Software Architecture For Enterprises
Übung 1

Aufgabe 1

“Zaghafte erste Glühwürmchen” (Monolith – Multi-Threaded)

1. Ansatz

Der Ansatz der Glühwürmchen-Simulation basiert auf der Erstellung eines Programms, das mehrere Glühwürmchen als Threads in einer Torus-Struktur simuliert. Das Ziel war es, den Synchronisationsprozess der Glühwürmchen visuell darzustellen, wobei jedes Glühwürmchen durch ein farbiges (schwarzes) Rechteck repräsentiert wird.

2. Vorgehensweise

a. Modellierung der Glühwürmchen als Threads

- **Klasse Firefly**: Jedes Glühwürmchen wurde als Thread implementiert. Die Klasse `Firefly` implementiert das `Runnable` Interface, was es ermöglicht, jedes Glühwürmchen in einem eigenen Thread auszuführen.

```
public class Firefly implements Runnable {
     // Code zur Implementierung der Firefly-Klasse
}
```

- Die Phase jedes Glühwürmchens wird zufällig initialisiert, und der Zeitraum der Phase wurde auf 2.0 Sekunden gesetzt, um eine langsamere Synchronisation zu ermöglichen.

```
public Firefly(double x, double y) {
     rect = new Rectangle(x, y, 40, 40);
     phase = Math.random() * 2 * Math.PI;
     period = 2.0;
     rect.setFill(Color.BLACK);
}
```

b. Implementierung des Kuramoto-Modells

- Das Kuramoto-Modell zur Synchronisation wurde implementiert, indem die Phasen der Glühwürmchen entsprechend ihrer Nachbarn im Torus aktualisiert wurden. Die Nachbarn sind in der Umsetzung alle direkten Nachbarn, also: oben, unten, links und rechts neben dem Firefly.

```
private void syncWithNeighbors() {
      for (Firefly neighbor : neighbors) {
      double phaseDifference = neighbor.phase - this.phase;
      this.phase += COUPLING_STRENGTH *
                        Math.sin(phaseDifference);
      }
}
```

c. Erstellung der Benutzeroberfläche (GUI)

- Die GUI wurde mit JavaFX erstellt. Die Klasse `FireflySimulation` erweitert `Application` und erstellt eine Pane, auf der die Glühwürmchen als farbige Rechtecke angezeigt werden.

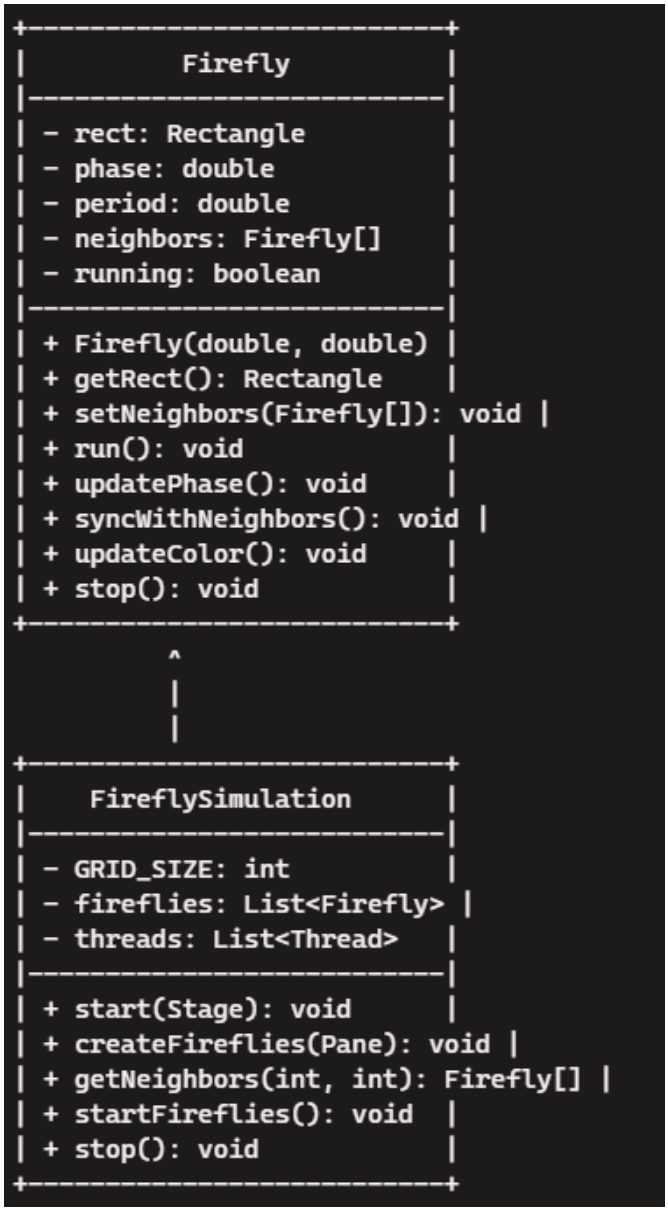
```
public class FireflySimulation extends Application {
      // Code zur Implementierung der GUI und Simulation
}
```

- Jedes Glühwürmchen wird als `Rectangle` dargestellt, dessen Helligkeit entsprechend der aktuellen Phase geändert wird.

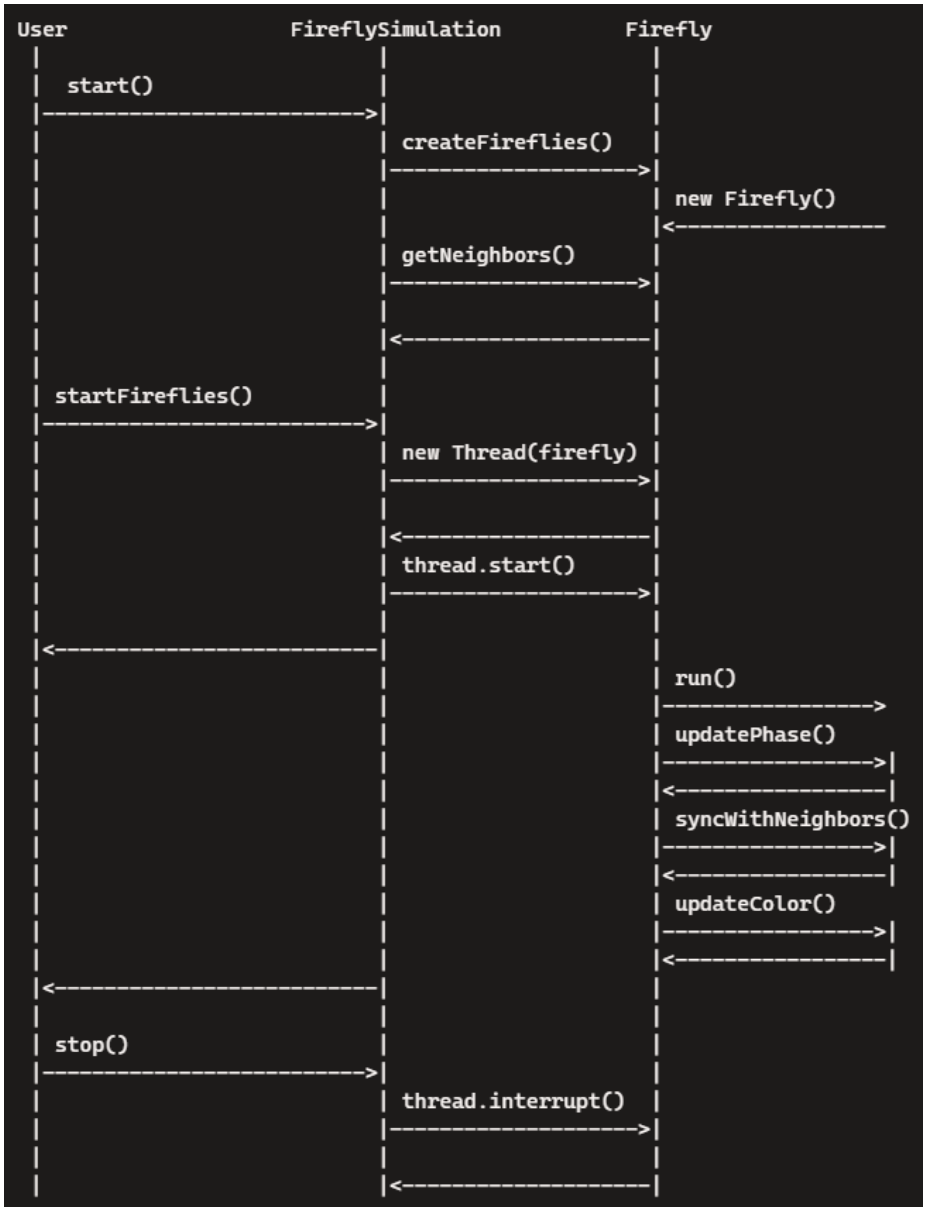
```
private void updateColor() {
      double brightness = (Math.sin(phase) + 1) / 2;
      javafx.application.Platform.runLater(() -> {
            rect.setFill(Color.gray(brightness));
      });
}
```

3. UML-Diagramm

Klassendiagramm:



Sequenzdiagramm:



Aufgabe 2

„Kommunizierende Glühwürmchen“ (Distributed – Java RMI)

1. Ansatz

Die verteilte Simulation der Glühwürmchen basiert darauf, jedes Glühwürmchen als eigenständiges Programm zu realisieren, das über ein Netzwerk mit den anderen Glühwürmchen kommuniziert. Ziel war es, den Synchronisationsprozess durch den Austausch von Zustandsinformationen zu steuern. Um die Glühwürmchen als eigenständige Programme zu realisieren, wurde für jedes ein eigener Server erstellt. Die Server wurden mit Java RMI erstellt.

2. Vorgehensweise

Ein direkte “Umwandlung” des Codes aus Aufgabe 1 ist mit Hilfe von KI nicht möglich gewesen. Deshalb war es nötig, den Aufbau zunächst von Grund auf neu zu erstellen.

Hierfür wurden zuerst die Server erstellt und anschließend randomisierte Phasen auf den Servern erstellt, die in der Konsole ausgegeben werden.

Anschließend wurde erneut die grafische Oberfläche mit JavaFX realisiert, auf der dann die Phasenwerte ausgegeben wurden, bevor es auf die Kacheln, wie in Aufgabe 1 umgestellt wurde.

Zuletzt wurde dann die Synchronisation mit den Nachbarn umgesetzt. Diese wird von der Klasse Multiserver angestoßen, die auch die Server erstellt.

Das fertige Programm sieht hatte dann folgenden Aufbau:

a. Modellierung der Glühwürmchen als eigenständige Programme

- Klasse MyRemoteService: Jedes Glühwürmchen wurde als eigenständiger RMI-Service implementiert. Die Klasse MyRemoteService ist ein Interface, das die Methoden definiert, die von den Remote-Glühwürmchen implementiert werden.

```
public interface MyRemoteService extends Remote {
        double getPhaseValue(long currentTime) throws RemoteException;
        void updateNeighborPhases(double[] neighborPhases) throws
                RemoteException;
        void updatePhase() throws RemoteException;
}
```

- Klasse MyRemoteServiceImpl: Diese Klasse implementiert das MyRemoteService Interface und stellt die Logik zur Aktualisierung und Synchronisation der Glühwürmchen-Phasen bereit.

```
public class MyRemoteServiceImpl extends UnicastRemoteObject implements
        MyRemoteService {
            private double phase;
            private double[] neighborPhases;
            private static final double COUPLING_STRENGTH = 0.05;
            private static final double TIME_STEP = 0.1;
            private static double period = 2.0;

        protected MyRemoteServiceImpl() throws RemoteException {
            super();
            this.phase = Math.random() * 2 * Math.PI;
            this.neighborPhases = new double[0];
        }
```

Die Phase jedes Glühwürmchens wird zufällig initialisiert, und der Zeitraum der Phase wurde auf 2.0 Sekunden gesetzt, um eine langsamere Synchronisation zu ermöglichen.

```
        protected MyRemoteServiceImpl() throws RemoteException {
            super();
            this.phase = Math.random() * 2 * Math.PI;
            this.neighborPhases = new double[0];
        }
```

b. Implementierung des Kuramoto-Modells

Das Kuramoto-Modell zur Synchronisation wurde implementiert, indem die Phasen der Glühwürmchen entsprechend ihrer Nachbarn im Torus aktualisiert wurden. Die Nachbarn sind in der Umsetzung alle direkten Nachbarn, also: oben, unten, links und rechts neben dem Glühwürmchen.

```
private void syncWithNeighbors() {
    for (double neighborPhase : neighborPhases) {
    double phaseDifference = neighborPhase - this.phase;
    this.phase += COUPLING_STRENGTH * Math.sin(phaseDifference);
    }
```

```
}
```

c. Erstellung der Benutzeroberfläche (GUI)

Die GUI wurde mit JavaFX erstellt. Die Klasse Simulation erweitert Application und erstellt eine Pane, auf der die Glühwürmchen als farbige Rechtecke angezeigt werden.

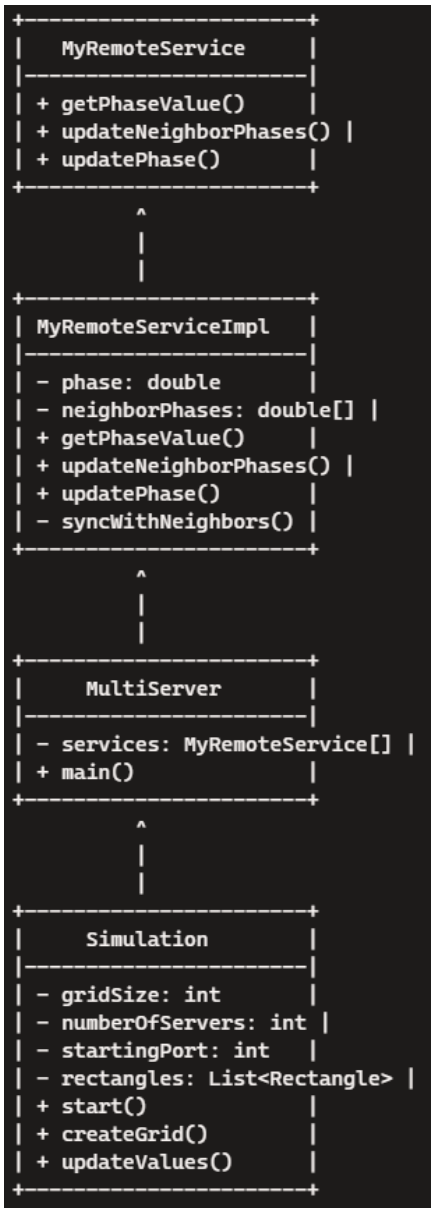
```
public class FireflySimulation extends Application {
    // Code zur Implementierung der GUI und Simulation
}
```

Jedes Glühwürmchen wird als `Rectangle` dargestellt, dessen Helligkeit entsprechend der aktuellen Phase geändert wird.

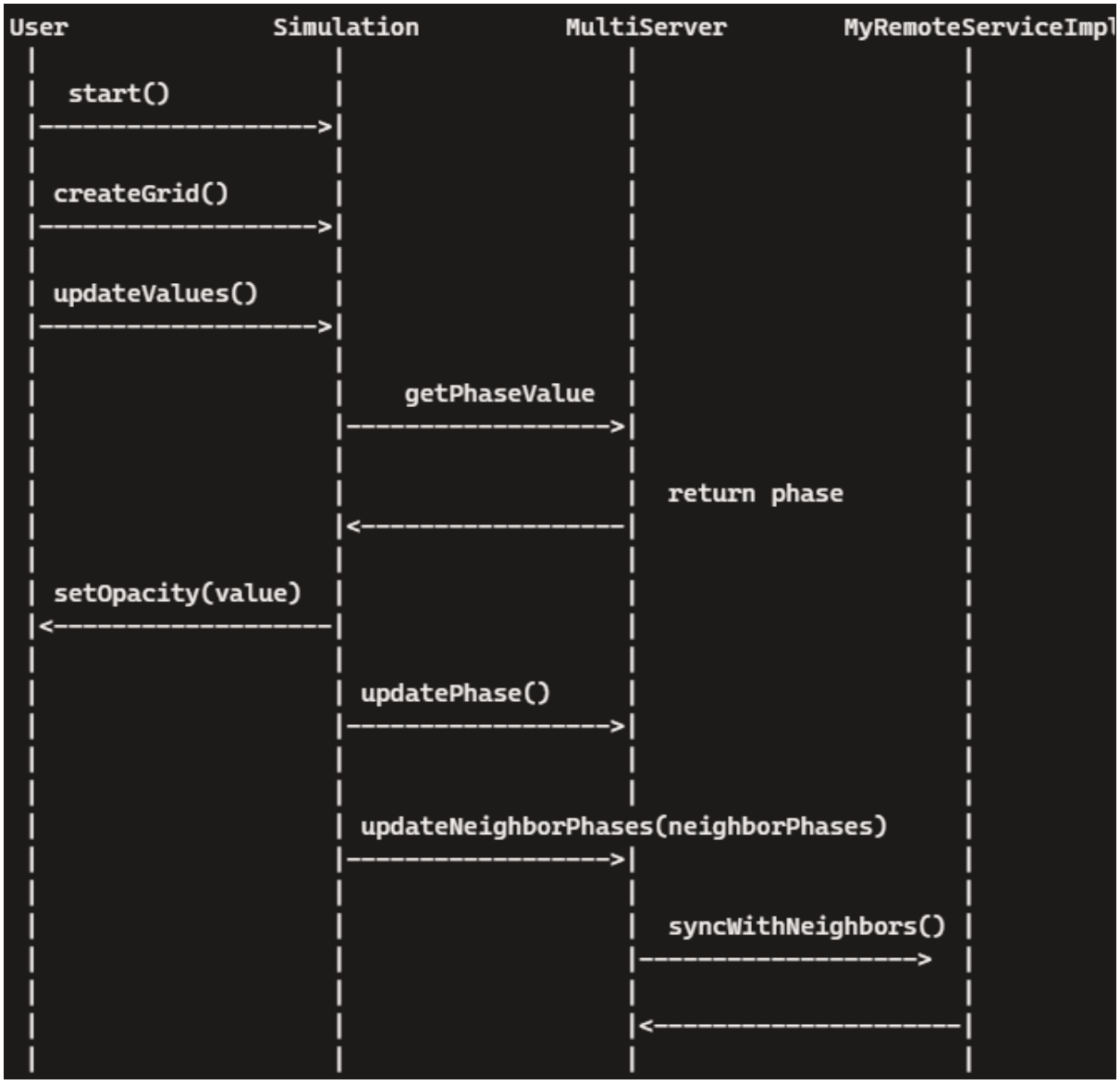
```
private void updateValues() {
    try {
        while (true) {
            long currentTime = System.currentTimeMillis();
            for (int i = 0; i < numberOfServers; i++) {
                int port = startingPort + i;
                MyRemoteService service = (MyRemoteService)
            Naming.lookup("rmi://localhost:" + port +
                                        "/MyRemoteService");
                double value = service.getPhaseValue(currentTime);
                int finalI = i;
                javafx.application.Platform.runLater(() ->
                    rectangles.get(finalI).setOpacity(value));
            }
        // Pause für 100 Millisekunden (0,1 Sekunden)
        Thread.sleep(100);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

3. UML

Klassendiagramm:



Sequenzdiagramm:



Überraschendes

Überraschend bei den Ergebnissen ist gewesen, dass es gelegentlich vorkam, dass die Fireflies nicht komplett synchron waren, sondern sich die “Phase” wie eine Welle von links nach recht oder von oben nach unten bewegt hat.

Besonders, wenn auch weniger überraschend, ist gewesen, dass die Phasen beim verteilten Ansatz nicht genauso gleichmäßig wurden, wie beim monolithischen Ansatz. Das lässt sich durch die Latenzen beim Abruf der Nachbarphase erklären, die dafür sorgen, dass die Phase nicht so gut synchronisiert werden kann, wie es bei dem Monolithen ist.