NewElements IT-Schulungen.com

Objektorientierte Programmierung Grundkurs (OOP)

Dr.sc.nat. Michael J.M. Wagner*

Revision 1.83

*michael@wagnertech.de

Inhaltsverzeichnis

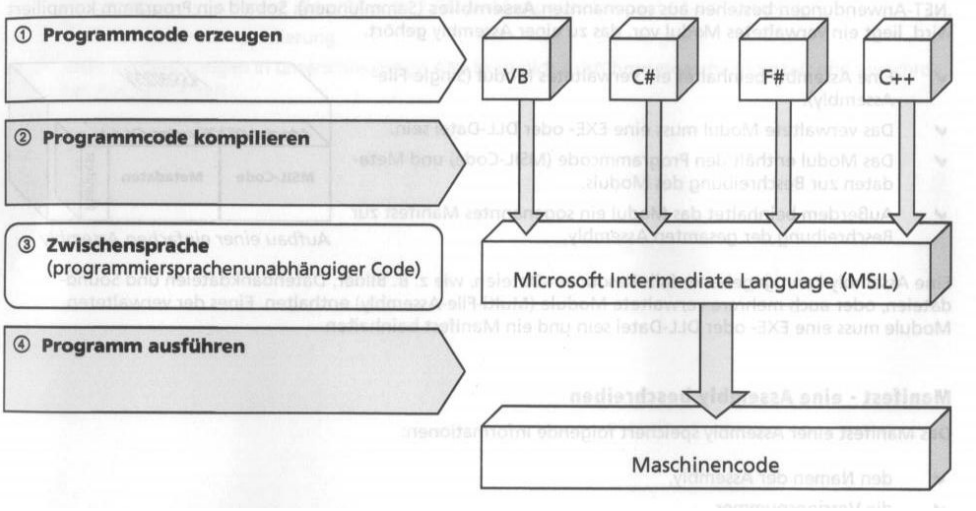


Abbildung 1: Programme gemäß .NET entwickeln[3]

1 Einführung

1.1 C# und .NET[1]

C#

- C# ist eine objektorientierte Programmiersprache von Microsoft
- „Das bessere Java“
- C# läuft auf .NET.

.NET

> .NET ... beschreibt eine Softwareumgebung zur Entwicklung *programmiersprachen- und plattformunabhängiger* Software, in dem Programmcode verschiedener Programmiersprachen mittels eines Compilers in eine Zwischensprache übersetzt wird, die für alle Programmiersprachen gleich ist. Diese Zwischensprache ist plattformneutral und kann auf unterschiedlichen Betriebssystemen wie Windows oder Linux ausgeführt werden.[2]

.NET besteht aus

- umfangreicher Klassenbibliothek, programmierspachenunabhängig
- *common language runtime* (CLR)
- *just in time compiler* (JIT)

Assembly

- Produkt des Compilers (exe oder dll)

[1] VC2015:Kap. 2
[2] VC2012: S. 7
[3] VC2012: S. 7.

- *Microsoft intermediate language* (MSIL)
- benötigt CLR um ausgeführt werden zu können
- enthält Meta-Information

Bei der Ausführung eines Assemply übersetzt der JIT der .NET-Plattform die MSIL in Maschinencode (S. Abb. 1).

1.2 Visual Studio

Visual Studio

- Microsoft IDE
- Weitere IDEs:
  - Eclipse: Für viele Programmiersprachen, Java-basiert
  - VSCode: Microsoft, offen für Programmiersprachen, Plattformen. Riesige Community, die Plugins für die verschiedesten Einsatzszenarien erstellt.
- Visual Studio und Eclipse sind in der Bedienung ähnlich, VSCode ist ein besserer Editor und als Entwicklungsumgebung etwas gewöhnungsbedürfig.

Strukturierung des Codes

- Projektmappe/Solution, Workspace
- Projekt $\rightarrow$ Programm oder Bibliothek

Strukturierung der IDE

- Die Fensteranordnung kann individuell angepasst und abgespeichert werden.

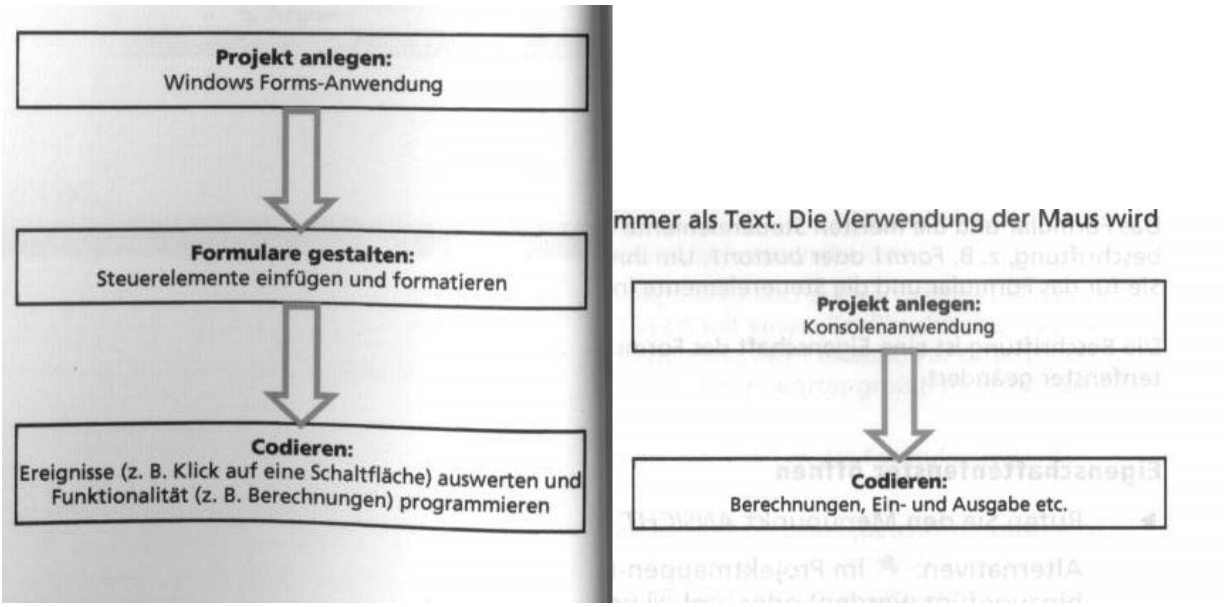
2 Sprachgrundlagen

2.1 Anwendungen erstellen[4]

Die einfachsten Formen einer Anwendung sind:

- WindowsForms-Anwendung (Windows)
- Konsolen-Anwendung

[4] VC2015: Kap. 4



(a) GUI-Anwendung (b) Konsolenanwendung

Abbildung 2: Anwendungserstellung

Konsolenanwendung

Konsolenanwendungen sind nützlich für:

- Alternativer Zugang zum Anwendungskern
- Testskripte
- Automatisierung

> Aufgabe:
>
> Erstellen Sie mit Ihrer Softwareentwicklungsumgebung eine "Hallo Welt" - Konsolenanwendung.

2.2 Sprachelemente

Bezeichner und Schlüsselwörter[5]

- Regeln wie in C/C++/Java
- Schlüsselwörter [VC:6.2]

[5] VC2015: Kap. 6.2

Aufbau eines Programms[6]

Jede Anwendung benötigt ein Hauptprogramm: Irgendeine Klasse mit einer `Main(string[] args)` - Methode. Mit .netcore ist es nun auch möglich, C# wie eine Skriptsprache zu verwenden: Hauptprogramm ist derjenige Code, der ohne weitere Struktur „ganz links" steht.

Programmcode dokumentieren[7]

`//` : Zeilenkommentar

`/* ... */` : Blockkommentar

`///` : Dokumentation, die extrahiert werden kann

Anweisungen in Visual C# erstellen

Anweisungen in Visual C# erstellen[8]

- Anweisungen werden mit ";" abgeschlossen
- Anweisungen werden mit `{ ... }` zu Blöcken zusammengefasst

Einfache Datentypen[9]

(1) Numerische Datentypen

Integer-Datentypen [VC:6.6]

Gleitkomma-Datentypen [VC:6.6a]

(2) Zeichenketten-Datentypen [VC:6.6b]

(3) Boolescher (logischer) Datentyp [VC:6.6c]

Literale[10]

- Zahlen
- Escape-Sequenzen werden in Zeichenketten mit "\" eingeleitet.
- Für Windows-Pfade gibt es Strings ohne „Escape": `@"c:\Users\Kurs"`

[6] VC2015: Kap. 6.3
[7] VC2015: Kap. 6.4
[8] VC2015: Kap. 6.5
[9] VC2015: Kap. 6.6]
[10] VC2015: Kap. 6.7

Mit Variablen arbeiten[11]

- Sichtbarkeit im Block (wie in C/C++/Java)
- Implizite Typisierung mit `var` [VC:6.8]

Typkompatibilität und Typkonversion[12]

- Implizit: `double d = 2;`
- Explizit mit `Convert` / `ToString()` / `Parse()` → Abb. 3

  Zur Beachtung: `Convert` und `Parse()` sind *prozedurale* Aufrufe, `ToString()` hingegen ist *objektorientiert*. Im ersten Fall steht vor dem Punkt das Modul, das die Prozedur enthält, im zweiten die Variable, die den Wert enthält.

Beispiele für Typkonversionen: *TypeCast.sln*

```
    int number = 4567;
    double size = 123.12;
    char ch = 'K';
    string txt;
①   number = (int)size;
②   size = 149;
③   txt = ch.ToString();
④   txt = number.ToString();
⑤   number = Convert.ToInt32("4567");
⑥   size = Double.Parse("4567,89");
⑦   size = Convert.ToDouble("4567,89");
⑧   ch = Convert.ToChar("k");
```

Abbildung 3: Beispiele für Typumwandlungen[13]

Formatierte Ausgabe: `Console.WriteLine ("{0,5};{1}", s, i);`

Hier wird z.B. die Variable `s` 5 Zeichen breit und die Variable `i` ausgegeben. Mit der Variableninterpolation kann dies auch so formuliert werden:

```
Console.WriteLine ($"{s,5};{i}");
```

Weitere Fotmatierungsanweisungen: [VC2022: S. 59.]

Konstanten[14]

- Konstanten werden mit `const` gekennzeichnet.

[11] VC2015: Kap. 6.8
[12] VC2015: Kap. 6.11
[13] VC2012: S. 66/68.
[14] VC2015: Kap. 6.12

Arithemtische Operatoren und Vorzeichen- und Verkettungsoperatoren[15]

- Arithmetisch: `+ - * / %`
  - Feldüberläufe lassen sich duch eine entsprechende Compilereinstellung oder das Schlüsselword checked überprüfen.
- Vorzeichenoperator: `-`
- Verkettungsoperator: `+`
- Inkrement/Dekrement: `++ --`

Logische Operatoren[16]

- Vergleichsoperatoren: `== != > < >= <=`
- Verknüpfungsoperatoren: `! & && | || ^`

Zuweisungsoperatoren für eine verkürzte Schreibweise verwenden[17]

`a = a + b;` → `a += b;`

Aufgabe:

- Fordern Sie in der Anwendung den Anwender dazu auf, eine ganze Zahl einzugeben, und speichern Sie den Wert in einer Variablen vom Typ `int`.
- Lassen Sie auch einen `double`- und einen `string`-Wert mithilfe der Tastatur eingeben und speichern Sie die Werte ebenfalls in geeigneten Variablen.
- Geben Sie die Werte der drei Variablen abschließend auf dem Bildschirm aus.

Hinweis: Von der Konsole lesen: `string s = Console.ReadLine();`

Namensräume

Damit Klassen keine systemweit eindeutigen Namen haben müssen, wird der Code mittels Namensräumen strukturiert. Überlicherweise wird der Namensraum wie das Assembly genannt. Mit `using` werden Namensräume in den aktuellen Kontext eingebunden.

- Erstellung eines Namensraums [VC:9.5]
- Verwendung eines Namensraums [VC:9.5a]
- Beispiele: `Console.ReadLine(); System.Math.Max(3,5);`

[15] VC2015: Kap. 6.13
[16] VC2015: Kap. 6.14
[17] VC2015: Kap. 6.15

2.3 Kontrollstrukturen[18]

Übersicht S. Abb. 4.

Anweisung (gekürzt)	Zweck
`if ...`	Es folgen Anweisungen, die abhängig von einer Bedingung einmal oder gar nicht ausgeführt werden sollen.
`if ...` `else ...`	Zwei alternative Anweisungsblöcke stehen zur Auswahl. In Abhängigkeit von einer Bedingung wird der `if`- oder `else`-Zweig mit seiner Anweisung oder seinem Anweisungsblock ausgeführt.

`switch ...`
`  case ...:`
`  ...`
`  default:`
	In Abhängigkeit von dem Wert eines Ausdrucks soll einer von mehreren Auswahlblöcken ausgeführt werden. Entspricht keine der `case`-Anweisungen dem Selektor, werden die Anweisungen nach `default:` ausgeführt.

`for ...`	Eine oder mehrere Anweisungen werden in Abhängigkeit von einer Schleifenbedingung ausgeführt.
`while ...`	Anweisungen sollen in Abhängigkeit von einer Bedingung einmal, mehrmals oder gar nicht ausgeführt werden. Die Bedingung wird am Anfang der Schleife geprüft.
`do ...` `while ...`	Anweisungen sollen in Abhängigkeit von einer Bedingung mindestens einmal oder mehrmals abgearbeitet werden. Die Bedingungsprüfung findet am Ende der Schleife statt.
`break`	In Abhängigkeit von einer Bedingung kann ein Anweisungsblock der `while`-, `do-while`- `for`- und `switch`-Anweisung abgebrochen werden. Die Kontrollstruktur wird beendet und die nachfolgende Anweisung ausgeführt.
`continue`	In Abhängigkeit von einer Bedingung kann in einer `while`-, `do-while`- oder `for`-Anweisung zur Auswertung der Bedingung gesprungen werden.

Abbildung 4: Übersicht Kontrollstrukturen[19]

Aufgabe:

Übung 7.12: Testauswertung

Aufgabe:

Mit dieser Aufgabe beginnt die Implementierung einer „Büchereiverwaltung".

- Legen Sie ein Projekt *Bucherei* an.
- In der Datei `Bucherei.cs` soll abgefragt werden, welche Entität gepflegt werden soll (*Medium, Nutzer, Ausleihe, [Ende]*)

[18] VC2015: Kap. 7
[19] VC2022

- Danach soll gefragt werden, welche Operation ausgeführt werden soll (*Anlegen, Lesen, Ändern, Löschen, Alle anzeigen*)
- Implementieren Sie eine Schleife um die Eingabe, die erst verlassen wird, wenn die Eingabe gültig ist.
- Fragen Sie je nach dieser Eingabe nach den weiteren benötigten Daten:
  - Anlegen/Ändern von Medien: *Signatur, Autor, Titel, Typ, Seitenzahl, Spieldauer*
  - Lesen/Löschen von Medien: *Signatur*
  - Anlegen von Nutzern: *Nachname, Vorname, Geburtsdatum*
  - Ändern von Nutzern: *Nutzernummer, Nachname, Vorname, Geburtsdatum*
  - Lesen/Löschen von Nutzern: *Nutzernummer*
  - Bei Ausleihen: *Signatur, Nutzernummer*
- Implementieren Sie Schleifen um die Eingaben, die erst verlassen werden, wenn die benötigten Felder gefüllt sind.
- Implementieren Sie um das Ganze eine weitere Schleife, die erst verlassen wird, wenn bei der Abfrage der Entität *Ende* gewählt wird.

2.4 Prozeduraufruf

Mit C# lässt sich neben dem objektorientierten Paradigma auch prozedural programmieren. Klassen sind dann Programmmodule. Im folgenden Beispiel ist skizziert, wie dies funktioniert:

```
Class MainModul{

  Main(string[] args){
    SubModul.ParameterloseMethode();
    SubModul.MethodeMitParameter("Hallo");
    int i;
    i = MethodeMitRuckgabwert("Hallo");
  }
}

class SubModul{

  internal static void ParameterloseMethode(){
    // hier kommt die Implementierung
  }

  internal static void MethodeMitParameter(string param){
    // hier kommt die Verarbeitung des Parameters
  }

  internal static int MethodeMitRuckgabwert(string param){
    // hier kommt die Verarbeitung des Parameters
    // und die Berechnung des Ruckgabewerts
```

```
    int k = 3;
    return k;
  }
}
```

Üblicherweise wird pro Klasse eine gleichnamige Programmdatei verwendet.

In der dargestellten Weise wird der Parameter als Wert übergeben. Wenn die aufgerufene Prozedur `param` ändert, dat dies keine Auswirkung auf das aufrufende Programm (s. Abb 5).

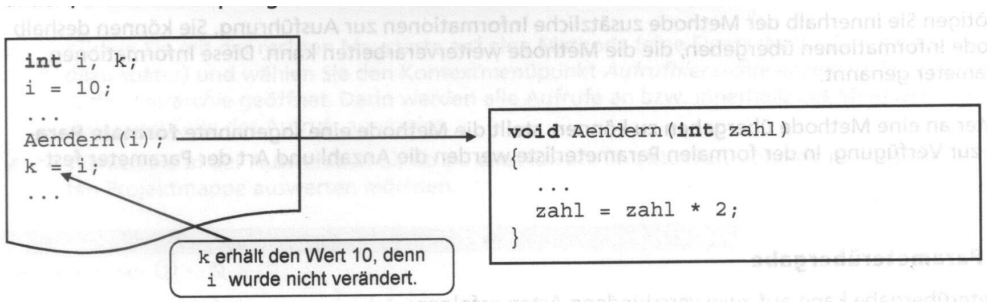


Abbildung 5: Call by Value[20]

Soll eine Rückwirkung auf in das aufrufende Programm erfolgen, so muss der Paramter als Verweis übergeben werden (s. Abb 6).

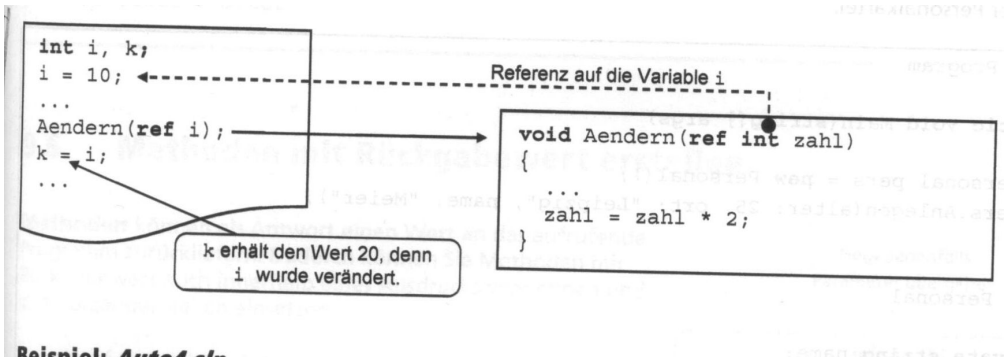


Abbildung 6: Call by Reference[21]

Aufgabe[22]:

Erstellen Sie eine Konsolenanwendung, die eine Klasse Txt enthält. Schreiben Sie in dieser Klasse eine Methode, die zwei Variableninhalte vom Typ string vertauscht. Verwenden Sie die Parameterübergabe per Referenz.

[20] VC2012: S. 106.
[21] VC2012: S. 107.
[22] VC2022: Kap. 8.12

3 System-, Datei- und Laufwerkszugriffe[23]

Mit Laufwerken, Ordnern und Dateien arbeiten[24]

Übersicht s. Abb. 7.

ie möchten ...	
aufwerksnamen ermitteln	Die Methode `System.IO.DriveInfo.GetDrives()` liefert eine Collection mit den am Computer des Anwenders verfügbaren Laufwerken.
len aktuellen Ordner ermitteln	`System.IO.Directory.GetCurrentDirectory()`
rüfen, ob eine Datei/ein Ordner xistiert	`System.IO.File.Exists()` bzw. `System.IO.Directory.Exists()`
ine Datei kopieren	`System.IO.File.Copy()`
ine Datei/einen Ordner erschieben bzw. umbenennen	`System.IO.File.Move()` bzw. `System.IO.Directory.Move()`
ine Datei/einen Ordner löschen	`System.IO.File.Delete()` bzw. `System.IO.Directory.Delete()`
lie in einem Ordner gespeicherten Dateien ermitteln	`System.IO.Directory.GetFiles()`

Abbildung 7: Befehle zum Arbeiten mit Laufwerken[25]

Mit Textdateien arbeiten[26]

.NET stellt für das Arbeiten mit Textdateien die Klassen `StreamWriter` und `StreamReader` zur Verfügung. Hier der zeilenweise Umgang:

```
using System.IO;

// Zeilenweises Lesen
StreamReader SR;
using (SR = new StreamReader("datei.name")) {
  while (SR.peek() != -1) {
    string line = SR.ReadLine();
    // Verarbeitung der gelesenen Zeile
  }
} // close findet beim Verlassen des Blockes statt

// Zeilenweises Schreiben
StreamWriter SW;
using (SW = new StreamWriter("datei.name")) {
                                    // ...", true) h"angt an Datei an
  for (int i=1; i<10; i++) {
    SW.Write("Das ist Zeile: ");
    SW.WriteLine(i);
```

[23] VC2015: Kap. 15
[24] VC2015: Kap. 15.3
[25] VC2012: S. 218.
[26] VC2015: Kap. 15.4

```
  }
} // close findet beim Verlassen des Blockes statt
```

Weitere Eigenschaften von `StreamWriter` und `StreamReader` → Abb. 8.

Ausgewählte Methoden und Eigenschaften der Klasse `StreamWriter`

Eigenschaft/Methode	Beschreibung
`BaseStream`	Gibt den zugrunde liegenden Stream an.
`Encoding`	Gibt die Codierung der Ausgabe an.
`Close()`	Schließt die `StreamWriter`-Instanz und den zugrunde liegenden Stream. Ein zusätzliches Schließen des Streams ist also nicht unbedingt erforderlich.
`Flush()`	Löscht den Puffer für die aktuelle `StreamWriter`-Instanz und veranlasst zuvor die Ausgabe aller im Puffer befindlichen Daten in den Stream.
`Write()`	Schreibt Daten fortlaufend in den Stream.
`WriteLine()`	Schließt das Schreiben von Daten mit einem Zeilenumbruch ab.

Ausgewählte Methoden und Eigenschaften der Klasse `StreamReader`

Eigenschaft/Methode	Beschreibung
`BaseStream`	Gibt den zu Grunde liegenden Stream an.
`CurrentEncoding`	Ruft die Codierung ab, die von der aktuellen `StreamReader`-Instanz verwendet wird.
`Close()`	Schließt die `StreamReader`-Instanz und den zugrunde liegenden Stream.
`EndOfStream`	Besitzt den Wert `true`, wenn das Ende der Datei erreicht ist.
`Peek()`	Gibt das nächste Zeichen zurück, ohne den Lesezeiger weiterzubewegen. Am Dateiende (kein Zeichen mehr verfügbar) wird -1 zurückgegeben.
`Read()`	Liest das nächste Zeichen oder den nächsten Block von Zeichen aus dem Stream.
`ReadLine()`	Liest eine Zeile von Zeichen aus dem Stream.
`ReadToEnd()`	Liest den Stream bis zum Ende.

Abbildung 8: Eigenschaften der Stream-Klassen[27]

Aufgabe (Bücherei):

- Legen Sie eine Klasse *Nutzerverwaltung* an.
- Schreiben Sie eine Prozedur *Anlegen* mit folgender Signatur:

  ```
  internal static void Anlegen(string Nachname, string Vorname, string Gebdatum)
  ```

- Diese Prozedur soll die Daten an die Datei `nutzer.csv` kommasepariert anhängen.
- Rufen Sie die Prozedur aus dem Hauptprogramm auf.

[27] VC2012: S. 221.

4 Verarbeitung von Zeichenketten[28]

Bei der Betrachtung der Stringoperationen ist zu beachten, dass diese teilweise dem prozeduralen Paradigma folgen, teilweise dem objektorientierten. Im ersten Fall werden die Methoden in der Form *Modulname.Prozedur(...)* (z.B. `s2 = String.Copy(s);`) aufgerufen, im zweiten Fall in der Form *variablenname.Methode(...)* (z.B. `s2 = s.Remove(1,2);`).

Eine Aufstellung von Methoden zur Stringverarbeitung findet sich in [VC:8.8]. Hier fehlt aber eine weitere nützliche Methode:

Name	Beschreibung	`string s, s2;`

`Split()`	Zerlegt einen String in ein Array von Strings. Das Trennzeichen wird als Parameter übergeben.
	`public string[] Split (char[] separator)`
		`s = "42,12,19";`
		`string[] sa = s.Split(',');`
		`sa[0] => "42"`
		`sa[1] => "12" ...`

Aufgabe (Bücherei):

Implementieren Sie in der `Nutzerverwaltung` die Prozedur `AlleAnzeigen`:

- Prüfen Sie, ob die Datei `nutzer.csv` vorhanden ist. Falls nein, geben Sie „Keine Nutzer im System" aus.
- Lesen Sie die Datei Zeile für Zeile
- Zerlegen Sie jede Zeile in ihre Bestandzeile und geben Sie diese aus.

5 Objektorientierung

Folgene Erfahrungen in der Softwareentwicklung haben zur Idee der Objektorientierung geführt:

- Strukturen sind eine sehr nützliche Sache, um Daten, die logisch zusammen gehören, zusammen zu verwalten.
- Wird eine Struktur im Speicher angelegt, wurde es in großen Programmwerken schnell unübersichtlich, wer diese Struktur für welchen Zweck gebraucht und wer Änderungen daran vornimmt.
- Unklar war oft, ab welchem Zeitpunkt welche Bestandteile einen gültigen Wert besitzen.

Vor diesem Hintergrund kam die Idee auf, die Zugriffe auf Strukturen (lesend, wie schreibend) zu kontrollieren. Der allgemeine Zugriff auf die Datenstruktur wurde also verboten, stattdessen wurden Funktionen geschaffen, über die auf die Daten zugegriffen werden konnte. Eine Datenstruktur mit den dazugeförigen Zugriffsfunktionen nennt sich *Klasse*.

Die Instanzierung einer Klasse bedeutet, einer Struktur einen konkreten Speicherbereich zuzuordnen. Nur eine instanzierte Struktur (=Objektinstanz) kann auch verwendet werden.

[28] VC2015: Kap. 8.8

Bei einer Klasse sind im Gegensatz zu einer Struktur die interen Datenelemente von außen nicht zugreifbar. Alle Zugriffe erfolgen prozedural über öffentliche *Methoden*.

Auf der anderen Seite dienen Klassen oft als Container, um Funktionen, die in logischem Zusammenhang stehen, in einer gemeinsamen Einheit zusammenzufassen. Sie bilden ein „Funktionsmodul". Funktionen einer Klasse, die nicht auf der „internen" Datenstruktur arbeiten, nennen sich *statisch*. Zur Verwendung von statischen Funktionen muss zuvor keine Objektinstanz angelegt werden.

Real existierende Klasse befinden sich irgenwo im Spektrum zwischen „Funktionsmodul" und „Datenverwaltungsmodul".

5.1 Klassen, Felder und Methoden[29]

Grundlagen der objektorientierten Programmierung

Ausgangspunkt: Strukturierte Programmierung bis in die 80'ger Jahre

- Daten, oft global
- Funktionen und Prozeduren, die auf den Daten arbeiten

Ansatz der Objektorienierung:

- Daten und Funktionen („Methoden") gehören zusammengefasst
- Zugriff ausschließlich über Schnittstellen (s. Abb. 9)

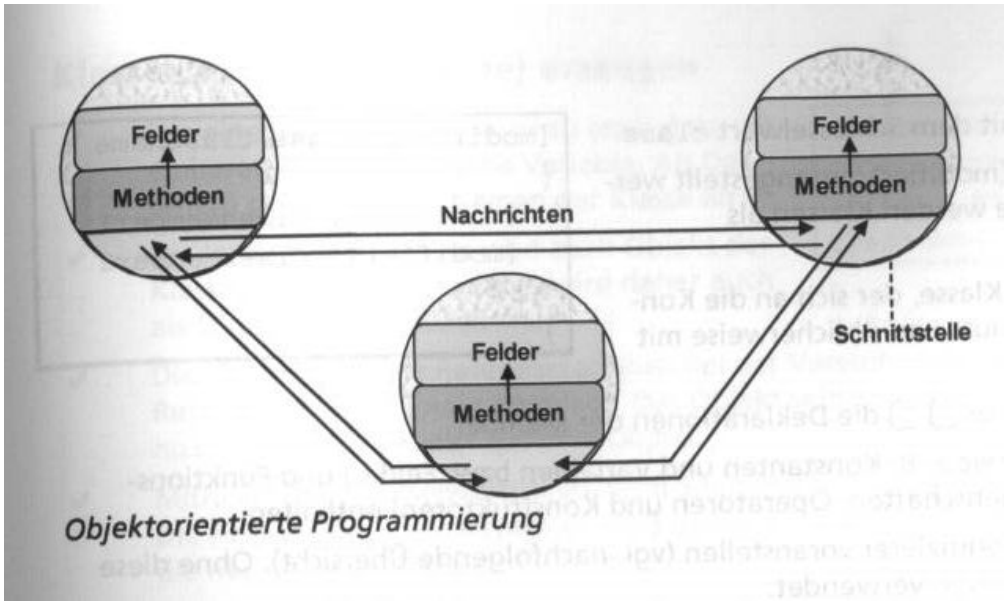


Abbildung 9: Prinzip der Objektorientierung[30]

[29] VC2015, Kap. 8

Klassen und Instanzen

Was ist eine Klasse, was sind Objekte?

> Eine Klasse beschreibt als Bauplan die Gemeinsamkeiten einer Menge von Objekten. Eine Klasse ist somit ein Modell, auf dessen Basis Objekte erstellt werden können. Die Klasse beinhaltet die vollständige Beschreibung dieses Modells. So vereint eine Klasse alle Felder (auch Datenelemente genannt), die diese Klasse kennzeichnen, und alle Methoden zur Verwendung der Daten und zur Beschreibung der Funktionalität. Da die Felder und Methoden zu der Klasse gehören, werden die auch als Member (Mitglieder) der Klasse bezeichnet.
>
> Objekte (Instanzen) stellen konkrete Exemplare der Klasse dar. Auf der Basis einer Klasse können beliebig viele Objekte erzeugt (instanziiert) werden. In den Feldern werden die objektspezifischen Werte für das jeweilige Objekt gespeichert, während die Methoden Aktionen ausführen. So können Methoden Daten für die Klasse in Empfang nehmen, Feldinhalte (Daten) der Klasse ausgeben oder die Feldinhalte der Klasse be- bzw. verarbeiten.[31]

Wenn wir allgemein von Autos sprechen, so ist „Auto“ die Klasse. Autos sind charakterisiert durch Marke/Typ, Farbe, Bereifung, Leistung, etc. Betrachten wir ein bestimmtes Auto, eine „Instanz der Klasse Auto“, so können wir diesen Eigenschaften konkrete Werte zuweisen: Ein konkretes Auto ist beispielsweise ein blauer Opel Zafira mit 85 kW Leistung etc.

- Klassen werden mit dem Schlüsselwort `class` definiert.
- Bestandteile (*member*) werden innerhalb der Klasse mit `[modifiers] type identifier;` definiert.
- Modifier [VC:8.2]
- Instanzen werden mit dem Schlüsselwort `new` erzeugt. [VC:8.2a]

C# arbeitet ausschließlich (wie Java, im Gegensatz zu C++) mit Referenzen (=Pointer) auf Objekte. Der Rückgabewert einer Instanzerzeugung mit `new` ist also eine Referenz auf ein Stückchen Speicher im Freispeicher (Heap). Die Freigabe des Freispeichers erfolgt automatisch (garbage collection, GC). Bei der Zuweisuns von Objekten wird lediglich die Referenz kopiert.

Abb. 10 zeigt diesen Vorgang einer Zuweisung. Nach der Zuweisung zeigen beide Variablen auf dasselbe Objekt. Das Fahrzeug mit der Geschwindigkeit 86 könnte nun vom GC aus dem Freispeicher entfernt werden.

[30] VC2012: S. 97.
[31] VC2012: S. 97.
[32] VC2012: S. 101.

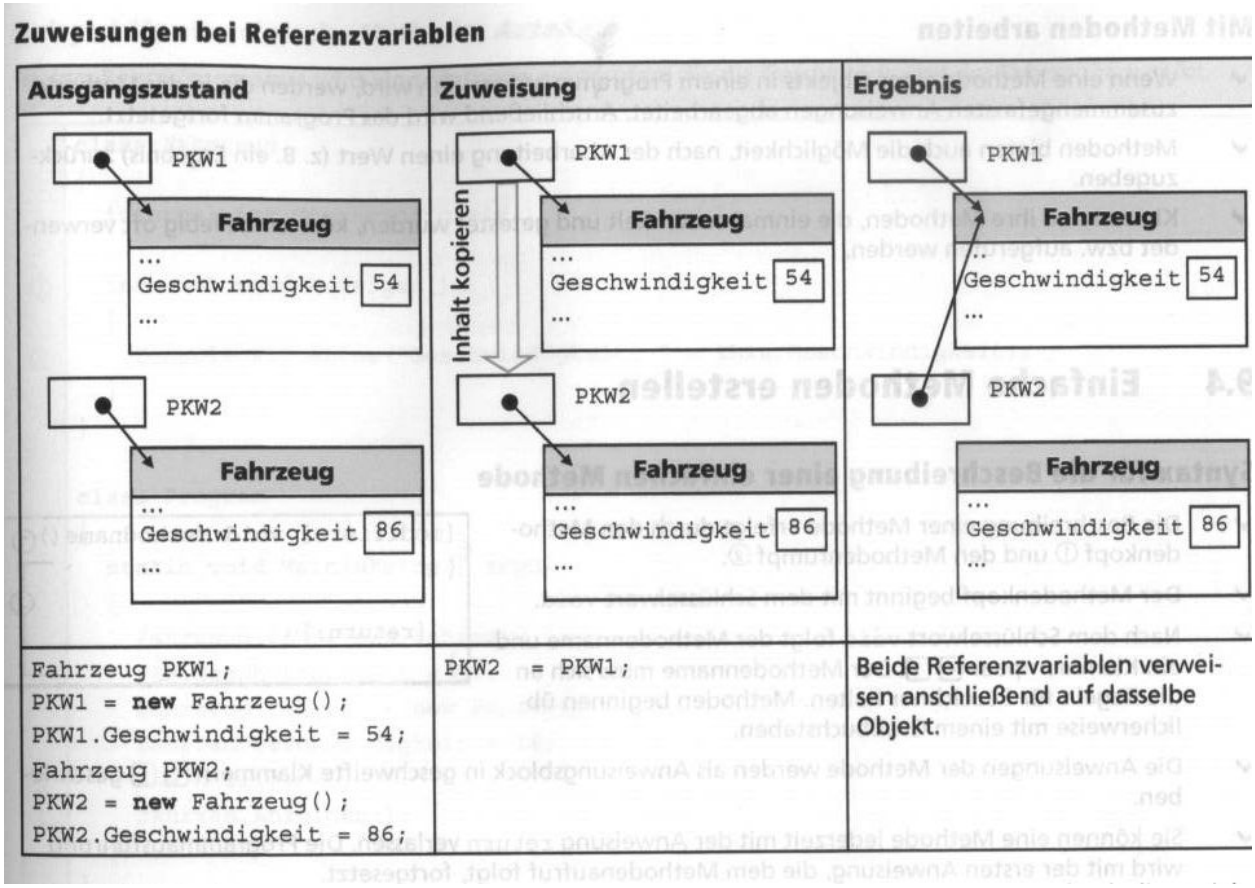


Abbildung 10: Kopieren von Instanzreferenzen[32]

Methoden mit Parametern erstellen

In der Klassendefinition werden die Methoden definiert:

```
[modifiers] void Methodenname([ref|out] type1 identifier1 [= default] [, ...]) {
    ...
}
```

- Bei nicht-elementaren Typen (Klassen) ist der Identifier immer eine Referenz.
- `ref/out` bezieht sich in diesem Fall auf die Referenz, nicht auf das Objekt!

Rückgabewerte werden wie bei statischen Methoden angegeben. Auch hier ist zu beachten, dass bei nicht-elementaren Typen eine Referenz auf eine Objektinstanz zurückgegeben wird.

Bei objektorientierten Sprachen können Methoden mit demselben Namen, aber unterschiedlichen Signaturen (= Übergabeparametern) definiert werden. Der Compiler wählt die richtige Implementierung [VC:8.9].

Aufgabe[33]:

- Erstellen Sie in einem Konsolenprogramm eine Klasse Wert. Deklarieren Sie in dieser Klasse überladene Methoden, die einen `int`- bzw. einen `double`-Wert oder einen `string` als Parameter entgegennehmen und eine entsprechende Ausgabe durchführen:
  - Die Ganzzahl lautet: übergebene Zahl
  - Die Dezimalzahl lautet: übergebener Wert
  - Der Text lautet: übergebener Text
- Testen Sie die Methoden, indem Sie sie je einmal mit einem Wert der drei verschiedenen Datentypen aufrufen.

5.2 Kapselung und Konstruktoren[34]

Kapselung

Auf Felder einer Klasse soll grundsätzlich nur kontrolliert zugegriffen werden. In der Klasse muss die Möglichkeit bestehen, eine Wertptüfung o.ä. vorzunehmen. Dies führt in anderen oo Sprachen (Java, C++) zu folgenden stereotypen Konstrukten:

```
private int wert; // Feld ist von aussen nicht zugreifbar
public void setWert(int i) { this.wert = i; }
public int getWert() { return this.wert; }
```

Mit dieser Konstruktion hat der Entwickler die Möglichkeit, eine Wertprüfung zu ergänzen, ohne die Schnittstelle nach außen zu verändern.

C# hat hier eine elegantere Möglichkeit. Das Feld selbst wird zwar von außen zugreifbar gemacht. Prüfungen können, wie im Folgenden gezeigt, ergänzt werden.

Eigenschaften

Das Konzept der Eigenschaften sieht vor, dass diese in der Verwendung wie öffentliche Membervariablen aussehen:

```
// set
Instance.Member = value;
// get
var = Instance.Member;
```

Soll der Zugriff also ohne weitere Prüfung oder Einschränkung erfolgen, so kann die Membervariable so definiert werden:

[33] VC2022: 8.12.3: Überladen einer Methode
[34] VC2015: Kap. 9

```
class MeineKlasse {
  public Type Member;
  ...
}
```

Soll beispielsweise das Schreiben der Variable nicht-öffentlich sein, so kann dies so formuliert werden:

```
class MeineKlasse {
  public Type Member { get; private set; }
  ...
}
```

Soll beim Schreiben zusätzlich eine Prüfung statt finden, muss zusätzlich eine interne Repräsentation der Membervariablen angelegt werden:

```
class MeineKlasse {
  private Type member; // interne Repr"asentation
  public  Type Member {
    get {
      return member;
    }
    private set {
      if ( value < 0 ) ... // Wertpr"ufung
      member = value;
    }
  }
  ...
}
```

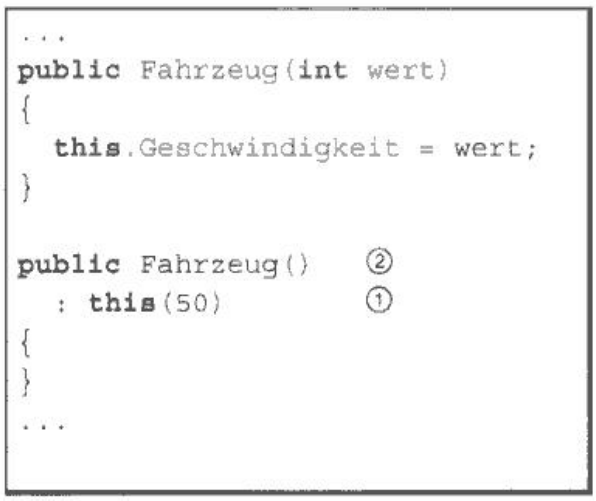
Der Name `value` ist dabei durch die Sprache vorgegeben.
Ein komplettes Beispiel: [VC:9/Fahrzeuge.sln].

Konstruktoren

- Konstruktoren sind spezielle Methoden einer Klasse.
- Ein Konstruktor wird beim Erzeugen einer Instanz aufgerufen (`new ...`).
- Das Entfernen einer Instanz aus dem Speicher erfolgt nicht deterministisch (*garbage collector*).
- Grundsätzlich existiert der Standardkonstruktor (Konstruktor ohne weitere Parameter).
- Selbstdefinierte Konstruktoren unterscheiden sich in der Parameterleite (=Signatur) .[VC:9/Fahrzeuge1.sln]
- Ein Kontruktor kann einen anderen Konstruktor derselben Klasse verwenden (s. Abb. 11).

Objektinstanzen können auch über *Objektinitialisierer* mit Werten belegt werden [VC:9/AutoInit.sln].

[35] VC2012: S. 125.



```
...
public Fahrzeug(int wert)
{
  this.Geschwindigkeit = wert;
}

public Fahrzeug()      ②
  : this(50)           ①
{
}
...
```

Abbildung 11: Konstruktoren[35]

Aufgabe (Bücherei):

- Erstellen Sie eine Klasse `Nutzer` mit den entsprechenden Bestandteilen, dazu einen Konstrutor mit den drei Parametern und eine `ToString`-Methode.
  `public override string ToString()`
- Ergänzen Sie die Klasse `Nutzerverwaltung` um eine Methode
  `internal static Nutzer Lesen(int Nutzerid)`, die aus der Datei `nutzer.csv` die Zeile `Nutzerid` liest, daraus eine Instanz von `Nutzer` anlegt und zurückgibt. Falls `Nutzerid` größer als die Anzahl der Zeilen ist, wird `null` zurückgegeben.
- Rufen Sie `Lesen()` im Hauptprogramm auf. Prüfen Sie den Rückgabewert auf `null` und geben Sie, falls vorhanden, den Nutzer aus.

5.3 Vererbung[36]

Die Vererbung ist ein wichtiges Merkmal objektorientierter Programmiersprachen. ...

- Eine Klasse kann nur von *einer* Klasse abgeleitet werden (*Einfachvererbung*).[37]

Die Syntax für die Vererbung lautet:

```
[modifier] class Klasse : Basisklasse
{
    ...
}
```

Soll in einem Konstruktor der abgeleiteten Klasse der Konstruktor der Basisklasse aufgerufen werden, geschieht das mit `: base()`.

[36] VC2015: Kap. 10
[37] VC2012: S. 138.

Alle Klassen erben direkt oder indirekt von `System.Object`. Von dieser Klasse erben alle Klassen die Methoden `GetType()` und `ToString()`.

Eine Instanz der abgeleiteten Klasse ist typkompatibel mit der Basisklasse. Ein Muster, das dies ausnützt, ist das Fabrikmuster, das hier in seiner vereinfachten Form der Fabrikmethode dagestellt ist:

```
class Base {};

class Derived1 : Base {};
class Derived2 : Base {};

Base Fabric(...)  {
  if (...)    return new Defived1(...);
  elseif(...) return new Defived2(...);
  return null;
}
```

Aufgabe (Bücherei):

- Leiten Sie von der Klasse `Medium` die Klassen `Buch` und `CD` ab (kann in derselben Datei erfolgen).
  - Verteilen Sie die Attribute sinnvoll auf die Klassen.
  - Ergänzen Sie jede Klasse um eine Methode `string Format()`, die den Datensatz als Zeichenkette in folfendem Format zurückgibt:
    `<Signatur>,<Autor>,<Titel>,<Typ>,<Seitenzahl>,<Spieldauer>`
    Für die jeweilige Klasse nicht zutreffende Werte werden mit Dummy-Werten belegt. In den abgeleiteten Klassen verlangt der Compiler ein zusätzliches `override`.
  - Der Konstruktor von `Buch` wird mit den Parametern
    `signatur, autor, titel, seitenzahl` aufgerufen.
  - Der Konstruktor von `CD` wird mit den Parametern
    `signatur, autor, titel, spieldauer` aufgerufen.
  - Beide Konstruktoren verwenden den Konstruktor von `Medium`, der die gemeinsamen Attribute übernimmt.
  - Zusätzlich habe `Medium` die statische Methode `Erzeugen`. Diese nimmt die sechs Parameter und erzeugt je nach Typ ein `Medium`, `Buch` oder `CD`.
- Legen Sie die Klasse `Medienverwaltung` mit der statischen Prozedur `Anlegen(Medium)` an. Diese öffnet die Datei `medien.csv` zum Anhängen und fügt über die `Format`-Methode einen Datensatz hinzu.
- Implementieren Sie im Hauptprogramm das Anlegen eines Mediums.

5.4 Polymorphismus

Wenn Sie nun über das Hauptprogramm Medien anlegen, werden Sie feststellen, dass in der Datei nur Datensätze vom Typ Medium erscheinen. An dieser Stelle wird nun ein Mechanismus benötigt, der zur Laufzeit die tatsächliche Klasse einer Instanz bestimmt und danach die passende Implentierung auswählt. Dieser Mechanismus nennt sich *Polymorphismus* (Vielgesichtigkeit).

Das polymorphe Überschreiben von Methoden erfolgt duch die *modifier* `virtual` in der Basisklasse und `override` in den abgeleiteten Klassen.

Aufgabe (Bücherei):

- Implementieren Sie die `Format()`-Methode polymorph.
- Überprüfen Sie erneut die Dateieinträge.

5.5 Abstrakte Klassen und Methoden[38]

> Abstrakte Klassen implementieren ihre Methoden und Eigenschaften nur teilweise oder auch gar nicht. Die Implementiereung der abstrakten Methoden muss in den abgeleiteten Methoden erfolgen. Von abstrakten Klassen können keine Instanzen gebildet werden. Mithilfe abstrakter Klassen können Sie eine gewisse Grundfunktionalität für alle abgeleiteten Klassen zur Verfügung stellen. Außerdem können Sie erzwingen, dass bestimmte Elemente in den abgeleiteten Klassen implementiert und angepasst werden. [39]

Abstrakte Klassen und Methoden erhalten das Schlüsselwort `abstract`. Eine Klasse muss als abstrakt gekennzeichnet werden, wenn mindestens eine Methode abstrakt ist ([VC:11/Datumsformate.sln], [Wurm: S. 212ff.]).

Aufgabe:

Ergänzen Sie die Bibliotheksanwendung:

- Verwandeln Sie `Medium` in eine abstrakte Klasse mit `Format()` als abstrakte Methode.

5.6 Schnittstellen[40]

- Klassen können beliebig viele Schnittstellen implementieren.
- Eine Schnittstelle darf keine Datenmembers haben
- Methoden dürfen statusfreie Implementierungen haben

[38] VC2015: Kap. 11.5, Wurm: 7.1.3
[39] VC2012: S. 155.
[40] VC2015: Kap. 12, Wurm: Kap. 7.1.14

- Interfaceimplementierungen müssen public sein.

Syntax:

```
[public|internal] interface Iname [: I1 [, I2, ...]] {
    ...
}
```

Implementiert eine Klasse eine bestimmte Schnittstelle, so lautet die Syntax dafür:

```
class Klasse : [Basisklasse, ] I1 [, I2, ...]] {
    ...
}
```

Damit eine Klasse vollständig implementiert ist und damit instanziert werden kann, müssen alle übernommenen Schnittstellenmethoden implementiert werden ([VC:12/Personen.sln], [Wurm: S. 218ff.]).

Anmerkung: Klassen, die die Schnittstelle `IDisposable` implementieren, können wie im Codebeispiel in Kap. 3 gezeigt, verwendet werden. Um das `IDisposable` zu implementieren, brauchen die Klassen eine `Dispose()`-Methode. Diese führt bei den Klassen zur Dateibearbeitung das Schließen der Datei aus.

Aufgabe[41]:

- Erstellen Sie eine Konsolenanwendung, um ein Programm für eine einfache Kontoführung zu schreiben.
- Entwickeln Sie in einer separaten Datei eine Klasse Konto. Als einziges Feld soll diese Klasse eine als private deklarierte Variable für den Kontostand besitzen.
- Deklarieren Sie in einer weiteren separaten Datei eine Schnittstelle, in der eine Eigenschaft zum Schreib-/ Lesezugriff auf den Kontostand deklariert ist.
- Importieren Sie die Schnittstelle durch die Klasse Konto und implementieren Sie dort die von der Schnittstelle deklarierte Eigenschaft.
- Implementieren Sie einen Konstruktor, über den Sie den Kontostand mit einem Wert von 5000 initialisieren.
- Das Programm sollte so aufgebaut sein, dass sowohl zu Beginn als auch nach jeder Ein- bzw. Auszahlung der aktuelle Kontostand angezeigt wird. Durch die Eingabe einer Zahl 1, 2 oder 0 soll der Anwender die Möglichkeit haben, eine Einzahlung (1) oder eine Auszahlung (2) vorzunehmen oder das Programm mit der Eingabe der Zahl 0 zu beenden. (Tipp: `switch-case`-Anweisung)

[41] [VC2015] Übung 12.6.2: Einfache Kontoführung

Aufgabe (Bücherei):

- Definieren Sie die Schnittstelle `IFormatable`, die eine `string Format()` vorgibt.
- `Medium` soll diese Schnittstelle implementieren (erben).
- Schreiben Sie in `IFormatable.cs` eine Funktion
  `static public void AddObjectToFile(IFormatable Object, string file)`,
  die die Datei zum schreiben öffnet und das Objekt über den Aufruf von `Format()` anhängt.
- Verwenden Sie in `Medienverwaltung.Anlegen` die neue Funktion.

5.7 Beziehungen

Objekte können Beziehungen zu anderen Objekten haben. Diese können unterschiedlich ausgestaltet werden. Entweder wird nur ein passender Schlüssel zum anderen Objekt gespeichert oder gleich eine Referenz auf das andere Objekt. Gundsätzlich haben beide Vorgehensweisen Vor- und Nachteile:

- Beim Anlegen geht es schneller, wenn vom referenzierten Objekt nur ein Schlüssel abgelegt werden muss.
- Beim Lesen geht es schneller, wenn das referenzierte Objekt bereits im Speicher vorliegt.

In objektorientierten Sprachen ist eher ein *eager loading* üblich, also Variante 2.

Aufgabe:

Schreiben Sie eine Klasse `Ausleihe`:

- Der Konstruktor hat als Parameter (damit die Klasse als Member) *Medium, Benutzer*.
- Die Klasse bekommt folgende Getter, die wahlweise das referenzierte Objekt oder eine Eigenschft dieses Objkekts zurückgeben:
  - `getNutzer()` gibt das Nutzerobjekt zurück.
  - `getNutzernummer()` gibt die Nummer aus dem Nutzerobjekt zurück.
  - `getMedium` gibt das Mediumobjekt zurück.
  - `getSignatur()` gibt die Signatur des Mediums zurück.

Verwendet wird die Klasse erst später.

6 Komplexe Datentypen[42]

6.1 Eindimensionale Arrays

Syntax für die Deklaration + Instanzierung + Initialisierung:

```
type[] var [ = new type[size] | = { wert1, wert2, ... } ] ;
```

- Der Elemetzugriff erfolt in der Form: `var[idx]`.
- Die Felder haben eine konstante Länge.
- Fehlerhafte Zugriffe führen zu einer `IndexOutOfRangeException`.

Wie bei Objekten ist in der Variable eine Referenz in den Freispeicher abgelegt (s. Abb. 12).

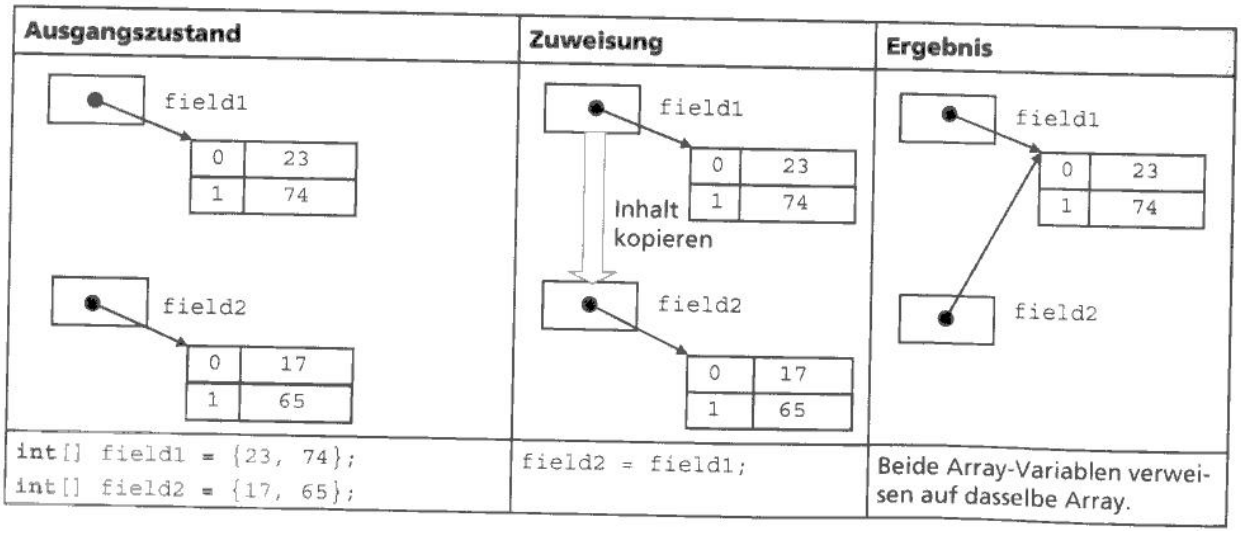


Abbildung 12: Zuweisung bei Array-Variablen[43]

Neben der Schleife über den Index (`for ( int i = 0, ...`) bietet C# die `foreach`-Schleife:

```
foreach ( type elem in ArrVar ) {
  ...
}
```

6.2 Mehrdimensionale Arrays

Mehrdimensionale Arrays [VC:13.2]]

Verzweigte (unregelmäßige) Arrays: Werden in der 1. Dimension Referenzen auf weitere Arrays abgelegt, müssen diese nicht notwendig gleich lang sein (s. Abb. 13).

Da Arrays in C# besondere Klassen sind, gibt es auch Methoden dafür [VC:13.3].

[42] VC2015: Kap. 13
[43] VC2012: S. 125.
[44] VC2012: S. 125.

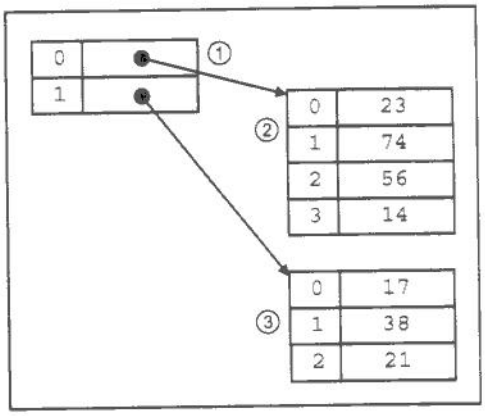
```
① int[][] field = new int[2][];
② field[0] = new int[] { 23, 74, 56, 14 };
③ field[1] = new int[] { 17, 38, 21 };
```

Beispiel: "VerzweigtesArray.sln"

① Ein Array wird erzeugt, in dem zwei Arrays mit `int`-Werten gespeichert werden sollen.

② Das erste enthaltene Array wird erzeugt. Das Array soll die Werte `23`, `74`, `56` und `14` enthalten. Über das Schlüsselwort `new` wird Speicherplatz für das Array mit 4 Elementen vom Typ `int` reserviert und dieses mit den vier Werten initialisiert.

③ Das zweite enthaltene Array besteht nur aus drei Elementen: `17`, `38` und `21`.



Ein verzweigtes Array

Abbildung 13: Verzweigte Arrays[44]

6.3 Auflistungen

> Das .NET Framework stellt verschiedene Klassen für Auflistungen bereit, um beispielsweise Listen, Warteschlangen, Stapel, Wörterbücher oder sogenannte Hashtabellen zu erstellen. ... Sie müssen dazu den Namespace System.Collections zu Beginn einbinden[45].

Wichtig ist es zu wissen, welche Grundtypen an Auflistungen es gibt. Benötigt man eine davon, schaut man die Verwendung nach.

Liste → `ArrayList`, Stapel → `Stack`, Wörterbuch → `Dictionary`, Hashtabelle → `Hashtable`

> Alle Auflistungen und Arrays implementieren die Schnittstelle IEnumerable. Dadurch entsteht ein Positionszeiger, ein sogenannter Enumerator, zur Verfügung, mit dessen Hilfe Sie einen iterativen Lesezugriff auf die Auflistungs- oder Arrayelemente erhalten. Im Unterschied zur foreach-Schleife lässt sich der Enumerator mit der Methode `GetEnumerator()` ermitteln, mit `Reset()` jederzeit auf den Anfang der Liste zurücksetzen und mit `MoveNext()` zum jeweils nächsten Element bewegen[46].

`foreach` verwendet auch den Enumerator. Jede Klasse, die die `IENumerable`-Schnittstelle implementiert, lässt sich mit `foreach` durchlaufen.

Was hier in C# „Enumerator" heißt, nennt sich oftmals auch „Iterator". Der Unterschied liegt in der konkreten Sprachunterstützung: Während der „Iterator" selbst eine Klasse ist, die die entsprechedenden Methoden (`MoveNext()`, ...) implementieren muss, existiert in C# mit `yield` eine entsprechende Sprachunterstützung.

Beispiel: [VC:13.8]

[45] VC2012: S. 183.
[46] VC2012: S. 186.

Aufgabe:

Bringen Sie das Beispiel `Iterator.sln` zum Laufen.

Grundsätzlich lassen sich bei den Aufzählungstypen getypte von ungetypten unterscheiden. Bei ungetypten Aufzählungstypen werden einfach *Objekte* hintereinander abgelegt (z.B. `ArrayList`). Bei getypten Aufzählungstypen wird festgelegt, von welchem Typ die Objekte sein müssen, die in einer Aufzählung abgelegt werden (z.B. `Dictionary`).

Übersicht über Collections: Choose a collection[47]

Beispiel für `Dictionary`:

```
using System;
using System.Collections.Generic;

public class Example
{
    public static void Main()
    {
        // Create a new dictionary of strings, with string keys.
        //
        Dictionary<string, string> openWith =
            new Dictionary<string, string>();

        // Add some elements to the dictionary. There are no
        // duplicate keys, but some of the values are duplicates.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // The indexer can be used to change the value associated
        // with a key.
        openWith["rtf"] = "winword.exe";

        // If a key does not exist, setting the indexer for that key
        // adds a new key/value pair.
        openWith["doc"] = "winword.exe";

        // When a program often has to try keys that turn out not to
        // be in the dictionary, TryGetValue can be a more efficient
        // way to retrieve values.
        string value = "";
        if (openWith.TryGetValue("tif", out value))
        {
            Console.WriteLine("For key = \"tif\", value = {0}.", value);
        }
        else
        {
            Console.WriteLine("Key = \"tif\" is not found.");
        }
```

[47] https://learn.microsoft.com/en-us/dotnet/standard/collections

```
        // ContainsKey can be used to test keys before inserting
        // them.
        if (!openWith.ContainsKey("ht"))
        {
            openWith.Add("ht", "hypertrm.exe");
            Console.WriteLine("Value added for key = \"ht\": {0}",
                openWith["ht"]);
        }

        // When you use foreach to enumerate dictionary elements,
        // the elements are retrieved as KeyValuePair objects.
        Console.WriteLine();
        foreach( KeyValuePair<string, string> kvp in openWith )
        {
            Console.WriteLine("Key = {0}, Value = {1}",
                kvp.Key, kvp.Value);
        }
    }
}
```

Aufgabe (Bücherei):

- Ändern Sie die Klasse `Medienverwaltung` so ab, dass sie als „richtige Klasse" vor ihrer Verwendung instanziert werden muss (keine statischen Methoden).
- In ihrem Konstuktor liest sie `medium.csv`, legt für jede Zeile über `Erzeugen` ein Medium an und fügt dieses dem Verzeichnis `MediaDict` hinzu: Schlüssel ist die Signatur, Wert die Instanz.
- Implementieren Sie in `Medienverwaltung` die Methode `Lesen(String Signatur)`, die prüft, ob in `MediaDict` die Signatur vorhanden ist. Falls ja, wird die entsprechende Medieninstanz, falls nein `null` zurückgegeben.
- `AlleLesen` bildet aus dem Verzeichnis eine Liste aller Medien und gibt diese zurück.
- `Anlegen` muss um das Neuladen von `MediaDict` ergänzt werden.
- Implementieren Sie im Hauptprogramm die Funktionen *Lesen* und *Alle lesen* für die Entität *Medium*. Die Ausgabe soll jeweils unter Verwendung von `Format()` erfolgen.

6.4 Delegaten und Ereignisse

Delegaten

Delegaten sind die C#-Variante von Funktionspointern. Dieses aus C stammende Konstrukt dient dazu, eine Implementierung zur Laufzeit festzulegen. Im Abschnitt 5.4 wurde bereits gezeigt, dass dieses Ziel auch mit Mitteln der Objektorientierung erreicht werden kann. Delegaten sind die Standardmethode ausführbaren Code an Ereignisse zu hängen.

Die Verwendung geschieht in drei Schritten:

Als ersten muss ein Typ für die gewünschte Signatur erstellt werden:

```
public delegate <type> <name> (<params>);
```

Beispiel: `public delegate void Del(string message);`

Als zweites muss eine statische Funktion mit dieser Signatur existieren.

Beispiel:

```
public static void DelegateMethod(string message)
{
    System.Console.WriteLine(message);
}
```

Als drittes wird der Delegat instanziert und verwendet.

Beispiel:

```
// Instantiate the delegate.
Del handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Schritt zwei und drei können durch die Verwendung von Lambda-Funktionen (anonyme Funktionen) zusammengefasst werden. Für die Definition von Labmbda-Funktionen gibt es zwei Varianten:

Lambda-Ausdruck: `(input parameters) => expression`

Beispiel[48]:

```
delegate int del(int i);
static void Main(string[] args)
{
    del myDelegate = x => x * x;
    int j = myDelegate(5); //j = 25
}
```

Lambda-Statement: `(input parameters) => {statement;}`

Beispiel:

```
delegate void TestDelegate(string s);
...
TestDelegate myDel =
  n => { string s = n + " " + "World"; Console.WriteLine(s); };
myDel("Hello");
```

[48]MSDN: Lambda Expressions (C# Programming Guide)

Ereignisse[49]

Gerade in der GUI-Programmierung wird viel mit Ereignissen umgegangen. Jede Aktion auf dem Bildschirm führt zu einem Ereignis, das entsprechend behandelt werden muss. C# stellt zur Ereignisbehandlung einen Standarddelgaten zur Verfügung:

```
public delegate void EventHandler(object sender, EventArgs e);
```

Ein Ereignis wird nun so definiert:

```
public event EventHandler MyEvent;
```

Anmelden eines Eventhandlers:

```
public void MyHandler(object sender, EventArgs e) {
  // do anything
}

MyEvent += new EventHandler(MyHandler);
```

Auslösen der Handler:

```
// Check, if there is any Handler
if (MyEvent != null)
  MyEvent(this, null);
```

Aufgabe (Bücherei):

Bei wichtigen datenändernen Operationen kann eine Bestätigung durch den Nutzer sinnvoll sein. Dazu wird die Anwendung folgendermaßen erweitert:

- Um einen Delegaten `DRuckfrage`, der als Paramter die Rückfrage entgegen nimmt und als Ergebnis `bool` liefert.
- In der `Medienverwaltung`
  - Um die Eigenschaft `event DRuckfrage Ruckfrage`,
  - um die Methode `RuckfrageAnmelden(DRuckfrage R)`, die diesen *handler* bei `Ruckfrage` anmeldet,
  - um eine Methode `Loschen(string Signatur)`, die
    - prüft, ob `Signatur` überhaupt im `MediaDict` vorhanden ist. Fall nicht, beendet sich die Methode.
    - Falls die Signatur vorhanden ist, wird, falls ein Delegat angemeldet wurde, `Ruckfrage` aufgerufen. Je nach Ergebnis wird `Signatur` aus `MediaDict` entfernt. Auf eine Persistierung der Datenänderung wird verzichtet.
- Im Hauptprogramm instanzieren Sie einen Delegaten, der über die Konsole die übergebene Frage stellt und je nach Antwort `True` oder `False` zurückgibt.
- Melden Sie den Delegaten bei der `Medienverwaltung` an und testen Sie den Code.

[49] Wurm: Kap.7.2

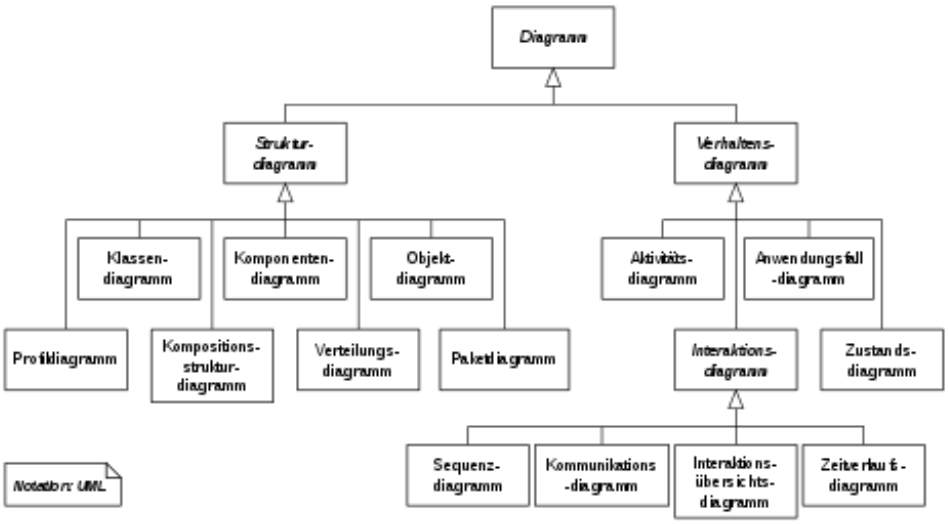


Abbildung 14: Hierarchie von Diagrammen in UML 2.2[50]

7 Unified Modeling Language (UML)

Zur graphischen Darstellung von Objektstrukturen hat sich die *Unified Modeling Language* (UML) etabliert. Der gesamte Umfang von UML ist sehr groß und umfasst auch die Modellierung des dynamischen Verhaltens. In der Hochzeit der Objektorientierung gab es Ideen, Software allein über UML zu spezifizieren und aus den UML-Spezifikationen generieren zu können. Abbildung 14 zeigt eine Übersicht der Diagramme.

Um Folgenden betrachten wir das Klassendiagramm näher.

Klassendiagramm (Wikipedia)[51]

Beispiel

Im Folgenden wird die Erstellung eines ER-Modells für eine Büchereianwendung exemplarisch durchgespielt.

Die zentralen starken Entitäten einer Büchereianwendung sind `Buch` und `Benutzer`. Schnell stellt man aber fest, dass eine Bücherei nicht nur Bücher zum ausleihen hat, sondern auch Zeitschriften, CDs und andere Medien. Eine erste Modellierung führt also zu Modell in Abb. 15.

Mit diesem einfachen Modell lassen sich bereits die Geschäftsvorfälle Anlegen/Ändern/Anzeigen von Medien und Benutzern beschreiben.

Interessant wird die Geschichte, wenn wir die Geschäftsvorfälle „Ausleihe und Rückgabe von Medien" hinzunehmen. Die Ausleihe bringt Medien und Benutzer miteinander in Verbindung, stellt also eine Relation dar.

Die Kardinalitäten sind dabei wie folgt:

[50] `https://de.wikipedia.org/wiki/Unified_Modeling_Language#Diagramme`
[51] `https://de.wikipedia.org/wiki/Klassendiagramm`

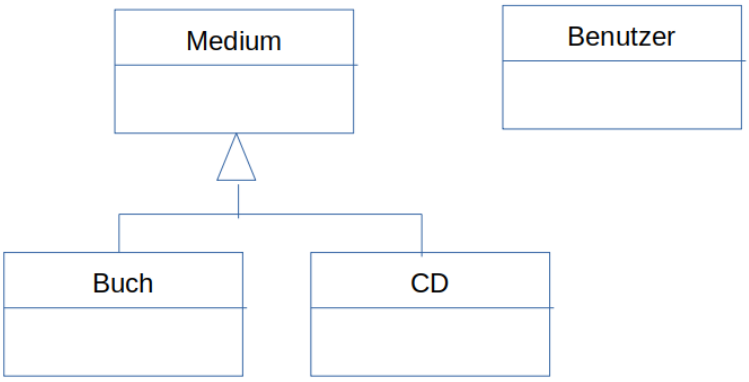


Abbildung 15: Erstes ER-Diagramm für die Büchereianwendung

- Ein Buch kann (nacheinander) von keinem bis vielen Nutzern entliehen werden (0..n).
- Ein Nutzer kann (gleichzeitig oder nacheinander) von keines bis viele Medien entleihen (0..n).

Unterscheiden wir also im Modell nicht zwischen aktuellen und vergangenen Leihen, was natürlich möglich wäre, lassen sich Randbedingungen wie „Ein Buch kann zu einem Zeitpunkt nur von einem Nutzer entliehen sein" oder „Ein Nutzer darf gleichzeitig maximal zehn Medien entleihen" nur textuell beschreiben. Diese werden dann in der Beschreibung der Geschäftsvorfälle niedergeschrieben.

Unter Hinzunahme der Geschäftsvorfälle Ausleihe/Rückgabe erhalten wir nun das Modell in Abb. 16

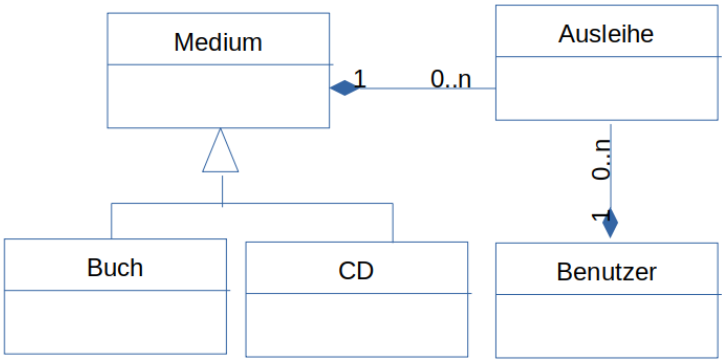


Abbildung 16: ER-Diagramm für die Büchereianwendung mit Ausleihe

Bei der tatsächlichen Implementierung treten weitere Klassen/Module hinzu. Je nach Programmiersprache wird zwischen Klassen und Modulen unterschieden (C++ ja, C#/Java nein). Das Modul ist dabei der Grenzfall einer Klasse, die keinerlei interne Daten enthält. Solche Klassen können sein:

- Klassen für die Erstellung einer graphischen Oberfläche nach dem *model view control pattern* (MVC)
- Klassen für die Datenhaltung nach dem *data access object pattern* (DAO)

- Klassen für technische Querschnittfunktionen wie das Logging.
- Je nach verwendeter Architektur kommen noch Klassen/Module für die Entitätsverwalter, die Geschäftsvorfälle, sowie Zugangsklassen für den Anwendungskern mit querschnittlichen Aufgaben hinzu.

Schon das kleine unvollständige Büchereibeispiel hat am Ende über 20 Klassen. Daher ist man davon abgekommen Diagramme der Implementierungsklassen zu erstellen. Es empfieht sich die Verwendung von Entwurfsmuster, wovon einige in Kap. 8 beschrieben sind. Dies ermöglicht den Zweck einer Klasse so zu beschreiben, dass auch nachfolgeden Entwicklergenerationen die Implementierung verstehen können.

8 Muster

> Jedes Muster beschreibt ein in unserer Umwelt beständig **wiederkehrendes Problem** und erläutert **den Kern der Lösung** für dieses Problem, so daß Sie diese Lösung **beliebig oft anwenden können**, ohne sie jemals ein zweites Mal gleich auszuführen. [52]
>
> Ein Software-Architektur-Muster beschreibt ein bestimmtes, in einem speziellen Entwurfskontext **häufig auftretendes Entwurfsproblem** und präsentiert ein erprobtes generisches **Schema zu seiner Lösung**. Dieses Lösungsschema spezifiziert die **beteiligten Komponenten**, ihre **jeweiligen Zuständigkeiten, ihre Beziehungen** untereinander und die **Art und Weise, wie sie kooperieren**.[53]

Jede Muster-Form enthält bestimmte Grundelemente:

- Kontext
- Problem
- Lösung
- Beispiel

Gamma et al. haben im Jahr 1996 ein Buch herausgebracht, in dem sie bestehenden Code analysiert und häufige Muster zur Lösung bestimmter Probleme identifiziert haben. Viele dieser Muster besitzen heute noch Gültigkeit, einige sind duch die weiterentwickelten Möglichkeiten der Programmiersprachen obsolet geworden. Zu diesem Buch:

- Katalog von 23 Entwurfsmustern
- Zielsetzung häufig Entkopplung
- Muster enthalten Diagramme zur Erklärung von Struktur und Dynamik
- Beispiel-Code in C++
- 10 Seiten pro Muster

[52] Christopher Alexander et al., 1977
[53] Frank Buschmann et al., 1996

Die Erwartungen:

- Hilfestellung zur Problemlösung
- Wiederverwendung der Problemlösungen
- Reduzierung der Abhängigkeiten
- Software wird flexibel, erweiterbar, wiederverwendbar
- Kommunikationsbasis sowohl in der Design-Phase als auch in der Realisierungs-Phase

Warnung: Software ist nicht dann gut, wenn sie möglichst viele Muster enthält. Muster sind kein Selbstzweck.

8.1 Fabrik

Problem

Die Erzeugung von Objekten hängt oftmals von verschiedenen Umstanden ab. So kann eine Klasse verschiedene Unterklassen haben. Und in jeder Klasse kann es verschiedene Konstruktoren geben.

Lösung

Eine Lösung des Problems kann durch die Implementierung eines „Objekterzeugers", einer Fabrik, sein. Je nach Komplexität des Problems kann diese Fabrik eine einfache statische Methode, ein Objekt oder eine Objekthierarchie sein.

Beispiel

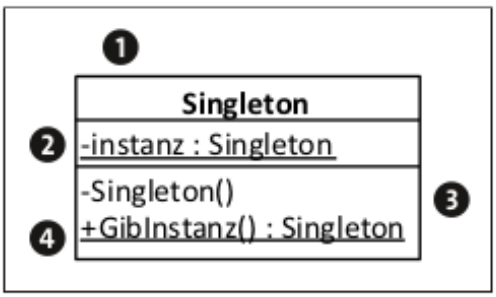
Ein einfaches Beispiel für eine Fabrik haben wir im Abschnitt 5.3 mit der Methode `Erzeugen` bereits implementiert.

8.2 Singleton

Problem

Werden (z.B.) Systemresourcen über Klassen verwaltet, ist es nötig, den Zugriff dergestalt zu regulieren, dass an genau einer Stelle der Zustand der Resource bekannt ist. Während man in früheren Zeiten globale Systemvariablen dafür verwendet hat, bietet sich in einer objektorientierten Umgebung das *Singleton* dafür an.

Lösung



Nr.	Erläuterung
❶	Der Name der Klasse, hier exemplarisch »Singleton« benannt
❷	Der eigentliche Speicher für das Objekt ist ein Klassenattribut. Es ist privat, weil nur die *Singleton*-Klasse selbst es zuweisen darf.
❸	Der private Konstruktor verhindert, dass die *Singleton*-Klasse außerhalb der *Singleton*-Klasse selbst instanziiert werden kann.
❹	Clients greifen auf das *Singleton*-Objekt über diese öffentliche Klassenmethode zu.

Anmerkungen:

- Die Einmaligkeit bezieht sich auf den Speicherbereich. Laufen auf einem System mehrere Instanzen eines Programms, gibt es auch mehrere Singleton-Instanzen.
- Das Sington ist nicht threadsicher. In einer *multi-threaded*-Umgebung müssen die Zugriffe auf die Instanz mit Semaphoren o.ä. abgesichert werden.

Beispiel

Hier sagen ein paar Zeilen Code mehr als viele Worte:

```
class Singleton{
  private static Singleton instance = null;

  private Singleton(){
    //...
  }

  public static Singleton getInstance(){
    if(instance = null){
       instance = new Singleton();
    }
    return instance;
  }
}
```

Aufgabe:

Bei der Instanzierung der Klasse `Medienverwaltung` wird jedesmal die Datei neu gelesen. Wenn es gleichzeitig mehrere Instanzen im System gibt, kann dies zu Inkonsistenzen führen.

Wandeln Sie die Klasse `Medienverwaltung` in ein *Singleton* um.

8.3 Erbauer (Builder)

Problem

> Dieses Pattern scheint, begrifflich gesehen, der Inbegriff aller Erzeugungsmuster zu sein, wird aber in der Praxis eher seltener verwendet. Es kommt immer dann zum Zuge, wenn es gilt, die Erzeugung eines Objekts von dessen Darstellung (Repräsentation) zu trennen.[54]

Lösung

.[Geirhos 2.5.2]

Beispiel

.[Geirhos 2.5.4]

Aufgabe:

Bringen Sie das gezeigte Beispiel zum Laufen.

8.4 Prototyp

Problem

Bei manchen Problemen gibt es zu einem Objekt viele, sich nur in wenigen Eigenschaften unterscheidende Untertypen. In diesem Fall ist es sinnvoller nicht viele formale Untertypen zu definieren, sondern nur wenige Grundtypen. Die Einzelheiten werden dann in Prototypen festgelegt. Dies sind Instanzen, die sinnvoll initialisiert sind und für den eigentlichen Gebrauch kopiert statt instanziert werden.

[54] Geirhos 2015, S. 92.

Lösung

.[Geirhos 2.6.2]

Beispiel

.[Geirhos 2.6.4]

Aufgabe:

Entwickeln Sie das Teilmodell *Medium* (auf dem Papier) weiter: Die Untertypen `Buch` und `CD` sollen derart mit weiteren Attributen versehen werden, dass sich Prototypen für Monographie, Sammelband, Klassik-CD (mehrere Werke, eingespielt von einem Orchester), Band-CD (eine Interpretengruppe) und Sampler (verschiedenste Werke) abgebildet werden können.

8.5 Adapter

Problem

Softwarekomponente A möchte zur Lösung eines bestimmten Problems eine Schnittstelle IA bedienen. Die Softwarekomponente B leistet zwar funktional das Gewünscht, stellt dies aber mit einer Schnittstelle IB bereit. In IA-IB-Adapter stellt nun die Verbindung her: Er implementiert Schnittstelle IA unter Verwendung von Schnittstelle IB.

Lösung

.[Geirhos 3.1.2: Objektadapter]

Beispiel

.[Geirhos: Abb. 3.4]

Aufgabe:

- Zur Pflege von Bestandsdaten sei für den Zugriff eine generische Schnittstelle `ICRUD` mit den Methoden `Create, Read, Update, Delete` definiert (s. `ICRUD.cs`). Schreiben Sie eine KLasse `MedienCrudAdapter`, die `ICRUD` implementiert und dafür die Funktionalität von `Medienverwalter` verwendet.
- Rufen Sie den Code aus dem Hauptprogramm heraus auf.

8.6 Proxy

Problem

Ähnlich wie beim Adapter sollen zwei Komponenten über eine Indirektion miteinander verbunden werden. Im Gegensatz zum Adapter passt bei diesem Muster die Schnittstelle. Aber es soll zusätzliche Funktionalität dazwischen geschoben werden: Zum Beispiel Logging, Sicherheitsüberprüfung, Kommunikation zu einem entfernten System.

Lösung

.[Geirhos 3.7.2]

Beispiel

.[Geirhos 3.7.4]

Aufgabe:

- Packen Sie vor den `MedienCrudAdapter` noch den `MedienCrudLoggingProxy`, der gleichfalls `ICRUD` implementiert. `MedienCrudLoggingProxy` soll alle datenändernden Operationen in einer Datei mitschreiben.
- Rufen Sie den Code aus dem Hauptprogramm heraus auf.

8.7 Zuständigkeitskette (Chain of Responsibility)

Problem

> Für gewöhnlich gibt es in der objektorientierten Kommunikation immer einen Sender (die sendende Klasse) und einen Empfänger (die empfangende Klasse), die eine Operation durchführen. Nicht so bei der Zuständigkeitskette. Dort richtet ein Objekt eine Anfrage an eine Kette von Objekten, und die Anfrage wird entlang dieser Kette weitergeleitet, bis ein Objekt die Anfrage beantwortet.[55]

Lösung

.[Geirhos 4.1.2]

[55] Geirhos 2015: S. 211.

Beispiel

.[Geirhos 3.7.4]

> Aufgabe:
>
> Bringen Sie das gezeigte Beispiel zum Laufen.

8.8 Iterator

Problem

> Beim Iterator geht es darum, die Elemente einer Menge zu durchlaufen, und zwar mithilfe eines Zeigers auf das jeweils nächste Objekt in der Menge.[56]

Lösung

Der *Iterator* ist in C# in die Sprache integriert und heißt *Enumerator* und wurde bereits in Kap. 6.3 besprochen.

8.9 Beobachter (Observer)

Problem

Heutige Benutzeroberflächen arbeiten alle Ereignisorientiert: Jede Eingabe von Tastatur oder Maus führt zu einem Ereignis, das abgearbeitet werden muss. Dazu werden Beobachter (*observer, handler*) zu dem Ereignis registriert. Tritt dann das entsprechende Ereignis auf, werden die Beobachter gerufen.

Lösung

.[Geirhos: Abb. 4.29]

Beispiel

C# weist für dieses Problem eine sprachspezifische Lösung auf (`delegate`, `event`), s. Kap. 6.4.

[56] Geirhos 2015: S. 271.

8.10 Schablone (Template)

Problem

Oftmals liegt die Tücke im Detail: Zwei Anwendungsfälle sind „im Prinzip" gleich, unterscheiden sich aber in kleinen Details. Die identischen Anteile werden dann in einer Schablonenmethode implementiert, die für die Details erst mal abstrakte Methoden aufruft. In abgeleiteten Klassen werden dann diese Detailmethoden implementiert und somit der Algorithmus vervollständigt.

Lösung

.[Geirhos: Abb. 4.39]

Beispiel

.[Geirhos 4.1.2]

9 Quellen

Geirhos 2015	Entwurfsmuster. Das umfassende Handbuch. Rheinwerk 2015
VC2022	Visual C# 2022. Grundlagen der Programmierung. Herdt 2022.