Alfa 1.0 February 2013

A Control Flow Java library based on Apache Commons SCXML state machines.

Control Isolator:

When an application life cycle is designed, isolation between states is highly desired, in case of failure it should be possible to tell where the state the application was in. Due to the general nature of common OOB programming languages like C++ and JAVA there is no difference in algorithm control sentences and application flow control.

The purpose of this library is making it possible to define a state machine that defines the life cycle phases of an application or process allowing to isolate working phases separating flow into "Works", each of them capable of executing one or more activities.
As you have probably guessed already being faithful to the UML documentation of a project was in mind.

In essence what we will be controlling is an activity diagram that pauses each time a decision point is reached allowing us to evaluate the activities results at that point and determine which control path the lifecycle should follow by launching a control event. The event that is fired normally depends on:

- The results of the work done.
- There were errors, warnings or exceptions produced during the work activities.

In fact many activities be part of a single work, for example, an Initialization Work can include a "Logging activity" ,"Make backup" and "Prepare Values" ,
the work can be canceled if any of its activities fail. Regardless of the outcome, a flow control point will be reached and decisions taken from the results of the work, at that point it
can be determined to stop, continue, take a way or another depending on the success of the work done or its failure, making the application change its (lifecycle) state.

During the process the engine will have decision control points where flow decision logic should be executed. As a result of it an event will be generated that will determine the path in the lifecycle execution.

With the help of an automatic code generator the controller will be isolated from the

internal branching. Once defined, each lifecycle determines a control-flow-container.

The state machine is defined visually with the help of a WYSIWYG designer and stored in a common xml format:

State Chart XML (SCXML): State Machine Notation for Control Abstraction
http://www.w3.org/TR/scxml/

During execution it is possible to track the state of the application **visually** by connecting to a socket.

Under the hoods of the library the SCXML - Apache Commons library is being used to run the state machine. http://commons.apache.org/scxml/

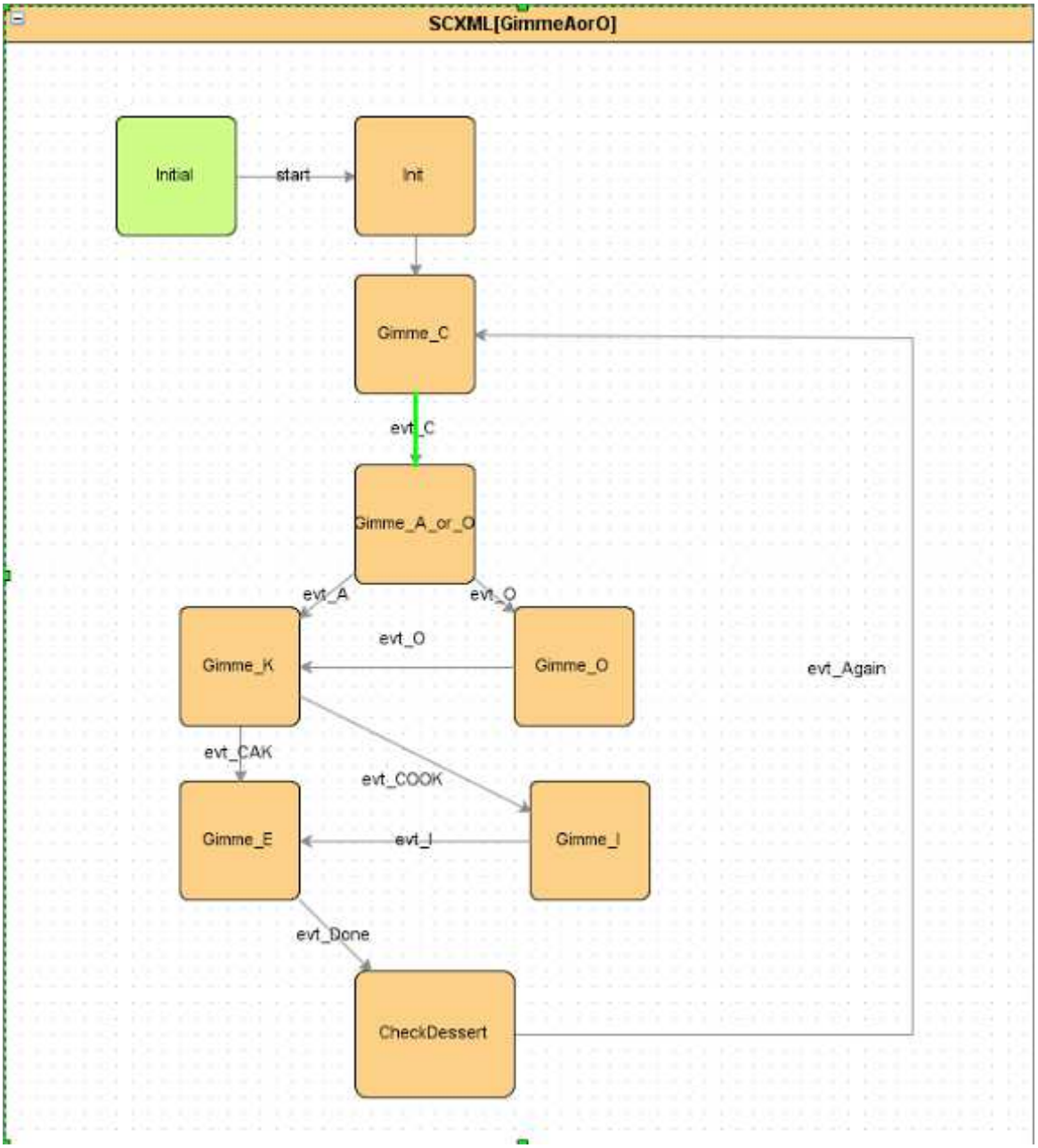
The state machine visual editor I normally use is:
scxmlgui https://code.google.com/p/scxmlgui/

Cake Or Cookie Example:

In this example a kitchen is supposed to have seven types of ingredients,
[C,I,K,E,O,A,S],
from them Cakes or Cookies can be produced. After the kitchen is prepared (inited), a C ingredient is taken , but until the second stage is reached the type of the dessert is unknown, at this stage a random ingredient is chosen, A or O which determines the path to make a CAKE or a COOKIE.

Obviously COIKAOOEIASS is not a valid dessert.

Control flow diagram:
The control diagram was designed with scxmlgui.



And its corresponding xml is not that complicated at all:

```
<scxml initial="Initial" name="GimmeAorO" version="0.9" xmlns="http://www.w3.org/2005/07/scxml"><!--   node-size-and-position x=0.0 y=0.0 w=650.0 h=720.0  -->
 <state id="Initial"><!--   node-size-and-position x=70.0 y=70.0 w=75.0 h=75.0  -->
  <transition event="start" target="Init"></transition>
 </state>
 <state id="Gimme_C"><!--   node-size-and-position x=220.0 y=170.0 w=75.0 h=75.0  -->
  <transition event="evt_C" target="Gimme_A_or_O"></transition>
 </state>
 <state id="Gimme_A_or_O"><!--   node-size-and-position x=220.0 y=290.0 w=75.0 h=75.0  -->
  <transition event="evt_A" target="Gimme_K"></transition>
  <transition event="evt_O" target="Gimme_O"></transition>
 </state>
 <state id="Gimme_K"><!--   node-size-and-position x=110.0 y=380.0 w=75.0 h=75.0  -->
```

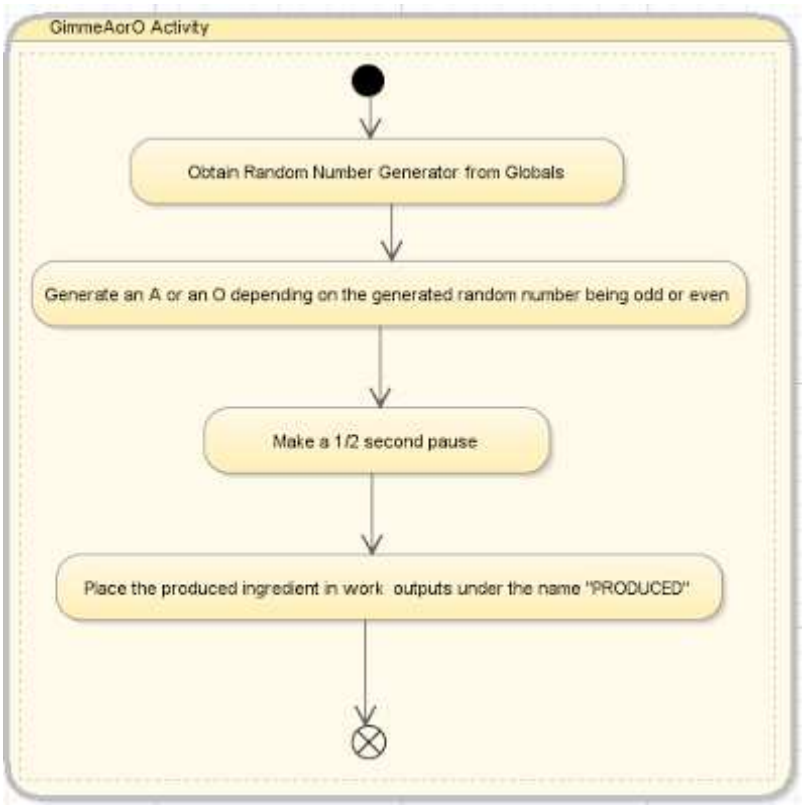
```
  <transition event="evt_CAK" target="Gimme_E"></transition>
  <transition event="evt_COOK" target="Gimme_I"><!--   edge-path [Gimme_I]  pointx=0.0 pointy=-19.0 offsetx=1.0 offsety=1.0  --></transition>
 </state>
 <state id="Gimme_O"><!--   node-size-and-position x=320.0 y=380.0 w=75.0 h=75.0  -->
  <transition event="evt_O" target="Gimme_K"><!--   edge-path [Gimme_K]  pointx=0.0 pointy=-17.0 offsetx=-2.0 offsety=0.0  --></transition>
 </state>
 <state id="Gimme_E"><!--   node-size-and-position x=110.0 y=490.0 w=75.0 h=75.0  -->
  <transition event="evt_Done" target="CheckDessert"></transition>
 </state>
 <state id="Gimme_I"><!--   node-size-and-position x=330.0 y=490.0 w=75.0 h=75.0  -->
  <transition event="evt_I" target="Gimme_E"></transition>
 </state>
 <state id="CheckDessert"><!--   node-size-and-position x=220.0 y=610.0 w=100.0 h=80.0  -->
  <transition event="evt_Again" target="Gimme_C"><!--   edge-path [Gimme_C]  x=570.0 y=650.0 x=570.0 y=210.0 pointx=0.0 pointy=40.0 offsetx=0.0 offsety=3.0  --
></transition>
 </state>
 <state id="Init"><!--   node-size-and-position x=220.0 y=70.0 w=75.0 h=75.0  -->
  <transition target="Gimme_C"></transition>
 </state>
</scxml>
```

The Initial State and the Init State:

Once the control flow is designed, we associate works to each state, being a work a collection of one or more activities. **The initial state does not carry associated work**. The Init state performs the following work, making it possible to put global-to-the-state-machine information available to all the states of that state machine:

```
public class InitWork extends StateWork{

    @Override
    public void doWork()  {
        _logger.log(Level.INFO,"[INIT]");

        this.getGlobals().put("RANDOM_GENERATOR", new Random());

    }

}
```

The GimmeAorO Work Activity:

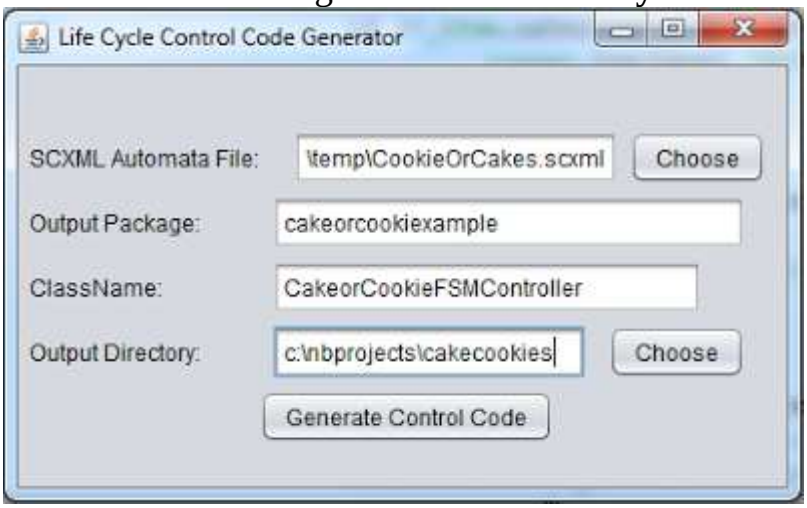


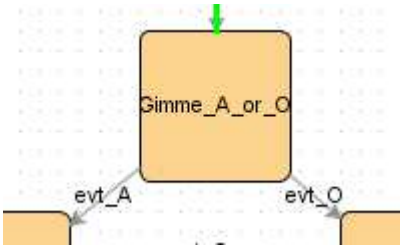
The corresponding JAVA code for this diagram is:

```
package cakeorcookiexample.works;

import controlisolation.lifecyclefsm.stateexecutor.StateWork;
import java.util.Random;
import java.util.logging.Level;

public class GenerateAorOWork extends StateWork{

    @Override
    public void doWork()  {
        _logger.log(Level.INFO,"[A or O]");

        Random random= (Random)this.getGlobals().get("RANDOM_GENERATOR");

        String produced="";

        if (random.nextInt()%2==0)
        {
            produced="A";
        } else
        {
            produced="O";
        }
```

```
        pauseFor(500);

        String previously_produced=(String)this.getWorkInputs().get("PRODUCED");
        this.getWorkOutputs().put("PRODUCED",previously_produced+produced);

    }

}
```

Control code is auto generated with **LifeCycleFSMControlCodeGeneratorGUI**:





The decissionControl method has to be hand-coded to represent the control flow logic, observe that the flow control logic is determined by the results of the work done, if an O was produced an evt_O is returned, on the other hand, if an A was produced, an evt_A is returned:

(part of CakeorCookieFSMController.java)

```
private String decissionControlFrom_Gimme_A_or_O(StateWork work) {
/* Possibe target states from this state:
| evt_O |
 will result in jumping to state: Gimme_O
| evt_A |
 will result in jumping to state: Gimme_K
Do event selection logic here */
        String produced=((String)work.getWorkOutputs().get("PRODUCED"));
```

```
        if (produced.endsWith("O"))
            return "evt_O";
        if (produced.endsWith("A"))
            return "evt_A";
return "ERROR";
}
```

LifeCycle State Machine Implementation:

One the works and control have been coded, a LifeCycleFSM _lifeCycleFSM should be created indicating the control scxml file, assigning each work to its corresponding
state and binding the control flow logic controller:

```
public class CakeOrCookieFSM
{

    private String  _fsmSCXMLPath="c:/temp/CookieOrCakes.scxml";
    private LifeCycleFSM _lifeCycleFSM;

    public void init() throws Exception
    {

        File f=new File(_fsmSCXMLPath);
        if (!f.exists())
        {
            throw new FileNotFoundException(_fsmSCXMLPath);
        }

        URL url= f.toURI().toURL();

        setLifeCycleFSM(new LifeCycleFSM(url));
        getLifeCycleFSM().setWorkForState("Init",new InitWork());
        getLifeCycleFSM().setWorkForState("Gimme_C",new GenerateCWork());
        getLifeCycleFSM().setWorkForState("Gimme_A_or_O",new GenerateAorOWork());
        getLifeCycleFSM().setWorkForState("Gimme_O",new GenerateOWork());
        getLifeCycleFSM().setWorkForState("Gimme_K",new GenerateKWork());
        getLifeCycleFSM().setWorkForState("Gimme_I",new GenerateIWork());
        getLifeCycleFSM().setWorkForState("Gimme_E",new GenerateEWork());
        getLifeCycleFSM().setWorkForState("CheckDessert",new CheckDessertWork());

        getLifeCycleFSM().bindTo(new CakeorCookieFSMController());

    }


     public void start()
     {
        getLifeCycleFSM().start();
```

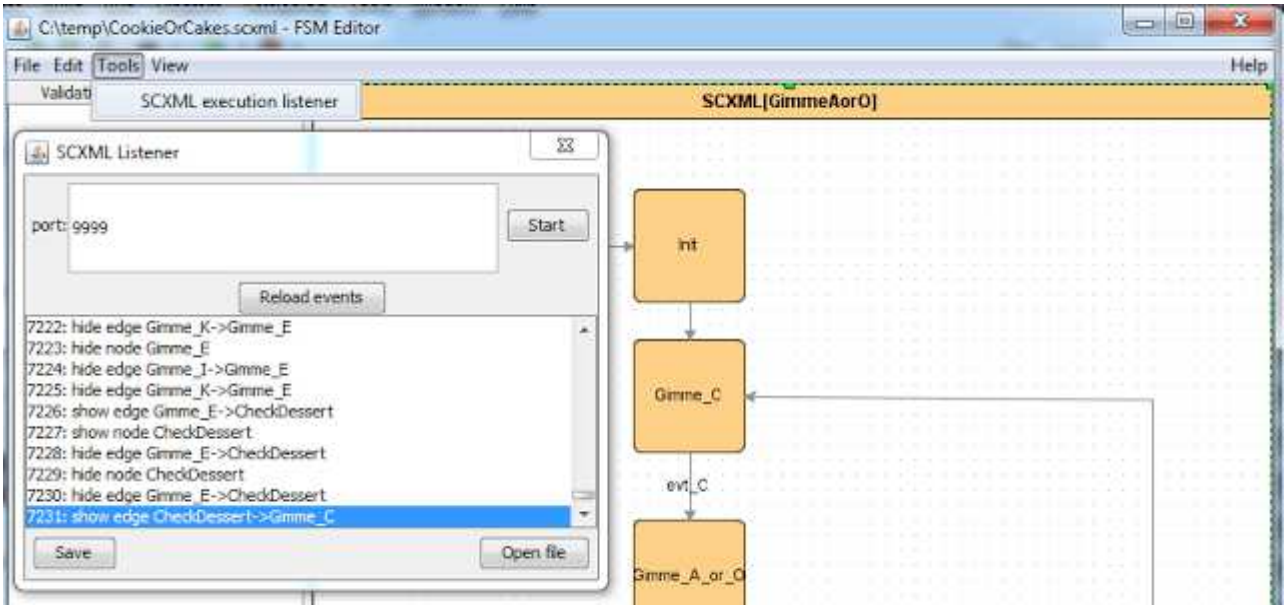
```
    }


    public LifeCycleFSM getLifeCycleFSM() {
        return _lifeCycleFSM;
    }

    public void setLifeCycleFSM(LifeCycleFSM lifeCycleFSM) {
        this._lifeCycleFSM = lifeCycleFSM;
    }


public static void main(String[] args) throws Exception
    {
        CakeOrCookieFSM fsm=new CakeOrCookieFSM();


        fsm.init();
        fsm.getLifeCycleFSM().setDebugMessages(true);
        fsm.getLifeCycleFSM().startRemoteDebugServer(9999);
        fsm.start();
    }


}
```

Loggin , Event Listening and Remote Viewing:

Life Cyle Work's events can be listened by the classes that implement:
WorkStartedEventListener,
WorkFinishedEventListener,
WorkFailedEventListener,
WorkWarningsEventListener.

Work processing loggin can be activated by performing a call to setDebugMessages(true); on the corresponding LifeCycleFSM.

A scxml remote debugger may be launched indicating the port where the messages will be published:
startRemoteDebugServer(9999);



Once done, the state flow can be visually followed real time.