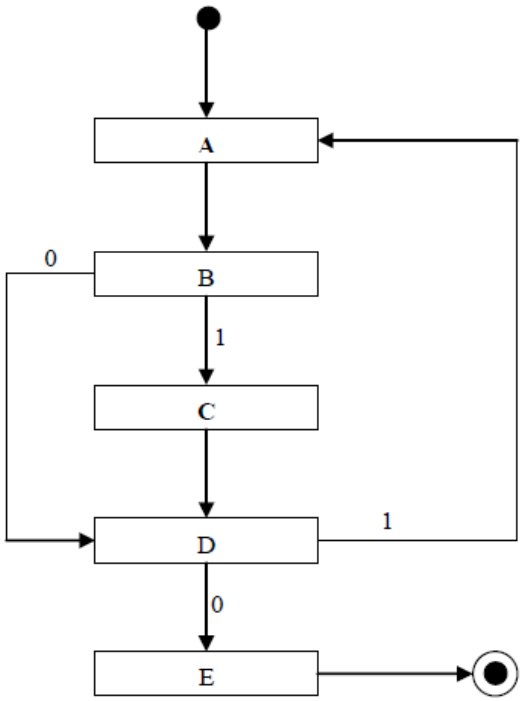
Fabiola Dąbroś

Inżynieria Obliczeniowa gr.1

Ćwiczenia 8 – Zachowania (2)

Wykonanie ćwiczenia rozpoczęłam od utworzenia klasy agenta o nazwie Klasa_1_2 . Agent ten wykonuje zachowanie, które odwzorowuje 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 skorzystamy z zachowania FSMBehaviour, które oparte jest na harmonogramie potomstwa. Sami definiujemy zachowania przy użyciu konkretnych metod.

Kod prezentuje się następująco:

```
public class Klasa_1_2 extends Agent{

    //nazwy stanów
    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();
        }
    };
    //rejestracja pojedynczego zachowania jako stan początkowy
    fsm.registerFirstState(new NamePrinter(),  STATE_A);
    //rejestracja jako stany pośrednie
    fsm.registerState(new RandomGenerator(1),  STATE_B);
    fsm.registerState(new NamePrinter(),  STATE_C);
    fsm.registerState(new RandomGenerator(1),  STATE_D);
    //rejestracja jako stan końcowy
    fsm.registerLastState(new NamePrinter(),  STATE_E);

    //przejście ze stanu do stanu niezależnie od
    //zdarzenia zakończenia stanu źródłowego
    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;
    }
}
```

Gdzie po uruchomieniu dostajemy różne wyjścia w zależności od wylosowanej liczby.

```
------------------------------------------
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 utworzeni klasy o nazwie Klasa_2_3 , która polegała na zmodyfikowaniu ostatniego kodu zachowania generycznego. Tym razem zachowania te wykonują się równolegle .

W tym celu skorzystamy z ParallelBehaviour, które zapewni nam równoległe wykonanie.

Kod prezentuje się następująco:

```
public class Klasa_2_3 extends Agent{
    protected void setup() {

        System.out.println("startuje");



        ParallelBehaviour par = new ParallelBehaviour();
        par.addSubBehaviour( new OneShotBehaviour()
        {
            public void action() {
                System.out.println( "krok pierwszy" );
            }
        });

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

        par.addSubBehaviour( new OneShotBehaviour()
        {
            public void action() {
                System.out.println( "krok trzeci" );
                removeBehaviour(par);
                System.out.println( "usuwam" );
            }
        });
        addBehaviour( par );
    }
}
```

W wyniku czego otrzymujemy:

```
------------------
startuje
krok pierwszy
krok trzeci
usuwam
```

Kolejno należało wykonać to trzy zachowania generyczne sekwencyjnie. W tym celu skorzystamy z zachowania SequentialBehaviour.

Kod prezentuje się następująco:

```
import jade.core.Agent;
public class Klasa_2_4 extends Agent{
    protected void setup() {
        System.out.println("startuje");

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

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

        threeStepBehaviour.addSubBehaviour( new OneShotBehaviour()
        {
            public void action() {
                System.out.println( "krok trzeci" );
                removeBehaviour(threeStepBehaviour);
                System.out.println( "usuwam" );
            }
        });
        addBehaviour(threeStepBehaviour);
    }
```

W wyniku czego otrzymujemy:

```
-------------------------------------------
startuje
krok pierwszy
krok drugi
krok trzeci
usuwam
```

Tym razem sekwencyjnie wykonały się wszystkie zachowania.

//

Kolejnym krokiem było otworzenie klasy agenta o nazwie Klasa_2_5 , który wykonuje dwa zachowania cykliczne w dwóch osobnych wątkach.

W tym celu również skorzystamy z zachowania ParallelBehaviour.

Kod prezentuje się następująco:

```
public class Klasa_2_5 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);
    }
}
```

W wyniku czego otrzymujemy:

```
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
```

Możemy zauważyć, że równolegle wykonują się obydwa zachowania.

Potwierdza nam to uruchomienie introspektora: