Materialien zu den zentralen Abiturprüfungen im Fach Informatik ab 2012

Java

Inhalt

1. Vorwort 3
2. Basis-Sprachelemente und -Datentypen 4
3. Klassendiagramme 5
  Klassen 5
  Beziehungen zwischen Klassen 5
4. Klassendokumentationen 8
  4.1. Lineare Strukturen 8
    Die Klasse Queue 8
    Die Klasse Stack 9
    Die Klasse List 10
  4.2 Baumstrukturen 12
    Die Klasse BinaryTree 12
    Die Klasse `Item` 14
    Die Klasse BinarySearchTree 15
  4.3. Graphen 17
    Die Klasse GraphNode 17
    Die Klasse Graph 18
  4.4 Netzstrukturen 20
    Die Klasse Connection 20
    Die Klasse Client 21
    Die Klasse Server 22
5. Datenbanken 20
6. Automaten, Grammatiken 31
7. Beispiele 34
  7.1 Beispiel für die Anwendung der Klasse `BinaryTree` 34
  7.2 Beispiel für die Anwendung der Klasse BinarySearchTree 39
  7.3 Beispiel für die Anwendung der Klassen Graph und GraphNode 42
  7.4 Beispiel für die Anwendung der Klassen Client und Server 45
  7.5 Beispiel für eine Datenbankanwendung 48

1. Vorwort

Grundlage für die zentral gestellten schriftlichen Aufgaben der Abiturprüfung sind neben dem gültigen Lehrplan die Vorgaben für das Zentralabitur 2012. Diese beschreiben die inhaltlichen Schwerpunkte, die in der Qualifikationsphase der gymnasialen Oberstufe und den entsprechenden Semestern der Weiterbildungskollegs im Unterricht behandelt werden müssen. Außerdem enthalten sie Hinweise für den Fachlehrer zu der von ihm vorzunehmenden Auswahl der Abituraufgaben seines Kurses. Diese Materialien sollen dazu dienen, die Vorgaben zu präzisieren und zu ergänzen, um Fehlinterpretationen zu vermeiden, Fragen zu klären und dem Fachlehrer weitere fachliche Hinweise, Beispiele und Materialien zur unterrichtlichen Vorbereitung der Schülerinnen und Schüler auf die zentrale Abiturprüfung an die Hand zu geben. Der Inhalt dieser Materialien wurde mit den Fachwissenschaftlern und den Fachdidaktikern der unabhängigen Kommission für das Zentralabitur im Fach Informatik abgestimmt.

In Kapitel 2 werden die Basis-Sprachelemente der Programmiersprache Java angegeben, die den Abiturienten bekannt sein müssen. Darüber hinausgehende spezifische Java-Klassen werden in den Abituraufgaben nicht verwendet. Insbesondere werden keine Programmierkenntnisse zur Gestaltung einer grafischen Benutzeroberfläche benötigt.
In Kapitel 3 werden die für die Abiturprüfung relevanten Klassendiagramme und ihre Beziehungen dargestellt. Die in der UML beschriebenen Klassendiagramme wurden für den Schulbereich didaktisch reduziert, indem lediglich die gerichtete Assoziation (bisher als "kennt-Beziehung" bezeichnet) und die Vererbung (bisher "ist-Beziehung" genannt) als Beziehungen zwischen Klassen verwendet werden. Auf die in den alten Materialien und Aufgaben benutzte "hat-Beziehung" wird aus Vereinfachungsgründen und um aufgetretene Missverständnisse und Unklarheiten zu vermeiden in Zukunft verzichtet.
Kapitel 4 enthält die Dokumentationen der in den Aufgaben des Zentralabiturs verwendeten Klassen zu linearen Strukturen, Baumstrukturen, Graphen sowie den Netzstrukturen. Diese Klassen müssen intensiv im Unterricht besprochen und im Anwendungskontext genutzt werden. In einigen Klassen wurden gegenüber der letzten Fassung dieser Materialien Methoden ergänzt oder ihre Namen oder Funktionalität geändert. Die Dokumentationen enthalten jetzt keine Vor - und Nachbedingungen für die Anwendung der Klassenmethoden mehr, sondern beschreiben deren Funktionalität inklusive aller Sonderfälle. Auf das Auslösen von Ausnahmen (Exceptions) wurde bei der Implementation der Klassen aus didaktischen Gründen verzichtet.
Die Kapitel 5 und 6 enthalten zusätzliche fachliche Hinweise und Beispiele zu den Themenbereichen "Datenbanken“ und „Automaten/Grammatiken“. Die dort verwendete Terminologie ist auch für die Abiturklausur maßgeblich.
In Kapitel 7 findet man einige Anwendungsbeispiele zu den Klassen aus Kapitel 4. Es handelt sich nicht um Abituraufgaben, sondern Übungen, die selbstverständlich auch im Unterricht verwendet werden dürfen.
Die Implementationen zu den in diesen Materialien dokumentierten Java-Klassen sowie die Quelltexte der Beispielprogramme können vom Server `www.Standardsicherung.nrw.de/abitur/` heruntergeladen werden.

Für Hinweise auf Fehler in dieser Schrift oder auch inhaltliche Verbesserungsvorschläge wären wir dankbar. (klaus.dingemann@brms.nrw.de)

Klaus Dingemann, Fachkoordinator für Informatik

2. Basis-Sprachelemente und -Datentypen

Kenntnisse über Java-spezifische Klassen auch zur Gestaltung einer grafischen Benutzungsoberfläche werden bei den Abituraufgaben nicht vorausgesetzt.

Sprachelemente

- Klassendefinitionen
- Beziehungen zwischen Klassen
  - gerichtete Assoziation
  - Vererbung
- Attribute und Methoden (mit Parametern und Rückgabewerten)
- Wertzuweisungen
- Verzweigungen (if - , switch -)
- Schleifen (while -, for -, do - while)

Datentypen

Datentyp	Operationen	Methoden

int	+, -, *, /, %, <, >, <=, >=,==,!=	`Methoden der Klasse Math`
		`Integer.toString(int i)`
		`Wrapper-Klasse`

double	+, -, *, /, <, >, <=, >=,==,!=	`Methoden der Klasse Math`
		`Double.toString(double d)`
		`Double.NaN`
		`Double.isNaN`
		`Wrapper-Klasse`

boolean	&&,||, !, ==, !=,	`Wrapper-Klasse`
char	<, >, <=, >=,==,!=	`Wrapper-Klasse` `Umwandlung in String`

Klasse String		`length()`
		`indexOf(String str)`
		`substring(int beginIndex)`
		`substring (int beginInd, int endInd)`
		`charAt(int index)`
		`equals(Object anObject)`
		`compareTo(String anotherString)`
		`Typumwandlungen in int und double (parse)`

Statische Strukturen

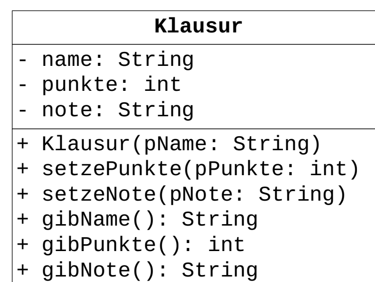
Ein- und zweidimensionale Felder (arrays) von einfachen Datentypen und Objekten

3. Klassendiagramme

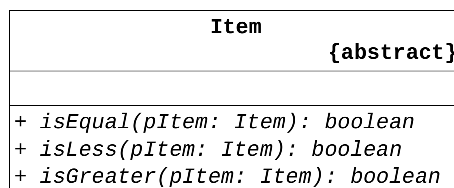
Klassendiagramme beschreiben die vorhandenen Klassen mit ihren Attributen und Methoden sowie die Beziehungen der Klassen untereinander.

Klassen

Klassen werden durch Rechtecke dargestellt, die entweder nur den Namen der Klasse tragen oder zusätzlich auch Attribute und / oder Methoden enthalten. Attribute und Methoden können zusätzliche Angaben zu Parametern und Sichtbarkeit (`public` (+), `private` (-)) besitzen.



Bei abstrakten Klassen, also Klassen, von denen kein Objekt erzeugt werden kann, wird unter den Klassennamen im Diagramm `{abstract}` geschrieben. Abstrakte Methoden, also Methoden, für die keine Implementierungen angegeben werden und die nicht aufgerufen werden können, werden in Kursivschrift dargestellt. Bei einer handschriftlichen Darstellung werden sie mit einer Wellenlinie unterschlängelt.



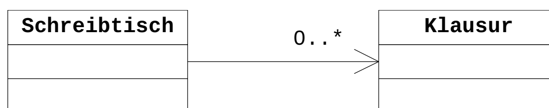
Beziehungen zwischen Klassen

Assoziation

Eine gerichtete Assoziation von einer Klasse A zu einer Klasse B modelliert, dass Objekte der Klasse B in einer Beziehung zu Objekten der Klasse A stehen bzw. stehen können. Bei einer Assoziation kann man angeben, wie viele Objekte der Klasse B in einer solchen Beziehung zu einem Objekt der Klasse A stehen bzw. stehen können. Die Zahl nennt man **Multiplizität**.

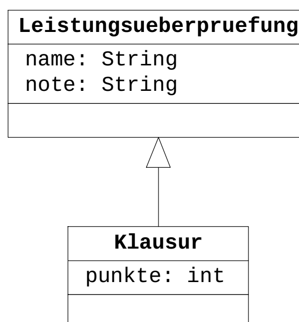
Mögliche Multiplizitäten:

- 1 genau ein assoziiertes Objekt
- 0..1 kein oder ein assoziiertes Objekt
- 0..* beliebig viele assoziierte Objekte
- 1..* mindestens ein, beliebig viele assoziierte Objekte



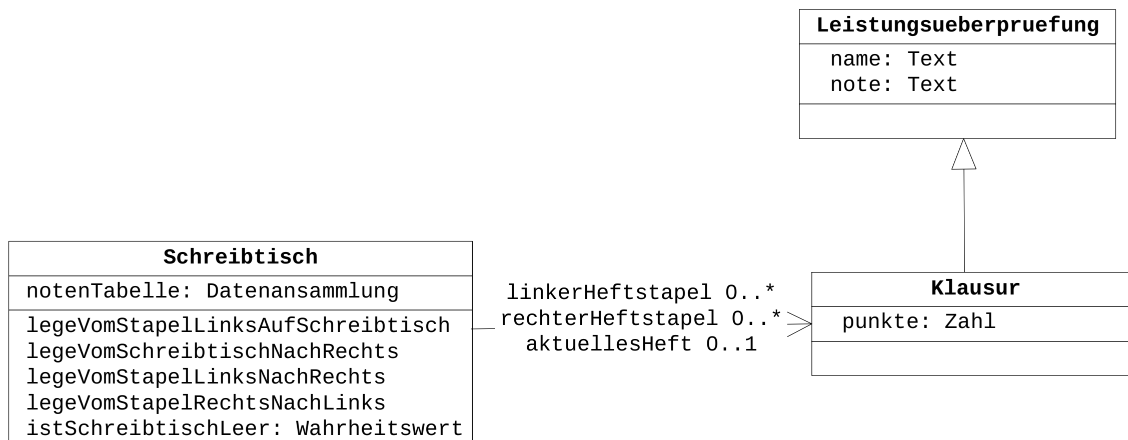
Ein Objekt der Klasse `Schreibtisch` steht zu keinem oder beliebig vielen Objekten der Klasse `Klausur` in Beziehung.

Vererbung
Die Vererbung beschreibt die Beziehung zwischen einer allgemeineren Klasse (Oberklasse) und einer spezialisierten Klasse (Unterklasse). Die Unterklasse stellt alle öffentlichen Attribute und alle nicht überschriebenen öffentlichen Methoden der Oberklasse zur Verfügung. In der Unterklasse können Attribute und Methoden ergänzt oder auch überschrieben werden.



Die Klasse `Klausur` spezialisiert hier die Klasse `Leistungsueberpruefung`, da eine Klausur das Ergebnis zusätzlich zur Note auch in Punkten ausweist.

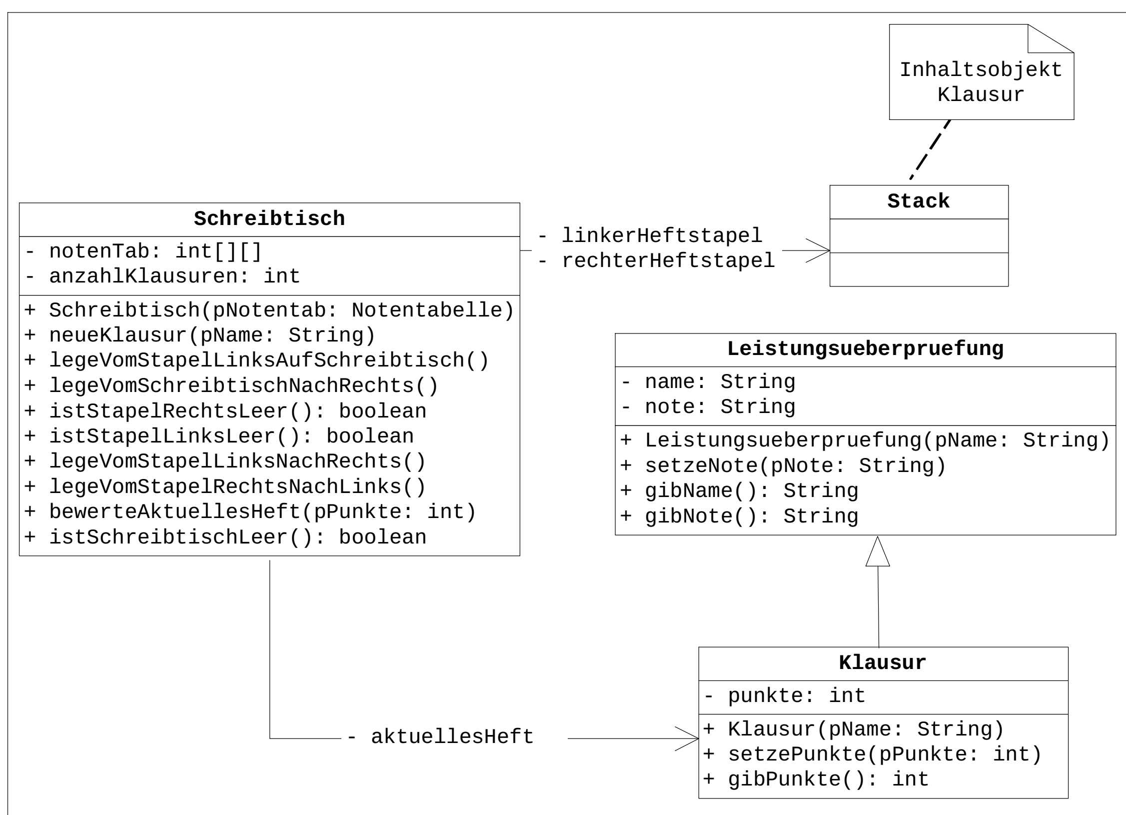
Entwurfsdiagramm
Bei einem Entwurf werden die in der Auftragssituation vorkommenden Objekte identifiziert und ihren Klassen zugeordnet.
Das Entwurfsdiagramm enthält Klassen und ihre Beziehungen mit Multiplizitäten. Als Beziehungen können Vererbung und gerichtete Assoziationen gekennzeichnet werden. Gegebenenfalls werden wesentliche Attribute und / oder Methoden angegeben. Die Darstellung ist programmiersprachenunabhängig ohne Angabe eines konkreten Datentyps, es werden lediglich `Zahl`, `Text`, `Datenansammlung` und `Wahrheitswert` unterschieden. Anfragen werden durch den Datentyp des Rückgabewertes gekennzeichnet.



Implementationsdiagramm

Ein Implementationsdiagramm ergibt sich durch Präzisierung eines Entwurfsdiagramms und orientiert sich stärker an der verwendeten Programmiersprache. Für die im Entwurfsdiagramm angegebenen Datensammlungen werden konkrete Datenstrukturen gewählt, deren Inhaltstypen in Form von Kommentardiagrammen angegeben werden. Die Attribute werden mit den in der Programmiersprache (hier Java) verfügbaren Datentypen versehen und die Methoden mit Parametern incl. ihrer Datentypen.
Bei den in dieser Schrift dokumentierten Klassen (`List, BinaryTree, ..`) wird auf die Angabe der Attribute und der Methoden verzichtet.

Beispiel für ein Implementationsdiagramm mit Assoziationen und Vererbung



Erläuterung:
Bezeichner der Attribute, durch die die Assoziationen realisiert werden, stehen an den Pfeilen.
Objekte der Klasse `Stack` verwalten Objekte der Klasse `Object`. Der Kommentar an der Klasse `Stack` verdeutlicht, welche Inhaltsobjekte im `Stack` hier verwendet werden sollen.
Das Rechteck um das Klassendiagramm zeigt an, dass alle Klassen im gleichen Namensraum liegen.

4. Klassendokumentationen

In Java werden Objekte über Referenzen verwaltet, d.h., eine Variable `pObject` von der Klasse `Object` enthält eine Referenz auf das entsprechende Objekt. Zur Vereinfachung der Sprechweise werden jedoch im Folgenden die Referenz und das referenzierte Objekt sprachlich nicht unterschieden.

4.1. Lineare Strukturen

Die Klasse Queue

Objekte der Klasse **`Queue`** (Warteschlange) verwalten beliebige Objekte nach dem First-In-First-Out-Prinzip, d.h., das zuerst abgelegte Objekt wird als erstes wieder entnommen.

Dokumentation der Klasse Queue

Konstruktor	**`Queue()`** Eine leere Schlange wird erzeugt.
Anfrage	**`boolean isEmpty()`** Die Anfrage liefert den Wert `true`, wenn die Schlange keine Objekte enthält, sonst liefert sie den Wert `false`.
Auftrag	**`void enqueue(Object pObject)`** Das Objekt `pObject` wird an die Schlange angehängt. Falls `pObject` gleich `null` ist, bleibt die Schlange unverändert.
Auftrag	**`void dequeue()`** Das erste Objekt wird aus der Schlange entfernt. Falls die Schlange leer ist, wird sie nicht verändert.
Anfrage	**`Object front()`** Die Anfrage liefert das erste Objekt der Schlange. Die Schlange bleibt unverändert. Falls die Schlange leer ist, wird `null` zurück gegeben.

Die Klasse Stack

Objekte der Klasse **`Stack`** (Keller, Stapel*)* verwalten beliebige Objekte nach dem Last-In-First-Out-Prinzip, d.h., das zuletzt abgelegte Objekt wird als erstes wieder entnommen.

Dokumentation der Klasse Stack

Konstruktor **`Stack()`**
Ein leerer Stapel wird erzeugt.

Anfrage **`boolean isEmpty()`**
Die Anfrage liefert den Wert `true`, wenn der Stapel keine Objekte enthält, sonst liefert sie den Wert `false`.

Auftrag **`void push(Object pObject)`**
Das Objekt `pObject` wird oben auf den Stapel gelegt. Falls `pObject` gleich `null` ist, bleibt der Stapel unverändert.

Auftrag **`void pop()`**
Das zuletzt eingefügte Objekt wird von dem Stapel entfernt. Falls der Stapel leer ist, bleibt er unverändert.

Anfrage **`Object top()`**
Die Anfrage liefert das oberste Stapelobjekt. Der Stapel bleibt unverändert. Falls der Stapel leer ist, wird `null` zurück gegeben.

Die Klasse List

Objekte der Klasse **`List`** verwalten beliebig viele, linear angeordnete Objekte. Auf höchstens ein Listenobjekt, aktuelles Objekt genannt, kann jeweils zugegriffen werden. Wenn eine Liste leer ist, vollständig durchlaufen wurde oder das aktuelle Objekt am Ende der Liste gelöscht wurde, gibt es kein aktuelles Objekt. Das erste oder das letzte Objekt einer Liste können durch einen Auftrag zum aktuellen Objekt gemacht werden. Außerdem kann das dem aktuellen Objekt folgende Listenobjekt zum neuen aktuellen Objekt werden.
Das aktuelle Objekt kann gelesen, verändert oder gelöscht werden. Außerdem kann vor dem aktuellen Objekt ein Listenobjekt eingefügt werden.

Dokumentation der Klasse List

Konstruktor **`List()`**
Eine leere Liste wird erzeugt.

Anfrage **`boolean isEmpty()`**
Die Anfrage liefert den Wert `true`, wenn die Liste keine Objekte enthält, sonst liefert sie den Wert `false`.

Anfrage **`boolean hasAccess()`**
Die Anfrage liefert den Wert `true`, wenn es ein aktuelles Objekt gibt, sonst liefert sie den Wert `false`.

Auftrag **`void next()`**
Falls die Liste nicht leer ist, es ein aktuelles Objekt gibt und dieses nicht das letzte Objekt der Liste ist, wird das dem aktuellen Objekt in der Liste folgende Objekt zum aktuellen Objekt, andernfalls gibt es nach Ausführung des Auftrags kein aktuelles Objekt, d.h. `hasAccess()` liefert den Wert `false`.

Auftrag **`void toFirst()`**
Falls die Liste nicht leer ist, wird das erste Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.

Auftrag **`void toLast()`**
Falls die Liste nicht leer ist, wird das letzte Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.

Anfrage **`Object getObject()`**
Falls es ein aktuelles Objekt gibt (`hasAccess() == true`), wird das aktuelle Objekt zurückgegeben, andernfalls (`hasAccess()== false`) gibt die Anfrage den Wert `null` zurück.

Auftrag **`void setObject(Object pObject)`**
Falls es ein aktuelles Objekt gibt (`hasAccess() == true`) und `pObject` ungleich `null` ist, wird das aktuelle Objekt durch `pObject` ersetzt. Sonst bleibt die Liste unverändert.

Auftrag **`void append(Object pObject)`**

Ein neues Objekt `pObject` wird am Ende der Liste eingefügt. Das aktuelle Objekt bleibt unverändert. Wenn die Liste leer ist, wird das Objekt `pObject` in die Liste eingefügt und es gibt weiterhin kein aktuelles Objekt (`hasAccess() == false`).
Falls `pObject` gleich `null` ist, bleibt die Liste unverändert.

Auftrag **`void insert(Object pObject)`**

Falls es ein aktuelles Objekt gibt (`hasAccess() == true`), wird ein neues Objekt vor dem aktuellen Objekt in die Liste eingefügt. Das aktuelle Objekt bleibt unverändert. Falls die Liste leer ist und es somit kein aktuelles Objekt gibt (`hasAccess() == false`), wird `pObject` in die Liste eingefügt und es gibt weiterhin kein aktuelles Objekt. Falls es kein aktuelles Objekt gibt (`hasAccess() == false`) und die Liste nicht leer ist oder `pObject` gleich `null` ist, bleibt die Liste unverändert.

Auftrag **`void concat(List pList)`**

Die Liste `pList` wird an die Liste angehängt. Anschließend wird `pList` eine leere Liste. Das aktuelle Objekt bleibt unverändert. Falls `pList` `null` oder eine leere Liste ist, bleibt die Liste unverändert.

Auftrag **`void remove()`**

Falls es ein aktuelles Objekt gibt (`hasAccess() == true`), wird das aktuelle Objekt gelöscht und das Objekt hinter dem gelöschten Objekt wird zum aktuellen Objekt. Wird das Objekt, das am Ende der Liste steht, gelöscht, gibt es kein aktuelles Objekt mehr (`hasAccess() ==` false). Wenn die Liste leer ist oder es kein aktuelles Objekt gibt (`hasAccess() == false`), bleibt die Liste unverändert.

4.2 Baumstrukturen

Die Klasse BinaryTree

Mithilfe der Klasse `BinaryTree` können beliebig viele Inhaltsobjekte in einem Binärbaum verwaltet werden. Ein Objekt der Klasse stellt entweder einen leeren Baum dar oder verwaltet ein Inhaltsobjekt sowie einen linken und einen rechten Teilbaum, die ebenfalls Objekte der Klasse `BinaryTree` sind.

Dokumentation der Klasse BinaryTree

Konstruktor **`BinaryTree()`**
Nach dem Aufruf des Konstruktors existiert ein leerer Binärbaum.

Konstruktor **`BinaryTree(Object pObject)`**
Wenn der Parameter `pObject` ungleich `null` ist, existiert nach dem Aufruf des Konstruktors der Binärbaum und hat `pObject` als Inhaltsobjekt und zwei leere Teilbäume. Falls der Parameter `null` ist, wird ein leerer Binärbaum erzeugt.

Konstruktor **`BinaryTree(Object pObject, BinaryTree pLeftTree, BinaryTree pRightTree)`**
Wenn der Parameter `pObject` ungleich `null` ist, wird ein Binärbaum mit `pObject` als Inhaltsobjekt und den beiden Teilbäume `pLeftTree` und `pRightTree` erzeugt. Sind `pLeftTree` oder `pRightTree` gleich `null`, wird der entsprechende Teilbaum als leerer Binärbaum eingefügt. Wenn der Parameter `pObject` gleich `null` ist, wird ein leerer Binärbaum erzeugt.

Anfrage **`boolean isEmpty()`**
Diese Anfrage liefert den Wahrheitswert `true`, wenn der Binärbaum leer ist, sonst liefert sie den Wert `false`.

Auftrag **`void setObject(Object pObject)`**
Wenn der Binärbaum leer ist, wird der Parameter pObject als Inhaltsobjekt sowie ein leerer linker und rechter Teilbaum eingefügt. Ist der Binärbaum nicht leer, wird das Inhaltsobjekt durch `pObject` ersetzt. Die Teilbäume werden nicht geändert. Wenn `pObject` `null` ist, bleibt der Binärbaum unverändert.

Anfrage **`Object getObject()`**
Diese Anfrage liefert das Inhaltsobjekt des Binärbaums. Wenn der Binärbaum leer ist, wird `null` zurückgegeben.

Auftrag **`void setLeftTree(BinaryTree pTree)`**
Wenn der Binärbaum leer ist, wird `pTree` nicht angehängt. Andernfalls erhält der Binärbaum den übergebenen Baum als linken Teilbaum. Falls der Parameter `null` ist, ändert sich nichts.

Auftrag **`void setRightTree(BinaryTree pTree)`**
Wenn der Binärbaum leer ist, wird `pTree` nicht angehängt. Andernfalls erhält der Binärbaum den übergebenen Baum als rechten Teilbaum. Falls der Parameter `null` ist, ändert sich nichts.

Anfrage **`BinaryTree getLeftTree()`**
Diese Anfrage liefert den linken Teilbaum des Binärbaumes. Der Binärbaum ändert sich nicht. Wenn der Binärbaum leer ist, wird `null` zurückgegeben.

Anfrage **`BinaryTree getRightTree()`**
Diese Anfrage liefert den rechten Teilbaum des Binärbaumes. Der Binärbaum ändert sich nicht. Wenn der Binärbaum leer ist, wird `null` zurückgegeben.

Die Klasse *`Item`*

Die Klasse *`Item`* ist abstrakte Oberklasse aller Klassen, deren Objekte in einen Suchbaum (`BinarySearchTree`) eingefügt werden sollen. Die Ordnungsrelation wird in den Unterklassen von `Item` durch Überschreiben der drei abstrakten Methoden `isEqual`, `isGreater` und `isLess` festgelegt.

Die Klasse `Item` gibt folgende abstrakte Methoden vor:

`abstract boolean isEqual(Item pItem)`
Wenn festgestellt wird, dass das Objekt, von dem die Methode aufgerufen wird, bzgl. der gewünschten Ordnungsrelation gleich dem Objekt `pItem` ist, wird `true` geliefert. Sonst wird `false` geliefert.

`abstract boolean isLess(Item pItem)`
Wenn festgestellt wird, dass das Objekt, von dem die Methode aufgerufen wird, bzgl. der gewünschten Ordnungsrelation kleiner als das Objekt `pItem` ist, wird `true` geliefert. Sonst wird `false` geliefert.

`abstract boolean isGreater(Item pItem)`
Wenn festgestellt wird, dass das Objekt, von dem die Methode aufgerufen wird, bzgl. der gewünschten Ordnungsrelation größer als das Objekt `pItem` ist, wird `true` geliefert. Sonst wird `false` geliefert.

Die Klasse BinarySearchTree

In einem Objekt der Klasse `BinarySearchTree` werden beliebig viele Objekte in einem Binärbaum (binärer Suchbaum) entsprechend einer Ordnungsrelation verwaltet. Ein Objekt der Klasse stellt entweder einen leeren Baum dar oder verwaltet ein Inhaltsobjekt sowie einen linken und einen rechten Teilbaum, die ebenfalls Objekte der Klasse `BinarySearchTree` sind. Dabei gilt:
Die Inhaltsobjekte sind Objekte einer Unterklasse von `Item`, in der durch Überschreiben der drei Vergleichsmethoden `isLess`, `isEqual`, `isGreater` (s. Item) eine eindeutige Ordnungsrelation festgelegt sein muss.
Alle Objekte im linken Teilbaum sind kleiner als das Inhaltsobjekt des Binärbaumes. Alle Objekte im rechten Teilbaum sind größer als das Inhaltsobjekt des Binärbaumes.
Diese Bedingung gilt auch in beiden Teilbäumen.
Die Klasse `BinarySearchTree` ist keine Unterklasse der Klasse `BinaryTree`, sodass deren Methoden nicht zur Verfügung stehen.

Dokumentation der Klasse BinarySearchTree

Konstruktor	**`BinarySearchTree()`** Der Konstruktor erzeugt einen leeren Suchbaum.
Anfrage	**`boolean isEmpty()`** Diese Anfrage liefert den Wahrheitswert `true`, wenn der Suchbaum leer ist, sonst liefert sie den Wert `false`.
Auftrag	**`void insert(Item pItem)`** Falls ein bezüglich der verwendeten Vergleichsmethode `isEqual` mit `pItem` übereinstimmendes Objekt im geordneten Baum enthalten ist, passiert nichts. Andernfalls wird das Objekt `pItem` entsprechend der vorgegebenen Ordnungsrelation in den Baum eingeordnet. Falls der Parameter `null` ist, ändert sich nichts.
Anfrage	**`Item search(Item pItem)`** Falls ein bezüglich der verwendeten Vergleichsmethode `isEqual` mit `pItem` übereinstimmendes Objekt im binären Suchbaum enthalten ist, liefert die Anfrage dieses, ansonsten wird `null` zurückgegeben. Falls der Parameter `null` ist, wird `null` zurückgegeben.
Auftrag	**`void remove(Item pItem)`** Falls ein bezüglich der verwendeten Vergleichsmethode `isEqual` mit `pItem` übereinstimmendes Objekt im binären Suchbaum enthalten ist, wird dieses entfernt. Falls der Parameter `null` ist, ändert sich nichts.
Anfrage	**`Item getItem()`** Diese Anfrage liefert das Inhaltsobjekt des Suchbaumes. Wenn der Suchbaum leer ist, wird `null` zurückgegeben.

Anfrage **`BinarySearchTree getLeftTree()`**
Diese Anfrage liefert den linken Teilbaum des binären Suchbaumes. Der binäre Suchbaum ändert sich nicht. Wenn er leer ist, wird `null` zurückgegeben.

Anfrage **`BinarySearchTree getRightTree()`**
Diese Anfrage liefert den rechten Teilbaum des Suchbaumes. Der Suchbaum ändert sich nicht. Wenn er leer ist, wird `null` zurückgegeben.

4.3 Graphen

Graphen

Ein ungerichteter Graph besteht aus einer Menge von Knoten und einer Menge von Kanten. Die Kanten verbinden jeweils zwei Knoten und können ein Gewicht haben.

Die Klasse GraphNode

Objekte der Klasse `GraphNode` sind Knoten eines Graphen. Ein Knoten hat einen Namen und kann markiert werden.

Dokumentation der Klasse `GraphNode`

Konstruktor	**`GraphNode(String pName)`** Ein Knoten mit dem Namen `pName` wird erzeugt. Der Knoten ist nicht markiert.
Auftrag	**`void mark()`** Der Knoten wird markiert, falls er nicht markiert ist, sonst bleibt er unverändert.
Auftrag	**`void unmark()`** Die Markierung des Knotens wird entfernt, falls er markiert ist, sonst bleibt er unverändert.
Anfrage	**`boolean isMarked()`** Die Anfrage liefert den Wert `true`, wenn der Knoten markiert ist, sonst liefert sie den Wert `false`.
Anfrage	**`String getName()`** Die Anfrage liefert den Namen des Knotens.

Die Klasse Graph

Objekte der Klasse `Graph` sind ungerichtete, gewichtete Graphen. Der Graph besteht aus Knoten, die Objekte der Klasse `GraphNode` sind, und Kanten, die Knoten miteinander verbinden. Die Knoten werden über ihren Namen eindeutig identifiziert.

Dokumentation der Klasse `Graph`

Konstruktor **`Graph()`**
Ein neuer Graph wird erzeugt. Er enthält noch keine Knoten.

Anfrage **`boolean isEmpty()`**
Die Anfrage liefert `true`, wenn der Graph keine Knoten enthält, andernfalls liefert die Anfrage `false`.

Auftrag **`void addNode(GraphNode pNode)`**
Der Knoten `pNode` wird dem Graphen hinzugefügt. Falls bereits ein Knoten mit gleichem Namen im Graphen existiert, wird dieser Knoten nicht eingefügt. Falls `pNode` `null` ist, verändert sich der Graph nicht.

Anfrage **`boolean hasNode(String pName)`**
Die Anfrage liefert `true`, wenn ein Knoten mit dem Namen `pName` im Graphen existiert. Sonst wird `false` zurück gegeben.

Anfrage **`GraphNode getNode(String pName)`**
Die Anfrage liefert den Knoten mit dem Namen `pName` zurück. Falls es keinen Knoten mit dem Namen im Graphen gibt, wird `null` zurück gegeben.

Auftrag **`void removeNode(GraphNode pNode)`**
Falls `pNode` ein Knoten des Graphen ist, so werden er und alle mit ihm verbundenen Kanten aus dem Graphen entfernt. Sonst wird der Graph nicht verändert.

Auftrag **`void addEdge(GraphNode pNode1, GraphNode pNode2, double pWeight)`**
Falls eine Kante zwischen `pNode1` und `pNode2` noch nicht existiert, werden die Knoten `pNode1` und `pNode2` durch eine Kante verbunden, die das Gewicht `pWeight` hat. `pNode1` ist also Nachbarknoten von `pNode2` und umgekehrt. Falls eine Kante zwischen `pNode1` und `pNode2` bereits existiert, erhält sie das Gewicht `pWeight`. Falls einer der Knoten `pNode1` oder `pNode2` im Graphen nicht existiert oder `null` ist, verändert sich der Graph nicht.

Anfrage **`boolean hasEdge(GraphNode pNode1,GraphNode pNode2)`**
Die Anfrage liefert `true`, falls eine Kante zwischen `pNode1` und `pNode2` existiert, sonst liefert die Anfrage `false`.

Anfrage **`void removeEdge(GraphNode pNode1,GraphNode pNode2)`**
Falls `pNode1` und `pNode2` nicht `null` sind und eine Kante zwischen `pNode1` und `pNode2` existiert, wird die Kante gelöscht. Sonst bleibt der Graph unverändert.

Anfrage **`double getEdgeWeight(GraphNode pNode1, GraphNode pNode2)`**
Die Anfrage liefert das Gewicht der Kante zwischen `pNode1` und `pNode2`. Falls die Kante nicht existiert, wird `Double.NaN` (not a number) zurück gegeben.

Auftrag **`void resetMarks()`**
Alle Knoten des Graphen werden als unmarkiert gekennzeichnet.

Anfrage **`boolean allNodesMarked()`**
Die Anfrage liefert den Wert `true`, wenn alle Knoten des Graphen markiert sind, sonst liefert sie den Wert `false`. Wenn der Graph leer ist, wird `true` zurückgegeben.

Anfrage **`List getNodes()`**
Die Anfrage liefert eine Liste, die alle Knoten des Graphen enthält.

Anfrage **`List getNeighbours(GraphNode pNode)`**
Die Anfrage liefert eine Liste, die alle Nachbarknoten des Knotens `pNode` enthält.

4.4 Netzstrukturen

Die Klasse Connection

Objekte der Klasse `Connection` ermöglichen eine Netzwerkverbindung mit dem TCP/IP-Protokoll. Es können nach Verbindungsaufbau zu einem Server Zeichenketten (Strings) gesendet und empfangen werden. Zur Vereinfachung geschieht dies zeilenweise, d. h., beim Senden einer Zeichenkette wird ein Zeilentrenner ergänzt und beim Empfangen wird er entfernt.
Eine Fehlerbehandlung, z.B. ein Zugriff auf eine bereits geschlossene Verbindung, ist in dieser Klasse aus Gründen der Vereinfachung nicht vorgesehen.

Dokumentation der Klasse Connection

Konstruktor **`Connection(String pServerIP, int pServerPort)`**
Es wird eine Verbindung zum durch IP-Adresse und Portnummer angegebenen Server aufgebaut, so dass Daten gesendet und empfangen werden können.

Auftrag **`void send(String pMessage)`**
Die angegebene Nachricht `pMessage` wird - um einen Zeilentrenner erweitert - an den Server versandt.

Anfrage **`String receive()`**
Es wird auf eine eingehende Nachricht vom Server gewartet und diese Nachricht zurückgegeben, wobei der vom Server angehängte Zeilentrenner entfernt wird. Während des Wartens ist der ausführende Prozess blockiert.

Auftrag **`void close()`**
Die Verbindung wird getrennt und kann nicht mehr verwendet werden.

Die Klasse Client

Über die Klasse Client werden Netzwerkverbindungen mit dem TCP/IP-Protokoll ermöglicht. Es können - nach Verbindungsaufbau zu einem Server - Zeichenketten (Strings) gesendet und empfangen werden, wobei der Empfang nebenläufig geschieht. Zur Vereinfachung geschieht dies zeilenweise, d. h., beim Senden einer Zeichenkette wird ein Zeilentrenner ergänzt und beim Empfangen wird er entfernt.
Die empfangene Nachricht wird durch eine Ereignisbehandlungsmethode verarbeitet, die in Unterklassen überschrieben werden muss.
Eine Fehlerbehandlung ist in dieser Klasse aus Gründen der Vereinfachung nicht vorgesehen.

Dokumentation der Klasse Client

Konstruktor `Client(String pServerIP, int pServerPort)`
Es wird eine Verbindung zum durch IP-Adresse und Portnummer angegebenen Server aufgebaut, so dass Zeichenketten gesendet und empfangen werden können.

Auftrag `void send(String pMessage)`
Die angegebene Nachricht `pMessage` wird - um einen Zeilentrenner erweitert - an den Server versandt.

Auftrag `void processMessage(String pMesssage)`
Nachdem der Server die angegebene Nachricht `pMesssage` gesendet hat wurde der Zeilentrenner entfernt. Der Client kann auf die Nachricht `pMesssage` in dieser Methode reagieren. Allerdings enthält diese Methode keine Anweisungen und muss in Unterklassen überschrieben werden, damit die Nachricht verarbeitet werden kann.

Auftrag `void close()`
Die Verbindung zum Server wird getrennt und kann nicht mehr verwendet werden.

Die Klasse Server

Über die Klasse `Server` ist es möglich, eigene Serverdienste anzubieten, so dass Clients Verbindungen gemäß dem TCP/IP-Protokoll hierzu aufbauen können. Nachrichten werden grundsätzlich zeilenweise verarbeitet, d. h., beim Senden einer Zeichenkette wird ein Zeilentrenner ergänzt und beim Empfangen wird er entfernt.
Verbindungsaufbau, Nachrichtenempfang und Verbindungsende geschehen nebenläufig. Durch Überschreiben der entsprechenden Methoden kann der Server auf diese Ereignisse reagieren.
Eine Fehlerbehandlung ist in dieser Klasse aus Gründen der Vereinfachung nicht vorgesehen.

Dokumentation der Klasse Server

Konstruktor **`Server(int pPortNr)`**
Nach dem Aufruf dieses Konstruktors bietet ein Server seinen Dienst über die angegebene Portnummer an. Clients können sich nun mit dem Server verbinden.

Auftrag **`void closeConnection(String pClientIP, int pClientPort)`**
Unter der Voraussetzung, dass eine Verbindung mit dem angegebenen Client existiert, wird diese beendet. Der Server sendet sich die Nachricht `processClosedConnection`.

Auftrag **`void processClosedConnection(String pClientIP, int pClientPort)`**
Diese Methode ohne Anweisungen wird aufgerufen, bevor der Server die Verbindung zu dem in der Parameterliste spezifizierten Client schließt. Durch das Überschreiben in Unterklassen kann auf die Schließung der Verbindung zum angegebenen Client reagiert werden.

Auftrag **`void processMessage(String pClientIP, int pClientPort, String pMessage)`**
Der Client mit der angegebenen IP und der angegebenen Portnummer hat dem Server eine Nachricht gesendet. Dieser ruft daraufhin diese Methode ohne Anweisungen auf. Durch das Überschreiben in Unterklassen kann auf diese Nachricht des angegebenen Client reagiert werden.

Auftrag **`void processNewConnection(String pClientIP, int pClientPort)`**
Der Client mit der angegebenen IP-Adresse und der angegebenen Portnummer hat eine Verbindung zum Server aufgebaut. Der Server hat daraufhin diese Methode aufgerufen, die in dieser Klasse keine Anweisungen enthält. Durch das Überschreiben in Unterklassen kann auf diesen Neuaufbau einer Verbindung von dem angegebenen Client zum Server reagiert werden.

Auftrag **`void send(String pClientIP, int pClientPort, String pMessage)`**

Wenn eine Verbindung zum angegebenen Client besteht, dann wird diesem Client die angegebene Nachricht - um einen Zeilentrenner erweitert - gesendet.

Auftrag **`void sendToAll(String pMessage)`**

Die angegebene Nachricht wird - um einen Zeilentrenner erweitert - an alle verbundenen Clients gesendet.

Auftrag **`void close()`**

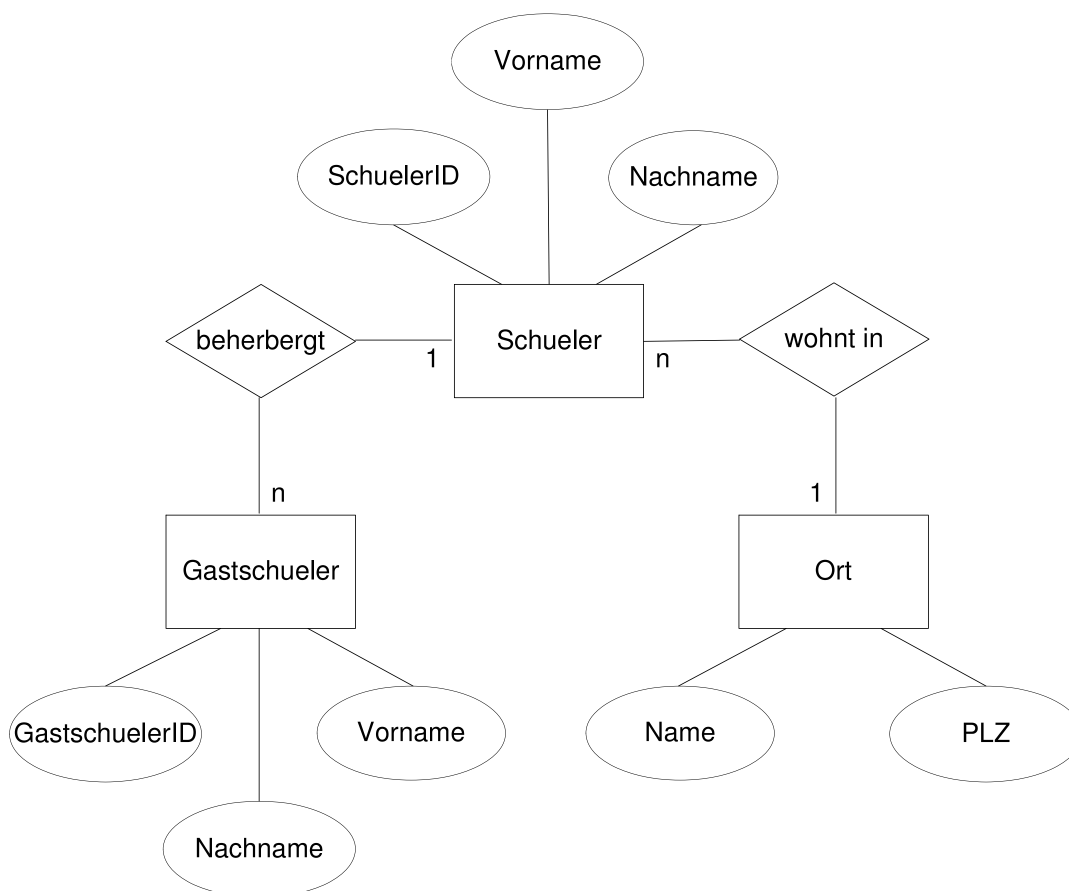
Alle bestehenden Verbindungen werden getrennt. Der Server kann nicht mehr verwendet werden.

5. Datenbanken

Entity-Relationship-Modell

Eine Entität ist ein gedanklich abgegrenzter Gegenstand. Gleichartige Entitäten fasst man zu Entitätsmengen zusammen. Diese erhalten Rechtecke als Symbole.
Zwischen Entitäten kann es Beziehungen geben, wobei man gleichartige Beziehungen zwischen Entitäten zweier Entitätsmengen zu Beziehungsmengen zusammen fasst. Diese erhalten Rauten als Symbole. Eine Raute wird durch Linien mit den Rechtecken der zugehörigen Entitätsmengen verbunden. Diese Linien werden mit der Kardinalität (1:1, 1:n, n:1, n:m) beschriftet.
Attribute sind Eigenschaften aller Elemente einer Entitäts- bzw. Beziehungsmenge. Sie erhalten Ellipsen als Symbole und diese werden durch eine Linie mit dem Rechteck bzw. der Raute der entsprechenden Entitäts- bzw. Beziehungsmenge verbunden.

<u>Beispiel:</u>
In einer Datenbank sollen Informationen über Schüler sowie deren Gastschüler verwaltet werden.



Ein Schüler hat nur einen Wohnort, in einem Ort können aber mehrere Schüler wohnen, also Kardinalität n : 1.
Ein Schüler kann mehrere Gastschüler beherbergen, ein Gastschüler wohnt aber bei nur einem Schüler, also Kardinalität 1 : n.

Datenbankschema und Schlüssel
In einem Datenbankschema werden zeilenweise die Namen der Tabellen einer Datenbank aufgelistet. Dem Tabellennamen folgen jeweils in Klammern die Namen der einzelnen Spalten der entsprechenden Tabelle.
Durch einen Primärschlüssel ist eine Entität eindeutig festgelegt. Ein Fremdschlüssel verweist auf eine andere Entität. Beide Schlüssel können aus einem oder mehreren Attributen bestehen.
Im Datenbankschema werden Primärschlüssel unterstrichen und Sekundärschlüssel mit einem Pfeil versehen.

Beispiel:
Für obiges Beispiel sieht dies dann folgendermaßen aus:
`Schueler (SchuelerID, Vorname, Nachname, ↑PLZ)`
`Gastschueler (GastschuelerID, ↑SchuelerID, Vorname, Nachname)`
`Ort (PLZ, Name)`

Normalisierung
Ein Datenbankschema ist in der **1. Normalform**, wenn alle Attribute einen atomaren Wertebereich haben.

Ein Datenbankschema ist in der **2. Normalform**, wenn es in der 1. Normalform ist und zusätzlich jedes Attribut, das nicht selbst zum Schlüssel gehört, nur von allen Schlüsselattributen funktional abhängig ist und nicht bereits von einem Teil der Schlüsselattribute.

Ein Datenbankschema ist in der **3. Normalform**, wenn es in der 2. Normalform ist und es zusätzlich kein Nichtschlüsselattribut gibt, das transitiv von einem Schlüsselattribut abhängig ist. Es darf also keine funktionalen Abhängigkeiten von Attributen geben, die selbst nicht zum Schlüssel gehören.

Beispiel:
Wir gehen von folgendem Datenbankschema aus:

```
Schuelerkontakt (SchuelerID, VorNachname, PLZOrt, GastschuelerID,
                                        GastschuelerVorNachname)
```

SchuelerID	*VorNachname*	*PLZOrt*	*GastschuelerID*	*Gastschueler-VorNachname*
1	Peter Meier	12321 Infostadt	8	Marc Pierre
1	Peter Meier	12321 Infostadt	13	Bob Thomson
2	Eva Müller	11111 Infodorf	10	Chantal Paris
3	Maria Dorf	12321 Infostadt	NULL	NULL

Das Datenbankschema ist nicht in erster Normalform, da die Attribute `VorNachname`, `PLZOrt` und `GastschuelerVorNachname` nicht atomar sind. Eine Überführung in die erste Normalform führt zu folgendem Datenbankschema:

```
Schuelerkontakt (SchuelerID, Vorname, Nachname, PLZ, Ort,
        GastschuelerID, GastschuelerVorname, GastschuelerNachname)
```

Schueler-ID	*Vor-name*	*Nach-name*	*PLZ*	*Ort*	*Gast-schueler-ID*	*Gastschueler-Vorname*	*Gastschueler-Nachname*
1	Peter	Meier	12321	Infostadt	8	Marc	Pierre
1	Peter	Meier	12321	Infostadt	13	Bob	Thomson
2	Eva	Müller	11111	Infodorf	10	Chantal	Paris
3	Maria	Dorf	12321	Infostadt	NULL	NULL	NULL

Der Primärschlüssel ist aus `SchuelerID` und `GastschuelerID` zusammengesetzt. Dieses Datenbankschema ist nicht in zweiter Normalform, da z.B. `GastschuelerVorname` und `GastschuelerNachname` nur von `GastschuelerID` und nicht vom gesamten Schlüssel abhängen. Eine Überführung in die zweite Normalform führt zu folgendem Datenbankschema:

```
Schueler (SchuelerID, Vorname, Nachname, PLZ, Ort)
Gastschueler (GastschuelerID, ↑SchuelerID, Vorname, Nachname)
```

SchuelerID	*Vorname*	*Nachname*	*PLZ*	*Ort*
1	Peter	Meier	12321	Infostadt
2	Eva	Müller	11111	Infodorf
3	Maria	Dorf	12321	Infostadt

GastschuelerID	*SchuelerID*	*Vorname*	*Nachname*
8	1	Marc	Pierre
13	1	Bob	Thomson
10	2	Chantal	Paris

Dieses Datenbankschema ist nicht in dritter Normalform, da `Ort` abhängig ist von `PLZ` und somit transitiv abhängig ist von `SchuelerID`. Eine Überführung in die dritte Normalform führt zu folgendem Datenbankschema:

```
Schueler (SchuelerID, Vorname, Nachname, ↑PLZ)
Gastschueler (GastschuelerID, ↑SchuelerID, Vorname, Nachname)
Ort (PLZ, Name)
```

SchuelerID	*Vorname*	*Nachname*	*PLZ*
1	Peter	Meier	12321
2	Eva	Müller	11111
3	Maria	Dorf	12321

GastschuelerID	*SchuelerID*	*Vorname*	*Nachname*
8	1	Marc	Pierre
13	1	Bob	Thomson
10	2	Chantal	Paris

PLZ	*Name*
12321	Infostadt
11111	Infodorf

Sprachelemente

Folgende SQL-Sprachelemente werden vorausgesetzt:

```
SELECT (DISTINCT) … FROM
WHERE
GROUP BY
ORDER BY
ASC, DESC
(LEFT / RIGHT) JOIN … ON
UNION
AS
NULL
```

Vergleichsoperatoren: `=, <>, >, <, >=, <=, LIKE, BETWEEN, IN, IS NULL`
Arithmetische Operatoren: `+, -, *, /, (…)`
Logische Verknüpfungen: `AND, OR, NOT`
Funktionen: `COUNT, SUM, MAX, MIN`

Es werden SQL-Abfragen über eine und mehrere verknüpfte Tabellen vorausgesetzt.
Es können auch verschachtelte SQL-Ausdrücke vorkommen.

Relationenalgebra

Es werden Selektion, Projektion, Vereinigung, Differenz, kartesisches Produkt, Umbenennung und Join vorausgesetzt.

Beispiel:
Wir gehen von folgendem Datenbankschema aus:
Lehrer (ID, Vorname, Nachname)
Lehrerin (ID, Vorname, Nachname)
Schulleitung (ID, Vorname, Nachname)
Klassenleitungsteam (↑LehrerID, ↑LehrerinID)

Lehrer:

ID	*Vorname*	*Nachname*
Me	Peter	Meier
Sz	Peter	Schulz
Bm	Hans	Baum

Lehrerin:

ID	*Vorname*	*Nachname*
Be	Petra	Blume
Sr	Clara	Sommer
Kr	Helga	Kremer

Klassenleitungsteam:

LehrerID	*LehrerinID*
Bm	Be
Me	Kr
Me	Sr

Schulleitung:

ID	*Vorname*	*Nachname*
Bm	Hans	Baum
Kr	Helga	Kremer

Selektion:
Es werden die Zeilen ausgewählt, die eine bestimmte Bedingung erfüllen.

```
SELECT * FROM Lehrer WHERE Vorname = “Peter“
```

ID	*Vorname*	*Nachname*
Me	Peter	Meier
Sz	Peter	Schulz

Projektion:
Es werden nur bestimmte Spalten ausgewählt. Doppelte Zeilen werden entfernt.

```
SELECT DISTINCT Vorname FROM Lehrer
```

Vorname
Peter
Hans

Vereinigung:
Zwei Tabellen mit gleichen Attributen werden zu einer vereinigt. Doppelte Zeilen werden entfernt.

```
SELECT * FROM Lehrer UNION SELECT * FROM Lehrerin
```

ID	*Vorname*	*Nachname*
Me	Peter	Meier
Sz	Peter	Schulz
Bm	Hans	Baum
Be	Petra	Blume
Sr	Clara	Sommer
Kr	Helga	Kremer

Differenz:
Es werden die Zeilen einer Tabelle ausgewählt, die in einer zweiten Tabelle nicht enthalten sind.

```
SELECT * FROM Lehrer WHERE Lehrer.ID
     NOT IN (SELECT ID FROM Schulleitung)
```

ID	*Vorname*	*Nachname*
Me	Peter	Meier
Sz	Peter	Schulz

Kartesisches Produkt:
Es werden alle Zeilen einer Tabelle mit allen Zeilen einer zweiten Tabelle verknüpft.

```
SELECT * FROM Lehrer, Lehrerin
```

ID	*Vorname*	*Nachname*	*ID*	*Vorname*	*Nachname*
Me	Peter	Meier	Be	Petra	Blume
Sz	Peter	Schulz	Be	Petra	Blume
Bm	Hans	Baum	Be	Petra	Blume
Me	Peter	Meier	Sr	Clara	Sommer
Sz	Peter	Schulz	Sr	Clara	Sommer
Bm	Hans	Baum	Sr	Clara	Sommer
Me	Peter	Meier	Kr	Helga	Kremer
Sz	Peter	Schulz	Kr	Helga	Kremer
Bm	Hans	Baum	Kr	Helga	Kremer

Umbenennung:
Ein Attribut wird umbenannt.

```
SELECT ID AS Kuerzel, Vorname, Nachname FROM Lehrerin
```

Kuerzel	*Vorname*	*Nachname*
Be	Petra	Blume
Sr	Clara	Sommer
Kr	Helga	Kremer

Join:
Ein Join ist die Bildung eines karthesischen Produktes gefolgt von einer Selektion.

```
SELECT * FROM Lehrer JOIN Klassenleitungsteam
     ON Lehrer.ID = Klassenleitungsteam.LehrerID
```

ID	*Vorname*	*Nachname*	*LehrerID*	*LehrerinID*
Bm	Hans	Baum	Bm	Be
Me	Peter	Meier	Me	Kr
Me	Peter	Meier	Me	Sr

Beim Left-Join werden auch die Zeilen aus der ersten Tabelle aufgeführt, die keinen Partner in der zweiten Tabelle haben. Es wird mit „NULL“ aufgefüllt.

```
SELECT * FROM Lehrer LEFT JOIN Klassenleitungsteam
     ON Lehrer.ID = Klassenleitungsteam.LehrerID
```

ID	*Vorname*	*Nachname*	*LehrerID*	*LehrerinID*
Me	Peter	Meier	Me	Kr
Me	Peter	Meier	Me	Sr
Sz	Peter	Schulz	NULL	NULL
Bm	Hans	Baum	Bm	Be

Beim Right-Join werden auch die Zeilen aus der zweiten Tabelle aufgeführt, die keinen Partner in der ersten Tabelle haben. Es wird mit „NULL“ aufgefüllt.

```
SELECT * FROM Klassenleitungsteam RIGHT JOIN Lehrer
     ON Lehrer.ID = Klassenleitungsteam.LehrerID
```

LehrerID	LehrerinID	ID	Vorname	Nachname
Me	Kr	Me	Peter	Meier
Me	Sr	Me	Peter	Meier
NULL	NULL	Sz	Peter	Schulz
Bm	Be	Bm	Hans	Baum

SQL-Abfrage über mehrere verknüpfte Tabellen:

```
SELECT Lehrer.Nachname AS Klassenlehrer,
     Lehrerin.Nachname AS Klassenlehrerin
     FROM Lehrer, Lehrerin, Klassenleitungsteam
     WHERE Lehrer.ID = Klassenleitungsteam.LehrerID
     AND Klassenleitungsteam.LehrerinID = Lehrerin.ID
```

Klassenlehrer	*Klassenlehrerin*
Baum	Blume
Meier	Kremer
Meier	Sommer

6. Automaten, Grammatiken

Ein **(deterministischer, endlicher) Automat** erhält ein Wort als Eingabe und erkennt (oder akzeptiert) dieses oder nicht. Häufig wird er daher auch als Akzeptor bezeichnet. Die Menge der akzeptierten Wörter bildet die durch den Automaten dargestellte oder definierte Sprache.

Definition:
Ein deterministischer, endlicher Automat wird durch ein 5-Tupel (A, Z, d, q_0, E) spezifiziert:

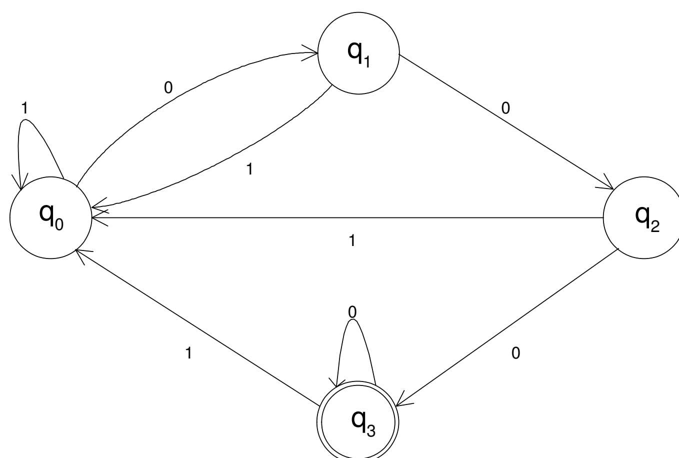
- Das Eingabealphabet A ist eine endliche, nicht leere Menge von Symbolen,
- Z ist eine endliche, nicht leere Menge von Zuständen Z und es gilt $A \cap Z = \emptyset$,
- d: $Z \times A \rightarrow Z$ ist die Zustandsübergangsfunktion,
- $q_0 \in Z$ ist der Anfangszustand und
- $E \subseteq Z$ die Menge der Endzustände.

Die Zustandsübergangsfunktion kann z.B. durch einen Übergangsgraphen oder eine Tabelle dargestellt werden. Die von deterministischen, endlichen Automaten erkannten Sprachen heißen regulär.

Beispiel:
Ein deterministischer, endlicher Automat soll erkennen, ob eine eingegebene Dualzahl (von links nach rechts gelesen) durch 8 teilbar ist.
Grundidee: Nur wenn die letzten drei Ziffern 0 sind, hat eine Dualzahl diese Eigenschaft.
Der Automat wird definiert durch

- das Eingabealphabet `A = {0, 1}`
- die endliche nicht leere Menge von Zuständen `Z = {q0, q1, q2, q3}`
- die Zustandsübergangsfunktion (s. Übergangsgraph)
- den Anfangszustand (ein Element aus Z) q0
- die Menge der Endzustände `E = {q3}`

Der Übergangsgraph des Automaten:



Die Übergangstabelle des Automaten:

Eingabe / Zustand	0	1
q0	q1	q0
q1	q2	q0
q2	q3	q0
q3	q3	q0

Die Menge $\{0,1\}^*$ besteht aus allen Sequenzen der Elemente des Eingabealphabets. Dieser Automat akzeptiert eine Teilmenge dieser Sequenzen, nämlich nur die Eingabewörter (Dualzahlen) mit drei Nullen am Ende und definiert dadurch eine Sprache.

Eine **Grammatik** erzeugt eine Sprache, die aus allen Wörtern besteht, die sich durch Anwendung der Regeln der Grammatik erzeugen (ableiten) lassen.

Definition
Eine **Grammatik** G wird durch ein 4-Tupel (N, T, S, P) spezifiziert:

- N ist die Menge der Nichtterminalsymbole,
- T ist die Menge der Terminalsymbole ($N \cap T = \emptyset$),
- $S \in N$ ist das Startsymbol,
- P ist die Menge der Regeln oder Produktionen.

Grammatiken, bei denen alle Produktionen die Form $A \rightarrow a$ (für $A \in N$ und $a \in T$) oder $A \rightarrow aB$ (für $a \in T$ und $A, B \in N$) haben, heißen **regulär.**

Zu jedem deterministischen, endlichen Automaten A gibt es eine reguläre Grammatik G, welche die von A akzeptierte Sprache erzeugt. Zu jeder regulären Grammatik G gibt es einen deterministischen, endlichen Automaten A, der die von G erzeugte Sprache akzeptiert.

Beispiel:

Die folgende reguläre Grammatik definiert genau die Sprache, deren Wörter von dem obigen Automaten akzeptiert werden.

Grammatik G = (N, T, S, P) der gegebenen Sprache

Menge der Nichtterminalsymbole: N = {S, A, B}
Menge der Terminalsymbole: T = {0, 1}
Startsymbol: S
Produktionen: P = {

$$S \rightarrow 1S \mid 0A,$$
$$A \rightarrow 1S \mid 0B,$$
$$B \rightarrow 1S \mid 0B \mid 0$$

}

Die Ableitung des Wortes 1000 lautet z.B.: $S \rightarrow 1S \rightarrow 10A \rightarrow 100B \rightarrow 1000$

Ein **Parser** zu einer Grammatik ist ein Algorithmus, mit dem sich überprüfen lässt, ob ein Wort zu der von der Grammatik erzeugten Sprache gehört, d.h. ob es syntaktisch korrekt ist. Man könnte auch sagen: Ein Parser simuliert den zu der Grammatik gehörenden Automaten.

Beispiel:

Das folgende Programm simuliert den o.a. deterministischen, endlichen Automaten und überprüft, ob das als Parameter übergebene Wort vom Automaten akzeptiert wird, d.h. zu der von der zugehörigen regulären Grammatik erzeugten Sprache gehört.

```
public boolean parse(String pWort) {
  int lZustand = 0;    // Startzustand = 0
  char symbol;     // das aktuell zu verarbeitende Symbol
  int lZaehler = 0;
  while (lZaehler < pWort.length()) {
    symbol = pWort.charAt(lZaehler);
    switch (lZustand) {
      case 0: switch (symbol) {
                case '0': {lZustand = 1; break;}
                case '1': {lZustand = 0; break;}
              }
              break;
      case 1: switch (symbol) {
                case '0': {lZustand = 2; break;}
                case '1': {lZustand = 0; break;}
              }
              break;
      case 2: switch (symbol) {
                case '0': {lZustand = 3; break;}
                case '1': {lZustand = 0; break;}
              }
              break;
      case 3: switch (symbol) {
                case '0': {lZustand = 3; break;}
                case '1': {lZustand = 0; break;}
              }
              break;
    } // switch
    lZaehler = lZaehler+1;
  } // while
  return lZustand == 3;
}
```

Hinweis:
Ein Werkzeug zum Umgang mit Automaten findet man unter
http://www.cs.duke.edu/csed/jflap/
Das Java-Programm JFLAP ist dort kostenlos erhältlich.

7. Beispiele

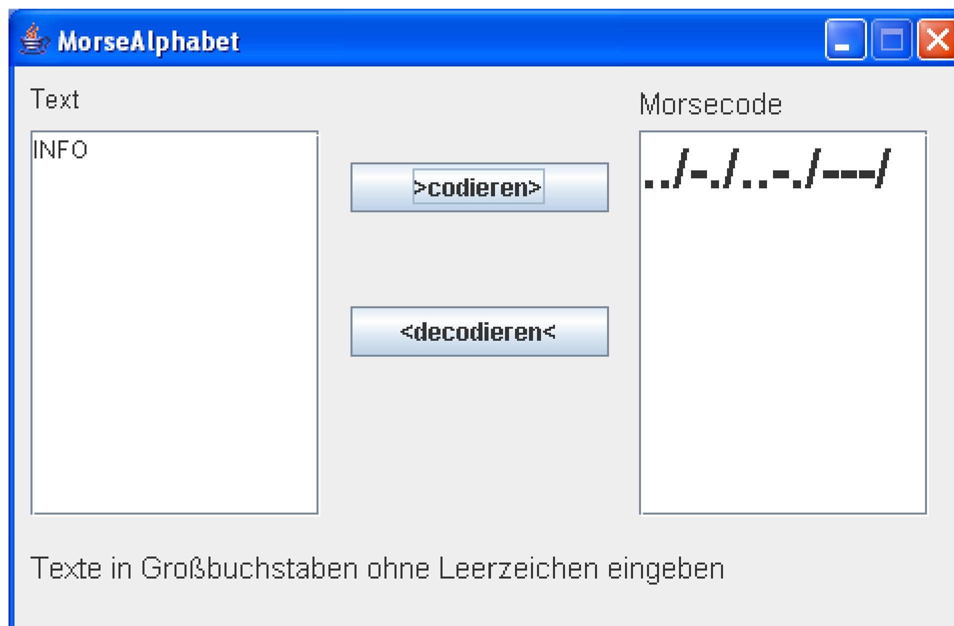
7.1 Beispiel für die Anwendung der Klasse `BinaryTree`

Aufgabe
Es soll ein Programm entwickelt werden, das in der Lage ist, eine Zeichenfolge aus Großbuchstaben in das Morsealphabet zu codieren und eine Folge von Morsezeichen, die durch „/“ getrennt sind, zu dekodieren.

Das Morsealphabet :

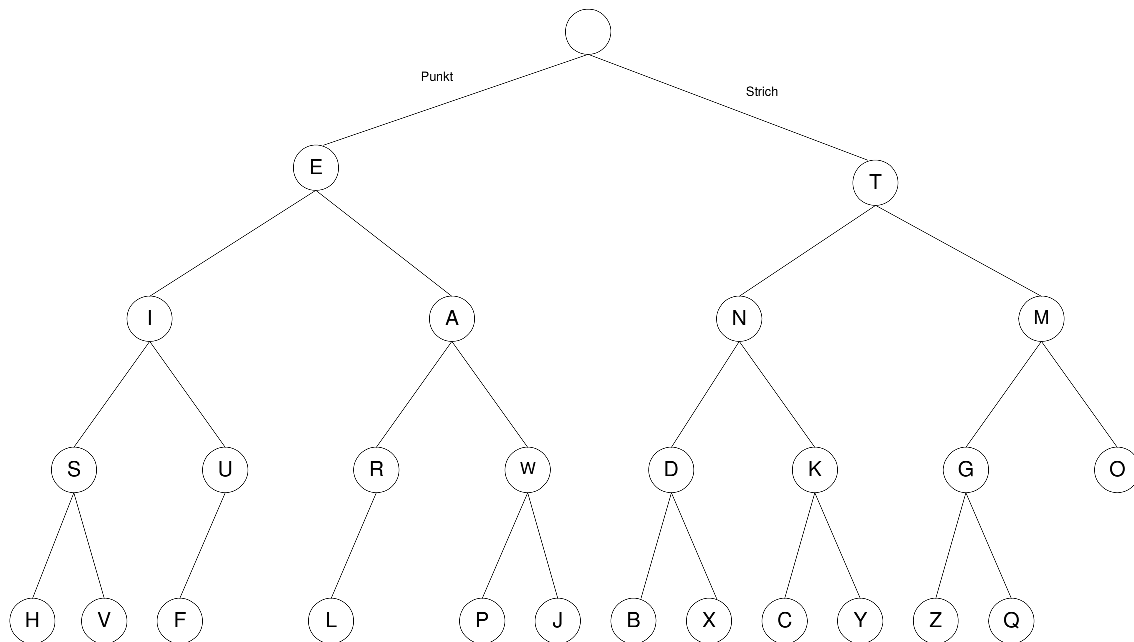
A ·-	F ··-·	K -·-	P ·--·	U ··-	Z --··
B -···	G --·	L ·-··	Q --·-	V ···-	
C -·-·	H ····	M --	R ·-·	W ·--	
D -··	I ··	N -·	S ...	X -··-	
E ·	J ·---	O ---	T -	Y -·--	

Benutzerschnittstelle

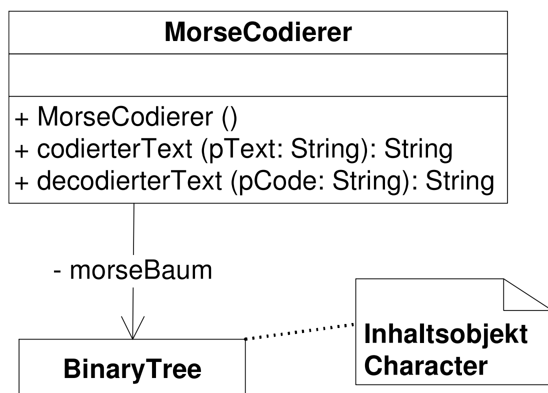


Lösungsskizze:

Da der Morsecode nur aus zwei Zeichen besteht, lässt er sich in einem Binärbaum darstellen:



Implementationsdiagramm



Implementation einiger Methoden

Der Konstruktur `MorseCodierer()` der Klasse `MorseCodierer` erzeugt den Morsebaum. Man beachte die Verwendung verschiedener Konstruktoren der Klasse `BinaryTree`.

```
public MorseCodierer() {
  BinaryTree lBaum4Links = new BinaryTree(new Character('H'));
  BinaryTree lBaum4Rechts = new BinaryTree(new Character('V'));
  BinaryTree lBaum3Links = new BinaryTree(new Character('S'), lBaum4Links,lBaum4Rechts);
  lBaum4Links = new BinaryTree(new Character('F'));
  lBaum4Rechts = new BinaryTree();
  BinaryTree lBaum3Rechts = new BinaryTree(new Character('U'), lBaum4Links,lBaum4Rechts);
  BinaryTree lBaum2Links = new BinaryTree(new Character('I'), lBaum3Links, lBaum3Rechts);
  lBaum4Links = new BinaryTree(new Character('L'));
  lBaum4Rechts = new BinaryTree();
  lBaum3Links = new BinaryTree(new Character('R'), lBaum4Links, lBaum4Rechts);
  lBaum4Links = new BinaryTree(new Character('P'));
  lBaum4Rechts = new BinaryTree(new Character('J'));
  lBaum3Rechts = new BinaryTree(new Character('W'), lBaum4Links, lBaum4Rechts);
  BinaryTree lBaum2Rechts = new BinaryTree(new Character('A'), lBaum3Links, lBaum3Rechts);
  BinaryTree lBaum1Links = new BinaryTree(new Character('E'), lBaum2Links, lBaum2Rechts);
  lBaum4Links = new BinaryTree(new Character('B'));
  lBaum4Rechts = new BinaryTree(new Character('X'));
  lBaum3Links = new BinaryTree(new Character('D'), lBaum4Links, lBaum4Rechts);
  lBaum4Links = new BinaryTree(new Character('C'));
  lBaum4Rechts = new BinaryTree(new Character('Y'));
  lBaum3Rechts = new BinaryTree(new Character('K'), lBaum4Links, lBaum4Rechts);
  lBaum2Links = new BinaryTree(new Character('N'), lBaum3Links, lBaum3Rechts);
  lBaum4Links = new BinaryTree(new Character('Z'));
  lBaum4Rechts = new BinaryTree(new Character('Q'));
  lBaum3Links = new BinaryTree(new Character('G'), lBaum4Links, lBaum4Rechts);
  lBaum4Links = new BinaryTree();
  lBaum4Rechts = new BinaryTree();
  lBaum3Rechts = new BinaryTree(new Character('O'), lBaum4Links, lBaum4Rechts);
  lBaum2Rechts = new BinaryTree(new Character('M'), lBaum3Links, lBaum3Rechts);
  BinaryTree lBaum1Rechts = new BinaryTree(new Character('T'), lBaum2Links, lBaum2Rechts);
  morseBaum = new BinaryTree(new Character(' '), lBaum1Links, lBaum1Rechts);
}
```

Die Methode `codierterText` liefert den Morsecode von `pText.` Die von ihr aufgerufene private Methode `erzeugeMorsecode` sucht den Morsecode eines einzelnen Zeichens im Morsebaum.

```
public String codierterText (String pText) {
  int lZaehler = 0;
  String lCode = "";
  while ((lZaehler < pText.length()) && (pText.charAt (lZaehler) >= 'A') &&
                                                         (pText.charAt (lZaehler) <= 'Z')) {
    lCode = lCode + erzeugeMorsecode (pText.charAt(lZaehler), morseBaum, "") + "/";
    lZaehler++;
  }
  return lCode;
}

public String erzeugeMorsecode (char pZeichen, BinaryTree pmorseBaum, String pCode) {
  if (!pmorseBaum.isEmpty()){
    if (pZeichen == ((Character) pmorseBaum.getObject()).charValue())
      return pCode;
    else {
//rekursiver Aufruf der Methode mit dem linken Teilbaum
      String lCodeLinkerTeilbaum = erzeugeMorsecode(pZeichen, pmorseBaum.getLeftTree(),
                                                                                pCode+".");
      if (lCodeLinkerTeilbaum.equals(""))
//Zeichen im linken Teilbaum nicht gefunden, rekursiver Aufruf mit dem rechten Teilbaum
        return erzeugeMorsecode(pZeichen, pmorseBaum.getRightTree(), pCode+"-");
      else
        return lCodeLinkerTeilbaum;
    }
  } else
      return "";
}
```

Hinweis für die Lehrkraft: Im Unterricht sollten auch alternative Realisierungen (z.B. mit einem Array) behandelt und mit der Binärbaum-Lösung hinsichtlich ihrer Effizienz verglichen werden.

Die Methode `decodierterText` liefert die Decodierung des Morsecodes `pCode`. Die von ihr aufgerufene private Methode `decodiertesZeichen` decodiert ein einzelnes Zeichen mit Hilfe des Morsebaums.

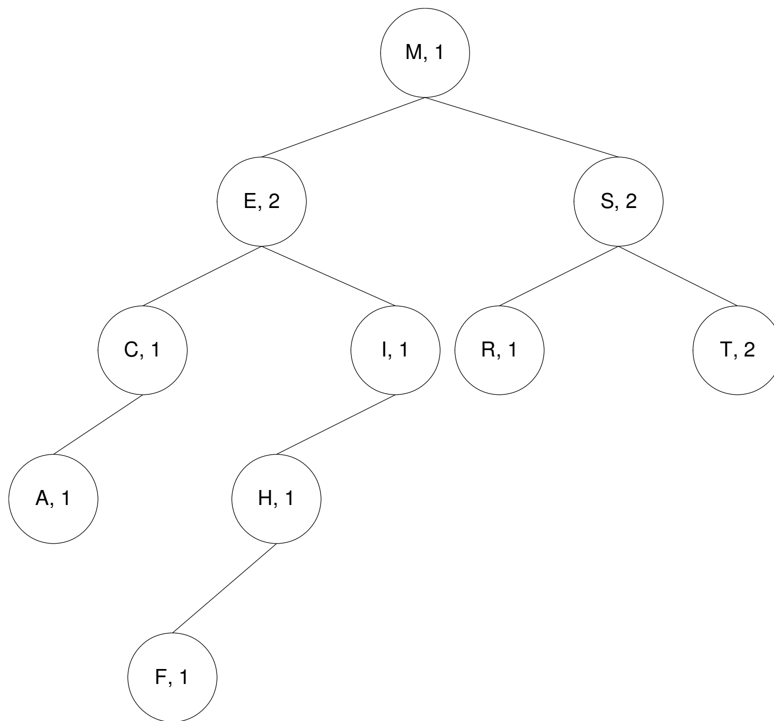
```
public String decodierterText (String pCode) {
  int lZaehler = 1;
  String lText = "";
  String lMorsezeichen;
  do {
    lMorsezeichen=pCode.substring(0, pCode.indexOf('/'));
    pCode = pCode.substring(pCode.indexOf('/')+1,pCode.length());
    if (lMorsezeichen != "") {
      lText=lText+decodiertesZeichen(lMorsezeichen);
    }
  } while (!pCode.equals(""));
      return lText;
}

public Character decodiertesZeichen( String pZeichenCode){
  String lText = "";
  int lZaehler = 0;
  BinaryTree lBaum = morseBaum;
  while (lZaehler < pZeichenCode.length()) {
    if (pZeichenCode.charAt(lZaehler) == '.')
      lBaum = lBaum.getLeftTree();
    else
      lBaum = lBaum.getRightTree();
    lZaehler++;
  }
  return (Character)lBaum.getObject();
}
```

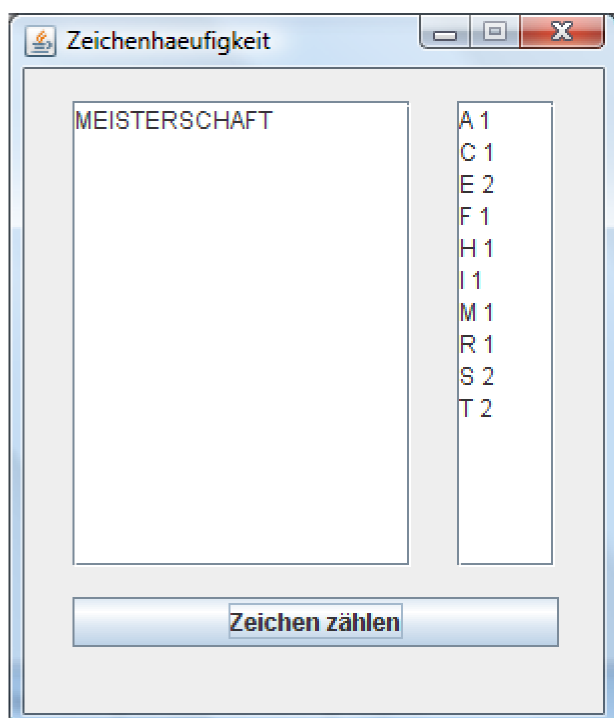
7.2 Beispiel für die Anwendung der Klasse BinarySearchTree

Aufgabenstellung

Es soll ein Programm entwickelt werden, das in der Lage ist zu zählen, wie oft jedes verwendete Zeichen in einem beliebigen Text vorkommt.
Die Zeichen mit ihrer Häufigkeit werden in einem binären Suchbaum gespeichert.
So ergibt sich für das Wort **MEISTERSCHAFT** folgender Baum:

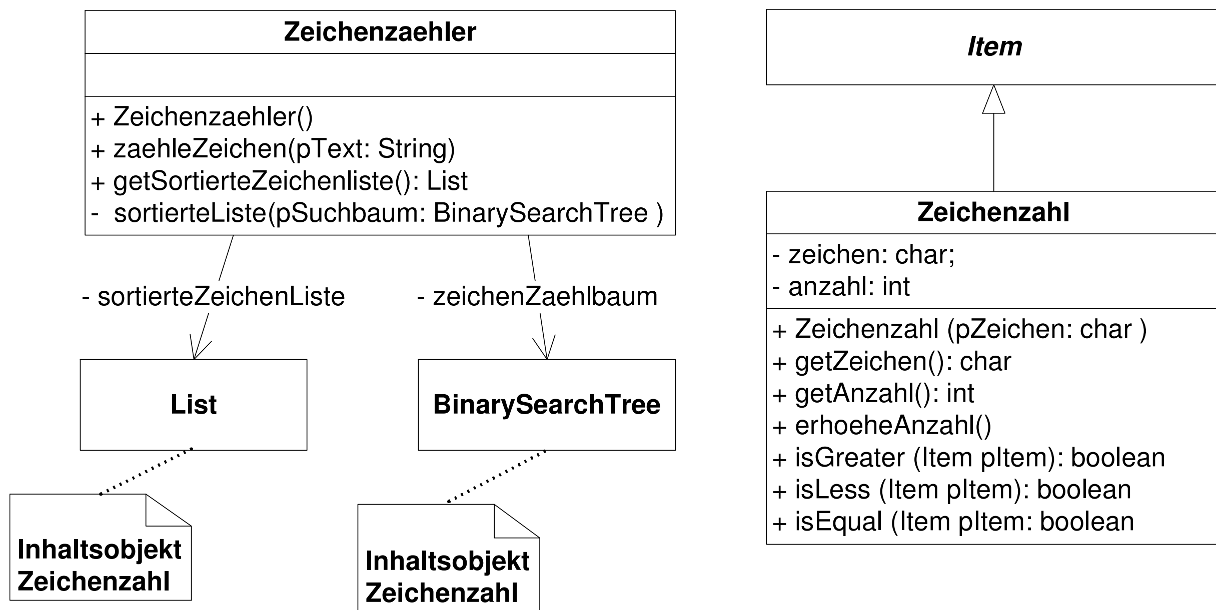


Benutzerschnittstelle



Lösungsskizze:

Implementationsdiagramm



Die Klasse Zeichenzahl

Objekte, die in einem binären Suchaum gespeichert werden, müssen zu einer Unterklasse von `Item` gehören. Die Ordnungsrelation für den Suchbaum wird durch Überschreiben der abstrakten Methoden `isEqual`, `isLess` und `isGreater` festgelegt.

```
class Zeichenzahl extends Item{
  private char zeichen;
  private int anzahl = 0;

  public Zeichenzahl(char pZeichen){
    zeichen = pZeichen;
    anzahl = 1;
  }

  public char getZeichen(){
    return zeichen;
  }

  public int getAnzahl(){
    return anzahl;
  }

  public void erhoeheAnzahl(){
    anzahl = anzahl+1;
  }

  public boolean isGreater (Item pItem){
    return (this.zeichen > ((Zeichenzahl)pItem).getZeichen());
  }

  public boolean isLess (Item pItem){
   return (this.zeichen < ((Zeichenzahl)pItem).getZeichen());
  }

  public boolean isEqual (Item pItem){
    return (this.zeichen == ((Zeichenzahl)pItem).getZeichen());
  }
}
```

Implementation einiger Methoden der Klasse Zeichenzaehler

Die Methode `zaehleZeichen` durchläuft den Text `pText` Zeichen für Zeichen. Jedes Zeichen, das zum ersten Mal vorkommt, wird mit der Häufigkeit eins im binären Suchbaum `zeichenZaehlbaum` gespeichert. Bei jedem weiteren Auftreten wird lediglich die Häufigkeit des bereits gespeicherten Zeichens um eins erhöht. Am Ende der Methode wird die private Methode `sortierteListe(zeichenZaehlbaum)` aufgerufen, die aus dem binären Suchbaum mit Hilfe eines inorder-Durchlaufs eine entsprechend der Ordnungsrelation des binären Suchbaums sortierte Liste `sortierteZeichenListe` mit den Häufigkeiten generiert.

```
public void zaehleZeichen(String pText){
    char lZeichen;
    Zeichenzahl lZeichenzahlAlt;
    zeichenZaehlbaum = new BinarySearchTree();
    for (int lZaehler=0; lZaehler<pText.length()-1;lZaehler++ ) {
      lZeichen = pText.charAt(lZaehler);
      Zeichenzahl lZeichenzahlNeu=new Zeichenzahl(lZeichen);
//Es wird geprüft, ob lZeichen bereits im Suchabum ist
      lZeichenzahlAlt=((Zeichenzahl)zeichenZaehlbaum.search(lZeichenzahlNeu));
      if (lZeichenzahlAlt==null)
 //lZeichen ist noch nicht im Suchbaum
        zeichenZaehlbaum.insert(lZeichenzahlNeu);
      else {
 //lZeichen ist bereits im Suchbaum, die Zeichenzahl wird inkrementiert
        lZeichenzahlAlt.erhoeheAnzahl();
      }
    }
    sortierteZeichenListe = new List();
    sortierteListe(zeichenZaehlbaum);
}
```

```
/** Durch Inorder-Traversierung des Suchbaums wird eine sortierte
   Liste der Zeichen mit ihren Häufigkeiten generiert.
   @param der Suchbaum
*/
private void sortierteListe(BinarySearchTree pSuchbaum){
  if (!pSuchbaum.isEmpty()) {
    sortierteListe(pSuchbaum.getLeftTree());
    sortierteZeichenListe.append(pSuchbaum.getItem());
    sortierteListe(pSuchbaum.getRightTree());
  }
}
```

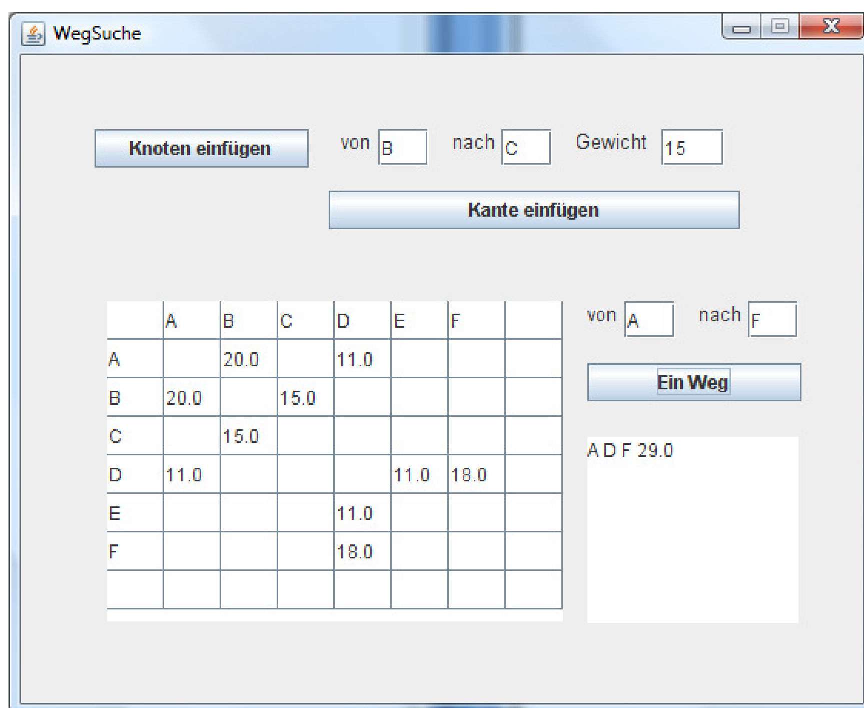
7.3 Beispiel für die Anwendung der Klassen Graph und GraphNode

Aufgabe

Es soll ein Programm entwickelt werden, das mit Hilfe eines Backtrackingalgorithmus (Tiefensuche) einen Weg zwischen zwei Knoten in einem Graphen ermittelt und die Weglänge anzeigt.

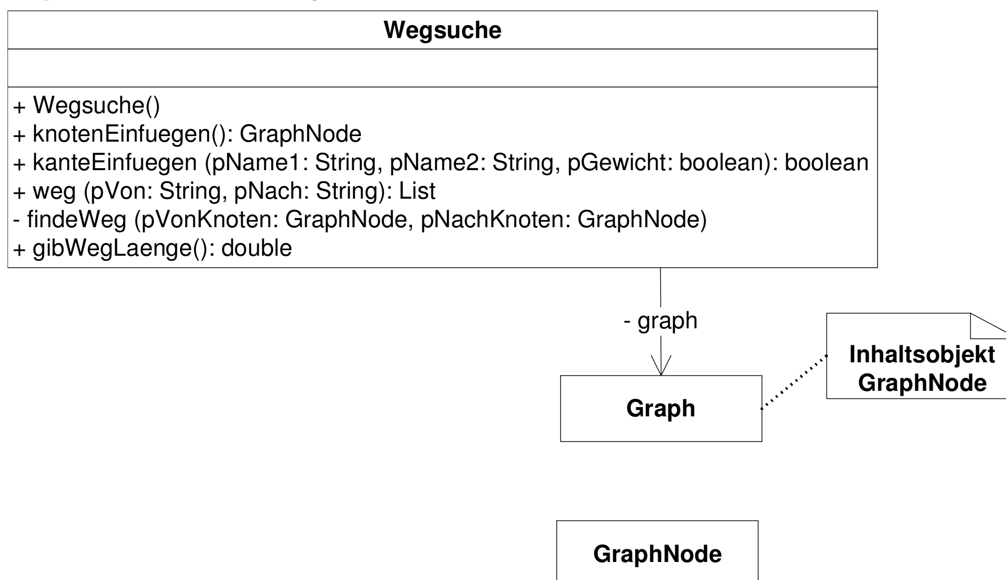
Der Graph wird durch Eingabe der Knoten (die Namen werden automatisch alphabetisch bei „A“ beginnend vergeben) und der Kanten mit ihren Gewichten erzeugt und mithilfe der Adjazenzmatrix angezeigt.

Benutzerschnittstelle



Lösungsskizze:

Implementationsdiagramm



Implementation der Klasse Wegsuche

Die Klasse `Wegsuche` verwaltet einen Graphen (Attribut `graph`), der sich mit Hilfe der Methoden `knotenEinfuegen` und `kanteEinfuegen` aufbauen lässt. Die Knotennamen werden, beginnend bei „A“, automatisch in alphabetischer Reihenfolge erzeugt. Die Methode `gibWeg` liefert in Form einer Liste von Graphknoten einen Weg zwischen zwei Knoten, deren Namen als Parameter übergeben werden. Die Funktion `gibWeg` ruft dazu die Hilfsmethode `findeWeg` auf, die mit Hilfe des Backtracking-Algorithmus‘ einen möglichen Weg zwischen den beiden Knoten berechnet und als `List` von Objekten der Klasse `graphNode` zurückgibt. Die Methode `gibWeglaenge` berechnet mit Hilfe der als Parameter übergebenen Knotenliste des Weges und den Kantengewichten zwischen den Wegeknoten die Länge des Weges.

```
public class Wegsuche {
  private Graph graph;
  private char naechsterKnoten;
  private boolean zielErreicht = false;

  public Wegsuche(){
    graph = new Graph();
    naechsterKnoten = 'A';
  }

  public GraphNode knotenEinfuegen() {
    GraphNode knoten = new GraphNode("" + naechsterKnoten);
    graph.addNode(knoten);
    naechsterKnoten = (char)(naechsterKnoten+1);
    return knoten;
  }

  public boolean kanteEinfuegen(String pName1, String pName2, double pGewicht) {
    GraphNode knoten1 = graph.getNode(pName1);
    GraphNode knoten2 = graph.getNode(pName2);
    boolean moeglich = (knoten1 != null) && (knoten2 != null);
    if (moeglich)
      graph.addEdge(knoten1,knoten2,pGewicht);
    return moeglich;
  }

  public List gibWeg(String pVon, String pNach) {
    GraphNode vonKnoten = graph.getNode(pVon);
    GraphNode nachKnoten = graph.getNode(pNach);
    List weg = null;
    if ((vonKnoten != null) && (nachKnoten != null)) {
      List knotenliste = new List();
      graph.resetMarks();
      vonKnoten.mark();
//Anfangsknoten des Weges wird in die Liste eingefügt
      knotenliste.append(vonKnoten);
      zielErreicht = false;
      weg = findeWeg(vonKnoten, nachKnoten, knotenliste);
    }
    return weg;
  }
```

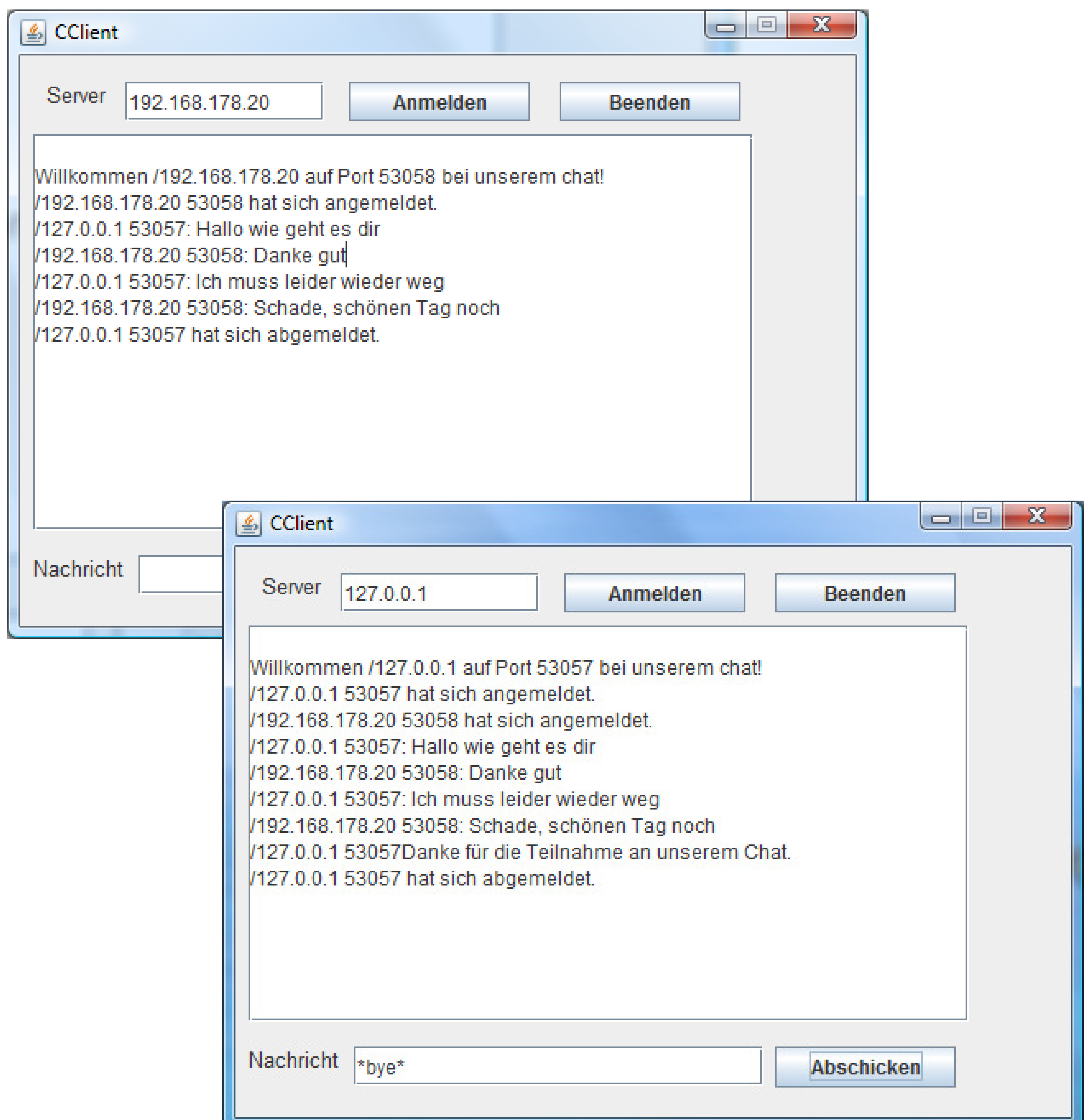
```
  private List findeWeg( GraphNode pVonKnoten, GraphNode pNachKnoten, List pWeg) {
    if (pVonKnoten != pNachKnoten) {
      List nachbarKnoten = graph.getNeighbours(pVonKnoten);
      nachbarKnoten.toFirst();
      while ((nachbarKnoten.hasAccess())&& (!zielErreicht)) {
        GraphNode knoten = (GraphNode) nachbarKnoten.getObject();
        if (!knoten.isMarked()) {
//Knoten ist noch nicht in der Wegeliste
          knoten.mark();
          pWeg.append(knoten);
//rekursiver Aufruf der Methode, um den nächsten Knoten anzufügen
          findeWeg (knoten, pNachKnoten, pWeg);
          if (!zielErreicht) {
//Sackgasse: Der zuletzt eingefügte Knoten wird aus der Liste entfernt
            knoten.unmark();
            pWeg.toLast();
            pWeg.remove();
          }
        }
        nachbarKnoten.next();
      }
    } else
       zielErreicht = true;
       if (zielErreicht)
         return pWeg;
      else
        return new List();
  }

  double gibWegLaenge(List wegliste) {
    double wegLaenge = 0;
    if (!((wegliste == null) && (!wegliste.isEmpty()))){
      wegliste.toFirst();
      GraphNode wegKnoten1 = (GraphNode) wegliste.getObject();
      wegliste.next();
      while (wegliste.hasAccess()) {
        GraphNode wegKnoten2 = (GraphNode) wegliste.getObject();
        double distanz = graph.getEdgeWeight(wegKnoten1, wegKnoten2);
        wegLaenge = wegLaenge + distanz;
        wegKnoten1 = wegKnoten2;
        wegliste.next();
      }
    }
    return wegLaenge;
  }
}
```

7.4 Beispiel für die Anwendung der Klassen Client und Server

Aufgabenstellung
Auf den Computern einer Schule soll ein interner Chat-Dienst eingerichtet werden. Dazu müssen ein Chat-Client und ein Chat-Server-Programm entwickelt werden. Zunächst muss ein Protokoll *Chat-Dienst* erstellt werden, mit dem alle Teilnehmer Nachrichten austauschen, die dann auf jedem angeschlossenen Computer angezeigt werden. Dabei sollen in der ersten Version keine Spitznamen, sondern zur Identifikation der Chat-Teilnehmer nur die IP- und Port-Nummern verwendet werden. Eine weitere Vereinfachung ergibt sich dadurch, dass zunächst alle Chat-Nachrichten an alle Teilnehmer geschickt werden und Nachrichten an einzelne Teilnehmer noch nicht möglich sein sollen.

Benutzerschnittstelle für zwei Benutzer, die miteinander chatten



Lösungsskizze:

Das Protokoll für den Chat-Dienst

Anmeldephase:
Nach der Anmeldung eines neuen Clients schickt ihm der Server die Nachricht *"Willkommen, " + IP-Nummer und Port des neu angemeldeten Clients + ", bei unserem Chat!".* Danach schickt er allen Chat-Teilnehmern die Nachricht *IP-Nummer und Port des neu angemeldeten Clients + " hat sich angemeldet."*

Arbeitsphase:
Die Clients schicken ihre Chatbeiträge als Nachricht an den Server.
Der Server verteilt jeden Beitrag an alle Chat-Teilnehmer und setzt dabei die IP-Nummer und den Port des Absenders, gefolgt von ": ", davor.

Abmeldephase:
Der Client beendet die Verbindung, indem er dem Server die Nachricht *"*bye*"* schickt. Daraufhin antwortet der Server dem Client mit *"Danke für die Teilnahme bei unserem Chat!"*, schickt an alle Chat-Teilnehmer die Nachricht *IP-Nummer und den Port des Clients, der sich abmeldet + " hat sich abgemeldet."* Zum Abschluss sendet er dem abmeldenden Client die Nachricht *"*bye*"*, worauf beide jeweils die Verbindung trennen.

Zum technischen Teil des Protokolls gehört die Festlegung auf eine Portnummer. Bei selbstdefinierten Diensten sollte sie größer als 1023 sein, hier wird 2000 benutzt.

Implementationen einiger Methoden

Die Klasse ChatClient

```
public class ChatClient extends Client {

  final String ENDE = "*bye*";
  JTextArea textbereich;

  public ChatClient(String serverIP, JTextArea pTextbereich) {
    super(serverIP, 2000);
    textbereich = pTextbereich;
  }

  public void processMessage(String pMessage){
     textbereich.setText(textbereich.getText() + "\n" + pMessage);
     if ((pMessage.equals(ENDE))
       this.close();
  }

  public void disconnect() {
    this.send(ENDE);
  }
}
```

Senden einer Nachricht

```
public void jButtonNachrichtActionPerformed(ActionEvent evt) {
    String lNachricht = jTextField2.getText();
    if (lNachricht.length() > 0) {
      chatClient.send(lNachricht);
    }
}
```

Die Klasse ChatServer

```
public class ChatServer extends Server {

  final String ENDE = "*bye*";

  public ChatServer() {
    super(2000);
  }

  public void processNewConnection(String pClientIP, int pClientPort) {
    this.send(pClientIP, pClientPort, "Willkommen " + pClientIP + " auf Port "
                                        + pClientPort + " bei unserem chat!");
    this.sendToAll(pClientIP + " " + pClientPort + " hat sich angemeldet.");
  }

  public void processMessage(String pClientIP, int pClientPort, String pMessage) {
    if (pMessage.equals(ENDE))
      this.closeConnection(pClientIP, pClientPort);
    else
      this.sendToAll(pClientIP + " " + pClientPort + ": " + pMessage);
  }

  public void processClosedConnection(String pClientIP, int pClientPort) {
    this.send(pClientIP, pClientPort, pClientIP + " " + pClientPort +
                                  "Danke für die Teilnahme an unserem Chat.");
    this.sendToAll(pClientIP + " " + pClientPort + " hat sich abgemeldet.");
    this.send(pClientIP, pClientPort, ENDE);
  }
}
```

7.5 Beispiel für eine Datenbankanwendung

Aufgabenstellung
Das Fernlerninstitut IFA (**I**nformatik **F**ür **A**lle) bietet eine Reihe von Kursen an: z.B.

- Kursnummer 100, Einführung in Java, Kosten 200,- €
- Kursnummer 113, Datenbanksysteme, Kosten 250,- €
- ...

Das Institut möchte wichtige Verwaltungsbereiche auf EDV umstellen. Dazu werden alle angebotenen Kurse und alle Kunden mit Adressen gespeichert. Zusätzlich wird die Information gespeichert, welche Kurse ein Kunde gebucht hat und welche Geldbeträge zu welchem Zeitpunkt überwiesen wurden.

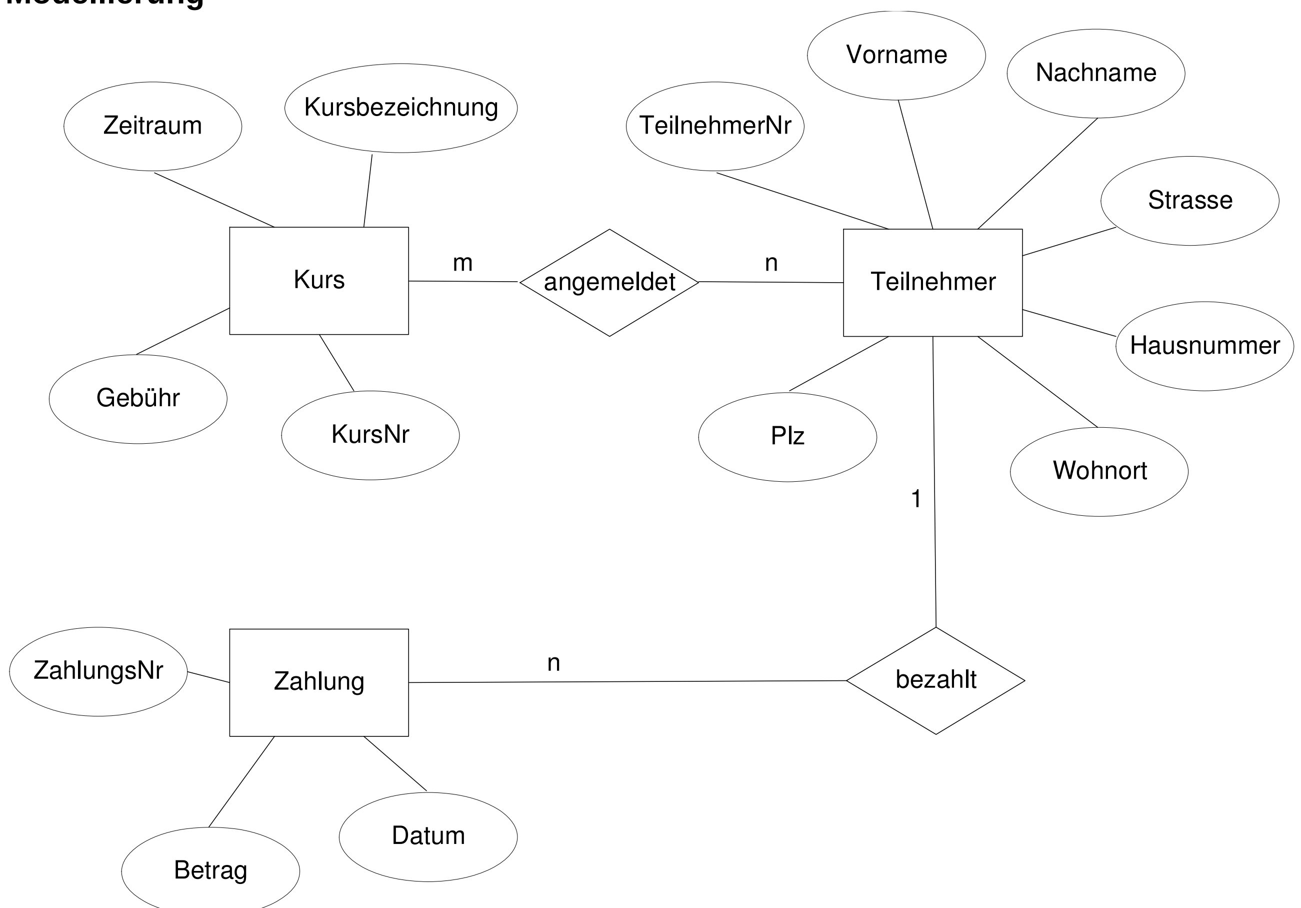
- Modellieren Sie für das Fernlerninstitut ein ER-Diagramm. Geben Sie auch die Kardinalitäten an und erläutern Sie Ihre Entscheidungen.

- Überführen Sie das Diagramm in ein Datenbankschema.

- Die Geschäftsleitung benötigt folgende Information:
    - Die Anzahl der Kunden, die am Kurs „Einführung in Java" teilnehmen.
    - Alle Kundennummern mit der Summe der eingegangenen Zahlungen.

  Geben Sie die entsprechenden SQL-Befehle an.

Modellierung



Ein Kunde kann sich für m Kurse anmelden, an einem Kurs können n Kunden teilnehmen. Ein Kunde macht mindestens eine Überweisung, eine Zahlung kann eindeutig einem Kunden zugeordnet werden.

Datenbankschema
Jede Entitätsmenge muss in eine Tabelle überführt werden, jede Beziehungsmenge kann in eine Tabelle überführt werden.

Kurs (<u>KursNr</u>, Kursbezeichnung, Gebühr, Zeitraum)
Teilnehmer (<u>TeilnehmerNr</u>, Vorname, Nachname, Strasse, Hausnr, Plz, Wohnort)
angemeldet (↑KursNr, ↑TeilnehmerNr)
Zahlung (<u>ZahlungsNr</u>, ↑TeilnehmerNr, Betrag, Datum)

SQL-Abfragen

```
SELECT COUNT(*)
     FROM angemeldet, Kurs
     WHERE Kurs.Kursbezeichnung = "Einführung in Java"
     AND angemeldet.KursNr = Kurs.KursNr
```

```
SELECT angemeldet.TeilnehmerNr, SUM(Zahlung.Betrag)
     FROM angemeldet, Zahlung
     WHERE angemeldet.TeilnehmerNr = Zahlung.TeilnehmerNr
     GROUP BY angemeldet.TeilnehmerNr
```