

	Familienname:	
	Matrikelnummer:	Kennzahl:
	Lehrveranstaltung: Objektorientiertes Programmieren	

Schriftliche Prüfung			Datum: 28.06.2016	Ort: GM Audi. Max.
	Punkte		Schriftliche Note:	
Bsp1	31,5		B3	Negative Note: die erforderliche Anzahl von _____ Punkten wurde nicht erreicht.
Bsp2	8+4=12			
Bsp3	8			
Bsp4	11,5			
	63,0			
B	30			
	93			

Beispiel 1 - Binärbaum (55 Punkte)

Ihre Aufgabe besteht darin Teile eines binären Baumes zu implementieren. Ein Binärbaum ist dadurch definiert, dass jeder Knoten maximal zwei, einen linken und einen rechten, Kindknoten haben kann. Zusätzlich ist der Binärbaum dieses Beispiels noch geordnet. Geordnet heißt, dass Knoten abhängig von ihrem Wert verwaltet werden. Das heißt, dass Knoten nicht willkürlich, sondern nur geordnet in den Baum eingesetzt bzw. gelöscht werden dürfen. Kleinere Knoten werden dabei links von einem Betrachtungsknoten geführt, größere rechts von einem Betrachtungsknoten. Abbildung 1 zeigt wie ein solcher Binärbaum aufgebaut ist.

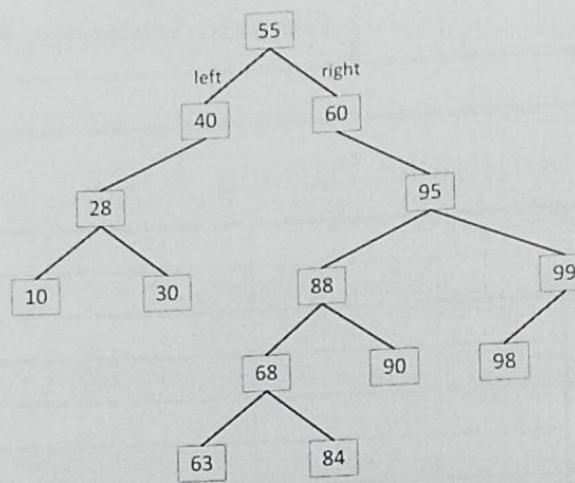
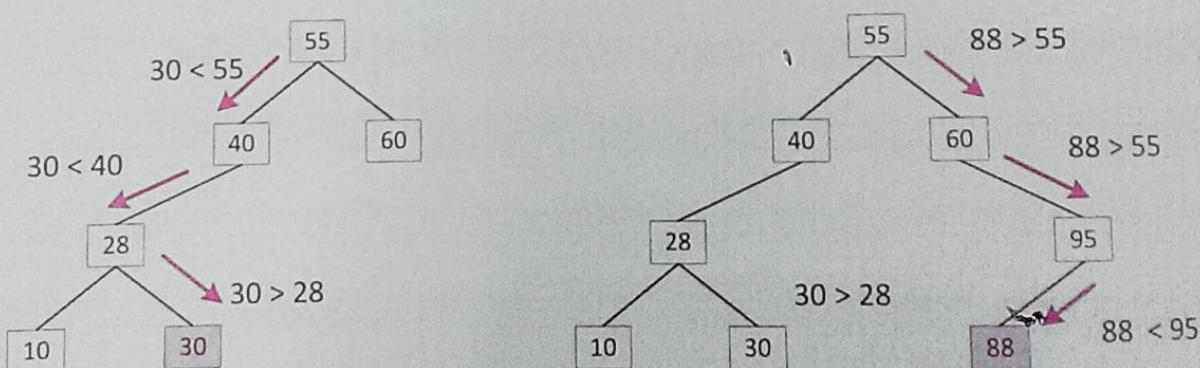


Abbildung 1 - Binärbaum

Auf einem Binärbaum sind mehrere Operationen möglich, die es erlauben den Baum zu ändern. Unter anderem ist es möglich Elemente einzufügen, zu löschen und zu aktualisieren.

Einfügen von Elementen erfolgt dabei nach dem in Abbildung 2 gezeigten Schema. Ein neuer Wert wird dabei links von einem Betrachtung-Knoten eingefügt, sofern dieser kleiner als der Knotenwert ist. Ist der neue Wert größer als der Wert des Betrachtung-Knoten, so wird er rechts davon eingefügt. Dieses Schema wird solange fortgesetzt, bis ein Knoten gefunden wurde, welcher noch einen freien Platz für einen passenden Kindknoten hat.



Einfügen des neuen Werts (30) in den Baum.

Einfügen des neuen Werts (88) in den Baum.

Abbildung 2 - Einfügen in Binärbaum

Die Knotenwerte eines Baums können aktualisiert werden. Damit der Baum aber nicht korrumptiert wird, ist das Aktualisieren des Knotenwertes eines Betrachtungs-Knotens nur in einem gewissen Intervall möglich. Dieses Intervall wird durch den größten Wert des linken Sub-Baums, sofern vorhanden, und dem kleinsten Wert des rechten Sub-Baums, sofern vorhanden, bestimmt. Ist kein linker Sub-Baum vorhanden, so wird der Wert des Betrachtungs-Knotens als untere Grenze verwendet. Ist kein rechter Sub-Baum vorhanden, so wird der Wert des Betrachtungs-Knotens als obere Grenze verwendet. Abbildung 3 zeigt wie dieses Intervall bestimmt wird.

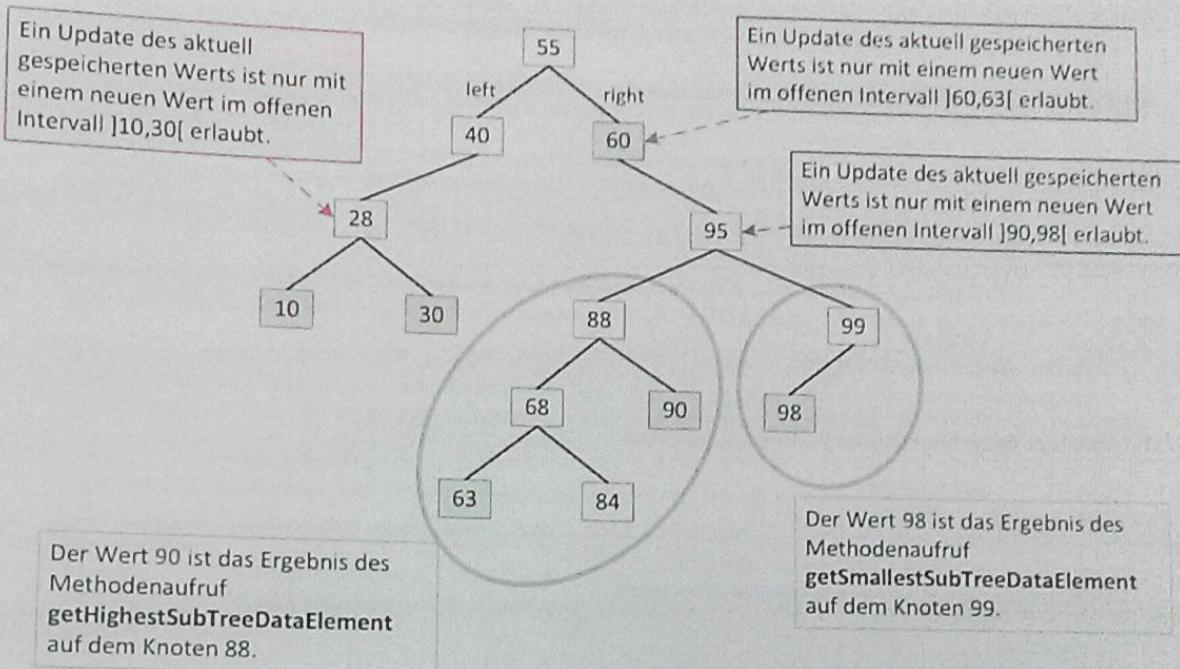


Abbildung 3 - Aktualisieren eines Knotenwertes

Weiters können Knoten gelöscht werden. Zur Vereinfachung können Knoten nur dann gelöscht werden, wenn sie ein Blatt sind und daher keine Kindknoten haben.

Der Binärbaum kann seine Knoteninhalte als absteigend sortierte Liste ausgeben. Für den Baum aus Abbildung 4 ergibt dies folgende Liste:

99	98	95	90	88	84	68	63	60	55	40	30	28	10
----	----	----	----	----	----	----	----	----	----	----	----	----	----

Abbildung 4 - Absteigend sortierte Liste

Implementierung



Die im Klassendiagramm (Abbildung 5) **rot** markierten Klassen sind **Klassen**, von denen Sie Teile implementieren müssen. Die dazugehörigen Beschreibungen der Methoden finden sie im Abschnitt **Methoden-Beschreibungen**. Weiters existieren zu diesen Methoden-Beschreibungen zugehörige Sequenzdiagramme. Bitte beachten Sie, dass die Sequenzdiagramme nur Ausschnitte aus der gesamt Funktionalität der Methode darstellen. Die vollständige Funktionalität ist aus der Beschreibung der Methode zu entnehmen.

- Das zu dem Binärbaum gehörige Klassendiagramm ist in Abbildung 5 auf Seite 9 angegeben. Die darin rot markierten Klassen sind Klassen, von denen Sie Teile implementieren müssen.
- Genauere Beschreibungen der zu implementierenden Teile sind im Abschnitt **Methoden-Beschreibungen** (nächste Seite) gegeben.
- Die dabei zu berücksichtigenden Sequenzdiagramme finden Sie auf den Seiten 10 bis 12. Diese Sequenzdiagramme bieten jeweils nur einen Ausschnitt aus der Gesamtfunktionalität der Methode. Die vollständige Funktionalität ist aus der Methoden-Beschreibung zu entnehmen.

Name: _____

Methoden-Beschreibungen

Implementieren Sie

- die komplette **TreeException**-Klasse,
- sowie den Header der **TreeNode** Klasse. Der Header umfasst die Deklaration der Klasse und der Attribute.

Erstellen Sie weiters

- eine **Observer**-Klasse, die alle Änderungen auf der Konsole ausgibt.

Implementieren Sie von den folgenden Klassen nur die **rot** markierten Methoden.**Tree-Klasse:****Collection<IComparable>****flattenToSortedList()**

Gibt alle Knotenwerte des Baums in absteigender Reihenfolge zurück. Das zugehörige Sequenzdiagramm ist in Abbildung 6 dargestellt.

Node-Klasse:**INode getLeft()**

Gibt den linken Kindknoten zurück.

void setLeft(INode leftChild)

Setzt den übergebenen Knoten als linken Kindknoten.

INode getRight()

Gibt den rechten Kindknoten zurück.

void setRight(INode rightChild)

Setzt den übergebenen Knoten als rechten Kindknoten.

IComparable getData()

Gibt Inhalt des Knotens zurück.

void setData(IComparablenodeValue)

Setzt den Inhalt des Knotens auf den übergebenen Wert.

TreeNode-Klasse:**void setLeft(INode leftChild)**

Unterbindet das Setzen des linken Kindknoten. Das heißt, dass der linke Kindknoten des Betrachtungs-Knoten nach Ausführung der Methode unverändert ist.

void setRight(INode rightChild)

Unterbindet das Setzen des rechten Kindknoten. Das heißt, dass der rechte Kindknoten des Betrachtungs-Knoten nach Ausführung der Methode unverändert ist.

void setData(IComparablenodeValue)Aktualisiert den Wert des Knotens und setzt ihn auf den übergebenen Wert. Der Wert des Knotens darf nur gesetzt werden sofern dadurch der Baum nicht korrumptiert wird. Ein Setzen ist daher nur zulässig, wenn der übergebene Wert zwischen einer unteren und einer oberen Grenze liegt. Diese Grenzen können mittels den Methoden **getSmallestSubTreeDataElement** und **getHighestSubTreeDataElement** ermittelt werden. Es muss dabei jeweils das größte Element des linken Sub-Baums und das kleinste Element des rechten Sub-Baums verwendet werden. Ist einer der beiden Sub-Bäume nicht vorhanden, so wird der aktuelle Knotenwert als Grenzwert herangezogen. Fehlt der linke Sub-Baum so definiert der Knoten den unteren Grenzwert. Analoges gilt für den rechten Sub-Baum (oberer Grenzwert). Liegt der übergebene Wert innerhalb dieser Grenzen, darf der Wert des Knotens auf den neuen Wert gesetzt werden. In diesem Fall werden alle Observer notifiziert (**notifyObservers**) und TRUE als Resultat zurück geliefert. Konnte der Wert nicht aktualisiert werden, so wird FALSE zurück geliefert. Das zugehörige Sequenzdiagramm ist in Abbildung 8 dargestellt.**boolean isLeaf()**

Hat der Knoten keine Kindknoten, so ist er ein Blatt und das Resultat TRUE. Ist der Knoten kein Blatt, so wird FALSE zurückgegeben.

Collection<INode> findNodes(IComparablenodeValue)Vergleicht, mit der Methode **compareTo** in **IComparable**, den Knoteninhalt des Betrachtungs-Knotens mit dem übergebenen Wert. Stimmt dieser überein, wird der Knoten in die Resultatliste übernommen. Weiters wird auf beiden Kindknoten ebenfalls die Methode **findNodes** aufgerufen.

Name: _____

Mat.Nr.: _____

boolean removeNode(IComparable data) throws TreeException
Löscht einen Kindknoten, falls dieser mit dem übergebenen Wert übereinstimmt (Methode equals in IComparable). Es wird dabei der Knotenwert (Attribut data) mit dem übergebenen Wert verglichen und bei Übereinstimmung der Knoten gelöscht. Falls der zu löschen Knoten selbst Kindknoten hat und daher kein Blatt ist, wird eine TreeException mit dem Typ TreeTypeException NODE_DELETE geworfen.

Ist der Knoten kein Blatt und nicht der zu löschen Knoten, so wird auf den vorhandenen Kindknoten die removeNode-Methode aufgerufen.

Konnte ein Knoten erfolgreich gelöscht werden, so wird TRUE zurückgegeben. Konnte kein zu löschen Knoten identifiziert werden, wird FALSE zurückgegeben. Das zugehörige Sequenzdiagramm ist in Abbildung 7 dargestellt.

INode insertData(IComparable data) throws IncomparableTypesException
Fügt einen neuen Knoten in dem Baum ein. Der übergegne IComparable-Wert wird mit dem Inhalt (Attribut data) des Knotens verglichen (Methode compareTo in IComparable) und bei Übereinstimmung wird der Knoten als Resultat zurückgeliefert. In diesem Fall wird also kein neuer Knoten im Baum eingefügt.
Ist keine Übereinstimmung vorhanden, so werden die Kindknoten des Knotens überprüft. Ist der einzufügende Wert KLEINER als der Wert des Knotens, so wird der neue Knoten im LINKEN Sub-Baum eingefügt. Ist der einzufügende Wert GRÖSSER als der Wert des Knotens, so wird der neue Knoten im RECHTEN Sub-Baum eingefügt.

boolean updateNodeData(IComparable data)
Das Aktualisieren des Knotenwertes wird unterbunden und nach Ausführung der Methode ist der Knotenwert unverändert.

void addDataToSortedList(Collection<IComparable>)
Erweitert die übergebene Liste um den Inhalt des Knotens und seiner Kinder. Die Inhalte werden in absteigender Reihenfolge eingefügt. Das zugehörige Sequenzdiagramm ist in Abbildung 6 zu sehen.

void notifyObservers()
Notifiziert mit der Methode notifyAboutUpdate in IObserver alle registrierten Observer.

ISubject-Interface:

boolean registerObserver(IObserver observer)
Registriert den übergebenen IObserver als Observer am Subject. True, falls der IObserver registriert werden konnte, ansonsten false.

boolean unregisterObserver(IObserver observer)
Löscht den übergebenen IObserver als Observer am Subject. True, falls der IObserver gelöscht werden konnte, ansonsten false.

IObserver-Interface:

void notifyAboutUpdate(ISubject subject)
Wird vom ISubject aufgerufen um zu signalisieren, dass Änderungen am Subject durchgeführt wurden. Um das ISubject identifizieren zu können, wird dieses als Parameter mit übergeben.

IComparable-Interface:

int compareTo(Object obj)
Vergleicht zwei Objekte hinsichtlich ihrer Reihenfolge miteinander und liefert ein entsprechendes Resultat. Sofern das übergebene Objekt GRÖSSER als das zugrundeliegende Objekt ist, wird -1 zurückgegeben. 0 bei GLEICHHEIT beider Objekte und 1 falls das übergebene Objekt KLEINER als das zugrundeliegende Objekt ist.

boolean equals(Object obj)
Vergleicht zwei Objekte miteinander. Sind die zwei Objekte gleich, wird TRUE zurückgegeben, ansonsten FALSE.

IComparable createCopy()
Erstellt eine Kopie des Objektes und gibt diese Kopie zurück.

TreeException-Klasse:

Leitet von der Klasse Exception ab und definiert eine Exception für den Baum. Es wird dabei eine TreeExceptionType übergeben, welcher den Ursachen-Typ der Exception festlegt.

TreeException(TreeExceptionType type)
Initialisiert die Exception anhand des übergebenen TreeExceptionTypes. Dabei wird der TreeExceptionType verwendet um eine aussagekräftige Exception-Message zu erstellen.

Implementieren Sie hier die geforderten Klassen und Methoden:

// Tree - Klasse

```
public Collection< Comparable> flattenTo SortedList() {
    Collection< Comparable> col = new Collection<>();
    NPE root.addDataTo SortedList(col);
    return col;
}
```

3,5P

// TreeNode Klasse extends Node

```
public class TreeNode implements Node { Subclass }
    Observer private vector<Observer> Collection observer;
    public void addDataTo SortedList(Collection< Comparable> col) {
        Collection col = this.addDataTo SortedList(col);
        if (getRight() != null)
            col = getRight(). addDataTo SortedList(col); Void - Return
        if (getLeft() != null)
            col = getLeft(). addDataTo SortedList(col); Void
        col.add(getData());
        Comparable buff = getData();
        col.add(buff buff.createCopy());
        sorted! (Sequenzdiagramm)
    }
}
```

2,5P

col
3

}

Implementieren Sie hier die geforderten Klassen und Methoden:

// Tree Node Klasse

~~NodeData~~

```

public boolean setData (Comparable comp) { 7P
    if (getSmallestSubTreeDataElement() < comp && comp < getHighestSubTreeDataElement())
        if (Node left = getLeft(); left == null) { left = this; } else
            Comparable highest = left.getHighestSubTreeDataElement();
        Comparable right = getRight(); if (right == null) { right = this; } else
            Comparable small = right.getSmallestSubTreeDataElement();
        if (comp.compareTo(highest) == -1 && comp.compareTo(small) == 1) {
            super
            this.setData(comp.createCopy());
            this.notifyObservers();
            return true;
        }
    return false;
}

```

comp = nodeValue

Variable aus Block, kein Zugriff. Else nicht korrekt

}

private void notifyObservers () { 3P

```

for (IObserver ob : observers)
    ob.notifyAboutUpdate(this);
}

```

}

Implementieren Sie hier die geforderten Klassen und Methoden:

// Tree Node Klasse

10

```
public boolean removeNode(IComparable data) throws TreeException;
```

{

~~Comparable high = this.get~~
~~if (this.getLeft() != null & this.getRight() != null) {~~
~~new TreeException(TreeExceptionType.NODE_DELETE);~~

{
elseif (~~this == null~~)

NPE

else Comparable left = getLeft().getNodeData();

if ((left.equals(data)) == true) {

! left.isLeaf()

if ((left.getLeft() != null || left.getRight() != null) {

new TreeException(TreeExceptionType.NODE_DELETE),

{ else }

super

left = null; setLeft(null);
return true;

}

{
else

Comparable right = getRight().getNodeData();

if (right.equals(data) == true) {

if (! right.isLeaf())

new TreeException(TreeExceptionType.NODE_DELETE);

else { super
setRight(null); }
return true;

}

Name: _____

Mat.Nr.: _____

**Implementieren Sie hier die geforderten Klassen und Methoden:
Klassendiagramm:**

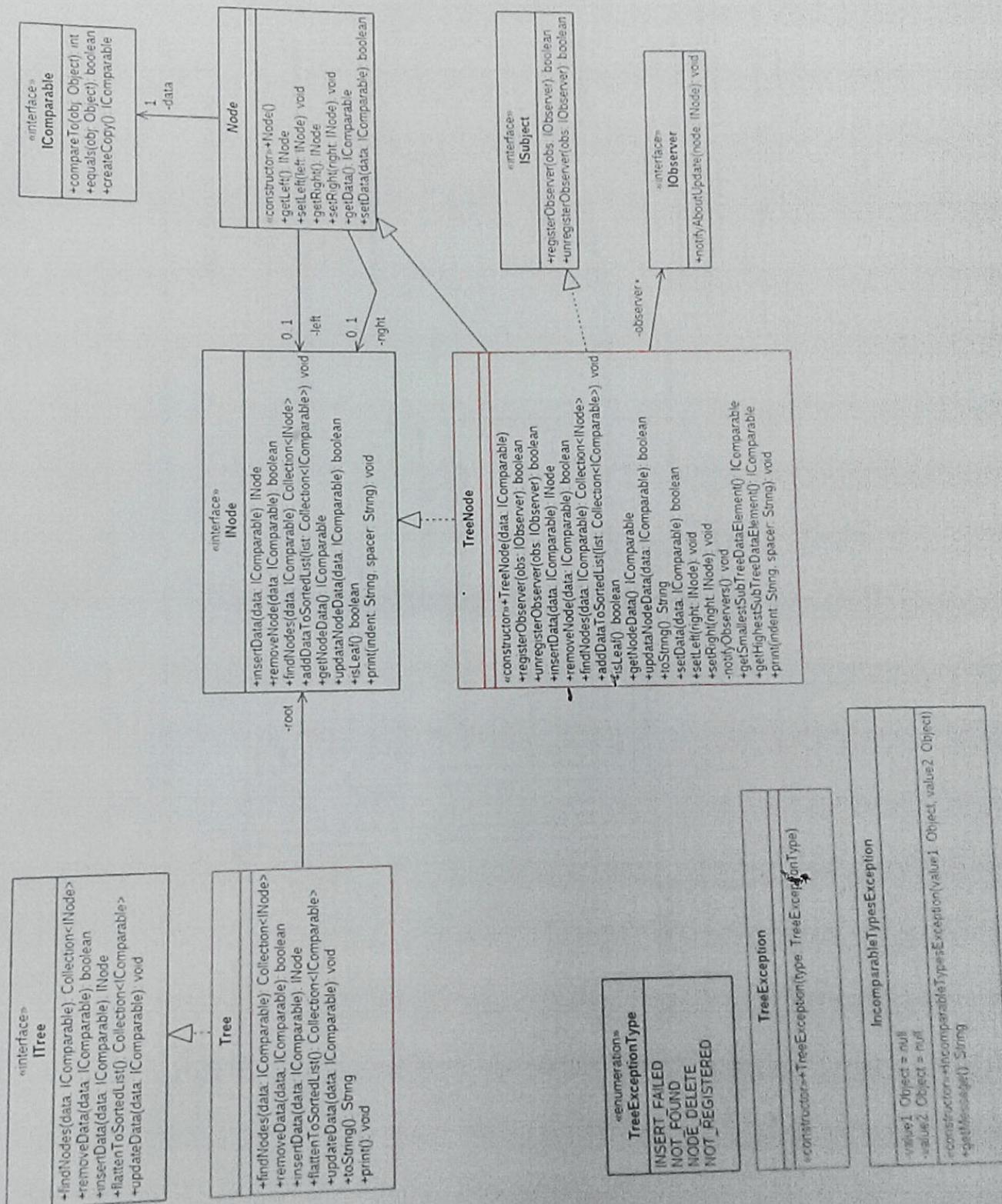
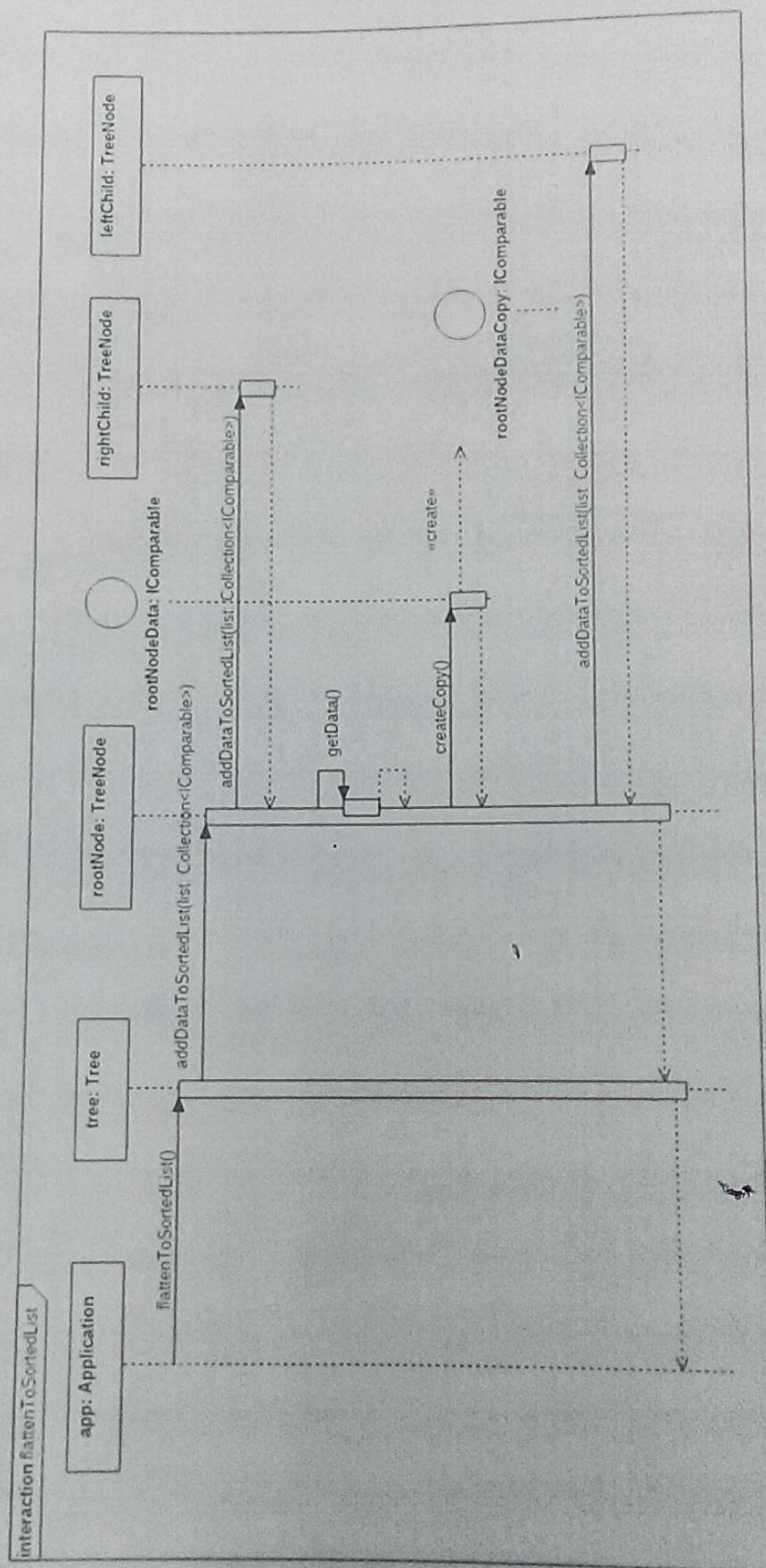
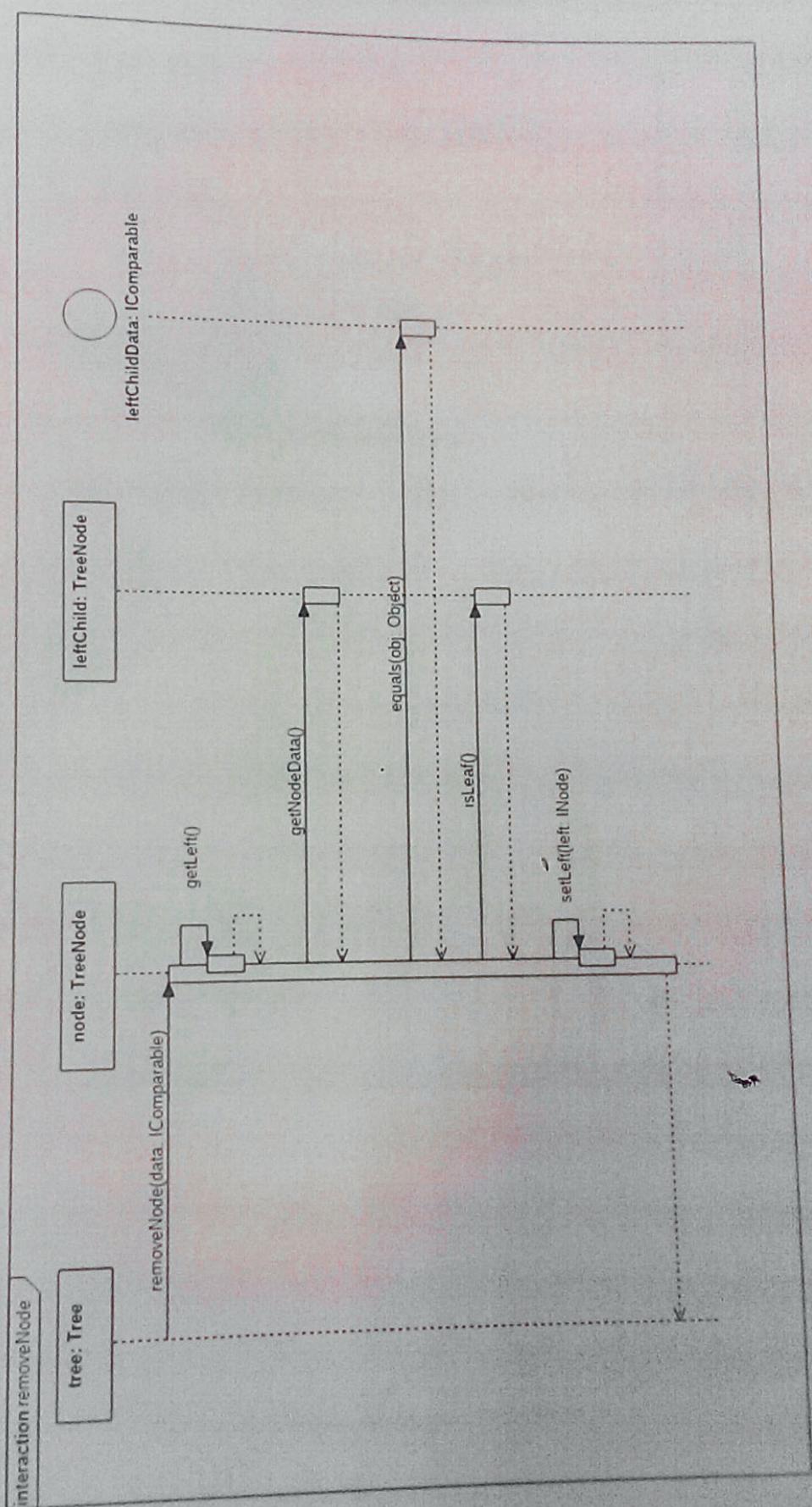
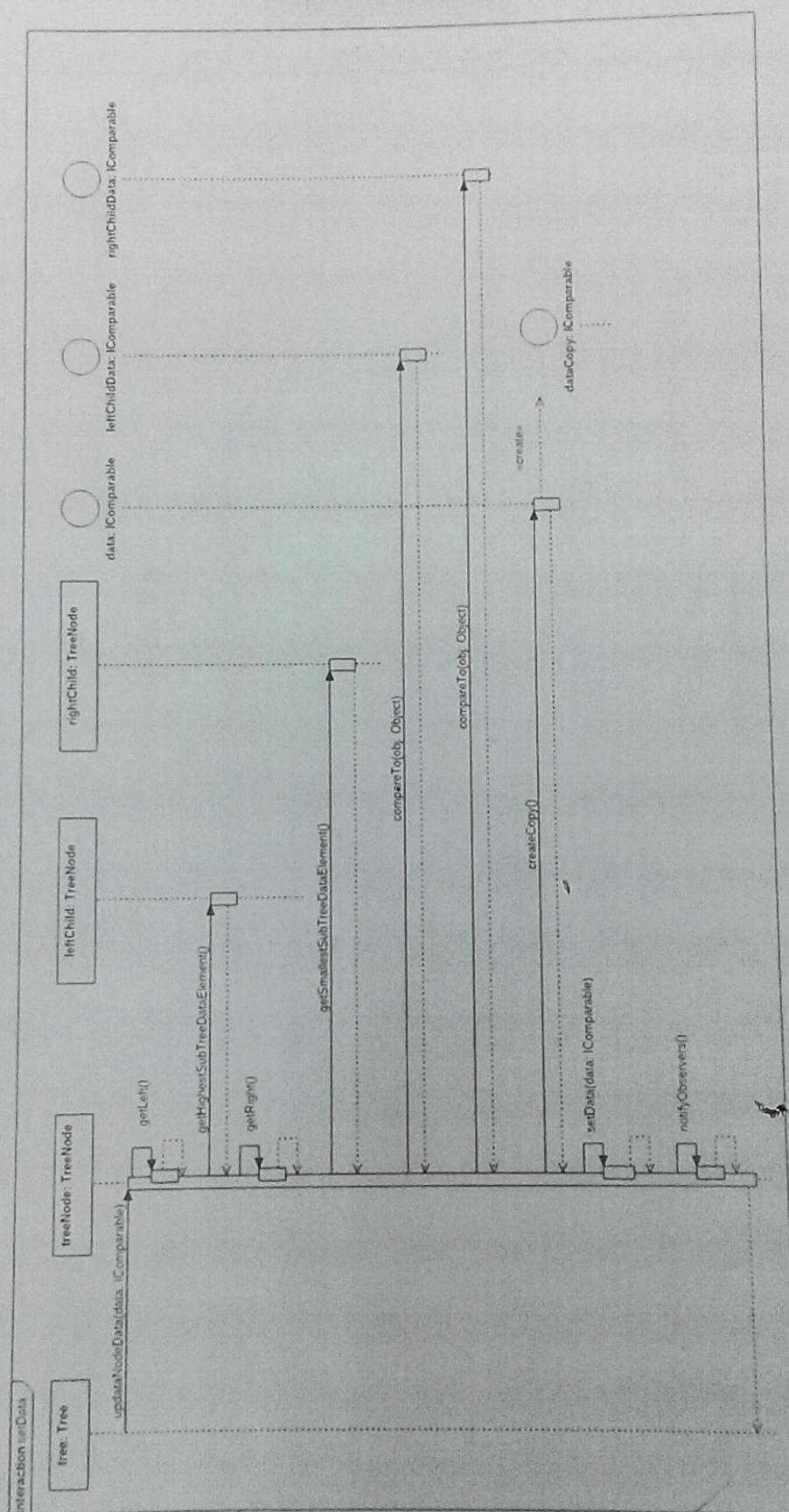


Abbildung 5 - Binärbaum Klassendiagramm

Sequenzdiagramme:Abbildung 6 - *flattenToSortedList* Sequenzdiagramm

Sequenzdiagramme:Abbildung 7 - `removeNode` Sequenzdiagramm

Sequenzdiagramme:Abbildung 8 - `setData` Sequenzdiagramm

Implementieren Sie hier die geforderten Klassen:

```
if( left != null ) {  
    if( left.removeNode(data) )  
        return true;  
    else  
        if( left.removeNode(data) == true )  
            return true;  
        else  
            if( right.removeNode(data) )  
                return true;  
            else  
                return false;  
}
```

Implementieren Sie hier die geforderten Klassen:

public class TreeException extends Exception { } 3,5 P

public TreeException (TreeException type) { }

 System.out.println(type);

~~final~~ switch (type) { }

 case NODE_DELETE:

 System.out.println("Bla ... Knoten nicht gefunden");

 break; *sagen*

 Case ... -

 default:

}

}

}

Beispiel 2**a) (5 Punkte)***4/5*

Analysieren Sie die folgenden Java-Klassen `Printer` und `NewPrinter`. Die `main`-Methode der Klasse `Test` erzeugt Objekte und sendet diesen Objekten Nachrichten, die gewisse Ausgaben bewirken. Geben Sie die jeweiligen Ausgaben zu den gefragten Zeitpunkten (T1 bis T5, siehe Kommentare in `main`) an, genauso, wie es für den Zeitpunkt T0 gezeigt ist.

```
public class Printer {
    public void show( String val ) {
        System.out.println( "PRT " + val );
    }

    public void show( boolean val ) {
        if ( val )
            System.out.println( "true" );
        else
            System.out.println( "false" );
    }
}
```

```
public class NewPrinter extends Printer {
    public void show( String val ) {
        System.out.println( "NEWPRT " + val );
    }

    public void show( boolean val ) {
        if ( ! val )
            System.out.println( "falsch" );
        else
            System.out.println( "wahr" );
    }
}
```

Ausgaben:

T0: START
 T1: true ✓
 T2: PRT TRUE ✓
 T3: NEWPRT FALSE ✓
 T4: falsch ✓
 T5: PRT FALSE X

```
public class Test {
    public static void main( String[] args ) {
        Printer p = new Printer();
        NewPrinter newP = new NewPrinter();

        System.out.println( "START" ); // T0
        p.show( true ); // T1
        p.show( "TRUE" ); // T2
        newP.show( "FALSE" ); // T3
        p = newP;
        p.show( false ); // T4
        ((Printer) p).show( "FALSE" ); // T5
    }
}
```

b) (10 Punkte)*8/10*

Gegeben sind die vier Java-Klassen `Worker`, `QueueWorker`, `TestWorker` und `UI` sowie das Interface `Action`.

```
public class Worker { ... }

public class QueueWorker extends Worker
    implements Action { ... }

public class TestWorker extends QueueWorker {
    ...
}

public interface Action {
    public doAction(); ...
}

public class UI {
    private Action action;
    public void setAction (Action action); ...
}
```

```
public class Testklasse {
    public static void main( String[] args ) {
        ...
        Worker w = new QueueWorker(); // C1
        Action a = new Worker(); // C2
        UI ui = new UI();
        ui.setAction( new TestWorker() ); // C3
        Action b = new QueueWorker(); // C4
        w = (Worker) ( TestWorker ) b;
        Action c = ui.action; // C5
    }
}
```

Sind die Aktionen/Operationen C1 bis C5 in der Klasse `Testklasse` erlaubt?
 Kreuzen Sie an und begründen Sie Ihre Antworten (Antworten ohne Begründung werden nicht gewertet).

Codezeile:	Erlaubt?		Begründung:
	Ja	Nein	
C1:	X	<input type="checkbox"/>	QueueWorker ist von Worker abgeleitet
C2:	<input type="checkbox"/>	X	Action ist nicht von Worker implementiert
C3:	X	<input type="checkbox"/>	Action ist von QueueWorker implementiert und darf von QueueWorker überladen werden
C4:	<input type="checkbox"/>	X	QW kann nicht zu TW umgewandelt werden
C5:	<input type="checkbox"/>	X	action ist nicht initialisiert

Beispiel 3 (10 Punkte)

8/10

Kreuzen Sie bei den folgenden zehn Fragen RICHTIG an, wenn die Aussage richtig ist, oder FALSCH, wenn sie nicht richtig ist.

Bewertungsschema:

Für jede korrekt angekreuzte Aussage wird +1 Punkt gezählt. Wenn eine Frage **nicht korrekt** angekreuzt ist, wird 1 Punkt abgezogen (also -1). Wenn bei einer Frage weder RICHTIG noch FALSCH angekreuzt sind, gibt es 0 Punkte für diese Frage. Sie können in Summe bei Beispiel 3 **nicht weniger** als 0 Punkte erzielen, selbst wenn sich rein rechnerisch eine negative Punkteanzahl ergeben würde.

RICHTIG FALSCH

- | | | |
|--|------------------------------------|------------------------------------|
| 1) Das Singleton Pattern gewährleistet, dass alle Methoden der Singleton-Klasse global verfügbar sind. | <input type="radio"/> O | <input checked="" type="radio"/> X |
| 2) Instanzvariablen definieren das Verhalten und Instanzmethoden den Zustand eines Objekts. | <input type="radio"/> O | <input checked="" type="radio"/> X |
| 3) Das Verhalten einer Subklasse kann auch vom Verhalten der Superklasse abweichen. | <input checked="" type="radio"/> O | <input type="radio"/> O |
| 4) Eine abstrakte Klasse kann auch mehr als ein Interface implementieren. | <input type="radio"/> O | <input checked="" type="radio"/> X |
| 5) Testen kann zur Verifikation des spezifizierten Verhaltens verwendet werden. | <input checked="" type="radio"/> O | <input type="radio"/> O |
| 6) Ein Objekt im Sinne von OOP nimmt keinen Speicherplatz ein, da es nur ein abstraktes Konstrukt ist. | <input type="radio"/> O | <input checked="" type="radio"/> X |
| 7) Abstrakte Klassen sind nicht Teil der Klassenhierarchie. | <input type="radio"/> O | <input checked="" type="radio"/> X |
| 8) Eine abstrakte Klasse kann nur genau eine Instanz zur Laufzeit haben. | <input type="radio"/> O | <input checked="" type="radio"/> X |
| 9) Jede Instanz eines Subtyps kann dort verwendet werden, wo eine Instanz eines Supertyps erwartet wird. | <input checked="" type="radio"/> O | <input type="radio"/> O |
| 10) Es gibt immer nur einen Konstruktor in einer Klasse. | <input type="radio"/> O | <input checked="" type="radio"/> X |

Beispiel 4 (20 Punkte) 11,5/5

Beantworten Sie die vier nachfolgenden Fragen kurz aber prägnant:

2,5/5

a) Was ist Polymorphismus? Demonstrieren Sie dieses Konzept anhand eines eigenen Beispiels.

bedeutet mehrfachigkeit, dadurch versteht man wenn 2 "Dinge" gleichen Namens unterschiedliches ausführen und zu Laufzeit das passende automatisch gewählt wird.

Bsp: ~~Klasse A~~ ~~Methoden~~

Dies kann bei Klassenableitungen, Overloading, overwriting, ... auftreten

public class Test {

 public ~~int~~ void ^{Funktion} ~~int~~ a, int b) { ... }

 public void Funktion (int a) { ... }

\Rightarrow Overloading

Op

}

} es wird anhand der Anzahl der Parameter unterschieden welche Funktion/Methode ausgetragen wird
keine Polymorphismus
Details in Overloading.

3,5/5

b) Was versteht man unter „Patterns“? Erklären Sie zwei Patterns genauer.

Beim Programmieren entstehen oft wiederkehrende Probleme. Um diese zu beheben ~~man~~ hat man sich auf Design-Patterns geeinigt. Daraus versteht man die immer gleiche Vorgangsweise von ~~soft~~ Software Herausforderungen.

Singelton Pattern

Es ~~kann~~ kann das jeweilige Objekt nur 1x erzeugt werden. Der Konstruktor wird private gesetzt und eine eigene Methode detoniert erstellt, die überprüft ob das Objekt schon existiert.

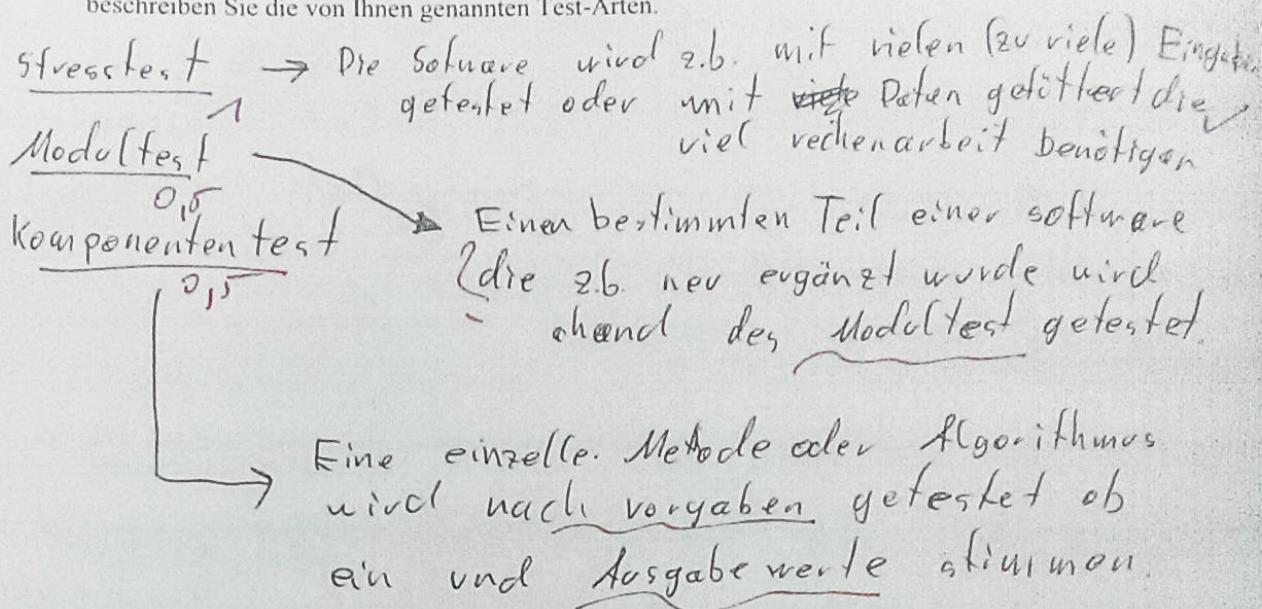
Factory Pattern

Es gibt ein eigenes Objekt diese andere Objekte erzeugt. Dadurch kann diese neuen Objekten eine ID zugewiesen werden.

Beispiel 4

2/5

- c) Welche Arten des Testens kennen Sie außer Black-Box und White-Box-Testen? Erklären und beschreiben Sie die von Ihnen genannten Test-Arten.



3,5/5

- d) Erklären Sie das Konzept der Datenkapselung (Data encapsulation) allgemein und anhand eines Beispiels. Wie könnten Sie das Konzept der Datenkapselung verletzen? Wie steht dieses Konzept mit Klassenhierarchie und Vererbung in Zusammenhang?

Datenkapselung

-) ~~nicht~~ kein direkter Zugriff auf Objekte/ Variablen
-) AP einer Klasse (sind private)
-) Es kann die Variable/Objekt nur durch eine vorhandene ~~public~~ Methode geändert werden.

Verletzen Schnittstellen?

Bsp:

```

public class test {
    private int a;
    public void setint-a (int buff) {
        this.a = buff;
    }
    public int getint-a() {
        return this.a;
    }
}
  
```

Vererbung:

protocol D
AP ✓/o