Aflevering 5: Boids

Boids er en flokadfærdssimulator. Den demonstrerer små digitale livsformer, der opstår af simple regler i kombination med hinanden. Boids kigger på sine naboer og træffer en beslutning om hvor de skal bevæge sig hen. Naboer er de nærmeste boids inden for en fastdefineret radius.

Denne aflevering har følgende dele beskrevet længere nede:

- Delopgave 1: Strategy Pattern
- Delopgave 2: Lav en ny strategi
- Delopgave 3: Microbenchmark

Commit en fil der hedder RAPPORT.md hvor du meget kort beskriver, hvad du har gjort.

Blandt andet skal du skrive dine valg af parametre i microbenchmark'et.

Info: Bevægelse

Boids bevæger sig ud fra tre mønstre, man lægger sammen:

- **Separation:** En boid der er for tæt på sine naboer, søger væk fra dem.
- **Cohesion:** En boid der er for langt fra sine naboer, søger hen mod dem.
- **Alignment:** En boid der peger anderledes end sine naboer, skifter retning.

For hvert trin i simulationen udregnes bevægelse for hver boid og deres nærmeste naboer som resultat af hver af de tre kræfter.

Delopgave 1: Strategy Pattern

Den udleverede simulator har én type boid, Boid, som altid har faste parametre for de tre bevægelsesmønstre. Boid-klassen har en funktion, `calculateForces()` som kalder tre underfunktioner, `calculateSeparation()`, `calculateAlignment()` og `calculateCohesion()`.

Problemet er, at hvis man gerne vil udvide simulatoren til at understøtte flere typer boids som har andre påvirkninger, fx at nogle søger mad eller at nogle jager andre boids, så bliver Boid-klassen hurtigt meget rodet med kode der kun er aktivt i nogle tilfælde.

Du skal udvide simulatoren, så det er nemmere at tilføje forskellige slags boids med forskellige strategier.

Det skal du gøre ved at benytte interfacet `BehaviorStrategy`;

```
public interface BehaviorStrategy {
    Forces calculateForces(Boid boid, List<Boid> neighbors);
}
```

- Tag alt kode der har noget med `calculateForces()` (dvs. også de andre `calculate...()`-funktioner) og `getFlockWeights()` og flyt det fra Boid-klassen til en separat klasse der implementerer BehaviorStrategy-interfacet. Et godt navn til sådan en klasse kunne være FlockBehavior.
- Når du flytter funktionen `calculateForces()`, vil stedet den bliver kaldt selvfølgelig mangle den. Sørg i stedet for at gemme en objekt-instans af den klasse, den bliver flyttet til. Husk at bruge interfaces, når det giver mening.
- Selvom der kun er én klasse, kan du i princippet lave boids med forskellig opførsel, fordi FlockWeights ikke er hardcoded ind i hver enkelte Boid-instans.

Du kan betragte første delopgave som en refaktoriseringsopgave.

Delopgave 2: Lav en ny strategi

Når koden er omskrevet sådan at `calculateForces()` bor i en FlockBehavior-klasse der implementerer interfacet BehaviorStrategy, er det på tide at lave en klasse mere som også implementerer interfacet. Du må selv vælge hvad den skal gøre og hedde, men her er nogle forslag:

- RandomBehavior: I stedet for at bevæge sig i flok, ændrer den retning tilfældigt med nogle få grader. Denne adfærd kræver kun at man beregner en ændring i retning ved hjælp af trigonometri, men er ellers meget simpel fordi den ikke kræver at man ved noget om kortet inde i adfærdsklassen.
- AvoidanceBehavior: Foruden at bevæge sig i flok, søger boids med denne adfærd også væk fra bestemte objekter som er fast forankret på kortet. Denne adfærd kræver at du også placerer disse objekter, men er i øvrigt meget simpel, fordi objekterne ikke har andre regler eller bevæger sig.
- ForageBehavior: Foruden at bevæge sig i flok, søger boids også hen imod mad. Denne adfærd kræver at du også placerer mad-objekter rundt omkring på kortet, og evt. at de bliver fjernet når en boid flyver tæt nok på.

Der er ikke nogen konkrete krav til hvor svær den valgte strategi er, så længe der er mere end én, så man kan se forskel på simulationen, og så valg af strategi bliver en udskiftining af et objekt.

Info: Naiv nabosøgning

For at bestemme hvem der er naboer, skal man bruge en datastruktur, man kan søge i.

Hvis boids kun var 1-dimensionelle og altså bevægede sig frem og tilbage på en linje, så var det nok at sortere deres placering og vælge naboerne ved at bevæge sig lineært væk fra placeringen indtil vi er uden for den fastdefinerede radius. På den måde kan man slippe for at sammenligne boids der ligger endnu længere væk end boids man allerede har vurderet ligger uden for afstand.

Men boids er desværre 2-dimensionelle, så en simpel sortering er ikke nok.

Den naive tilgang er en liste:

```
for (Boid targetBoid : boids) {
    List<Boid> neighbors = new ArrayList<>();
    for (Boid otherBoid : boids) {
        if (targetBoid.getId() != otherBoid.getId()) {
            double dx = targetBoid.getX() - otherBoid.getX();
            double dy = targetBoid.getY() - otherBoid.getY();
            double distanceSquared = dx * dx + dy * dy;
            if (distanceSquared <= radiusSquared) {
                neighbors.add(otherBoid);
            }
        }
    }
    // targetBoid's neighbors found
}
```

Den naive tilgang vil medføre, at hver boid bliver sammenlignet med hver anden boid for at se om de er naboer, hvilket betyder $O(n^2)$ sammenligninger. Det påvirker hvor stor en simulation man kan køre, før man løber tør for CPU-kraft.

Spørgsmålet er så, om der findes en datastruktur som kan udelade nogle sammenligninger, ligesom en sorteret liste ville kunne, hvis boids var 1-dimensionelle. Og svaret er ja.

Fælles for de optimerende datastrukturer err, at de fungerer godt når boids er nogenlunde jævnt fordelt, men mister deres fordel hvis alle boids klumper sammen i ét hjørne, da alle så er naboer igen.

Info: Nabosøgning med K-d trees
K-d trees (k-dimensionelle træer) er en datastruktur der kan forbedre nabosøgning ved at organisere boids i et træ baseret på deres position.

Et K-d tree fungerer ved at opdele rummet rekursivt langs forskellige dimensioner:

- For 2D boids skifter man mellem at opdele langs x-aksen og y-aksen
- Hvert niveau i træet opdeler rummet med en linje gennem medianen af punkterne
- Bladknuder indeholder et lille antal boids (fx 1-10 stykker)

Når man søger efter naboer:

1. Start ved roden og gå ned gennem træet baseret på target-boid'ens position
2. Søg i det relevante område og tjek kun de grene der kan indeholde naboer inden for radius
3. Det reducerer antallet af sammenligninger fra $O(n^2)$ til $O(\log n)$ i gennemsnit

Info: Nabosøgning med Spatial Hashing
Spatial Hashing er en teknik der opdeler rummet i et gitter af firkantede celler og bruger hashing til hurtigt at finde naboer.

Sådan fungerer Spatial Hashing:

- Del hele spilleområdet op i et gitter af lige store kvadrater (fx 50x50 pixels hver)
- Hver boid bliver placeret i den celle dens koordinater tilhører
- Brug en hash-funktion til at mappe celle-koordinater til en hash-tabel
- For at finde naboer: tjek kun boids i samme celle og de 8 omkringliggende celler

Fordele og ulemper ved Spatial Hashing:

- Meget simpel at implementere sammenlignet med træer
- Celle-størrelsen skal vælges rigtigt (ca. 2x nabosøgnings-radius)
- Konstant tid $O(1)$ for at finde den rigtige celle

Info: Nabosøgning med Quad trees
Quad trees er en træstruktur der opdeler 2D-rummet i fire kvadranter rekursivt.

Sådan fungerer et Quad tree:

- Start med ét stort kvadrat der dækker hele spilleområdet
- Hvis et kvadrat indeholder for mange boids (fx mere end 10), del det op i 4 lige store kvadranter
- Fortsæt rekursivt indtil hvert kvadrat har få nok boids
- Hver knude i træet repræsenterer enten et område eller en leaf-knude med boids

Når man søger efter naboer:

1. Find hvilke kvadranter der overlapper med nabosøgnings-cirklen
2. Søg kun i disse relevante kvadranter
3. Tjek afstand kun til boids i de fundne kvadranter

Fordele ved Quad trees:

- Tilpasser sig automatisk til boid-fordelingen (tætte områder får mindre kvadranter)
- Effektiv til både jævnt fordelte og klumpede boids, men kan have dybere træer hvis boids klumper i hjørner
- Intuitivt at visualisere og debugge

Ulemper:

Delopgave 3: Microbenchmark

I denne delopgave skal du sammenligne performance af de forskellige datastrukturer:

- NaiveSpatialIndex
- KDTreeSpatialIndex
- QuadTreeSpatialIndex
- SpatialHashIndex

ved at lave et microbenchmark.

Lav en separat klasse kaldet `Microbench` ved siden af `Boids` med en main-metode der:

1. **Opretter en FlockSimulation**
2. **Vælger den rigtige SpatialIndex**
3. **Tilføjer et bestemt antal boids til simulationen**
4. **Kører simulationen et bestemt antal trin mens der tages tid**

Vælg antallet af boids, antallet af trin i simulationen, og nabosøgnings-radius og argumentér for dit valg.

Parametre at eksperimentere med

- **Antallet af boids**: Prøv forskellige størrelser for at se om alle datastrukturer er lige gode til alle antal boids
- **Antallet af iterationer**: Brug nok iterationer (mindst 100) til at få stabile målinger
- **Nabosøgnings-radius**: Prøv både små (20-30 pixels) og store (100+ pixels) radiuser

Måling

- Mål kun tiden for nabosøgning, ikke GUI-opdatering
- Brug `System.nanoTime()` før og efter hver iteration
- Beregn gennemsnitlig tid per iteration i millisekunder
- Overvej at "varme op" med nogle iterationer før tidsmålingen starter

Forventede resultater

- NaiveSpatialIndex: $O(n^2)$ - skal blive langsom ved mange boids
- De andre implementeringer: $O(n \log n)$ eller $O(n)$ - skal skalere bedre

Vigtige tips

- **Luk andre programmer** under benchmark for at undgå forstyrrelser
- **Kør hver test flere gange** og tag gennemsnittet
- **Deaktiver GUI** under benchmark hvis muligt (headless mode)
- **Gem resultaterne** i en tabel så du kan sammenligne