Zusammenfassung M226

Objektorientiert implementieren

2008-11-12

Emanuel Duss

Über

Autor	Emanuel Duss
Erstellt	2008-08-03
Bearbeitet	2008-11-12
Heute	2008-11-12
Bearbeitungszeit	10:46:57
Lehrjahr des Moduls	1. Lehjahr 2006/2007
Pfad	/ home/emanuel/Daten/Lehre/Zwischenprüfungen/Zusammenfassungen_von_mir/ M226/M226_Zusammenfassung.odt
CC-Lizenz	 Creative Commons Namensnennung-Keine kommerzielle Nutzung-Weitergabe unter gleichen Bedingungen 2.5 Schweiz http://creativecommons.org/licenses/by-nc-sa/2.5/ch/
Powered by	

Bearbeitungsprotokoll

Datum	Änderung(en)
2008-08-03	Erstellt
2008-08-27	Ergänzungen und Fertigstellung

Inhaltsverzeichnis

Modulbaukasten

Modulidentifikation

Modulnummer	226
Titel	Objektorientiert implementieren
Kompetenz	Ein objektorientiertes Design (OOD) implementieren, testen und dokumentieren.
Handlungsziele	1. Ein objektorientiertes Design (OOD) nachvollziehen und durch technische Klassen ergänzen. 2. Dynamische und statische Strukturen zwischen Objekten resp. Klassen mittels Unified Modeling Language, UML (Klassen-/Sequenzdiagramme) darstellen. 3. Objektorientiertes Design implementieren. 4. Klassen systematisch prüfen (Unit Test). 5. Klassen- und Systemdokumentation vervollständigen.
Kompetenzfeld	Application Engineering
Objekt	Applikation mit 3-5 fachlichen Klassen (z.B. Karteikasten, CD-Sammlung, Adressverwaltung usw.).
Niveau	2
Voraussetzungen	• Analysieren und objektbasiert programmieren mit Komponenten, bzw. • Analysieren und strukturiert implementieren
Anzahl Lektionen	80
Anerkennung	Eidg. Fähigkeitszeugnis Informatiker/Informatikerin
Modulversion	2.1
MBK Release	R3
Harmonisiert am	16.09.2005

Handlungsnotwendige Kenntnisse

Handlungsnotwendige Kenntnisse beschreiben Wissenselemente, die das Erreichen einzelner Handlungsziele eines Moduls unterstützen. Die Beschreibung dient zur Orientierung und hat empfehlenden Charakter. Die Konkretisierung der Lernziele und des Lernwegs für den Kompetenzerwerb sind Sache der Bildungsanbieter.

Modulnummer	226
Titel	Objektorientiert implementieren
Kompetenzfeld	Application Engineering
Modulversion	2.1
MBK Release	R3

Handlungsziel	Handlungsnotwendige Kenntnisse
1.	1. Kennt das Paradigma des objektorientierten Ansatzes und kann an Beispielen erläutern, welche prinzipiellen Unterschiede gegenüber dem funktionalen Ansatz bestehen. 2. Kann aufzeigen, wie durch Klassen und deren Attribute und Methoden die reale Welt im (vorgegebenen) Design abgebildet wird. 3. Kennt die Beziehungstypen (Assoziation, Aggregation, Komposition) zwischen Klassen und das Konzept der Vererbung und kann aufzeigen, wie diese umgesetzt werden. 4. Kennt die Bedeutung der technischen Klassen und kann an Beispielen erläutern, wie diese zusammen mit den fachlichen Klassen das objektorientierte Design vervollständigen.
2.	1. Kennt die Notation des Klassen- und Objektdiagramms und kann aufzeigen, wie sich diese unterscheiden. 2. Kennt die Notation von Sequenz- und Kollaborationsdiagramm und kann aufzeigen, wie sich damit Abläufe darstellen lassen. 3. Kennt die Notation von Schnittstellen und Paketen und kann aufzeigen, wie sich diese auf die Implementation (Typisierung, Programmierung im Team usw.) auswirken.
3.	1. Kann erläutern, wie Klassenmodelle mit einer objektorientierten Programmiersprache umgesetzt werden können. 2. Kann an einem Codebeispiel den Effekt von Polymorphie aufzeigen. 3. Kann aufzeigen, wie durch die Nutzung von Schnittstellen der Code unabhängig erstellt und getestet werden kann. 4. Kennt die grundlegenden Funktionen eines CASE Tools und kann erläutern, mit welchen Funktionen die Implementation unterstützt wird.
4.	1. Kennt die grundlegenden Schritte, die bei einem Unit Test durchlaufen werden müssen und kann aufzeigen, welchen Beitrag diese zu einem qualitativ guten Ergebnis leisten. 2. Kann zu einer Klasse Testfälle und Grenzwerte festlegen und diese in einer Testklasse implementieren.
5.	1. Kann den Programmcode vollständig und korrekt dokumentieren, um daraus die API Spezifikation abzuleiten. 2. Kennt die Struktur einer Systemdokumentation und kann ihre Bedeutung für Wartung und Nachvollziehbarkeit darlegen.

1 Grundlagen OOP

1.1 Strukturiert vs. Objektorientiert

- Bis in die 80er-Jahren wurde strukturiert Programmiert. Die Gesamtaufgabe wird in Funktionen zerlegt. Das Programm beginnt in der Main-Funktion.
  - Das Programm ist beendet, wenn die Main-Funktion beendet ist.
- Heutzutage sind alle gängigen Programmiersprachen objektorientiert. Prinzipiell sind diese Aufgaben auch mit der strukturierten Programmierung zu lösen.
  - Das Programm ist beendet, wenn es keine Objekte mehr gibt.
- Viele Vorteile liegen in der Vererbung.
- Daten können geschützt werden (Datenkapselung)
- Daten und Unterprogramme gehören zusammen. Diese Bilden eine Einheit (Objekt).
- Redundanter Programmiercode kann mit der Vererbung eliminiert werden.

1.2 Begriffe der OOP

Daten	Das sind die Eigenschaften eines Objekts. Ein Auto hat z.B.eine Farbe, Geschwindigkeit etc.
Eigenschaften	Daten eines Objekts.
Grundsatz der OOP	Vgl. Datenkapselung.
Instanz	Andere Bezeichnung für ein Objekt. Aus einer Klasse kann man eine Instanz machen. D.h. wir erstellen aus der Klasse ein Objekt.
Instanziierung	Wenn man ein Objekt erzeugt macht man eine Instanziierung.
InstanzVariablen	Instanzvariablen werden in einer Klasse aber nicht in einer Methode definiert. Auch MemberVariablen oder Eigenschaften genannt. Diese besitzen eine Sichtbarkeit (private, public)
Klasse	„Bauplan“ eines Objekts. Es beschreibt was das Objekt kann und was es für Eigenschaften besitzt. Es können beliebig viele Objekte erstellt werden. Während der Programmierung existieren keine Objekte, nur Klassen.
Klassenmethoden, statische Methoden	Sie existieren genau einmal, ganz egal ob kein, ein oder hunderte von Objekten der Klasse erzeugt wurden.
Klassenvariablen, statische Eigenschaften	Sie existieren genau einmal, ganz egal ob kein, ein oder hunderte von Objekten der Klasse erzeugt wurden.
Konstruktor	Methode, die beim erzeugen eines Objekts aufgerufen werden kann. Konstruktoren haben keine Rückgabewert. Diese setzen Eigenschaften auf einen bestimmten Wert (Default-Wert) und initialisieren Daten des Objekts. Man muss nicht zwingend für jede Klasse einen Konstruktor erstellen. Der Compiler reklamiert nicht!
Lokale Variablen	Eigenschaften leben während der ganzen Lebensdauer des Objekts. Lokale Variablen nicht. Besitzen keine Sichtbarkeit.

	Lokale Variablen gehören zu Methoden und leben nur während dem Aufruf der Methode.
Member	Eigenschaft
Memberfunktionen	Methoden eines Objekts.
MemberVariablen	Eigenschaften eines Objekts.
Methode	Funktion eines Objekts. z.B. LieferungAusloesen(), RechnungErstellen(), etc...
Objekt, Instanz	Eine Instanz einer Klasse. Besteht aus Eigenschaften und Methoden. Während der Programmausführung entstehen Instanzen (Objekte) aus den Klassen. z.B. BestellNr, BestellDatum, etc...
priavte	Auf private Eigenschaften und Methoden kann von ausserhalb der Klasse NICHT zugegriffen werden.
Properties (Get/Set-Methoden)	Eigenschaften herauslesen bzw. setzen. Die Daten werden über diese Properties gekapselt.
Protected	Auf protected Eigenschaften und Methoden können die erbenden Klassen zugreifen. Von ausserhalb kann man jedoch nicht zugreifen.
pulbic	Auf public Eigenschaften und Methoden kann von ausserhalb der Klasse zugegriffen werden.
Standardkonstruktor	Methode, die beim erzeugen eines Objekts standardmässig aufgerufen wird. Dabei wird beim erzeugen kein Argument mitgegeben.

1.3 Datenkapselung

- Mit der Datenkapselung können wir den Zugriff auf Eigenschaften und Methoden einschränken bzw. sperren.
- Eigenschaften (Daten) sollten immer privat sein. Diese dürfen nur über öffentliche (public) Methoden angesprochen werden. Dies nennt man auch Schnittstelle.
- Es darf also nur eine public Methode eine private Eigenschaft bearbeiten können.

1.4 Klassen erstellen

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

namespace NamespaceName
{
    //Klasse
    public class Klassenname
    {
        //Membervariablen
        private int m_ID;
        private string m_Name;

        //Konstruktoren
        public CPerson() { <blabla> }
        public CPerson(int ID, string Name, string Vorname, string Strasse)
{ ...}
```

```
        //get- set Methoden
        public int getID(){ return m_ID; }
        public void setID(int ID){ m_ID = ID; }
        public string getName(){ return m_Name; }
        public void setName(string Name){ m_Name = Name; }

        //Methoden
        public void Ausgabe() { ... }
        }
    }
}
```

1.5 Konstruktoren / Dekonstruktoren / Standardkonstruktor

- Methode, die beim erstellen eines Objekts aufgerufen wird.
- Zweck: Werte initialisieren, Werte überprüfen
- Die Konstruktoren besitzen den selben Namen die die Klasse selbst. Heisst also die Klasse „Auto“ heissen die Konstruktoren auch „Auto“.
- Es darf kein Rückgabetyp angegeben werden.
- Die Konstruktoren können public, private oder keinen Zugriffsmodifizierer haben.
- Wenn man mehrere Konstruktoren hat, dann hat man überladene Konstruktoren.
- Man kann mehrere Konstruktoren haben. Diese müssen sich aber in der Signatur (Reihenfolge und Datentypen der Parameterliste) unterscheiden.
- Wenn beim Konstruktor keine Argumente verlangt, ist es der Standardkonstruktor.

Die Klasse Auto	`class Auto {`
	`  //Standardkonstruktor`
	`  public Auto(){`
	`  start(0);`
	`  }`
	`  //Konstruktor`
	`  public Auto(int maxV, int aktV){`
	`  setze(maxV, aktV);`
	`  }`
	`}`

Objekt erzeugen (Standardkonstruktor)	`Auto a = new Auto();`
Objekt erzeugen (Konstruktor)	`Auto a = new Auto(250, 50);`

- Ein Dekonstruktor wird aufgerufen, wenn das Objekt freigegeben wird. Bei modernen Sprachen werden die Objekte automatisch von selbst zerstört. Diese automatische Zerstörung wird vom Garbage-Collector ausgeführt.

1.6 Properties (get- / set-Methoden)

Die Klasse Auto	`class Auto {`
	
	`private int m_MaxGeschw;`
	
	`public int getMaximalGeschwindigkeit (){`
	`return m_MaxGeschw;`
	`}`
	`public void setMaximalGeschwindigkeit(int v){`
	`m_MaxGeschw = v;`
	`}`
	`}`

Objekt erzeugen	`Auto a = new Auto();`
Eigenschaft setzen	`a.setMaximalGeschwindigkeit (210);`
Eigenschaft aufrufen	`MsgBox.show(a.getMaximalGeschwindigkeit());`

1.7 Statische Eigenschaften, statische Methoden

Auch genannt als Klassenvariablen oder Klassenmethoden.

- Statische Eigenschaften werden beim Programmstart erstellt.
- Sie existieren genau einmal, ganz egal ob kein, ein oder hunderte von Objekten der Klasse erzeugt wurden.
- Statische Methoden dürfen nur auf statische Eigenschaften zugreifen.
- Auf statische Elemente kann man ohne Instanz zugreifen.
- Die Main-Methode ist standardmässig static.

Beispiel

```
class cAuto
{
      public int m_Geschw;
      public static int s_AnzahlAutos;
      public static TuEtwas( ... );
}
```

- Diesen Wert könnte man bei jedem erzeugen eines Autos um 1 erhöhen, damit man weiss, wie viele Autos existieren.

1.8 Modifizierer

Quelle: http://www.guidetocsharp.de/Old/referenz/protecte.html

Modifizierer	innerhalb der Klasse	innerhalb der Ableitungen	ausserhalb der Klasse	ausserhalb der Anwendung
public	ja	ja	ja	ja
internal	ja	ja	ja	nein
protected	ja	ja	nein	nein
private	ja	nein	nein	nein

1.9 Typisches Vorgehen beim erstellen von Klassen in der OOP

1. Klassen erstellen
2. Eigenschaften erstellen
3. Standardkonstruktor erstellen
4. Get-Set-Methoden erstellen
5. Konstruktoren mit Parameter erstellen
6. Methoden erstellen
7. Statische Eigenschaften erstellen
8. Statische Methoden erstellen

1.10 Objekte in Liste aufnehmen

Erstellen der Liste	`private List<Person> m_Personen = new List<Person>();`
Objekt erstellen	`Person p = new Person();`
Eigenschaften setzen	`p.setName(„Karl Koch“);`
Objekt der Liste hinzufügen	`m_Personen.Add(p);`

1.11 Der Stack und der Heap

- **Heap**: Bereich, in dem alle Objekte leben
- **Stack**: Hier leben Methoden und lokale Variablen

1.11.1 Methoden stapeln

- Wenn eine Methode aufgerufen wird, dann kommt diese auf den Stack.
- Die oberste Methode auf dem Stack ist immer die, die aktuell ausgeführt wird.
- Nach der Bearbeitung wird dieses „Frame“ abgeschoben.

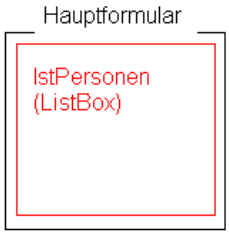
Funktionsname() VariableA, VariableB, VariableC
foo() a, b, s
bar() s
main()

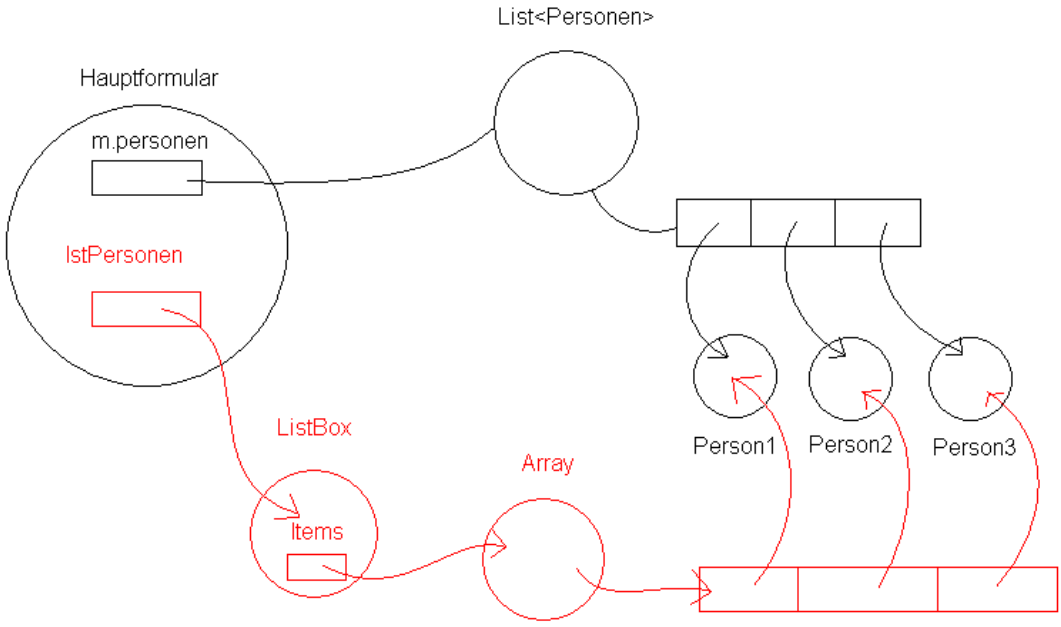
1.11.2 Objekte auf dem Heap

```
public class StackRef{
    public void foof(){
        barf();
    }
    public void barf(){
        Ente e = new Ente(24);
    }
}
```

Instanzvariablen zeichnet man im Objekt (Ente).

Beispiel





Grafik by SRRL

2 Vererbung / Polymorphie

Alle Elemente einer Klasse können an eine andere Klasse vererbt werden. Die Klasse, welche vererbt, nennt man Basis- oder auch Superklasse. Die Klasse, die von der Superklasse erbt, kann man noch um weitere Eigenschaften oder Methoden erweitern oder bestehende Methoden überschreiben.

Die abgeleitete Klasse kann also mindesten so viel wie die Superklasse. Mit dem Vererben findet also eine Spezialisierung statt.

2.1 Vererben

Ein Manager ist eine spezielle Form von einem Mitarbeiter. Deshalb erbt der Manager alle Eigenschaften und Methoden von einem Mitarbeiter.

```
public class Mitarbeiter
{
      private int m_PersNr;
      private string m_Name;
      private double m_Monatslohn;

      public int getPersNr() {return m_PersNr;}
}

public class Manager : Mitarbeiter
{
      private double m_Bonus;

      public double getBonus() {return m_Bonus;}
}
```

Auf privaten Eigenschaften der Basisklasse können von der Vererbten Klasse nicht zugegriffen werden. Man könnte also nicht in der Manager-Klasse „return m_PersNr;“ machen!

Wenn von der erbenden Klasse auf eine Membervariable der Basisklasse zugegriffen werden will, vergibt man der Membervariable die Sichtbarkeit protected.

2.2 Konstruktoren aufrufen

Der Standardkonstruktor der Basisklasse wird immer als erstes aufgerufen!

Besitzt die Basisklasse nur Konstruktoren mit Parametern, dann kann der mit dem Standardkonstruktor der erbenden Klasse dem Schlüsselwort :base aufgerufen werden:

```
public Zahl2() : base(parameter_vom_konstruktor_der_basisklasse){}
```

```
class Program
{
      public static void Main(string[] args)
      {
            OnlineShop s = new OnlineShop();
            OnlineShop o = new OnlineShop("Name");
            Console.ReadKey(true);
      }
}
class Shop
```

```
{
	protected string m_Name;

	public Shop()
	{
		Console.WriteLine("Basisklasse: Keine Argumente");
	}
	public Shop(string name)
	{
		m_Name = name;
		Console.WriteLine("Basisklasse: Mit Argumente");
	}
}
class OnlineShop : Shop
{
	public OnlineShop() : base()
	{
		Console.WriteLine("Abgeleitete Klasse: Keine Argumente");
	}
	public OnlineShop(string name) : base(name)
	{
		Console.WriteLine("Abgeleitete Klasse: Mit Argumente");
	}
}
```

Ausgabe

```
Basisklasse: Keine Argumente
Abgeleitete Klasse: Keine Argumente
Basisklasse: Mit Argumente
Abgeleitete Klasse: Mit Argumente
```

Es werden also immer beide Konstruktoren aufgerufen. Zuerst der Konstruktor der Basisklasse und dann der Konstruktor der abgeleiteten Klasse.

2.3 Methoden der Basisklasse aufrufen

Man kann mit dem Schlüsselwort base eine Methode oder einen Konstruktor aus einer Methode heraus aufrufen:

```
public B : A
{
	public Zahl(int foo, string bar)
	{
		base();
		base.MethodeAusKlasseA(foo);
		base.AndereMethodeAusKlasseA(bar);
	}
}
```

2.4 Polymorphismus

Polymorphie (Vielgestaltigkeit) beschreibt die Fähigkeit, eine gleichartige Operation mit dem gleichen Namen für Objekte verschiedener Klassen aufzurufen. Beim Aufruf dieser Operation muss nur bekannt sein, dass ein Objekt diese Operation kennt, aber nicht, von welcher Klasse das Objekt erzeugt wurde.

2.4.1 Member überschreiben

Methoden, Eigenschaften und auch Indexer können in abgeleiteten Klassen überschrieben werden. Dies muss jedoch bei der Deklaration der Basisklasse freigegeben werden.

Rückgabetypen und Parameter müssen übereinstimmen!

```
class Klasse1
{
      // Die Methode wird mit virtual als überschreibbar deklariert
      public virtual void print() {Operation01;}
}
class Klasse2 : Klasse 1
{
      // Die Methode wird mit override überschrieben
      public override void print() {AndereOperation;}
}
```

2.4.2 Abstrakte Klassen und Methoden

Quelle: http://www.galileocomputing.de/openbook/csharp/

- Von abstrakten Klassen können keine Objekte instanziiert werden.
- Eine Abstrakte Klasse ist eine Klasse mir der die Funktionen beschrieben werden, die eine Abgeleitete Klasse implementieren muss.
- Abstrakte Funktionen sind automatisch virtuelle Funktionen.

```
public abstract class MusicServer
{
    public abstract void Play();
}
public class WinAmpServer: MusicServer
{
    public override void Play()
    {
        Console.WriteLine("WinAmpServer.Play()");
    }
}
public class MediaServer: MusicServer
{
    public override void Play()
    {
        Console.WriteLine("MediaServer.Play()");
    }
}
public static void Main()
{
      MusicServer ms = new WinAmpServer();
      ms.Play();  // Ausgabe: WinAmpServer.Play()
      ms = new MediaServer();
      ms.Play();  // Ausgabe: MediaServer.Play()
}
```

3 UML

3.1 Klassendiagramm

3.1.1 Klassen

Klassenname (abstract:Kursiv)
+Public Variablen : Datentyp = Defaultwert - Private Variablen : Datentyp # Protected Variablen : Datentyp = Defaultwert + Public Variable (static: unterstrichen)
+Public Methoden() : Rückgabewert - Private Methoden(datentyp Argument) : Rückgabewert # Protected Methoden(datentyp Argument) : Rückgabewert - Private Methode() : Rückgabewert (static: unterstrichen)

3.1.2 Beziehungen

Assoziation

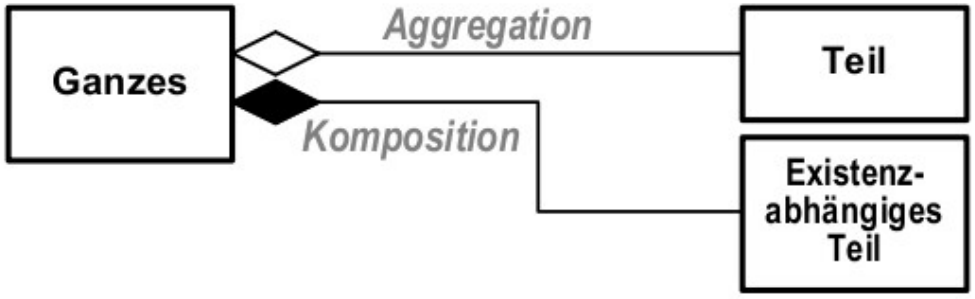
Vererbung

- Beschreibung siehe Kapitel Vererbung

Aggregation / Komposition

- Relationale Beziehungen zwischen Objekten.
- Übergeordnete Objekte enthalten untergeordnete Objekte.
- Das Ganze besteht aus Teilen

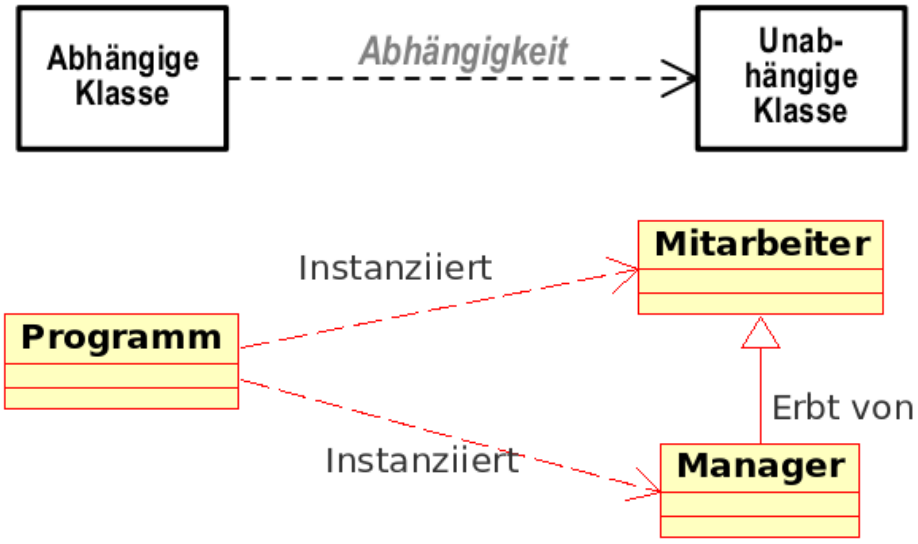
> Man gibt auf der Verbindungslinie den dazugehörigen Member an!



- Die Aggregation ist existenziell nicht abhängig.
  - Das Ganze kann den Teil enthalten, muss aber nicht.
- Bei der Komposition ist das Ganze ist existenziell abhängig vom Teil.
  - Das Ganze muss den Teil zwingend enthalten, damit es funktioniert.

Abhängigkeit

- Die Main-Funktion erstellt eine Klasse.



Kardinalitäten

- Bei den Beziehungen muss man auch die Kardinalitäten angeben. Diese sind die gleichen wie beim Entity-Ralationship-Diagramm (ERD).

1	Exakt eins
0...1	Optional (null oder eins)
0..*	Null oder mehr
1..*	Eins oder mehr
1..5,23	Eins bis 5 oder 23
*	Viele, nicht spezifiziert, ob kann- oder muss-Beziehung

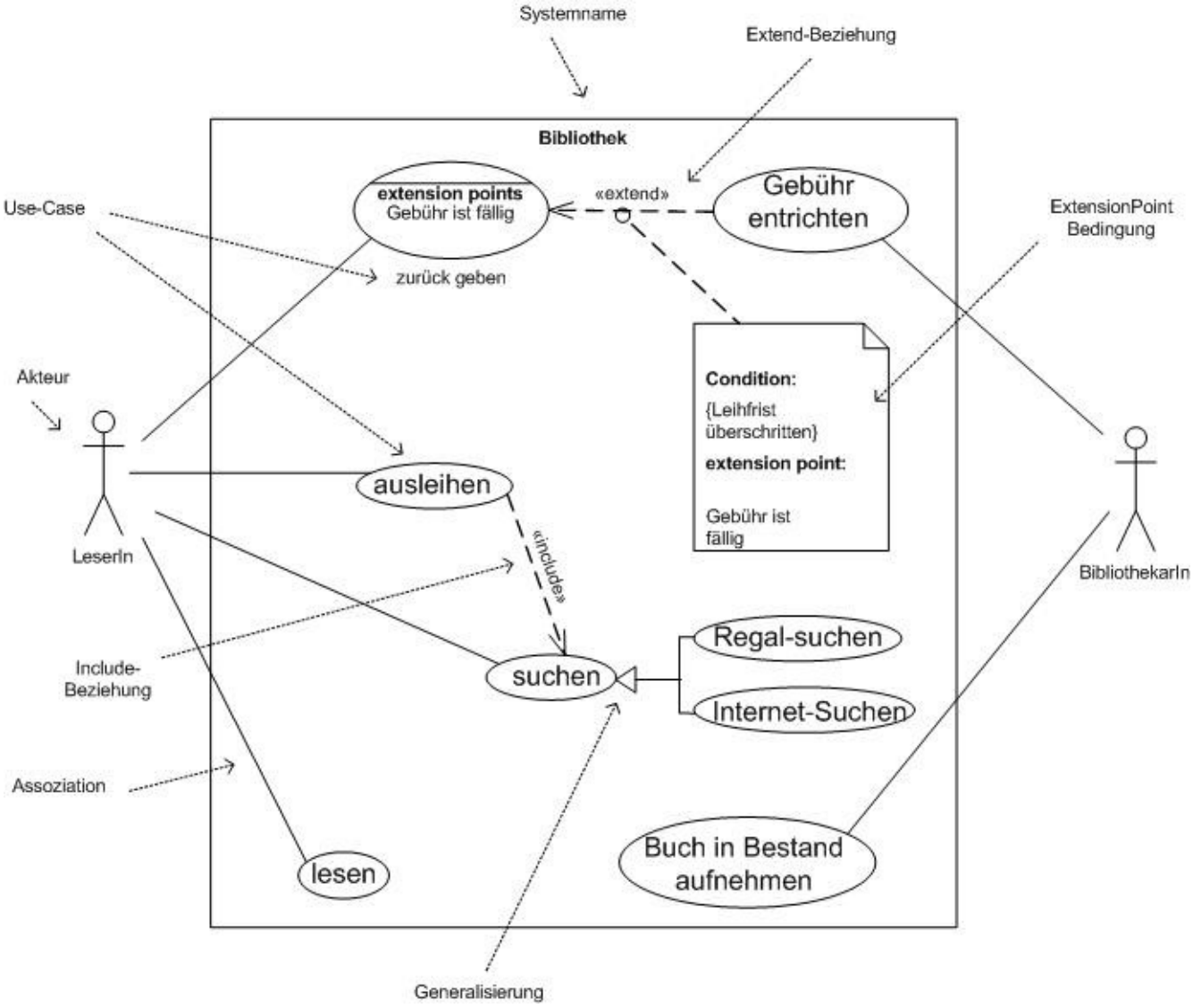
- Diese werden jeweils bei beiden in der Beziehung vorkommenden Klassen angegeben.

3.2 UML-Use-Case-Diagramm (Anwendungsfalldiagramm)

3.2.1 Zweck

- Modelliert die Funktionalität des Systems.
- Nur Anwendungsfälle, die für den externen Betrachter wahrnehmbar sind und einen nutzen erbringen.
- Nicht was im System eigentlich geschieht, sondern was der Anwender vom System erwarten kann.
- Es gibt keine Reihenfolge
- Anwenderwünsche werden erfasst und dokumentiert. Anwender kommentiert das Diagramm.
- Sollen möglichst einfach gehalten werden.

3.2.2 Übersicht



3.2.3 Notationselemente

Systemgrenze (System Boundary)

- System, dass die benötigten Anwendungsfälle bereitstellt.
- Alle Elemente innerhalb des Systems stellen Bestandteile des Systems dar.
- Ob man die Systemgrenze zeichnet, ist nicht obligatorisch.

Akteur (Actor)

- Typ oder Rolle, die ein externer Benutzer oder ein externes System.
- Werden ausserhalb des Systems gezeichnet.
- Man darf selber aussagekräftige Symbole verwenden.

Anwendungsfall (Use Case)

- Abgeschlossene Menge von Aktionen im System.
- Liefern erkennbaren Nutzen
- Die Funktionalität wird gezeigt und nicht wie genau etwas funktioniert.

Assoziation (Association)

- Beziehung zwischen Akteuren und Anwendungsfällen
- Kardinalitäten werden angegeben (1 : 1...*, etc...)
- Ungerichtete Assoziation: Die Kommunikationsrichtung ist unspezifiziert. Jeder kann mit jedem reden.
- Gerichtete Assoziation: Weg der Kommunikation. Einwegkommunikation.
- Alle im Anwendungsfall vorkommende Personen müssen vorhanden sein für die Ausführung.

Generalisierung / Spezialisierung (Generalization)

- Kann mit Akteuren und mit Anwendungsfällen geschehen.
- Anwendungsfälle mit ähnlichen Funktionen können hierarchisch zugeordnet werden und wiederverwendet werden.

Include-Beziehung (Include Relationship)

- Verknüpft einen Anwendungsfall mit einem anderen Anwendungsfall, der zwingend zusätzlich ausgeführt werden muss.

Extend-Beziehung (Exclude Relationship)

- Verknüpft einen Anwendungsfall mit einem anderen Anwendungsfall, der nicht zwingend zusätzlich ausgeführt werden muss.
- Es müssen Bedingungen eingefügt werden. Diese werden mit condition benannt.
- Man muss extensions points setzen (diese kommen zweimal vor!).

> Bei Beziehungen sollte man keinen „Teufelskreis“ entstehen lassen!

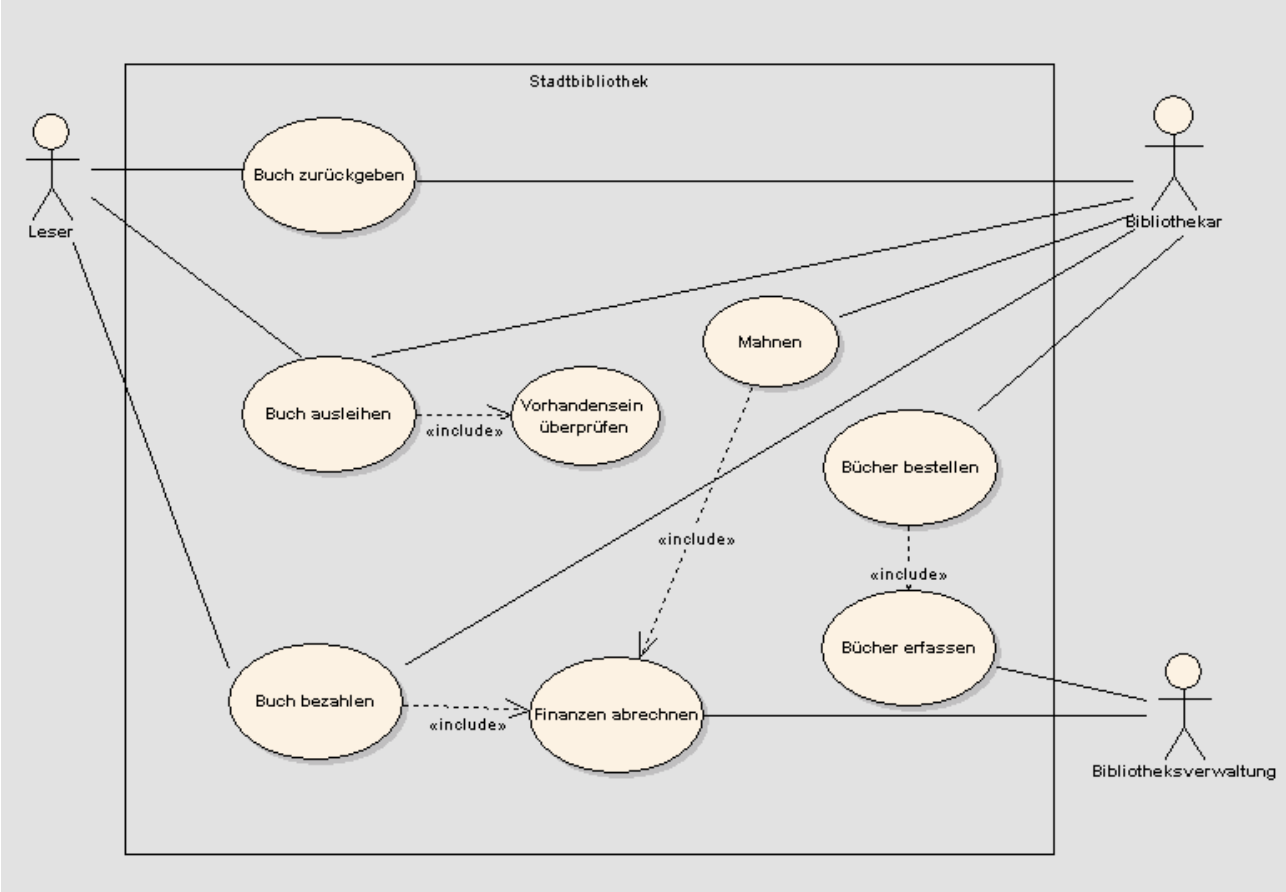
3.2.4 Anwendungsfälle beschreiben

Jeder Anwendungsfall muss folgendermassen beschrieben werden:

- Vor- und Nachbedingungen (Auslöser, Ergebnisse)
- Nicht-Funktionale Anforderungen
- Ablauf
- Variation
- Regeln

- Services
- Ansprechpartner, Sitzungen
- Offene Punkte
- Dialogbeispiele oder -muster
- Diagramme
- etc...

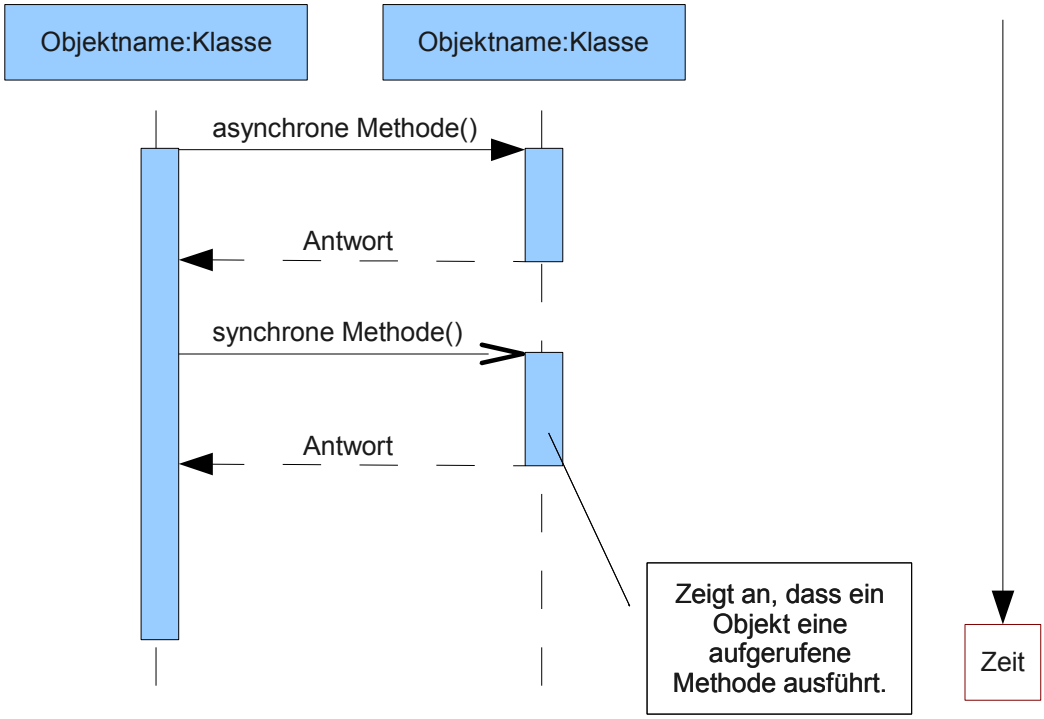
3.2.5 Beispiel



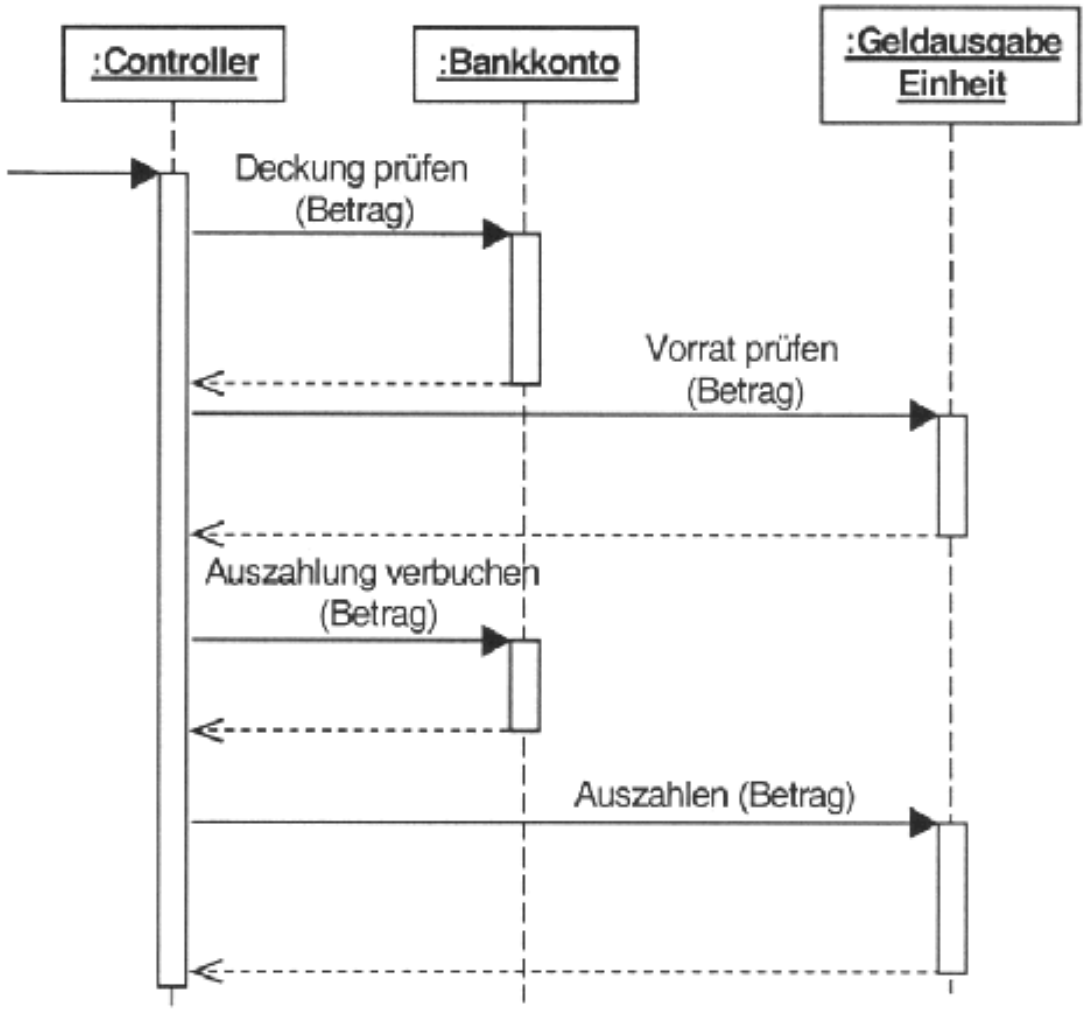
3.3 UML-Sequenzdiagramme

Quelle: http://www.fbi.h-da.de/labore/case/uml/sequenzdiagramm.html

- Mit einem UML-Sequenzdiagramm kann man die Kommunikation von Objekten beschreiben.
- Zeitlicher Ablauf wird als Abfolge von Nachrichten zwischen den Objekten dargestellt. Die Zeit schreitet von oben nach unten fort.



3.3.1 Beispiel



3.4 Singleton-Entwurfsmuster

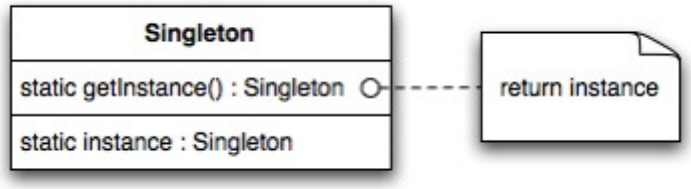
Erzeuge genau eine Instanz einer Klasse und stelle einen zentralen Zugriffspunkt auf diese Instanz bereit.

3.4.1 Zweck

Bisweilen muss verhindert werden, dass von einer Klasse mehrere Instanzen erzeugt werden können - genau dies leistet das Singleton-Muster.

Ein Beispiel für solch einen Fall ist eine Klasse, die die Verbindung zu einer Datenbank verwaltet und für den Rest des Programms Funktionen zur Verfügung stellt, um auf die Datenbank zuzugreifen. Angenommen die Datenbank bietet selbst keine Mechanismen, um beispielsweise atomare Operationen zu gewährleisten; dies bleibt dann der Datenbankklasse vorbehalten. Gäbe es nun mehrere Instanzen dieser Datenbankklasse, könnten wiederum verschiedene Teile des Programms gleichzeitig Änderungen an der Datenbank vornehmen, indem sie sich unterschiedlicher Instanzen bedienen; kann sichergestellt werden, dass es nur genau ein Exemplar der Datenbankklasse gibt, tritt dieses Problem nicht auf.

3.4.2 UML



3.4.3 Entscheidungshilfen

Ein Singleton sollte dann eingesetzt werden, wenn sichergestellt sein muss, dass nicht mehr als ein Objekt einer Klasse erzeugt werden kann.

3.4.4 Funktionsweise

Anstatt selbst eine neue Instanz durch Aufruf des Konstruktors zu erzeugen, müssen sich Benutzer der Singleton-Klasse eine Referenz auf eine Instanz über die statische getInstance()-Methode besorgen - die Singleton-Klasse ist also selbst für die Verwaltung ihrer einzigen Instanz zuständig. Die getInstance()-Methode kann nun sicherstellen, dass bei jedem Aufruf eine Referenz auf dieselbe und einzige Instanz - gehalten in einer (versteckten) Klassenvariable - zurückgegeben wird. Üblicherweise wird diese eine Instanz von getInstance() beim ersten Aufruf erzeugt.

In nebenläufigen Programmen muss der Programmierer beim Schreiben der getInstance()-Methode besondere Vorsicht walten lassen (siehe Code-Beispiele).

3.4.5 Code

```
using System;
using Org.Wikibooks.De.Csharp.Pattern;

namespace Org.Wikibooks.De.Csharp.Pattern.Creational
{
 class Singleton
 {
  // Eine (versteckte) Klassenvariable vom Typ der eigene Klasse
  private static Singleton m_Instance;

  // Konstuktor
  // Dieser Konstruktor kann von außen nicht erreicht werden.
  private Singleton() {}

  // Instanziierung
  public static Singleton getInstance()
  {
   // lazy creation
   if (m_Instance == null)
   {
    m_Instance = new Singleton();
   }
   return m_Instance;
  }
 }
}
```

4 Testing

4.1 Warum?

- Ich will Fehler finden!
- Es kann Schaden an Mensch und Natur entstehen.
- Finanzielle Folgen können entstehen.
- Imageschänden können entstehen
- Man kann sich Strafen einholen.

4.2 Grundlagen

- Jeder Fehler mindert die Qualität.
- Fehler möglichst früh finden
- Aus Fehlern lernen und zukünftig vermeiden
- Unabhängige Tester sollen auch testen
- Testziele formulieren und messen
- Testfälle professionell handhaben
- Testaktivitäten planen
- Tests dokumentieren

4.3 Testfälle ermitteln

- Testfälle sollen minimalischtisch sein
- Testfälle sollen das gesammte Testobjekt überdecken
- Testfälle sollen nicht nur dem Normalfall entsprechen.

4.3.1 Blackbox-Testing

- Testen anhand von Daten
- Grenzwerte prüfen.
- Fehlererwartung
- Äquivalentsklasse: Eine Zahl z.B. Erfüllt mehrere Werte. (5 Testet die zugelassenen Werte zwischen 0 und 10)

4.3.2 Whitebox-Testing

- Testen anhand der Logik
- Möglichst viele Programmteile testen

Anweisungsüberdeckung

- Man will jede Anweisung überprüfen

Anweisungsüberdeckung = Anzahl ausgeführte Anweisungen / Gesamtzahl der Anweisungen * 100%

Zweigüberdeckung

Pfadüberdeckung

Bedingungsüberdeckung

5 Glossar

- Vgl. Begriffe OOP, 1. Kapitel

6 Gute Links

- UML: http://www.fbi.h-da.de/labore/case/uml.html
-

Stichwortverzeichnis