BlockingQueue

In diesem Arbeitsblatt implementieren Sie eine Order-Processing-Pipeline. Das System besteht aus drei Komponenten. Kunden, Validierung und Processing. Kunden erzeugen Bestellungen und geben Sie auf. Die Validierer überprüfen die Bestellungen bezüglich Konsistenz und werfen ungültige Bestellungen weg. Processors führen dann die validierten Bestellungen aus. Alle drei Komponenten sind als separate Threads implementiert.

Eine zentrale Eigenschaft des Systems ist, dass sich die Komponenten gegenseitig nicht direkt kennen, sondern über generische Datenstrukturen (BlockingQueues) miteinander kommunizieren. Dieses Muster nennt man Producer-Consumer-Pattern. Eine unvollständige Implementierung des Systems finden Sie im LE_Synchronizers.zip Archiv im blockingQueue Package. In der main Methode werden die einzelnen aktiven Komponenten gestartet:

```
public static void main(String[] args) {
    int nCustomers = 10;
    int nValidators = 2;
    int nProcessors = 3;
    /*TODO: Create Queues */

    for(int i = 0; i < nCustomers; i++) {
        new Customer(""+i, /*TODO */).start();
    }

    for(int i = 0; i < nValidators; i++) {
        new OrderValidator(/*TODO */).start();
    }

    for(int i = 0; i < nProcessors; i++) {
        new OrderProcessor(/*TODO */).start();
    }
}
```

Aufgaben:

1. Zeichnen Sie eine Skizze des Systems.
2. Vervollständigen Sie das System, indem Sie die einzelnen Komponenten mittels BlockingQueues verbinden. Lesen Sie dazu die JavaDoc der `j.u.c.BlockingQueue` und suchen Sie sich eine geeignete Implementierung.
3. (Optional) Bauen Sie das System so um, dass ein Kunde maximal drei Sekunden wartet, bis seine Bestellung angenommen wird. Sonst wird die Bestellung verworfen und mit einer neuen weitergemacht.

CountDownLatch

Ein Thread soll warten und erst weiterlaufen, wenn ein anderer Thread seine Arbeit getan hat. Das ist ein übliches Szenario in der Programmierung mit Threads und kann mit der Klasse CountDownLatch realisiert werden.

In diesem Arbeitsblatt implementieren Sie ein Restaurant mit drei Parteien. Sie haben einen Koch, mehrere Kunden und einen Abwascher. Alle drei Klassen sind als Thread implementiert. Der Koch bereitet für alle Gäste in einer grossen Pfanne dasselbe Menü zu.
Zugegeben, es ist ein eigenartiges Restaurant =).

Die Abhängigkeiten der Abläufe sind nun wie folgt:

- Die Gäste können erst essen, wenn der Koch das Essen fertig gekocht hat.
- Der Abwascher darf erst mit dem Abwasch beginnen, wenn die Gäste fertig gegessen haben.

Eine unvollständige Implementierung des Restaurants finden Sie im LE_Synchronizers.zip Archiv im latch Package. Hier sehen Sie die main-Methode, in der die drei Parteien erzeugt und gestartet werden:

```
public static void main(String[] args) {
    int nrGuests = 2;

    /*TODO Create latches. */

    new Cook(/*TODO */).start();

    for(int i = 0; i < nrGuests; i++) {
        new Guest(/*TODO */).start();
     }

     new DishWasher(/*TODO */).start();
}
```

Aufgaben

1. Lassen Sie das Programm laufen und beachten Sie, dass der Abwascher schon abgewaschen hat, bevor der Gast sein Menü vertilgt hat.

2. Lesen Sie die JavaDoc des `j.u.c.CountDownLatch` und verwenden Sie dieses um die Abläufe in eine geordnete Reihenfolge zu bringen. Hinweis: Sie benötigen mehr als ein Latch und müssen diese über die Konstruktoren übergeben.

3. Was passiert wenn ein Gast ins Restaurant eintritt nachdem der Koch bereits fertig gekocht hat?

4. (Optional) Erweitern Sie das Restaurant so, dass mehre Köche Ihre Speise fertig gekocht haben müssen, bevor ein Gast essen kann.