Übungsblatt 12

- Exceptions und Assertions -

Aufgabe 1: Checked- und Unchecked-Exceptions

Entwickeln Sie 2 eigene Exception-Klassen `Fehler1` und `Fehler2`, wobei `Fehler2` eine checked Exception sein soll, `Fehler1` nicht.

Schreiben Sie eine Klasse mit zwei Methoden `testen1` und `testen2`, in denen jeweils eine Exception `Fehler1` bzw. `Fehler2` ausgelöst wird. Müssen Sie im Methodenkopf einer oder beider Methoden zwingend etwas angeben? Was können Sie angeben / weglassen?

Schreiben Sie ein Testprogramm mit einer main Methode, in dem die beiden Methoden aufgerufen werden. Sorgen Sie dafür, dass **alle** auftretenden Exceptions auch abgefangen werden. Machen Sie die überprüfte Region in ihrem Programmcode pro Exception möglichst klein (nur die wirklich zu überprüfenden Codezeilen). Welche Exceptions müssen hier abgefangen werden, welche können Sie abfangen? Was passiert, wenn Exceptions auftreten, die nicht abgefangen werden müssen, und Sie diese auch nicht abfangen?

Aufgabe 2: Zusicherungen und Fehlerbehandlungen in alten Aufgaben

Schauen Sie sich ihre alten Lösungen an.

a) Wo würden welche Assertions in ihren Lösungen Sinn machen? Wo hätten diese Ihnen bei der Entwicklung geholfen?

b) Wo würden welche Exceptions in ihren Lösungen Sinn machen?

Aufgabe 3: Stack

Gegeben ist folgende Klasse für int-Stacks:

```java
/**
 * Integer Stack (original)
 * @author Rudolf Berrendorf
 * @version 1.0
 */
public class IntStack {

    static int belegt;          // belegter Speicher aller Stacks in Bytes
    int maximalGroesse;         // maximale Groesse
    int sp;                     // aktuelle Groesse
    int[] a;                    // Daten

    /**
     * Konstruktor
     * @param n maximale Groesse des Stacks
     */
    IntStack(int n) {
        maximalGroesse = n;
        sp = -1;
        a = new int[n];
        belegt += n * 4;
    }
```

```
    /**
     * Test, ob Stack leer ist
     * @return true, falls Stack leer ist, false sonst
     */
    public boolean isemptystack() {
        return (sp == -1);
    }

    /**
     * legt ein Element auf den Stack ab
     * @param x abzulegendes Element
     */
    public void push(int x) {
        a[++sp] = x;
    }

    /**
     * holt oberstes Element vom Stack und loescht es dort
     * @return oberstes Element vom Stack
     */
    public int pop() {
        return a[sp--];
    }

    /**
     * liefert den aktuellen Speicherverbrauch aller Stacks in Bytes
     * @return aktueller Stand
     */
    public static int gibSpeicherverbrauch() {
        return belegt;
    }

    /**
     * Testprogramm
     */
    public static void main(String[] args) {
        IntStack s1 = new IntStack(5);
        IntStack s2 = new IntStack(10);

        for (int i = 0; i <= 5; i++) {
            s1.push(i);
            s2.push(i);
        }

        System.out.println(s1.isemptystack());

        for (int i = 0; i <= 5; i++) {
            System.out.println(s1.pop());
            s1.pop();
            s2.pop();
        }

        System.out.println(s1.isemptystack());
    }
}
```

a) Implementieren Sie eine `StackoverflowException`, welche ausgelöst wird, wenn auf einem vollständig gefüllten Stack eine `push`-Operation ausgeführt wird.

b) Implementieren Sie eine `StackunderflowException`, welche ausgelöst wird, wenn auf einem leeren Stack eine `pop`-Operation ausgeführt wird.

c) Ergänzen Sie diese Klasse sinnvoll mit Assertions. Überlegen Sie sich dazu, wie eine (möglichst vollständige) Invariante zu dieser Stack-Implementierung aussehen kann und wie sie mit Assertions genutzt werden kann,

Info

Eine Invariante ist eine logische Bedingung (die aus mehreren verknüpften Bedingungen bestehen kann), die für jeden Stack vor und nach jeder Operation auf dem Stack gelten muss.

Aufgabe 4: Prüfstand

Es sollen in dieser Aufgabe Meßdaten verarbeitet werden, wie sie auf einem Labormeßstand in der Fahrzeugentwicklung anfallen.

- Eine einzelne Messung gekapselt in der Klasse `MessDatum` besteht dabei aus folgenden Angaben:
  - Umdrehungszahl $u \mid 0 \leq u \leq 12000, u \in \mathbb{N}$. Liegt eine Umdrehungszahl zwischen 8000 und 12000 einschließlich, so ist ein kritischer Zustand erreicht.
  - Öltemperatur t in Grad Celsius $t \mid 0 \leq t \leq 200, t \in \mathbb{R}$ (Frost lassen wir weg). Steigt die Öltemperatur in den Bereich $150 \leq t \leq 200$ Grad Celsius, so wird ein kritischer Wert erreicht.
  - Ladedruck d des Turboladers $d \mid 0 \leq d \leq 6, d \in \mathbb{R}$. Steigt der Ladedruck in den Bereich 5 - 6 bar einschießlich, so wird ein kritischer Zustand erreicht.

Beispiel

Eine Messung wäre also: `6500 85.2 4.4` und die dazugehörige Instanziierung eines Objektes wäre:

```
new MessDatum(6500, 85.2, 4.4);
```

- Sehen Sie für möglicherweise auftretende Probleme drei eigene Exception-Klassen vor: `IllegalValue` bei Auftretens eines Wertes außerhalb der Spezifikation, `CriticalValue` bei dem Vorliegen eines erlaubten Wertes im kritischen Bereich und `TooMuchData`, wenn mehr Daten als in einer Meßreihe erlaubt, angefallen sind. `IllegalValue` ist eine *unchecked* Exception, die anderen beiden sollen *checked* Exceptions sein.
- Eine Meßbereichsangabe (von, bis, ab-hier-kritisch), wie sie für alle drei Meßparameter definiert ist, lässt sich ebenfalls kapseln in einer Klasse MessBereich. Sehen Sie in einer Instanzmethode `void pruefen(double wert)` in dieser Klasse eine Überprüfung eines Wertes zu dem Meßbereich vor, der mit diesem Meßbereichsobjekt festgelegt ist. Liegt der Wert `wert` im „grünen Bereich“ dieses Meßbereichs, passiert nichts. Liegt ein erlaubter aber kritischer Wert vor, so soll eine Exception `CriticalValue` geschmissen werden. Bei einem Wert außerhalb der Spezifikation soll eine Exception `IllegalValue` geschmissen werden.

Beispiel

Ein Beispiel für die Instanziierung eines Meßbereichsobjektes zu Umdrehungszahlen ist

```
MessBereich mb = new MessBereich(0, 12000, 8000);
```

eine anschließende Überprüfung wäre etwa `mb.pruefen(9000)`, was zu einer Exception `CriticalValue` führen sollte.

- Ein `Meßreihe` (zugehörige Klasse MessReihe) ist ein Folge von maximal n Datensätzen. Die maximale Anzahl n ist ein Parameter des Konstruktors dieser Klasse. Definieren Sie zu einem Meßreihenobjekt (nicht zu einem MessDatum) drei Meßbereichsobjekte, jeweils passend für Umdrehungszahl, Öltemperatur und Ladedruck.

Einzelne Meßdatensätze können über die Instanzmethode `void neueMessung(int u, double t, double d)` zu einer Meßreihe hinzugefügt werden. Jeder einzelne dieser drei Werte wird zuerst anhand des passenden `Meßbereichsobjekts` auf einen gültigen Wert und kritischen Wert überprüft. Führt eine solche Überprüfung zu einer Exception, so soll die Exception hier nicht behandelt werden, sondern *propagiert* werden. Wurde bereits die Maximalzahl an Datensätzen erreicht, so führt diese Methode zu einer `TooMuchData` Exception, die ebenfalls *propagiert* werden soll. Ansonsten (alle drei Werte sind im erlaubten nicht-kritischen Bereich und es ist noch Platz da) wird der Datensatz zum Datensatzbestand hinzugefügt. Zum aktuellen Stand einer Meßreihe kann man auch Mittelwerte zu den einzelnen Daten ermitteln über die Methode `double[] ermittleMittelwerte()`. Das Resultat sind also drei Mittelwerte, einen für Umdrehungszahl, einen für Öltemperatur und einen für Ladedruck. Das Ergebnis der Methode ist ein Feld mit diesen drei Werten.

Beispiel

Zu den beiden Datensätzen `(5000, 85, 4)` und `(6000, 95, 5)` wären die Mittelwerte `(5500, 90, 4.5)`

- Einer Klasse `LaborStand` werden nun als Kommandozeilenparameter der Methode main zwei Argumente übergeben:
  - die Anzahl n der maximalen Datensätze. Damit wird ein Meßreihenobjekt erzeugt.
  - der Name einer Datei, in der Datensätze stehen

  Öffnen Sie die Datei über einen Scanner (siehe Grundgerüst weiter unten). Solange in dieser Datei Datensätze vorhanden sind (sc.hasNext() ist true) lesen Sie von dieser Datei jeweils genau einen Datensatz. Jeder Datensatz besteht aus 3 Zahlen (int, double, double). Mit diesen drei gelesenen Werten rufen Sie die Methode void neue-Messung(int u, double t, double d) zu dem Meßreihenobjekt auf. Dort werden die Werte auf Zulässigkeit überprüft (siehe oben, wie das geschehen soll) und falls kein illegaler oder kritischer Wert vorliegt, wird der Datensatz zum vorhandenen Datenbestand hinzugefügt. Alle Exceptions werden ja propagiert und sollen nur in main (und nur dort) behandelt werden. Als Reaktion auf eine IllegalValue- oder TooMuchData-Exception soll eine Fehlermeldung auf dem Bildschirm ausgegeben werden (Fehler: ...), ansonsten aber weiter gemacht werden. Bei einer CriticalValue-Exception soll eine Fehlermeldung ausgegeben werden (Fehler: ...„) und eine Notabschaltung erfolgen (System.exit(1)). Sind alle Datenätze aus der Datei verarbeitet worden, sollen die Mittelwerte zu den Meßdaten ermittelt und auf dem Bildschirm ausgegeben werden.

Das Grobgerüst der main in LaborStand (ohne Exception-Handling etc.) wäre also:

```
// maximale Anzahl der Datensaetze holen
int maxAnzahlDatensaetze = Integer.parseInt(args[0]);

// Scanner von der Datei anlegen
Scanner sc = new Scanner(new File(args[1]));

// Messreihenobjekt anlegen
MessReihe messReihe = new MessReihe(maxAnzahlDatensaetze);

// alle vorhandenen Datensaetze lesen
while (sc.hasNext()) {

    // einen Datensatz lesen
    int u = sc.nextInt();
    double t = sc.nextDouble();
    double d = sc.nextDouble();

    messReihe.neueMessung(u, t, d);
}

// Scanner schliessen
sc.close();
```

```
// Mittelwerte berechnen
double[] mittelwerte = messReihe.ermittleMittelwerte();

System.out.println(
    "Durchschnitt von Umdrehungszahl: " + mittelwerte[0]
);
System.out.println(
    "Durchschnitt von Temperatur: " + mittelwerte[1]
);
System.out.println(
    "Durchschnitt von Ladedruck: " + mittelwerte[2]
);
```

Programmieren Sie die Laborstandklasse so, dass alle möglichen Fehler über Exception-Handling dort abgefangen werden. Dazu gehört, dass zu wenig oder falsche Parameter in der Kommandozeile übergeben werden, dass die Datei nicht existiert und dass falsche Daten in der Datei stehen (z.B. Character statt Zahlen).