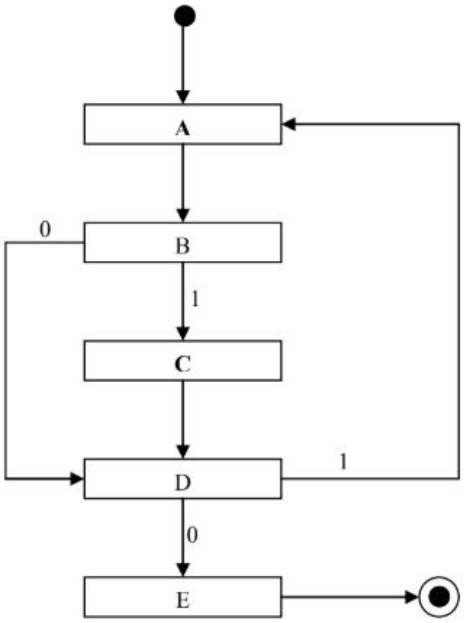
Klaudia Dziewit
Inżynieria Obliczeniowa
III rok gr. 1

Rozproszona sztuczna inteligencja - ćwiczenia 8

Wykonanie
Wykonanie zadania rozpoczęłam od utworzenia klasy agenta o nazwie *class12*. Agent ten wykonuje zachowanie, odwzorowujące następującą maszynę skończenie stanową:



Gdzie stany A, C i E polegają na wypisaniu nazwy stanu. Przejścia z tych stanów następują bezwarunkowo dalej. W stanach B i D również następuje wypisanie nazwy stanu, ale oprócz tego losowana jest liczba ze zbioru 0 i 1, która jest zwracana w chwili kończenia się zachowań związanych ze stanami.

W tym celu skorzystałam z zachowania FSMBehaviour, które oparte jest na harmonogramie potomstwa. Sami definiujemy zachowania przy użyciu konkretnych metod. Fragment kodu:

```
public class class12 extends Agent{

    private static final String STATE_A = "A";
    private static final String STATE_B = "B";
    private static final String STATE_C = "C";
    private static final String STATE_D = "D";
    private static final String STATE_E = "E";

    protected void setup() {
        FSMBehaviour fsm = new FSMBehaviour(this) {
            public int onEnd() {
                System.out.println("FSM behaviour completed.");
                myAgent.doDelete();
                return super.onEnd();
            }
        };

        fsm.registerFirstState(new NamePrinter(),  STATE_A);
        fsm.registerState(new RandomGenerator(1),  STATE_B);
        fsm.registerState(new NamePrinter(),  STATE_C);
        fsm.registerState(new RandomGenerator(1),  STATE_D);
        fsm.registerLastState(new NamePrinter(),  STATE_E);
        fsm.registerDefaultTransition( STATE_A,  STATE_B);
        fsm.registerTransition( STATE_B,  STATE_C, 1);
        fsm.registerTransition( STATE_B,  STATE_D, 0);
        fsm.registerDefaultTransition( STATE_C,  STATE_D);
        fsm.registerTransition( STATE_D, STATE_E, 0);
        fsm.registerTransition( STATE_D,  STATE_A, 1);

        addBehaviour(fsm);
    }

    private class NamePrinter extends OneShotBehaviour{
        public void action() {
            System.out.println("State name: " + getBehaviourName());
        }
    }

    private class RandomGenerator extends NamePrinter{
        private int maxExitValue;
        private int exitValue;

        private RandomGenerator(int max) {
            super();
            maxExitValue = max;
        }
        public void action() {
            super.action();
            exitValue = (int)(Math.round(Math.random())*maxExitValue);
            System.out.println("\tExit value is: "+exitValue);
        }
        public int onEnd() {
            return exitValue;
        }
    }
}
```

Przykład działania:

```
State name: A
State name: B
        Exit value is: 1
State name: C
State name: D
        Exit value is: 0
State name: E
FSM behaviour completed.
State name: A
State name: B
        Exit value is: 1
State name: C
State name: D
        Exit value is: 1
State name: A
State name: B
        Exit value is: 1
State name: C
State name: D
        Exit value is: 0
State name: E
FSM behaviour completed.
```

Kolejnym krokiem było utworzenie klasy o nazwie *class23*. Kod poprzedniego zachowania generycznego został zmodyfikowany tak, że zachowania wykonują się równolegle. W tym celu skorzystałam z *ParallelBehaviour*, które zapewnia równoległe wykonanie. Fragment kodu:

```
public class class23 extends Agent{
    protected void setup() {

        System.out.println("Starting");
        ParallelBehaviour par = new ParallelBehaviour();
        par.addSubBehaviour( new OneShotBehaviour()
        {
            public void action() {
                System.out.println("First step");
            }
        });

        par.addSubBehaviour( new OneShotBehaviour()
        {
            public void action() {
                System.out.println("Second step");
            }
        });

        par.addSubBehaviour( new OneShotBehaviour()
        {
            public void action() {
                System.out.println("Third step");
                removeBehaviour(par);
                System.out.println("Deleting");
            }
        });
        addBehaviour( par );
    }
}
```

Przykład działania:

```
Starting
First step
Third step
Deleting
```

Następnie należało wykonać 3 zachowania generyczne sekwencyjnie. W tym celu skorzystałam z zachowania *SequentialBehaviour*. Fragment kodu:

```
public class class24 extends Agent{
    protected void setup() {
        System.out.println("Starting");

        SequentialBehaviour threeStepBehaviour = new SequentialBehaviour();
        threeStepBehaviour.addSubBehaviour(new OneShotBehaviour()
        {
            public void action() {
                System.out.println("First step");
            }
        });

        threeStepBehaviour.addSubBehaviour(new OneShotBehaviour()
        {
            public void action() {
                System.out.println("Second step");
            }
        });

        threeStepBehaviour.addSubBehaviour(new OneShotBehaviour()
        {
            public void action() {
                System.out.println("Third step");
                removeBehaviour(threeStepBehaviour);
                System.out.println("Deleting");
            }
        });
        addBehaviour(threeStepBehaviour);
    }
}
```

Przykład działania:

```
Starting
First step
Second step
Third step
Deleting
```

Kolejnym krokiem było utworzenie klasy agenta *class25*, który wykonuje dwa zachowania cykliczne w dwóch osobnych wątkach. W tym celu również skorzystałam z *ParallelBehaviour.* Fragment kodu:

```java
public class class25 extends Agent{
    protected void setup() {

        ParallelBehaviour par = new ParallelBehaviour();
        par.addSubBehaviour(new CyclicBehaviour()
        {
            public void action() {
                System.out.println("Cyclic 1");
            }
        });

        par.addSubBehaviour( new CyclicBehaviour()
        {
            public void action() {
                System.out.println("Cyclic 2");
            }
        });

        addBehaviour(par);
    }
}
```

Przykład działania:

```
Cyclic 1
Cyclic 2
Cyclic 1
Cyclic 2
Cyclic 1
Cyclic 2
Cyclic 1
Cyclic 2
Cyclic 1
Cyclic 2
Cyclic 1
Cyclic 2
Cyclic 1
Cyclic 2
Cyclic 1
```