1. **Decorator**

   Mit Hilfe des Dekorator-Pattern ist eine Familie von Zählern zu implementieren.

   Das Interface `Counter` definiert ganzzahlige Zähler mit nicht negativen Werten. Die Methoden lesen einen Zähler ab (`read()`) und zählen ihn weiter (`tick()`). Zähler liefern endlos Werte.

```
public interface Counter {
  /**
   * Liefert den aktuellen Zaehlerstand.
   *
   * @return aktueller Zaehlerstand.
   */
   int read();

   /**
    * Zaehlt weiter.
    *
    * @return dieser Zaehler.
    */
   Counter tick();
}
```

   Elementare Zähler - konkrete Komponenten

   Die konkrete Klasse `UCounter` implementiert das Interface `Counter`. Ein `UCounter` startet mit 0 und zählt in Einerschritten hoch. (Ignorieren Sie in der ganzen Aufgabe den begrenzten Wertebereich von `int`.) Ein Beispiel:

```
Counter counter = new UCounter();
System.out.println(counter.read());             // 0
System.out.println(counter.tick().read());      // 1
System.out.println(counter.tick().read());      // 2
```

   Er liefert also die ganzen Zahlen 0, 1, 2, ...
   Hinweis: In diesem und den nachfolgenden Beispielen steht ”liefert” für abwechselnde Aufrufe von `read()` und `tick()`.

   Definieren Sie außerdem die folgenden elementaren Zähler:

   (a) `LoopCounter`
   Liefert die Werte, die der Konstruktor als Argumente erhält (`NoSuchElementException` falls keine angegeben). Fängt nach dem letzten Wert wieder von vorne an.
   Beispiel: `new LoopCounter(1, 2, 3)` liefert `1, 2, 3, 1, 2, 3` und so fort.

   (b) `BaseCounter`
   Liefert Zahlen in dem Zahlensystem mit der Basis, die der Konstruktor als Argument erhält (2 - 9, sonst `IllegalArgumentException`).
   Beispiel: `new BaseCounter(3)` liefert `0, 1, 2, 10, 11, 12, 20` und so weiter im Zahlensystem zur Basis 3.

Filterzähler - Dekorierer

Eine andere Art von Zählern sind Filterzähler. Filterzähler kapseln genau einen anderen, zugrunde liegenden Zähler und modifizieren dessen Verhalten auf eine bestimmte Art. Darüber hinaus sind Filterzähler selbst Zähler. Alle Filterzähler-Konstruktoren erwarten als erstes Argument (`NullPointerException` falls `null`) einen zugrunde liegenden Zähler. Darüber hinaus können Sie weitere Argumente akzeptieren.

Definieren Sie die folgenden Filterzähler:

(a) `PrintCounter`
Gibt den Zählerstand des zugrunde liegenden Zählers vor dem Weiterzählen aus. Der `PrintCounter`-Konstruktor erhält einen zugrunde liegenden Zähler und ein Zeichen, das er an jeden ausgegebenen Zählerstand anfügt.

```
Counter counter = new PrintCounter(new UCounter(), '\n');
counter.tick().tick().tick();
// Ausgabe auf System.out: Zeilen mit 0, 1, 2
```

(b) `JumpCounter`
Zählt den zugrunde liegenden Zähler bei jedem `tick()` so oft weiter, wie der Konstruktor als zweites Argument (`IllegalArgumentException` falls negativ) erhält.
Beispiel: `new JumpCounter(new UCounter(), 2)` liefert `0, 2, 4, 6, ...`

(c) `LimitedCounter`
Liefert den Wert des zugrunde liegenden Zählers nur dann zurück, wenn er unter dem Grenzwert liegt, den der Konstruktor als zweites Argument (`IllegalArgumentException` falls negativ) erhält. Andernfalls liefert er den Grenzwert selbst zurück.
Beispiel: `new LimitedCounter(new UCounter(), 2)` liefert `0, 1, 2, 2, 2, 2, ...`

(d) `MultiCounter`
Liefert jeden Wert des zugrunde liegenden Zählers so oft zurück, wie das zweite Argument (`IllegalArgumentException` falls ≤ 0) angibt.
Beispiel: `new MultiCounter(new UCounter(), 2)` liefert `0, 0, 1, 1, 2, 2, ...`

Testprogramm

Die Implementierung sollte mit Hilfe des folgenden Testprogramms die angegebene Ausgabe liefern:

```
public class CounterTest {

  public static void main(String[] args) {
    Counter c1 = new BaseCounter(5);
    System.out.print(c1.read() + " ");
    for(int i = 0; i < 20; i++) {
      System.out.print(c1.tick().read() + " ");
    }
    System.out.println();
    Counter c2 = new LoopCounter(1,3,5,7,9);
    System.out.print(c2.read() + " ");
    for(int i = 0; i < 20; i++) {
      System.out.print(c2.tick().read() + " ");
```

```
    }
    System.out.println();
    Counter c3 = new PrintCounter(new UCounter(), 'x');
    c3.tick();
    for(int i = 0; i < 20; i++) {
      c3.tick();
    }
    System.out.println();
    Counter c4 = new PrintCounter(new JumpCounter(new UCounter(), 3), ' ');
    c4.tick();
    for(int i = 0; i < 20; i++) {
      c4.tick();
    }
    System.out.println();
    Counter c5 = new LimitedCounter(new JumpCounter(new UCounter(), 3), 20);
    System.out.print(c5.read() + " ");
    for(int i = 0; i < 20; i++) {
      System.out.print(c5.tick().read() + " ");
    }
    System.out.println();
    Counter c6 = new MultiCounter(new JumpCounter(new UCounter(), 3), 2);
    System.out.print(c6.read() + " ");
    for(int i = 0; i < 20; i++) {
      System.out.print(c6.tick().read() + " ");
    }
    System.out.println();
    System.out.println();
    Counter c7 = new MultiCounter(new BaseCounter(2), 5);
    System.out.print(c7.read() + " ");
    for(int i = 0; i < 20; i++) {
      System.out.print(c7.tick().read() + " ");
    }
  }
}
```

Ausgabe:

```
0 1 2 3 4 10 11 12 13 14 20 21 22 23 24 30 31 32 33 34 40
1 3 5 7 9 1 3 5 7 9 1 3 5 7 9 1 3 5 7 9 1
0x1x2x3x4x5x6x7x8x9x10x11x12x13x14x15x16x17x18x19x20x
0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60
0 3 6 9 12 15 18 20 20 20 20 20 20 20 20 20 20 20 20 20 20
0 0 3 3 6 6 9 9 12 12 15 15 18 18 21 21 24 24 27 27 30
0 0 0 0 0 1 1 1 1 1 10 10 10 10 10 11 11 11 11 11 100
```

Hinweise und Abgabe

- Es ist vorteilhaft, zwischen dem Interface `Counter` und den konkreten Zählern eine abstrakte Klasse einzuschalten, die den Zählerstand kapselt.
- Das Interface `Counter` darf nicht verändert werden.
- Bei der Benotung ist auch ein UML-Klassendiagramm vorzuweisen, das die Architektur dieses Pattern zeigt.

2. Observer

ObservableTreeSet

Es ist ein `ObservableTreeSet<T>` zu implementieren, das Beobachter benachrichtigt, wann immer sich die Größe der Collection ändert. Zur Lösung dieser Aufgabe ist wie folgt vorzugehen:

- Erstelle eine Klasse, die von `java.util.TreeSet<T>` erbt.
- Implementiere alle notwendigen Konstruktoren und rufe die entsprechenden Superklassenkonstruktoren auf.
- Erstelle ein geeignetes Interface und ein zugehöriges geeignetes EventObject. Dieses sollte wenigstens die Informationen der Eventsource, einen Timestamp und eine Information zur Operation enthalten, mit welcher die Größe der Collection verändert wurde.
- Implementiere die notwendige Observable-Funktionalität.
- Überschreibe die Methoden `add()`, `remove()`, `pollFirst()`, `pollLast()`, `addAll()`, `removeAll()` und `retainAll()`, rufe dort jeweils die Superklassenmethode für die eigentliche Funktionalität auf und verständige alle registrierten Observer.

Anwendung

Teste das ObservableTreeSet mit Hilfe eines Programms mit grafischer Oberfläche (JavaF) mit folgender Funktionalität:

- Die Anwendung verwaltet ein `ObsevableTreeSet<Integer>`.
- Es gibt zwei Observer, die über geeignete Oberflächenelemente jederzeit vom Benutzer als Observer an- bzw. abgemeldet werden können. Einen LogObserver, der Veränderungen des ObservableTreeSet in einer Datei mitloggt. Der Controller der grafischen Anwendung selbst ist ein Observer und nach jeder Veränderung wird mit Hilfe eines geeigneten grafischen Elements die Art der Veränderung des ObservableTreeSet samt `toString()` - Darstellung angezeigt.
- Für jede Methode, mit welcher das ObservableTreeSet verändert werden kann, ist auf der Oberfläche ein Button zur Verfügung zu stellen, damit die entsprechende Funktionalität vom Benutzer ausgelöst werden kann. Es sind dabei zufällige Integerwerte oder Collections mit Werten im Bereich von 1 bis 99 zu erzeugen und die entsprechende Methode ist aufzurufen.