Lösungen zu den Übungen Concurrency – Cooperation

1. Concurrency 3 — Thread Synchronisation

1.1. Konto-Übertrag [PU]

a. Was stellen Sie fest, wenn Sie die Simulation laufen lassen? Erklären Sie wie die Abweichungen zustande kommen.

> Bei den Transaktionen passieren sog. *lost updates*. Eine Transaktion ist nicht atomar und besteht aus mehreren Schritten (Wert aus Speicher lesen, verändern, Wert in Speicher schreiben). Durch die gleichzeitige Operation auf den Konten aus mehreren Threads können einzelne dieser Schritte ignoriert werden und verloren gehen. Beide Threads Lesen den gleichen Wert, aktualisieren diesen gleichzeitig, schreiben das neue Resultat (der letzte gewinnt). Am Schluss ist die Geldsumme über alle drei Konten oft grösser oder kleiner als am Anfang.

b. Im Unterricht haben Sie gelernt, dass kritische Bereiche Ihres Codes durch Mutual-Exclusion geschützt werden sollen. Wie macht man das in Java?

   Versuchen Sie mittels von Mutual-Exclusion sicherzustellen, dass keine Abweichungen entstehen.

   - Reicht es die kritischen Bereiche in Account zu sichern?
   - Welches sind die kritischen Bereiche?

Siehe Klasse: `Account`

Monitor Objekte müssen so gewählt werden, dass sie die 'geteilten' Ressourcen schützen. Oft ist deshalb die geteilte Ressource selber das Monitor Objekt.
In diesem Fall wären es die `Account`-Objekte auf welche die von verschiedenen Threads zugegriffen wird.
Den Thread selber als Monitor zu verwenden macht wenig Sinn, da dann jeder Thread seinen eigenen Monitor besitzen würde.

Der Monitor von Thread-Objekten wird intern für `Thread.join()` verwendet. Beim Beenden eines Threads werden mit `notifyAll()` alle in einem `join()` wartenden Threads informiert. Deshalb sollte auf Thread-Instanzen niemals `wait()`, `notify()` oder `notifyAll()` verwendet werden. Siehe JavaDoc von Thread.join().

Kritische Bereiche sind diejenigen, in welchen auf eine geteilte Variable zugegriffen wird. Das kann sowohl schreibend, wie auch lesend erfolgen. Auch lesende Zugriffe sollten geschützt werden, da sonst nicht sichergestellt werden kann, dass der Wert während des Lesens nicht von einem anderen Thread verändert wird.
In `Account` sollte deshalb sowohl die Methode `setBalance(int amount)` wie auch `getBalance()` als `synchronized` deklariert werden.

Untersuchen Sie mehrere Varianten von Locks (Lock auf Methode oder Block, Lock auf Instanz oder Klasse).

Ihre Implementierung muss noch nebenläufige Transaktionen erlauben, d.h. wenn Sie zu stark synchronisieren, werden alle Transaktionen in Serie ausgeführt und Threads ergeben keinen Sinn mehr.

Stellen Sie für sich folgende Fragen:

- Welches ist das Monitor-Objekt?
- Braucht es eventuell das Lock von mehr als einem Monitor, damit eine Transaktion ungestört ablaufen kann?

Siehe Klasse: `AccountTransferTask`

Auch in `AccountTransferTask` ist immer noch `Account` das Monitor-Objekt, da dort die Daten effektiv verändert werden. Da in `transfer()` jedoch gleichzeitig mit zwei Konten gearbeitet wird, sollte sichergestellt werden, dass nur ein Thread auf exakt die beiden Konten Zugriff hat, damit die Überprüfung und Abbuchung von einem Konto und der Übertrag auf das Andere als eine atomare Funktion erfolgt. Das heißt wir erhalten einen neuen *kritischen Bereich* den wir mit synchronized schützen müssen. Diesmal in dem der Monitor von `fromAccount` und `toAccount` im Voraus schon akquiriert und somit sicherstellt wird, dass `transferAmmount` von beiden Konten nicht mehr blockieren kann.
In Java kann das mit einer geschachtelten Synchronisation erfolgen:

```
synchronized(fromAccount) {
    synchronized(toAccount) {
        if (fromAccount.getBalance() >= amount) {
            fromAccount.transferAmount(-amount);
            toAccount.transferAmount(amount);
        }
    }
}
```

Warum kann man nicht einfach die Methode `transfer()` synchronized deklarieren? Damit würde die `AccountTransferTask`-Instanz selber als drittes Monitor-Objekt verwendet. Damit werden nur die Threads synchronisiert, welche das gleiche Task-Objekt verwenden. Da drei Task-Objekte erstellt werden, könnten zum Beispiel je ein Thread für `task2` und `task3` gleichzeitig auf `account2` zugreifen. Würde für jede Transaktion sogar eine eigene `AccountTransferTask`-Instanz erzeugt, hätte das synchronized auf `transfer()` überhaupt keinen Effekt mehr, da dann jeder Thread seinen eigenen Monitor besitzt.

c. Wenn Sie es geschafft haben die Transaktion thread-safe zu implementieren, ersetzen Sie in `AccountTransferSimulation` die folgende Zeile:
`AccountTransferTask task1 = new AccountTransferTask(account3, account1, 2);`
durch
`AccountTransferTask task1 = new AccountTransferTask(account1, account3, 2);`
und starten Sie das Programm noch einmal. Was stellen Sie fest? (evtl. müssen Sie es mehrfach versuchen, damit der Effekt auftritt).
Was könnte die Ursache sein und wie können Sie es beheben?

Falls Sie die Frage noch nicht beantworten können, kommen sie nach der Vorlesung "Concurrency 4" nochmals auf diese Aufgabe zurück und versuchen Sie sie dann zu lösen.

Durch die Umstellung kann es zu einem zirkulären Deadlock kommen, da sich die Threads gegenseitig blockieren (jeder hat ein Konto bereits gelockt und wartet auf ein anderes).

Die einfachste Lösung ist es die Konten immer in einer übergeordneten festen Reihenfolge zu akquirieren. In diesem Fall zum Beispiel immer zuerst die tiefere Kontonummer. Dadurch kann es keine zirkulären Abhängigkeiten mehr geben.

Lösung: `AccountTransferTask.transferDLfree()`

1.2. Traffic Light [PU]

In dieser Aufgabe sollen Sie die Funktionsweise einer Ampel und deren Nutzung nachahmen. Benutzen Sie hierzu die Vorgabe im Modul `TrafficLight`.

a. Ergänzen Sie zunächst in der Klasse `TrafficLight` 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!

Die Zwischenphase „gelb“ spielt keine Rolle – Sie können diesem Zustand ignorieren!

Lösung siehe: `ch.zhaw.prog2.trafficlight.TrafficLight`

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 für das Passieren des Signals sollten Sie folgende Anweisung verwenden: `sleep((int)(Math.random() * 500));`

Lösung Siehe: `ch.zhaw.prog2.trafficlight.Car`

Beantworten Sie entweder (c) oder (d) (nicht beide):

c. 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!

Mit `notifyAll()` erhalten alle Autos die Gelegenheit die Ampel zu überqueren. Natürlich betreten sie den Monitor (`TrafficLight`) immer noch einzeln und können, falls die Zeitspanne nicht, reicht trotzdem stehen bleiben. Dann erhalten Sie bei der nächsten "Grünen Welle" erneut eine Notifikation und können es nochmals probieren.

Bei `notify()` würde immer nur ein Auto an der Ampel losfahren.
Man könnte das beheben, indem jedes Auto vor dem Losfahren noch ein weiteres `notify()` erzeugt. Das ist aber eher ein work-around und nicht sehr schön.

d. Falls Sie bei der Implementierung der Klasse Ampel die Methode `notify()` benutzt haben: Begründen Sie, warum `notifyAll()` nicht unbedingt benötigt wird!

Siehe oben.

e. Testen Sie das Programm `TrafficLightOperation.java`. Die vorgegebene Klasse 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.

1.3. Producer-Consumer Problem [PU]

In dieser Aufgabe wird ein Producer-Consumer Beispiel mittels einer Queue umgesetzt.

Dazu wird eine Implementation mittels eines Digitalen Ringspeichers umgesetzt.

[Circular Buffer Animation] | *Circular_Buffer_Animation.gif*

Figure 1. Circular Buffer [Wikipedia]

Hierzu sind zwei Klassen (`CircularBuffer.java`, `Buffer.java`) vorgegeben, mit folgendem Design:

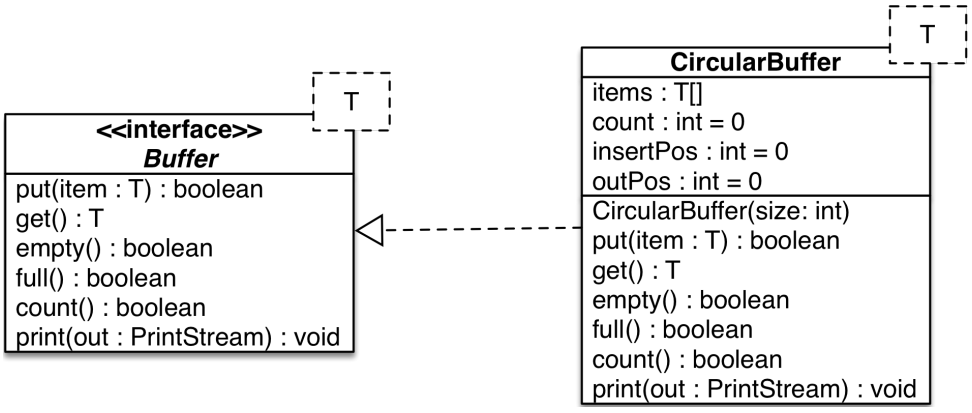

Figure 2. Circular Buffer Design

Analyse der abgegebenen CircularBuffer Umsetzung.

Mit dem Testprogramm `CircBufferSimulation` kann die Funktion der `CircularBuffer` Implementierung analysiert werden. Der mitgelieferte `Producer`-Thread generiert kontinuierlich Daten (in unserem Fall aufsteigende Nummern) und füllt diese mit `buffer.put(…)` in den Buffer. Der `Consumer`-Thread liest die Daten kontinuierlich mit `buffer.get()` aus dem Buffer aus. Beide Threads benötigen eine gewisse Zeit zum Produzieren bzw. Konsumieren der Daten. Diese kann über die Variablen `maxProduceTime` bzw. `maxConsumeTime` beeinflusst werden. Es können zudem

mehrere Producer- bzw. Consumer-Threads erzeugt werden.

a. Starten Sie `CircularBufferSimulation` und analysieren Sie die Ausgabe. Der Status des Buffers (belegte Positionen und Füllstand) wird sekündlich ausgegeben. Alle fünf Sekunden wird zudem der aktuelle Inhalt des Buffers ausgegeben.
   - Wie ist das Verhalten des `CircularBuffer` bei den Standard-Testeinstellungen?

     > `Producer` und `Consumer` arbeiten etwa im gleichen Rhythmus. Das heißt der Buffer ist immer leicht gefüllt, aber nie ganz voll oder leer.

b. Analysieren Sie die Standard-Einstellungen in `CircularBufferSimulation`.
   - Welche Varianten gibt es, die Extrempositionen (Buffer leer, bzw. Buffer voll) zu erzeugen?

     > Damit der Buffer immer etwa gleich gefüllt ist und alles reibungslos funktioniert muss `prodCount * maxProducetime ≈ consCount * maxConsumeTime` sein. Das heißt es wird gleichviel produziert, wie konsumiert.
     >
     > **Buffer leer** → es wird mehr konsumiert als produziert
     >
     > - Mehr Zeit für Produktion: `maxProduceTime` > `maxConsumeTime` setzen.
     > - Mehr Konsumenten als Produzenten: `prodCount` < `consCount`
     >
     > **Buffer voll** → es wird mehr produziert als konsumiert
     >
     > - Mehr Zeit für Konsumation: `maxProduceTime` < `maxConsumeTime` setzen.
     > - Mehr Produzenten als Konsumenten: `prodCount` > `consCount`

   - Was ist das erwartete Verhalten bei vollem bzw. leerem Buffer? (bei Producer bzw. Consumer)

     > **Buffer leer** → `Consumer` muss warten, bis wieder Daten vorhanden sind.
     > **Buffer voll** → `Producer` muss warten, bis wieder Platz für Daten vorhanden ist

c. Testen Sie was passiert, wenn der Buffer an die Kapazitätsgrenze kommt. Passen Sie dazu die Einstellungen in `CircularBufferSimulation` entsprechend an.

   

   Belassen sie die Anzahl Producer-Threads vorerst auf 1, damit der Inhalt des Buffers (Zahlenfolge) einfacher verifiziert werden kann.

   - Was Stellen Sie fest? Was passiert wenn der Buffer voll ist? Warum?

Damit es einfacher verfolgt werden kann, sollte nur `maxProduceTime` verkleinert bzw. `maxConsumeTime` vergrössert werden.

Sobald der Buffer voll ist, werden die neue produzierten Daten *ignoriert*; d.h. sozusagen weggeworfen. Das ist gut an den Lücken in der Zahlenfolge im Buffer zu erkennen.

d. Wiederholen Sie das Gleiche für einen leeren Buffer. Passen Sie die Einstellungen so an, dass der Buffer sicher leer wird, d.h. der `Consumer` keine Daten vorfindet.
   - Was stellen Sie fest, wenn der Buffer leer ist? Warum?

Geben Sie gegebenenfalls die gelesenen Werte des Konsumenten-Threads aus.

Hierzu muss die `maxProduceTime` vergrössert bzw. `maxConsumeTime` verringert werden.

Damit man den Effekt sehen kann, muss im `Consumer` der Inhalt des konsumierten Strings ausgegeben werden. Sie stellen fest, dass `null`-Werte geliefert werden, sobald der Buffer leer ist. Es werden also *Fake-Daten* konsumiert bzw. weiterverarbeitet.

Thread-Safe Circular Buffer

In der vorangehenden Übung griffen mehrere Threads (`Producer` & `Consumer`) auf den gleichen Buffer zu. Die Klasse `CircularBuffer` ist aber nicht thread-safe. Deshalb soll jetzt eine Wrapper Klasse geschrieben werden, welche die CircularBuffer-Klasse "thread-safe" macht. Das führt zu folgendem Design:

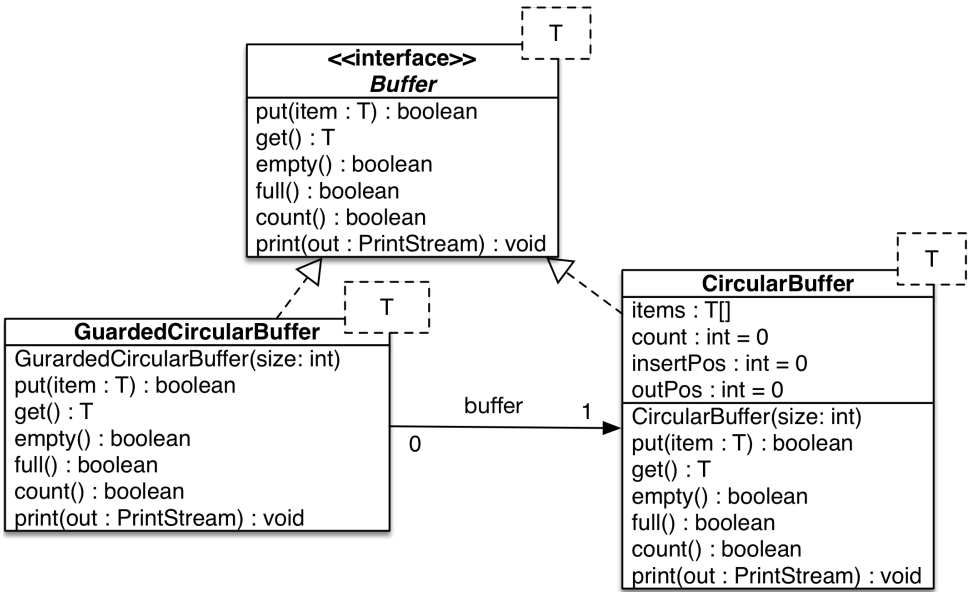

Figure 3. Guarded Circular Buffer Design

Beachten Sie, dass es sich hier um einen Wrapper (keine Vererbung) handelt. Der `GuardedCircularBuffer` hält eine Referenz auf ein `CircularBuffer`-Objekt welches er im Hintergrund für die Speicherung verwendet. Das heißt, viele Methodenaufrufe werden einfach an das gekapselte Objekt weitergeleitet. Einzelne Methoden werden jedoch in ihrer Funktion erweitert. Man spricht auch von "Dekorieren" des Original-Objektes (siehe Decorator-Pattern).

e. Ergänzen Sie die vorhandene Klasse `GuardedCircularBuffer` sodass:
   - Die Methoden `empty`, `full`, `count` das korrekte Resultat liefern.
   - Aufrufe von `put` blockieren, solange der Buffer voll ist, d.h. bis mindestens wieder ein leeres Buffer-Element vorhanden ist.
   - Analog dazu Aufrufe von `get` blockieren, solange der Buffer leer ist, d.h, bis mindestens ein Element im Buffer vorhanden ist.

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

Siehe Lösung: `ch.zhaw.prog2.circularbuffer.GuardedCircularBuffer`

Als erstes müssen sicher die Methoden `get` und `put` als `synchronized` deklariert werden. Da jedoch auch die anderen Methoden auf den Status des Buffers zugreifen, müssen auch diese `synchronized` sein.

Bei `get` und `put` wird jeweils in einer `while`-Schleife der Zustand überprüft und falls nicht erfüllt (`get` → buffer leer, `put` → buffer voll) mit `wait()` gewartet.

Sobald ein Element hinzugefügt bzw. gelesen wurde, werden die wartenden `Consumer` oder `Producer` Threads mit `notify()/notifyAll()` benachrichtigt.

Die eigentlichen Operationen werden auf einem gekapselten `CircularBuffer`-Objekt ausgeführt, welches im Konstruktor erzeugt wird.

Beantworten Sie entweder (i) oder (ii) (nicht beide):

i. 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 benötigt? Begründen Sie Ihre Antwort!

ii. Falls Sie bei der Implementierung der Klasse `GuardedCircularBuffer` die Methode `notify()` benutzt haben: Begründen Sie, warum Sie `notifyAll()` nicht unbedingt benötigten!

Da bei vernünftiger Dimensionierung des Buffers & der Anzahl Threads, sollten jeweils nur entweder Produzenten oder Konsumenten am Warten sein. Dann reicht beim Entfernen bzw. Hinzufügen eines Elementes ein `notify()`. Es wird dann einer der wartenden Threads aufgeweckt, der ein Element hinzufügen bzw. entfernen kann. Beim nächsten Hinzufügen bzw. Entfernen wird ja wieder eine neue Notifikation erzeugt.

Wenn der Buffer aber sehr klein und die Zahl der Produzenten / Konsumenten gross ist, könnte der Fall auftreten, dass sowohl Konsumenten, wie auch Produzenten am Warten sind.

Um sicherzugehen, dass sicher ein Produzent bzw. Konsument zum Zug kommt, wecken wir deshalb am besten alle wartenden Threads.

Das hat aber den Nachteil, dass alle ihre Bedingung überprüfen müssen, obwohl nur der erste passende Thread zum Zuge kommt, da ja nur ein Platz frei wurde, der gefüllte bzw. ein Element vorhanden ist, das abgeholt werden kann.

2. Concurrency 4 — Lock & Conditions, Deadlocks

2.1. Single-Lane Bridge [PU]

Die Brücke über einen Fluss ist so schmal, dass Fahrzeuge nicht kreuzen können. Sie soll jedoch von beiden Seiten überquert werden können. Es braucht somit eine Synchronisation, damit die Fahrzeuge nicht kollidieren. Um das Problem zu illustrieren wird eine fehlerhaft funktionierende Anwendung, in welcher keine Synchronisierung vorgenommen wird, zur Verfügung gestellt. Ihre Aufgabe ist es, die Synchronisation der Fahrzeuge einzubauen.

Die Anwendung finden Sie im Praktikumsverzeichnis im Modul `Bridge`. Gestartet wird sie indem die Klasse `Main` ausgeführt wird (z.B. mit `gradle run`). Das GUI sollte selbsterklärend sein. Mit den zwei Buttons können sie Autos links bzw. rechts hinzufügen. Sie werden feststellen, dass die Autos auf der Brücke kollidieren, bzw. illegalerweise durcheinander hindurchfahren.

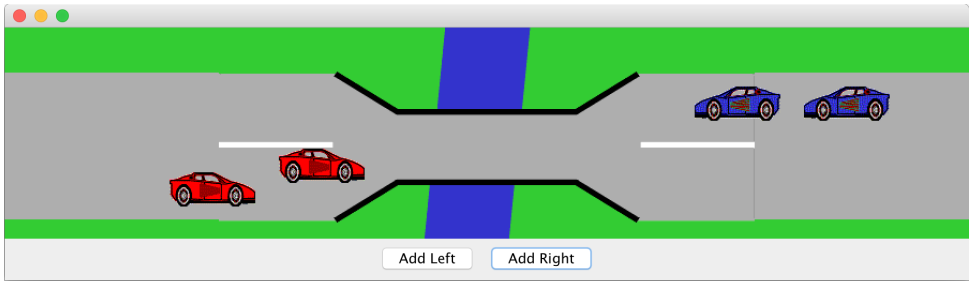

Figure 4. Single-Lane Bridge GUI

Um das Problem zu lösen, müssen Sie die den GUI Teil der Anwendung nicht verstehen. Sie müssen nur wissen, dass Fahrzeuge, die von links nach rechts fahren, die Methode `controller.enterLeft()` aufrufen bevor sie auf die Brücke fahren (um Erlaubnis fragen) und die Methode `controller.leaveRight()` aufrufen, sobald sie die Brücke verlassen. Fahrzeuge in die andere Richtung rufen entsprechend die Methoden `enterRight()` und `leaveLeft()` auf. Dabei ist `controller` eine Instanz der Klasse `TrafficController`, welche für die Synchronisation zuständig ist. In der mitgelieferten Klasse sind die vier Methoden nicht implementiert (Dummy-Methoden).

a. Erweitern sie `TrafficController` zu einer Monitorklasse, die sicherstellt, dass die Autos nicht mehr kollidieren. Verwenden Sie dazu den Lock und Conditions Mechanismus.

Verwenden Sie eine Statusvariable, um den Zustand der Brücke zu repräsentieren (e.g. `boolean bridgeOccupied`).

Siehe Code: `ch.zhaw.prog2.bridge.TrafficControllerA`

b. Passen Sie die Klasse `TrafficController` so an, dass jeweils abwechslungsweise ein Fahrzeug von links und rechts die Brücke passieren kann. Unter Umständen wird ein Auto blockiert, wenn auf der anderen Seite keines mehr wartet. Verwenden Sie für die Lösung mehrere Condition Objekte.

Um die Version aus a. zu behalten, können sie auch eine Kopie (z.B. `TrafficControllerB`) erzeugen und `Main` anpassen, damit eine Instanz dieser Klasse verwendet wird.

Siehe Code: `ch.zhaw.prog2.bridge.TrafficControllerB`

c. Bauen Sie die Klasse `TrafficController` so um, dass jeweils alle wartenden Fahrzeuge aus einer Richtung passieren können. Erst wenn keines mehr wartet, kann die Gegenrichtung fahren.

Mit `ReentrentLock.hasWaiters(Condition c)` können Sie abfragen ob Threads auf eine bestimmte Condition warten.

Siehe Code: `ch.zhaw.prog2.bridge.TrafficControllerC`

2.2. The Dining Philosophers [PA]

Die Lösungen zu den bewerteten Pflichtaufgaben erhalten Sie nach der Abgabe und Bewertung aller Klassen.