-Library.

```
class wuerfel
{
public:
    wuerfel( int seed ) { srand( seed ); }
    int wurf() { return rand() % 6 + 1; }
};
```

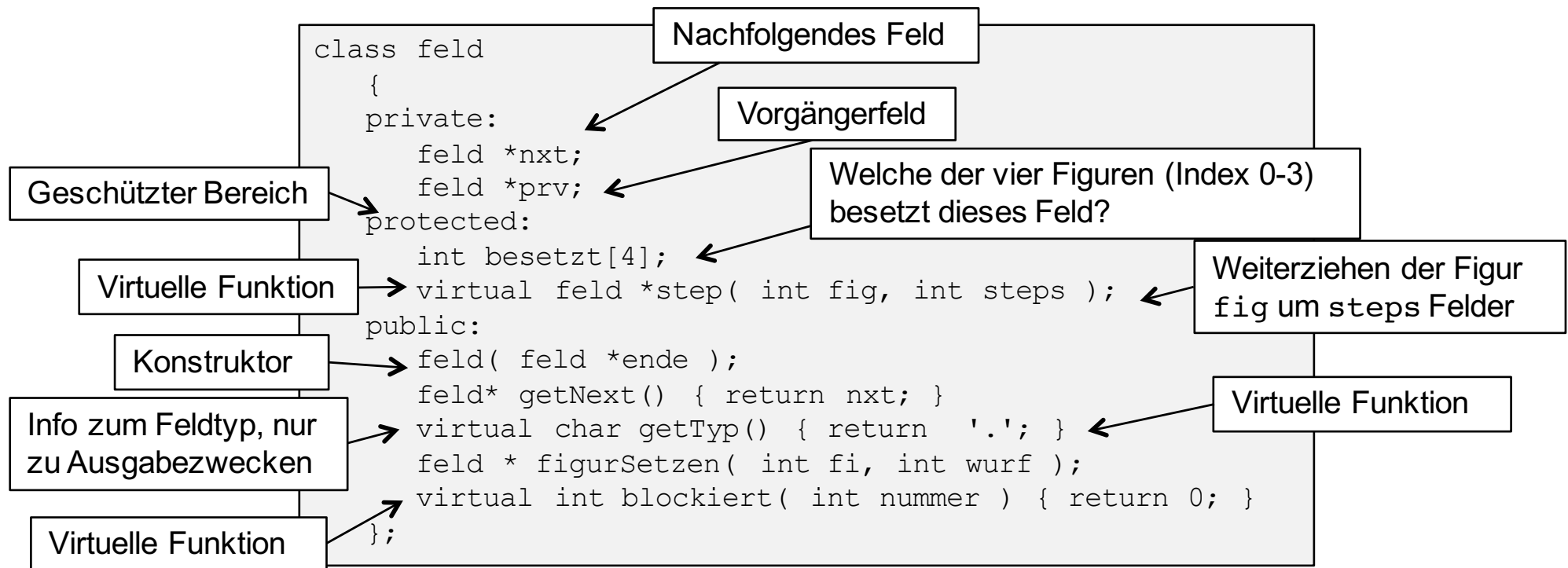
Initialisieren des
Zufallszahlengenerators



Rückgabe der gewürfelten
Augenzahl

Ein Spielfeld

Wir erstellen die Definition der Klasse `feld`. Die einzelnen Felder werden in Form einer verketteten Liste miteinander verknüpft werden.

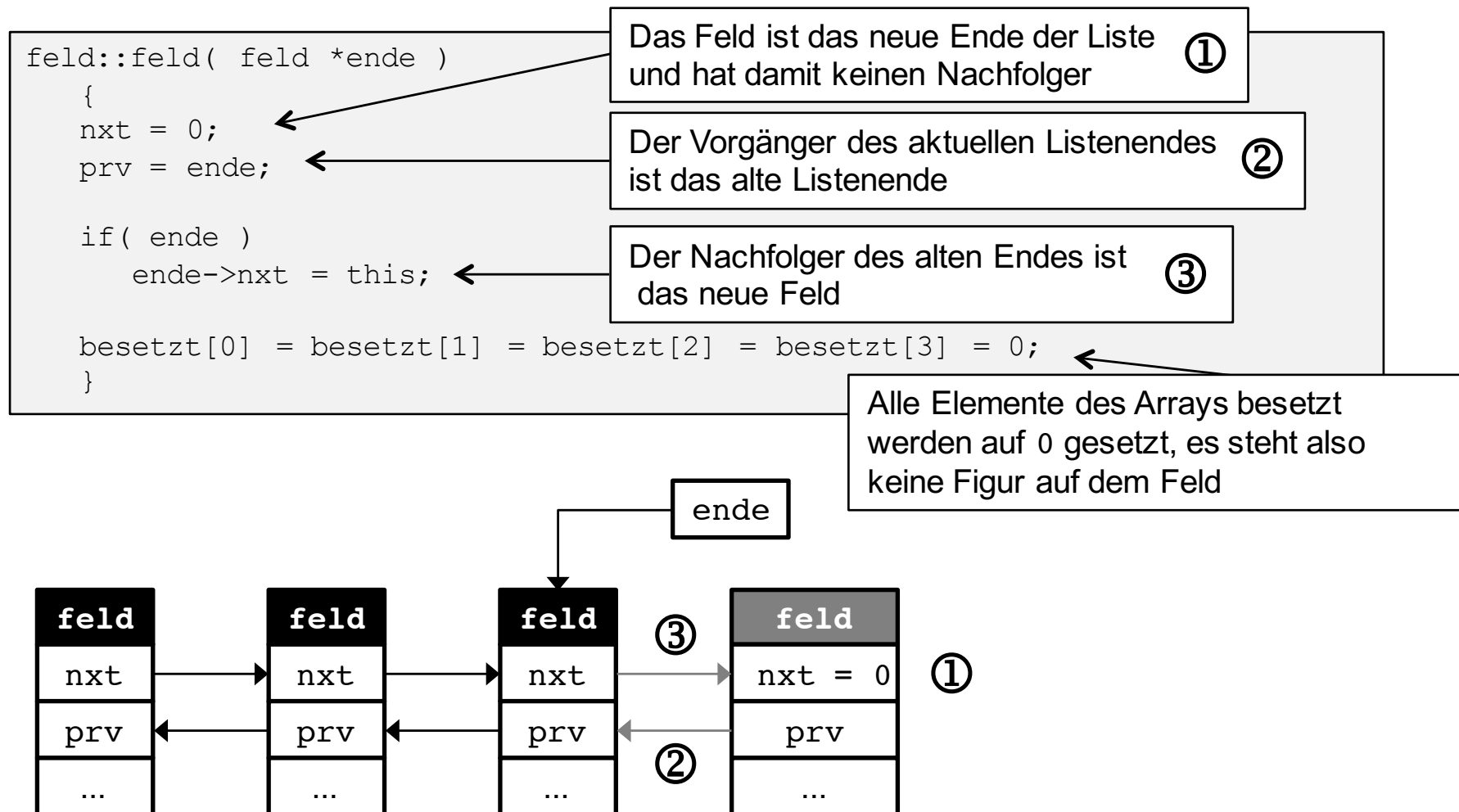


Alle drei Funktionen, für die in den spezialisierten, abgeleiteten Klassen ein besonderes Verhalten implementiert wird, sind als `virtual` deklariert, damit zur Laufzeit die richtige Funktion aufgerufen wird.

Die Funktionen, die später vollständig auf der Abstraktionsebene der Klasse `feld` abgehandelt werden, sind nicht als `virtuell` gekennzeichnet.

Konstruktor eines Spielfeldes

Der Konstruktor hat im wesentlichen die Aufgabe, die Verkettung der einzelnen Spielfelder herzustellen. Dazu wird ihm als Parameter der Wert des aktuellen Listenendes `ende` angegeben. Um das neue Feld an das Ende der Liste anzuhängen, müssen die folgenden Operationen durchgeführt werden:



Setzen einer Figur

Das Spielfeld enthält auch die Implementierung zum Setzen einer Figur:

```
feld *feld::figurSetzen( int fig, int wurf )  
{  
    if( !besetzt[fig] || blockiert( fig ) )  
        return 0;  
    besetzt[fig] = 0;  
    return step( fig, wurf );  
}
```

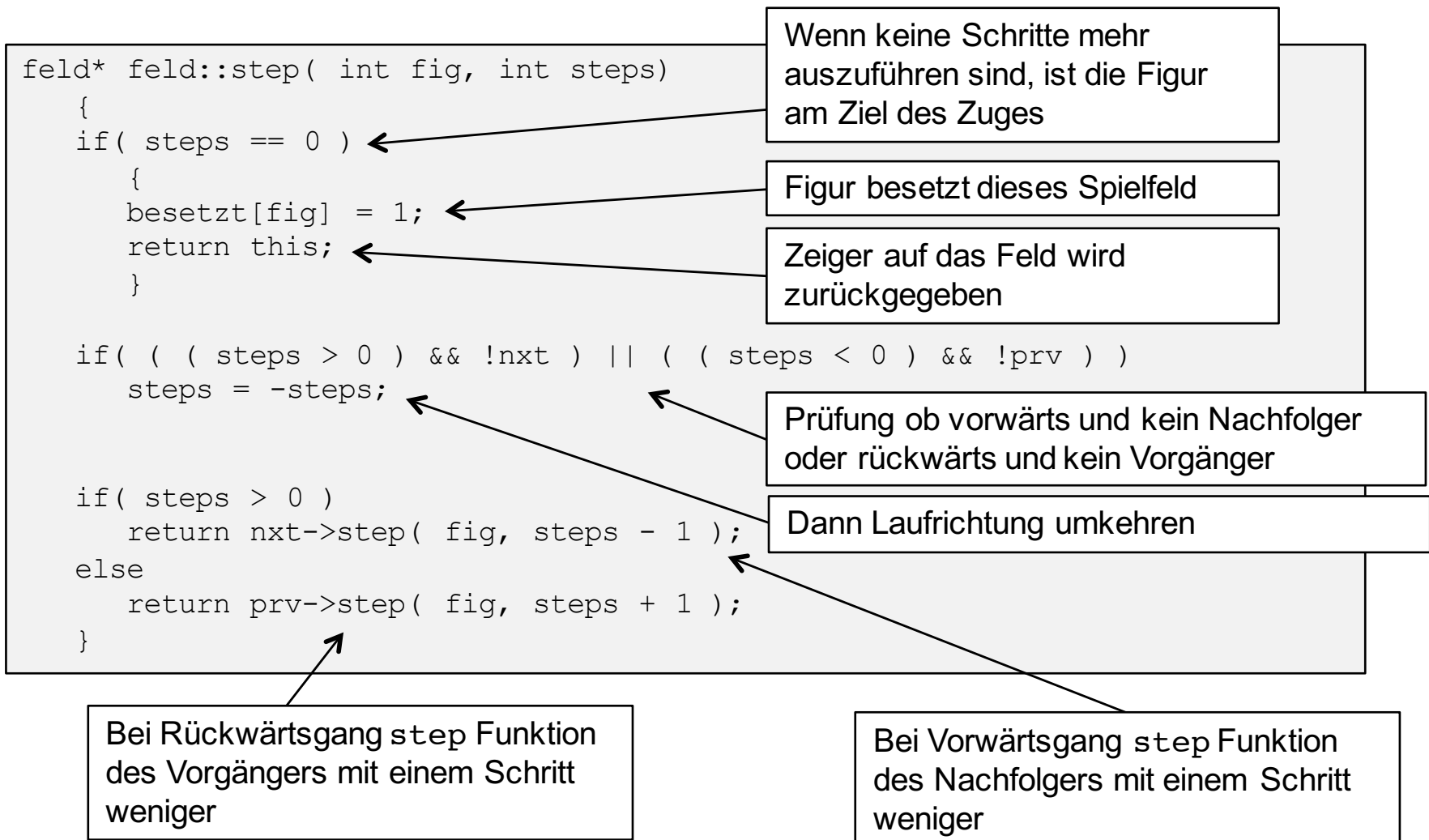
Wenn die Figur nicht auf dem
Feld oder blockiert ist, wird nicht
gesetzt

Figur verlässt dieses Spielfeld

Weitergehen der Figur wird durch den
Aufruf der `step` Funktion angestoßen

Weiterziehen einer Figur mit der step Funktion

Die step Funktion ist so implementiert, dass sie einen Zeiger auf das Feld zurückgibt, auf dem die Figur am Ende ihres Zuges zu stehen kommt. Je nachdem ob der Eingabeparameter positiv oder negativ ist, ist die Zugrichtung vorwärts oder rückwärts.



Spezialisierung der Spielfelder durch Vererbung mit der Klasse `ziel`

Durch Vererbung leiten wir nun von dem gewöhnlichen `Feld` die Klasse für das Zielfeld `ziel` und ab und verfeinern sie damit.

```
class ziel: public feld
{
public:
    ziel( feld* ende ) : feld( ende ) {};
    char getTyp() { return 'Z'; }
};
```

Die Klasse benötigt keinen besonderen Konstruktor, sie ruft nur den Konstruktor der Basisklasse auf und reicht den Parameter `ende` weiter.

Rückgabe von `'Z'` für Zielfeld als Typ

Das Zielfeld unterscheidet sich von einem einfachen `Feld` nur durch seine Kennung, die es über `getTyp` Methode als `'Z'` zurückmeldet. Der Parameter für den Aufruf des Konstruktors wird nur an den Konstruktor der Basisklasse weitergereicht. Dort wird die Verkettung vorgenommen.

Das Startfeld

Auch das Feld `start` leiten wir von dem gewöhnlichen Feld ab.

```
class start: public feld
{
public:
    start() : feld( 0 ) {};

    char getTyp() { return 'S'; }

    void figurAufstellen( int fig ) { besetzt[fig] = 1; }
};
```

Das Startfeld hat einen parameterlosen Konstruktor, der ein vorgängerloses Feld (ende = 0) instanziiert

Rückgabe von 'S' für Startfeld als Typ

Das Startfeld hat eine zusätzliche Methode, um eine Figur aufzustellen,

Neben dem modifizierten Konstruktor und der angepassten Methode `getTyp` mit dem Rückgabewert 'S', hat das Startfeld eine Methode zum Aufstellen einer Spielfigur am Spielbeginn. Alle anderen Felder können nur durch das Ziehen nach dem Würfeln erreicht werden.

Die Klasse für Sprungfelder

Das Sprungfeld erhält bei seiner Konstruktion die Information, wie viele Felder in welche Richtung es eine Figur versetzen soll:

```
class sprung: public feld
{
private:
    int offset;
    feld *step( int fig, int steps );

public:
    sprung( feld * ende, int off ) : feld( ende ) { offset = off; }
    char getTyp() { return offset > 0 ? '+' : '-'; }
};
```

Private Variable,
die Richtung und
Länge des
Sprunges
speichert

Der Konstruktor erhält neben dem
Listenende einen zusätzlichen
Parameter zu Sprungrichtung und
Länge

Als Typ wird '+' für ein Vorwärtssprungfeld
und '-' für ein Zurücksprungfeld geliefert

Der Konstruktor speichert die Information zu Sprungweite und -richtung in einer privaten Variable. Das Listenende wird an den Konstruktor des gewöhnlichen Feldes weitergereicht, damit dort die restlichen Initialisierungen und die Einkettung des Sprungfeldes in die Liste vorgenommen werden.

Die Methode `step` des Sprungfeldes

Wie zu erwarten, hat das Sprungfeld eine angepasste Methode `step`, die das Weiterziehen einer Figur handhabt, die auf das Feld trifft.

```
feld* sprung::step( int fig, int steps )  
{  
    if( steps == 0 )  
        steps = offset;  
  
    return feld::step( fig, steps );  
}
```

Wenn ein Zug auf dem Sprungfeld endet, wird der Zug um das `offset` in positive oder negative Richtung verlängert

Die weitere Behandlung des Zuges erfolgt über die `step` Methode der Basisklasse

In der Klasse wird der Zug um den Wert in der Variable `offset` verlängert, wenn der Zug auf dem Feld endet. Zur weiteren Bearbeitung des Zuges wird dann die Kontrolle an die `step` Methode der Basisklasse übergeben, die dann die weiteren Schritte ausführt.

Das Wartefeld

Als letztes Spezialfeld fehlt jetzt nur noch das Wartefeld:

```
class warte: public feld
```

```
{  
private:
```

```
    int timeout;
```

```
    int wait[4];
```

```
    feld *step( int fig, int steps );
```

```
public:
```

```
    warte( feld* ende, int t ) : feld( ende ) { timeout = t; }
```

```
    char getTyp() { return 'W'; }
```

```
    int blockiert( int nummer );
```

```
};
```

Zähler mit der individuellen Wartezeit
für jede wartende Figur

Private Variable
für die Wartezeit
des Feldes

Der Konstruktor erhält neben dem
Listenende einen zusätzlichen Parameter
zur Wartezeit

Als Typ gibt das Wartefeld 'W' zurück

Das Wartefeld verfügt über eine spezifische Methode
blockiert, die zusammen mit einer spezifischen
step Methode die Funktionalität des Wartefeldes
implementiert

Die spezifischen Methoden des Wartefeldes

Das Wartefeld hat eine `blockiert` Methode und eine spezifische `step` Methode:

```
int warte::blockiert( int nummer )
{
    if( wait[nummer] )
    {
        wait[nummer] --;
        return 1;
    }
    return 0;
}
```

Bei jeder Anfrage ob eine Figur blockiert ist, wird die eventuell noch vorhandene Wartezeit reduziert, die Figur bleibt weiter blockiert

Steht keine Wartezeit für die Figur an, so ist sie nicht blockiert und kann ziehen

```
feld * warte::step( int fig, int steps )
{
    if( steps == 0 )
        wait[fig] = timeout;
    return feld::step( fig, steps );
}
```

Wenn der Zug einer Figur auf dem Feld endet, wird sie für die eingestellte Anzahl von Zügen blockiert

Die weitere Behandlung des Zuges erfolgt über die `step` Methode der Basisklasse

Optionale weitere Ereignisfelder

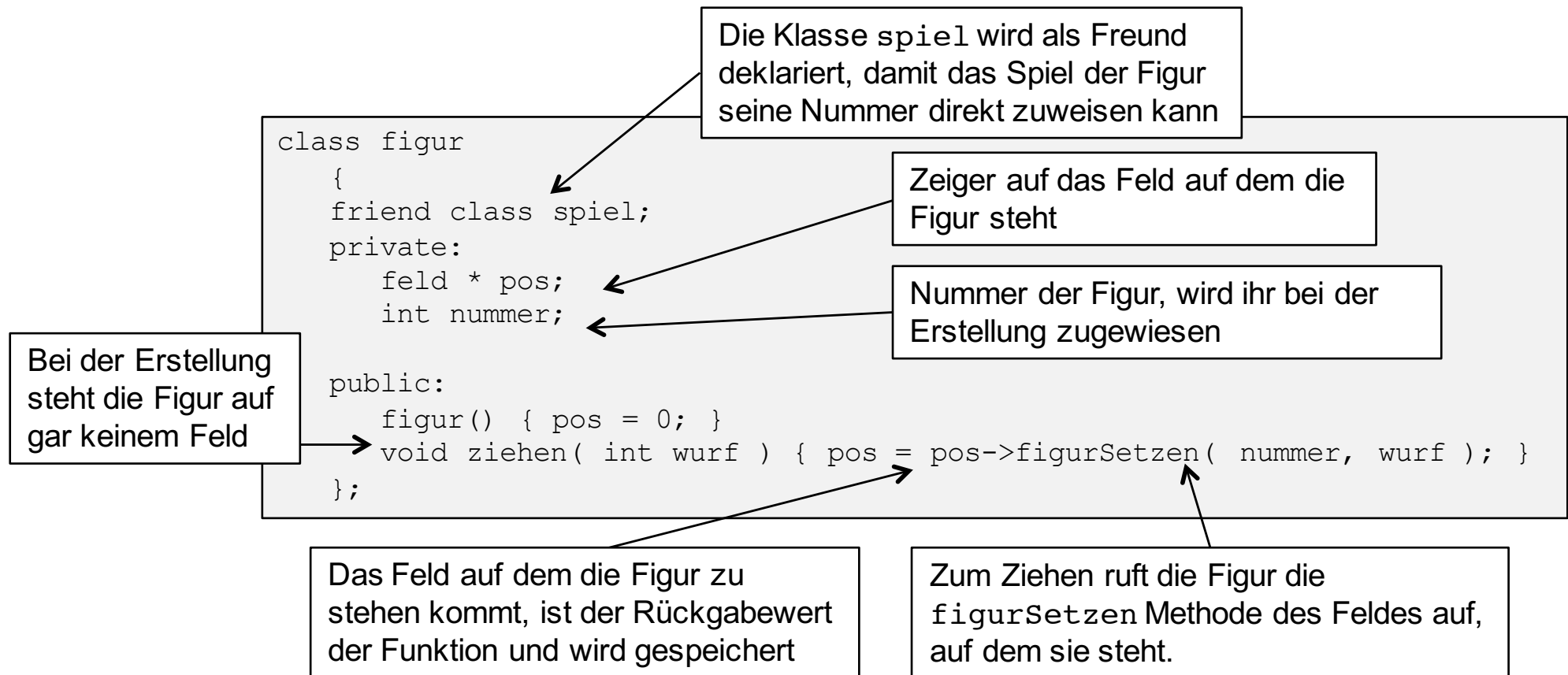
Einige mögliche Ereignisfelder haben wir nun beispielhaft implementiert. Durch das Design unseres Spiels könnten weitere Ereignisfelder leicht umgesetzt und integriert werden:

- Ein Feld, von dem man nur nach Wurf einer bestimmten Augenzahl weiterziehen darf
- Ein Feld bei dem der nächste Wurf rückwärts zählt
- Ein Feld das man nur überspringen kann, wenn man mindestens über drei Augen über das Feld hinaus gewürfelt hat. Andernfalls bleibt man auf dem Feld stehen oder muss den Rest des Wurfes aussetzen

Bei der Implementierung wird man feststellen, dass es mit der objektorientierten Entwicklung sehr einfach ist, neue Klassen hinzuzufügen und diese konsistent in die bestehende Programmstruktur einzubauen.

Implementierung der Spielfigur **figur**

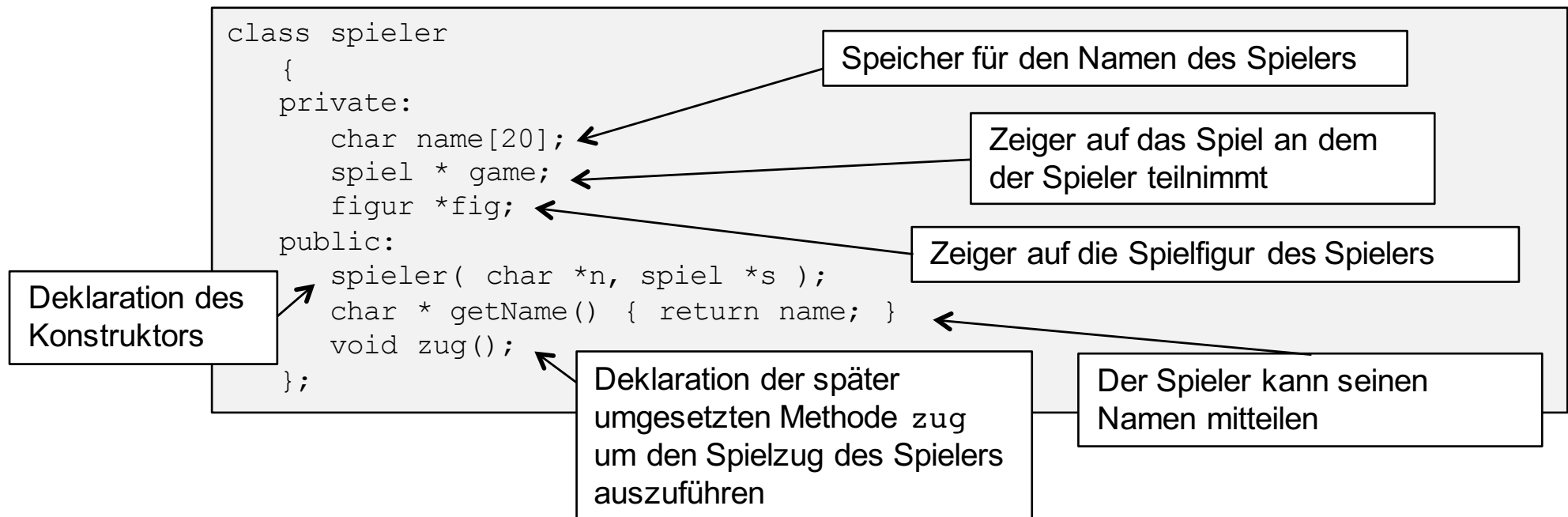
Zur Implementierung der Spielfigur muss nur noch wenig Code erstellt werden:



Die `figurSetzen` Methode des Feldes auf dem die Figur steht, ruft jeweils fortlaufend die `step` Methoden der verketteten Felder des Spielplanes auf, bis die Figur ihr endgültiges Ziel erreicht hat, das dann als Rückgabewert an die `figur` übergeben wird.

Implementierung des Spielers `spieler`

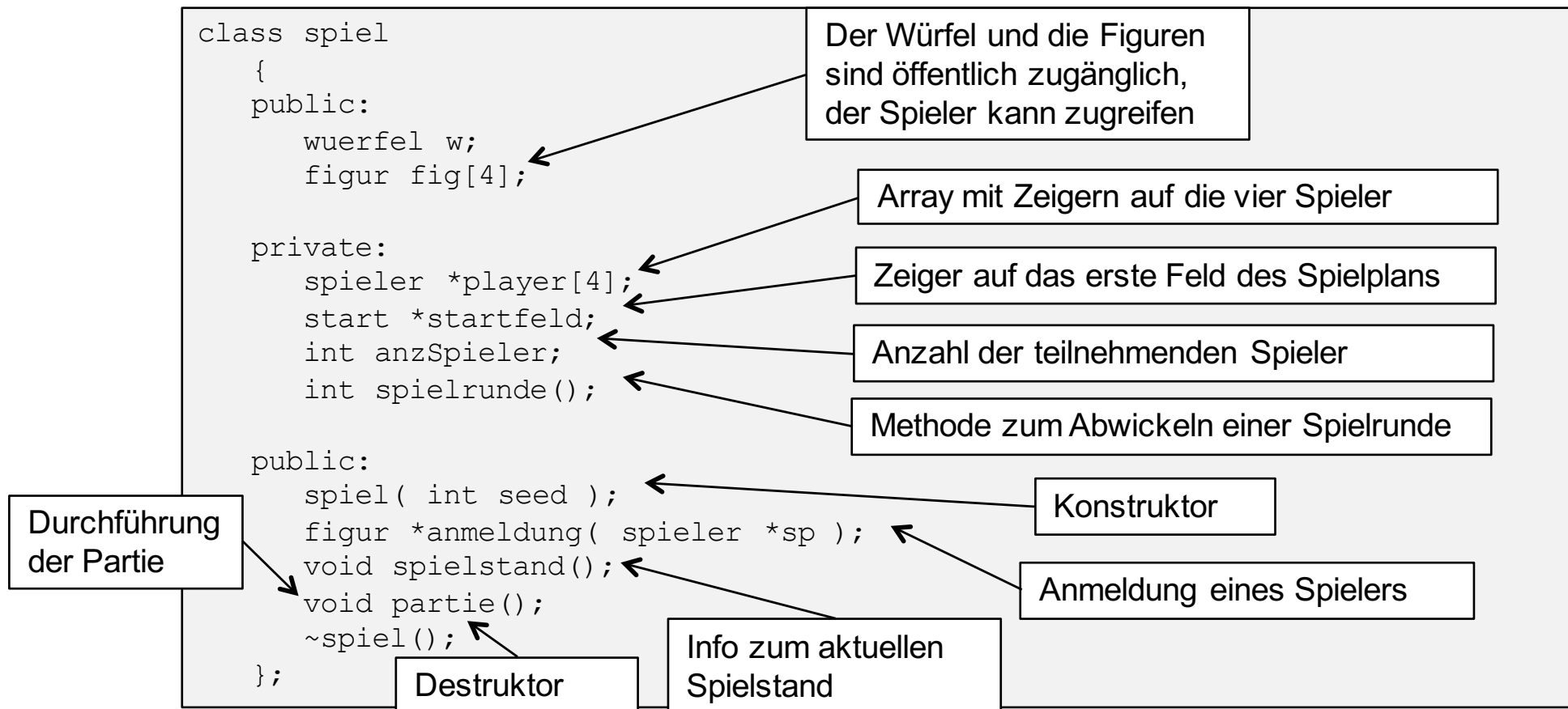
Der Spieler muss nur noch wenig tun, da das notwendige Verhalten in den anderen Elementen implementiert ist. Daher ist auch hier die Deklaration übersichtlich, insbesondere da eine Methode und der Konstruktor auch erst später implementiert werden:



Den Konstruktor und die Methode `zug` können wir erst später implementieren, da diese Methoden auf das Spiel in der Klasse `spiel` zugreifen, das wir vorab umsetzen müssen.

Die Klasse `spiel`

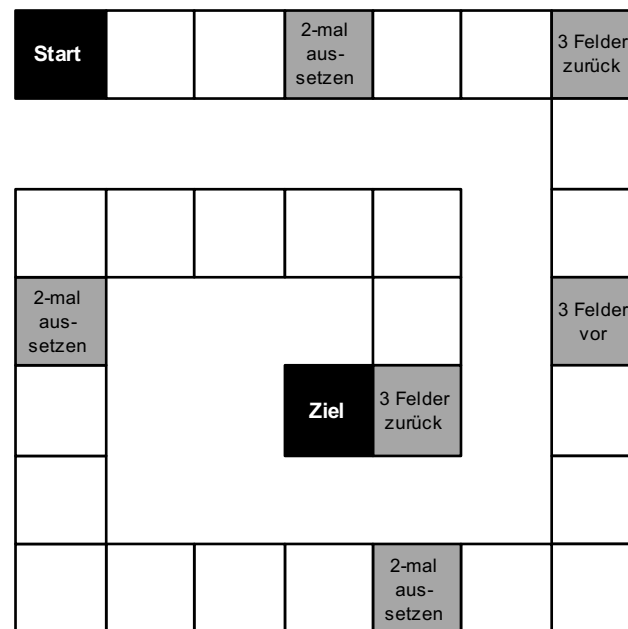
Die Deklaration der Klasse `spiel` zeigt sich folgendermaßen:



Über das erste Feld des Spielplans ist das gesamte Spielfeld zugreifbar, da über dieses Feld die komplette verkettete Liste zugreifbar ist. Die Methode `spielrunde` ist privat, das sie nur über die (öffentlich verfügbare) Methode `partie` verwendet wird. Diese wickelt die gesamte Partie rundenweise ab.

Aufbau des Spielfeldes

Im Konstruktor der Klasse `spiel` wird das Spielfeld aufgebaut, indem die einzelnen Felder des Spielplan instanziiert werden. Die Verkettung der Felder erfolgt jeweils über deren Konstruktor. Dazu wird immer das aktuell letzte Feld (`last`) im Spielplan an den Konstruktor übergeben. Dort wird das Feld eingekettet. Das erstellte Feld ist nach der Einkettung dann das neue letzte Feld, der Rückgabewert wird entsprechend gespeichert. Bei Aufbau des Spielplans gibt es noch keine Spieler. Die Anzahl der Spieler wird später in der Methode `anmeldung` für jeden neu angemeldeten Spieler inkrementiert.



```
spiel::spiel( int seed ) : w( seed )
{
    feld *last;
    anzSpieler = 0;
    last = startfeld = new start;
    last = new feld( last );
    last = new feld( last );
    last = new warte( last, 2 );
    last = new feld( last );
    last = new feld( last );
    last = new sprung( last, -3 );
    last = new feld( last );
    last = new feld( last );
    last = new sprung( last, +3 );
    last = new feld( last );
    last = new feld( last );
    last = new feld( last );
    last = new feld( last );
    last = new warte( last, 2 );
    last = new feld( last );
    last = new feld( last );
    last = new feld( last );
    last = new feld( last );
    last = new feld( last );
    last = new warte( last, 2 );
    last = new feld( last );
    last = new feld( last );
    last = new feld( last );
    last = new feld( last );
    last = new feld( last );
    last = new sprung( last, -3 );
    last = new ziel( last );
}
```

Aktuelle Spielerzahl

Komplexere Spielpläne

Im Aufbau des Spielplanes wird jeweils nur eine einheitliche Sprungweite (hier +3 und -3) für die Sprungfelder vorwärts und rückwärts verwendet. Dieses Vorgehen ist gewählt worden, um die Ausgabe des Spielplanes übersichtlich zu halten. Für die Sprungfelder wird dort jeweils nur das Vorzeichen der Sprungrichtung ohne Sprungweite dargestellt. Die Sprungfelder sind aber so implementiert, dass jedes Feld einen individuellen Wert verwalten kann. Wollte man dies nutzen, müsste allerdings die Ausgabe angepasst werden. Gleiches gilt für die Wartefelder, die im hier implementierten Spielplan alle eine einheitliche Wartezeit haben und dort immer als 'W' dargestellt werden.

In einer Erweiterung wäre es auch problemlos möglich, die Konfiguration des Spielplanes aus einer entsprechenden Datei zu laden und den Plan mit diesen Daten zu erstellen.

Der Destruktor der Klasse `spiel`

Im Destruktor der Klasse `spiel` werden die notwendigen Aufräumarbeiten erledigt:

```
spiel::~~spiel()
```

```
{
```

```
    feld *f, *t;
```

```
    for( f = startfeld; t = f; )
```

```
    {
```

```
        f = f->getNext();
```

```
        delete t;
```

```
    }
```

```
}
```

Zuweisung von `t = f` als Schleifenbedingung, bricht ab, wenn ein Feld keinen Nachfolger mehr hatte

Alle Felder des Spielfeldes werden abgefahren

Feld wird freigegeben

Anmeldung eines neuen Spielers

Neue Spieler können sich am `spiel` anmelden:

```
figur * spiel::anmeldung( spieler *sp )
{
    if( anzSpieler >= 4 )
        return 0;
    player[anzSpieler] = sp;

    fig[anzSpieler].nummer = anzSpieler;
    startfeld->figurAufstellen( anzSpieler );
    fig[anzSpieler].pos = startfeld;

    cout << anzSpieler + 1 << ".ter Spieler ist " << sp->getName() << '\n';

    anzSpieler++;
    return fig+anzSpieler-1;
}
```

Anmeldung erfolgt nur, wenn noch keine vier Spieler gemeldet sind

Zeiger auf den neu angemeldeten Spieler wird zur späteren Verwendung gespeichert

Nummer des Spielers wird gesetzt (`spiel` ist friend von `spieler`)

Figur des neuen Spielers wird gestellt

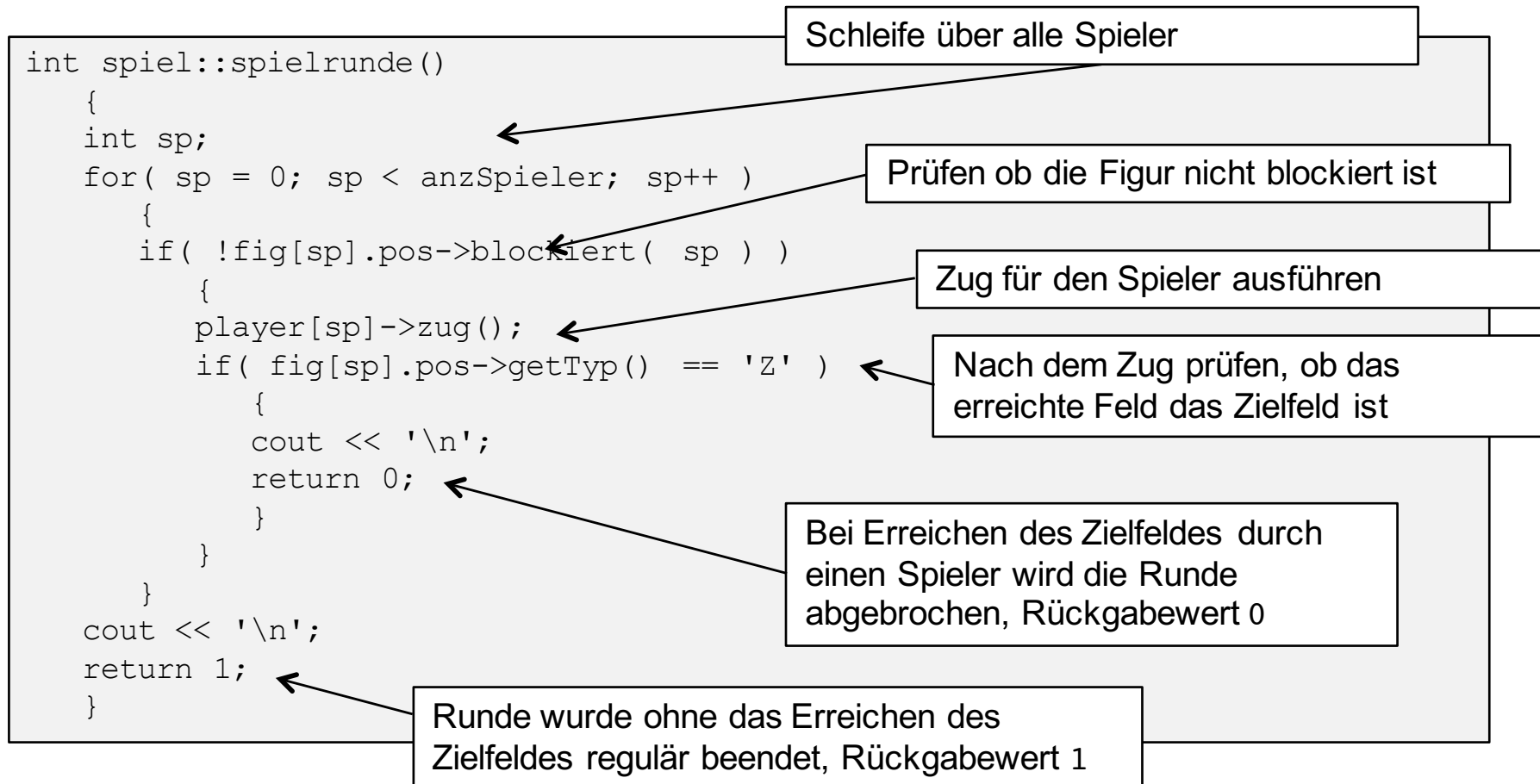
Erhöhen der Anzahl der Spieler im aktuellen Spiel

Das Startfeld wird als aktuelles Feld der Figur gesetzt

Rückgabe eines Zeigers auf die Spielfigur per Zeigerarithmetik, entspricht dem Code `&fig[anzSpieler - 1]`

Durchführen einer Spielrunde

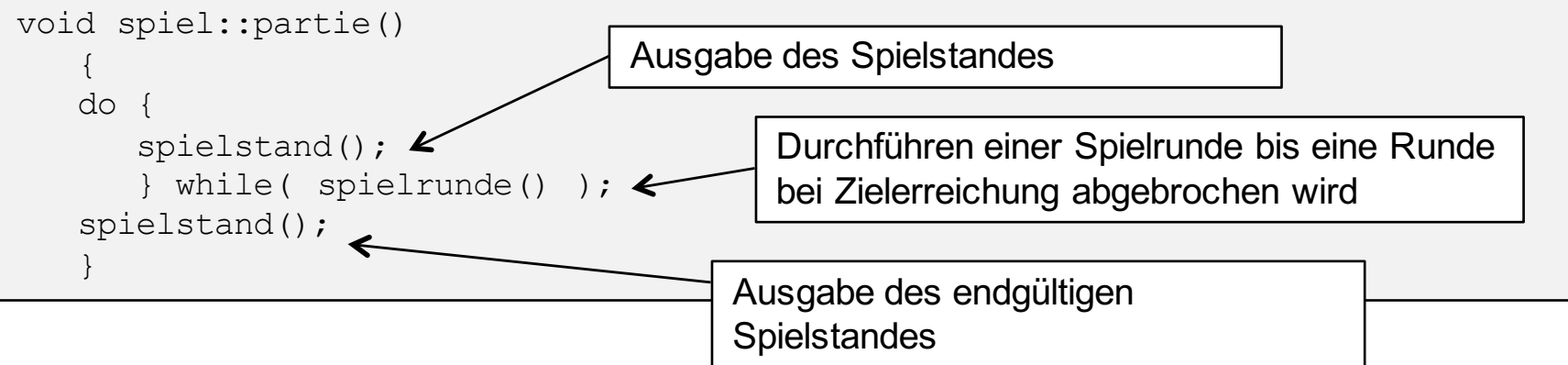
Jetzt sind alle Vorbereitungen getroffen, um eine Spielrunde auch durchführen zu können:



Da das Verhalten in den einzelnen Klassen implementiert ist, muss in einer Spielrunde für jede Figur lediglich überprüft werden, ob die Figur blockiert ist oder ziehen darf. In dem Fall wird der Zug ausgeführt und geprüft, ob das erreichte Feld das Zielfeld ist. Ist das der Fall, wird die aktuelle Spielrunde abgebrochen.

Die Durchführung der **partie**

Zur Durchführung einer Partie müssen in der entsprechenden Methode nur noch die Spielrunden abgearbeitet werden:



Die Ausgabe des Spielstandes

Zur Ausgabe des Spielstandes wird eine einfache Repräsentation des Spielfeldes und der Position der jeweiligen Spielfigur ausgegeben:

```
void spiel::spielstand()
```

```
{
```

```
feld *fld;
```

```
int fg;
```

```
for( fld = startfeld; fld; fld = fld->getNext() )
```

```
    cout << fld->getTyp();
```

```
    cout << '\n';
```

```
for( fg = 0; fg < 4; fg++ )
```

```
{
```

```
    for( fld = startfeld; fld; fld = fld->getNext() )
```

```
    {
```

```
        if( fld == fig[fg].pos )
```

```
            cout << *( player[fg]->getName() );
```

```
        else
```

```
            cout << '-';
```

```
    }
```

```
    cout << '\n';
```

```
}
```

```
cout << '\n';
```

```
}
```

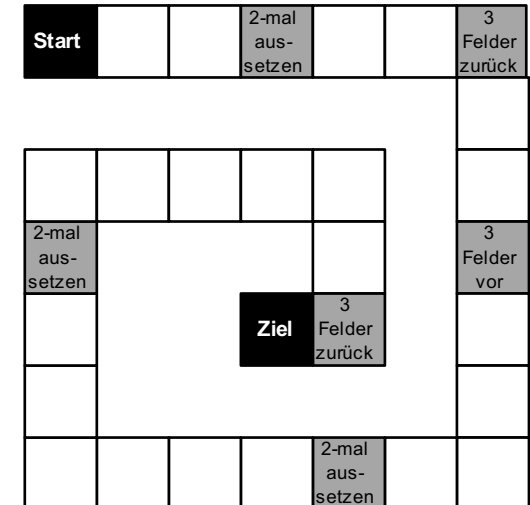
Ausgabe des Typs
aller Spielfelder

Für alle Spieler

Durchgehen
aller Felder

Wenn Figur auf dem
Feld steht, dann Name
(Initial) ausgeben

Ansonsten Platzhalter '-'
drucken



```
S..W.-...+....W.....W.....-Z
-A-----
---B-----
---C-----
---D-----
```

Konstruktor der Klasse `spieler`

Den Konstruktor der Klasse `spieler` hatten wir bis nach der Erstellung der Anmeldung verlagert. Da diese nun ausgeführt ist, implementieren wir nun auch den Konstruktor:

```
spieler::spieler( char *n, spiel *s )  
{  
    strcpy( name, n );  
    game = s;  
    fig = game->anmeldung( this );  
}
```

Kopieren des Namens in das interne Namensarray (maximal 20 Zeichen)

Verbindung zum Spiel speichern

Spieler beim Spiel anmelden und zurückgegebenen Zeiger auf die Figur speichern

Implementierung der Methode zug

Abschließend fehlt nun nur noch die Methode `zug`, um einen einzelnen Zug für den Spieler auszuführen:

```
void spieler::zug()
```

```
{
```

```
    int wurf;
```

```
    wurf = game->w.wurf();
```

```
    cout << *name << '=' << wurf << ' ';
```

```
    fig->ziehen( wurf );
```

```
}
```

Speichern der Augenzahl des
Würfelwurfes

Ausgabe des
Spielernamens und der
gewürfelten Augenzahl

Ziehen der Spielfigur mit dem
gewürfelten Wert

Das Hauptprogramm

Das Hauptprogramm instanziert die vier Spieler und startet die Partie:

```
int main()
{
    spiel sp( 1234 );

    spieler anton( "Anton", &sp );
    spieler berta( "Berta", &sp );
    spieler claus( "Claus", &sp );
    spieler doris( "Doris", &sp );
    cout << '\n';
    sp.partie();

    return 0;
}
```

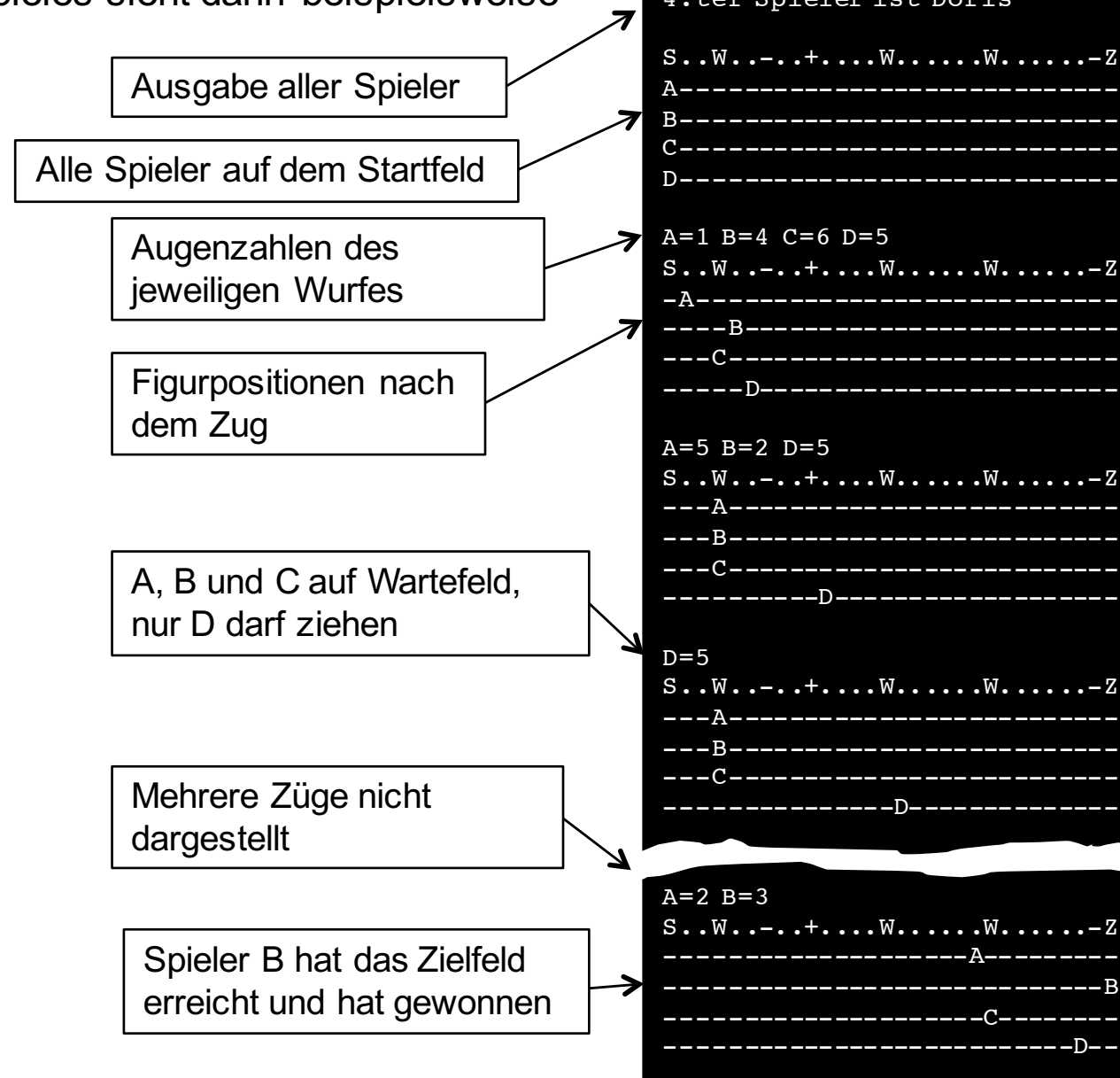
Instanzieren des Spieles mit Übergabe
eines Wertes zur Initialisierung des
Zufallszahlengenerators

Instanziierung der Spieler

Starten der Partie

Protokoll eines Spieles

Das Protokoll eines Spieles sieht dann beispielsweise folgendermaßen aus:



Nachbetrachtung der virtuellen Methoden

Wir wollen noch einmal die Bedeutung der virtuellen Methoden für die Implementierung betrachten. In der Deklaration von `feld` haben wir drei Methoden als virtuell gekennzeichnet

```
class feld
{
private:
    feld *nxt;
    feld *prv;
protected:
    int besetzt[4];
    virtual feld *step( int fig, int steps );
public:
    feld( feld *ende );
    feld* getNext() { return nxt; }
    virtual char getTyp() { return '.'; }
    feld * figurSetzen( int fi, int wurf );
    virtual int blockiert( int nummer ) { return 0; }
};
```

Virtuelle Methoden

wir können die Kennzeichnung entfernen, indem wir das Schlüsselwort `virtual` löschen. Das Programm lässt sich danach weiterhin problemlos übersetzen. Wenn wir das Programm danach aber starten, dann erhalten wir eine geänderte Ausgabe:

Geänderte Ausgabe des Programms

Bereits an der ersten Ausgabe des Programms ist eine deutliche Veränderung zu erkennen:

```
1.ter Spieler ist Anton  
2.ter Spieler ist Berta  
3.ter Spieler ist Claus  
4.ter Spieler ist Doris
```

```
.....  
A-----  
B-----  
C-----  
D-----
```

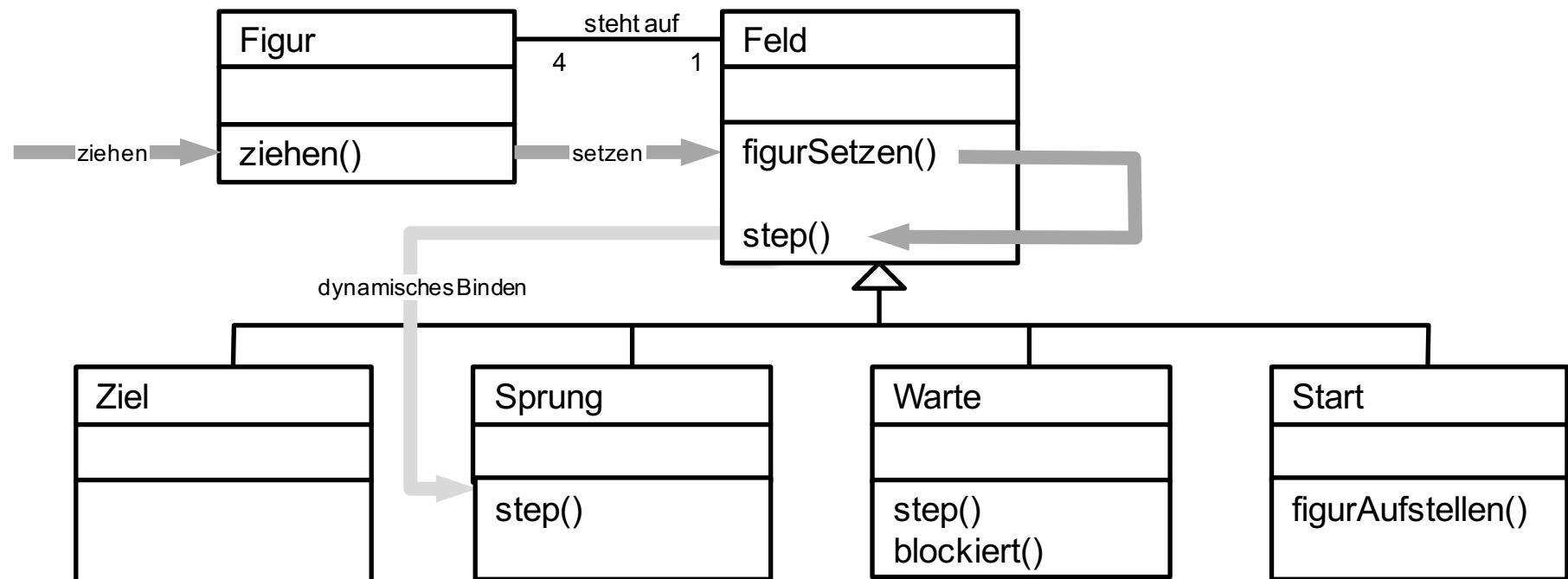
In der Ausgabe von `getTyp` für das Spielfeld taucht nur noch der Punkt `'.'` des gewöhnlichen Felds auf

Auch die weiteren Ausgaben zeigen, dass jetzt alle Felder wie gewöhnliche Felder behandelt werden. Wir haben Start-, Ziel-, Warte- und Sprungfeld im Konstruktor des Spieles eingerichtet, sie werden jetzt aber offensichtlich nicht mehr als solche erkannt.

Dynamisches Binden zur Laufzeit

Die Felder sind korrekt als Start-, Werte oder Spungfelder instanziiert worden, im Laufe des Spiels werden sie dann aber als gewöhnliche Felder angesprochen, da der Spielplan als verkettete Liste die Zeiger gewöhnlicher Felder verwaltet.

Ohne die virtuellen Methoden verhalten sich die Felder dann auch wie gewöhnliche Felder. Erst bei der Verwendung virtueller Funktionen wird zur Laufzeit geprüft, um welche konkreten Feldtypen es sich im einzelnen handelt und welche Methoden tatsächlich aufgerufen werden sollen.

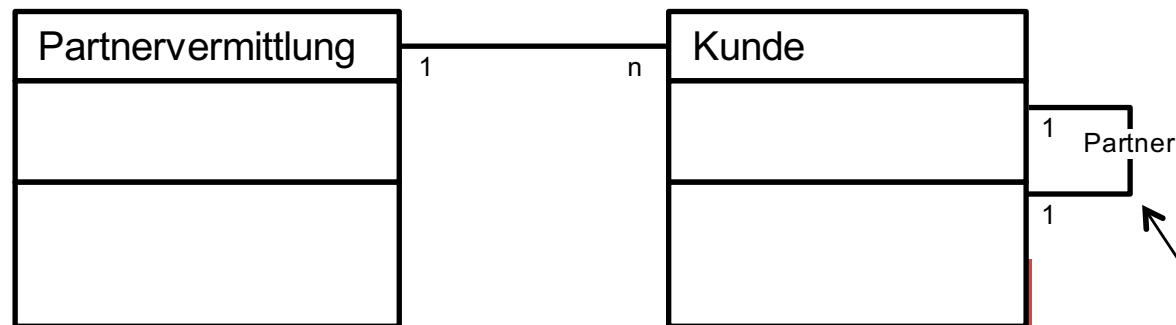


Partnervermittlung

Als weiteres Beispiel wollen wir uns dem delikaten Problem der Partnervermittlung zuwenden und einen entsprechenden Dienst betrachten, der seinen Kunden die Vermittlung anbietet.

Auch hier wollen wir zunächst die beteiligten Klassen identifizieren und ein Klassendiagramm erstellen. Im Zentrum unserer Betrachtung steht eine Partnervermittlung und deren Kunden. Die Kunden suchen Partner, die Agentur ist bestrebt, Paare zusammenzubringen.

Damit ergeben sich direkt die beiden ersten Klassen:



Der Bezug der Klasse auf sich selbst, ergibt sich aus der Verwendung zur Laufzeit. Dort ist der Partner eine andere Instanz der gleichen Klasse Kunde

Das Zusammenspiel von Partnervermittlung und Kunde

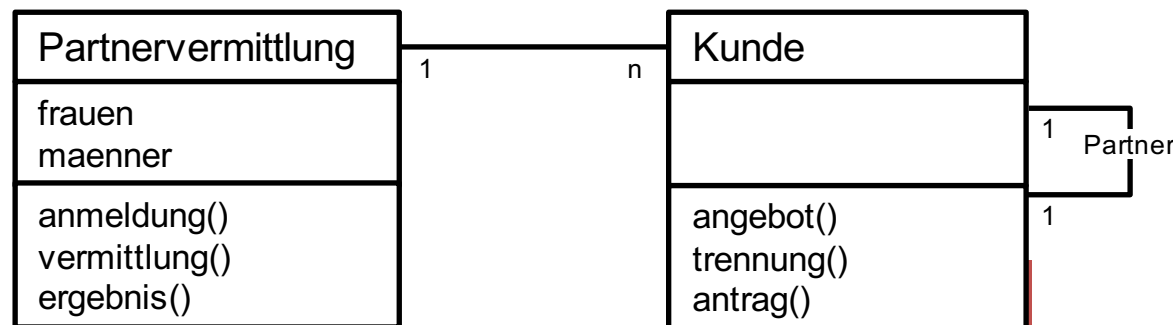
Die Partnervermittlung hat eine Kartei mit Kunden. Diese Kunden werden in der Agentur in **Männer** und **Frauen** unterschieden. Ein Kunde sucht eine **Partnerschaft** zu einem anderen Partnersuchenden und hat gegebenenfalls bereits einen anderen **Partner** gefunden.

Die Dynamik im Verhältnis von Partnervermittlung und Kunde und auch der Kunden untereinander beschreiben wir durch Methoden und Botschaften. Die Agentur kann:

- Eine Anmeldung neuer Kunden entgegennehmen
- Vermittlungen durchführen
- Vermittlungsergebnisse bekanntgeben

Ein Kunde kann:

- Sich mit einem Angebot der Agentur auseinandersetzen
- Einem anderen Kunden einen Antrag machen bzw. einen solchen Antrag von einem anderen Kunden entgegennehmen
- Sich von seinem derzeitigen Partner trennen, um sich mit einem anderen Partner zu verbinden

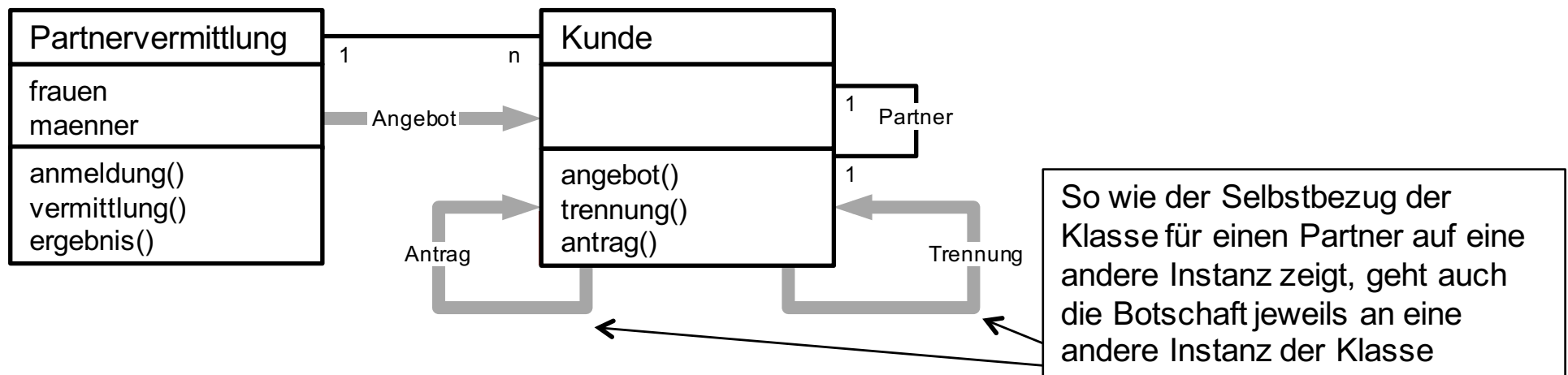


Die Botschaften zwischen den Objekten

Der Ablauf der Vermittlung und Partnersuche wird über Botschaften gesteuert und läuft wie folgt ab:

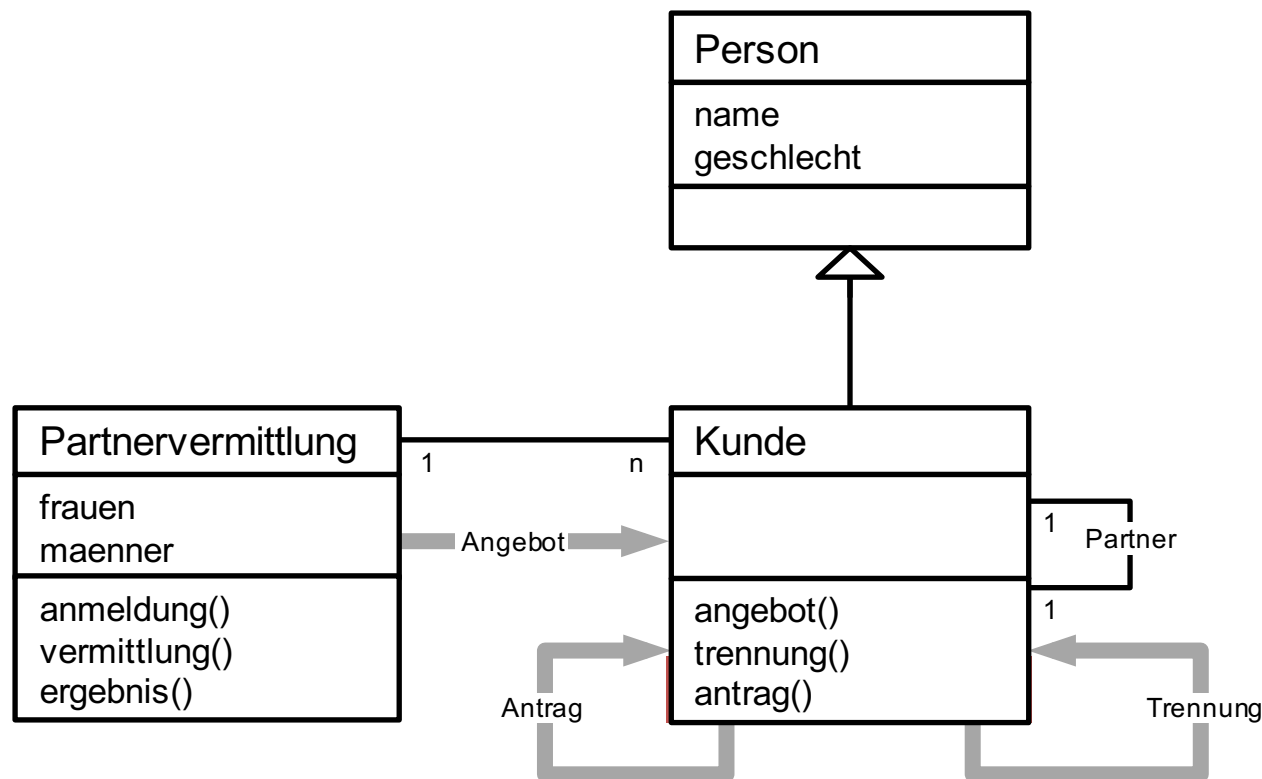
Die Agentur schickt einem Kunden ein **Angebot**, das Informationen über einen anderen Kunden enthält. Der Empfänger des Angebotes prüft, ob der angebotene Partner seinen Vorstellungen entspricht und gegebenenfalls seinem derzeitigen Partner vorzuziehen ist. Ist das der Fall, so macht der Empfänger dem von der Agentur angebotenen Partner einen **Antrag**.

Der so angesprochene Wunschpartner prüft natürlich auch, ob er sich durch den Antrag verbessern kann. Ist das der Fall, so trennt er sich von seinem bisherigen Partner (**Trennung**) und nimmt den Antrag an. Andernfalls lehnt er den Antrag ab. Im Falle eines positiven Bescheids trennt sich dann auch der Antragsteller von seinem bisherigen Partner (sofern er einen hatte) und die neue Partnerschaft ist besiegelt.



Ableitung des Kunden von der Basisklasse person

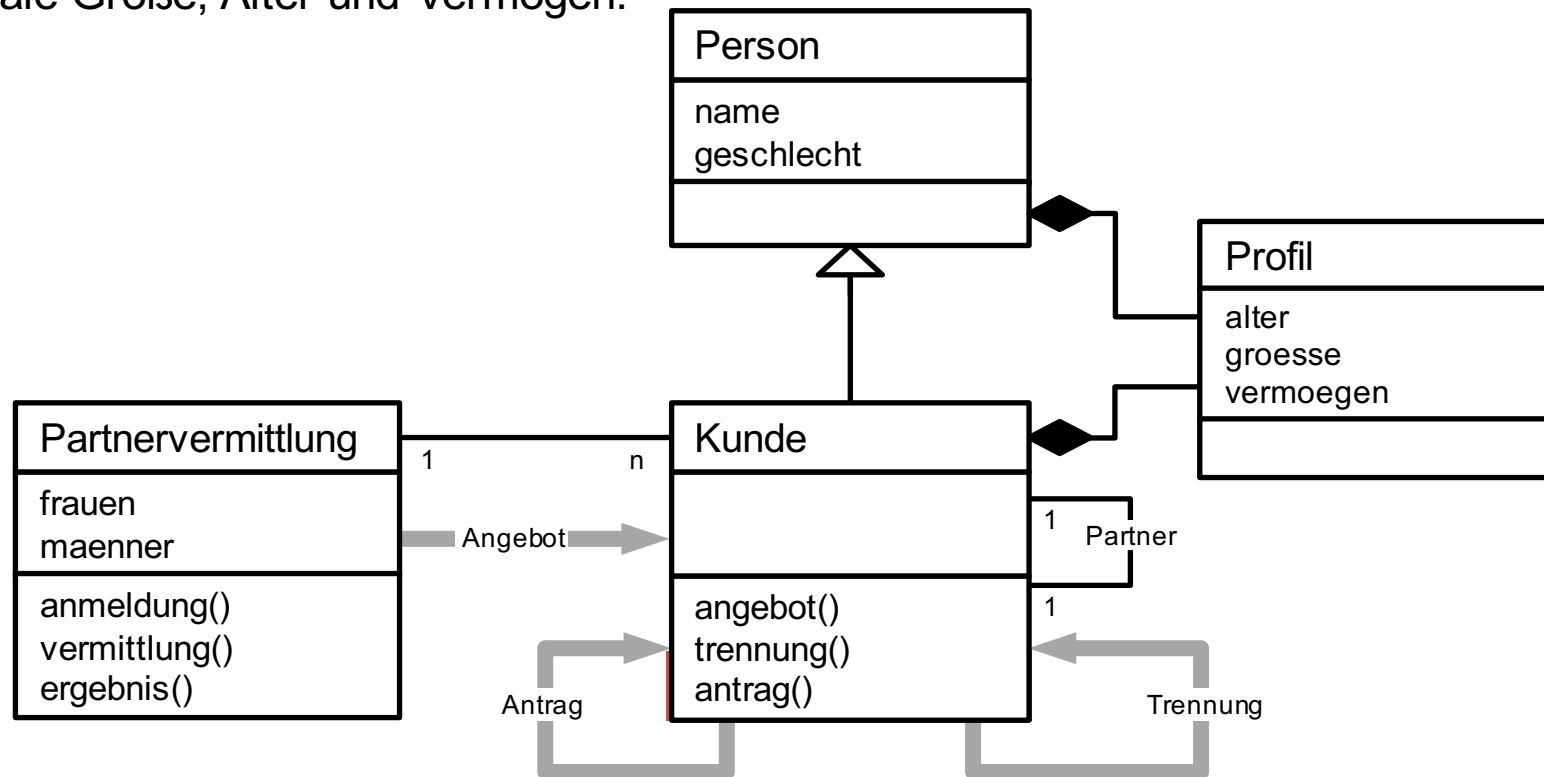
Ein Kunde hat bisher keine besonderen Merkmale, außer dass er gegebenenfalls einen Partner hat. Wir geben ihm jetzt weitere Eigenschaften, indem wir ihn zu einer **Person** machen:



Eine Person hat einen Namen und ein Geschlecht. Letzteres ist bei der Partnersuche natürlich wichtig. Der Kunde **ist eine** Person und erbt deren Eigenschaften.

Erweiterung der Modells um Profile

Die beiden Eigenschaften Name und Geschlecht reichen natürlich noch nicht aus, damit sich ein Kunde ein Bild von seinem möglichen Partner machen kann. Er benötigt dazu weitere Informationen, nämlich ein **Profil** des Partners. Wir beschränken uns für das Profil auf die Merkmale Größe, Alter und Vermögen.



Ein Profil kommt in unserem Modell in zweierlei Bedeutung vor. Zum einen hat jede person ein Eigenprofil und zum anderen hat jeder kunde ein Wunschprofil, (Partnerprofil) seines zukünftigen Partners. Unser Modell haben wir bereits in diesem Sinne ergänzt.

Abgleich der Profile

Die Partnerwahl besteht damit im Wesentlichen aus einem Abgleich des Wunschprofils mit dem Eigenprofil eines möglichen Partners.

Nun gibt es sicherlich unterschiedliche Typen von Kunden. Wir werden im weiteren drei Spezialisierungen implementieren:

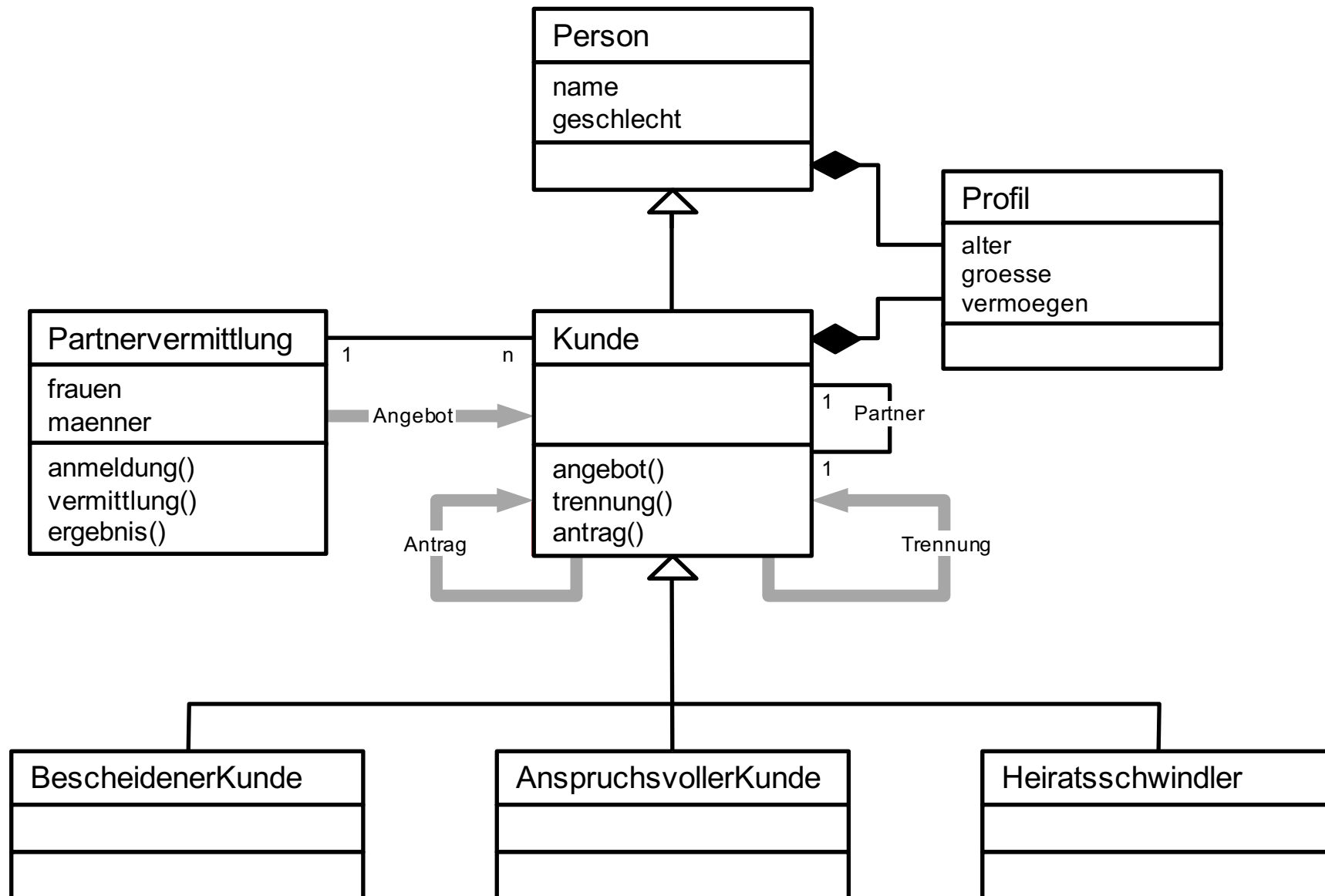
- Den **bescheidenen Kunden**
- Den **anspruchsvollen Kunden**
- Den **Heiratsschwindler**

Die drei Kundentypen werden bei der Partnersuche auch verschiedene Kriterien anlegen, die wir später noch kennenlernen.

Wir übernehmen diese Spezialisierungen der Klasse `kunde` in unser Design und erhalten dann folgendes Diagramm:

Vollständige Klassendiagramm Partnervermittlung

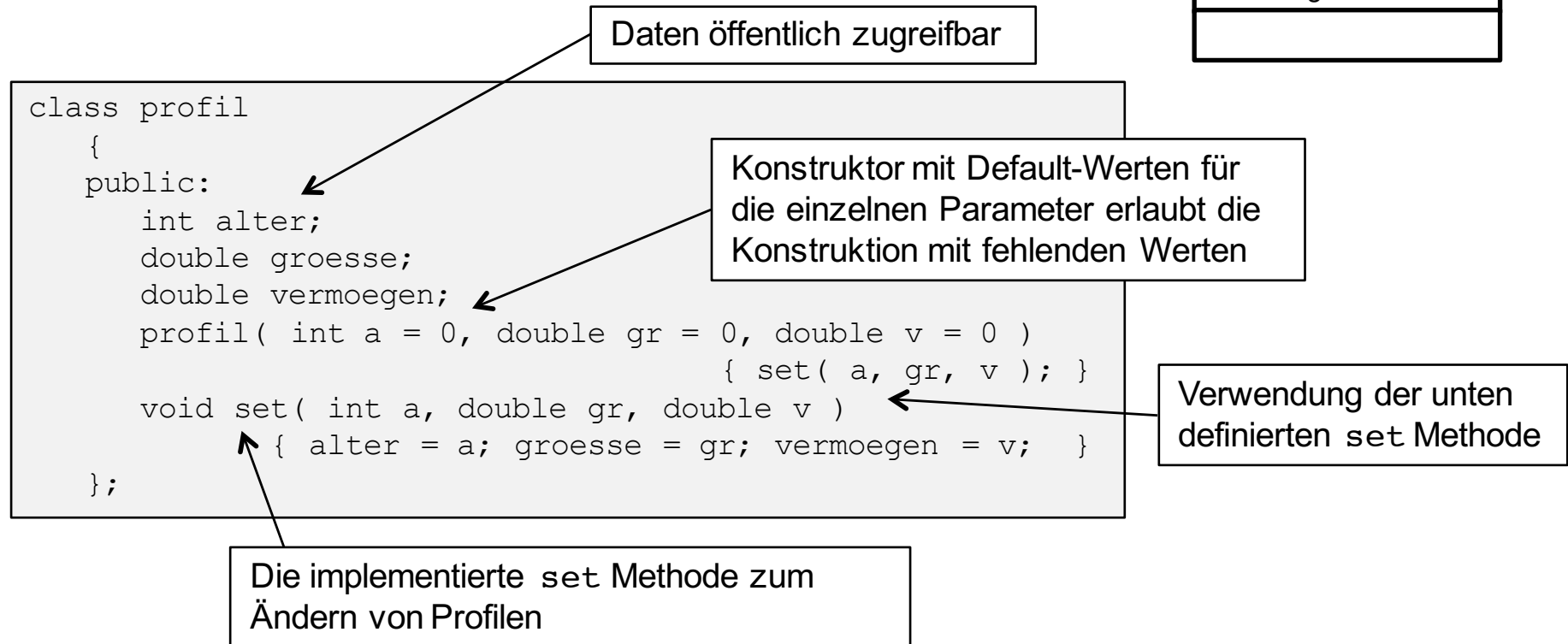
Damit ist unser Klassendiagramm vollständig und wir können mit der Implementierung starten.



Realisierung

Wir beginnen mit der Implementierung der Klasse `profil`. Das Design gibt uns dabei die grobe Struktur bereits vor:

Profil
alter groesse vermoegen



Die Daten werden im Konstruktor über eine `set` Methode gesetzt, die wir auch noch an anderer Stelle verwenden werden, um die Werte eines Profils zu modifizieren.

In der Klasse `profil` sind alle Attribute und Methoden als öffentlich deklariert worden, um das Beispiel übersichtlicher zu halten. Generell sollten zumindest entsprechende Getter vorgesehen werden.

Relative Abweichung

Die Kunden sollen anhand ihres Partnerprofils und des Eigenprofils eines anderen Kunden ermitteln können, wie groß die „Abweichung“ der Profile ist. Dazu wollen wir die Abweichung zweier Profile voneinander messbar machen.

Aus der Mathematik wissen wir, dass man die relative Abweichung einer Zahl b von einer Zahl a durch die Formel

$$\left| \frac{a - b}{a} \right|$$

messen kann, zumindest solange a nicht 0 ist

Diese Abweichung lassen wir durch eine kleine Hilfsfunktion berechnen:

```
double abweichung( double a, double b )  
{  
    if( !a )  
        return 0;  
  
    return ( double ) fabs( a - b ) / a;  
}
```

Ermittlung des Betrages über die
Absolutwertfunktion aus der
Standardbibliothek



Abweichung zwischen zwei Profilen

Aufbauend auf unsere Hilfsfunktion können wir nun die Abweichung zweier Profile berechnen:

Gleicher Funktionsname wie für die Hilfsfunktion,
durch unterschiedliche Parametersignaturen aber
problemlos möglich

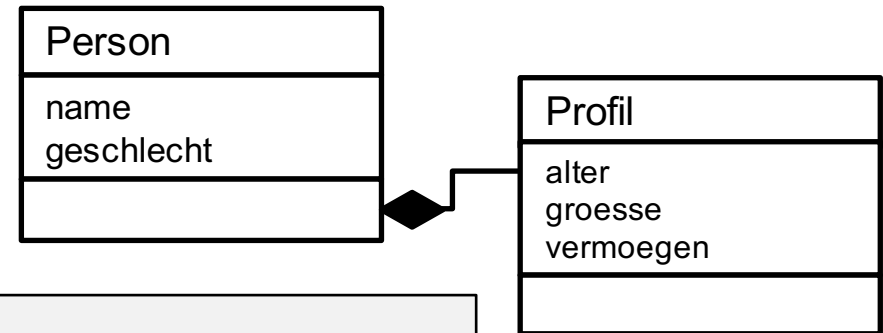
```
double abweichung( profil &wunsch, profil &real )
{
    double sum = 0;

    sum = abweichung( wunsch.alter, real.alter );
    sum += abweichung( wunsch.groesse, real.groesse );
    if( wunsch.vermoegen > real.vermoegen )
        sum += abweichung( wunsch.vermoegen, real.vermoegen );
    return sum / 3;
}
```

Wir bestimmen in der Funktion die relative Abweichung von Alter und Größe und addieren sie. Die relative Abweichung zwischen dem gewünschten und tatsächlichen Profil beim Vermögen berücksichtigen wir nur, wenn das Vermögen kleiner ist als gewünscht. Wir nehmen damit an, dass sich alle damit arrangieren können, wenn der mögliche Partner vermögender ist als gewünscht.

Die Klasse person

Wir erstellen nun die Klasse person. Diese ist die Basisklasse für alle Kunden ist, seien es nun bescheidene, anspruchsvolle oder auch Heiratsschwinder:



```
class person
{
private:
    char name[20];
    char geschlecht;
public:
    profil eigenprofil;
    person( char * n, char g, int a, double gr, double v );
    char *getName() { return name; }
    char getGeschlecht() { return geschlecht; }
};
```

Speicherung des Geschlechts als 'm' oder 'w'

eigenprofil im öffentlichen Bereich, von überall zugreifbar

```
person::person( char * n, char g, int a, double gr, double v )
{
    strcpy( name, n );
    geschlecht = g;
    : eigenprofil( a, gr, v )
}
```

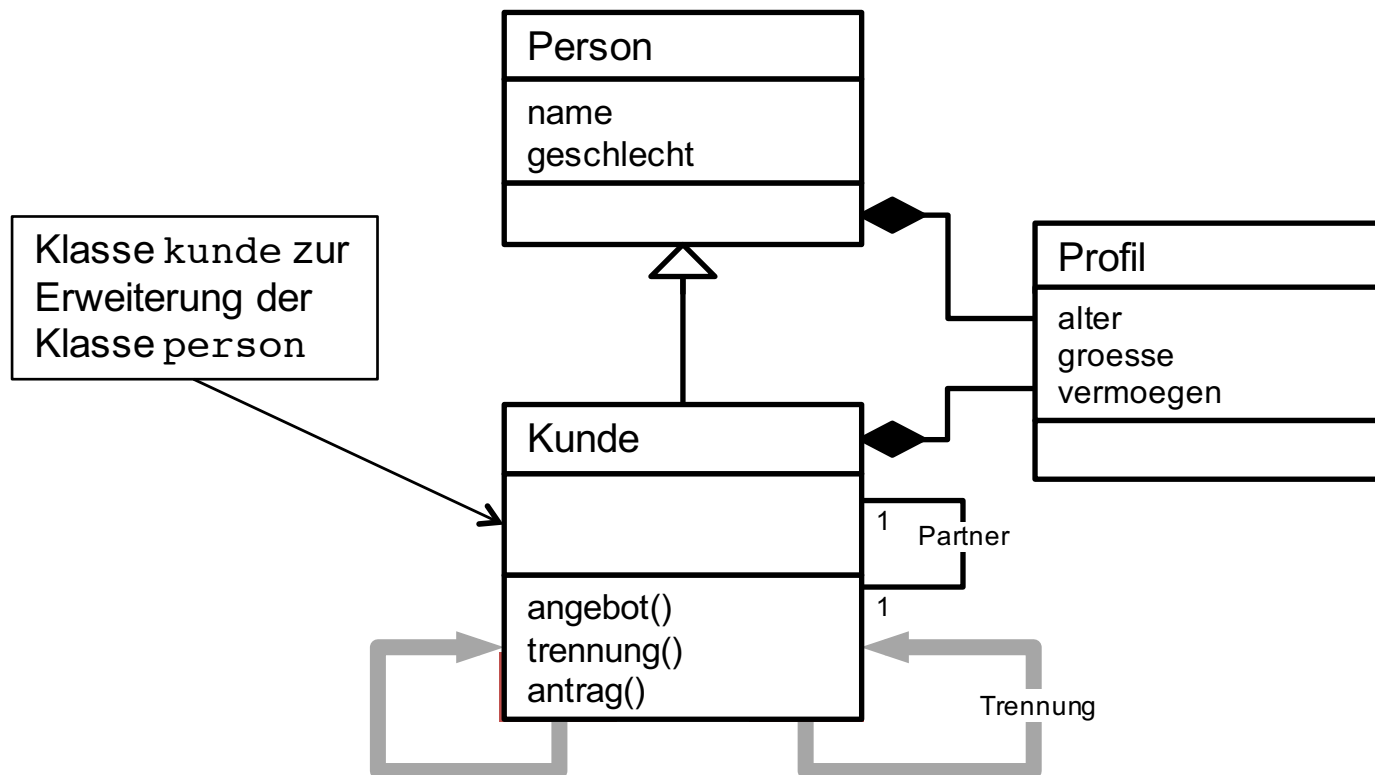
Kopieren des Namens und Zuweisen des Geschlechtes

Übergabe der Parameter aus dem Konstruktor

Im Konstruktor übergeben wir die Parameter für Namen und Geschlecht sowie Alter, Größe und Vermögen. Name und Geschlecht kopieren wir in die Attribute der Person, die Daten zu Alter, Größe und Vermögen reichen wir an das Eigenprofil der Person weiter.

Die Klasse kunde

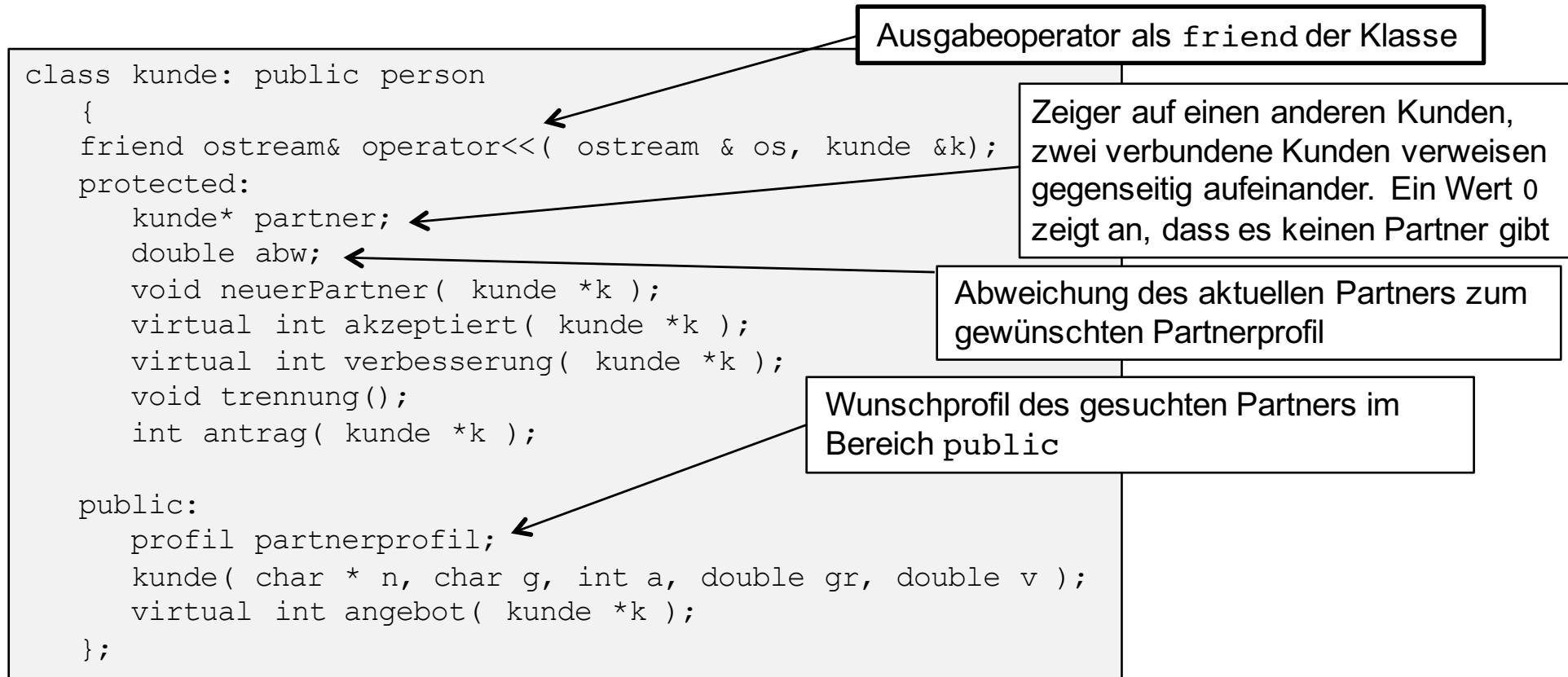
Eine Person hat noch kein bestimmtes Verhalten, außer dass sie über Namen und Geschlecht informieren kann. Das konkrete Verhalten ergibt sich erst, wenn wir die Person zu einem Kunden verfeinern. In der Klasse kunde finden wir einen Großteil der Funktionalität unseres Programms. Das Klassendesign gibt hierzu bereits Hinweise.



Wir werden die Verfeinerungen nun in der von `person` abgeleiteten Klasse `kunde` umsetzen.

Die Deklaration der Klasse kunde

In der Klasse kunde übertragen wir die Elemente des Klassendesigns in Code:



Im protected Bereich der Klasse sind die Elemente umgesetzt, die von außen unzugänglich bleiben, für Kinder der Klasse kunde aber erreichbar sein sollen. Über die Methoden in diesem Bereich erfolgt die interne Steuerung der Klasse.

Alle Methoden die in abgeleiteten Klassen angepasst werden, sind in der Basisklasse kunde bereits als virtual gekennzeichnet. Hierbei handelt es sich um die Methoden akzeptiert, verbesserung und neuerPartner.

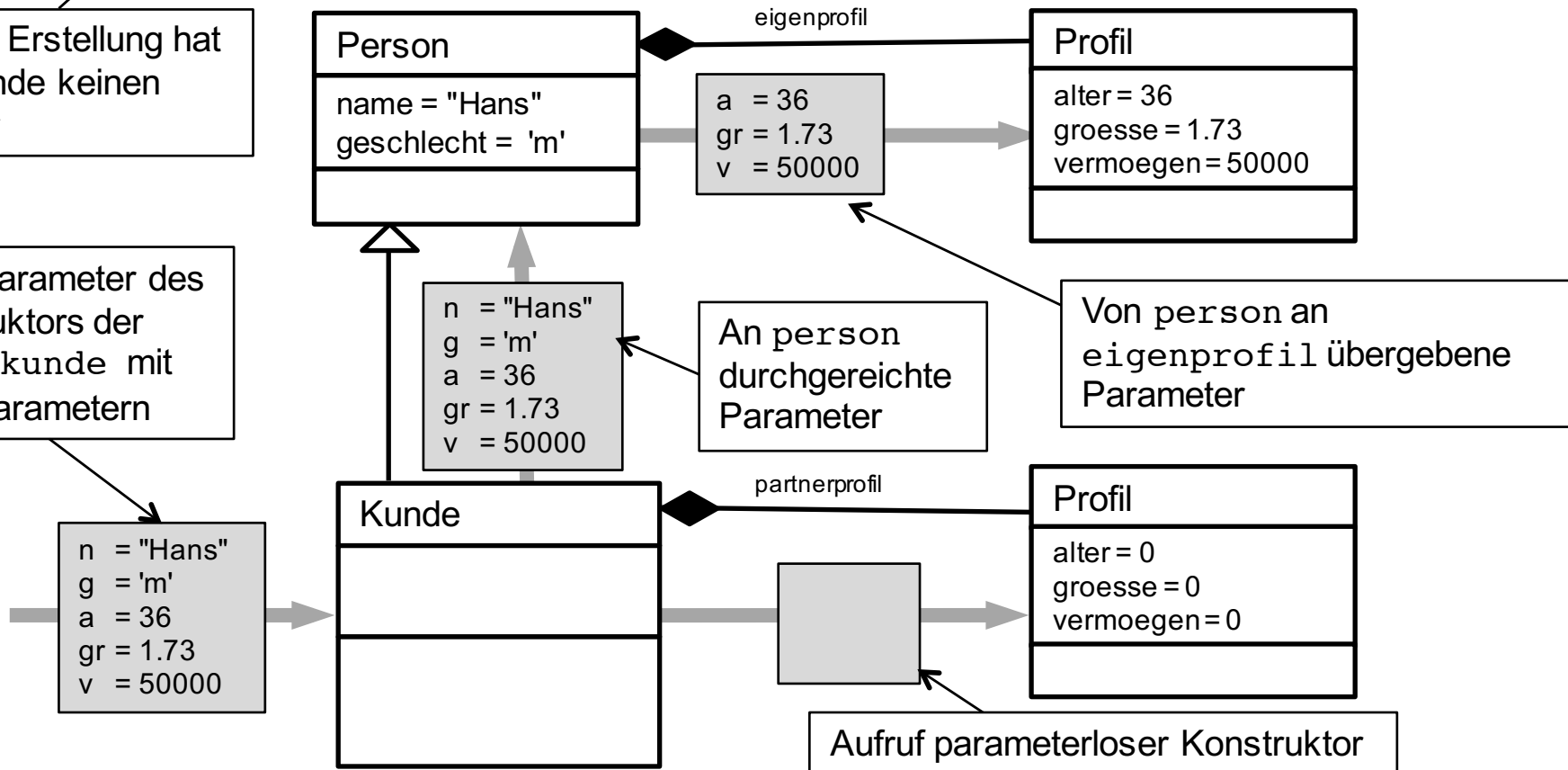
Der Konstruktor der Klasse kunde

Ein Kunde ist eine Person und enthält ein Profil. Bei der Instanziierung müssen die Basisklasse person und das enthaltene Objekt partnerprofil korrekt initialisiert werden. Der Ablauf ist im folgenden dargestellt:

```
kunde::kunde( char *n, char g, int a, double gr, double v )  
    : person( n, g, a, gr, v ), partnerprofil()  
{  
    partner = 0;  
}
```

Bei der Erstellung hat der Kunde keinen Partner

Aufrufparameter des Konstruktors der Klasse kunde mit allen Parametern



Das Vorgehen bei der Partnerwahl

Bei der Partnerwahl prüft ein Kunde zuerst, ob ein vorgeschlagener Partner prinzipiell in Frage kommt und akzeptiert werden kann:

Die Methode ist virtuell in der Basisklasse Person

```
int kunde::akzeptiert( kunde *k )  
{  
    return abweichung( partnerprofil, k->eigenprofil ) < 0.25;  
}
```

Ein vorgeschlagener Partner wird generell akzeptiert, wenn sein Profil weniger als 25% vom Wunschprofil abweicht.

Mit der Methode `verbesserung` prüft ein Kunde, ob die Entscheidung zur Bindung mit einem angebotenen Kunden zu einer Verbesserung seiner Situation führen würde.

virtual in person

```
int kunde::verbesserung( kunde *k )  
{  
    if( !akzeptiert( k ) )  
        return 0;  
    if( !partner )  
        return 1;  
    if( abweichung( partnerprofil, k->eigenprofil ) < abw )  
        return 1;  
    return 0;  
}
```

Wenn der Partner nicht akzeptabel ist, ist das keine Verbesserung, es erfolgt auf keinen Fall eine Bindung

Wenn der Partner akzeptabel ist und der Kunde selbst noch keinen Partner hat, ist das eine Verbesserung

Gibt es bereits einen Partner, wird geprüft, ob die Profilabweichung mit dem neuen Partner geringer wäre

Keine Verbesserung

Die Art der Prüfung führt dazu, dass ein Kunde eine Verbindung nicht eingeht, wenn das Profil des angebotenen Partners zu stark vom Wunschprofil abweicht, selbst wenn er noch keinen Partner hat.

Verbindung mit einem neuen Partner

Die Klasse kunde wickelt mit der Methode `neuerPartner` die Verbindung zu einem neuen Partner ab. Dies beinhaltet die Trennung von einem eventuell vorhandene Partner:

```
void kunde::neuerPartner( kunde *k )
{
    if( partner )
        partner->trennung();

    partner = k;
    abw = abweichung( partnerprofil, k->eigenprofil );
}
```

Wenn ein Kunde einen neuen Partner annehmen möchte und bereits einen Partner hat, schickt er diesem die Botschaft, dass er eine Trennung vollzieht

Der neue Partner wird gespeichert

Die neue Abweichung zum Wunschprofil wird festgehalten

Die Methoden `akzeptiert`, `verbesserung` und `neuerPartner` liegen im geschützten Bereich der Klasse. Die Methode `trennung` ist öffentlich und kann auch von außen aufgerufen werden.

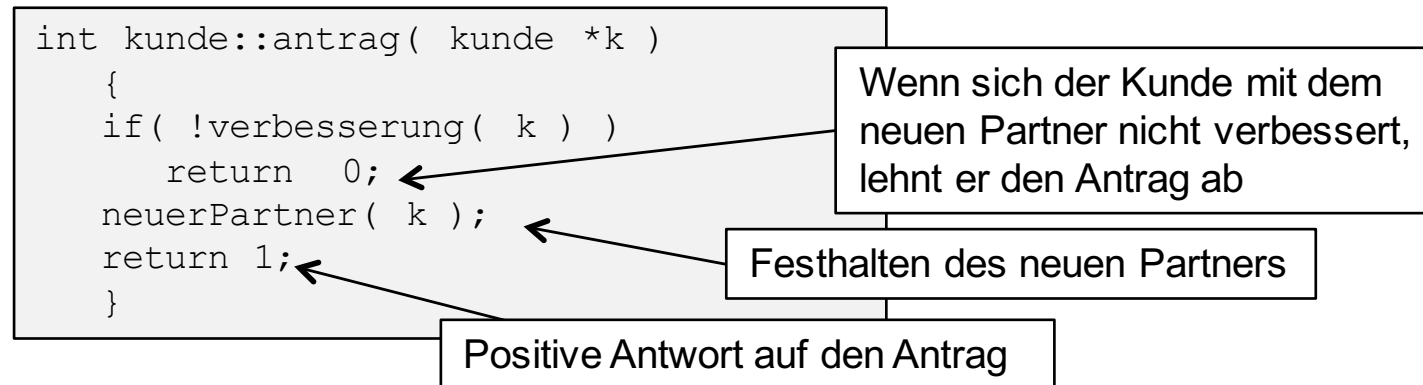
```
void kunde::trennung()
{
    cout << "Trennung: " << getName() << " >< " << partner->getName() << '\n';
    partner = 0;
}
```

Kunde hat keinen Partner mehr

Der Betroffenen erhält die Botschaft zur Trennung und hat keine Möglichkeit zu reagieren, außer die Trennung bekanntzugeben und die Verbindung zum bisherigen Partner zu löschen.

Antrag an einen neuen Partner stellen

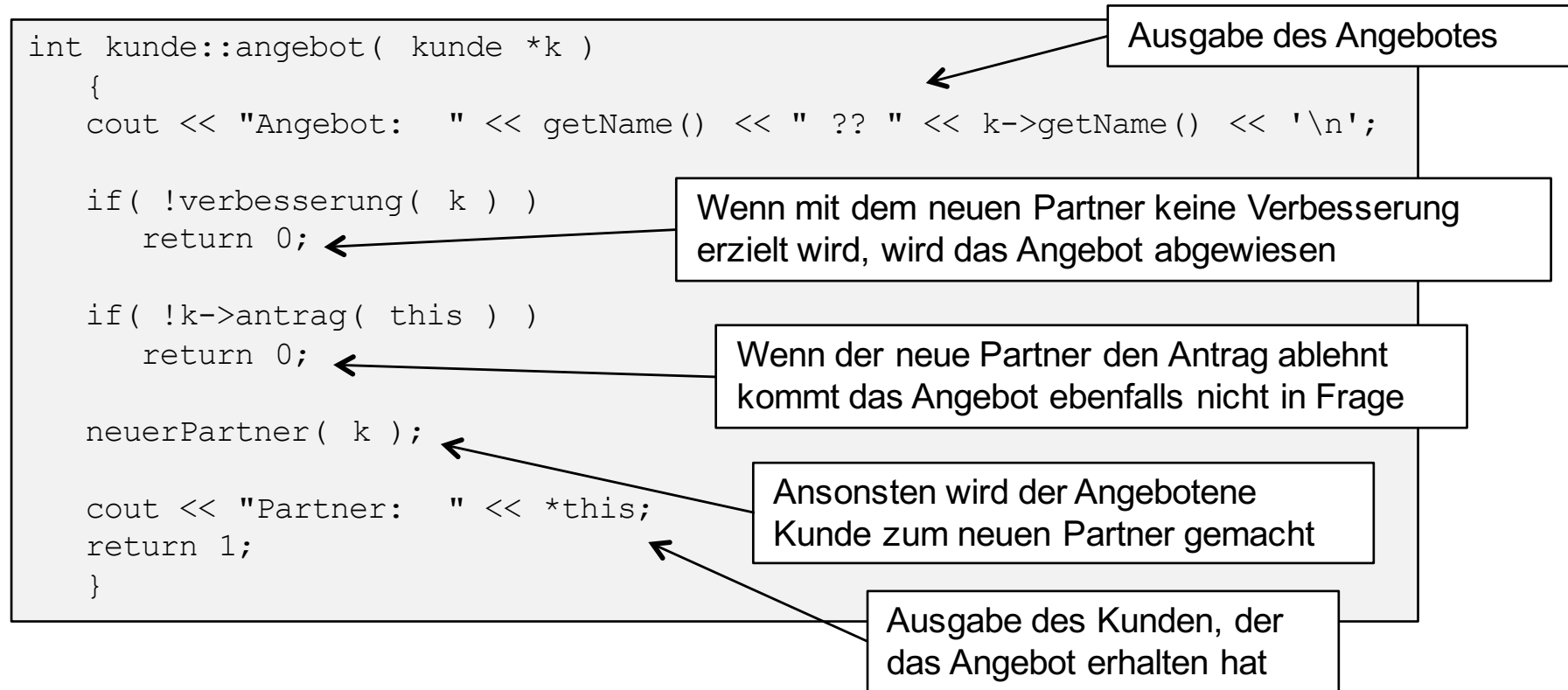
Über die öffentliche Methode `antrag` nimmt einen Kunde den Antrag eines anderen partnersuchenden Kunden entgegen:



Auch der Antragsempfänger prüft zuerst, ob sich seine Situation mit dem neuen Partner verbessern würde. Ist das nicht der Fall, lehnt er den Antrag ab. Ansonsten nimmt der den Antrag an und hält den neuen Partner direkt fest, womit er sich von einem gegebenenfalls vorhanden Partner trennt. Er vertraut dabei darauf, dass der Antragssteller ihn auch nimmt. Dieser hat vor seinem Antrag ja auch bereits geprüft, dass der potentielle neue Partner eine Verbesserung darstellt.

Angebote der Partnervermittlung

Die noch zu implementierende Partnervermittlung wird den Kunden über deren öffentliche Schnittstelle andere Partnersuchende anbieten indem sie entsprechende Nachrichten sendet. Dazu hat die Klasse kunde die passende Methode `angebot`:



Wenn sich der Kunde mit dem neuen Partner verbessert und dieser den Antrag angenommen hat, wird der zum neuen Partner erklärt. Der angebotene Partner der den Antrag akzeptiert hat diesen Schritt dann schon vollzogen. Die Klasse `kunde` ist nun fast vollständig implementiert, wir müssen nun nur noch den Ausgabeoperator erstellen, den wir in der Methode bereits verwenden.

Der Ausgabeoperator

Um einen Kunden möglichst einfach auf dem Bildschirm ausgeben zu können, müssen wir noch die Überladung des Ausgabeoperators implementieren, die wir in der Methode `angebot` schon verwendet haben. Wir haben den Operator in der Klasse `kunde` als `friend` deklariert, daher kann er auf alle Elemente der Klasse zugreifen:

```
ostream& operator << ( ostream &os, kunde &k )
{
    os << k.getName();
    os << " == ";
    if( k.partner )
        os << k.partner->getName();
    else
        os << '-';
    os << '\n';
    return os;
}
```

Ausgabe des Namens des Kunden

Ausgabe des Namen des Partners,
falls vorhanden

Ansonsten Ausgabe eines '-'

Es ergeben sich damit zwei Ausgabemöglichkeiten:

Anton == Berta

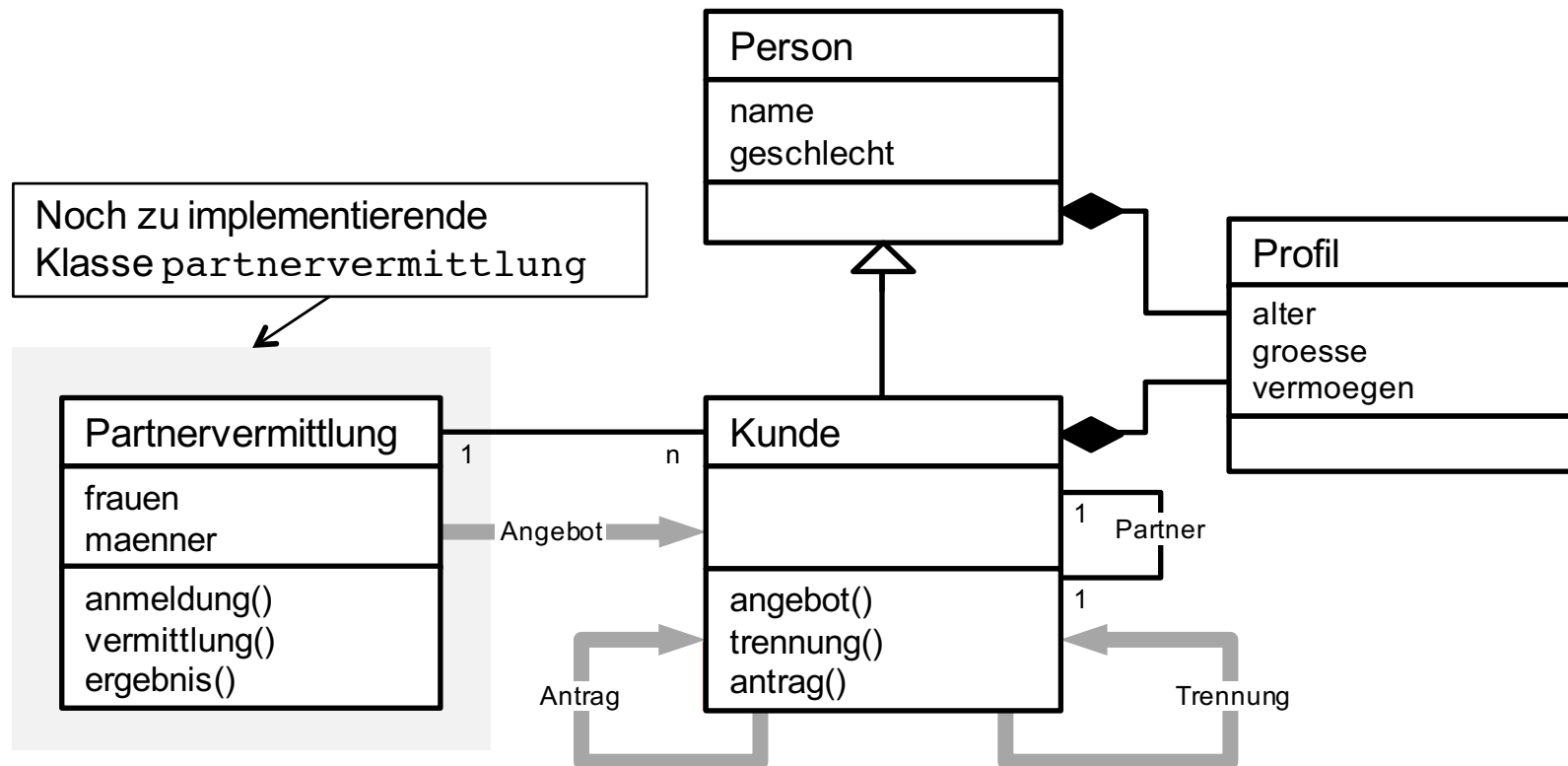
Der Kunde, hier Anton, hat einen
Partner, in diesem Fall Berta

Anton == -

Der Kunde hat aktuell keinen Partner

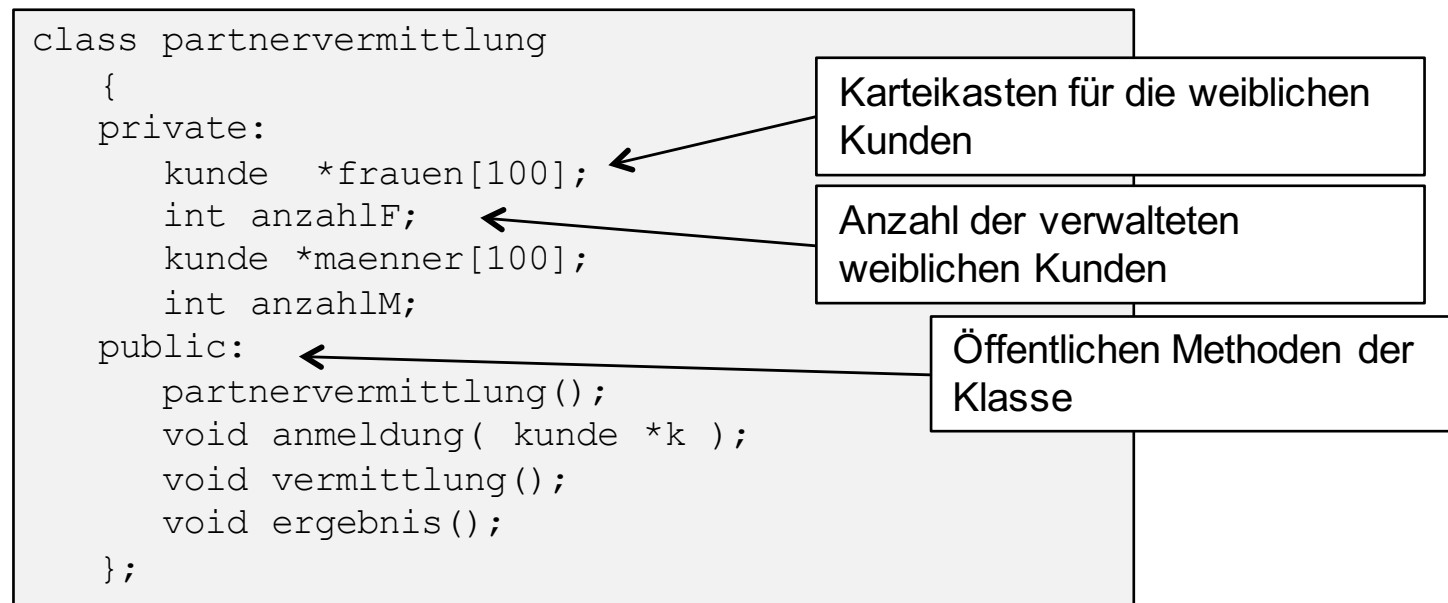
Ermittlung noch fehlender Elemente

Ein Blick auf das Klassendesign zeigt uns, dass die Implementierung der Partnervermittlung selbst noch offen ist:



Die Klasse **partnervermittlung**

Die Partnervermittlung verwaltet jeweils einen Karteikasten für die weiblichen und männlichen Kunden und hält die Anzahl der verwalteten Karteikarten nach:



Neben dem Konstruktor sind nur die öffentlichen Methoden für die Neuanmeldung eines Kunden, den Start der Vermittlung selbst und die Ausgabe des aktuellen Vermittlungsstandes für alle Kunden enthalten.

Konstruktor der Partnervermittlung

Der Konstruktor und die Neuanmeldung eines Kunden gestalten sich relativ einfach:

```
partnervermittlung::partnervermittlung()  
{  
    anzahlF = 0;  
    anzahlM = 0;  
}
```

Die Anzahl der vorhandenen
Karteikarten wird jeweils auf 0
gesetzt

```
void partnervermittlung::anmeldung( kunde *k )  
{  
    if( k->getGeschlecht() == 'm' )  
        maenner[anzahlM++] = k;  
    else  
        frauen[anzahlF++] = k;  
}
```

Ein neuangemeldeter Herr wird der
Kartei der Männer hinzugefügt

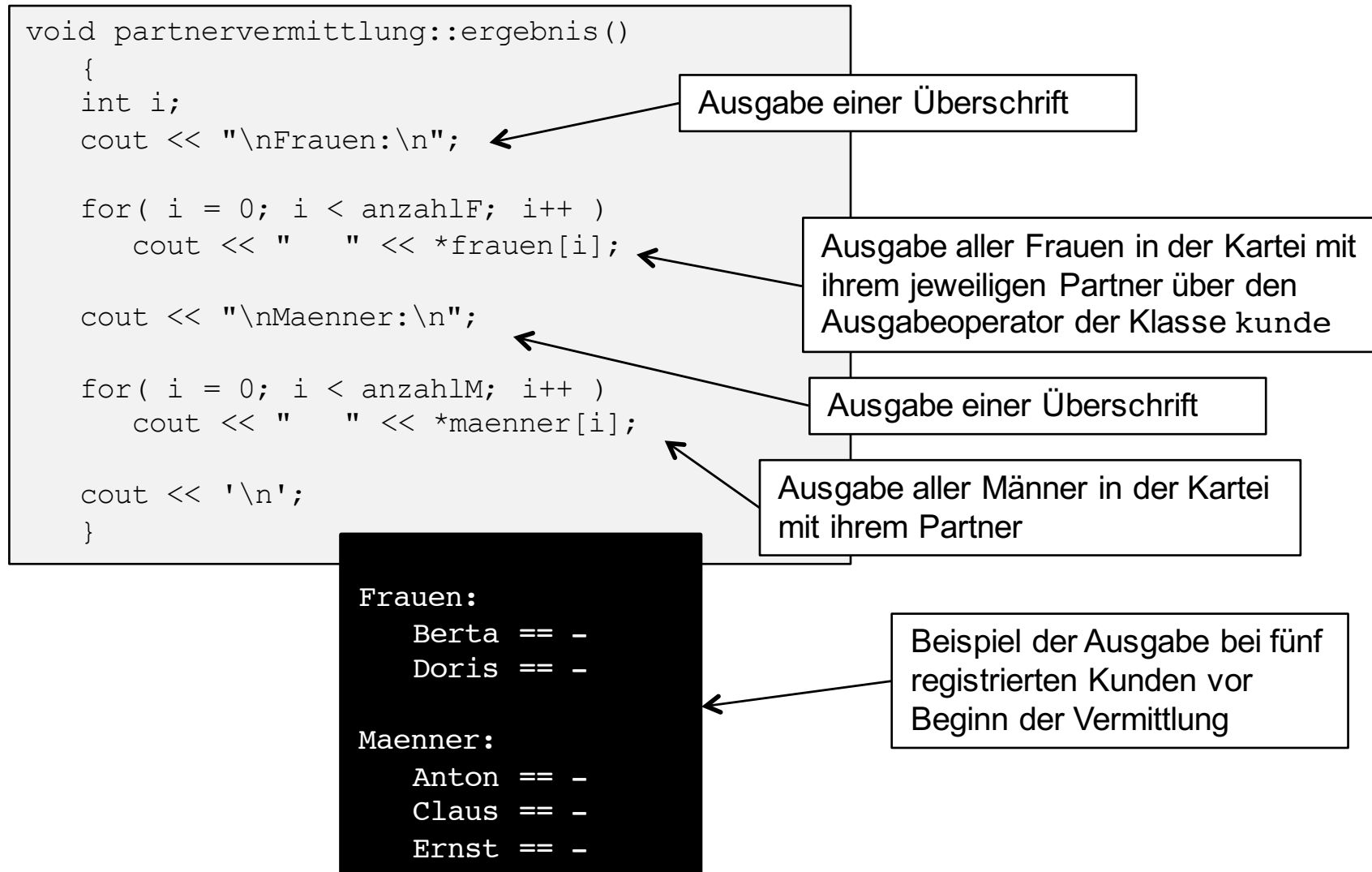
Ein weiblicher Kunde wird der Datei
der Frauen hinzugefügt

Inkrementieren des Zählers mit dem Postfix-
Operator in Verbindung mit der Zuweisung

Wir verzichten in unserem Beispiel auf die Prüfung der Anzahl der bereits vorhandenen Kunden im jeweiligen Karteikasten, die sonst zu einer Ablehnung eines neuen Kunden oder zur Erweiterung eines bereits vollen Karteikastens führen müsste.

Ausgabe des aktuellen Vermittlungszustandes

Die Ausgabe des aktuellen Vermittlungszustandes der Partnervermittlung erfolgt mit einer entsprechenden Methode unter Verwendung des bereits implementierten Ausgabeoperators der Klasse kunde:



Durchführung der Vermittlung

Bei der Durchführung der eigentlichen Partnervermittlung macht es sich die Agentur sehr einfach:

```
void partnervermittlung::vermittlung()  
{  
    int i, j;  
  
    for( i = 0; i < anzahlM; i++ )  
    {  
        for( j = 0; j < anzahlF; j++ )  
            maenner[i]->angebot( frauen[j] );  
    }  
  
    for( i = 0; i < anzahlF; i++ )  
    {  
        for( j = 0; j < anzahlM; j++ )  
            frauen[i]->angebot( maenner[j] );  
    }  
}
```

Schleife über alle Männer in
der Kartei

Schleife über alle Frauen in
der Kartei

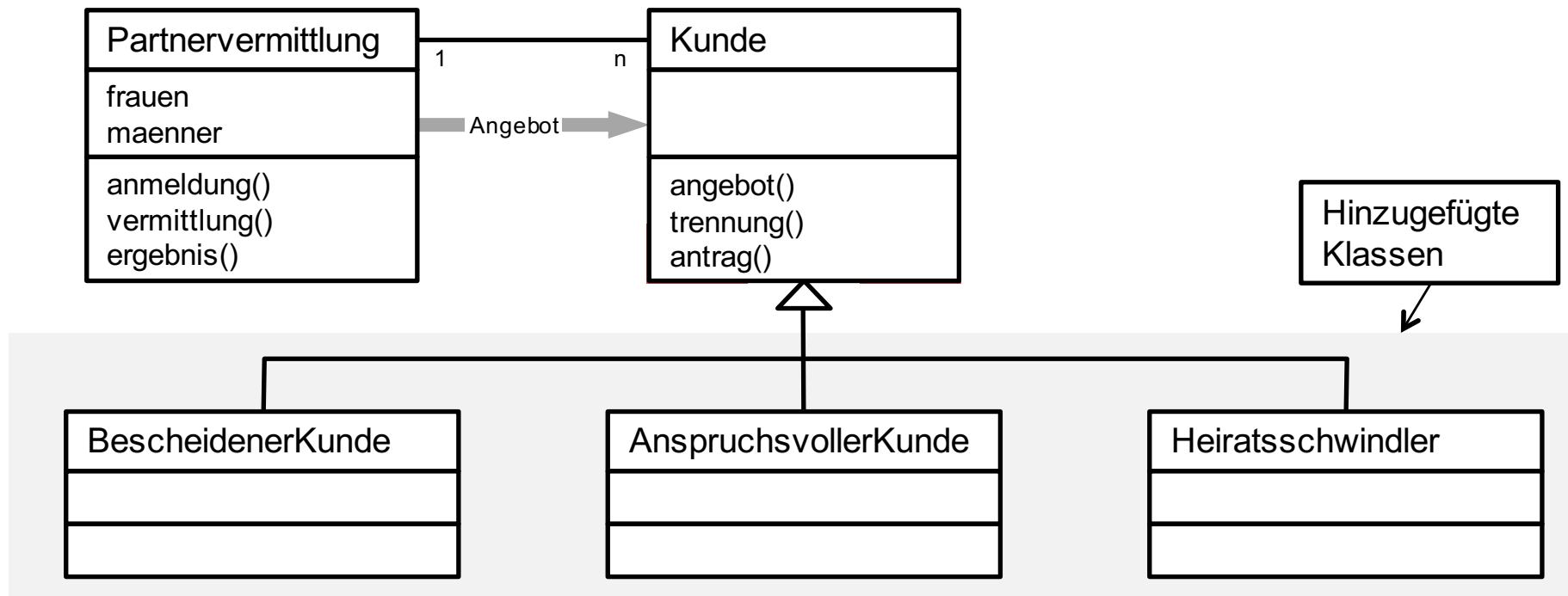
Jedem Mann alle
Frauen anbieten

Jeder Frau alle Männer
anbieten

Die Agentur prüft die Profile und Wünsche der Kunden gar nicht, sondern bietet einfach der Reihe nach allen Männern alle Frauen an und umgekehrt. Die eigentliche Arbeit erledigen dann die Kunden selbst mit ihren implementierten Methoden.

Die spezialisierten Kunden

Generell ist unsere Partnervermittlung nun arbeitsfähig. Nach Erstellung eines Hauptprogramms, dass Kunden instanziiert, könnte die Vermittlung starten. Allerdings haben aktuell alle Kunden die gleiche Strategie bei der Auswahl ihres Partners. Bevor wir die Vermittlung beginnen, wollen wir daher jetzt noch besondere Kundentypen wie den anspruchsvollen Kunden, den bescheidenen Kunden und den Heiratsschwindler hinzufügen.



Dabei ist hervorzuheben, dass wir die neuen Kundentypen und deren Klassen hinzufügen können, ohne die restlichen Implementierungen ändern zu müssen. Die Partnervermittlung kennt die noch zu erstellenden Kundentypen gar nicht. Sie behandelt nur Kunden und weiß gar nichts von der Vererbung.

Der anspruchsvolle Kunde

Der anspruchsvolle Kunde verhält sich im Konstruktor nicht anders als ein gewöhnlicher Kunde:

```
class anspruchsvollerKunde: public kunde
{
private:
    int akzeptiert( kunde *k );
public:
    anspruchsvollerKunde( char *n, char g, int a, double gr, double v )
        : kunde( n, g, a, gr, v ) {}
};
```

Die Klasse leitet sich ab von
der Basisklasse kunde

Kein Code im
Konstruktor

Weiterleitung der Parameter an den
Konstruktor der Basisklasse

Die Methode, die über die Akzeptanz eines vorgeschlagenen Partners entscheidet, ist allerdings angepasst. Diese Methode ist in der Basisklasse bereits als `virtual` deklariert:

```
int anspruchsvollerKunde::akzeptiert( kunde *k )
{
    return abweichung( partnerprofil, k->eigenprofil ) < 0.10;
}
```

Bei Profilabweichung
unter 10% akzeptiert

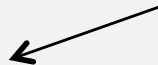
Bei der Akzeptanz eines vorgeschlagenen Partners ist der anspruchsvolle Kunde wählerischer. Während ein gewöhnlicher Kunde bei einem Vorschlag bereits eine Abweichung von weniger als 25% zu seinem Wunschprofil akzeptiert, muss für den anspruchsvollen Kunden die Abweichung unter 10% liegen.

Der bescheidene Kunde

Der bescheidene Kunde akzeptiert jedes Angebot, egal welches Profil der Partner hat, er schaut sich das Angebot nicht einmal an:

```
class bescheidenerKunde: public kunde
{
private:
    int akzeptiert( kunde *k ) { return 1; }
public:
    bescheidenerKunde( char *n, char g, int a, double gr, double v )
        : kunde( n, g, a, gr, v ) {}
};
```

Angebote werden ohne
Prüfung akzeptiert



Weiterleitung der Parameter an den
Konstruktor der Basisklasse



Konstruktor ohne Code



Der Heiratsschwindler

Anders als die anderen Kunden hat der Heiratsschwindler ein ganz anderes Vorgehen bei der Akzeptanz, Verbesserung von Vorschlägen und dem Umgang mit den Angeboten der Agentur:

```
class heiratsschwindler: public kunde
{
private:
    int akzeptiert( kunde *k );
    int verbesserung( kunde *k );

public:
    heiratsschwindler( char *n, char g, int a, double gr, double v )
        : kunde( n, g, a, gr, v ) {}

    int anbot( kunde *k );
};
```

Konstruktor wie bei den anderen
Kindklassen von kunde

Eigene Methoden akzeptiert, verbesserung und
anbot, alle bereits als virtual in der Basisklasse
deklariert

```
int heiratsschwindler::akzeptiert( kunde *k )
{
    return k->eigenprofil.vermoegen > 50000.00;
}
```

Akzeptanzkriterium ist ein
Vermögen > 50000


Die Strategie des Heiratsschwindlers zur Akzeptanz eines Angebotes ist sehr einfach, er schaut nur auf das Geld und akzeptiert ausschließlich Partner mit einem Vermögen über 50000 EUR. Alle anderen Profileigenschaften sind ihm egal.

Die Verbesserung aus der Sicht des Heiratsschwindlers

Da der Heiratsschwindler nur am Geld interessiert ist, stellt für ihn jeder Partner, der ein höheres Vermögen hat als sein bisheriger Partner, eine Verbesserung dar:

```
int heiratsschwindler::verbesserung( kunde *k )
{
    if( !akzeptiert( k ) )
        return 0;
    if( !partner )
        return 1;
    return k->eigenprofil.vermoegen > partner->eigenprofil.vermoegen;
}
```

Wenn der vorgeschlagene Partner ein höheres Vermögen hat als der aktuelle, ist das für den Heiratsschwindler immer eine Verbesserung



Bis hierhin kann man das Verhalten des Heiratsschwindlers noch akzeptabel nennen. Auf das Vermögen des zukünftigen Partners zu schauen, ist ja kein Verbrechen, wenn auch etwas eindimensional. Die kriminellen Absichten zeigen sich darin, wie der Heiratsschwindler mit den Angeboten an ihn selbst umgeht.

Der Heiratsschwindler und seine Masche

Wenn der Heiratsschwindler ein Angebot erhält, dann verstellt er sich und passt seine Angaben zu Alter und Vermögen an die Wünsche des suchenden Partners an und erhofft sich damit bessere Chancen:

```
int heiratsschwindler::angebot( kunde *k )
{
    eigenprofil.set( k->partnerprofil.alter,
                    eigenprofil.groesse,
                    k->partnerprofil.vermoegen );

    return kunde::angebot(k );
}
```

Alter und Vermögen werden aus dem Wunschprofil des suchenden Partners ausgelesen und in das eigene Profil übernommen

Nach der Verstellung verläuft die weitere Prüfung des Angebotes wie bei anderen Kunden auch.

Der Heiratsschwindler lügt bei Alter und Vermögen, nur bei der Größe sagt er die Wahrheit, vermutlich aus Sorge, dass ein Betrug hier zu offensichtlich wäre.

Nachdem er sich verstellt hat, kann er die Methode `angebot` der Basisklasse verwenden, wie die anderen Kunden auch. Durch seine eigene und modifizierte Methode `akzeptiert` ist dafür gesorgt, dass nur die wohlhabenden Kunden berücksichtigt werden.

Das Hauptprogramm

Im Hauptprogramm müssen nun nur noch die notwendigen Objekte instanziiert und der Vermittlungsprozess gestartet werden:

```
int main()
{
    partnervermittlung pv;

    kunde anton( "Anton", 'm', 55, 1.75, 100000 );
    anton.partnerprofil.set( 50, 1.70, 0 );
    pv.anmeldung( &anton );

    kunde berta( "Berta", 'w', 50, 1.70, 60000 );
    berta.partnerprofil.set( 50, 1.80, 10000 );
    pv.anmeldung( &berta );

    heiratsschwindler claus( "Claus", 'm', 30, 1.80, 10000 );
    claus.partnerprofil.set( 25, 1.70, 0 );
    pv.anmeldung( &claus );

    anspruchsvollerKunde doris( "Doris", 'w', 60, 1.65, 100000 );
    doris.partnerprofil.set( 65, 1.80, 10000 );
    pv.anmeldung( &doris );

    bescheidenerKunde ernst( "Ernst", 'm', 50, 1.80, 8000 );
    ernst.partnerprofil.set( 50, 1.80, 20000 );
    pv.anmeldung( &ernst );

    pv.ergebnis();
    pv.vermittlung();
    pv.ergebnis();
}
```

Instanziierung der partnervermittlung

Instanziierung des gewöhnlichen Kunden
„Anton“ und Setzen seines Eigenprofils

Anmeldung des Kunden bei der Vermittlung

Des Heiratsschwindlers Claus wird
erstellt und angemeldet

Weitere Kunden werden erstellt und
hinzugefügt

Der initiale Vermittlungsstand wird ausgegeben

Die Vermittlung wird ausgeführt

Der endgültige Stand nach der Vermittlung wird ausgegeben


Der Ablauf des Hauptprogrammes

Wir können unser Hauptprogramm nun starten und uns dessen Ausgabe ansehen:

```
Frauen:  
  Berta == -  
  Doris == -
```

```
Maenner:  
  Anton == -  
  Claus == -  
  Ernst == -
```

An Anfang sind insgesamt fünf Personen
in der Kartei, alle noch ohne Partner,
dann startet die Vermittlung



Das Programm startet mit der Ausgabe der Vermittlungssituation. In unserem System sind zwei Frauen und drei Männer registriert. Alle sind zum Start der Vermittlung ohne Partner.

Nach der Ausgabe beginnt die eigentliche Vermittlung:

Vermittlung der Frauen an die Männer

Die Vermittlung startet, indem allen Männer alle Frauen als Partner angeboten werden:

```
Angebot: Anton ?? Berta  
Partner: Anton == Berta  
Angebot: Anton ?? Doris
```

Die Vermittlung beginnt mit Anton und Berta, die sich auch prompt verpartnern. Eine Verbindung zwischen Anton und Doris kommt nicht zustande, vermutlich ist Doris für Anton zu alt

```
Angebot: Claus ?? Berta  
Trennung: Anton >< Berta  
Partner: Claus == Berta
```

Der Heiratsschwindler Claus greift ein, ist an der vermögenden Berta (60000) interessiert und spannt sie Anton aus

```
Angebot: Claus ?? Doris  
Trennung: Berta >< Claus  
Partner: Claus == Doris
```

Wenn Claus auf die 100000 Euro schwere Doris trifft, verbindet er sich unter falschen Angaben mit ihr und lässt Berta fallen

```
Angebot: Ernst ?? Berta  
Partner: Ernst == Berta  
Angebot: Ernst ?? Doris
```

Jetzt ist Ernst an der Reihe. Er liiert sich mit Berta, die von Claus abserviert worden war, eine Verbindung zwischen Ernst und Doris kommt nicht zustande, Doris lässt ihren „Traummann“ auch im folgenden nicht mehr los

Fortsetzung der Vermittlung

Die erste Runde der Vermittlung ist abgeschlossen, nun werden allen Frauen alle Männer angeboten:

```
Angebot: Berta ?? Anton
Trennung: Ernst >< Berta
Partner: Berta == Anton

Angebot: Berta ?? Claus
Angebot: Berta ?? Ernst
```

Berta wird Anton vorgestellt und gibt für ihn dem Ernst den Laufpass. Vermutlich ist Ernst für Berta nicht vermögend genug. Die weiteren Angebote nimmt Berta nicht wahr

```
Angebot: Doris ?? Anton
Angebot: Doris ?? Claus
Angebot: Doris ?? Ernst
```

Weitere Vermittlungsversuche bei Doris ergeben keine Änderungen mehr, sie bleibt bei Claus

```
Frauen:
  Berta == Anton
  Doris == Claus
```

```
Maenner:
  Anton == Berta
  Claus == Doris
  Ernst == -
```

Damit ist die Vermittlung beendet und es hat sich folgendes Ergebnis eingestellt:
Der Heiratsschwindler hat sich die reiche Dame geangelt, die als anspruchsvolle Kundin auf den Betrüger hereingefallen ist. Anton und Berta haben nach Wirrungen zueinander gefunden und der arme aber anspruchslose Ernst geht leer aus