Programowanie Równoległe i Rozproszone
Projekt

Radosław Churski
Paweł Kurbiel

10 czerwca 2018

Spis treści

1 Temat projektu

Jako temat projektu wybraliśmy model symulacyjny prostego ruchu drogowego. Można w nim rozróżnić następujące podmioty:

samochód porusza się po drogach i skrzyżowaniach. Jeśli napotka przed sobą skrzyżowanie może na nie wjechać tylko wtedy, kiedy odpowiedni sygnalizator świetlny ma zapalone zielone światło. Samochód zna tylko najbliższe otoczenie.

skrzyżowanie jest wyposażone w dwa sygnalizatory świetlne, po jednym dla każdej z dwóch osi. Z założenia, samochody jadące na przeciw siebie "mają"to samo światło. Skrzyżowanie odpowiada za sterowanie sygnalizacją świetlną.

sygnalizacja świetlna może mieć jeden z dwóch kolorów, zielony lub czerwony.

mapa reprezentuje świat w którym poruszają się samochody i znajdują się drogi oraz skrzyżowania. Z mapy samochody dowiadują się o swoim najbliższym otoczeniu.

2 Realizacja

2.1 Uruchomienie; dane wejściowe

Program przyjmuje trzy argumenty:

- ścieżka do pliku z mapą
- liczba samochodów
- czas pracy programu

2.1.1 Plik z mapą

Można stwierdzić, że plik z mapą jest podzielony na dwie części. Jedną z nich jest pierwszy wiersza zawierający dwie liczby naturalne. Stanowią one informację o rozmiarach mapy.

Drugą częścią jest właściwa mapa. Rozróżnialne są trzy rodzaje znaków składające się na mapę:

- znak drogi (`R`)
- znak skrzyżowania (`C`)
- znak pustego pola (`.`)

Przykładowy plik z mapą:

```
5 10
RRRR......
...R......
RRRCRCRCRR
.....R.R.R
RRRRRR.RRR
```

Poniżej zamieszczony został fragment klasy `mapReader`:

```
private final static char roadChar = 'R';
private final static char crossingChar = 'C';

public static Map readMap(String path) {
    [...]
}
```

Atrybuty `roadChar` oraz `crossingChar` przechowują znaki, które podczas czytania pliku są interpretowane jako odpowiednio drogi i skrzyżowania. Każdy inny napotkany znak (oczywiście, oprócz końca linii) jest interpretowany jako puste pole. Takie rozwiązanie pozostawia możliwość na łatwe rozbudowanie obsługiwanych rodzajów pól mapy.

2.1.2 Liczba samochodów

Argument ten determinuje ile samochodów zostanie "wypuszczonych„ na drogi. Są one generowane w następujący sposób:

```
int numCars = Integer.parseInt(args[1]);
ArrayList<Road> roads = map.getallRoads();
Random random = new Random();

for (int i = 0; i < numCars; i++) {
    cars.add(new Car(
        i,
        roads.get(random.nextInt(roads.size())).getPosition(),
        new Velocity(random.nextInt(maxVelocity - minVelocity) + minVelocity),
        map)
    );
}
```

W pętli `for` dodajemy do listy `cars` kolejne samochody, przekazując do konstruktora kolejne liczby naturalne stanowiące identyfikatory samochodów, pozycję losowanych pól dróg, losowaną prędkość jako obiekt klasy `Velocity` oraz mapę. Klasa `Main` posiada statyczne atrybuty determinujące minimalną i maksymalną prędkość z jaką mają się poruszać samochody.

2.1.3 Czas pracy programu

Ostatnim parametrem przyjmowanym przez program jest informacja przez ile sekund ma pracować. Po tym, jak wszystkie potrzebne obiekty zostaną utworzone główny wątek programu jest usypiany na odpowiedni czas, po czym wywołuje metody terminujące na wszystkich obiektach pracujących z własnymi wątkami.

2.2 Implementacja samochodów

Samochody są reprezentowane za pomocą klasy `Car`. Jej konstruktor jako argumenty przyjmuje identyfikator samochodu, pozycję na mapie, prędkość oraz mapę do której będzie się odwoływał. Po ustaleniu atrybutów wywoływana jest metoda uruchamiająca wątek odpowiadający za asynchroniczne poruszanie się po mapie. Metoda `run` w odstępach czasowych zależnych od prędkości wywołuje `move` odpowiedzialną za kolejny ruch samochodu. Główna pętla chodzi, dopóki prywatny atrybut `stop` ma wartość negatywną.

```
public void run() {
    try {
        while (!stop) {
            move();
            Thread.sleep(velocity.getTimePerField());
        }
    } catch (InterruptedException e) {
        System.out.println("Interrupted");
    }
    System.out.println("Car " + id + " terminates on " + position.toString());
}
```

Metoda `move` zaczyna swoje działanie od pobrania otoczenia z mapy. Otoczenie to zbiór pól sąsiadujących z polem, na którym znajduje się samochód. Następnie wyodrębniane są te pola, na które samochód może się poruszyć - odrzucane są pola puste oraz pole na którym samochód był poprzednio. Jeśli nie ma pola, na które samochód mógłby przejechać, wypisuje informację o tym i nie porusza się w żadnym kierunku. W przeciwnym wypadku losuje pole na które się poruszy i wywołuje odpowiednią metodę, w zależności od tego, jakiego typu jest wylosowane pole (`moveToPosition` lub `moveToCrossing`). Warto zaznaczyć, że obecne rozwiązanie pozwala w łatwy sposób rozbudować program o dodanie nowych typów pól (np. pole zajęte przez samochód czy pole jednokierunkowe).

```
private void move() {
    MapItem[] surroundings = map.getSurroudings(position);
    ArrayList<MapItem> goable = new ArrayList<>();
    for (int i = 0; i < surroundings.length; i++) {
        if (surroundings[i] != null) {
            if (this.prevPosition == null)
                goable.add(surroundings[i]);
            else if (!surroundings[i].getPosition().equals(this.prevPosition))
                goable.add(surroundings[i]);
        }
    }
    if (goable.size() == 0) {
        System.out.println("[APP]" + toString() + " has no way to go.");
        return;
    }
    MapItem nexItem = goable.get(random.nextInt(goable.size()));
    switch (nexItem.getType()) {
        case ROAD: {
            moveToPosition(nexItem.getPosition());
            break;
        }
        case CROSSING: {
            moveToCrossing((Crossing) nexItem);
            break;
        }
    }
}
```

Ostatnią metodą z klasy `Car` którą chcielibyśmy omówić jest `moveToCrossing`, o użyciu której wspomniane zostało przy okazji metody `move`. Zadaniem tej metody jest zadbanie o to, aby samochód mógł „wjechać" na skrzyżowanie tylko, jeśli światło które „widzi" jest zielone. Najpierw uzyskiwana jest informacja ze skrzyżowania, jakiego koloru jest światło widziane z pozycji samochodu. Następnie, dopóki światło jest czerwone (i samochód nie został zatrzymany) metoda odczekuje 1 sekundę oraz ponownie sprawdza kolor światła. Po opuszczeniu pętli, samochód „wjeżdża" na skrzyżowanie.

```
private void moveToCrossing(Crossing crossing) {
    Color color = crossing.getColor(this.position);
    System.out.println(this.toString() + " met light " + color);
    while ((color == RED) && (!stop)) {
        try {
            System.out.println(this.toString() + " waits for green light.");
            Thread.sleep(1000);
            color = crossing.getColor(this.position);
        } catch (InterruptedException e) {
            System.out.println("[APP] Waiting for green light interrupted.");
        }
    }
    if (stop) return;
    moveToPosition(crossing.getPosition());
}
```

2.3 Implementacja skrzyżowań ze światłami

Najpierw przytoczmy konstruktor klasy `Light`. Do świateł przypisana jest oś, dzięki czemu wiadomo których samochodów stojących na skrzyżowaniu dotyczy dane światło. Pozostałe argumenty konstruktora to identyfikator i początkowy kolor świateł.

```
public Light(String id, Color color, Axis axis) {
    this.id = id;
    this.color = color;
    this.axis = axis;
}
```

Samo skrzyżowanie `Crossing` implementuje `Runnable` oraz dziedziczy po `MapItem` (`MapItem` to klasa posiadająca dwa atrybuty: `position` i `MapItemType` oraz gettery). Jej konstruktor wywołuje konstruktor nadklasy, tworzy *sobie* parę świateł, po jednej dla każdej osi, oraz uruchamia wątek zmieniający światła.

```
public Crossing(Position position) {
    super(position, MapItemType.CROSSING);
    lightX = new Light("X", RED, X);
    lightY = new Light("Y", GREEN, Y);
    this.start();
}
```

Metoda `run` zawiera jedną pętlę odpowiadającą za zmianę świateł. Czas pomiędzy zmianami koloru świateł zdefiniowany jest na prywatnym atrybucie klasy `Crossing`.

```
public void run() {
    try {
        while (!stop) {
            lightX.change();
            lightY.change();
            Thread.sleep(changeTime);
        }
    } catch (InterruptedException e) {
        System.out.println("Interrupted");
    }
    System.out.println(toString() + " terminates.");
}
```

Istotną metodą jest również `getColor` zwracająca kolor świateł widziany z pozycji `position` otrzymywanej jako argument. Metoda ta najpierw określa wspólną oś dla `position` oraz samego skrzyżowania a następnie zwraca kolor odpowiedniego światła.

```
public Color getColor(Position position) {
    Axis commonAxis = this.getPosition().getCommonAxis(position);
    if (commonAxis == X)
        return lightX.getColor();
    return lightY.getColor();
}
```

2.4 Wyjście

Wyjście programu można podzielić na 3 części:

- inicjalizacja
- wyjście wątków
- finalizacja Najpierw, w inicjalizacji, wypisana jest mapa wraz z wymiarami oraz informacje o stworzonych samochodach.

  Potem, w wyjściu wątków, są zapisy o działaniach pojazdów.

  Ostatnią częścią są informacje o zakańczaniu pracy przez samochody i skrzyżowania.

```
9x5
      R R R
      R   R
R R R C   R
R     R   R
R R R C R C

Car 0 on [3, 1] with velocity 1.157 f/s was created.
Car 1 on [0, 4] with velocity 0.569 f/s was created.
Car 2 on [5, 0] with velocity 0.899 f/s was created.
Car 2 moves from [5, 0] to [4, 0]
Car 1 moves from [0, 4] to [0, 3]
Car 0 moves from [3, 1] to [3, 0]
Car 1 moves from [0, 3] to [0, 2]
Car 2 moves from [4, 0] to [3, 0]
Car 1 moves from [0, 2] to [1, 2]
Car 0 moves from [3, 0] to [4, 0]
Car 1 moves from [1, 2] to [2, 2]
Car 2 moves from [3, 0] to [3, 1]
Car 1 on [2, 2] met light RED
Car 1 on [2, 2] waits for green light.
Car 0 moves from [4, 0] to [5, 0]
Car 2 on [3, 1] met light GREEN
Car 2 moves from [3, 1] to [3, 2]
Car 1 on [2, 2] waits for green light.
Car 0 moves from [5, 0] to [5, 1]
Car 2 moves from [3, 2] to [2, 2]
Car 1 on [2, 2] waits for green light.
Car 2 moves from [2, 2] to [1, 2]
Car 0 moves from [5, 1] to [5, 2]
Car 1 moves from [2, 2] to [3, 2]
Car 2 terminates on [1, 2]
Car 0 terminates on [5, 2]
Car 1 terminates on [3, 2]
Crossing at [5, 4] terminates.
Crossing at [3, 2] terminates.
Crossing at [3, 4] terminates.
```

3 Podsumowanie

Projekt ten dał szansę na realizację ciekawego tematu z wykorzystaniem wielowątkowości oraz programowania nastawionego na obiektowość. Ciekawie było obserwować wiele obiektów tych samych klas, kiedy natrafiały na różne warunki środowiskowe. Losowanie początkowych pozycji oraz prędkości pozwoliło uzyskać sytuacje, w których jedne samochody faktycznie czekały na światłach, żeby inne mogły „bezpiecznie" przejechać.

Temat który zrealizowaliśmy ma duży potencjał w kontekście interfejsu graficznego, ale niestety ze względu na dużą ilość pracy związaną z innymi przedmiotami zabrakło czasu na GUI.