Ergebnisbericht Übungsblatt 1

Karim Amri

29. November 2024

Einleitung

Ich habe zur Lösung der Aufgaben Java und RMIs genutzt. Im Ergebnisbericht von Aufgabe 1 werden außer der Kommunikation von Glühwürmchen noch Details der Modellierung und des Programmablaufs beschrieben. Im Ergebnisbericht von Aufgabe 2 dann nur noch wesentliche Unterschiede.

Aufgabe 1

Klassen

Es gibt drei Klassen `Firefly`, `World` und `FireflySimulation`. `Firefly` und `World` modellieren den Torus der Glühwürmchen, während `FireflySimulation` sich um die Visualisierung kümmert und die `main()`-Methode beinhaltet. Ein Klassendiagramm ist in Abb. 1 zu sehen.

Die Funktionsweisen der einzelnen Klassen werden in den nächsten Abschnitten näher beschrieben.

Programmstart

Beim Start des Programms wird ein anonymes Objekt der Klasse `FireflySimulation` erstellt und dessen `startSimulation`-Methode aufgerufen. Ein Ablaufdiagramm ist in Abb. 2 zu sehen.

Beim Erstellen des `FireflySimulation`-Objektes wird sein `World`-Objekt `world` mit denselben Argumenten konstruiert. `world` enthält ein zweidimensionales `Firefly`-Array `grid`, in dem die simulierten Glühwürmchen gespeichert werden. Beim Konstruieren von `world` wird `grid` mit neuen `Firefly`-Objekten gefüllt und diese dann in einer weiteren doppelten Schleife erst anständig konstruiert, indem deren `init(Firefly[])`-Methode aufgerufen wird. Diese Konstruktion beinhaltet das Übergeben von Nachbarn und muss in zwei Schritten durchgeführt werden, da einem `Firefly`-Objekt in der `init(Firefly[])`-Methode gegebenenfalls „unfertige“ Objekte übergeben werden. Kurz gesagt: Man kann einem `Firefly` bei seiner Konstruktion nicht seine Nachbarn mitgeben, wenn diese noch gar nicht existieren.

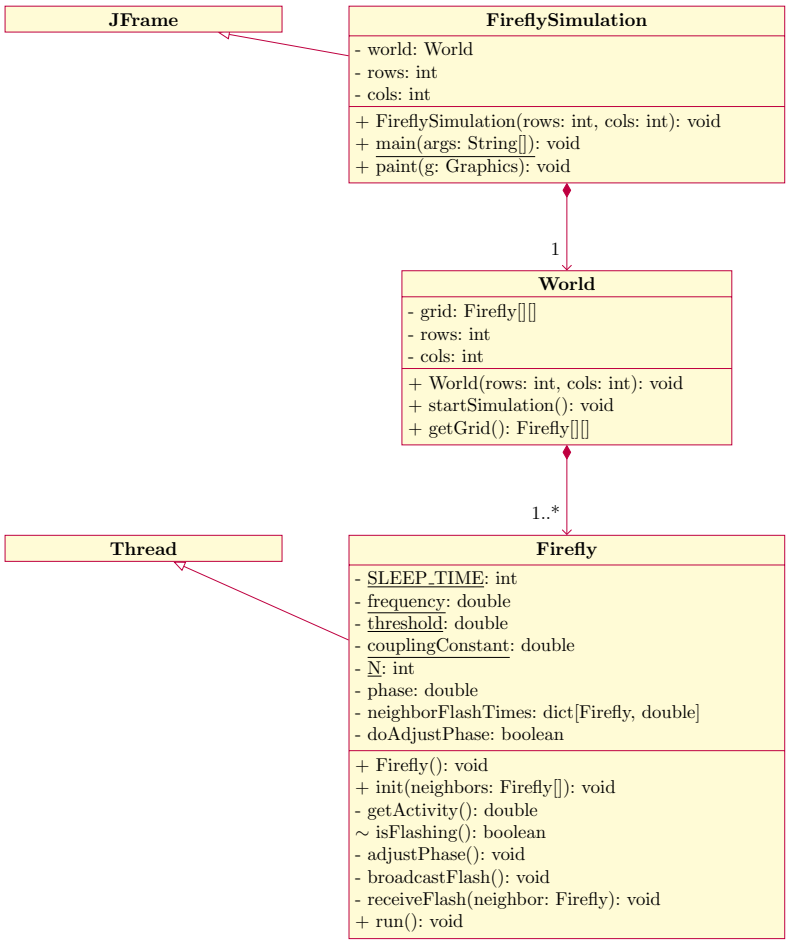

Abb. 1: Klassendiagramm Aufgabe 1

Durch Aufrufen von `startSimulation` werden dann die `Firefly`-Threads gestartet.

`FireflySimulation` erbt von `javax.swing.JFrame` und implementiert durch die Methode `paint(java.awt.Graphics)` die Visualisierung des Torus in Form eines Rechtecks.

Modellierung der Glühwürmchen

Die Klasse `Firefly` beinhaltet neben der Modellierung eines Glühwürmchens nach dem Kuramoto-Modell noch Funktionalität zur Kommunikation mit anderen Glühwürmchen. Die statischen Felder beinhalten Werte, die für alle Glühwürmchen gleich sind, wie etwa die **Frequenz** f des Aufleuchtens (`frequency`).

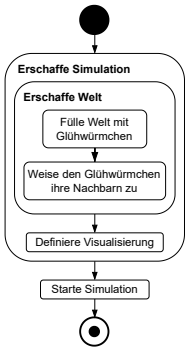

Abb. 2: Programmstart (Aufg. 1)

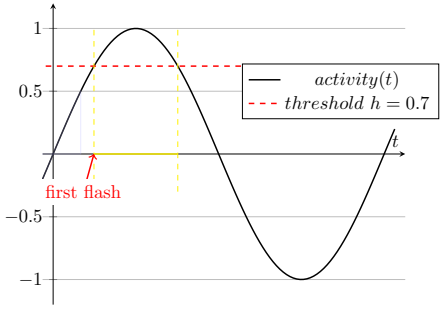

Abb. 3: Aktivität eines Glühwürmchens

Abhängig vom aktuellen **Zeitpunkt** t und der eigenen **Phase** p (`phase`) wird die **Aktivität** a (`getActivity()`) eines Glühwürmchens berechnet durch

$$a := \sin(t \cdot f \cdot 2 \cdot \pi + p).$$

Ein Glühwürmchen leuchtet genau dann auf (`isFlashing()`), wenn die Aktivität größer als der - für alle Glühwürmchen identische - **Threshold** h (`threshold`) ist. Die Aktivität eines Glühwürmchens und dessen Beziehung zum Threshold sind in Abb. 3 dargestellt.

Zustände der Glühwürmchen

Die `run()`-Methode beinhaltet eine Dauerschleife in der kontinuierlich - mit zeitlichem Abstand von `SLEEP_TIME` Millisekunden - überprüft wird, welchen Zustand das Glühwürmchen gerade hat. Ein Glühwürmchen kann entweder:

1. Das erste Mal in der aktuellen Periode leuchten („flashen“),
2. das mindestens zweite Mal in der aktuellen Periode leuchten oder
3. gar nicht leuchten.

Die Zustände eines Glühwürmchens sind genauer im Zustandsdiagramm in Abb. 4 dargestellt. Fall 1. ist der Zustand unten links, Fall 2. ist der Zustand unten rechts und Fall 3. beschreibt die beiden Zustände oben. Nur im Fall des ersten Leuchtens der Periode wird den Nachbarn durch `broadcastFlash()` mitgeteilt, dass das Glühwürmchen aufgeleuchtet hat.

Kommunikation der Glühwürmchen

Die Nachbarn eines Glühwürmchens sind in der `HashMap neighborFlashTimes` gespeichert. Dort stellen sie die Schlüssel dar, zu denen der letzte bekannte Zeitpunkt des jeweiligen Aufleuchtens gespeichert werden soll. Ruft nun

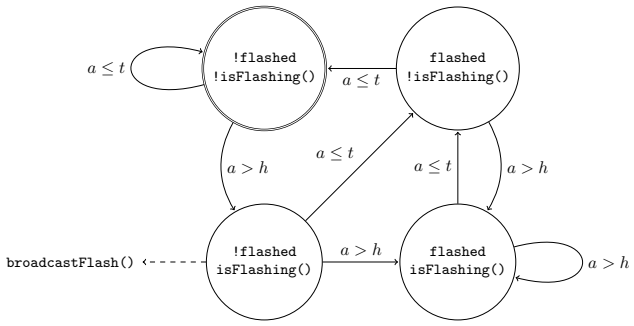

Abb. 4: Zustandsdiagramm Glühwürmchen (Aufg. 1); Der Zustand oben links ist der Startzustand eines Glühwürmchens

Glühwürmchen A `broadcastFlash()` auf, so wird die `receiveFlash(Firefly)`-Methode aller Nachbarn aufgerufen. Das übergebene `Firefly`-Objekt ist A selbst. Zusätzlich zum Setzen des `doAdjustPhase`-Flags auf `true` wird im `receiveFlash(Firefly)`-Aufruf der Nachbarn die aktuelle Zeit gemessen und als Wert zum Schlüssel A in der jeweiligen `HashMap` des Nachbarn gespeichert. Diese Werte sind vor der ersten Aktualisierung übrigens für ein einzelnes Glühwürmchen alle gleich und werden aus der Phase des Glühwürmchens konstruiert. Genauer: $t^* := \frac{\arcsin h+p}{f \cdot 2 \cdot \pi}$. So haben Nachbarn vor ihrem ersten mitgeteilten Aufleuchten quasi keinen Einfluss auf die Anpassung der Phase, können aber trotzdem schon in `neighborFlashTimes` gespeichert sein. Andernfalls müsste man irgendwie Fallunterscheidungen beim Anpassen der Phase machen, wobei ein Fall wahrscheinlich nur einmal am Anfang eintritt. Diese Lösung erscheint mir weitaus eleganter.

Anpassen der Phase

Ist t_i der in `neighborFlashTimes` gespeicherte Zeitpunkt des letzten Aufleuchtens des Nachbarn i, so ist die (vermutete) Phase

$$p_i := t_i \cdot f \cdot 2 \cdot \pi - \arcsin(h).$$

Für Nachbarn $1, ..., n$ und deren Werte $t_1, ..., t_n$ ergibt sich eine Anpassung der Phase um

$$\Delta p := \frac{\texttt{couplingConstant}}{n} \sum_{i=1}^{n} \sin(p_i - p).$$

Mit n=`Firefly.N`=4 wird dieser Wert in `adjustPhase()` berechnet und auf `phase` addiert. `adjustPhase()` wird allerdings nur aufgerufen, nachdem mindestens ein Nachbar sein Aufleuchten mitgeteilt hat, `doAdjustPhase` also `true` ist. Dieses Flag wird zu Beginn von `adjustPhase()` auf `false` gesetzt, kann am Ende der Phasenanpassung allerdings wieder `true` sein, da wir es hier mit Nebenläufigkeit zu tun haben (so macht es auch Sinn, das Flag direkt am Anfang auf `false` zu setzen).

Aufgabe 2

Da jedes Glühwürmchen nun ein eigenständiges Programm ist, gibt es die Klassen `FireflySimulation` und `World` nicht mehr. Dafür gibt es jetzt das Interface `FireflyInterface`, das von `java.rmi.Remote` erbt. Ein Klassendiagramm ist in Abb. 5 zu sehen.

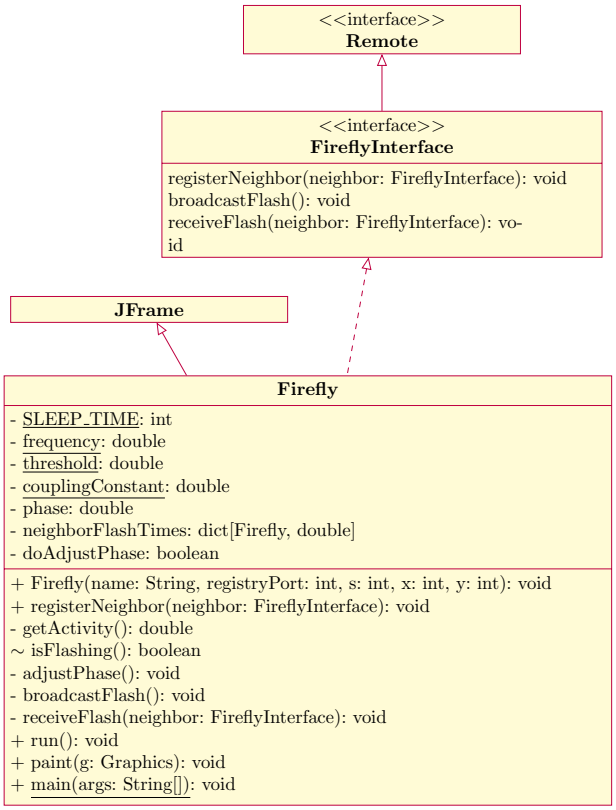

Abb. 5: Klassendiagramm (Aufg. 2)

Die Schnittstelle `FireflyInterface`

Die drei in `FireflyInterface` definierten Methoden sind diejenigen, für die die Remote Method Invocation bereitgestellt wird. `broadcastFlash()` und `receiveFlash(FireflyInterface)` kennen wir bereits, an deren Implementierungen in `Firefly` hat sich auch nichts geändert. Die neue Methode ist `registerNeighbor(FireflyInterface)`. Jedes Glühwürmchen fungiert nun **sowohl als Server, als auch als Client**. Um den beiden Glühwürmchen A und B also ihre Nachbarschaft bekannt zu machen, müssen sowohl A`.registerNeighbor(`B`)` als auch B`.registerNeighbor(`A`)` aufgeru-

fen werden. Wie in Aufgabe 1 wird der Nachbar wieder mit einem Default-Wert in `neighborFlashTimes` gespeichert.

Visualisierung

Wesentliche Unterschiede zu Aufgabe 1 liegen nur im Konstruktor `Firefly(String, int, int, int, int)` und in der `main()`-Methode. Die Klasse `Firefly` übernimmt nun die Rolle der Klasse `FireflySimulation` aus Aufgabe 1 und implementiert die Visualisierung im Konstruktor und in `paint(Graphics)`. Statt eines Rasters wird allerdings nur ein einziges blinkendes Rechteck erzeugt. Die Größe des Fensters und die gewünschten Koordinaten auf dem Bildschirm werden dem Konstruktor in Form der Argumente `s` (wie „size“), `x` und `y` übergeben. Dem Programm werden unter anderem ein Name und eine Portnummer mitgegeben, welche ebenfalls dem Konstruktor übergeben werden, damit das erzeugte Fenster diese im Titel tragen kann.

Das eigenständige Glühwürmchen

Die RMI-Initialisierung findet in der `main()`-Methode statt. Nachdem die Argumente des Programmaufrufs (`name, s, x, y, registryPort, neighborPort, neighborPort, ...`) abgefragt werden, wird ein `Firefly`-Objekt erstellt.

Dann wird der Serveranteil des Glühwürmchens, der zum Broadcasten genutzt wird, zum Laufen gebracht. Es wird ein Stub erzeugt, der unter dem mitgegebenen `registryPort` registriert wird.

Ist der Server erfolgreich gestartet, wird (aus Client-Perspektive) anhand der mitgegebenen `neighborPort`s eine Verbindung mit den anderen Glühwürmchen (=Servern) hergestellt, indem die Methode `registerNeighbor(FireflyInterface)` per RMI ausgeführt wird. Für jeden Nachbarn wird das solange probiert, bis es klappt. Hier gibt es wieder dasselbe Problem wie vorher bei der zweischrittigen Konstruktion eines Glühwürmchens; sie müssen erstmal existieren bzw. online sein, bevor man sich mit ihnen verbinden kann.

Besteht nun zu allen Nachbarn eine Verbindung, wird ein anonymer Thread gestartet, der die `run()`-Methode des Glühwürmchens als eigene implementiert.

Beispiel

Das Shell-Skript `runFireflies.sh` kann genutzt werden, um einen Torus mit 64 Glühwürmchen zu erstellen. Mit `killFireflies.sh` werden die Prozesse wieder geschlossen.