Klassenschema

Wechseln zu:Navigation, Suche

Dieser Artikel erfüllt die GlossarWiki-Qualitätsanforderungen **nur teilweise**:

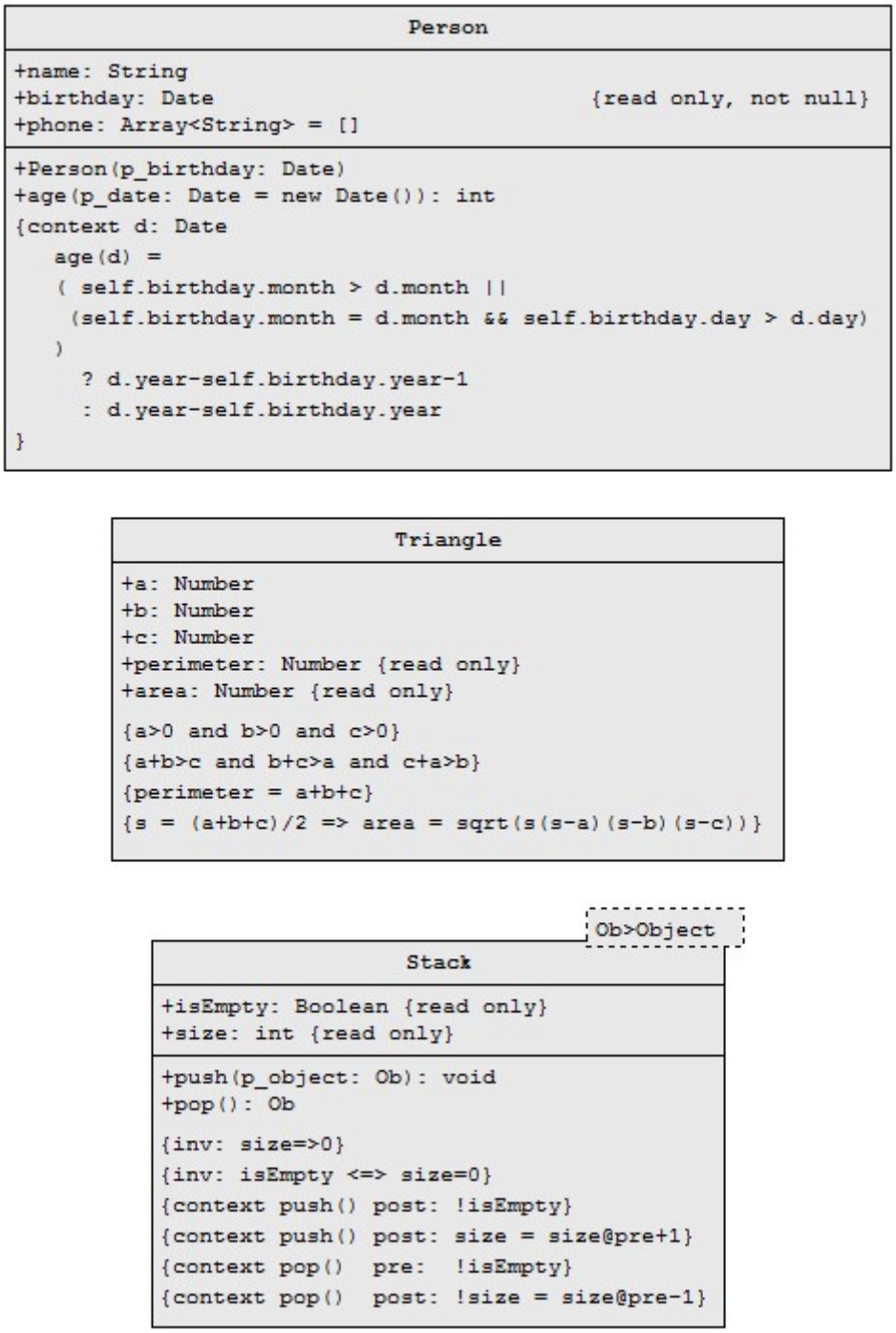
Korrektheit: 3 (zu größeren Teilen überprüft)	**Umfang**: 4 (unwichtige Fakten fehlen)	**Quellenangaben**: 1 (fehlen großteils)	**Quellenarten**: 4 (sehr gut)	**Konformität**: 5 (ausgezeichnet)

Inhaltsverzeichnis

1 Informelle Definition (von W. Kowarschick)[1]
2 Definition (von W. Kowarschick)[2]
3 Bemerkungen
- 3.1 Methoden-Signaturen
- 3.1.1 Beispiel
- 3.1.2 Anmerkung
- 3.2 Zustandsvariablen
- 3.2.1 Anmerkung
- 3.3 Attribute
- 3.3.1 Anmerkung
- 3.4 Methoden-Constraints
- 3.5 Zustandsbeschränkungen
- 3.6 Beziehungsbeschränkungen
- 3.6.1 Anmerkung
- 3.7 Subklassen
- 3.7.1 Anmerkung

4 Quellen
5 Siehe auch

1 Informelle Definition (von W. Kowarschick)[1]



UML-Klassendiagramm: Zwei Klassen und ein Klassentemplate deren Klassenschemata aus zahlreichen Integritätsbedingungen bestehen

Ein Klassenschema (oder eine **Klassenspezifikation**) definiert die Softwareschnittstelle, die alle Objekte der zugehörigen Klasse mindestens unterstützen müssen.

2 Definition (von W. Kowarschick)[2]

Das zu einer Klasse gehörende Klassenschema umfasst eine (endliche) Menge von Integritätsbedingungen, die alle zugehörigen Objekte (d.h. alle in der Extension der zugehörigen

Klasse enthaltenen Elemente) zu jedem Zeitpunkt erfüllen müssen. Das Schema deklariert insbesondere die Methoden, die für jedes Objekt der Klassenextension jederzeit als Kommunikationsschnittstelle zur Verfügung stehen müssen.

3 Bemerkungen

Ein Klassenschema ist laut Definition nichts weiter als eine Menge von Integritätsbedingungen, die jedes Objekt erfüllen muss, dass der zugehörigen Klasse angehört.

Es gibt eine ganze Reihe von Integritätsbedingungen, die ein Klassenschema enthalten kann. Die wichtigsten sind:

Methoden-Signaturen
Zustandsvariablen
Attribute
Methoden-Constraints
Zustandsbeschränkungen
Beziehungsbeschränkungen

3.1 Methoden-Signaturen

Eine Methoden-Signatur ist eine spezielle Integritätsbedingung, bestehend aus Zugriffsbeschränkungen (public, private, protected, internal etc.), Methodennamen, Eingabeparametern samt zugehörigen Datentypen sowie den Datentypen der Methodenergebnisse.

Das Vorhandensein einer Methoden-Signatur in einem Klassenschema definiert eine Invariante: Eine Methode, die dieser Signatur genügt, muss dauerhaft zur Kommunikation mit den Objekten der Klasse, die gemäß der in der Signatur angegebenen Zugriffsbeschränkung Zugriff haben, zur Verfügung stehen.

Außerdem definiert jede Signatur eine Vorbedingung (zum Zeitpunkt des Methodenaufrufes müssen die Argumente den Bedingungen genügen, die in der Signatur an die Eingabeparameter gestellt werden) und eine Nachbedingung (die Ergebnisse von Methodenaufrufen genügen den in der Signatur geforderten Ergebnisdatentypen).

3.1.1 Beispiel

```
public class Person
{
  private var v_birthday: Date;

  // Method "age"
  public function age(p_date: Date = null): int
  {
    if (p_date == null)
      p_date = new Date();

    return (    (v_birthday.month > p_date.month)
            || ((v_birthday.month = p_date.month) &&
                (v_birthday.date  > p_date.date)
               )
           )
           ? p_date.fullYear - v_birthday.fullYear - 1
           : p_date.fullYear - v_birthday.fullYear
  }

  // Constructor
  public function Person(p_birthday: Date)
  {
    v_birthday = p_birthday;
  }
}
```

Jedes Personenobjekt, d.h. jedes Element der Extension der Klasse `Person` stellt eine Methode `age` zur Verfügung,

```
public function age(p_date: Date = null): int
```

auf die jedes (`public`) andere Objekt zugreifen darf:

```
var wolfgang: Person = new Person(new Date(1961, 5, 5));

trace(wolfgang.age());
trace(wolfgang.age(new Date(2011,5,2)));
trace(wolfgang.age(new Date(2011,5,11)));
```

Der Eingabeparameter muss nicht angegeben werden (Defaultwert `null`), aber falls er angegeben wird, muss er vom Typ `Date` sein (Vorbedingung).

```
trace(wolfgang.age('2011-05-11'));  // Fehler (zur Compilezeit),
                                    // da die Vorbedingung
                                    // „das Argument ist vom Typ Date“
                                    // nicht erfüllt ist.
```

Die Methode `age` liefert eine Integerzahl zurück (Nachbedingung).

Dass `age` (als Nachbedingung!) das Alter der aktuellen Person zu einem bestimmten Datum bzw. zum aktuellen Datum (falls `p_date == null`) ermittelt, lässt sich der Signatur dagegen nicht entnehmen. Hierfür wäre die Angabe von weiteren Integritätsbedingungen notwendig, was aber nur von wenigen Sprachen, wie z.B. Eiffel, unterstützt wird. Eine weitere (deutlich schwächere) Nachbedingung wäre z.B., dass das Resultat positiv sein muss, wenn das übergebene Datum `p_date` größer ist, als das Geburtsdatum. Diese Bedingung wird z.B. verletzt, wenn `p_date` so groß gewählt wird, dass das Alter größer ist als die größte darstellbare Integerzahl. Um gerade vor solchen Überraschungen gefeit zu sein, ist der Einsatz von Integritätsüberprüfungen, die über die Angabe von Signatur-Informationen hinaus gehen, sehr empfehlenswert.

Bislang wurde gezeigt, dass die Signatur der Methode `age` eine Vor- und eine Nachbedingung formuliert. Sie formuliert allerdings auch eine Invariante: Die Methode `age` existiert dauerhaft, d.h. sie kann nicht entfernt oder verändert werden:

```
wolfgang.age = null; // Fehler (zur Compilezeit)
```

Dies ist nicht so selbstverständlich, wie es zunächst scheint. Wäre (in ActionScript) die Klasse Person als dynamisch deklariert worden

```
public dynamic class Person
{
  ...
}
```

so könnten jedem Personenobjekt jederzeit neue Methoden zugefügt werden. Und diese können auch wieder entfernt werden:

```
wolfgang.ageChristmas2011 = function (): int { return wolfgang.age(new
Date(2011,12,24)); };

trace(wolfgang.ageChristmas2011());

wolfgang.age = null;                // immer noch ein Fehler (zur
Compilezeit)
wolfgang.ageChristmas2011 = null;   // kein Fehler

trace(wolfgang.ageChristmas2011()); // jetzt ein Fehler (zur Laufzeit),
                                    // ageChristmas2011 existiert nicht
mehr
```

3.1.2 Anmerkung

Im Falle von Methoden und Funktionen, die den Praogrammzustand nicht ändern, d.h. im Falle von Anfragemethoden oder seiteneffektfreien Funktionen kann man die Datentypen einer Signatur nicht nur (so wie im obigen Beispiel) als Beschreibung einer Vor- und einer Nachbedingung auffassen, sondern auch als Beschreibung einer Invarianten.

Es sei [math]f: A \times B \rightarrow C[/math] eine Signatur.

Aus prozeduraler Sicht steht [math]A \times B[/math] für folgende *Vorbedingung*: Vor Aufruf der Funktion/Methode [math]f\,[/math] müssen je ein Objekt/Wert vom Typ [math]A\,[/math] und vom Typ [math]B\,[/math] der Funktion als aktuelle Argumente übergeben werden (z.B. indem diese Objekte in richtiger Reihenfolge auf den Programmstack gelegt werden). Und [math]C\,[/math] steht für folgende *Nachbedingung*: Nachdem die Funktion/Methode [math]f\,[/math] bearbeitet wurde, liefert sie ein Objekt/Wert vom Typ [math]C\,[/math] als Ergebnis (z.B., indem sie die Argumente vom Programmstack löscht und ein passendes Ergebnisobjekt auf den Programmstack ablegt).

Aus funktionaler Sicht steht die obige Signatur für folgende *Invariante*:

> $\forall a \in A, b\in B: f(a,b) \in C$ oder auch $\forall a, b: a \in A \wedge b\in B \Rightarrow f(a,b) \in C$

Das heißt, [math]f\,[/math] erfüllt folgende Bedingung **dauerhaft**: Wenn [math]a\,[/math] ein Element von [math]A\,[/math] ist und [math]b\,[/math] ein Element von [math]B\,[/math], dann ist [math]f(a,b)\,[/math] ein Element von [math]C\,[/math].

Man kann diese Invariante noch um eine zweite Invariante ergänzen:

> $\forall a, b: a \notin A \vee b\notin B \Rightarrow f(a,b)$ throws error

Das heißt, wenn entweder [math]a\,[/math] kein Element von [math]A\,[/math] ist oder [math]b\,[/math] kein Element von [math]B\,[/math], dann meldet [math]f(a,b)\,[/math] einen Fehler.

Ob man die Datentypen einer Signatur eher als Vor- und Nachbedingung oder als Invarianten auffasst, ist Geschmackssache. Wichtig ist, wie man sicherstellt, dass alle Integritätsverletzungen erkannt werden, um geeignete Gegenmaßnahmen zu ergreifen zu können. In einer Programmiersprache mit starker Typisierung erkennt ein Compiler viele Verletzungen von Integritätsbedingungen schon zur Übersetzungszeit. Allerdings muss der Programmierer sogar in derartigen Sprachen häufig zur Laufzeit sicherstellen (z.B. mittels Assert-Befehlen), dass alle Bedinungen eingehalten werden. Beispielsweise muss bei der Division

```
operator/ (dividend: double, divisor: double\{0}): double
```

i. Allg. zur Laufzeit überprüft werden, ob die Bedingung `divisor!=0` erfüllt ist.

3.2 Zustandsvariablen

Die Deklaration einer Zustandsvariablen kann ebenfalls als Integritätsbedingung aufgefasst werden:

```
public var birthday: Date;
```

Diese Deklaration legt beispielsweise fest, dass jedes (`public`) Objekt jederzeit lesend und schreibend auf die Variable `birthday` zugreifen darf. Dabei muss sie allerdings beachten, dass in dieser Zustandsvariablen nur Werte vom Typ `Date` abgelegt werden dürfen.

3.2.1 Anmerkung

In Sprachen wie Java, die keine Attribute unterstützen, werden öffentlich zugängliche Zustandsvariablen häufig als Attribute missbraucht.

3.3 Attribute

Die Definition eines Attributes, wie z.B. `birthday`, im Klassenschema sollte als Definition von zwei speziellen Methoden aufgefasst werden (deren Signaturen wiederum spezielle Integritätsbedingungen sind).

Zum einen gibt es eine Getter-Methode zum Lesen des Attributwertes:

```
public function get birthday(): Date
```

Zum anderen gibt es eine Setter-Methode zum Modifizieren des Attributwertes:

```
public function set birthday(p_birthday: Date): void
```

Bei Read-only-Attributen (Integritätsbedingung `{read only}`) fehlt die Setter-Methode und bei Add-only-Attributen (Integritätsbedingung `{add only}`) fehlt die Getter-Methode. Im Falle des Geburtstages ist es z.B. sinnvoll, das Geburtsdatum nur beim Erzeugen des Objektes mit Hilfe des Konstruktors zu initialisieren und daher auf eine Setter-Methode zu verzichten.

3.3.1 Anmerkung

In Sprachen wie Java, die keine Attribute unterstützen, sollte man Methoden wie `getBirthday` und `setBirthday` zur Simulation von Getter- und Setter-Methoden verwenden.

3.4 Methoden-Constraints

Für Methoden können neben der Signatur weitere Invarianten sowie Vor- und Nachbedingungen spezifiziert werden. Für die Modifikationsmethoden `push` und `pop` der Klasse `Stack` wurden im obigen Klassendiagramm beispielsweise mehrere Vor- und Nachbedingungen angegeben.

Nachdem ein Element auf den Stack gelegt wurde, ist dieser nicht leer und enthält ein Element mehr als zuvor:

```
{context push() post: !isEmpty }
{context push() post: size = size@pre+1}
```

Bevor ein Element vom Stack herunter geholt werden kann, muss dieser nicht leer sein. Danach enthält er ein Element weniger als zuvor.

```
{context pop()  pre:  !isEmpty}
{context pop()  post: !size = size@pre-1}
```

Für die Anfragemethode `age` der Klasse Person wurde die Semenatik mittels einer Invarianten spezifiziert.

Sofern eine Person im als Argument übergebenen Jahr `d` noch nicht Geburtstag gehabt hat, ermittelt man das Alter, indem man das Geburtsjahr und noch ein Jahr vom übergebenen Jahr abzieht. Hatte die Person zum angegeben Zeitpunkt schon Geburtstag, muss lediglich das Geburtsjahr vom übergebenen Jahr abgezogen werden.

```
{context d: Date
   age(d) =
   ( self.birthday.month > d.month ||
    (self.birthday.month = d.month && self.birthday.day > d.day)
   )
     ? d.year-self.birthday.year-1
     : d.year-self.birthday.year
}
```

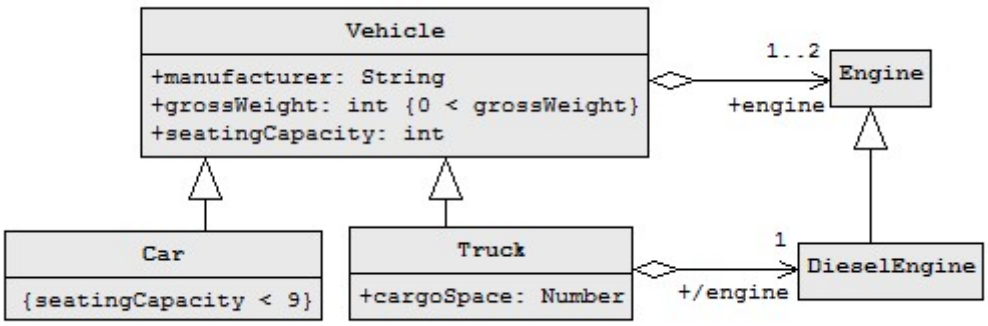
3.5 Zustandsbeschränkungen

Neben Methodensignaturen und -Constraints, Attributen und Zustandsvariablen können weitere Integritätsbedingungen im Klassenschema angegeben werden. Man kann Integritätsbeziehungen formulieren, die die erlaubten Zustände der Objekte einschränken. Diese Integritätsbedingungen haben vor allem auf Attribute Auswirkungen.

Beispielsweise kann es keine Dreiecke mit den Seiten `a`, `b` und `c` geben, bei denen die Summe von zwei Seiten kleiner ist, als die dritte Seite:

```
{a+b>c and b+c>a and c+a>b}
```

3.6 Beziehungsbeschränkungen



UML-Klassendiagramm: Eine Klassenhierachie mit Beziehung zu einer zweiten Klassenhierarchie

Beziehungen zwischen Klassen beschreiben ebenfalls Integritätsbedingungen. Allerdings gibt es mehrere Möglichkeiten, eine Beziehung zu implementieren. Zum einen kann dies durch eigenständige Beziehungsobjekte geschehen, zum anderen ist es auch möglich, Beziehungen als Attribute der beteiligten Klassen zu implementieren. Dabei erhält jede an der Beziehung beteiligte Klasse ein eigenes Attribut, sofern dies für bestimmte Klassen nicht explizit ausgeschlossen wird. Dieser Fall soll hier betrachtet werden.

Im Allgemeinen sind zwei Klassen an einer Beziehung beteiligt. In diesem Fall kann man bidirektionale und unidirektionale Beziehungen unterscheiden: Im ersten Fall erhalten beide beteiligte Klassen ein Beziehungsattribut, im zweiten Fall nur eine.

Im nebenstehenden Beispiel wurde zwischen `Vehicle` und `Engine` eine unidirektionale (erkenntlich an der Pfeilspitze) Part-of-Beziehung (erkenntlich an der Raute) `engine` definiert. Dies bedeutet, dass der Klasse `Vehicle` ein Attribut `engine` vom Typ `Set<Engine>` hinzugefügt werden muss. Die Menge muss (wegen der Vielfachheit `1..2`) ein bis zwei Objekte vom Typ `Engine` enthalten, z.B. einen Ottomotor und einen Elektromotor.

3.6.1 Anmerkung

Im nebenstehenden Diagramm sind zwei Arten von Beziehungen dargestellt. Eine Vererbungsbeziehung besteht nur zwischen Klassen, aber nicht zwischen den zugehörigen Objekten. Eine Part-of-Beziehung (Aggregation) oder – allgemeiner – eine Assoziation besteht dageben zwischen Objekten. Wenn eine derartige Beziehung in einem Klassendiagramm angegeben wird, beschreiben sie (als spezielle Integritätsbedingungen), welche Beziehungen zwischen den zugehörigen Objekten bestehen müssen.

Die Angabe von Vielfachheiten oder Multiplizitäten macht in einem Objekt-Diagramm keinen Sinn (daher werden insbesondere in einem Klassendiagramm bei einer Vererbungsbeziehung keine Vielfachheiten angegeben; hier werden die beiden beteiligten Klassen als Objekte einer Metaklasse `Class` aufgefasst). Die Angabe von Vielfachheiten bei einer Assoziation in einen Klassendiagramm legt fest, wie viele entsprechende Objektbeziehungen in einem zugehörigen Objektdiagramm

existieren müssen. Eine Vielfachheit ist also ebenfalls eine spezielle Integritätsbedingung.

3.7 Subklassen

Subklassen erben laut Definition alle Integritätsbedingungen von ihrer Superklasse und können weitere Integritätsbedingungen hinzufügen. In der zuvor definierten Klassenhierachie erbt beispielsweise die Klasse `Car` alle Integritätsbedingungen der Klasse `Vehicle`. Für ein Auto gibt es also auch einen Hersteller (`manufacturer`), ein zulässiges Gesamtgewicht (`grossWeight`) das nicht gleich Null oder gar negativ sein darf, eine Anzahl von Sitzplatzen `seatingCapacity` sowie ein oder zwei Motoren (auch die Beziehung zur Klasse `Engine` wird vererbt).

In der Klasse `Car` wird eine weitere Integritätsbeziehung hinzugefügt: Die Anzahl der der Sitze darf höchstens acht betragen.

Auch in der Klasse `Truck` werden weitere Integritätsbedingungen hinzugefügt: Zum einem muss jedes Objekt dieser Klasse zusätzlich eine Ladefläche (`cargoSpace`) enthalten und zum anderen hat jeder Lastwagen (laut obiger Spezifikation!) genau einen Motor: Die Vielfachheit des geerbten Beziehungsattributs `engine` wird eingeschränkt. (Die Tatsache, dass diese Beziehung geerbt wurde, wir im UML-Diagramm mit einen Schrägstrich `/` vor dem Rollennamen symbolisiert.)

3.7.1 Anmerkung

Zusätzliche Integritätsbedingungen bedeuten immer Einschränkungen, das gilt auch für zusätzliche Attribute und Operationen. Wenn beispielsweise ein zusätzliches Attribut angegeben wird, bedeutet dies, dass jedes Objekt der Klassenextension dieses Attribut anbieten muss. Fehlt die Angabe dieses Attributs in der Klassenextension, so heißt dies jedoch nicht, dass ein zugehöriges Objekt kein entsprechendes Attribut haben darf. Beispielsweise muss jedes Objekt der Klasse `Truck` ein Attribut `cargoSpace` haben, Objekte der Klasse `Vehicle` können dagegen so ein Attribut haben, müssen dies aber nicht. Ja sogar Autos (d.h. Objekte vom Typ `Car`) könnten dieses Attribut haben. Entweder wird es bei Bedarf dem ein oder anderen Auto dynamisch zur Laufzeit zugewiesen oder es wird in einer Subklasse (z.B. `Van`) der Klasse `Car` für spezielle Autos verpflichtend eingeführt.

4 Quellen

Kowarschick (MMProg): Wolfgang Kowarschick; Vorlesung „Multimedia-Programmierung“; Hochschule: Hochschule Augsburg; Adresse: Augsburg; Web-Link; 2018; Quellengüte: 3 (Vorlesung)
Kowarschick (MMProg): Wolfgang Kowarschick; Vorlesung „Multimedia-Programmierung“; Hochschule: Hochschule Augsburg; Adresse: Augsburg; Web-Link; 2018; Quellengüte: 3 (Vorlesung)

5 Siehe auch

Kowarschick (2002a): Wolfgang Kowarschick; Multimedia-Programmierung – Objektorientierte Grundlagen; Hrsg.: Michael Lutz und Christian Märtin; Reihe: Informatik interaktiv; Verlag: Fachbuchverlag Leipzig im Carl Hanser Verlag; ISBN: 3446217002; 2002; Quellengüte: 5 (Buch)

Kategorien:
Objektorientierte Programmierung
Glossar

Kapitel:Multimedia-Programmierung