SoSe 2017

Nichtsequentielle und verteilte Programmierung

4. Übungsblatt

1. Aufgabe (4 P.)

Verändern Sie die 7. Aufgabe des 1. Übungsblatts, so dass die Glücksspieler gegen eine zentrale Casino-Bank gleichzeitig spielen.

1. Alle Spieler starten mit 5 Dollar und nach jedem Spiel verlieren oder gewinnen sie einen Dollar. Die Spieler hören auf zu spielen, wenn diese kein Geld mehr haben oder sich ihr Budget verdoppelt hat.
2. Die Casino-Bank startet mit 20 Dollar und akzeptiert Spieler, solange Geld in der Bank-Kasse vorhanden ist.
3. Die Simulation wird beendet, wenn alle Spieler aufgehört haben zu spielen oder wenn die Casino-Bank kein Geld mehr hat.

Welche Invariante bezüglich das Geldes soll im System gelten, wenn die Synchronisation während der gesamten Simulation korrekt gelaufen ist?

Protokollieren Sie der Spielverlauf, sodass diese Invariante kontrolliert werden kann.

2. Aufgabe (4 P.)

Nehmen wir an, in einer private Kita gelten strenge Regeln bezüglich der minimalen Anzahl von Erzieherinnen, die für die Pflege der Babies tagsüber notwendig sind.

1) Pro Erzieherin dürfen maximal 5 Babies aufgenommen werden.
2) Neue Babies werden nicht akzeptiert, wenn nicht genug Erzieherinnen bereits eingetroffen sind.
3) Die Erzieherinnen dürfen nicht einfach weggehen und eine Überzahl von Babies den restlichen Erzieherinnen überlassen.

Definieren Sie eine **Kita**-Klasse in Java, indem Sie mit Hilfe von Java-Semaphoren die Aufnahme und das Abholen der Babies sowie das Eintreffen und die Entlastung der Erzieherinnen synchronisieren.

Sowohl die Erzieherinnen als auch die Eltern mit Babies sollen mit Hilfe von Threads simuliert werden.

Testen Sie Ihre Klasse, indem Sie den Personenverkehr simulieren. Sie sollen dabei wie im 1.Übungsblatt die **Nap** Klasse dafür verwenden.

3. Aufgabe (22 P.)

a) (0 P.) Importieren Sie das Framework „*Across the Bridge*" in einer IDE Ihrer Wahl. Die Frameworks, sowie dazugehörigen Anleitungen, finden Sie auf der Veranstaltungsseite. Sollten Sie Schwierigkeiten mit dem Import haben, benutzen Sie die Eclipse IDE auf einem der Pool-Rechner der Keller-Räume. Dort wurden die Frameworks getestet.

b) (6 P.) Entwickeln Sie drei BridgeControl Klassen, die lediglich ein Auto gleichzeitig auf die Brücke lassen. Bedienen Sie sich dazu verschiedener Mechanismen:

- einem Semaphor,
- einem ReentrantLock,
- dem Monitorkonzept,

c) (5 P.) Entwickeln Sie ein BridgeControl, das abwechselnd je 5 Fahrzeuge aus Westen und Osten über die Brücke passieren lässt. Vermeiden Sie das aktive Warten.

d) (6 P.) Entwickeln Sie ein faires BridgeControl, das unter Verwendung der Default Settings (FPS darf erhöht werden) nach 15.000 Frames folgende Scores erreicht:

Efficiency: 12.000 +
Fairness: 6.500 +

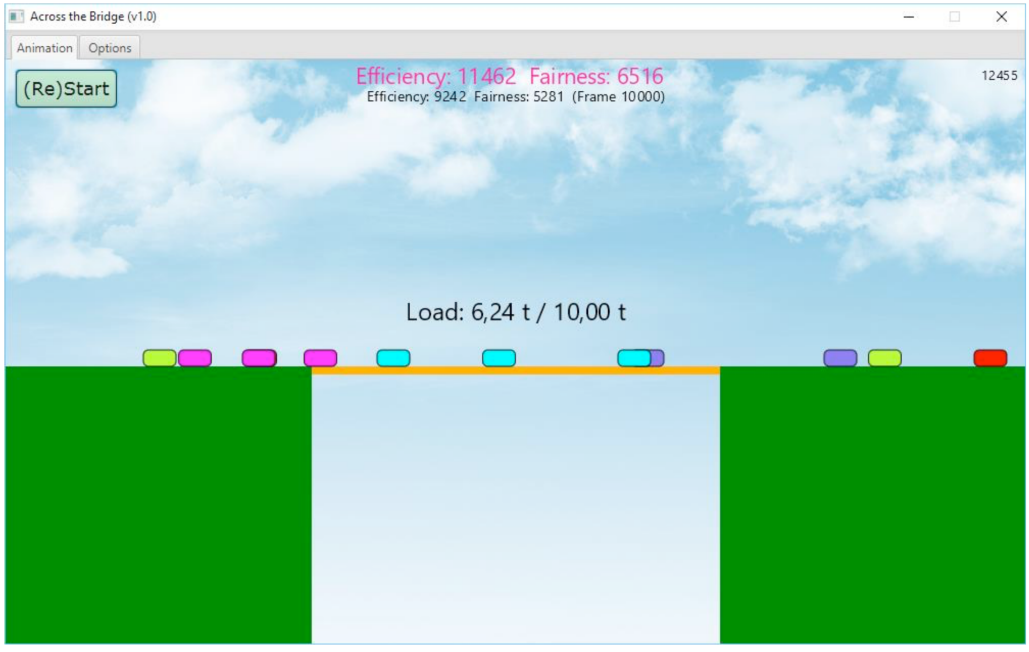
Vermeiden Sie das Aktive Warten. Erläutern Sie, warum Ihre Lösung fair ist.

e) (5 P.) Erweitern Sie ihr BridgeControl aus Aufgabe **d)**, so dass sich keine Lowrider auf der Brücke begegnen können. Mehrere Lowrider aus derselben Richtung sollen die Brücke zeitgleich passieren können. Leichte Einbußen bei der Fairness-Score können hingenommen werden.

f) (2 bis 4 Bonuspunkte) Implementieren Sie ein BridgeControl mit einem („innovativen") Verhalten Ihrer Wahl.

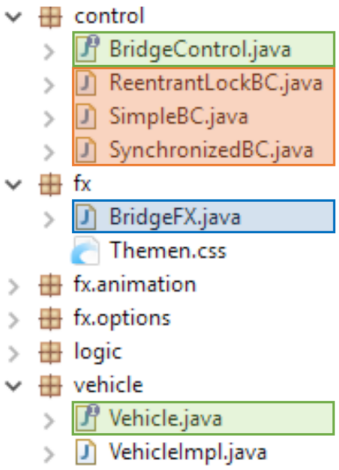
Szenario

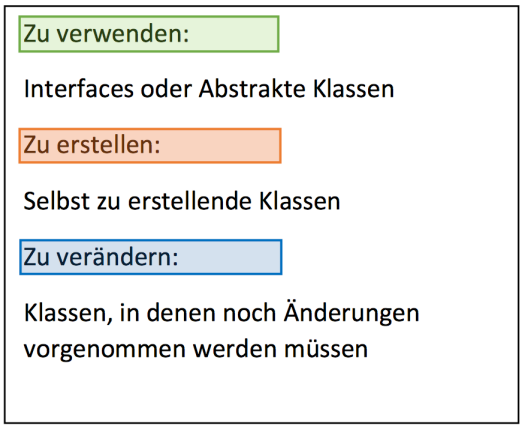
Mehrere Autos möchten eine Brücke passieren. Die Brücke besitzt jedoch nur eine begrenzte Tragkraft. Um eine Überlastung zu vermeiden, müssen Mechanismen entwickelt werden, die den Zugriff auf die Brücke kontrollieren.



Vervollständigung

Zur Vervollständigung dieses Frameworks müssen Klassen erstellt werden, die das BridgeControl Interface implementieren.





Die Interface BridgeControl verfügt über folgende 3 Methoden:

```
void init (Double maxLoad);

    /* Einmalig aufgerufen unmittelbar nach dem Instanzieren. Diesem Aufruf kann die
    maximale Tragkraft der Brücke entnommen werden. */

void requestCrossing (Vehicle v);

        /* Einmalig aufgerufen von jedem Fahrzeug, welches die Brücke passieren möchte.
        Dieser Aufruf muss blockieren, falls dem Fahrzeug derzeit nicht erlaubt werden kann,
        die Brücke zu passieren. */

void leaveBridge (Vehicle v);

        /* Einmalig aufgerufen von jedem Fahrzeug, welches die Brücke verlässt. Der Aufruf
        dient lediglich dazu, das BridgeControl über das Verlassen zu informieren. Fahrzeuge
        können durch das Blockieren dieser Methode selbst nicht blockiert werden. */
```

Beim Aufruf der Methoden *requestCrossing* und *leaveBridge* wird jeweils eine Referenz auf das aufrufende Fahrzeug mitgegeben, auf welche sich folgende Aufrufe ausführen lassen:

```
Double getWeight()              /* Gewicht der Autos */

VOrigin getOrigin()             /* Herkunft des Autos (VOrigin.EAST oder VOrigin.WEST) */

boolean isLowrider()            /* Wahr, wenn es sich bei dem Fahrzeug um einen
                                    Lowrider handelt  */

long getVehicleId();            /* Eindeutige ID des Fahrzeugs
                                   (Sollten nur zum Debuggen genutzt werden) */

void setMessage(String msg);    /* Setzt einen neuen Info-Text für den Info-Dialog fest
                                      (Sollten nur zum Debuggen genutzt werden) */
```

Um eine erstellte BridgeControl Implementierung in der GUI auswählen zu können, muss ihr Name im Array mutexClasses (zu finden in der Klasse BridgeFX) hinterlegt werden. Im folgenden Beispiel wurden die zwei BridgeControl Klassen SmartBC und SimpleBC aus dem Package control hinterlegt.

```
final static private String[] mutexClasses = { "control.SmartBC", "control.SimpleBC" };
```

Regeln:

- Das Gewicht aller Fahrzeuge auf der Brücke darf die maximale Tragkraft nicht überschreiten.
- Es gibt spezielle Fahrzeuge, Lowrider, die auf der Brücke nicht aufeinander zufahren dürfen. (Mehrere Lowrider aus derselben Richtung stellen kein Problem dar.)
- Die Effizienz-Score betrachtet nach jedem Durchlauf, wieviel Prozent der Tragkraft genutzt bzw. bereits Fahrzeuge zugewiesen wurden und addiert diesen Wert zur Score.
- Die Fairness-Score betrachtet (einmal alle 100 Durchläufe) die Wartezeit der letzten 10 Fahrzeuge, die die Brücke fertig passiert haben. Diese wird ermittelt und anschließend durch die Wartezeit des Fahrzeugs, welches am längsten warten musste, geteilt. Dieser Wert wird zur Score addiert.

Lösungshinweise:

Bei den Autos handelt es sich um tatsächliche Java-Threads. Daher ist es wichtig, sicher zu stellen, dass alle Operationen innerhalb der BridgeControl tatsächlich thread-sicher sind.

Busy-waiting sollte vermieden werden, da dies bei entsprechend hoher Anzahl von Threads/ Fahrzeugen sehr zu Lasten der Performance gehen kann. (z.B. while (kein Platz frei) {})

Jedes Fahrzeug befindet sich stets in einem der folgenden Zustände:

Zustand	Das Fahrzeug ...
ARRIVING	... kommt gerade an und fährt zum Wartepunkt.
WAITING	... ist am Wartepunkt angekommen und wartet die Erlaubnis von BridgeControl ab.
DRIVENOFF	... fährt zur Brücke.
ONBRIDGE	... ist auf der Brücke.
PASSED	... hat die Brücke überquert.
DRIVENAWAY	... Fahrzeug ist davon gefahren und kann gelöscht werden.
SHOCKED	... ist erstarrt, da die Brücke zusammengebrochen ist.
FALLING	... das Fahrzeug ist von der Brücke gefallen.

Durch das Klicken auf ein Fahrzeug öffnet sich ein Info-Dialog. Diesem können die ID, der aktuelle Zustand und der optional gesetzte Info-Text des Fahrzeugs entnommen werden.

Alle Lösungen müssen Thread-Sicherheit garantieren.

Ihre digitale Abgabe muss lediglich die von Ihnen erstellten *BridgeControl* Klasse Implementierungen als .java-Datei enthalten.

Allgemeine Wichtige Hinweise zur Übungsabgabe:

1. Es muss der Quellcode hochgeladen werden (.java-Dateien), nicht der kompilierte Bytecode (.class-Dateien).

2. Zusätzlich zur Online-Abgabe muss der selbstprogrammierten Quellcode ausgedruckt werden. Der ausgedruckte Quellcode muss lesbar sein, insbesondere ist auf automatische Zeilenumbrüche zu achten.

3. Für die gute Verständlichkeit des Quellcodes sind sinnvolle Bezeichnernamen zu wählen und der Code ausreichend zu kommentieren.

4. Das Programm muss Testläufe enthalten, die die Aufgabe vollständig abdecken. Das heißt, zum Testen sollen keine Änderungen am Code durch die Tutorin/den Tutor mehr notwendig sein. Die Testläufe müssen in einer eigenen Klasse Main enthalten sein.

5. Das Programm muss mit Java 8 kompatibel sein.

6. Alle Übungsaufgaben, die nicht Programmieraufgaben sind, müssen ebenfalls digital und nicht handschriftlich sowohl online als auch ausgedruckt zu dem im KVV angegebenen Datum (in der Regel Dienstag um 11:55) abgegeben werden. Zu spät abgegebene Lösungen werden mit 0 Punkten bewertet.