Nebenläufige Programme in Java

Problembeschreibung

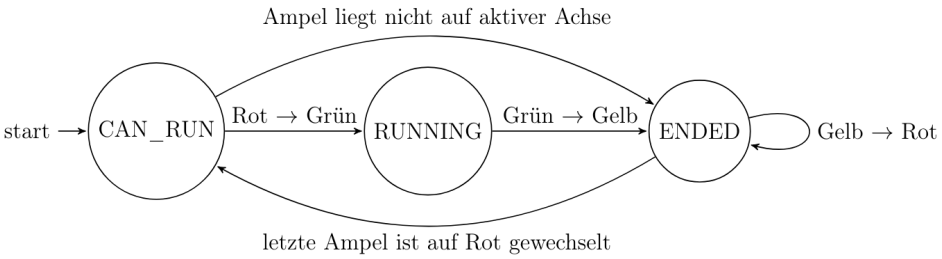
Es soll eine Ampelanlage für eine Kreuzung im Straßenverkehr implementiert werden. Diese Kreuzung besitzt vier Ampeln, die jeweils für eine Richtung (North, East, South, West) der Kreuzung zuständig sind. Wenn eine Ampel schaltet, lautet die Abfolge der Ampelphasen: red, green, yellow. Der Startwert einer Ampel soll red sein. Jede Ampel soll parallel in separaten Threads laufen. Die einzelnen Threads sollten durch einen gemeinsamen Speicher und nicht von einer zentralen Instanz synchronisiert werden. Außerdem ist zu beachten, dass die Beschaltung der einzelnen Ampeln nicht zu Unfällen auf der Kreuzung führen soll.

Lösung

Enum “Cycle”

Bei diesem Enum handelt es sich um den ‘shared State‘ der TrafficLights.
Der Zustand CAN_RUN zeigt an, dass die Ampel in der Lage ist, auf Grün zu wechseln.
Der Zustand RUNNING zeigt an, dass die Ampel im Moment nicht im “State” Rot ist.
Der Zustand ENDED zeigt an, dass die Ampel eine Grünphase durchlaufen hat und nun wieder Rot ist.

```
enum Cycle {
 CAN_RUN,
 RUNNING,
 ENDED,
}
```



Variablen der Ampel

Die Klasse TrafficLight besitzt mehrere Variablen, die Entweder der Klasse exklusiv zu verfügung stehen oder mit anderen Threads geteilt werden müssen.

```
private static volatile boolean running = true;
```

Die Variable “running” ist eine geteilte Variable, die für die While-Schleife in der Run-Methode benutzt wird. Solange diese Variable “true” ist, wird der Thread abgearbeitet. Der Aufruf der Halt-Methode setzt diese Variable auf “false” und stoppt somit alle Instanzen von TrafficLight.

```
private final CardinalDirection cd;
```
Die Variable “cd” ist vom Typ CardinalDirection. Diese zeigt an, auf welcher Himmelsrichtung sich die Ampel befindet.

```
private static volatile CardinalDirection dir;
```
Die Variable “dir” ist ebenfalls vom Typ CardinalDirection. Diese zeigt die Achse an, auf der die Ampeln Grün haben dürfen. Die Achse besteht aus einer Himmelsrichtung und dessen Gegenüber.

```
private static volatile Cycle[] cycle = new
Cycle[CardinalDirection.cardinality];
```
Bei dem Cycle-Array “cycle” handelt es sich um den geteilten Speicher der Threads. Dabei entspricht die Array-Laenge der Anzahl der Werte im Enum CardinalDirection.

```
private Colour state;
```
In der Colour-Variable “state” wird jeweils die Farbe der Ampel gespeichert.
Die Ampelfarbe ist kein geteilter Speicher.

```
private static final Object lock = new Object();
```
Das Objekt “lock” wird für die Synchronisation der einzelnen Threads benötigt.

Main-Loop

Jede Ampel arbeitet kontinuierlich die folgenden Instruktionen ab:

Wenn eine Ampel Rot ist, überprüft sie, ob sie auf der Achse liegt, die auf Grün wechseln darf.
Sollte dies der Fall sein, überprüft sie zusätzlich noch ob ihr Zustand auf CAN_RUN steht.
Wenn beide Bedingungen zutreffen, wechselt die Ampel in den Zustand RUNNING und schaltet auf Grün.
Sollte eine dieser Bedingungen nicht eintreffen, wechselt sie direkt in den ENDED Zustand.

```
if (state == RED) {
 synchronized (lock) {
   if (locationOnAxis() && cycle[cd.ordinal()] == CAN_RUN) {
     cycle[cd.ordinal()] = RUNNING;
     nextState();
   } else {
     cycle[cd.ordinal()] = ENDED;
   }
 }
```

Wenn die Ampel nicht im Zustand rot ist, wechselt sie ohne weitere Prüfungen in den nächsten Zustand.

Sollte die Ampel danach Rot sein, wechselt sie in den ENDED Zustand.
Darauf überprüft sie, ob sie die letzte Ampel ist, die in den ENDED Zustand gewechselt hat.

Sollte dies der Fall sein, aktualisiert sie die Achse, die auf Grün wechseln darf und setzt sie alle State-Maschinen zurück.
Sollte die Ampel erste sein, die in den ENDED Zustand wechselt, passiert nichts und die While-Schleife wird weiter ausgeführt.

```
else {
      nextState();
      if (state == RED) {
        synchronized (lock) {
          cycle[cd.ordinal()] = ENDED;
          if (allEnded()) {
            dir = CardinalDirection.next(dir);
            Arrays.fill(cycle, CAN_RUN);
          }
        }
      }
    }
  }
}
```

Die “allEnded” Funktion überprüft ob, sich alle Ampeln im Zustand “Ended” befinden. Wenn dies Fall ist, gibt sie “true” zurück, ansonsten “false”.

```
 private boolean allEnded() {
 for (Cycle c : cycle) {
   if (c != ENDED) {
     return false;
    }
 }
 return true;
}
```

Diese Methode überprüft ob die Ampel in der gerade zugelassene Achse ist.

```
private boolean locationOnAxis() {
   return cd == dir || cd == opposite(dir);
}
```