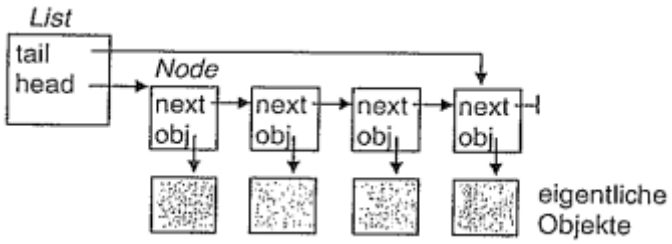
Listen und Iteratoren (40 Punkte)

Implementieren Sie eine Klasse List, die Objekte belieben Typs gemäß folgender Skizze in einer sequentiell verketteten Liste speichert:



Neben einem Konstruktor soll es eine Methode *add(obj)* geben, um ein Objekt an das Ende der Liste anzuhängen sowie eine Methode *remove(obj),* um das Objekt obj aus der Liste zu löschen, wenn es darin enthalten ist.

Ferner soll die Liste eine Funktionsmethode *iterator()* haben, die einen Iterator liefert, mit dem man die Liste durchlaufen kann. Der Iterator soll folgendes Iterface *Iterable* implementieren:

```
Interface Iterable {
        Object next();
        Boolean hasNext();
}
```

Implementieren Sie die Klasse Node, List und Iterator ohne Zuhilfenahme der Collection-Klassen der Java-Bibliothek.

class Node {
    // Deklaration der Node-Felder und des Node-Kostruktors

```
public class Node {
      Object obj;
      Node next;

      public Node(Object obj) {
            this.obj = obj;
            next = null;
      }
}
```

class List {
    // Deklaration der List-Felder und des List-Konstruktors

```
public class List {
      Node head;
      Node tail;

      public List() {
            head = null;
            tail = null;
      }
}
```

// Methode add(obj): fügt das Objekt obj an das Ende der Liste hinzu

```
public void add(Object obj) {
            Node newNode = new Node(obj);
            if (head == null) {
                  head = newNode;
                  tail = newNode;
            } else {
                  tail.next = newNode;
                  tail = newNode;
            }
      }
}
```

// Methode remove(obj): entfernt das erste Vorkommen von obj aus der Liste
// Annahme: Objekte lassen sich mit Methode equals vergleichen

```
public boolean remove (Object obj) {
            if (head == null) return false;
            if (head != null) {
                  Node akt = head;
                  while (akt != null) {
                        if (akt.equals(obj)) {
                              akt = akt.next;
                        }
                  }
            }
            return true;
      }
```

// Methode iterator(): liefert einen Iterator zum Durchlaufen der Liste
Iterable iterator() {

```
Iterable iterator(Object obj) {
            Iterator it = new Iterator();
            return it;
      }
```

class Iterator implements Iterable {

// Felder und Konstruktor

```
public class Iterator implements Iterable{
      Node next;

      public Iterator(Node head) {
            next = head;
      }

      // Methode next(): liefert das nächste Objekt der Liste

      public Object next() {
            Node current = next;
            next = next.next;
            return current;
      }
```

```
        // Methode hasNext(): liefert true, wenn der Iterator noch nicht am Ende der Liste
        // angelangt ist

        public boolean hasNext() {
              return next != null;
        }
}
```

Schreiben Sie ein Codefragment, das eine neue Liste erzeugt und darin die Strings „Anton“ und „Berta“ einfügt:

```
List l = new List();
            l.add("Anton");
            l.add("Berta");
```

Schreiben Sie ferner ein Codefragment, das die soeben erzeugte Liste mit Hilfe eines Iterators durchläuft und die Objekte darin auf dem Bildschirm ausgibt:

```
Iterator it = l.iterator();
            while (it.hasNext()){
                  Node next = it.next;
                  System.out.println(next);
            }
```

Entwurfsmuster Dekorator (20 Punkte)

Gegeben sei eine Klasse Rectangle mit folgender Schnittstelle:

```
class Rectangle {
        int x, y;          // Position der linken oberen Ecke
        int width;         // Breite
        int height;        // Höhe

        Rectangle (int x, int y, int w, int h) { … }

        void move (int dx, int dy) { … }   // verschiebt die Position um (dx, yx)
        void resize (int dx, int dy) { … } // erhöht die Breite um dx und die Höhe um yx
```

Implementieren Sie mit Hilfe des Dekoraor-Musers eine Klasse GridRectangle. Objekte dieser Klasse sollen vor Rectangle-Objekte geschaltet werden können und durch Rundung dafür sorgen, dass die Position sowie die Breite und Höhe von Rechtecken immer ganze Vielfache von 10 sind.

Überschreiben Sie dazu die Methoden move() und resize(). Implementieren Sie auch einen entsprechenden Konstruktor für GridRectangle, so dass man ein GridRectangle-Objekt vor ein Rectangle-Objekt schalten kann.

// Klasse GridRectangle

```
public class GridRectangle extends Rectangle {
	Rectangle toDecorate;

	public GridRectangle(Rectangle rect) {
		super(rect.x, rect.y, rect.height, rect.height);
		toDecorate = rect;
		toDecorate.y = grid(toDecorate.y);
		toDecorate.x = grid(toDecorate.x);
		toDecorate.height = grid(toDecorate.height);
		toDecorate.width = grid(toDecorate.width);
	}

	public int grid (int toGrid) {
		toGrid = ((toGrid + 5) / 10 * 10);
		return toGrid;
	}
	public void move(int dx, int dy) {
		dx = grid(dx);
		dy = grid(dy);
		toDecorate.move(dx, dy);
	}

	public void resize(int dx, int dy) {
		dx = grid(dx);
		dy = grid(dy);
		toDecorate.resize(dx, dy);
	}
}
```

Schreiben Sie ein Codefragment, das ein Rechtecht an der Position (17, 24) mit der Breite 25 und der Höhe 35 erzeugt, mit einem GridRectangle-Objekt dekoriert und dieses anschließend um dx = 12 und dy = 17 verschiebt.

```
public class Main {
	public static void main(String[] args) {
		Rectangle rect = new Rectangle(17, 24, 25, 35);
		GridRectangle gridRect = new GridRectangle(rect);
		gridRect.move(12, 17);
	}
}
```

Welche Position, Breit und Höhe hat das Rechteck anschließend?

```
x = 30; y = 40; Breite = 30; Höhe = 40;
```

Testen (22 Punkte)

a) Erklären Sie den Unterschied zwischen Flack-Box-Testen und White-Box-Testen. Welche Ziele verfolgt man beim Aufstellen der Testfälle?

Black-Box-Testen: Unit Tests, Funktionale Tests, Schnittstellentest.

Hier wird das Input/Output Verhalten getestet. Der Quellcode selbst wird ignoriert. Die Testfälle werden nur aus den Schnittstellen bzw. Spezifikationen abgeleitet. Der Tester muss sich nicht in den Code einarbeiten und ist somit unvoreingenommen.

White-Box-Testen: Strukturtest

Hier kennt man die Implementierung. Man versucht dabei den gesamten Quellcode mit Testfällen abzudecken. Man kann dafür sorgen, dass das Programm keinen ungetesteten Code enthält und die Kenntnis über das Programm stellt sicher, dass man weiß wo besonders intensiv getestet werden muss.

b) Gegeben sei folgendes Programmstück:

```
if (a < 0) {
        If (b < 0) x = 1; else x = 2;
}
if (c == 0) x = 3;
```

Mit welchen Werten von a, b und c müssen Sie dieses Programm testen, um 100% Anweisungsabdeckung zu erhalten?

a = -1; b = -1; c = 0

a = -1; b = 0; c = -1

Mit welchen Werten von a, b und c müssen Sie es testen, um 100% Bedingungsabdeckung zu erhalten?

A = -1; b = -1; c -1

A = -1, b = 0; c = 0;

A = 0; c = 0; b = beliebig

Mit welchen Werten von a, b und c müssen Sie es testen, um 100% Pfadabdeckung zu erhalten?

A = -1; b = -1; c = 0

A = -1; b = -1; c = 1

A = -1; b = 0; c = 0

A = -1; b = 0; c = 1

A = 0; c = 1, b = beliebig

A = 0; c = 0, b = beliebig

Welche Komplexität hat obriges Programmstück nach McCabe?

3 Abfragen ➔ Komplexität von 4

Fragen (8 Punkte)

Kreuzen Sie die richtigen Antworten an. Es kann mehrere richtige Antworten pro Frage geben. Falsche Antworten bringen aber Punkteabzug:

a) Ein Type Case (T)obj

- ○ Ändert den Wert von obj auf T
- ⊗ Ändert den statischen Typ des Ausdrucks (T)obj auf T
- ○ Ändert den dynamischen Typ des Ausdrucks (T)obj auf T

b) Eine anonyme Klasse

- ⊗ Hat keinen Namen
- ○ Definiert einen zur Compilezeit unbekannten Typ
- ○ Definiert einen zur Laufzeit unbekannten Typ

c) In welcher Weise dürfen die Vor- und Nachbedingungen einer Methode beim Überschreiben einer Unterklasse abgeändert werden, so dass keine Kontrakte verletzt werden? Geben Sie eine prinzipielle Antwort, d.h. unabhängig von Java.

Damit der Kontrakt nicht verletzt wird, darf der Subkontraktor nicht mehr erwarten und nicht weniger garantieren als der ursprüngliche Kontraktor (d.h. die Oberklasse). Kovarianz: Der Rückgabetyp einer Unterklasse darf spezifischer werden, als in der Superklasse. Kontravarianz: Die Eingabedaten einer Unterklasse müssen mindestens jenen der Superklasse entsprechen