Übungen zu Kapitel 25

Callcenter

25.1 Die Klasse `PriorityBlockingQueue`

Anstatt eine eigene Klasse für die Warteschleifen zu erstellen, kann man auch die im Paket `java.util.concurrent` enthaltene `PriorityBlockingQueue` verwenden. Diese Klasse implementiert selbstredend eine blockierende Prioritätswarteschleife (vgl. [SUN05C]).

Um nun möglichst wenig am bestehenden Coding ändern zu müssen, erstellen wir die folgende Klasse

```
class MyPrioBlockingQueue<T>
   extends PriorityBlockingQueue<T> implements IBlockingQueue<T>
```

und ersetzen im Konstruktor von `Callcenter` die beiden Aufrufe von

`new PrioBlockingQueue()` durch `new MyPrioBlockingQueue()`

Zuvor ist die neue Klasse um dem entsprechenden Konstruktoraufruf zu erweitern.

Dass wir im Buch auf diese Klasse verzichten und die beiden benötigten Methoden `run()` und `take()` selbst implementieren, hat nicht nur didaktische Gründe, wie die Übungsaufgabe 25.5 zeigt.

25.2 Schützen der CallQueue

Ändern Sie das Coding, so dass im CallCreator keine Referenz auf die CallQueue des Callcenters verwendet wird, um `put()` auszuführen, sondern eine Methode **`putCallQ()`** des Callcenters. Im CallCreator ist dann nicht die Queue, sondern das Callcenter zu registrieren.

25.3 Ende der Call-Warteschleife

Um dem Callcenter mitzuteilen, dass - bei einer endlichen Anzahl - alle Calls eingestellt wurden, gibt es verschiedene Möglichkeiten. So kann die Zahl der zu erwartenden Calls dem Callcenter (etwa im Konstruktor) mitgeteilt und die `while()`-Schleife der `run()`-Methode entsprechend angepasst werden. Eine flexiblere Möglichkeit besteht darin, am Ende einen speziell gekennzeichneten Pseudo-Call in die Queue zu stellen und im Callcenter auf ein solches Signal entsprechend zu reagieren.

Erweitern Sie die Klasse `Call` um das Attribut

```
private boolean termCall = false;
```

sowie einen Getter für dieses Attribut und um die Methode

```
public static Call createTermCall()
```

zum Erzeugen eines "Terminator-Calls". In der `run()`-Methode der Klasse `CallCreator` ist dann außerhalb der Schleife eine solche Instanz zu erzeugen und an das `Callcenter` zu schicken.

Im Callcenter wird nach der Entgegennahme eines Calls zunächst geprüft, ob es sich um einen Terminator-Call handelt und im Erfolgsfall die Schleife mit `break` verlassen. Um außerhalb der Schleife auf das Ende aller Worker zu warten, legt man zuvor eine Liste an, bei der jeder Worker registriert wird.

```
Worker w = new Worker( a, c, agentQ );
w.start();
workerList.add( w );
```

Das Warten auf die Worker geschieht dann folgendermaßen (hier ohne Ausnahmebehandlung):

```
for( Worker w : workerList )
   w.join();
```

Vorsicht: Die Verwendung eines Objekts mit "Sondersemantik" (z.B. Terminator-Call) hat weitreichende Auswirkungen auf das bestehende Coding. An vielen Stellen sind Sonderbehandlungen notwendig, etwa beim Sortierkriterium der Warteschleife (der Terminator muss als letztes aus der Schleife genommen werden) bis hin zu den Protokoll-Methoden (er darf nicht als "echter" Call mitgezählt werden).

25.4 Callcenter-Statistik

Erstellen Sie am Ende des Callcenter-Betriebs eine Statistik, die die Anzahl der Agenten, die summarische und deren durchschnittliche Gesprächszeit ausgibt, sowie für die Calls die summarische und durchschnittleiche Gesprächs- und Wartezeit. Hierbei ist es auch interessant, hinsichtlich Premium-, Firmen- und Privatkunden zu differenzieren. Führen Sie anschließend Testläufe mit repräsentativen Randbedingungen (Anzahl der Calls, Anzahl der Agenten, CallParams) durch.

Tipp: Um auch nach dem Betrieb des Callcenters auf alle Calls und Agenten zugreifen zu können, empfiehlt es sich, diese in jeweiligen Collections zu verwalten und `Callcenter` zu erweitern um:

```
private List<Call> callList = new ArrayList<Call>();
private List<Agent> agentList = new ArrayList<Agent>();
```

Beim Erzeugen der Agenten sind diese in `agentList` aufzunehmen,

```
agentList.add( a );  // nach Erzeugen von a
```

die Calls werden nach dem Entnehmen aus der Queue in `callList` registriert:

```
callList.add( c );   // nach Entnahme aus Queue
```

Rufen Sie nach dem `join()` auf alle Worker (vgl. 25.3) eine statische Methode `summary()` auf, die Auswertung von `agentList` und `callList` sowie die Ausgabe der Ergebnisse vornimmt.

25.5 Priorität der Calls nach Wartezeit und Status

Die eingehenden Calls ausschließlich nach dem Kundenstatus zu sortieren (Premium- vor Firmen- vor Privatkunden), ist sehr ungerecht, da Privatkunden bei einer hohen Call-Dichte möglicherweise nie oder erst ganz am Ende bearbeitet werden. (Verifizieren Sie dies durch Wahl geeigneter Parameter).

Erstellen Sie einen Comparator `Status_Waiting_Order`, der die Wartezeiten der Calls vergleicht, für diesen Vergleich die Wartezeit von Premiumkunden 3-fach, die von Firmenkunden doppelt gewichtet.

Wie man sich an einfachen Beispielen klarmachen kann, hängt die Sortierreihenfolge vom Zeitpunkt des Vergleichs ab. Calls von Premiumkunden rücken etwa nach vorne, je später Calls aus der Warteschleife genommen werden. Daher lässt sich in diesem Fall die `PriorityBlockingQueue` aus Übung 25.1 nicht so einfach einsetzen, da dort die Objekte beim Einfügen einsortiert werden. Bei unserer eigenen Implementierung der Warteschleife wird das nächste zu entnehmende Objekt bei der Entnahme bestimmt (vgl. Methode `take()` auf S. 405).

```
class StatusWaitingOrder implements Comparator<Call> {
   public int compare(Call c1, Call c2) {
      long now = System.currentTimeMillis();
      long t1 = now-c1.called;
      if( c1.status==PREMIUM ) t1 *= 3;
      if( c1.status==FIRMEN )  t1 *= 2;
      long t2 = now-c2.called;
      ...
      //Vergleich von t1 und t2
      ...
   }
}
```

Verifizieren Sie die korrekte Implementierung dieses Sortierkriteriums durch geeignete Tests.

Anmerkung: Die Implementierung dieses Kriteriums ist verbesserungswürdig und streng betrachtet nicht ganz korrekt. Zur Bestimmung des Minimums der Collection wird `compare()` mehrmals aufgerufen, wobei jeweils unterschiedliche Vergleichszeitpunkte `now` verwendet werden. Formulieren Sie `now` als privates Attribut der Klasse, das mit einer zusätzlichen Methode `reset()` von außen gesetzt werden kann. In der Methode `take()` ist dann `priority.reset()` zu ergänzen. Damit dies syntaktisch korrekt funktioniert, ist `Comparable` um diese Methode zu erweitern, die man für die anderen Sortierkriterien leer implementiert.