OBJEKTORIENTIERTE SOFTWARETECHNIK II

ÜBUNGSAUFGABEN

KONTEN, TRANSAKTIONEN UND BUCHUNGEN (ABGABE: 7. OKTOBER 2016)

Vervollständigen Sie das beiliegende Java-Projekt `transaction`, so dass Buchungen (`Transfer`) und ganze Transaktionen (`Transaction`) korrekt verbucht und die Historien der Konten (i. e. ihre Listen von Konto-Einträgen) richtig geführt werden. Gehen sie dabei stufenweise vor, und zwar so:

1. Verschaffen Sie sich einen Überblick über das Zusammenwirken der Systemteile in den Paketen `view`, `viewModel` und `model`; am besten, indem Sie den Kontrollfluss einiger Operationen und Methoden aus der Klasse `view.View` verfolgen (siehe entsprechende TODOs).
2. Verschaffen Sie sich einen Überblick über die Klassenstrukturen im Paket `model` des Java-Projekts, indem Sie ein entsprechendes UML-Fachklassen-Diagramm zeichnen und eine Liste der verwendeten Entwurfsmuster anfertigen.
3. Schreiben Sie ausreichend Testfälle zur Verbuchung einzelner Buchungen (`Transfer`).
4. Implementieren Sie die Buchung von einzelnen `Transfers`.
5. Schreiben Sie ausreichend Testfälle zur Verbuchung von `Transactions`.
6. Implementieren Sie die Buchung von `Transactions`.
7. Verändern Sie Ihr Projekt so, dass die Salden von Konten niemals das global eingestellte Limit (`model.Account.UniversalAccountLimit`) unterschreiten. Buchungen und Transaktionen, die zur Unterschreitung führen, sollen (1) eine Ausnahme erzeugen und (2) keinerlei Auswirkungen auf sämtliche Salden im Kontosystem haben.
8. Einer Buchung/Transaktion soll man „ansehen“ können, ob ihre Verbuchung bereits versucht wurde und wie oft sie gescheitert ist. Realisieren Sie das mit Hilfe eines State-Patterns!

(Hinweis: Beachten Sie die spezifizierten Kontenlimits erst in der dafür vorgesehenen Ausbaustufe 7!)

Entwickeln Sie einen Puffer mit unbegrenzter Kapazität. Prozesse, die in solch einen Puffer schreiben, müssen also niemals warten. Prozesse, die aus dem Puffer lesen, sollen auf den nächsten Eintrag warten, wenn der Puffer leer ist. (Hinweis: Sie können zur Realisierung von Kritischen Abschnitten und zum Warten die im Skript dargestellten Sperren verwenden! Eine Vorstück für eine Implementierung solcher Puffer finden sie im Observer-Projekt aus der Vorlesung.) Die Elemente, die in einem solchen Puffer gespeichert werden können, sollen eine Schnittstelle „BufferEntry“ implementieren. Zur Steuerung der Leseprozesse können die Schreiber spezielle Kommandos in den Puffer schreiben. Zur Lösung der ersten Aufgaben reicht ein „Stop-Command“, das den Lese-Prozess veranlassen soll, (nach einer jeweils geeigneten Finalisierung) zu terminieren.

MASSENHAFTES RECHNEN (ABGABE: ???)

Entwickeln Sie – auf der Basis der Puffer oben – Prozesse, die die arithmetische Operationen Addition, Subtraktion, Multiplikation und Division (Achtung bei Division durch Null!) auf den ganzen Zahlen realisieren. Die Prozesse erhalten ihre Eingaben aus zwei Eingabepuffern und schreiben ihre Ergebnisse in einen Ausgabepuffer.[1] Durch „Verschalten" von Eingabe- mit Ausgabepuffern kann man größere Arithmetische Ausdrücke „zusammenbauen" und massenhaft komplexere Berechnungen durchführen.

Integrieren Sie in Ihre Lösung eine Möglichkeit zur Verknüpfung mit Konstanten, z. B. `+(*(3,x),5)`.

Integrieren Sie in Ihre Lösung eine Möglichkeit zur Wiederverwendung von Zwischenergebnissen, z. B. `[z --> *(x,y)] *(+(z,5),z)`.

Natürlich darf es bei der Berechnung keine Zyklen geben. Deswegen soll in der letzten Ausbaustufe das gesamte Prozess-Puffer-System aus einem Arithmetischen Ausdruck erzeugt werden. Realisieren Sie dazu die Klassen zum Aufbau von Arithmetischen Ausdrücken (Composite Pattern), eine Prüfung auf Zyklenfreiheit und eine Funktion, die aus einem zyklenfreien Ausdruck ein entsprechendes Prozesssystem erzeugt.

Entwickeln Sie wieder geeignete Testfälle für Ihr System!

NEBENLÄUFIGES SORTIEREN

Jetzt wollen wir uns mit nebenläufigen Varianten von Sortieralgorithmen befassen.

Nebenläufiges Bubble-Sort (Abgabe: ???)

Entwickeln Sie schrittweise eine nebenläufige Version von Bubble-Sort! Der Sortierprozess erhält die Liste der Objekte, die sortiert werden sollen und dazu die Schnittstelle „Comparable" implementieren, über einen Eingabepuffer (Puffer siehe oben!). Die sortierte Liste soll ebenfalls über einen Puffer ausgegeben werden.

Entwickeln Sie für eine erste Lösung einen Prozess, der die Eingabeliste einmal durchläuft, benachbarte falsch sortierte Einträge vertauscht und das Ergebnis in einem Puffer ausgibt. Nachdem solch ein Lauf fertig ist, muss entschieden werden, ob ein weiterer Lauf nötig ist (wurde beim letzten Lauf noch mindestens einmal vertauscht). Ist kein weiterer Lauf nötig, wird die sortierte Liste in den Ausgabepuffer geschrieben. Diese Lösung ist eigentlich noch sequentiell.

Bei der nebenläufigen Lösung, die Sie im zweiten Schritt entwickeln, soll ein Lauf (direkt oder indirekt) den nächsten Lauf sofort starten, wenn er die erste Vertauschung vornimmt.

Nebenläufiges Merge-Sort (Abgabe: ???)

Entwickeln Sie eine möglichst nebenläufige Version des Merge-Sort-Algorithmus, ebenfalls mit Hilfe Ihrer unbeschränkten Puffer!

Nebenläufiges Quick-Sort (Abgabe: ???)

Entwickeln Sie eine möglichst nebenläufige Version des Quick-Sort-Algorithmus, ebenfalls mit Hilfe Ihrer unbeschränkten Puffer!

Nebenläufiges Insertion-Sort (Abgabe: ???)

Entwickeln Sie eine möglichst nebenläufige Version des Insertion-Sort-Algorithmus, ebenfalls mit Hilfe Ihrer unbeschränkten Puffer!

Entwickeln Sie jeweils geeignete Testfälle!

---

[1] Alternativ kann man auch annehmen, dass es für jede Operation nur einen Eingabepuffer gibt, der Paare von Eingaben liefert. Dann benötigt man Prozesse, die zwei Puffer zu einem zusammenführen.

INDETERMINISTISCHE ENDLICHE AUTOMATEN MIT AUSGABE (ABGABE ???)

Realisieren Sie nicht-deterministische endliche Automaten mit Ausgabe (Mealy-Automaten). Dabei sei das Eingabe- und das Ausgabealphabet die Menge der in Java darstellbaren Zeichen (`char`). Ein *Automat* hat genau einen *Anfangszustand* und genau einen *Endzustand*. Anfangs- und Endzustand sind *verschieden*! Erweitern Sie zur Erledigung dieser Aufgabe Ihre Lösung zu nicht-deterministischen Automaten ohne Ausgabe (*ohne leeres Wort*) aus dem letzten Theoriequartal!

Lassen Sie Ihre Mealy-Automaten auch „laufen". Dabei kann man nicht mehr das Modell des Potenzautomaten nutzen. Entwickeln Sie stattdessen eine Klasse `Configuration`, die *einen* möglichen Zustand des Automaten während des Ablaufes darstellt. Eine `Configuration` verwaltet den aktuellen Zustand, die noch nicht verarbeitete Eingabe und die schon erzeugte Ausgabe. Jeder Konfiguration wird ein Thread zugeordnet. (Das heißt, alle möglichen Abläufe in dem Automaten mit einer vorgegebenen Eingabe werden nebenläufig realisiert.) Solch ein Thread endet *erfolgreich*, wenn der Endzustand erreicht wird und die noch nicht verarbeitete Eingabe leer ist. Er endet *nicht erfolgreich*, wenn für den aktuellen Zustand und die nächste Eingabe kein Übergang definiert ist. Er macht einen weiteren Schritt, wenn für den aktuellen Zustand und die nächste Eingabe *genau ein* Übergang definiert ist. Ist *mehr als ein Übergang*, also $n > 1$ Übergänge, definiert, so spaltet der Thread $n - 1$ neue Threads für die Übergänge 2 bis n ab. Er selbst bearbeitet den ersten Übergang weiter. (Jeder abgespaltene Thread muss dabei eine „Kopie" der noch nicht verarbeiteten Eingabe und eine „Kopie" der schon erzeugten Ausgabe erhalten. (Überlegen Sie sich hier geschickte Verfahren zur Verwaltung der Kopien.)

Jetzt können Sie für Ihre Automaten leicht eine Operation `String run(String input)` ergänzen. Diese Operation soll *eine mögliche Ausgabe* (*nicht alle möglichen Ausgaben*) liefern, die der Automat mit der Eingabe `input` erzeugen kann. Solch eine Ausgabe wird durch *eine erfolgreiche* Konfiguration definiert!

Hinweis: Zur Realisierung dieser Aufgabe müssen Sie einen Manager schreiben, der stets weiß, welche Threads noch aktiv sind. Jeder neue Thread muss sich dort registrieren. Jeder Thread, der zum Ende kommt, muss sich auch beim Manager melden und Erfolg oder Misserfolg anzeigen. Der Manager wiederum muss in der Lage sein, laufende Threads abzuschalten, wenn er eine Erfolgsmeldung erhält. (Den Abschaltemechanismus besprechen wir in einer der nächsten Vorlesungen.)

Schreiben Sie ausreichend viele Testfälle!

Zusatzaufgabe: Wenn Sie eine einfache Oberfläche zum manuellen Test dieser Automaten herstellen wollen, erweitern Sie einfach Ihre Lösung (und mein Framework inkl. Oberfläche und Parser) zu Regulären Ausdrücken aus dem letzten Quartal. Sie benötigen dann *Reguläre Ausdrücke mit Ausgabe*. Die sind so aufgebaut:

1. Für je zwei Buchstaben *e* und *a* ist *e*:*a* ein Regulärer Ausdruck. (Elementarer Ausdruck, *e* ist Eingabe und *a* ist Ausgabe)
2. Wenn r_i für $0 < i \leq n$, $n > 0$ Reguläre Ausdrücke sind, dann ist auch $(r_1\ r_2\ \ldots\ r_n)$ Regulärer Ausdruck. (Sequenz. Achtung: Die leere Sequenz ist nicht zugelassen!)
3. Wenn r_i für $0 < i \leq n$, $n \geq 0$ Reguläre Ausdrücke sind, dann ist auch $[r_1\ r_2\ \ldots\ r_n]$ Regulärer Ausdruck. (Auswahl. Achtung: Die leere Auswahl ist zugelassen.)
4. Wenn r Regulärer Ausdruck ist, dann ist auch r+ Regulärer Ausdruck. (Iteration)

PARALLELE BERECHNUNG IN EINER STÜCKLISTE (ABGABE ???)

Eine Stückliste dient der Berechnung von Materiallisten für die enthaltenen Materialien und Produkte, vgl. entsprechende Übungsaufgabe aus dem letzten Quartal. Die Berechnung ist rekursiv: Zur Berechnung der Materialliste für ein Produkt *P* müssen die Materiallisten für *alle* Teile von *P* berechnet werden, die dann anschließend aggregiert („aufaddiert") werden.

Die Berechnung der Materiallisten für die Teile eines Produkts kann nebenläufig geschehen. Die Aggregation „im Produkt“ kann schon beginnen, sobald genügend Materiallisten der Teile zur Verfügung stehen. Sie kann also selbst nebenläufig zur Teileberechnung stattfinden.

Verändern Sie die Lösung für Ihre Stückliste aus dem letzten Quartal so, dass die oben diskutierte Nebenläufigkeit realisiert wird.

DINIERENDE PHILOSOPHEN (ABGABE ???)

Realisieren Sie ein Java-Programm, das geeignet ist, das Problem der Dinierenden Philosophen mit mindestens zwei Philosophen zu simulieren. (Eine ausführliche Beschreibung dieses Problems finden Sie im Skript ab Seite 41.):

1. Philosophen sollen Objekte zu einer Klasse `Philosopher` sein, die die Schnittstelle `Runnable` implementiert. Die `run()`-Methode dieser Klasse soll den „Philosophen-Prozess“ darstellen. Dieser Prozess besteht aus einem steten Wechsel der Zustände „Denken“ und „Essen“ (ggf. später mit weiteren Zwischenzuständen). Dabei soll die Dauer jeder Denk- und Essenszeit zufällig gewählt werden. (Benutzen Sie Objekte zur Klasse `java.util.Random` zur Erzeugung von Zufallszahlen und die Operation `void wait(long timeout)` zum Warten eines Threads für `timeout` viele Millisekunden. Achten sie darauf, dass der Wert für `timeout` stets echt größer als 0 ist!)
2. Essensberechtigungsmarken sind Objekte zu einer Klasse `EBM`, die die Operationen `get()` und `put()` zur Verfügung stellt. Dabei muss folgendes Protokoll eingehalten werden:
    - Als erster Aufruf gelingt nur `get()`.
    - Zwischen je zwei `get()`–Aufrufen ist stets genau ein `put()`–Aufruf und
    - Zwischen je zwei `put()`–Aufrufen ist stets genau ein `get()`–Aufruf.
3. Es gibt ein Objekt im System (z. B. zu einer Singleton-Klasse `PTOMonitor`, das alle Philosophen kennen und an das sie jeden Zustandswechsel melden. Dieses Objekt führt Buch über das Verhalten der gesamten Philosophengesellschaft:
    - Verletzungen der Philosophentischordnung PTO werden erkannt und geeignet gemeldet (z. B. über eine Konsolenausgabe).
    - Die maximale und die durchschnittliche Anzahl gleichzeitig essender Philosophen während einer Simulation wird ermittelt und immer aktuell gehalten.

    In diesem Objekt soll auch die maximale Denkzeit und die maximale Essenszeit für alle Philosophen einstellbar sein.

Realisieren sie folgende zwei Protokolle für die Abläufe pro Philosoph und testen Sie das Verhalten durch Starten von Simulationen mit unterschiedlich vielen Philosophen:

1. Die Philosophen kümmern sich nicht um die Essenberechtigungsmarken, sie wechseln einfach zufällig ihren Zustand.
2. Die Philosophen folgen alle demselben Protokoll: denken – linke EBM nehmen – rechte EBM nehmen – essen – rechte EBM abgeben – linke EBM abgeben – denken – …

ANALYSE VON STELLEN-TRANSITIONEN-NETZEN (ABGABE ???)

1. Implementieren Sie endliche Stellen-Transistionen-Netze (Stellenmenge und Transitionenmenge sind endlich) und zugehörige Systeme (Netz mit Anfangsmarkierung), und zwar so wie sie im Skript definiert sind.

2. Schreiben Sie eine Analyse, die feststellt, ob die Menge der erreichbaren Markierungen für ein System endlich ist oder nicht.
3. Schreiben Sie eine Analyse, die für alle beschränkten Systeme (System mit endlichem Netz und mit endlicher Menge erreichbarer Markierungen) feststellt, ob das System zyklisch ist oder nicht.
4. (Zusatzaufgabe) Kann man auch für ein endliches nicht beschränktes System (System mit endlichem Netz aber mit *nicht endlichem* Zustandsraum) feststellen, ob es zyklisch ist? Wenn ja, wie?