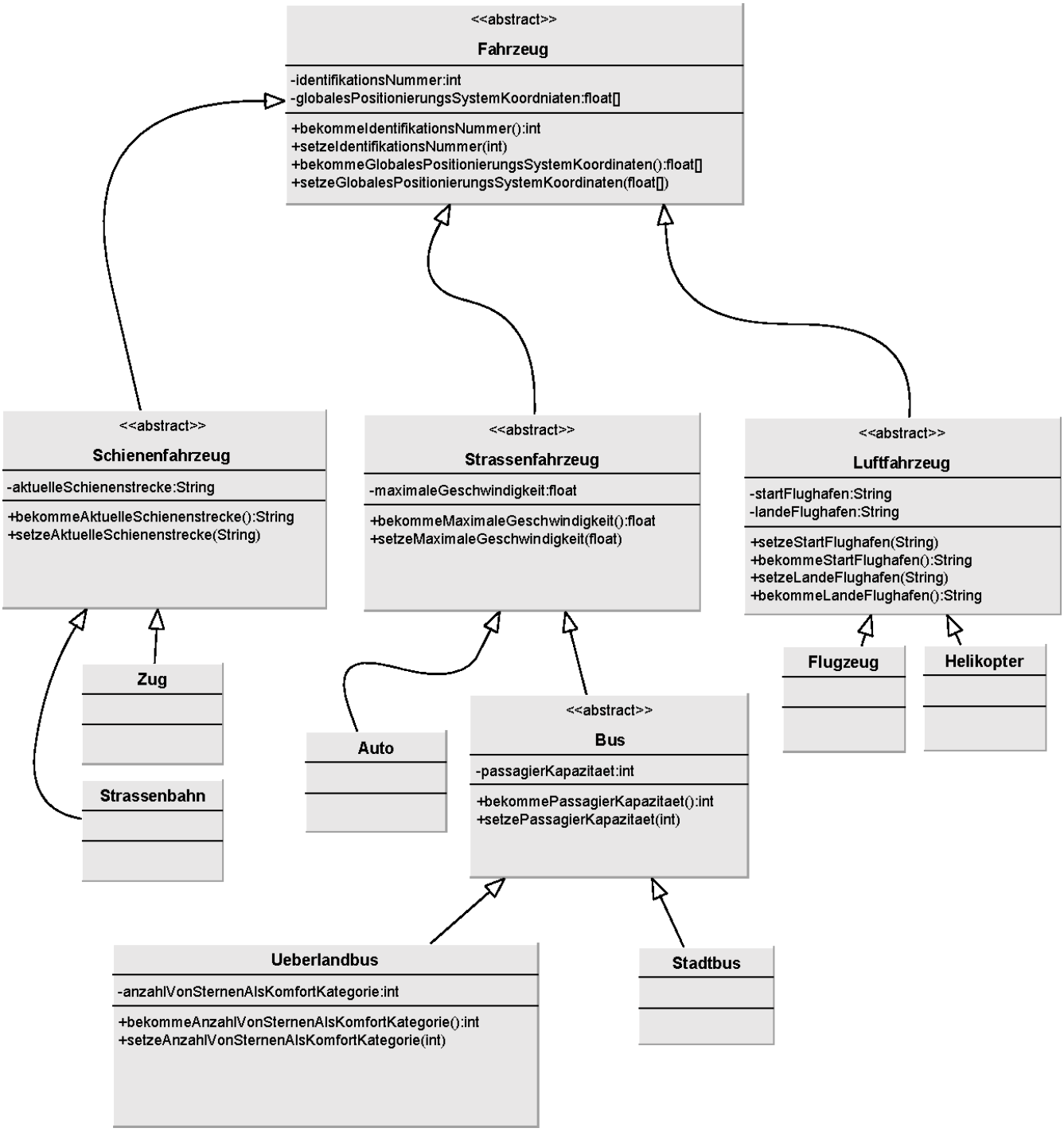
Übungsgruppe 3; Zweiergruppe 23
274196 Lucas Brutschy & 273784 Philipp Fischer

Übung Programmierung
Blatt 7

Aufgaben 1 und 2 – Mergesort und Graphen

Der Quelltext zu den Aufgaben befindet sich im Anhang.

Aufgabe 3a – UML Diagramm



Übungsgruppe 3; Zweiergruppe 23
274196 Lucas Brutschy & 273784 Philipp Fischer

Aufgabe 4

a)

Der Quelltext der zu implementierenden Methode befindet sich im Anhang

b)

Man könnte der Klasse `Item` eine entsprechende abstrakte Methode `getBruttoPrice(float voucher)` hinzufügen, die den entsprechenden Preis des Artikels zurückliefern muss. Hierbei werden Steuersätze und Gutschriften gemäß Definition berücksichtigt. Man würde also diese abstrakte Methode `getBruttoPrice(float)` in den Klassen `Buch`, `Kleidung` und `Lebensmittel` implementieren und in Ausnahmefällen wie bei Bier und Kaffee in den entsprechenden Unterklassen überschreiben, um entsprechende Ausnahmeregelungen zu berücksichtigen. Die Implementierung der Methode `getBruttoPrice(float)` in der Klasse `Buch` würde also beispielsweise den Kundenrabatt ignorieren und einen Steuersatz von 10% berechnen.
So bräuchte man in der Methode `berechneSumme(Item[],float)` nur noch die zurückgegebenen Werte von den Aufrufen der Methode `getBruttoPrice` addieren.

MergeSort.java

```java
/**
 * SHEET 7 - EXERCISE 1
 * MergeSort implementation
 *
 * @author 273784 Philipp Fischer & 274196 Lucas Brutschy
 * @version 04.12.2006
 * platform: J2RE 1.5.0_08-b03, Linux 2.6.17-gentoo
*/
public class MergeSort {
    /**
     * This method calls the real recursive msort method by provding
     * the left and right limits of the array
     * @param data
     */
    public static void msort(int[] data) {
        if(data.length <= 1) return;
        msort(data, 0, data.length - 1);
    }

    /**
     * This method implements the recursive Mergesort algorithm.
     * It ascendingly sorts the array within the the given range
     * @param data The array to sort
     * @param leftLimit The left limit of the range to sort
     * @param rightLimit The right limit of the range to sort
     */
    private static void msort(int[] data, int leftLimit, int rightLimit) {
        // If we have only one element to sort, it's sorted already
        if(leftLimit >= rightLimit) return;

        /* Here we calculate the split-index t
         * n = rightLimit - leftLimit + 1
         * t = floor(n/2)-1 + leftLimit
         *   = floor(rightLimit + leftLimit + 1) / 2 - 1
         *   = floor(leftLimit + rightLimit - 1) / 2
         */
        int t = (leftLimit+rightLimit-1) / 2;

        /* "Divide and Conquer"
         * Call the algorithm recursively to operate on the two ranges
         * [leftlimit, t] and [t+1, rightLimit]
         */
        msort(data, leftLimit, t);
        msort(data, t+1, rightLimit);

        /* As we now have two sorted parts of the range,
         * merge them together by iterating through the parts
         * and always putting the lower element to a buffer array
         * wich will replace the part [leftLimit, rightLimit] on
         * the actual array afterwards
         */
        int[] buffer = new int[rightLimit-leftLimit+1];
        int leftPointer = leftLimit, rightPointer = t+1, outPointer = 0;
        while(leftPointer <= t && rightPointer <= rightLimit) {
            if(data[leftPointer] <= data[rightPointer]) {
                buffer[outPointer++] = data[leftPointer++];
            } else {
                buffer[outPointer++] = data[rightPointer++];
            }
        }

        // If there are elements left on one of the two parts, get those too
        while(leftPointer <= t) {
            buffer[outPointer++] = data[leftPointer++];
        }
        while(rightPointer <= rightLimit) {
            buffer[outPointer++] = data[rightPointer++];
        }

        // Overwrite the unsorted part in the main-array with the sorted buffer
        for(int i = leftLimit; i<=rightLimit; i++) {
            data[i] = buffer[i-leftLimit];
        }
    }
}
```

SortTestBed.java

```java
import java.util.Random;

/**
 * SHEET 7 - EXERCISE 1
 * MergeSort Benchmark application
 *
 * @author 273784 Philipp Fischer & 274196 Lucas Brutschy
 * @version 04.12.2006
 * platform: J2RE 1.5.0_08-b03, Linux 2.6.17-gentoo
*/
public class SortTestBed {
    /**
     * This method generates an array filled with pseudo random numbers
     * @param numberCount The count of numbers to generate
     * @return The array filled with random numbers
     */
    public static int[] randomNumbers(int numberCount) {
        if(numberCount < 1) return null;
        Random generator = new Random();
        int[] numbers = new int[numberCount];
        for(int i = 0; i < numberCount; i++) {
            numbers[i] = generator.nextInt();
        }
        return numbers;
    }

    /**
     * This method measures the time it takes to sort a ceratain
     * number of integers using the mergesort algorithm
     * @param numberOfElements The number of integers
     * @return The time it took to sort the numbers in seconds
     */
    public static float measureSortingTime(int numberOfElements) {
        int[] toSort = randomNumbers(numberOfElements);

        long startTime = System.currentTimeMillis();
        MergeSort.msort(toSort);
        return (float)(System.currentTimeMillis() - startTime) / 1000.0f;
    }

    /**
     * The entry point of the program
     * @param args unused
     */
    public static void main(String[] args) {
        // Specify how many tests you would like to perform
        int testCount = 7;

        // This loop performs the tests and saves the results into an array
        String[][] statisticsTable = new String[testCount][2];
        for(int i = 0; i < testCount; i++) {
            int elementCount = (1<<i)*20000;
            statisticsTable[i][0] = String.format("%12d", elementCount);
            statisticsTable[i][1] = String.format("%7.2f", measureSortingTime(elementCount));
        }

        // Now we output the result table to the console
        System.out.println("+------------+-------+");
        System.out.println("|Elementcount|Seconds|");
        System.out.println("+============+=======+");
        for(int y = 0; y < statisticsTable.length; y++) {
            System.out.print("|");
            for(int x = 0; x < 2; x++) {
                System.out.print(statisticsTable[y][x]+"|");
            }
            System.out.println();
        }
        System.out.println("+------------+-------+");

    }
}

```

Element.java

```java
/** Datentyp fuer Elemente einer linearen Liste, deren Werte
 *  verglichen werden koennen.
 * @see Liste
 * @see GraphNode
 */
public class Element {

    GraphNode wert;
    Element next; //Element ist eine rekursive Datenstruktur.

    /** erzeugt ein neues Element ohne Nachfolger.
     *   @param wert Wert, den das neue Element erhalten soll
     */
    public Element (GraphNode wert) {
        this.wert = wert;
        next = null;
    }

    /** erzeugt ein neues Element mit Nachfolger.
     *   @param wert Wert, den das neue Element erhalten soll
     *   @param next Nachfolgerelement des neuen Elements
     */
    public Element (GraphNode wert, Element next) {
        this.wert = wert;
        this.next = next;
    }

    /** @return Wert des Elements
     */
    public GraphNode getWert () {
        return wert;
    }

    /** @param wert Wert, den das Element erhalten soll
     */
    public void setWert (GraphNode wert) {
        this.wert = wert;
    }
    /** @return Nachfolger des Elements
     */
    public Element getNext () {
        return next;
    }

    /** @param next Nachfolger, den das Element erhalten soll
     */
    public void setNext (Element next) {
        this.next = next;
    }

    public String toString () {
        return wert.toString();
    }


    /** vergleicht zwei Elemente inhaltlich nach ihrem Wert
     *  @param zuvergleichen das Element, mit dem
     *         das aktuelle Element verglichen werden soll
     *  @return true, falls die Werte der beiden Elemente
     *          inhaltlich gleich sind und sonst false
     */
    public boolean gleich (Element zuvergleichen) {
        return wert.gleich (zuvergleichen.wert);
    }

}
```

Liste.java

```java
/** Datentyp fuer lineare Listen von GraphNodes
 * @see Element
 * @see GraphNode
 */
public class Liste {

    /** Attribut, das auf das erste Element der Liste zeigt
     */
    private Element kopf;

    /** Attribut, das auf das letzte Element der Liste zeigt
     */
    private Element schluss;

    /** erzeugt eine neue leere Liste
     */
    public Liste () {
        kopf = schluss = null;
    }

    /** gibt das erste Element der List zurueck
     *   @return Das erste Element in der Liste, falls
     *           es ein solches gibt. Sonst wird null zurueckgegeben.
     */
    public Element getFirst() {
        return kopf;
    }
    public Element getLast() {
        return schluss;
    }

    /** sucht nach einem Element in der Liste.
     *   @param wert Der Wert des gesuchten Elements.
     *   @return Das erste Element in der Liste mit diesem Wert, falls
     *           es ein solches gibt. Sonst wird null zurueckgegeben.
     */
    public Element suche (GraphNode wert) {
        return suche (wert, kopf);
    }

    /** sucht nach einem Element in einer vorgegebenen Liste.
     *   @param wert Der Wert des gesuchten Elements.
     *   @param kopf Der Kopf der Liste, in der gesucht wird.
     *   @return Das erste Element in jener Liste mit diesem Wert, falls
     *           es ein solches gibt. Sonst wird null zurueckgegeben.
     */
    private static Element suche (GraphNode wert, Element kopf) {
        if (kopf == null) return null;
        else if (wert.gleich(kopf.wert)) return kopf;
        else return suche (wert, kopf.next);
    }

    /** erzeugt einen String, der die Elemente der Liste von vorne nach hinten
     *   aufzaehlt.
     *   @return Die Liste als Zeichenkette
     */
    public String toString () {
        return  "( " + durchlaufe(kopf) + ")";
    }

    /** gibt den Inhalt der Liste (von vorne nach hinten) auf dem Bildschirm aus.
     *   @return Die Liste als Zeichenkette
     */
    public void drucke() {
        System.out.println (this);
    }


    /** erzeugt einen String, der aus allen Elementen einer vorgegebenen Liste (von vorne
     *   nach hinten) besteht.
     *   @param kopf Der Kopf der zu durchlaufenden Liste.
     *   @return Die Zeichenkette aller Elemente jener Liste.
     */
    private static String durchlaufe (Element kopf) {
        if (kopf != null) return kopf.wert + " " + durchlaufe(kopf.next);
        else return "";
    }

    /** erzeugt einen String, der die Elemente der invertieren Liste
     *   (d.h., von hinten nach vorne) aufzaehlt.
```

Liste.java

```
81          *   @return Die invertierte Liste als Zeichenkette
82          */
83         public String toStringRueckwaerts () {
84             return  "(" + durchlaufeRueckwaerts(kopf) + " )";
85         }
86
87         /** gibt den Inhalt der invertierten Liste (d.h., von hinten nach
88          *   vorne) auf dem Bildschirm aus.
89          *   @return Die Liste als Zeichenkette
90          */
91         public void druckeRueckwaerts() {
92             System.out.println (this.toStringRueckwaerts());
93         }
94
95
96         /** erzeugt einen String, der aus allen Elementen einer
97          *   invertierten vorgegebenen Liste (von hinten
98          *   nach vorne) besteht.
99          *   @param kopf Der Kopf der zu durchlaufenden Liste.
100         *   @return Die Zeichenkette aller Elemente jener invertierten Liste.
101         */
102        private static String durchlaufeRueckwaerts (Element kopf) {
103            if (kopf != null) return durchlaufeRueckwaerts(kopf.next) + " " + kopf.wert;
104            else return "";
105        }
106
107
108        /** fuegt ein Element vorne in die Liste ein.
109         *   @param wert Der Wert des einzufuegenden Elements.
110         */
111        public void fuegeVorneEin (GraphNode wert) {
112            if  (kopf == null)  kopf = schluss = new Element (wert);
113            else kopf = new Element (wert, kopf);
114        }
115
116        /** fuegt ein Element hinten in die Liste ein.
117         *   @param wert Der Wert des einzufuegenden Elements.
118         */
119        public void fuegeHintenEin (GraphNode wert) {
120            if   (kopf == null) kopf = schluss = new Element (wert);
121            else {
122                schluss.next = new Element (wert);
123                schluss = schluss.next;
124            }
125        }
126
127
128        /** loescht die komplette Liste.
129         */
130        public void loesche () {
131            kopf = schluss = null;
132        }
133
134        /** loescht das erste Element mit dem angegebenen Wert aus der Liste.
135         *   @param wert Der Wert des zu loeschenden Elements.
136         */
137        public void loesche (GraphNode wert) {
138            kopf = loesche (wert, kopf);
139        }
140
141        /** loescht das erste Element mit dem angegebenen Wert, das ab einem vorgegebenen
142         *   Element in der Liste auftritt.
143         *   @param wert Der Wert des zu loeschenden Elements.
144         *   @param element Das Element der Liste, ab dem erst geloescht werden kann.
145         *   @return Das erste Element der Teilliste ab dem vorgegebenen Element,
146         *           wobei der zu loeschende Wert geloescht wurde.
147         */
148        private static Element loesche (GraphNode wert, Element element) {
149            if (element == null) return null;
150            else if (wert.gleich(element.wert)) return element.next;
151            else {
152                element.next = loesche (wert, element.next);
153                return element;
154            }
155        }
156
157
158    }
159
160
```

GraphNode.java

```java
/**
 * SHEET 7 - EXERCISE 2
 * Graph Data Structure - Node Class
 * represents a node of a graph and its edges
 * @see Graph
 * @see Liste
 * @see Element
 * @author 273784 Philipp Fischer & 274196 Lucas Brutschy
 * @version 10.12.2006
 * platform: J2RE 1.5.0_08-b03, Linux 2.6.17-gentoo
*/
public class GraphNode {
    /** a list of GraphNodes connected this GraphNode
     */
    private Liste adjListe;

    /** the ID of this Node
     */
    private int id;

    /** a reference pointing to the GraphNode that was returned by the last get*Neighbor
     */
    private Element current;

    /** flag indicating if this Node has been visited before
     */
    private boolean visitFlag;

    /** creates a GraphNode
     * @param id id of the new GraphNode
     */
    public GraphNode(int id) {
        this.id = id;
        adjListe = new Liste();
    }

    /** attribute access function
     * @return ID of the node
     */
    public int getID() {
        return id;
    }

    /** creates an edge to another GraphNode, if it doesnt exist yet
     * @param k the other GraphNode
     */
    public void connectToNode( GraphNode k) {
        if((k!=null)&&(adjListe.suche(k)==null)) {
            adjListe.fuegeHintenEin(k);
        }
    }

    /** returns the first neighbor in the list
     * @return a reference to the first neighbor
     */
    public GraphNode getFirstNeighbor() {
        current = adjListe.getFirst();
        return current.wert;
    }

    /** returns the next neighbor in the list. Call getFirstNeighbor() first
     * @return a reference to the next neighbor
     */
    public GraphNode getNextNeighbor() {
        if(current!=null) {
            current = current.next;
        }
        return (current==null)?null:current.wert;
    }

    /** attribute acces function
     * @return the visit flag
     */
    public boolean getVisitFlag() {
        return visitFlag;
    }

    /** attribute acces function
     * @param flag the visit flag
     */
    public void setVisitFlag(boolean flag) {
        visitFlag = flag;
    }

    /** compares the node and a given node
     * @param zuvergleichen the node that will be compared to the current node
     */
    public boolean gleich(GraphNode zuvergleichen) {
        return (zuvergleichen.getID()==id);
    }
}
```

Graph.java

```java
/**
 * SHEET 7 - EXERCISE 2
 * Graph Data Structure - Management Class
 * represents a graph
 * @see GraphNode
 * @author 273784 Philipp Fischer & 274196 Lucas Brutschy
 * @version 10.12.2006
 * platform: J2RE 1.5.0_08-b03, Linux 2.6.17-gentoo
*/
public class Graph {
    /** the array in which references to all nodes are stored
     */
    private GraphNode[] nodes;

    /** the number of nodes currently stored
     */
    private int length;

    /** prints the ID of the current object and calls itself for every object
     *  thats connected to the current object
     *  @param current reference to the current GraphNode object
     */
    private void recurseBesuch(GraphNode current) {
        if(current==null) return;
        System.out.println(current.getID());
        current.setVisitFlag(true);
        GraphNode iteration = current.getFirstNeighbor();
        do {
            if(!iteration.getVisitFlag()) {
                recurseBesuch(iteration);
            }
        } while((iteration = current.getNextNeighbor())!=null);
    }

    /** creates a graph, allocates memory for up to 100 nodes
     */
    public Graph () {
        this(100);
    }

    /** creates a graph, allocates memory for up nodes
     *  @param nodeArraySize number of nodes to allocate space for
     */
    public Graph (int nodeArraySize) {
        nodes = new GraphNode[nodeArraySize];
        length = 0;
    }

    /** creates a graph described by a matrix
     *  @param nodeArraySize the matrix describing the edges of the graph
     */
    public Graph (boolean[][] matrix) {
        this(matrix.length);

        // create nodes
        for(int i=0;i<matrix.length;i++) {
            newNode();
        }

        // create edges (bidiretional)
        for(int i=0;i<matrix.length;i++) {
            if(matrix[i].length==matrix.length) {
                for(int j=0;j<matrix[i].length;j++) {
                    if(matrix[i][j]) {
                        newEdge(i,j);
                    }
                }
            }
        }
    }

    /** creates a new node
     *  @return ID of the new element
     */
    public int newNode() {
        // if the current array of node is full, double the size
        if(length==nodes.length) {
            GraphNode[] newArray = new GraphNode[nodes.length*2];
            System.arraycopy(nodes,0,newArray,0,nodes.length);
            nodes=newArray;
```

Graph.java

```
81          }
82          nodes[length] = new GraphNode(length);
83          length++;
84          return (length-1);
85      }
86
87      /** creates a new edge (always bidirectional)
88       *  @param ID1 the id of the first element
89       *  @param ID2 the id of the second element
90       */
91      public void newEdge(int ID1,int ID2) {
92          if((ID1<length)&&(ID2<length)) {
93              nodes[ID1].connectToNode(nodes[ID2]);
94              nodes[ID2].connectToNode(nodes[ID1]);
95          }
96      }
97
98      /** node array access method
99       *  @param ID the ID of the object that should be returned
100      *  @return reference to the GraphNode object identified by ID, or null if no such item exists
101      */
102     public GraphNode getNode(int ID) {
103         if(ID<length) {
104             return nodes[ID];
105         }
106         return null;
107     }
108
109     /** starts a recursive process that visits every Node and prints the ID
110      *  @param the ID of the entry point
111      */
112     public void besuche (int ID) {
113         clearVisitFlag();
114         recurseBesuch(getNode(ID));
115     }
116
117     /** clears the visit flag of all nodes
118      */
119     public void clearVisitFlag() {
120         for(int i=0;i<length;i++) {
121             nodes[i].setVisitFlag(false);
122         }
123     }
124
125 }
```

berechneSumme.method.java

```java
public static float berechneSumme(Item[] list, float voucher) {
    final float reducedTax = 1.10f;
    final float normalTax = 1.20f;
    final float coffeeSurcharge = 1.05f;

    float total = 0;
    float voucherFactor = ((100f - voucher) / 100f);

    for(int i = 0; i < list.length; i++) {
        // Buecher:
        if(list[i] instanceof Buch) {
            total += list[i].getPrice() * reducedTax;
        // Lebensmittel:
        } else if(list[i] instanceof Lebensmittel) {
            // Beginnen mit Betrag abzgl. Rabatt
            float currentFoodPrice = list[i].getPrice() * voucherFactor;

            if(list[i] instanceof Bier || list[i] instanceof Kaffee) {
                // Wenn Bier oder Kaffee, voller Steuersatz
                currentFoodPrice *= normalTax;

                // Bei Kaffee zzgl. 5%
                if(list[i] instanceof Kaffee) {
                    currentFoodPrice *= coffeeSurcharge;
                }
            } else {
                // Wenn Lebensmittel (aber nicht Bier / Kaffee), reduzierte Steuer
                currentFoodPrice *= reducedTax;
            }
            total += currentFoodPrice;

        // Kleidung:
        } else if(list[i] instanceof Kleidung) {
            total += list[i].getPrice() * normalTax;
        }
    }

    return total;
}

```