How to code a *FSM* with *statemachine API*?

SwingStates-like

```
package com.example.conversy.multitouch;


import fr.liienac.statemachine.event.Move;
import fr.liienac.statemachine.event.Press;
import fr.liienac.statemachine.event.Release;
import fr.liienac.statemachine.StateMachine;
//import java.awt.Component; //java2d

public class DragMachine extends StateMachine {

    public State start = new State() {
        Transition press = new Transition<Press>() {
            public State goTo() {
                return dragging;
            }
        };
    };

    public State dragging = new State() {
        Transition up = new Transition<Release>() {
            public State goTo() { return start; }
        };
        Transition move = new Transition<Move>() {
            public void action() {
                System.out.println("Drag " + evt.p.x + evt.p.y);
            }
        };
    };
}
```

Create a new class extending *StateMachine*

Create a state:
declare an attribute of type *State* **inside the** *StateMachine*

Create a transition:
declare an attribute of type *Transition* **inside the** *State*

Define the destination state of the transition:
redefine its method *goTo()*

Describe the actions to be performed during the transition: redefine its method *action()*

If you need a guard to activate the transition:
also redefine its method *guard()*

dérivé de:
Caroline Appert and Michel Beaudouin-Lafon. 2006. SwingStates: adding state machines to the swing toolkit. In Proceedings of the 19th annual ACM symposium on User interface software and technology (UIST '06). ACM, New York, NY, USA, 319-322.

How it works?

```java
/*
 * Copyright (c) 2016-2018 Stéphane Conversy - ENAC - All rights Reserved
 * Modified by Nicolas Saporito - ENAC (06/04/2017):
 */
package fr.liienac.statemachine;

import java.util.HashMap;
import java.util.Map;
import java.util.ArrayList;

import java.lang.reflect.ParameterizedType;

public class StateMachine {

    protected State current = null;
    protected State first = null;

    public State getCurrentState() { return current; }

    public <Event> void handleEvent(Event evt) {
        current.handleEvent(evt);
    }

    private void goTo(State s) {
        if (current != s) {
            current.leave();
            current = s;
            current.enter();
        }
    }
```

Pass all events to the state machine via its public method *handleEvent(…)*

```java
public class State {

    public State() {
        initialize();
    }

    private void initialize() {
        // first state is the initial state
        if (first == null) {
            first = this;
            current = first;
        }
    }

    protected void enter() {}
    protected void leave() {}

    Map<Object, ArrayList<Transition>> transitionsPerType = new HashMap<>();
    // with static type checking
```

…to specify if necessary in the concerned states, by redefining the associated methods.

If there is a state change, we can make any pre/post treatments…

… which redirects them to the method *handleEvent(…)* **of the current state…**

```java
// Hack for generics reflection : Type arguments to generic classes
        // are not available for reflection at runtime, unless...
        // they come from a generic superclass, so here it is
        private class MotherOfAllTransitions<EventT> {}

        public class Transition<EventT> extends MotherOfAllTransitions<EventT> {

            public Transition() {
                // generics reflection to get the specific event type from the generic type specified in this class
                ParameterizedType parameterizedType = (ParameterizedType) this.getClass().getGenericSuperclass();
                Class clazz = (Class) parameterizedType.getActualTypeArguments()[0];

                // register transition in state
                Transition t = this;
                ArrayList<Transition> ts = transitionsPerType.get(clazz);
                if (ts == null) {
                    ts = new ArrayList<>();
                    ts.add(t);
                    transitionsPerType.put(clazz, ts);
                } else {
                    ts.add(t);
                }
            }

            protected EventT evt;

            protected boolean guard() { return true; }
            protected void action() {}
            protected State goTo() { return current; }
        }

        protected <EventT> void handleEvent(EventT evt) {
            ArrayList<Transition> ts = transitionsPerType.get(evt.getClass());
            if (ts == null) return;
            for (Transition t : ts) {
                t.evt = evt;
                if (t.guard()) {
                    t.action();
                    StateMachine.this.goTo(t.goTo());
                    break;
}   }   }   }   }
```

…to specify if necessary in the concerned transitions by redefining the associated methods (which will have access to the event via the attribute `evt`**)**

If the guard is satisfied then activate the transition, i.e. perform any action and state change…

Reusability

The library `statemachine` is designed for genericity:

```
public <Event> void handleEvent(Event evt)
```

As a parametric type, `Event` represents any type.

We can therefore use any class as an "event" potentially triggering a transition:

- a real event from any library (*JavaFX*, *Swing*, *Android*...),
- or even better, an agnostic class representing an event.

The whole logic (of an interaction or of the state changes of a component) can thus be extracted and reused with several libraries.

Reusability

SwingStates-like

```
package com.example.conversy.multitouch;

import fr.liienac.statemachine.event.Move;
import fr.liienac.statemachine.event.Press;
import fr.liienac.statemachine.event.Release;
import fr.liienac.statemachine.StateMachine;
//import java.awt.Component; //java2d

public class DragMachine extends StateMachine {

    public State start = new State() {
        Transition press = new Transition<Press>() {
            public State goTo() {
                return dragging;
            }
        };
    };

    public State dragging = new State() {
        Transition up = new Transition<Release>() {
            public State goTo() { return start; }
        };
        Transition move = new Transition<Move>() {
            public void action() {
                System.out.println("Drag " + evt.p.x + evt.p.y);
            }
        };
    };
}
```

Agnostic classes representing events, provided by the library (but we can use anything else).

Agnostic events can be used/ created by the developper to replace events coming from the used library.
This indirection makes the whole logic described by the *FSM* completely agnostic, thus completely reusable in other projects, even with other libraries.

dérivé de:
Caroline Appert and Michel Beaudouin-Lafon. 2006. SwingStates: adding state machines to the swing toolkit. In Proceedings of the 19th annual ACM symposium on User interface software and technology (UIST '06). ACM, New York, NY, USA, 319-322.

Access to data conveyed by the "events"

```
package com.example.conversy.multitouch;

import fr.liienac.statemachine.event.Move;
import fr.liienac.statemachine.event.Press;
import fr.liienac.statemachine.event.Release;
import fr.liienac.statemachine.StateMachine;
//import java.awt.Component; //java2d

public class DragMachine extends StateMachine {

    public State start = new State() {
        Transition press = new Transition<Press>() {
            public State goTo() {
                return dragging;
            }
        };
    };

    public State dragging = new State() {
        Transition up = new Transition<Release>() {
            public State goTo() { return start; }
        };
        Transition move = new Transition<Move>() {
            public void action() {
                System.out.println("Drag " + evt.p.x + evt.p.y);
            }
        };
    };
}
```

SwingStates-like

Whatever its type, the "event" instance transmitted to the state machine via its method

`public <Event> void handleEvent(Event evt)`

can be accessed from the redefined methods of a transition thanks to its protected attribute `evt`**.**

The agnostic classes provided by the library are simple objets holding in public properties the information you need to make the state machine work:

- `Point p`
- `Item graphicItem`
- …

dérivé de:
Caroline Appert and Michel Beaudouin-Lafon. 2006. SwingStates: adding state machines to the swing toolkit. In Proceedings of the 19th annual ACM symposium on User interface software and technology (UIST '06). ACM, New York, NY, USA, 319-322.