Aufgabenstellung für das *zweite* Übungsbeispiel der LVA "Objektorientiertes Programmieren"

Abgabeschluss für Source Code: Montag 16.05.2016!

WICHTIG: Bei Studierenden, deren Beispiel nicht spätestens am 16.05.2016 um 23:55 Uhr am TUWEL zur Verfügung steht, kann das Beispiel nicht abgenommen werden. Diese Studierenden haben in Folge nicht mehr die Möglichkeit, die Übung der LVA im laufenden Semester erfolgreich abzuschließen. Damit haben sie auch keine Möglichkeit, zur LVA-Prüfung anzutreten und diese im SS 2016 abzuschließen!

Abgabegespräch:

Am ***17.05.2016*** *oder* ***18.05.2016*** (Jeweils zwischen 09:00 und 18:00 Uhr: Den Tag und die genaue Uhrzeit wählen Sie selbst beim Upload Ihres Beispiels in TUWEL.)

Allgemeines

Jeder Teilnehmer muss die Beispiele **eigenständig** ausarbeiten. Bei der Abgabe jedes Beispiels gibt es ein kurzes Abgabegespräch mit einem Tutor oder Assistenten. Diese Abgabegespräche bestehen aus folgenden Punkten:

- Im Zuge der Abgaben müssen Sie eine Änderung am Programm vornehmen können.
- Beantworten von Fragen zu Ihrem Programm.
- Ihr Programm wird mit automatisierten Tests geprüft.
- Beantwortung theoretischer Fragen zu den jeweils in der Übung behandelten Konzepten.
- Stellen Sie sicher, dass die formalen Anforderungen (siehe **Abgabe**) erfüllt sind.
- Halten Sie sich exakt an die Beschreibungen der Klassen in den **Klassendiagrammen** und der **JavaDoc**.

Abgabe

- Orientieren Sie sich an den Java Code Conventions
  http://www.oracle.com/technetwork/java/codeconvtoc-136057.html
- Geben Sie Ihrem Projekt einen eindeutigen Namen, bestehend aus Ihrer Matrikelnummer und Ihrem Nachnamen**.**
  Eclipse-Projekt: *matrNr_Nachname*
- Name und Matrikelnummer muss zu Beginn jeder Klasse angeführt werden!
- Exportieren Sie das Projekt (inklusive aller für Eclipse notwendigen Dateien, z.B.: .classpath, .project) als ZIP-Datei. Eine Anleitung dazu finden Sie in TUWEL.

Kontakt:

Inhaltliche Fragen:	Diskussionsforum in TUWEL
Organisatorische Fragen:	oop@ict.tuwien.ac.at

Aufgabe: Datenverwaltung mittels einer Baumstruktur

Ziel:

Verstehen folgender Konzepte: Klassendiagramme, Packages, Abstrakte Datentypen, Interfaces, Generics, Casting, Polymorphismus, Overloading und Overriding von Methoden, JavaDocs, Exceptions und Testen; insbesondere das Verstehen des Unterschiedes zwischen Konkreten und Abstrakten Datentypen sowie die Fähigkeit, mit Hilfe beider wiederum Abstrakte Datentypen zu definieren.

Einleitung:

Das Beispiel besteht aus einer Minimalanforderung, mit der Sie eine „~“ bei der Abgabe erreichen können. Für die Extrapunkte durch ein „+“ müssen sie auch die Bonusaufgabe lösen. Beachten Sie aber wenn Sie die Bonusaufgabe abgeben, werden Sie auch in Theorie und Praxis intensiver dazu befragt!

Ihre Aufgabe besteht darin, einen ***Abstrakten Datentyp*** zu programmieren, der erlaubt Daten zu verwalten ohne die konkreten Datenstrukturen darin zu kennen. Dafür soll intern eine Baumstruktur erstellt werden um Daten geordnet zu speichern. Eine Übersicht der zu implementierenden Packages und Klassen finden Sie in Abbildung 1. Eine genaue Beschreibung der einzelnen Methoden dieser Klassen finden Sie in der beigelegten **JavaDoc**. Halten Sie sich **exakt** an die **Spezifikationen** der **Klassendiagramme** und der **JavaDoc**, da Ihr Programm im Rahmen der Abgabe automatisiert getestet werden wird.

Minimalanforderung:
Zur Erfüllung der Minimalanforderungen müssen folgende Aufgabenstellungen von Ihnen umgesetzt werden:

1. **Implementierung einer Klassenhierarchie für Produkte.**
   Dieser Teil ist in Abbildung 3 grün umrandet und betrifft das Package `domain.product`. In diesem Teil werden *Klassendiagramme* und *Vererbung* vertieft.
2. **Implementierung einer Baumstruktur zur geordneten Darstellung von Daten.**
   Die Implementierung erfolgt im Package `treeview`, welches in Abbildung 4 gelb umrandet dargestellt ist. In diesem Teil werden die Konzepte *Polymorphismus*, *Exceptions* und *Generics* vertieft. Verwenden Sie in Ihren Implementierungen (soweit erforderlich) Exception Handling.
3. **Implementierung einer Start-Klasse**
   Erstellen Sie eine Klasse *Application*, mit der Ihr Programm gestartet werden kann. Legen Sie in dieser Klasse einen einfachen Baum mit mindestens zwei Gruppen und zugehörigen Unterelementen an und geben Sie diesen auf der Konsole aus.
4. **Testen mittels JUnit-Test**
   Implementieren Sie zusätzlich noch einige Testfälle in Form von **JUnit-Tests**. Verwenden Sie dafür das Package `test`. Schreiben Sie Ihre Testfälle so, dass Sie die Funktionalität Ihres Programmes ausreichend überprüfen. (Sie müssen jedoch nicht jede Methode einzeln testen.) **Verwenden Sie für ihre Tests JUnit 4**, siehe http://www.vogella.com/tutorials/JUnit/article.html.

Bonusaufgabe:
In einem optionalen Zusatzteil können Sie zwischen einer von zwei Aufgaben wählen. Dieser Zusatzteil ist nicht verpflichtend, wird aber benötigt, um mögliche Zusatzpunkte auf den praktischen Teil des Beispiels zu erhalten:

5. **BONUS 1: Implementierung einer einfach verketteten Liste (Container).**
   Die Implementierung dieser Aufgabe erfolgt im Package `container`, welches in Abbildung 7 dargestellt ist. Grundsätzlich wird hier anhand des *Collection*-Interfaces eine eigene Implementierung für eine generische Liste implementiert. Die hier angewendeten Konzepte beinhalten unter anderem *Interfaces*, *Casting*, *Generics*, *Exceptions*, *Overriding*…. **Sobald Sie das `container`-Package implementiert haben, müssen Sie dieses auch in Ihren Implementierungen der Minimalanforderung verwenden.**

6. **BONUS 2: Serialisierung von Objekten**
   Implementieren Sie eine Möglichkeit Ihre Baumstruktur **Serializable** zu machen, um diese in weiterer Folge zu speichern. Erweitern Sie Ihr Programm so, dass Ihr Baum beim Beenden des Programmes gespeichert und beim Starten wieder geladen wird.

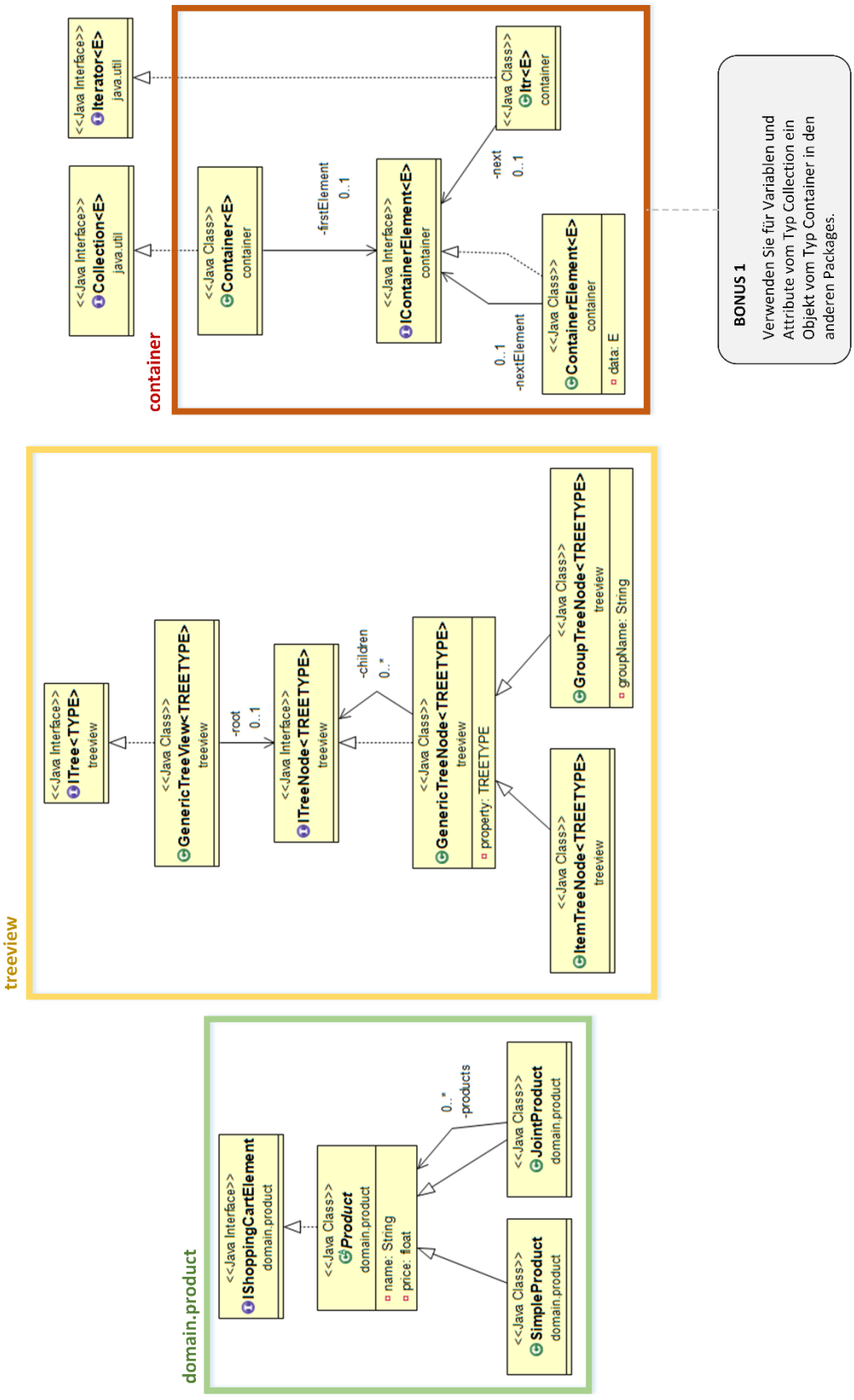

Abbildung 1 - Package Overview

Allgemeine Hinweise

Prinzipiell können alle Teile der Abgabe unabhängig voneinander implementiert und getestet werden. Allerdings verwenden einzelne Teilaufgaben Listen, um Elemente zu organisieren. Als Instanzen können Sie dabei im ersten Schritt auch eine Standardimplementierung aus dem JDK (beispielsweise ein `Vector`, siehe https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html) verwenden. Verwenden Sie dabei aber nur Methoden, die durch das Collection-Interface festgelegt sind (siehe Abbildung 2).

Das Collection Interface ist ein abstrakter Datentyp. Er legt öffentliche Methoden und mit diesen das Verhalten dieser Klasse fest. Ein konkreter Datentyp implementiert diesen abstrakten Datentyp. Beispielsweise implementieren Vector und ArrayList das Interface Collection. Eine Variable vom Typ Collection kann daher sowohl ein Objekt vom Typ Vector als auch ein Objekt vom Typ ArrayList referenzieren.

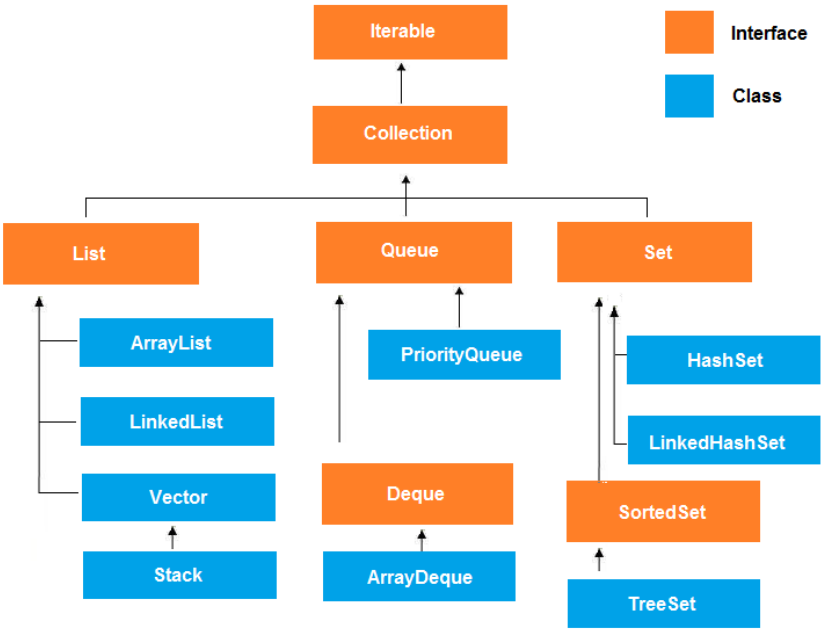

Abbildung 2 - Collection Hierarchie

Sofern Sie **Bonusaufgabe 1 – Implementierung einer einfach verketteten Liste** umgesetzt haben, müssen Sie diese Instanziierungen durch Ihre `Container` -Klasse ersetzen.

Minimalanforderung

Teilaufgabe 1: Package domain.product – Klassenhierarchie für Produkte

Implementieren Sie die Klassenhierarchie aus Abbildung 3 für das Package `domain.product`. Diese beschreibt eine Klassenhierarchie für Produkte, bestehenden aus den drei Klassen: *`Product`*, `SimpleProduct` und `JointProduct`.

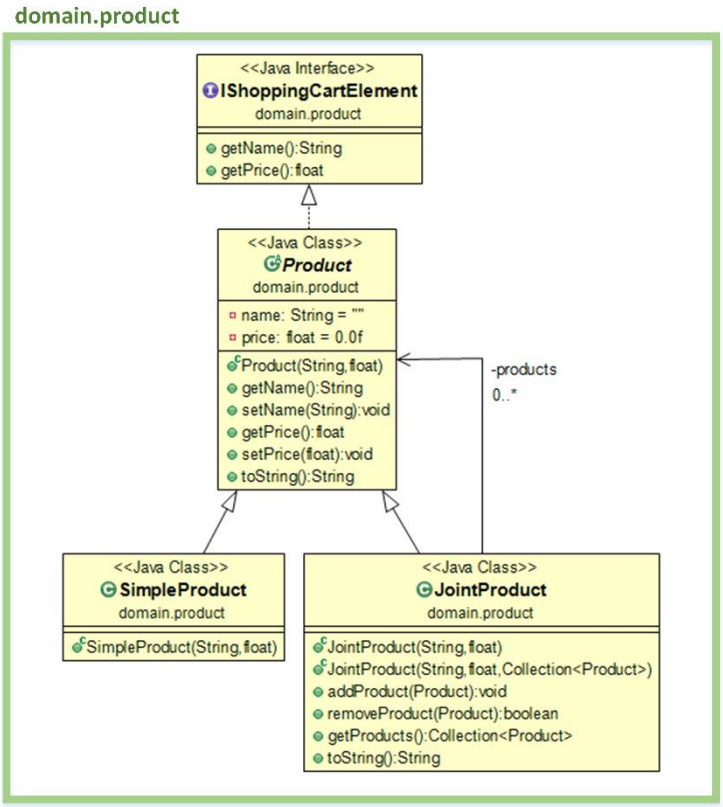

Abbildung 3 - Produkt Klassenhierarchie

`IShoppingCartElement` ist ein abstrakter Datentyp, der Zugriffsmethoden definiert, um den Namen und den Preis eines Warenkorbelements ermitteln zu können.

Die Klasse `Product` enthält zwei Attribute (name und price). Davon sind zwei Klassen abgeleitet, eine für einfache Produkte (z.B. Bier, 3,50€) und Produktsammlung (z.B. Mittagsmenü mit Gedeckpauschale und mehreren Gängen).

Überschreiben Sie die **toString()**-Methode in diesen Klassen und geben Sie Detailinformationen der Klassen übersichtlich zusammengefasst zurück.

BONUS: Sobald Sie das `container`–Package implementiert haben, verwenden Sie für die Sammlung von Produkten Ihre Implementierung der `Container` Klasse.

Teilaufgabe 2: Package treeview – Baumstruktur

Der zweite Teil der Abgabe besteht darin eine Baumstruktur für Elemente zu programmieren. Die Implementierung dazu erfolgt in dem Package `treeview`. Ein detailliertes Klassendiagramm dieses Packages ist in Abbildung 4 zu sehen. Sofern Sie auch die Bonusaufgabe Container gelöst haben, müssen Sie in diesem Teil diese Implementierung verwenden.

Die beiden Interfaces `ITree` und `ITreeNode` definieren zwei abstrakte Datentypen, mit denen bereits die Struktur des Baums festgelegt ist. Das Interface `ITree` legt die Methoden fest, um den Wurzelknoten zu setzen und abzufragen. Als weitere Funktion ist das Suchen nach einem Knoten definiert. Knoten im Baum sind über das Interface `ITreeNode` festgelegt.
Die Implementierung erfolgt zusätzlich mit konkreten Datentypen. Beispielsweise könnte ein konkreter Baum vom abstrakten Datentyp `ITree` ein Baum mit beliebig vielen Kindknoten oder ein Binärer-Baum mit einem linken und rechten Kindknoten sein. Beides ist mit dem abstrakten Datentyp abbildbar.

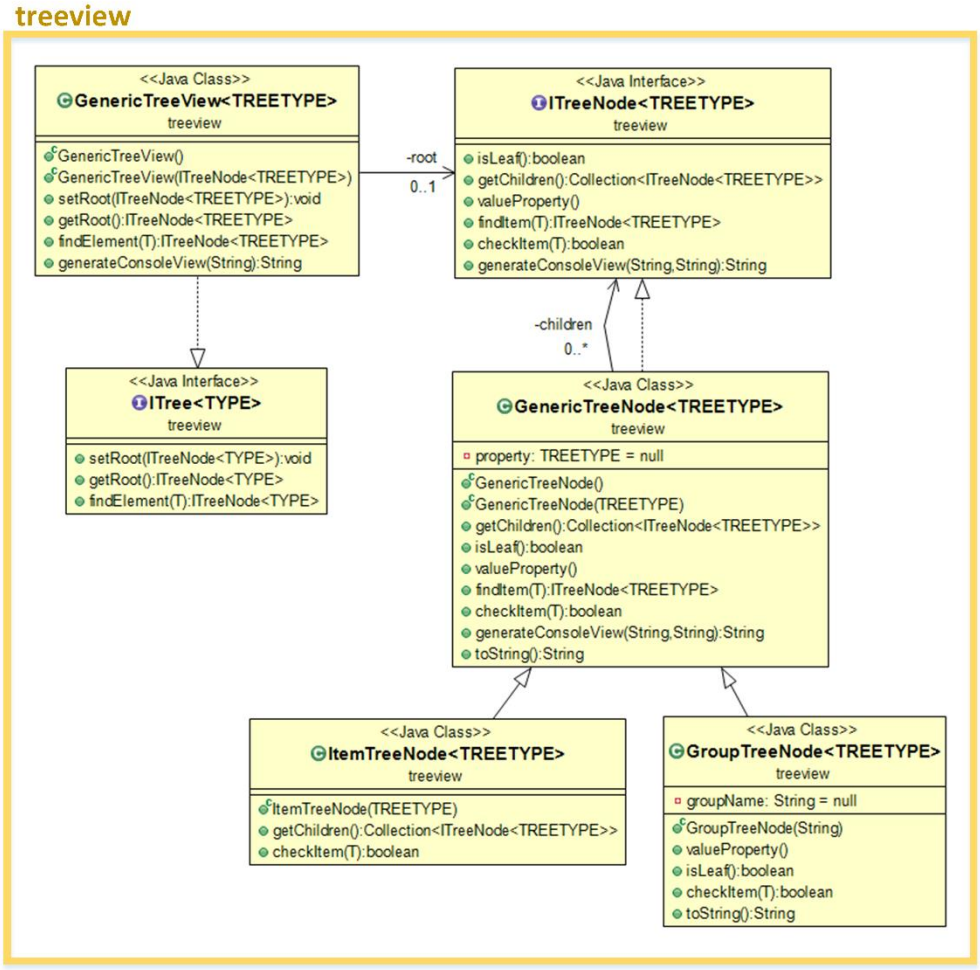

Abbildung 4 - Klassendiagram Baumstruktur

Allgemein kann ein Baum, ausgehend von einem Wurzel-Knoten (`rootNode`), Knoten hierarchisch in mehreren Ebenen anordnen. Es ist dabei möglich, dass ein Knoten Referenzen auf weitere Unter-Knoten halten kann. https://de.wikipedia.org/wiki/Baum_(Graphentheorie)

Knoten mit untergeordneten Knoten sind Gruppen (`Groups`). Knoten, die keine Unter-Knoten enthalten, sind Blatt-Knoten (`Leafs`). Abbildung 5 zeigt ein Beispiel für einen Baum mit dem Wurzel-Knoten *Root Gruppe*.

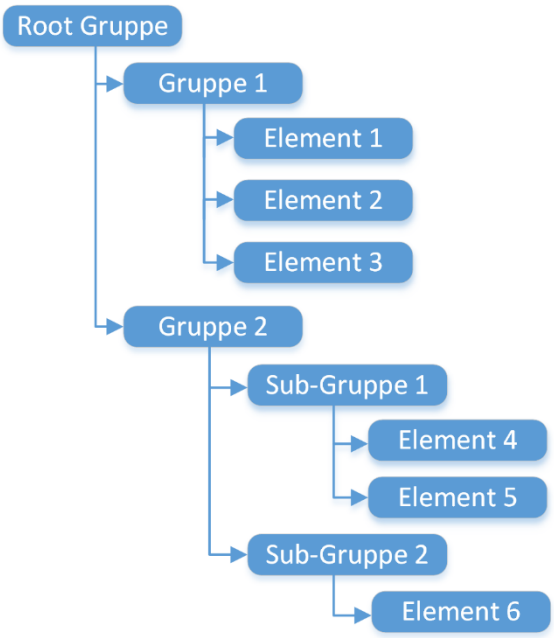

Abbildung 5 - Einfacher Baum

Jeder `TreeView` hat einen Wurzel-Knoten (den `rootNode`), welcher die oberste Ebene des Baumes darstellt. Der Einfachheit halber müssen Sie unterschiedliche Elemente auf einer Ebene nicht berücksichtigen. Das heißt, dass jede Ebene des Baumes **entweder** aus **Gruppen oder** aus **Blättern** besteht, diese aber nie gemischt werden. Entsprechend ist der in Abbildung 6 gezeigte Baum nicht zulässig, da auf der Ebene unterhalb der *Root Gruppe* sowohl Elemente (also Blätter, hier *Element 4*) als auch Gruppen (hier *Gruppe 1*) vorhanden sind.

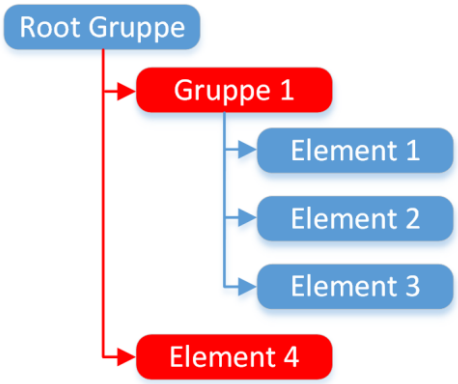

Abbildung 6 - Unzulässiger Baum

Die Baumstruktur wird in der Klasse `GenericTreeView` implementiert und besteht aus mehreren Knoten, die miteinander verknüpft sind. Der Wurzelknoten (das *RootElement*) des Baumes wird dabei als Attribut definiert:

- `root` ist eine Referenz auf den Wurzelnoten des Baumes.

Die Knoten des Baumes sind über das Interface `ITreeNode` festgelegt. Prinzipiell kann jedes dieser `ITreeNodes` mehrere oder gar keine Kinder (Unter-Knoten) haben. Sind keine Kind-Knoten vorhanden, so ist es ein Blatt des Baums. Methoden dieses Interfaces sind unter anderem:

- `isLeaf` überprüft, ob der aktuelle Knoten weitere Unter-Knoten (Kinder) hat
- `getChildren` liefert eine Liste der Kinder des aktuellen Knotens zurück
- `valueProperty` liefert den Inhalt des Knoten zurück

Die Implementierung eines solchen `ITreeNode` müssen Sie in der Klasse `GenericTreeNode` vornehmen. Dieser `GenericTreeNode` hat folgende Attribute:

- `children`, ist eine Liste von Kind-Knoten des Knotens. **BONUS: Sobald Sie das `container`–Package implementiert haben, verwenden Sie für diese Liste Ihre Implementierung der `Container` Klasse.**
- `valueProperty` hält den eigentlichen Informationsinhalt des Knotens

Da wir in unserem Baum zwischen Gruppen und Blatt-Knoten unterscheiden wollen, sind zwei Sub-Klassen des `GenericTreeNode` umzusetzen.

`ItemTreeNode` ist die Implementierung eines Blatt-Knoten. Überschreiben Sie die Methoden `isLeaf` und `getChildren` und geben sie entsprechende Rückgabewerte zurück.

`GroupTreeNode` hat ein zusätzliches Attribut `groupName` der den Gruppennamen enthält. Eine Instanz dieses Typs kann keine Information speichern (Attribut: `valuePropery`).

Teilaufgabe 3: Implementierung einer Start-Klasse

Erstellen Sie eine Klasse mit der Ihr Programm gestartet werden kann. Legen Sie in dieser Klasse einen einfachen Baum mit mindestens zwei Gruppen und zugehörigen Unterelementen an und geben Sie diesen mittels der `generateConsoleView`-Methode des Baumes auf der Konsole aus. Verwenden Sie für die Knoten dieser Gruppen jeweils andere Datentypen.

Name der Start-Klasse: Application.java

Teilaufgabe 4: Testen mittels JUnit-Tests

Implementieren Sie einige Testfälle in Form von **JUnit-Tests** und verwenden Sie diese um Ihr Programm zu überprüfen. Die Implementierung erfolgt im Package `test`. Schreiben Sie Ihre Testfälle so, dass Sie die Funktionalität Ihres Programmes ausreichend überprüfen (Sie müssen nicht jede Methode einzeln testen).

In der Angabe finden Sie in der Datei *JointProductTest.java* ein Beispiel für einen JUnit-Test. In diesem File befinden sich mehrere Testfälle, welche die Funktionalität des *JointProducts* überprüfen. Grundsätzlich ist die Idee von Unit-Tests das einzelne Teile einer Klasse (in diesem Fall Methoden) einzeln geprüft werden. Es werden dazu sogenannte *TestCases* geschrieben und ausgeführt. Eclipse bietet dabei die Möglichkeit die Resultate dieser Testfälle grafisch darzustellen.

In JUnit kommen *Assertions* zum Einsatz, um das Programm zu überprüfen. Es kann dabei beispielsweise überprüft werden, ob Werte identisch sind oder ob ein bestimmter Aufruf einen korrekten Wert zurück liefert. In JUnit wird dafür die `assertEquals`-Methode verwendet. Soll zum Beispiel überprüft werden, ob die Addition zweier Variablen das korrekte Ergebnis liefert, so könnte dies wie folgt implementiert werden:

```
int x = 5;
int y = 10;
assertEquals(15, x+y);
```

JUnit verwendet *Annotations* um Methoden entsprechend zu kennzeichnen und ihnen somit eine zusätzliche Bedeutung zu geben. Beispielsweise werden in einer Klasse alle Methoden, welche die Annotation `@Test` vorangestellt haben, automatisch als Testfall erkannt und ausgeführt. In Eclipse können Sie einen solchen TestCase in einem Projekt mittels *New -> Other -> Java ->JUnit -> JUnit Test Case* erstellen. Ausführen können Sie diesen mit der *rechten Maustaste -> Run As -> JUnit Test.*
Weitere nützliche Annotationen sind:

- `@Before`: kennzeichnet eine Methode die vor jedem Testfall (`@Test`) ausgeführt wird
- `@After`: kennzeichnet eine Methode die nach jedem Testfall (`@Test`) ausgeführt wird
- `@BeforeClass`: kennzeichnet eine Methode die noch vor Erstellung der Klasse ausgeführt wird (also die erste Methode die in der Testklasse ausgeführt wird)
- `@AfterClass`: kennzeichnet eine Methode die nach allen Testfällen ausgeführt wird

Zusätzlich ist es auch möglich in JUnit zu überprüfen, ob eine Methode eine bestimmte (erwartete) Exception wirft oder nicht. Dazu kann an die @Test-Annotation noch ein Parameter angehängt werden:

```
@Test(expected=IndexOutOfBoundsException.class)
```

Ist eine Methode mit dieser Annotation versehen, so ist der Testfall erfolgreich wenn diese Exception auftritt. Dies ist beispielsweise hilfreich zur Überprüfung ob der Zugriff auf einen Index außerhalb einer Liste auch tatsächlich zum Fehlerfall führt. Der Typ der Exception kann beliebig geändert werden.

Verwenden Sie für ihre Tests JUnit 4.
Siehe http://www.vogella.com/tutorials/JUnit/article.html

Bonusaufgaben

Wählen sie eine der Bonusaufgaben aus.

Bonusaufgabe 1: Package container – Einfach verkettete Liste

Die zweite mögliche Bonusaufgabe besteht darin eine einfach verkettete Liste zu programmieren. Ein detailliertes Klassendiagramm des Package `container` ist in Abbildung 7 zu sehen. Überlegen Sie sich den Vorteil, wenn sie den abstrakten Datentyp `Collection` als Ausgangsbasis für Ihren `Container` verwenden.

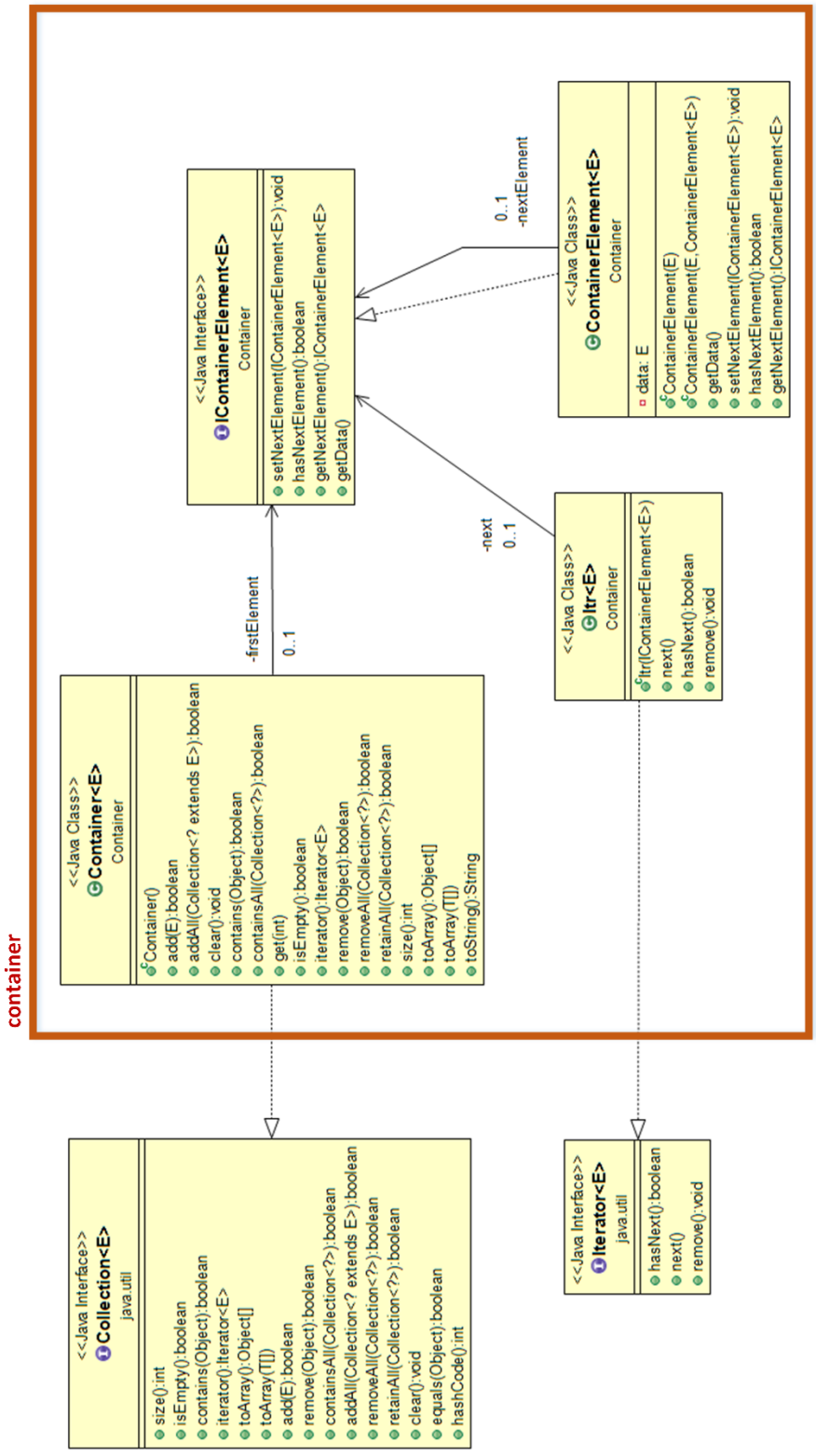

Abbildung 7 - Klassendiagramm Container

Eine einfach verkettete Liste (https://en.wikipedia.org/wiki/Linked_list) enthält mehrere `ContainerElemente` und jedes dieser Elemente hat einen Zeiger auf seinen Nachfolger. Dadurch ergibt sich eine zusammenhängende Liste an `ContainerElementen`. Für solch eine Liste wird ein `Container` benötigt, welcher unter anderem, eine Referenz auf das erste `ContainerElement` der Liste hält und die Operationen auf der Liste bereitstellt. Abbildung 8 zeigt den Aufbau einer einfach verketteten Liste.

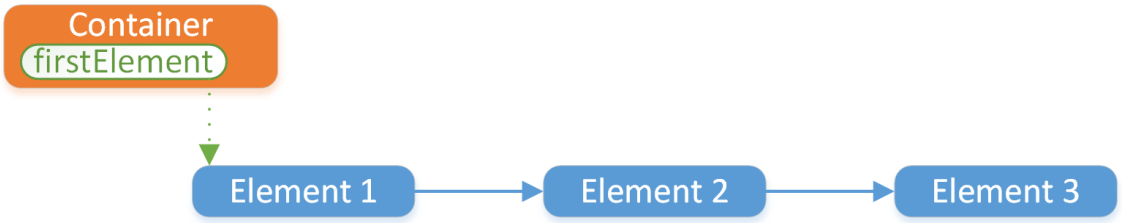

Abbildung 8 - Einfache, verkettete Liste

Java bietet bereits Schnittstellenbeschreibungen (`Interfaces`), die das Verhalten einer Liste spezifizieren, an. Ihre Aufgabe ist es nun eine Implementierung für ein solches `Interface` zu erstellen. Konkret müssen Sie eine Implementierung des `Collection`-Interfaces in der `Container`–Klasse erstellen.
http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html
Diese `Container`-Klasse wird Ihren Source-Code beinhalten und die im `Collection` Interface beschriebenen Methoden implementieren. Zusätzlich enthält diese Klasse ein Attribut `firstElement` (vom Typ `IContainerElement`), welches das erste Element in der Liste speichert.

Die Operationen die auf der verketteten Liste möglich sind, sind im `Collection` Interface definiert. Ein Auszug dieser Methoden wird hier genauer erklärt:

- `add` fügt der Liste am Ende ein neues Element hinzu (siehe Abbildung 9). Ist das Element null so soll eine **NullPointerException** geworfen werden.

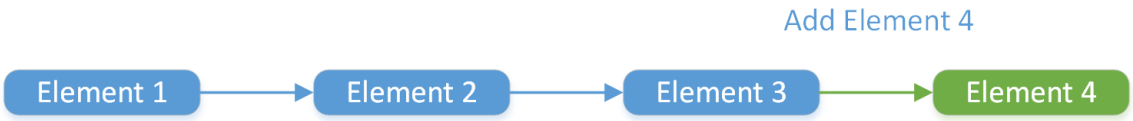

Abbildung 9 - Element hinzufügen

- `remove` entfernt ein Element aus der Liste (siehe Abbildung 10)
- `contains` überprüft ob ein bestimmtes Element in der Liste enthalten ist.
- `size` liefert die Anzahl der Elemente die in der Liste enthalten sind zurück.

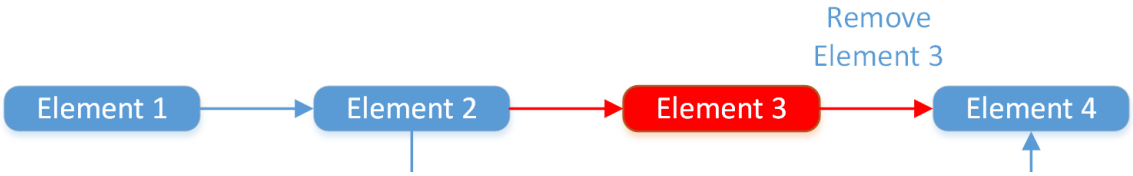

Abbildung 10 - Element entfernen

Zusätzlich definieren wir noch weitere Methoden, welche die `Container`–Klasse anbietet, die nicht Teil des `Collection`-Interface sind. Zur leichteren Handhabung der Liste sind aber auch diese zu implementieren.

- `get` liefert das Element an der übergebenen Position. Beispielsweise würde für die Liste in Abbildung 9 ein Aufruf `get(1)` das Element `2` zurückliefern. Hier ist zu beachten, dass der Index der Liste nicht bei 1 sondern bei 0 beginnt.

Operationen zum Hinzufügen, oder Entfernen, von Sammlungen (Collection) oder Listen von Elementen sind auch verfügbar. Ein möglicher Anwendungsfall hierfür wäre das Hinzufügen von mehreren Produkten zu einer andern Liste. Dafür sind eigene Methoden `addAll` und `removeAll` vorgesehen.

Die einzelnen Elemente der Liste werden in der Klasse `ContainerElement` implementiert und sind entsprechend des `IContainerElement` Interfaces zu umgesetzt. Das `ContainerElement` ist dabei recht einfach gehalten und enthält lediglich zwei Attribute:

- `data`, das den Inhalt des Elements speichert
- `nextElement` (vom Typ `IContainerElement`), das auf das nächste Element in der Liste zeigt

Um das Arbeiten und Iterieren auf der Liste einfacher zu gestalten, erstellen Sie eine Klasse `Itr`. Diese implementiert das `Iterator` Interface und wird verwendet, um über Listen zu navigieren.
http://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html
Der Iterator bietet dazu die Methoden `hasNext` und `next` an. Erstere überprüft, ob es in der Liste noch ein weiteres Element gibt, und die Zweite liefert dieses Element zurück.

Bonusaufgabe 2 – Speichern und Laden von Objekten

Implementieren Sie eine Möglichkeit Ihre Baumstruktur **Serializable** zu machen, um diese in weiterer Folge zu speichern. Erweitern Sie Ihr Programm so, dass Ihr Baum beim Beenden des Programmes gespeichert und beim Starten wieder geladen wird. Dazu müssen Sie Ihre Start-Klasse erweitern und diese zusätzliche Funktionalität in Ihrer *main*-Methode einbauen.

Speichern Sie also Ihren Baum beim Beenden des Programmes in einer Datei (*baum.txt*) und verwenden Sie diese Datei (sofern vorhanden), um beim Starten des Programmes eine existierende Baumstruktur zu laden. Verwenden Sie hierfür auch ein geeignetes **Exception Handling**.

Hinweis:
Die Klassen *FileInput-* und *ObjectInputStream* sowie *FileOutput-* und *ObjectOutputStream* können hier hilfreich sein, um Objekte in Dateien zu speichern.
https://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutputStream.html

Zusätzlich wird es notwendig sein Ihre Objekte, welche Sie speichern wollen, das *Serializable*-Interface implementieren zu lassen (dies gilt sowohl für Ihre Baum-Klassen als auch für die Elemente die Sie darin speichern wollen). Schauen Sie sich hierzu das *Serializable-Interface* an.
http://www.vogella.com/tutorials/JavaSerialization/article.html
https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html