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 5

Deep copy, information hiding

Abgabe: **30.05.2018, 13:45 Uhr**

I. Einschwingzeit und wiederholte Versuche

Wenn man sich für die Messwerte im laufenden (stationären) Betrieb interessiert, darf man bei der statistischen Auswertung die Messwerte aus der Einschwingzeit (transiente Phase) nicht berücksichtigen. In einem frisch erzeugten `Simulation`-Objekt sind alle Straßen leer. Deshalb haben die ersten Autos es leichter, an ihr Ziel zu kommen, und ihre Reisegeschwindigkeit ist nicht repräsentativ für das Verhalten im laufenden Betrieb.

Weil die Simulation stochastisch ist, also Zufallselemente enthält, ist ein einzelner Lauf nicht repräsentativ, sondern es müssen mehrere Läufe gemacht werden und Mittelwert und Varianz berechnet werden.

Um die Messwerte für den stationären Betrieb zu bekommen, soll die Simulation bis zu einem bestimmten Zeitpunkt laufen (settling-time). `DataSources` sollen während der während dieser Phase noch keine Werte protokollieren. Dann wird der Zustand der gesamten Simulation als Ausgangspunkt für mehrere Simulationsläufe benutzt. Dazu wird in einer Schleife jeweils eine Kopie der gesamten Simulation erstellt, simuliert und die Messwerte berechnet. Nach der Schleife wird Mittelwert und Varianz der stationären Messwerte ausgegeben.

Erweitere dein Programm um die Kommandozeilenparameter `-settling-time <value>` und `-repeat <value>`. Ein Aufruf mit `-settling-time 20 -duration 100 -repeat 10` bewirkt etwa, dass die Simulation sich 20 Zeitschritte lang einschwingt, dann das `Simulation`-Objekt 9 mal kopiert wird, und anschließend jede der 10 Simulationen um 100 Zeitschritte fortgeführt wird.

„Kopieren“ meint dabei, ein neues Objekt vom Typ `Simulation` zu erzeugen, das den gleichen Zustand hat wie das alte, aber beide unabhängig voneinander arbeiten können. Das bedeutet, dass alle internen Objekte mit veränderlichem Zustand – Autos, Fahrbahnen und die verschiedenen Kreuzungen – ebenfalls kopiert werden müssen.

II. information hiding

information hiding ist ein Modellierungs-Prinzip, das besagt, dass Objekte keine internen Informationen nach außen preisgeben sollten. Zum Beispiel hat eine Ampel einen Zähler, wie lange die aktuelle Grünphase noch dauert. Dieser Zähler sollte ausschließlich von der Ampel selbst ausgelesen oder überschrieben werden; es gibt keinen Grund, wieso ein fremdes Objekt darauf zugreifen sollte, also ist es mit Sicherheit ein Programmierfehler, wenn es doch so einen Zugriff gibt.

Der Compiler kann dich vor solchen Fehlern warnen, aber dafür musst du ihm sagen, welche Attribute und Methoden du als intern bzw. extern vorgesehen hast. In Java tust du dies mit den Schlüsselwörtern `public`, `protected` und `private`. Füge jeder Methode eines dieser Schlüsselwörter hinzu.

III. Fehlerbehandlung

In der Vorlesung wurden die exceptions vorgestellt, ein Mechanismus zum systematischen Erkennen und Behandeln von Fehlern.

Schreibe in deiner Simulation zu Beginn aller Methoden Prüfungen, ob die Methode auf den übergebenen Parametern überhaupt arbeiten kann und, sofern angemessen, ob das Objekt gerade in einem gültigen Zustand für diese Operation ist. Werfe bei unzulässigen Werten und Zuständen eine Exception. Dokumentiere die gültigen Wertebereiche und die geworfenen exceptions in den Javadocs.

IV. getter und setter

Viele Objektattribute solltest du nicht `public` sondern `protected` bzw. `private` deklarieren, damit du auch hier die Zuweisung von ungültigen Werten verhindern kannst. In Java hat sich deshalb das Prinzip durchgesetzt,

Methoden arbeiten nicht nur auf Parametern, sondern auch auf den Attributen des eigenen Objekts. Damit du diese nicht in jeder Methode wieder neu überprüfen musst, macht es Sinn, ihnen von Anfang an nur gültige Werte zuweisen zu lassen. Damit du die Wertzuweisung an Attribute kontrollieren kannst, hat sich in Java das Prinzip durchgesetzt, *alle* Attribute als `protected` bzw. `private` zu deklarieren und stattdessen sogenannte getter und setter für alle von außen les- bzw. schreibbaren Attribute zu schreiben.

Setze dieses Prinzip ebenfalls in deinem Programm um. Deklariere alle Attribute als `private` (oder `protected`, falls eine erbende Klasse darauf zugreifen muss); füge eine getter-Methode hinzu, falls andere Objekte diesen Zustand auslesen müssen; füge einen setter hinzu, wenn andere Objekte diesen Zustand setzen dürfen.

V. Lösungshinweise

1. Java sieht zum Kopieren von Objekten die Methode `clone` vor, die du in `Simulation` überschreiben musst.

2. Wenn du mehrere miteinander verknüpfte Objekte kopieren willst, brauchst du einen Startpunkt und eine eindeutige Richtung, sonst kannst es passieren, dass du in eine Endlosschleife läufst. (z.B. Kreuzung *A* kopiert in der eigenen `clone`-Methode die Fahrbahn *A* -> *C*, diese kopiert wiederum in `clone` die Kreuzung *A*, ...) Du brauchst also entweder einen Mechanismus, um Endlosrekursion zu erkennen und aufzulösen, oder du kopierst das Straßennetz aus `Simulation.clone` heraus und implementierst `clone` in den `model`-Klassen *nicht*.

3. Wenn du den Pseudo-Zufallszahlengenerator `java.util.Random` ebenfalls klonst, generiert jede Kopie identische Zahlenfolgen. Das ist zwar das gewünschte Verhalten von `Simulation.clone`. Da es aber nicht zweckdienlich für die statistische Auswertung ist, ersetze nach dem Klonen den Zufallszahlengenerator durch einen neuen, zufällig initialisierten.

VI. Bewertungskriterien

1. Die Klasse `Simulation` lässt sich kopieren, und die Kopie enthält ihren eigenen, vom Original unabhängigen Systemzustand.

2. Du kannst erklären, wieso du Methoden als `public` deklariert hast.
3. Alle Attribute sind `protected` oder `private`.
4. Du hast die Methoden in deinem Programm identifiziert, die nur auf einem begrenzten Wertebereich arbeiten, reagierst auf ungültige Parameter mit exceptions, und hast diese Fälle auch in den Javadocs dokumentiert.