Portfolio Prüfung

1. Dezember 2024

Inhaltsverzeichnis

1 Multi-Threaded-Monolith (FireflyGroupSimulation auf Git)

1.1 Aufbau

Die Aufgabe bestand darin, eine Simulation von synchronisierten Glühwürmchen zu implementieren, die gemäß dem Kuramoto-Modell interagieren. Die Simulation nutzt eine Torus-Anordnung, in der jedes Glühwürmchen als Thread realisiert ist. Es gibt drei zentrale Klassen:

- **Firefly**: Stellt ein Glühwürmchen dar, das seine Phase durch Interaktion mit Nachbarn gemäß dem Kuramoto-Modell aktualisiert.
- **FireflyGrid**: Organisiert die Glühwürmchen in einer Torus-Anordnung und definiert Nachbarschaften.
- **Main**: Verantwortlich für die Benutzeroberfläche, die Glühwürmchen als Bilder visualisiert. (jeweils 1 jpeg für status *blinken* und status *nicht blinken*).

1.2 Erklärung der Klassen

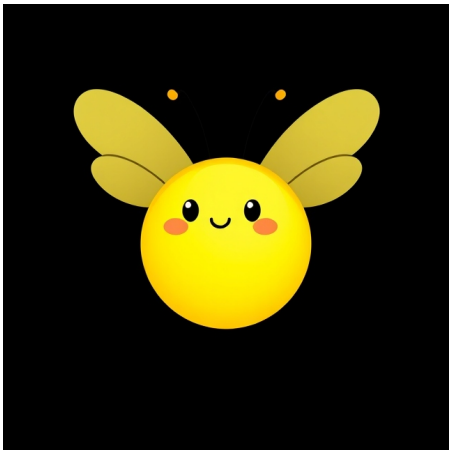
1.2.1 Firefly

Die Klasse `Firefly` implementiert das Verhalten eines einzelnen Glühwürmchens.

- Jedes Glühwürmchen besitzt eine Position (`x`, `y`), eine Eigenfrequenz und eine Phase, die seine Helligkeit bestimmt.
- Es wird ein Thread erstellt, der kontinuierlich die Phase des Glühwürmchens aktualisiert.
- Die Methode `updatePhase()` berücksichtigt die Phasen der Nachbarn und passt die Phase des aktuellen Glühwürmchens gemäß dem Kuramoto-Modell an:

$$\text{Neue Phase} = \text{Eigenfrequenz} + K \cdot \text{Einfluss der Nachbarn}$$

- Die Methode `getBrightness()` gibt die relative Helligkeit des Glühwürmchens zurück, basierend auf seiner Phase.



(a) Helles Glühwürmchen in der aktiven Phase.

(b) Dunkles Glühwürmchen in der inaktiven Phase.

Abbildung 1: Darstellung eines Glühwürmchens in zwei verschiedenen Phasen.

1.2.2 FireflyGrid

Die Klasse `FireflyGrid` erstellt ein Gitter von Glühwürmchen und definiert ihre Nachbarschaften.

- Jedes Glühwürmchen wird in einer Torus-Struktur angeordnet, d.h., die Ränder des Gitters sind miteinander verbunden.
- Die Methode `initializeGrid()` erzeugt die Glühwürmchen mit voreingestellten Phasen.
- Die Methode `setNeighbors()` definiert die Nachbarn jedes Glühwürmchens, wobei die Modulo-Arithmetik die Torus-Struktur gewährleistet.

1.2.3 Main

Die `Main`-Klasse ist für die Benutzeroberfläche und die Darstellung der Glühwürmchen zuständig.

- Mithilfe von Swing wird ein Fenster erstellt, das die Glühwürmchen in einem Gitter visualisiert.
- Die Methode `paintComponent(Graphics g)` zeichnet jedes Glühwürmchen als farbiges Rechteck, wobei das Bild von der Helligkeit abhängt.
- Threads werden gestartet, um die Phasen der Glühwürmchen parallel zu aktualisieren.

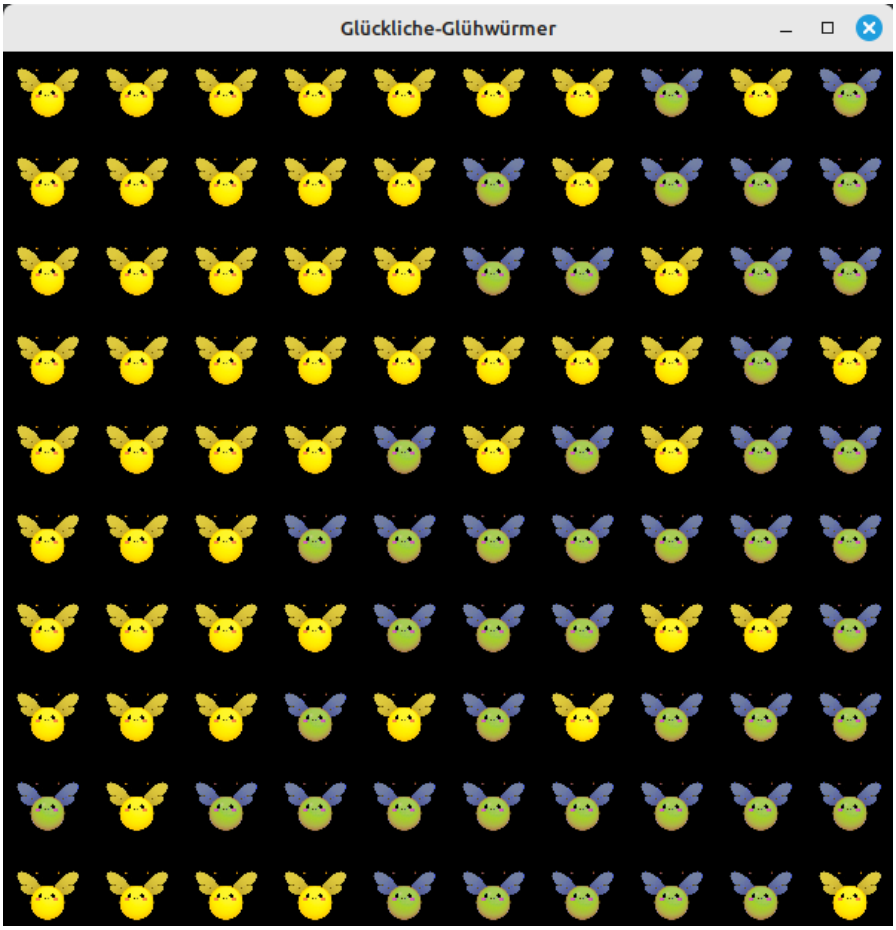


Abbildung 2: Torus-Anordnung der Glühwürmchen (10x10-Gitter, zyklische Verbindung).

1.3 Konzept

Die Eigenfrequenz (ω) eines Glühwürmchens beschreibt seinen natürlichen Rhythmus, in dem es zwischen hellen und dunklen Phasen wechselt. Die Phase (ϕ) gibt den aktuellen Zustand innerhalb dieses Zyklus an und bestimmt, wann das Glühwürmchen aufleuchtet. Durch die Interaktion mit Nachbarn, beeinflusst durch die Kopplungsstärke K, passt ein Glühwürmchen seine Phase an, sodass es sich nach und nach synchronisiert. Dabei nähert sich die Phase der Glühwürmchen einander an, bis sie gleichzeitig aufleuchten. Zum Start des Programm hat jedes Glühwürmchen eine Zufällige Phase, die sich mit der Zeit angleichen.

2 Verteiltes System (FireFlySimulation_2.0 auf Git)

2.1 Aufbau

Um die Synchronisation der Glühwürmchen dezentral zu simulieren, dienen die folgenden Strukturen: Es gibt Server, die jedes Glühwürmchen repräsentieren, und Clients, die dafür zuständig sind, Informationen zwischen den Servern auszutauschen. Alles basiert auf gRPC, was mir die Kommunikation zwischen den Prozessen erleichtert hat.

- **Server**: Dieser stellt ein Glühwürmchen dar und verwaltet seine Phase sowie Eigenfrequenz.
- **Client**: Der Client fragt die Phase der Nachbarn ab und sorgt dafür, dass die eigene Phase angepasst wird.

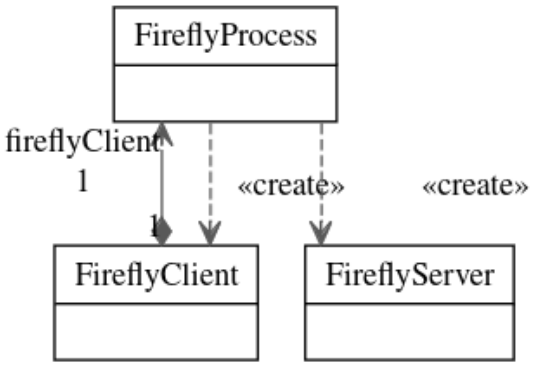


Abbildung 3: Programmstruktur

2.2 Kommunikation über gRPC

Über die nachfolgende *.proto* Datei, lassen sich die benötigten Java Klassen erzeugen welche später für die Client-Server Architektur benötigt werden. Dadurch erhalten wir die Funktionen zum Kommunizieren zwischen den Prozessen.

Proto-Datei

```
syntax = "proto3";

option java_package = "firefly.grpc";
option java_outer_classname = "FireflyProto";


service FireflyService {
  rpc notifyFirefly (FireflyRequest) returns (FireflyReply);
}

message FireflyRequest {
  double flash = 1;
}

message FireflyReply {
  double reply = 1;
}
```

2.2.1 Beispielkommandos zum Starten der Prozesse

Als Build-Script habe ich mich für Gradle entschieden, wobei man durch einen einfachen Befehl direkt Server und Client auf einmal starten kann. (In diesem Fall fungieren die Glühwürmer ja sowohl als auch). Die Synchronisation mit dem Kuramoto Modell verläuft Äquivalent zur ersten Aufgabe.

- **Glühwurm starten:**

  ```
  gradle runFirefly -Pports=5005,5001,5002
  ```

Mit diesem Befehl wird ein Glühwürmchen erstellt, welche auf Port 5005 läuft, und auf Port 5001 & 5002 lauscht. Man kann also direkt beim erstellen eine beliebige Anzahl an Ports festlegen, wobei der erste immer er eigene Port ist. Die restlichen Argumente sind die Ports der anderen Clients. Mit der notifyFirefly methode werden jeweils signale über die aktuelle eigene Phase an alle anderen Glühwürmchen gesendet. (In diesem fall an alle Ports die bei der Erstellung als Clients angegeben wurden) Für den einfacheren Gebrauch liegt ein Bash-Script im Projektordner mit dem Namen *Erstelle_Würmchen.sh*. Dieses nimmt ein Argument in Form einer einfachen Zahl und erstellt dann komplett automatisch die passende Anzahl an Glühwürmchen und kümmert sich um die passenden Port-Verbindungen.
Der Aufruf **./Erstelle_Würmchen 6** zum Beispiel ruft 6 FireflyProcess-Threads mit den Ports 5001,5002,5003, etc. Und lässt jeden der Ports auf jedem jeweils anderen lauschen.