Übungsblatt 2

Ausgabe: 06.05.2008
Abgabe: 14.05.2008, 16:00 Uhr

Aufgabe 1 (8 P)

Die örtliche Kirchturmuhr mit Glockenwerk soll durch eine modernere, Java-programmierte Uhr ersetzt werden. Natürlich soll die neue Uhr den Anwohnern die gewohnte Funktionalität bieten. Da der Kirchturm für Testzwecke nicht zur Verfügung steht, soll zunächst ein Prototyp mit Java erstellt werden. Der Hersteller liefert hierfür ein Entwicklungspaket mit einer multimedialen Repräsentation der Turmuhr (s. zusätzliche Materialien auf der Website).

Die Schnittstelle für die Uhr sieht wie folgt aus:

```
package alpiv.turmuhr;
interface Turmuhr
{
 void setTime(int hours, int mins, int secs);
 void highBell();
 void lowBell();
}
```

Ein Exemplar einer Uhr wird erzeugt mittels: `alpiv.turmuhr.TurmuhrFactory.createTurmuhr()`

Sie sollen nun unter Einsatz von Java-Threads eine Steuerung für diese Turmuhr entwickeln mit den folgenden Eigenschaften:

- Die Anzeige der Uhr wird im Sekundentakt korrekt weitergeschaltet. Zum Testen kann der Takt künstlich beschleunigt werden. Die Standardeinstellung ist 1 ms Realzeit für eine simulierte Sekunde. Zusätzlich kann der Sekundenzeiger auch in einem Takt mehr als 1 Sekunde überspringen. Die für die Abgabe geforderte Einstellung ist ein Sprung von 10 Sekunden.
  Das Weiterschalten der Uhr darf auf keinen Fall von anderen Aktivitäten der Uhr beeinträchtigt werden!
- Zusätzlich soll das Glockenwerk die Uhrzeit angeben. Das Glockenwerk hat zwei Glocken, wobei die hohe Glocke die Viertelstunden und die tiefe Glocke die vollen Stunden schlägt. In der Entwicklungsumgebung werden hierfür die Methoden `highBell()` und `lowBell()` eingesetzt. Die hohe Glocke schlägt einmal bei 15 Minuten, zweimal bei 30 Minuten, dreimal bei 45 Minuten, und viermal zur vollen Stunde. Zur vollen Stunde soll nach den Schlägen der hohen Glocke die tiefe Glocke die Uhrzeit schlagen, einmal um ein Uhr, zweimal um zwei Uhr und so fort bis zwölfmal am Mittag bzw. um Mitternacht.
- Es ist bei dem Anschlagen der Glocken darauf zu achten, dass der Klang der vorher angeschlagenen Glocke verhallt ist (also die Klangausgabe beendet ist), bevor die nächste Glocke angeschlagen wird.
- Über die Standardeingabe kann die Uhr direkt auf beliebige Zeiten eingestellt werden.
- Bei der Implementierung ist **kein** aktives Warten einzusetzen!

Für die Lösung dieser Aufgabe braucht Ihr: distribution.jar

Abgabe: Entwurf- und Programm-Dokumentation sowie relevante kommentierte Code-Fragmente auf Papier, sowie die vollen Quelldateien mit ausführbarem Code als startbares JAR-Archiv per E-Mail. Sollten die Quelldateien (.java) nicht im JAR-Archiv

enthalten sein, bitte diese zusätzlich gezippt per Email schicken.
Achtung: Die Packages beibehalten!

Aufgabe 2 (8 P)

Die zweite Version des LinearBuffer aus 3.1.2 ist so konzipiert, dass Sender und Empfänger überlappend tätig werden können (wenn gleich nicht immer fehlerfrei).

a) (2 P)

Die folgende Variante von `send` birgt eine weitere Fehlerquelle. Zeigen Sie dies durch Angabe eines Beispiels mit entsprechenden Zeitschnitten!

```
public void send(M m) throws Overflow {
  if(count()==size) throw new Overflow();
  synchronized(this) { // sender exclusion
    cell[rear] = m;
    ...
```

b) (2 P)

Die folgende Variante von `recv` ist auch fehlerhaft; warum?

```
public M recv() throws Underflow {
  synchronized(cell) { // receiver exclusion
    if(count()==0) throw new Underflow();
    M m = cell[front];
    front = (front+1)%(size+1);
  }
  return m;
}
```

Kann der Fehler dadurch behoben werden, dass `m` vor dem synchronized vereinbart wird?

c) (4 P)

Erweitern Sie die Klasse um eine zusätzliche Methode urgent, die eine dringliche Nachricht vorn im Puffer ablegt!
(Achtung: die richtige Ausschluss-Synchronisation ist hier etwas kniffliger.)

Aufgabe 3 (4 P)

Betrachten Sie die folgende Implementierung einer speziellen Menge:

```
class ColorSet {
  // zulässige Farben
  final static int RED = 0;
  final static int GREEN = 1;
  final static int BLUE = 2;

  private final boolean[] myColors = new boolean[3];
```

```
  // auf eine Überprüfung der Parameter wird hier
  // sträflicherweise verzichtet
  public void add(int col) {
    myColors[col] = true;
  }
  public void remove(int col) {
    myColors[col] = false;
  }
  public boolean contains(int col) {
    return myColors[col];
  }
}
```

Aufgrund der sehr einfachen und knappen Implementierung der Methoden erscheint eine Ausschlusssynchronisation auf den ersten Blick unnötig. Wie Sie aus der Vorlesung wissen, ist es allerdings möglich, dass unterschiedliche Prozesse, die ein Objekt der Klasse `ColorSet` benutzen, auf unterschiedliche Kopien der Elemente von `myColors` zugreifen.

Begründen Sie, warum die Deklaration von `myColors` als `volatile` diese Problematik *nicht* löst! Beschreiben Sie, welche überraschenden und i. Allg. unerwünschten Effekte bei einer nebenläufigen Benutzung von Objekten dieser Klasse auftreten können.

Hinweise zur Abgabe:

Die Abgabe der Lösungen erfolgt auf zwei Wegen:

- Abgabe auf Papier:

  Bitte schreiben Sie zur jeder Aufgabe eine Dokumentation, in der Sie anhand von *relevanten* Codefragmenten Ihre Implementierung erläutern. Die Codefragmente sollten möglichst kurz gehalten werden und nur der Orientierung für den Leser dienen; entscheidend sind Ihre Erläuterungen, was diese Fragmente machen und wieso Sie sich für diese Art der Implementierung entschieden haben. Die Dokumentation ist *bis Mittwoch 16:00 Uhr in den Tutorenfächern* abzugeben .

- Abgabe per E-Mail:

  Schicken Sie ein ZIP-Archiv an Ihren Tutor, welches den vollständigen Quellcode sowie ausführbare JAR-Dateien für die jeweiligen Aufgabenteile enthält. Der Betreff der E-Mail sollte wie folgt aussehen: "[ALP4] Blatt X - Namen Y".

Zum Bestehen des Übungsblattes müssen 50% der Punkte erreicht werden.