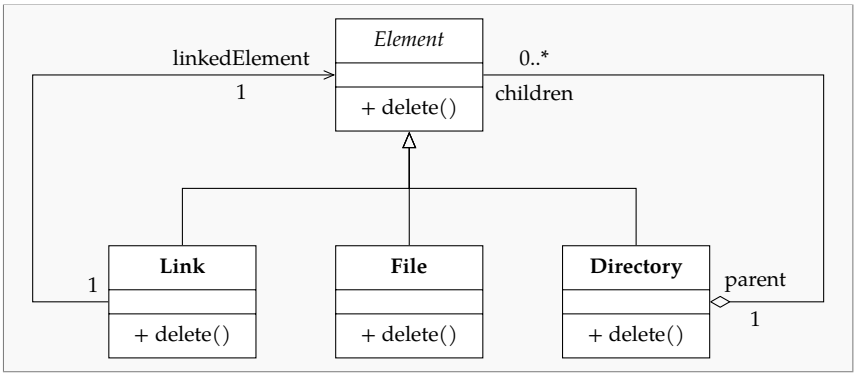
Aufgabe 1: (Modellierung und Muster)

Hierarchische Dateisysteme bestehen aus den FileSystemElements Ordner, Dateien und Verweise. Ein Ordner kann seinerseits Ordner, Dateien und Verweise beinhalten; jedem Ordner ist bekannt, welche Elemente (children) er enthält. Mit Ausnahme des Root-Ordners auf der obersten Hierarchieebene ist jeder Ordner, jede Datei und jeder Verweis Element eines Elternordners. Jedem Element ist bekannt, was sein Elternordner ist (parent). Ein Verweis verweist auf einen Verweis, eine Datei oder einen Ordner (link). Wenn ein Ordner gelöscht wird, werden alle seine Bestandteile ggf. rekursiv ebenfalls gelöscht. Sie dürfen die Lösungen für Aufgabenteil b) und c) in einem gemeinsamen Code kombinieren.

(a) Modellieren Sie diesen Sachverhalt mit einem UML-Klassendiagramm. Benennen Sie die Rollen von Assoziationen und geben Sie alle Kardinalitäten an. Ihre Lösung soll mindestens eine sinnvolle Spezialisierungsbeziehung enthalten.



(b) Implementieren Sie das Klassendiagramm als Java- oder C++-Programm. Jedes Element des Dateisystems soll mindestens über ein Attribut `name` verfügen. Übergeben Sie den Elternordner jedes Elements als Parameter an den Konstruktor; der Elternordner des Root-Ordners kann dabei als `null` implementiert werden. Dokumentieren Sie Ihren Code.

a

```
3  public abstract class Element {
4
5    protected String name;
6
7    protected Element parent;
8
9    protected Element(String name, Element parent) {
10     this.name = name;
11     this.parent = parent;
12   }
13
14   public String getName() {
15     return name;
16   }
17
```

```
18    public abstract void delete();
19
20    public abstract boolean isDirectory();
21
22    public abstract void addChild(Element child);
23  }
```

```
3   import java.util.ArrayList;
4   import java.util.List;
5
6   public class Directory extends Element {
7     private List<Element> children;
8
9     public Directory(String name, Element parent) {
10      super(name, parent);
11      children = new ArrayList<Element>();
12      if (parent != null)
13        parent.addChild(this);
14    }
15
16    public void delete() {
17      System.out.println("The directory "" + name +
        ↪  "" was deleted and it's children were also deleted.");
18      for (int i = 0; i < children.size(); i++) {
19        Element child = children.get(i);
20        child.delete();
21      }
22    }
23
24    public void addChild(Element child) {
25      children.add(child);
26    }
27
28    public boolean isDirectory() {
29      return true;
30    }
31  }
```

```
3   public class File extends Element {
4
5     public File(String name, Element parent) {
6       super(name, parent);
7       parent.addChild(this);
8     }
9
10    public void delete() {
11      System.out.println("The File "" + name + "" was deleted.");
12    }
13
14    public boolean isDirectory() {
15      return false;
16    }
17
18    /**
19     * Eine Datei kann keine Kinder haben. Deshalb eine Methode mit
      ↪  leerem
20     * Methodenrumpf.
21     */
22    public void addChild(Element child) {
23    }
24
```

```
25  }

3   public class Link extends Element {
4
5     private Element linkedElement;
6
7     public Link(String name, Element parent, Element linkedElement) {
8       super(name, parent);
9       this.linkedElement = linkedElement;
10      parent.addChild(this);
11    }
12
13    public void delete() {
14      System.out.println("The Symbolic Link "" + name +
        ↪  "" was deleted.");
15      linkedElement.delete();
16      System.out.println("The linked element "" + name +
        ↪  "" was deleted too.");
17    }
18
19    public void addChild(Element child) {
20      if (linkedElement.isDirectory())
21        linkedElement.addChild(child);
22    }
23
24    public boolean isDirectory() {
25      return linkedElement.isDirectory();
26    }
27  }
```

[a] TU Darmstadt: Dr. Michael Eichberg - Case Study Using the Composite and Proxy Design Patterns

(c) Ordnen Sie eine Methode `delete`, die Dateien, Ordner und Verweise rekursiv löscht, einer oder mehreren geeigneten Klassen zu und implementieren Sie sie. Zeigen Sie die Löschung jedes Elements durch eine Textausgabe von `name` an. `toString()` müssen Sie dabei nicht implementieren. Gehen Sie davon aus, dass Verweis- und Ordnerstrukturen azyklisch sind und dass jedes Element des Dateisystems höchstens einmal existiert. Wenn ein Verweis gelöscht wird, wird sowohl der Verweis als auch das verwiesene Element bzw. transitiv die Kette der verwiesenen Elemente gelöscht. Bedenken Sie, dass die Löschung eines Elements immer auch Konsequenzen für den dieses Element beinhaltenden Ordner hat. Es gibt keinen Punktabzug, wenn Sie die Löschung des Root-Ordners nicht zulassen.

Siehe Antwort zu Aufgabe b)

(d) Was kann im Fall von `delete` passieren, wenn die Linkstruktur zyklisch ist oder die Ordnerstruktur zyklisch ist? Kann es zu diesen Problemen auch dann führen, wenn weder die Linkstruktur zyklisch ist, noch die Ordnerstruktur zyklisch ist? Wie kann man im Programm das Problem lösen, falls man Zyklizitäten zulassen möchte?

Falls die Link- oder Ordnerstruktur zyklisch ist, kann es aufgrund der Rekursion zu einer Endlosschleife kommen. Diese Problem tritt bei azyklischen Strukturen nicht auf, weil der rekursive Löschvor-

gang beim letzten Element abgebrochen wird (Abbruchbedingung).

Das Problem kann zum Beispiel durch ein neues Attribut `gelöscht` in der Klasse `Link` oder `Directory` gelöst werden. Dieses Attribut wird auf `true` gesetzt, bevor es in die Rekursion einsteigt. Rufen sich die Klassen wegen der Zyklizität selbst wieder auf, kommt es durch entsprechende IF-Bedingungen zum Abbruch.

Außerdem ist ein Zähler denkbar, der sich bei jeder Rekursion hochsetzt und ab einem gewissen Grenzwert zum Abbruch führt.

(e) Was ist ein Design Pattern? Nennen Sie drei Beispiele und erläutern Sie sie kurz. Welches Design Pattern bietet sich für die Behandlung von hierarchischen Teil-Ganzes-Beziehungen an, wie sie im Beispiel des Dateisystems vorliegen?

Design Pattern sind wiederkehrende, geprüfte, bewährte Lösungsschablonen für typische Probleme.

Drei Beispiele

Einzelstück (Singleton) Stellt sicher, dass nur *genau eine Instanz einer Klasse* erzeugt wird.

Beobachter (Observer) Das Observer-Muster ermöglicht einem oder mehreren Objekten, automatisch auf die *Zustandsänderung* eines bestimmten Objekts *zu reagieren*, um den eigenen Zustand anzupassen.

Stellvertreter (Proxy) Ein Proxy stellt einen Platzhalter für eine andere Komponente (Objekt) dar und kontrolliert den Zugang zum echten Objekt.

Für hierarchischen Teil-Ganzes-Beziehungen eigent sich das Kompositum (Composite). Es ermöglicht die Gleichbehandlung von Einzelelementen und Elementgruppierungen in einer verschachtelten Struktur (z. B. Baum), sodass aus Sicht des Clients keine explizite Unterscheidung notwendig ist.

```
3  import org.junit.Before;
4
5  import org.junit.Test;
6
7  public class FileSystemTest {
8
9    Element wurzelOrdner;
10   Element unterOrdner;
11   Element datei;
12   Element verweis;
13
14   @Before
15   public void legeZoneAn() {
16     wurzelOrdner = new Directory("Wurzel", null);
17     unterOrdner = new Directory("Unterordner", wurzelOrdner);
```

```
18      datei = new File("Datei", wurzelOrdner);
19      verweis = new Link("Verweis", wurzelOrdner, datei);
20    }
21
22    @Test
23    public void löscheWurzelOrdner() {
24      wurzelOrdner.delete();
25    }
26
27  }
```