Objektorientierte Programmiertechniken
LVA 185.A01, VU, 3 Ects, 2013/2014 W

# 7. Übungsaufgabe

**Themen:**

Exceptions, nebenläufige Programmierung

**Termine:**

Ausgabe: 27.11.2013
Abgabe: 11.12.2013, 12:00 Uhr

**Abgabeverzeichnis:**

Aufgabe7

**Programmaufruf:**

java Test

**Grundlage:**

Skriptum, Schwerpunkt auf den Abschnitten 4.1 und 4.2

# Aufgabe

**Welche Aufgabe zu lösen ist:**

Das Züchten von Bakterien in einer Petrischale ist sehr teuer und erfordert Hochsicherheitslabors. Daher soll das Wachstum von Bakterien mittels eines Java-Programms simuliert werden. Eine Petrischale ist rechteckig und schachbrettartig in einzelne Felder aufgeteilt. Auf jedem Feld ist eine ganzzahlige Menge an Nährlösung vorhanden. Auf jedem Feld kann sich maximal eine Zelle eines Bakteriums oder eine Zelle eines Penicilinpilzes befinden. Ist auf einem freien benachbarten Feld (jedes Feld, ausser Randfeldern hat 8 Nachbarfelder, waagrecht, senkrecht und diagonal) ausreichend Nährlösung vorhanden und ist dieses freie Feld kein Nachbarfeld von einem mit einem Pilz belegten Feld, dann teilt sich ein Bakterium und eine Kopie des Bakteriums beginnt auf diesem benachbarten Feld zu wachsen. Sind mehrere benachbarte Felder frei und keine Pilznachbarn, wird das Feld mit der

höchsten Menge an Nährlösung ausgewählt. Ist viel Nährlösung vorhanden, teilt sich das Bakterium öfter, ist zu wenig Nährlösung vorhanden, dann stirbt das Bakterium. Auch der Pilz wächst in regelmäßigen Abständen. Der Pilz bevorzugt allerdings Felder, die von einem Bakterium belegt sind, vor freien Feldern mit der höchsten Menge an Nährlösung. Ist das Nachbarfeld des Pilzes mit einem Bakterium belegt, wird dieses abgetötet und der Pilz nimmt dieses Feld ein. Der Pilz vervielfältigt sich seltener als ein Bakterium und stirbt nicht ab, wenn zuwenig Nährlösung vorhanden ist, aber er vervielfältigt sich dann nicht.

Simulieren sie das Bakterienwachstum in einer Petrischale mittels eines nebenläufigen Java-Programms. Stellen sie dabei jede Bakterienzelle und jede Pilzzelle durch einen eigenen Thread dar. Jedes Bakterium teilt sich nach einer gewissen Zeit (wenige Millisekunden (5-50)) abhängig von der Menge an Nährlösung (z.B. 10ms 75%-100%, 20 ms 50%-75%, 30ms 25% bis 50%, Absterben unter 25%). Jeder Pilz vervielfältigt sich nach einer fixen Zeiteinheit (z.B. 50ms). Bei jeder Teilung oder Vervielfältigung wird eine fixe Menge an Nährlösung verbraucht (z.B. 3%). Zählen Sie die Anzahl der Teilungen für jedes Bakterium und jeden Pilz mit, eine kopierte Zelle erhält dabei als Startwert die Teilungsanzahl der Mutterzelle. Simulieren Sie Wartezeiten mittels der Methode *Thread.sleep(n)*. Achtung: *sleep* behält alle Monitore (= Locks); Sie sollten *sleep* daher nicht innerhalb einer *synchronized*-Methode oder -Anweisung aufrufen, wenn während der Wartezeit von anderen Threads aus auf dasselbe Objekt zugegriffen werden soll. Wenn ein Bakterium die maximale Teilungszahl (32) erreicht hat, oder alle Bakterien abgestorben sind, geben Sie von allen lebenden Zellen die Zellinformationen aus und beenden alle Threads. Verwenden Sie *Thread.interrupt()* um einen Thread zu unterbrechen, geben Sie die Zellposition (X- und Y-Koordinate eines Feldes bei zweidimensionaler Petrischalenimplementierung oder fortlaufende Nummer eines Feldes bei eindimensionaler Implementierung einer Petrischale) und die Teilungszahl aus, und beenden Sie den Thread. Eine Petrischale darf in jeder Dimension (waagrecht und senkrecht) nicht mehr als 80 Zellen haben. Geben sie immer, wenn sich eine Pilzgeneration vervielfältigt hat, eine Petrischale zeilenweise am Bilschirm aus. Verwenden Sie den Buchstaben "o" für eine Feld, das mit einem Bakterium belegt ist, den Buchstaben "x" für eine Feld, das mit einem Pilz belegt ist und die Ziffern 0-9 für die restlichen Felder, wobei die Ziffer die Menge an Nährlösung anzeigt (0 für 0%-10%, 1 für 10%-20%, ..., 9 für 90%-100%), zum Beispiel so:

```
9 9 9 x 9 9 x 9
o o 9 x x x x 9
o o o o 9 9 9 9
o o o o o o 9 9
o o o 9 9 9 9 9
```

Die Klasse *Test* soll (nicht interaktiv) Testläufe der Petrischale durchführen und die Ergebnisse in allgemein verständlicher Form in der Standardausgabe darstellen. Bitte achten Sie darauf, dass die Testläufe nach kurzer Zeit

terminieren (maximal 10 Sekunden für alle zusammen). Bitte achten Sie darauf, dass die Testläufe keine Systemlimits wie maximale Anzahl an gleichzeitig aktiven Threads auf dem Abgaberechner (g0) überschreiten. Führen Sie mindestens drei Testläufe mit unterschiedlichen Einstellungen durch:

- Jeder Testlauf soll eine unterschiedliche Menge an Nährlösung in der Petrischale verteilen und Bakterien- und Pilzzellen an unterschiedlichen Feldern positionieren. Für die Teilungs/Vervielfältigungsrate und den Verbrauch an Nährlösung sollen unterschiedliche Werte verwendet werden.
- Stellen Sie die Parameter so ein, dass irgendwann Bakterien und Pilze kollidieren.

Daneben soll die Datei *Test.java* wie gewohnt als Kommentar eine kurze, aber verständliche Beschreibung der Aufteilung der Arbeiten auf die einzelnen Gruppenmitglieder enthalten – wer was gemacht hat.

**Warum die Aufgabe diese Form hat:**

Die Simulation soll die nötige Synchronisation bildlich veranschaulichen und ein Gefühl für eventuell auftretende Sonderfälle geben. Beispielsweise müssen Zellen erkennen, wenn sie die maximale Teilungszahl erreicht haben. Einen speziellen Sonderfall stellt das Simulationsende dar, das (aus Sicht einer Zelle) jederzeit in jedem beliebigen Zustand auftreten kann. Dabei wird auch geübt, nach einer an einer beliebigen Programmstelle aufgetretenen Exception den Objektzustand so weit wie nötig zu rekonstruieren, um ein sinnvolles Ergebnis zurückliefern zu können.

**Was im Hinblick auf die Beurteilung zu beachten ist:**

Die insgesamt 100 für diese Aufgabe erreichbaren Punkte sind folgendermaßen auf die zu erreichenden Ziele aufgeteilt:

| | |
|---|---|
| Synchronisation richtig verwendet, auf Vermeidung von Deadlocks geachted, sinnvolle Synchronisationsobjekte gewählt, kleine Synchronisationsbereiche | 45 Punkte |
| Lösung wie vorgeschrieben und sinnvoll getestet | 20 Punkte |
| Zusicherungen richtig und sinnvoll eingesetzt | 15 Punkte |
| Geforderte Funktionalität vorhanden (so wie in Aufgabenstellung beschrieben) | 15 Punkte |
| Sichtbarkeit auf so kleine Bereiche wie möglich beschränkt | 5 Punkte |

Der Schwerpunkt bei der Beurteilung liegt auf korrekter nebenläufiger Programmerierung und der richtigen Verwendung von Synchronisation sowie dem damit in Zusammenhang stehenden korrekten Umgang mit Exceptions. Punkteabzüge gibt es für

- fehlende oder fehlerhafte Synchronisation,
- zu große Synchronisationsbereiche, durch die sich Threads gegenseitig unnötig behindern (z.B. die gesamte Petrischale als Synchronisationsobjekt),
- nicht richtig abgefangene Exceptions im Zusammenhang mit nebenläufiger Programmerierung,
- Nichttermination von *java Test* innerhalb von 10 Sekunden,
- unnötigen Code und mehrfache Vorkommen gleicher oder ähnlicher Code-Stücke,
- vermeidbare Warnungen des Compilers, die mit Generizität in Zusammenhang stehen,
- Verletzungen des Ersetzbarkeitsprinzips bei Verwendung von Vererbungsbeziehungen,
- mangelhafte Zusicherungen,
- schlecht gewählte Sichtbarkeit,
- unzureichendes Testen,
- und mangelhafte Funktionalität des Programms.

**Wie die Aufgabe zu lösen ist:**

Überlegen Sie sich genau, wie und wo Sie Synchronisation verwenden. Halten Sie die Granularität der Synchronisation möglichst klein, um unnötige Beeinflussungen anderer Threads zu reduzieren (Zellen in unterschiedlichen Feldern der Petrischale sollen sich gleichzeitig teilen/vervielfältigen können). Vermeiden Sie aktives Warten, indem Sie immer *sleep* aufrufen, wenn Sie eine bestimmte Zeit warten müssen. Beachten Sie, dass ein Aufruf von *sleep* innerhalb einer synchronized-Methode oder -Anweisung den entsprechenden Lock nicht freigibt.

Testen Sie Ihre Lösung bitte rechtzeitig auf der g0, da es im Zusammenhang mit Nebenläufigkeit große Unterschiede zwischen den einzelnen Plattformen geben kann. Ein Programm, das auf einem Rechner problemlos funktioniert, kann auf einem anderen Rechner (durch winzige Unterschiede im zeitlichen Ablauf) plötzlich nicht mehr funktionieren. Stellen Sie sicher, dass die maximale Anzahl an Threads und der maximale Speicher nicht überschritten werden. Dazu ist es sinnvoll, dass Sie im Threadkonstruktor explizit die Stackgröße mit einem kleinen Wert angeben (z.B. 16k).

Nebenläufigkeit kann die Komplexität eines Programms gewaltig erhöhen. Achten Sie daher besonders darauf, dass Sie den Programm-Code so klein und einfach wie möglich halten. Jede unnötige Anweisung kann durch zusätzliche Synchronisation (oder auch fehlende Synchronisation) eine versteckte Fehlerquelle darstellen und den Aufwand für die Fehlersuche um vieles stärker beeinflussen als in einem sequentiellen Programm.

**Was im Hinblick auf die Abgabe zu beachten ist:**

Gerade für diese Aufgabe ist es besonders wichtig, dass Sie (abgesehen von geschachtelten Klassen) nicht mehr als eine Klasse in jede Datei geben und auf aussagekräftige Namen achten. Sonst ist es schwierig, sich einen Überblick über Ihre Klassen und Interfaces zu verschaffen. Achten Sie darauf, dass Sie keine Java-Dateien abgeben, die nicht zu Ihrer Lösung gehören (alte Versionen, Reste aus früheren Versuchen, etc.).