Praktikum Concurrency 2

1. Ampel

In dieser Aufgabe sollen Sie die Funktionsweise einer Ampel und deren Nutzung nachahmen. Benutzen Sie hierzu die Vorgabe `TrafficLightOperation.java`.

Bemerkung zur Vorgabe: Der Einfachheit halber sind alle Klassen in einer Datei zusammengefasst. Natürlich steht es ihnen frei, die Klassen auf mehrere Dateien aufzuteilen.

a) Erweitern Sie zunächst eine Klasse `TrafficLight` mit drei Methoden:
- Eine Methode zum Setzen der Ampel auf „rot“.
- Eine Methode zum Setzen der Ampel auf „grün“.
- Eine Methode mit dem Namen `passby()`. Diese Methode soll das Vorbeifahren eines Fahrzeugs an dieser Ampel nachbilden: Ist die Ampel rot, so wird der aufrufende Thread angehalten, und zwar so lange, bis die Ampel grün wird. Ist die Ampel dagegen grün, so kann der Thread sofort aus der Methode zurückkehren, ohne den Zustand der Ampel zu verändern. Verwenden Sie `wait`, `notify` und `notifyAll` nur an den unbedingt nötigen Stellen!

Bemerkung: die Zwischenphase „gelb“ spielt keine Rolle – Sie können von diesem Zustand abstrahieren!

b) Erweitern Sie nun die Klasse `Car` (abgeleitet von Thread). Im Konstruktor wird eine Referenz auf ein Feld von Ampeln übergeben. Diese Referenz wird in einem entsprechenden Attribut der Klasse `Car` gespeichert. In der `run`-Methode werden alle Ampeln dieses Feldes passiert, und zwar in einer Endlosschleife (d.h. nach dem Passieren der letzten Ampel des Feldes wird wieder die erste Ampel im Feld passiert).
Natürlich darf das Auto erst dann eine Ampel passieren, wenn diese auf grün ist! Für die Simulation der Zeitspanne fürs Passieren können Sie die folgende Anweisung verwenden:

```
sleep((int)(Math.random() * 500));
```

Beantworten Sie entweder c1 oder c2 (nicht beide):

c1) Falls Sie bei der Implementierung der Klasse `TrafficLight` die Methode `notifyAll` benutzt haben: Hätten Sie statt `notifyAll` auch die Methode `notify` verwenden können, oder haben Sie `notifyAll` unbedingt gebraucht?

Begründen Sie Ihre Antwort!

ich hätte auch mit mehreren notify die Anzahl sich bewegenden Autos setzen können. notifyAll ist etwas codeSchonender.

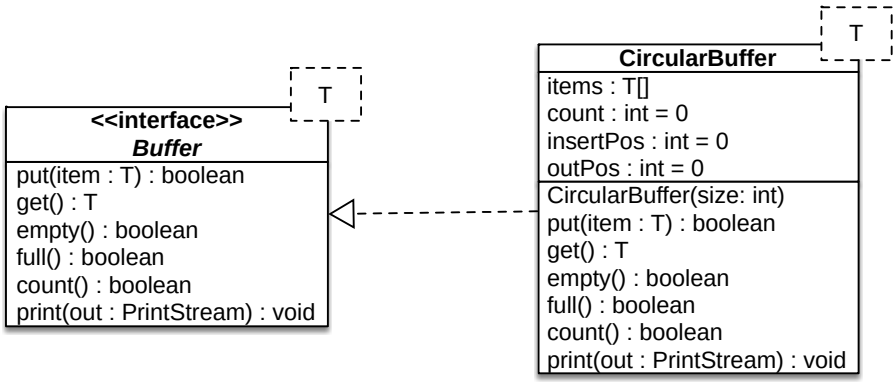
c2) Falls Sie bei der Implementierung der Klasse Ampel die Methode `notify` benutzt haben: Begründen Sie, warum Sie `notifyAll` nicht unbedingt gebraucht haben!

d) Testen Sie das Programm `TrafficLightOperation.java`. Die vorgegebene Klasse `TrafficLightOperation` implementiert eine primitive Simulation von Autos, welche die Ampeln passieren. Studieren Sie den Code dieser Klasse und überprüfen Sie, ob die erzeugte Ausgabe sinnvoll ist.

2. Producer-Consumer Problem

a) Producer- und Consumer-Thread

Vorgegeben ist eine Implementation eines zirkulären Buffers (`CircularBuffer.java, Buffer.java`) mit folgendem Design:



Als erstes soll ein "Producer" und ein "Consumer" implementiert werden. Unten ist das Gerüst für beide abgebildet (`CircBufferTest.java`):

```
class Producer extends Thread {
    public Producer(String name, Buffer buffer, int prodTime) {
            ...
    }
    public void run() {
            ...
    }
}

class Consumer extends Thread
{
    public Consumer(String name, Buffer buffer, int consTime) {
            ...
    }
    public void run() {
            ...
    }
}
```

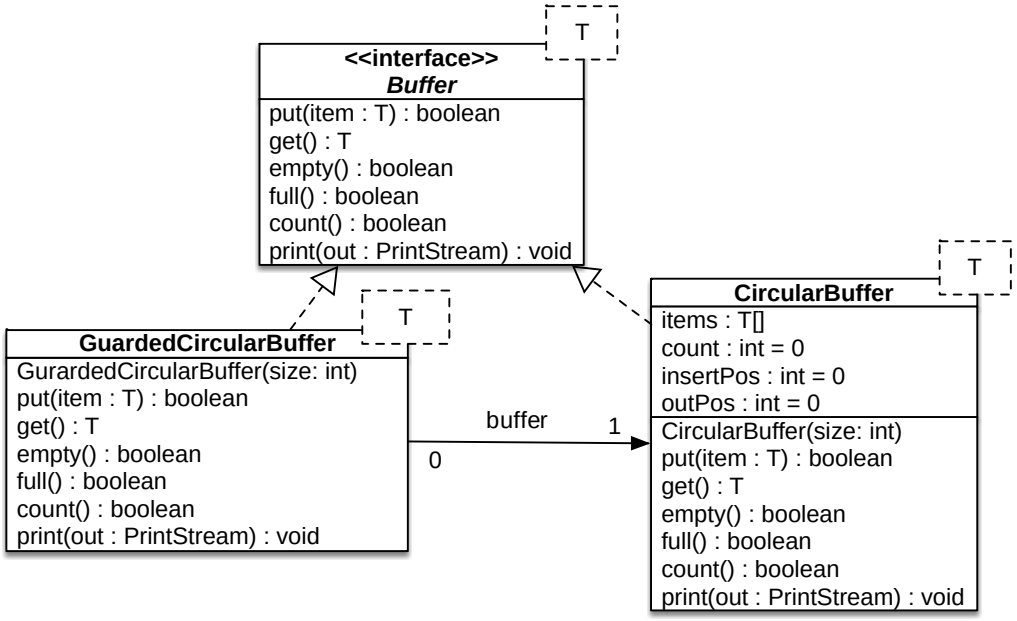
Der Producer soll Daten in den Buffer einfüllen, und der Consumer soll Daten herausholen. Auf den Buffer soll nur über das Interface zugegriffen werden. Das Zeitintervall, in dem ein Producer Daten einfüllen kann, ist mit `sleep((int)(Math.random()*prodTime))` zu definieren. Die Zeit fürs konsumieren können Sie entsprechend mit `sleep((int)(Math.random() * consTime))` bestimmen.

Für Producer und Consumer wurde bereits ein Testprogramm (`class CircBufferTest`) geschrieben. Testen Sie damit ihre Consumer- und Producer- Klassen. Versuchen sie den generierten Output auf der Console richtig zu interpretieren! Spielen sie mit den Zeitintervallbereichen von Producer (`maxProdTime`) und Consumer (`maxConsTime`) und ziehen sie Schlüsse. Erstellen sie über die Modifikation von `prodCount` und `consCount` mehrere Producer bzw. Consumer.

Hinweis: Generieren sie in den selber implementierten Klassen keine eigene Ausgabe. Ändern sie den bestehenden Code nicht. Es stehen zwei Ausgabefunktionen zur Auswahl: print() und print2().

b) Thread-Safe Circular Buffer

In der vorangehenden Übung griffen mehrere Threads auf den gleichen Buffer zu. Die Klasse `CircularBuffer` ist aber nicht thread-safe. Was wir gemacht haben, ist daher nicht tragbar. Deshalb soll jetzt eine Wrapper Klasse geschrieben werden, welche die `CircularBuffer`-Klasse "thread-safe" macht. Das führt zu folgendem Design:



Aufrufe von `put` blockieren, solange der Puffer voll ist, d.h., bis also mindestens ein leeres Puffer-Element vorhanden ist. Analog dazu blockieren Aufrufe von `get`, solange der Puffer leer ist, d.h, bis also mindestens ein Element im Puffer vorhanden ist.

Tipp: Verwenden Sie den Java Monitor des `GuardedCircularBuffer`-Objektes!
Wenn die Klasse fertig implementiert ist, soll sie in der `CircBufferTest` Klasse verwendet werden.

Beantworten Sie entweder b1 oder b2 (nicht beide):

b1) Falls Sie bei der Implementierung der Klasse `GuardedCircularBuffer` die Methode `notifyAll` benutzt haben: Hätten Sie statt `notifyAll` auch die Methode `notify` verwenden können oder haben Sie `notifyAll` unbedingt gebraucht? Begründen Sie Ihre Antwort!

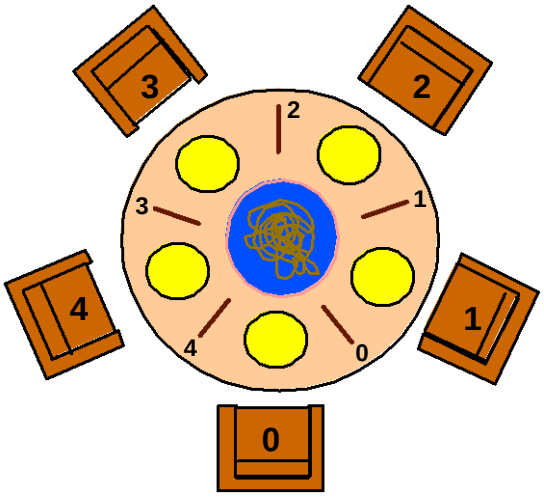
b2) Falls Sie bei der Implementierung der Klasse `GuardedCircularBuffer` die Methode `notify` benutzt haben: Begründen Sie, warum Sie `notifyAll` nicht unbedingt gebraucht haben. Begründen Sie Ihre Antwort!

Führen Sie das Endresultat dem Betreuer vor.

3. The Dining Philosophers

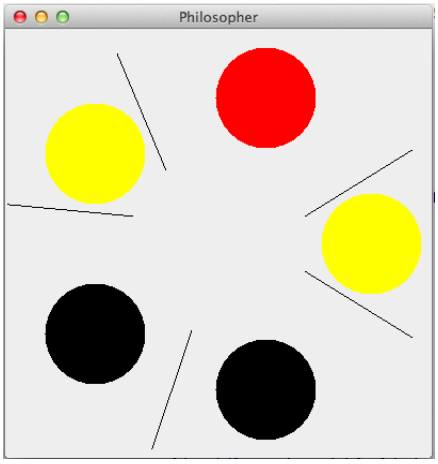
Beschreibung des Philosophen-Problems:

Fünf Philosophen sitzen an einem Tisch mit einer Schüssel, die immer genügend Spaghetti enthält. Ein Philosoph ist entweder am Denken oder am Essen. Um zu essen braucht er zwei Gabeln. Es hat aber nur fünf Gabeln. Ein Philosoph kann zum Essen nur die neben ihm liegenden Gabeln gebrauchen. Aus diesen Gründen muss ein Philosoph warten und hungern, solange einer seiner Nachbarn am Essen ist.



Unten abgebildet ist die Ausgabe des Systems, das wir in dieser Aufgabe verwenden. Die schwarzen Kreise stellen denkende Philosophen dar, die gelben essende und die roten hungernde. Bitte beachten Sie, dass eine Gabel, die im Besitz eines Philosophen ist, zu dessen Teller hinverschoben dargestellt ist.

die Dauer des Denkens der Philoso-phen verklei-nern.



a) Analysieren Sie die bestehende Lösung (`Philosopher-Gui.java`, `PhilosopherTable.java`), die bekanntlich nicht Deadlock-frei ist. Erzeugen Sie ein Projekt in ihrer IDE und starten Sie anschliessend die Applikation. Nach einiger Zeit geraten die Philosophen in eine Deadlock-Situation und verhungern. Überlegen Sie sich, wo im Code der Deadlock entsteht und versuchen Sie, dessen Auftreten schneller herbeizuführen.

b) Passen Sie die bestehende Lösung so an, dass keine Deadlocks mehr möglich sind. Sie müssen im Wesentlichen den `ForkManager` so anpassen, dass sich Gabelpaare in einer "atomaren" Operation belegen bzw. freigegeben lassen. Die Ausgabe müssen Sie nicht anpassen. Die Änderungen an der Klasse `Philosoph` sind minimal, da sie nur den Methodenaufruf für die Freigabe bzw. Belegung der Gabeln ändern müssen.
Verwenden Sie für die Synchronisation Locks und Conditions!
Testen Sie ihre Lösung auch auf Deadlock-Freiheit!

c) (optional) In der Vorlesung haben Sie mehrere Lösungsansätze kennen gelernt. Implementieren Sie einmal einen alternativen Ansatz, wie zum Beispiel die Nummerierung der Betriebsmittel.

d) (optional) Studieren Sie auch die untenstehende korrekte Lösung aus dem Buch von Tanenbaum mit Semaphoren. Lösen Sie die Aufgabe mit Semaphoren!

Anhang

(Auszug aus dem Buch von A. Tanenbaum: Moderne Betriebssysteme, 2. Auflage)

Abbildung 2.33 Eine Lösung für das Problem der speisenden Philosophen.

```
#define N                5                /* Anzahl der Philosophen */
#define LEFT             (i+N–1)%N        /* Linker Nachbar von Philosoph i*/
#define RIGHT            (i+1)%N          /* Rechter Nachbar von Philosoph i*/
#define THINKING         0                /* Philosoph denkt nach */
#define HUNGRY           1                /* Philosoph versucht 2 Gabeln zu bekommen */
#define EATING           2                /* Philosoph isst */
typedef int semaphore;                    /* Semaphoren sind Integer */
int state[N];                             /* Feld mit dem Status jedes Philosophen */
semaphore mutex = 1;                      /* Wechselseitiger Ausschluß im kritischen Bereich */
semaphore s[N];                           /* ein Semaphor pro Philosoph */

void philosopher (int i)                  /* i: Nummer des Philosophen, von 0 bis N–1 */
{
        while (TRUE) {                    /* Endlosschleife */
                think();                  /* Philosoph denkt nach */
                take_forks(i);            /* Nimm 2 Gabeln oder blockiere */
                eat();                    /* lecker, Spagetti! */
                put_forks(i);             /* lege beide Gabeln zurück auf den Tisch */
        }
}

void take_forks (int i)                   /* i: Nummer des Philosophen, von 0 bis N–1 */
{
        down(&mutex);                     /* Eintritt in den kritischen Bereich */
        state[i] = HUNGRY;                /* Merken, dass Philosoph i hungrig ist */
        test(i);                          /* versuche 2 Gabeln zu bekommen */
        up(&mutex);                       /* verlassen des kritischen Bereichs */
        down(&s[i]);                      /* blockiere, wenn Gabeln nicht erhalten */
}

void put_forks (i)                        /* i: Nummer des Philosophen, von 0 bis N–1 */
{
        down(&mutex);                     /* Eintritt in den kritischen Bereich */
        state[i] = THINKING;              /* Philosoph ist mit dem Essen fertig */
        test(LEFT);                       /* kann der linke Nachbar essen? */
        test(RIGHT);                      /* kann der rechte Nachbar essen? */
        up(&mutex);                       /* verlassen des kritischen Bereichs */
}

void test (i)                             /* i: Nummer des Philosophen, von 0 bis N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```