Objektorientierte Programmiertechniken
LVA 185.A01, VU, 3 ECTS, 2014/2015 W

7. Übungsaufgabe

Themen:

Exceptions, nebenläufige Programmierung

Termine:

Ausgabe:	26.11.2014
Abgabe:	03.12.2014, 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 Verhalten von Ameisen bei der Futtersuche dient als Vorlage von Ameisenalgorithmen (Ant Colony Optimization). Dabei legen Ameisen eine Pheromonspur (Duftspur), die von weiteren Ameisen verstärkt wird und dann den kürzesten Weg von der Ameisenkolonie zur Futterstelle markiert. Diese Methode soll nun genutzt werden, um in einem Labyrinth den kürzesten Weg vom Eingang zu einem bestimmten Punkt in dem Labyrinth zu finden. Das Labyrinth ist rechteckig und schachbrettartig in einzelne Felder aufgeteilt. Das Labyrinth ist zur Gänze von einer Mauer umgeben. Weiters kann ein Feld rechts und oben von einer Mauer begrenzt sein. Auf einem Feld dürfen sich maximal zwei Ameisen befinden. Eine Ameise kann sich von einem Feld zum nächsten Feld nach rechts, oben, links oder unten bewegen, falls der Weg nicht durch eine Mauer blockiert ist und sich weniger als zwei Ameisen auf dem Zielfeld

befinden. Auf dem Zielfeld wird dann eine Dosis des Pheromons deponiert und addiert sich zu dem Pheromon, das sich bereits auf diesem Feld befindet. Nach jedem Zug wird die Dosis reduziert. Befindet sich die Ameise auf dem Eingangsfeld des Labyrinths oder der Futterstelle, dann wird die Dosis wieder auf den Maximalwert gesetzt. Ameisen können das Labyrinth nicht verlassen.

Simulieren sie das Ameisenlabyrinth mittels eines nebenläufigen Java-Programms. Stellen sie dabei jede Ameise durch einen eigenen Thread dar. Jede Ameise bewegt sich nach einer gewissen Zeit (wenige Millisekunden (5-20) zufallsgesteuert) von einem Feld zum nächsten. Falls alle vier benachbarten Felder blockiert sind, bleibt die Ameise auf dem aktuellen Feld stehen. 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. Implementieren Sie zumindest drei unterschiedliche Fortbewegungsstrategien (z.B. zufallsgesteuert; zuerst nach oben, dann rechts, dann unten, dann links). Setzen Sie in der Anfangsphase der Simulation immer eine neue Ameise auf das Eingangsfeld, sobald sich darauf weniger als zwei Ameisen befinden und die Anzahl der Ameisen kleiner als ein Zehntel der Anzahl der Labyrinthfelder ist. Jede Ameise soll dabei unterschiedlich konfiguriert werden (Fortbewegungsstrategie, Fortbewegungszeit). Experimentieren Sie mit unterschiedlichen Werten für die Reduktion der Pheromondosis abhängig von der Größe des Labyrinths.

Wenn die erste Ameise die maximale Anzahl an Bewegungen (abhängig von der Größe des Labyrinths) erreicht hat, geben Sie von allen Ameisen die Position (X- und Y-Koordinate eines Feldes) und die aktuelle Pheromondosis aus und beenden alle Threads. Verwenden Sie *Thread.interrupt()* um einen Thread zu unterbrechen. Ein Labyrinth darf in jeder Dimension (waagrecht und senkrecht) nicht mehr als 50 Felder haben. Wählen Sie eine Ameise als Leitameise aus. Geben sie immer, wenn diese Leitameise eine Bewegung gemacht hat, das Labyrinth mit Mauern und Pheromonwert des Feldes zeilenweise am Bildschirm aus. Wechseln sie dabei immer eine Zeile mit Informationen und einer Zeile mit waagrechten Mauern ab. Verwenden Sie den Buchstaben "|" für eine senkrechte Mauer, die zwei Buchstaben "--" für eine waagrechte Mauer die Ziffern 0-9 für die Felder, wobei die Ziffer die Menge an Duftstoffen in Prozent des Maximalwertes aller Felder anzeigt (0 für 0%-10%, 1 für 10%-20%, ..., 9 für 90% und mehr), zum Beispiel so:

```
0 0|2 4 6 8 9|9
  --------  --
0|5 4 5 6 7|0 0
    -------
0|6 7 8 6 4 2|0

1|7 8 9|7|2 0|0
  ----      --
3 5 7 9 8|0 0 0
```

Die Klasse *Test* soll (nicht interaktiv) Testläufe von Ameisenlabyrinths 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 ein unterschiedliches Labyrinth mit einem unterschiedlichen Eingangsfeld und einer unterschiedlichen Futterstelle verwenden.
- Stellen Sie die Parameter so ein, dass irgendwann einige Ameisen die Futterstelle erreichen.

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 Ameisen erkennen, wenn sie die maximale Anzahl an Bewegungen erreicht haben. Einen speziellen Sonderfall stellt das Simulationsende dar, das (aus Sicht einer Ameise) 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. darf nicht das gesamte Labyrinth als Synchronisationsobjekt blockiert werden. Ausgenommen davon ist nur die Ausgabe des Labyrinths am Bildschirm.
- 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 (Ameisen auf unterschiedlichen Feldern des Labyrinths sollen sich gleichzeitig weiterbewegen 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.).