Vorlesung: Dr.-Ing. Matthias Becker - xmb@sim.uni-hannover.de
Übung: Philipp Schiffmann - schiffmann@sim.uni-hannover.de
Florian Chudigiewitsch - fch@thi.uni-hannover.de

Assignment 3

Vererbung, Javadoc

Abgabe: **09.05.2018, 13:45 Uhr**

I. Organisatorisches

Eine Musterlösung für assignment 2 steht ab dem 03.05. im StudIP zur Verfügung. Du kannst sie benutzen, um deinen aktuellen Lernfortschritt zu validieren, denn sie benutzt ausschließlich Sprachelemente, die bereits in der Vorlesung behandelt wurden. Du solltest also in der Lage sein, den Quellcode vollständig nachzuvollziehen. Verstehst du einen Teil der Implementierung nicht, lass ihn dir zeitnah von einem Kommilitonen oder Tutor erklären! Bitte denk dabei daran, dass die Modellierung in der Aufgabenstellung vorgegeben wurde und wir die Fähigkeit, eine solche Architektur selbst zu entwerfen, aktuell **nicht** von dir erwarten.

Wenn du möchtest, darfst du außerdem die Musterlösung anstatt deiner eigenen Abgabe als Grundlage für assignment 3 benutzen. Das erfordert aber natürlich zusätzliche Einarbeitungszeit, um dich in den fremden Quellcode hineinzudenken. Wir empfehlen es dir also nur dann, wenn du bei der Bearbeitung von assignment 2 erhebliche Fehler gemacht hast.

Hinweis: Die Musterlösung benennt alle Klassen, Methoden und Variablen auf englisch. Das ist guter Stil und du solltest es dir auf jeden Fall angewöhnen, deinen Code und die Dokumentation auf englisch zu verfassen! Da wir dir das aber nicht beibringen und auch nicht voraussetzen können, ist für Hausaufgaben auch deutsch akzeptabel.

II. Javadoc

Du wirst für dieses assignment den Zufallszahlengenerator (random number generator) der Java-Standardbibliothek benutzen. Dafür musst du aber nicht komplett verstehen, wie er intern arbeitet. Es reicht, wenn du in der Dokumentation nachschlägst, wie du ihn instanzieren und welche Methoden du auf ihm aufrufen kannst. Die Benutzbarkeit dieser Klasse hängt also unmittelbar davon ab, wie gut sie dokumentiert ist.

Sobald du in einem Unternehmen als Programmierer arbeitest oder eigene open source Projekte veröffentlichst, werden andere Menschen auch deinen Quellcode benutzen, also musst du lernen, ihn für sie zu dokumentieren. Du hast mit der `Random`-Klasse bereits ein Beispiel, an dem du dich orientieren kannst. Inhaltlich sind die wichtigsten Faustregeln:

1. Lass das offensichtliche weg. Wie die Programmiersprache funktioniert (`i++; // erhöht i um 1`) sollten deine Leser schon wissen. Und das die Funktion `int getLength()` einen `int` zurückgibt, sehen sie an der Signatur, das brauchst du also auch nicht hinzuschreiben.

2. Schreib die *Bedeutung* der Klasse / Methode / Parameter / Rückgabewerte auf. (Was repräsentiert ein Objekt der Klasse `Auto`, wo wird es benutzt und wofür? Wie ist ein Rückgabewert von `true` bei einer Methode zu interpretieren?)

3. Dokumentiere die gültigen Wertebereiche für Parameter und was passiert, wenn sie nicht eingehalten werden. (Liegen die Felder einer Straße im Intervall [0, length - 1], oder [1, length]? Was passiert wenn man versucht, ein Feld außerhalb des Intervalls zu belegen?)

Um deine Dokumentation für andere Entwickler festzuhalten, benutzt du in Java *doc comments*. Das Format wurde in der Vorlesung vorgestellt. Mit dem `javadoc`-Kommandozeilenprogramm kannst du daraus ein HTML-Dokument generieren. Kommentiere alle Klassen, Methoden und Attribute deines Quellcodes, die nicht selbsterklärend sind, und füge die HTML-Dokumentation zum repository hinzu.

III. Rückblick assignment 2

In der letzten Woche hast du ein Grundgerüst für eine Verkehrssimulation geschrieben, das wir nun weiter ausbauen. Die Simulationswelt besteht dabei aus Objekten drei verschiedener Klassen. Das Straßennetz ist aus Kreuzungen zusammengesetzt, die durch Fahrbahnen verbunden sind. Jede Fahrbahn enthält eine Reihe von Feldern, auf denen jeweils maximal ein Auto stehen darf. Die Kreuzungen dienen bisher lediglich als Knotenpunkte, damit die Fahrbahnen irgendwo hinführen. Das Straßennetz wird von Autos befahren. Jedes Auto folgt dabei dem Ablauf:

```
if I have not reached the end of the current street:
    if the next field is empty:
        Move away from the current field.
        Move onto the next field.
else if I have not reached my destination:
    Find the next road on my route.
    if field 0 of that road is empty:
        Move away from the current field.
        Move onto field 0 of the new road.
else:
    Move away from the current field.
    De-register from the system.
```

Die Autos sind also momentan die einzigen aktiven Objekte: Jedes von ihnen hat eine eigene Route abzufahren, und bewegt sich selbstständig durch das Straßennetz. Fahrbahnen hingegen werden nicht selbstständig aktiv, sondern *reagieren* lediglich auf Anfragen („Ist dieses Feld frei?“) und Anweisungen („Reserviere dieses Feld“, „Gib dieses Feld frei“) von Autos. Und Kreuzungen haben noch nicht einmal einen veränderlichen Zustand, sondern stellen einfach nur die Information bereit, welche Fahrbahnen an diesem Punkt aufeinandertreffen.

IV. Verkehrssimulation Fortsetzung

Diese Woche fügen wir der Verkehrssimulation weitere aktive Objekte in Form von neuen Kreuzungstypen hinzu.

Das erste davon ist die `Systemgrenze`. Das System, das wir simulieren, ist natürlich kleiner als ein reales Straßennetz. Das liegt daran, dass wir nur einen Ausschnitt betrachten. Autos werden also nicht an den Punkten A und B frisch montiert und an D und E wieder verschrottet, sondern betreten und verlassen einfach nur unser Sichtfeld. Objekte der Klasse `Systemgrenze` sollen von nun an genau diese Übergangspunkte darstellen. Jede Systemgrenze hat dabei eine eigene Wahrscheinlichkeit,

wie oft an ihr ein neues Auto das System betritt – liegt dieser Wert zum Beispiel bei 10%, wird durchschnittlich alle 10 Zeitschritte ein Auto dort erscheinen. Außerdem enthält jede Systemgrenze ein Array von Routen die dort beginnen, und jedes dort erscheinende Auto fährt eine zufällig ausgewählte dieser Routen ab.

Der zweite neue Objekttyp repräsentiert eine Ampel-gesteuerte Kreuzung. Steht ein Auto auf einer Fahrbahn, die an einer solchen Kreuzung endet, darf das Auto nur dann passieren, wenn die Straße gerade grün hat. Die Ampel hat einen internen Zähler, wie lange die aktuelle Ampelphase noch andauert. Erreicht der Zähler den Wert 0, erhält direkt die nächste Straße grün – eine Gelbphase muss nicht simuliert werden.

Schreibe für jeden der beiden Typen `Systemgrenze` und `Ampelkreuzung` eine eigene Klasse, und lasse beide neuen Klassen von `Kreuzung` erben. Außerdem wird jedes dieser Objekte in jedem Zeitschritt aktiv. Damit deine `Simulation` nicht für jeden Typ ein eigenes Array verwalten muss, füge eine Oberklasse `Akteur` hinzu und lasse `Auto` und `Kreuzung` davon erben.

V. Lösungshinweise

1. Der Befehl `git diff` zeigt dir die Änderungen an allen Dateien des repository seit dem letzten commit an. Wenn du also weißt, dass der letzte commit fehlerfrei war, aber der aktuelle Code fehlerhaft ist, dann musst du den Fehler nur in denjenigen Zeilen suchen, die dir der `diff`-Befehl anzeigt. Je öfter du committest, umso weniger Code musst du dabei vergleichen.

2. Im StudIP findest du die abstrakte Klasse `Actor`. Lege die Datei in dein repository und passe den package-Namen an. Lass deine `Auto`-Klasse von dieser Klasse erben. Deine Klasse `Simulation` speichert alle Autos, die gerade im System sind, in einem Array. Ändere diese Variable auf `Actor[] actors`. Passe beide Klassen an, sodass dein Code wieder compiliert und das gleiche Ergebnis ausgibt wie vorher. Sobald das Programm wieder funktioniert, committe deine Änderungen.

3. Lasse die Klasse `Kreuzung` von `Actor` erben und füge eine `update`-Methode mit leerem Funktionsrumpf (`{}`) hinzu, damit dein Code weiterhin compiliert.

4. Erstelle eine Klasse `Systemgrenze`, die von `Kreuzung` erbt. Objekte dieser Klasse enthalten das Attribut `Ankunfts-Wahrscheinlichkeit` und ein Array von Routen, die dort erscheinende Autos abfahren. Die `update`-Methode von `Systemgrenze` soll sich folgendermaßen verhalten:

```
Choose a random number.
If random number < spawn probability:
    choose a random route.
    If position 0 on the first road of this route is free:
        Place a new car with the chosen route on this road.
        Add the car to the simulation actor list.
```

   Um die Zufallszahlen zu generieren, benutze ein Objekt vom Typ `java.util.Random`. Du musst es importieren, wie andere Klassen auch. Passe das Straßennetz an, sodass die Kreuzungen *A*, *B*, *D*, und *E* nun Systemgrenzen-Objekte sind. *A* hat eine Wahrscheinlichkeit von 20%, Autos zu erzeugen, und vergibt die Routen *A, C, D* und *A, C, E*. *B* hat eine Wahrscheinlichkeit von 30% und die Routen *B, C, D* und *B, C, E*. Die Systemgrenzen *D* und *E* haben eine Wahrscheinlichkeit von 0% und keine Routen. Entferne die fix vorgegebenen Autos aus assignment 2. Sobald der Code funktioniert, committe deine Änderungen.

5. Erstelle eine Klasse `Ampelkreuzung` und lasse sie von `Kreuzung` erben. Eine Ampel hat ein Attribut `verbleibendeZeit`, das die verbleibende Zeit bis zum nächsten Phasenwechsel speichert, sowie ein Attribut, welche Phase gerade aktiv ist. Die `update`-Methode der Ampelkreuzung soll sich folgendermaßen verhalten:

```
Decrease remaining greenlight time by 1.
If remaining greenlight time = 0:
    Switch to next greenlight phase.
    Reset remaining greenlight time.
```

   Füge der Klasse `Fahrbahn` ein Attribut `grünPhase` hinzu. Passe die `update`-Methode von Auto an, sodass ein Auto nur dann über Ampelkreuzungen fährt, wenn die Grünphase der Ampel gleich der Grünphase der eigenen Straße ist.

   Ersetze die Kreuzung *C* durch eine Ampelkreuzung mit Phasendauer 4. Deklariere die Klasse Kreuzung als abstrakt und lösche die leere Methode *update*, da sie nun nicht mehr gebraucht werden.

VI. Bewertungskriterien

1. Alle Klassen, Methoden und Attribute sind mit Kommentaren versehen, die erläutern, welchen Zweck die jeweilige Entität erfüllt.
2. Die Kommentare von Methoden benutzen die javadoc-tags `@param` und `@return`, sofern angemessen. Du darfst gern weitere doc tags benutzen.
3. Die vom `javadoc`-Befehl generierte HTML-Dokumentation liegt im Ordner `doc/` im repository.
4. `Systemgrenze` und `Ampelkreuzung` sind Unterklassen von `Kreuzung`.
5. `Auto`, `Systemgrenze` und `Ampelkreuzung` sind Unterklassen von `Actor`. Die Simulationsschleife iteriert nun über alle Actors, statt nur über die Autos.
6. Autos warten vor einer Kreuzung, bis die Straße grünes Licht bekommt, auf der sie aktuell stehen.