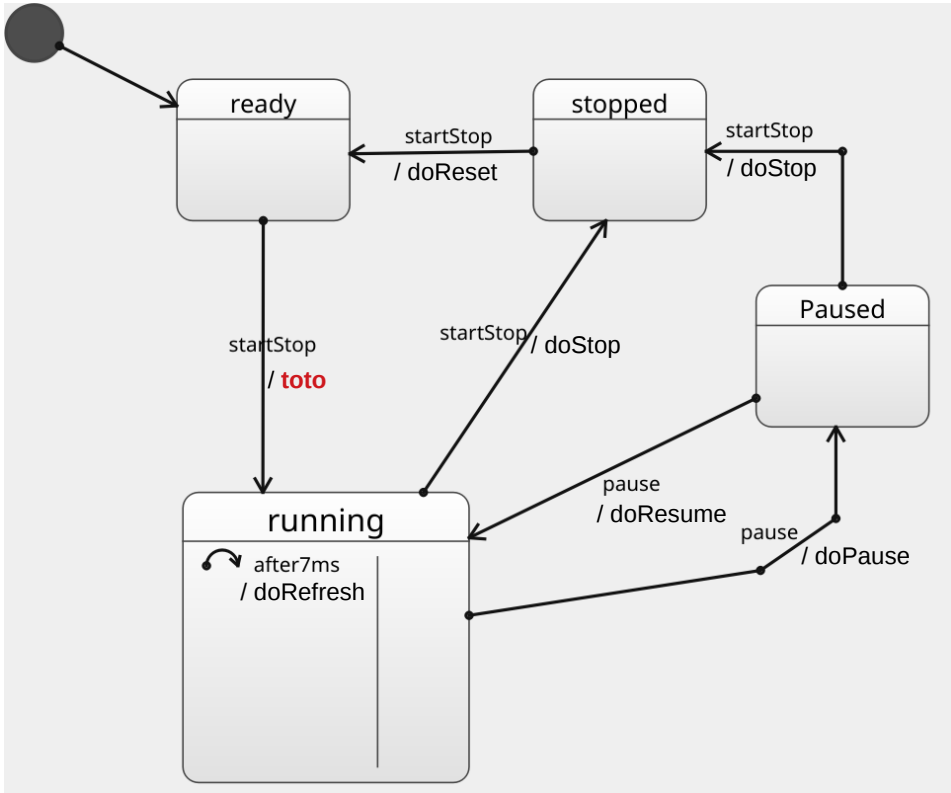
SCXML
State Chart XML

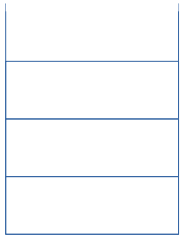
Détails et implémentations

FSM network

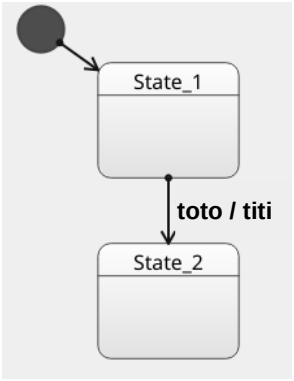
statecharts = state-diagrams + depth

+ orthogonality + broadcast-communication.



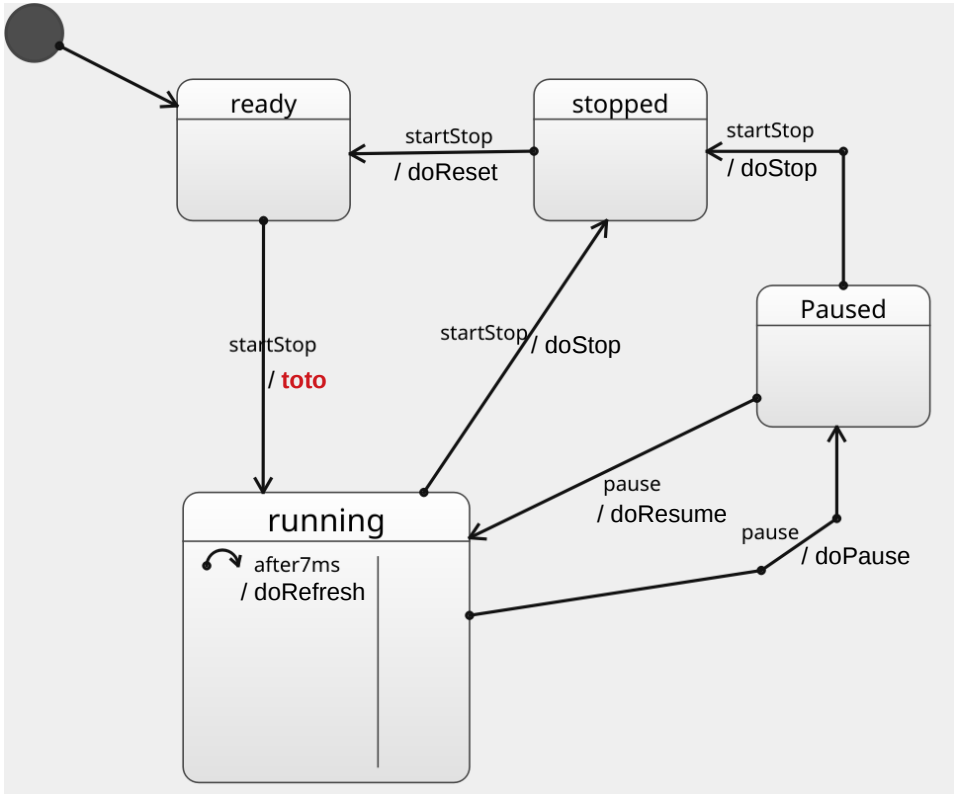


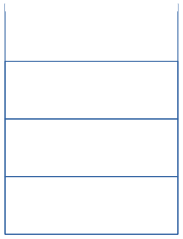
external queue



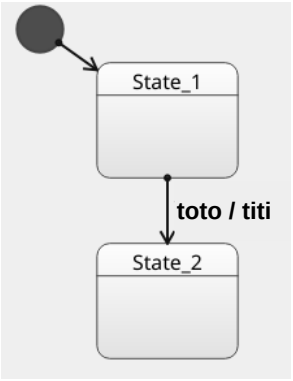
FSM network

```
controller.connectToEvent(QStringLiteral("toto"),this, &StopWatch::generateTotoToTester);

void StopWatch::generateTotoToTester(){
    tester.submitEvent("toto");
}
```



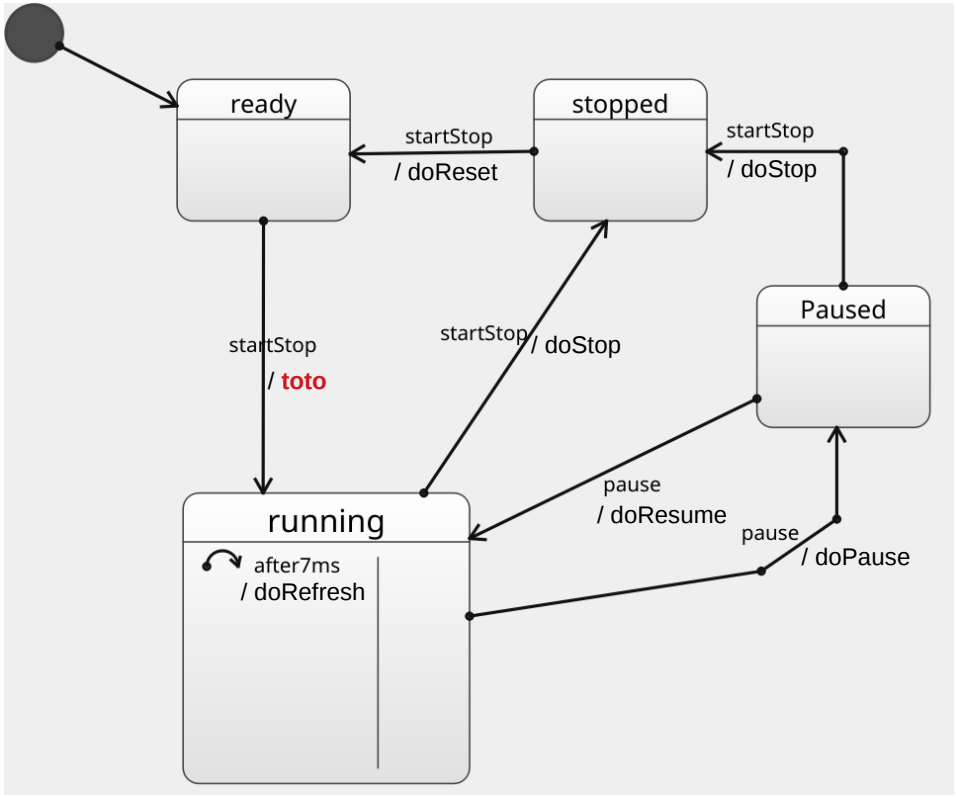


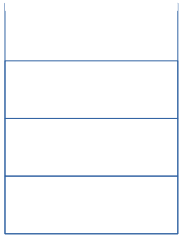
external queue

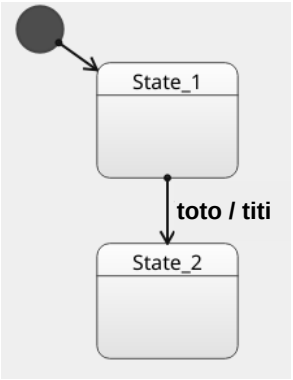


FSM network

```
controller.connectToEvent(QStringLiteral("toto"),this, &StopWatch::generateTotoToTester);

void StopWatch::generateTotoToTester(){
    tester.submitEvent("toto");
}
```







It is (easily) possible to obtain the language resulting from such network of FSM

Écrire/générer du code implémentant un State chart

- Un state chart “simple” s’implémente simplement.
- On trouve généralement trois manières d’implémenter un state chart[1]:
  - Une implémentation procédurale “à base de Switch” ou chaque “case” correspond à un état
  - Une implémentation basée sur une représentation tabulaire des state charts.
  - Une implémentation orientée objet tirant partie du polymorphisme et du typage dynamique

[1] *Il en existe une infinité de variante et de croisement, mais ce sont les plus fréquentes*

Écrire du code implémentant un State chart

- Un state chart “simple” s’implémente simplement.

Écrire du code implémentant un State chart

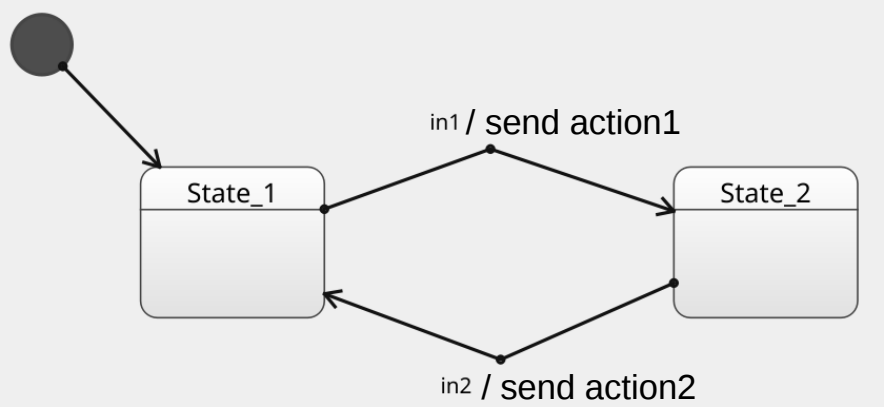
- Un state chart “simple” s’implémente simplement.
- Il peut être nécessaire/intéressant de faire des choix de “simplification” (exemple, sending action1 implique d’appeler la fonction action1())
- SCXML propose de nombreuse fonctionnalités de modélisation et de nombreuses subtilités dans la sémantique sous jacente.
- Il est important de savoir quelles sont les constructions supportées par le code que l’on veut écrire/générer.

Écrire du code implémentant un State chart

- Un state chart “simple” s’implémente simplement.
- On trouve généralement trois manières d’implémenter un state chart[1]:
  - Une implémentation procédurale “à base de Switch” ou chaque “case” correspond à un état
  - Une implémentation basée sur une représentation tabulaire des state charts.
  - Une implémentation orientée objet tirant partie du polymorphisme et du typage dynamique

[1] Il en existe une infinité de variante et de croisement, mais ce sont les plus fréquentes

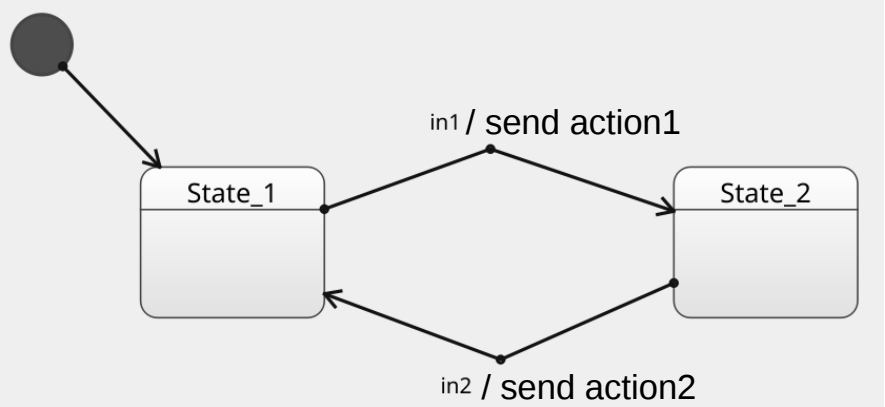
Écrire du code implémentant un State chart



Switch case

```
enum State {State_1, State_2};
enum Event {in1, in2};
```

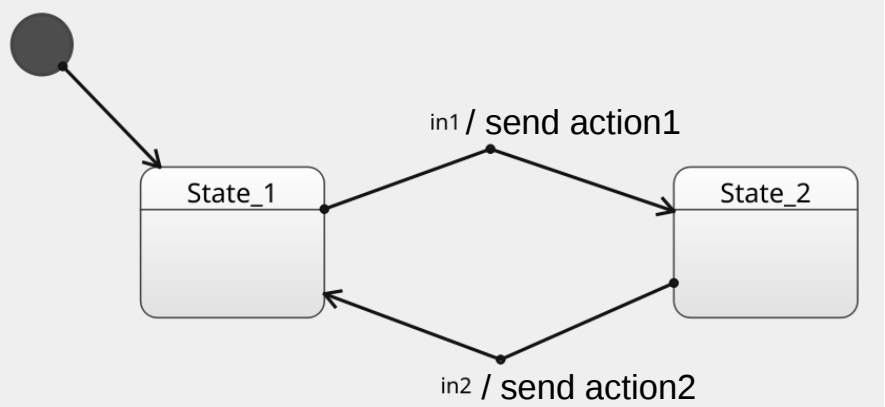
Écrire du code implémentant un State chart



Switch case

```
enum State {State_1, State_2};
enum Event {in1, in2};

void action1(){
    std::cout << "action 1 " << std::endl;
}
void action2(){
    std::cout << "action 2 " << std::endl;
}
```

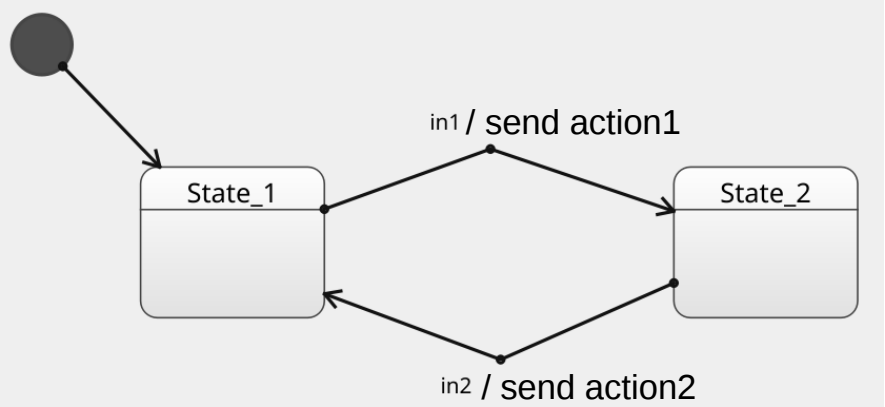
Écrire du code implémentant un State chart



Switch case

```
enum State {State_1, State_2};
enum Event {in1, in2};

void action1(){
    std::cout << "action 1 " << std::endl;
}
void action2(){
    std::cout << "action 2 " << std::endl;
}

State currentState = State_1;
```

Écrire du code implémentant un State chart

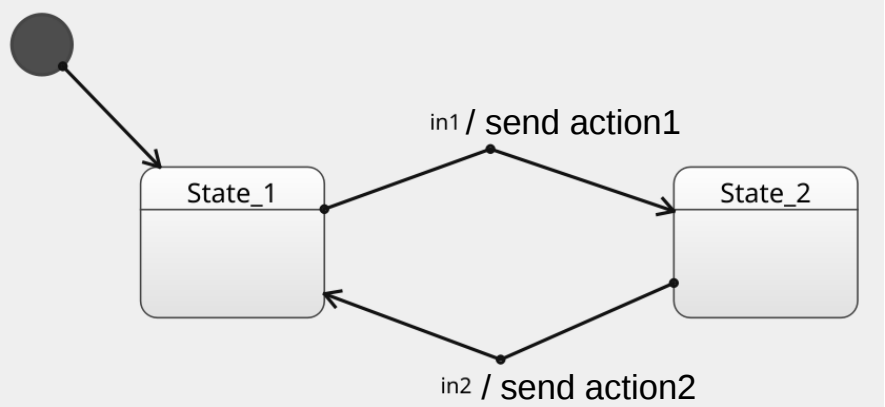


Switch case

```
enum State {State_1, State_2};
enum Event {in1, in2};

void action1(){
    std::cout << "action 1 " << std::endl;
}
void action2(){
    std::cout << "action 2 " << std::endl;
}

State currentState = State_1;
```

```
int activate(Event newEvent){

}
```

Écrire du code implémentant un State chart



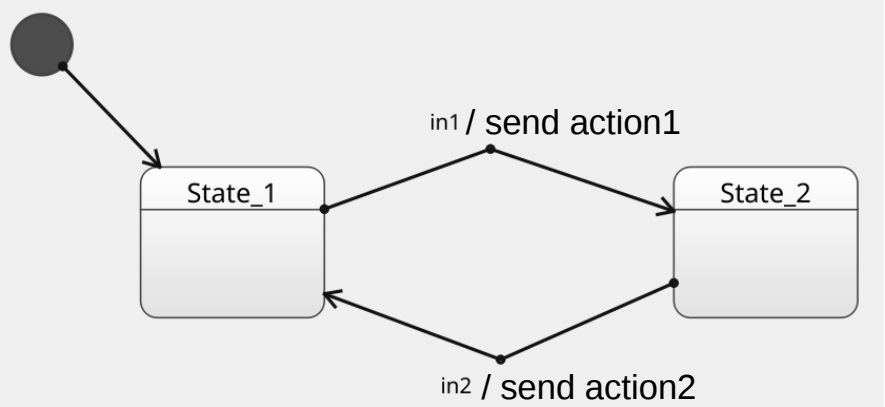
Switch case

```
enum State {State_1, State_2};
enum Event {in1, in2};

void action1(){
    std::cout << "action 1 " << std::endl;
}
void action2(){
    std::cout << "action 2 " << std::endl;
}

State currentState = State_1;
```

```
int activate(Event newEvent){
    switch(currentState){
        case State_1:
            if (newEvent == in1){
                action1();
                currentState = State_2;
                std::cout << "enter State_2" << std::endl;
            }
```

```
}
```

Écrire du code implémentant un State chart

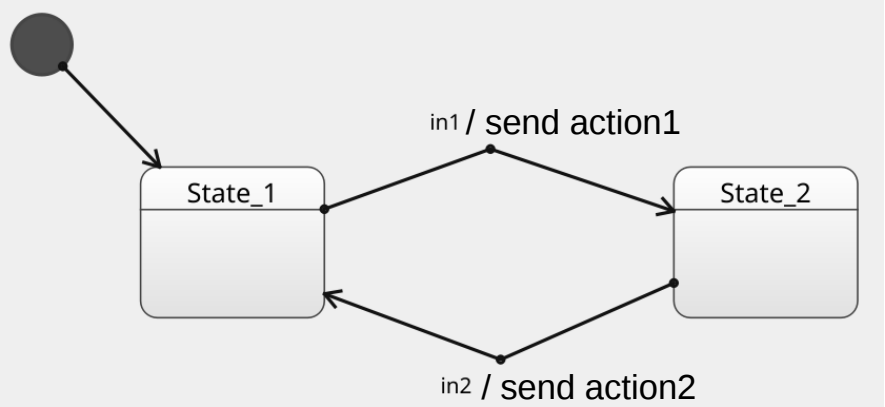


Switch case

```
int activate(Event newEvent){
    switch(currentState){
        case State_1:
            if (newEvent == in1){
                action1();
                currentState = State_2;
                std::cout << "enter State_2" << std::endl;
            }
            break;
        case State_2:
            if (newEvent == in2){
                action2();
                currentState = State_1;
                std::cout << "enter State_1" << std::endl;
            }
            break;
    }
}
```

```
enum State {State_1, State_2};
enum Event {in1, in2};

void action1(){
    std::cout << "action 1 " << std::endl;
}
void action2(){
    std::cout << "action 2 " << std::endl;
}

State currentState = State_1;
```

Écrire du code implémentant un State chart



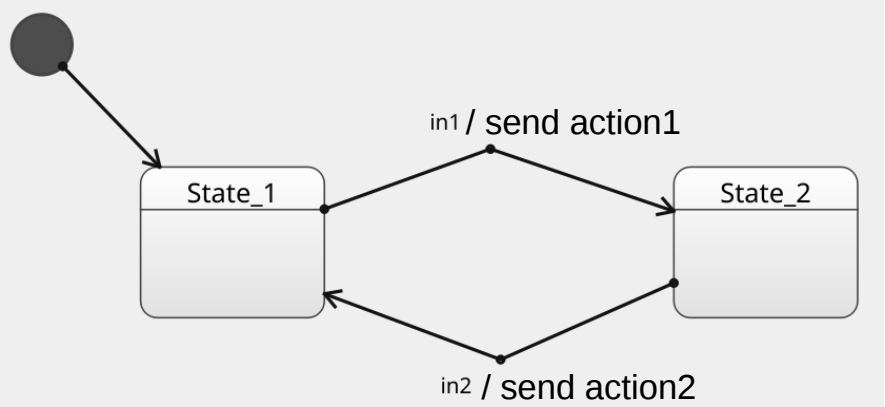
Switch case

```
enum State {State_1, State_2};
enum Event {in1, in2};

void action1(){
    std::cout << "action 1 " << std::endl;
}
void action2(){
    std::cout << "action 2 " << std::endl;
}

State currentState = State_1;
```

```
int activate(Event newEvent){
    switch(currentState){
        case State_1:
            if (newEvent == in1){
                action1();
                currentState = State_2;
                std::cout << "enter State_2" << std::endl;
            }
            break;
        case State_2:
            if (newEvent == in2){
                action2();
                currentState = State_1;
                std::cout << "enter State_1" << std::endl;
            }
            break;
    }
}
```

```
int main(){
  activate(in1);
  activate(in1);
  activate(in2);
}
```

Écrire du code implémentant un State chart




```
int activate(Event newEvent){
    switch(currentState){
        case State_1:
            if (newEvent == in1){
                action1();
                currentState = State_2;
                std::cout << "enter State_2" << std::endl;
            }
            break;
        case State_2:
            if (newEvent == in2){
                action2();
                currentState = State_1;
                std::cout << "enter State_1" << std::endl;
            }
            break;
    }
}
```

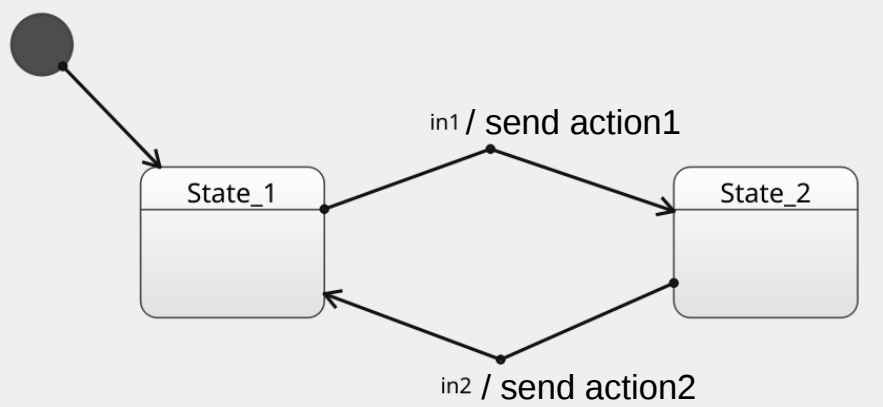
Switch case

```
enum State {State_1, State_2};
enum Event {in1, in2};

void action1(){
    std::cout << "action 1 " << std::endl;
}
void action2(){
    std::cout << "action 2 " << std::endl;
}

State currentState = State_1;
```

```
int main(){
  activate(in1);
  activate(in1);
  activate(in2);
}
```

```
action 1
enter State_2
action 2
enter State_1
```

Écrire du code implémentant un State chart



Switch case

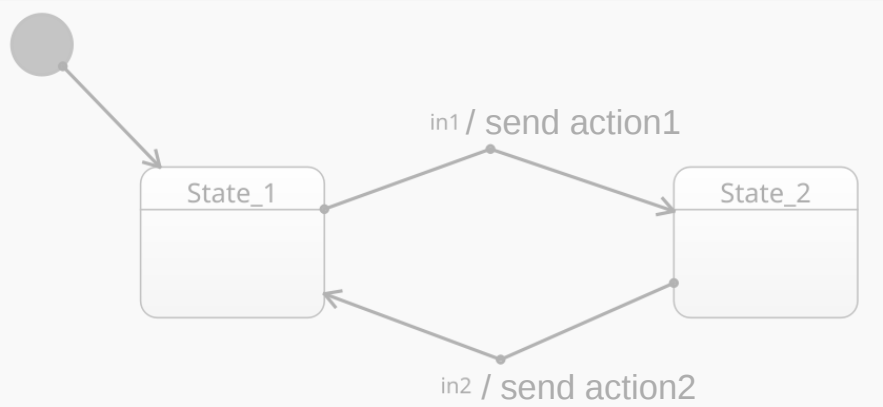
```
enum State {State_1, State_2};
enum Event {in1, in2};

void action1(){
    std::cout << "action 1 " << std::endl;
}
void action2(){
    std::cout << "action 2 " << std::endl;
}

State currentState = State_1;
```

```
int activate(Event newEvent){
    switch(currentState){
        case State_1:
            handleState_1(newEvent);
            break;
        case State_2:
            if (newEvent == in2){
                action2();
                currentState = State_1;
                std::cout << "enter State_1" << std::endl;
            }
            break;
    }
}
```

```
void handleState_1(Event newEvent){
    switch(newEvent){
        //transition 1 (t1)
        case in1:
                action1();
                currentState = State_2;
                std::cout << "enter State_2" << std::endl;
            break;
    }
}
```

```
int main(){
  activate(in1);
  activate(in1);
  activate(in2);
}
```

Ecrire du code implémentant un State chart



Switch case

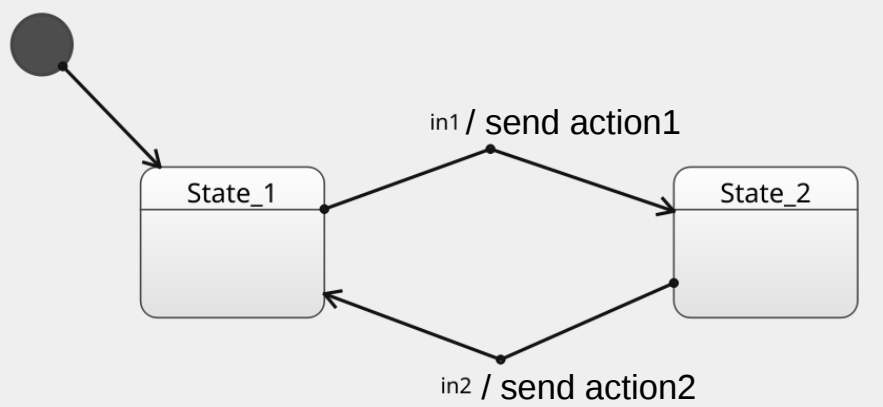
```
enum State {State_1, State_2};
enum Event {in1, in2};

void action1(){
    std::cout << "action 1 " << std::endl;
}
void action2(){
    std::cout << "action 2 " << std::endl;
}

State currentState = State_1;
```

```
int main(){
  activate(in1);
  activate(in1);
  activate(in2);
}
```

```
int activate(Event newEvent){
   switch(currentState){
      case State_1:
          handleState_1(newEvent);
         break;
      case State_2:
         if (newEvent == in2){
            action2();
            currentState = State_1;
            std::cout << "enter State_1" << std::endl;
         }
         break;
   }
}
```

```
void handleState_1(Event newEvent){
   switch(newEvent){
      //transition 1 (t1)
      case in1:
          //exit source of t1 (prefire)
          //actions of t1     (doActions)
          //set current state (postfire)
          //enter target of t1
      break;
   }
}
```

Ecrire du code implémentant un State chart



Switch case

```
enum State {State_1, State_2};
enum Event {in1, in2};

void action1(){
    std::cout << "action 1 " << std::endl;
}
void action2(){
    std::cout << "action 2 " << std::endl;
}

State currentState = State_1;
```

```
int activate(Event newEvent){
    switch(currentState){
        case State_1:
            handleState_1(newEvent);
            break;
        case State_2:
            if (newEvent == in2){
                action2();
                currentState = State_1;
                std::cout << "enter State_1" << std::endl;
            }
            break;
    }
}
```

```
int main(){
  activate(in1);
  activate(in1);
  activate(in2);
}
```

```
void handleState_1(Event newEvent){
    switch(newEvent){
        //transition 1 (t1)
        case in1:
            //exit source of t1 (prefire)
            //actions of t1     (doActions)
            //set current state (postfire)
            //enter target of t1
            break;
    }
}
```

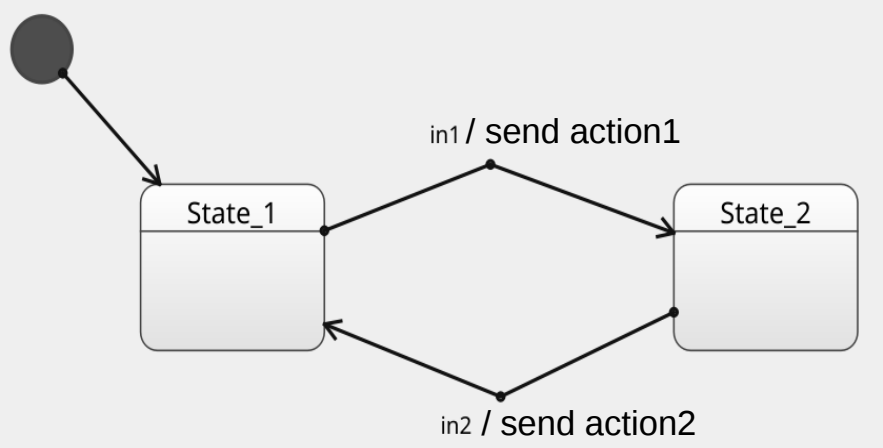
Peut bien sûr être encapsulé dans une classe !

Écrire/générer du code implémentant un State chart

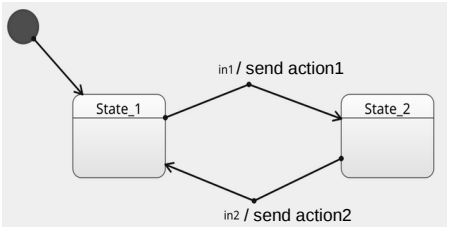
- Un state chart “simple” s’implémente simplement.
- On trouve généralement trois manières d’implémenter un state chart[1]:
  - Une implémentation procédurale “à base de Switch” ou chaque “case” correspond à un état
  - Une implémentation basée sur une représentation tabulaire des state charts.
  - Une implémentation orientée objet tirant partie du polymorphisme et du typage dynamique

[1] *Il en existe une infinité de variante et de croisement, mais ce sont les plus fréquentes*

State-Event table representation



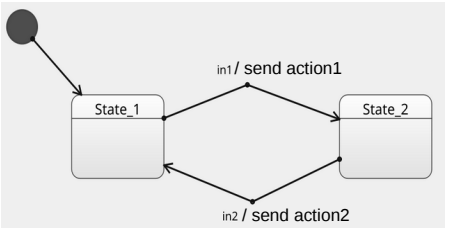
	in1	in2
State_1	State_2 (send action1)	X
State_2	X	State_1 (send action2)



State-Event table encoding

	in1	in2
State_1	State_2 (send action1)	X
State_2	X	State_1 (send action2)

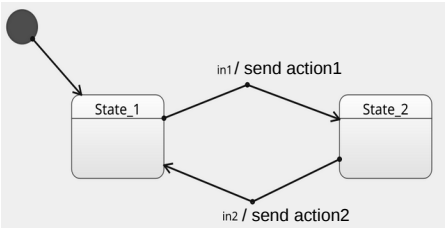
```
enum State {State_1, State_2};
enum Event {in1, in2};
const unsigned int nbState = 2;
const unsigned int nbEvent = 2;

State currentState = State_1;
```



State-Event table encoding

	in1	in2
State_1	State_2 (send action1)	X
State_2	X	State_1 (send action2)

```
enum State {State_1, State_2};
enum Event {in1, in2};
const unsigned int nbState = 2;
const unsigned int nbEvent = 2;

State currentState = State_1;

void action1(){
    std::cout << "action 1 " << std::endl;
}
void fire1(){
    action1();
    currentState = State_2;
}
```

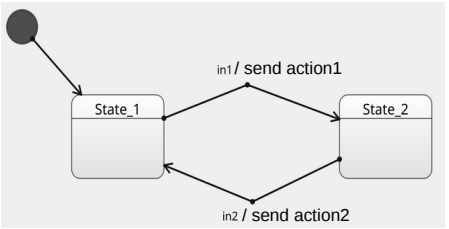


State-Event table encoding

	in1	in2
State_1	State_2 (send action1)	X
State_2	X	State_1 (send action2)

```
enum State {State_1, State_2};
enum Event {in1, in2};
const unsigned int nbState = 2;
const unsigned int nbEvent = 2;

State currentState = State_1;

void action1(){
    std::cout << "action 1 " << std::endl;
}
void fire1(){
    action1();
    currentState = State_2;
}
```

```
void action2(){
    std::cout << "action 2 " << std::endl;
}
void fire2(){
    action2();
    currentState = State_1;
}
```

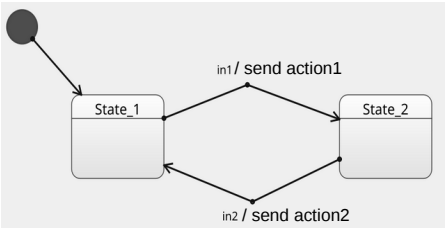


State-Event table encoding

	in1	in2
State_1	State_2 (send action1)	X
State_2	X	State_1 (send action2)

```
void action2(){
    std::cout << "action 2 " << std::endl;
}
void fire2(){
    action2();
    currentState = State_1;
}
void idle(){}
```

```
enum State {State_1, State_2};
enum Event {in1, in2};
const unsigned int nbState = 2;
const unsigned int nbEvent = 2;

State currentState = State_1;

void action1(){
    std::cout << "action 1 " << std::endl;
}
void fire1(){
    action1();
    currentState = State_2;
}
```



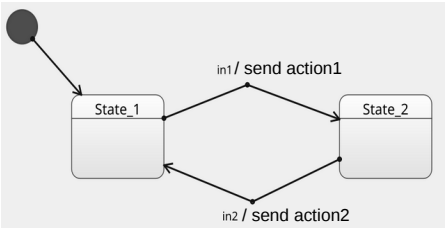
State-Event table encoding

	in1	in2
State_1	State_2 (send action1)	X
State_2	X	State_1 (send action2)

```
enum State {State_1, State_2};
enum Event {in1, in2};
const unsigned int nbState = 2;
const unsigned int nbEvent = 2;

State currentState = State_1;

void action1(){
   std::cout << "action 1 " << std::endl;
}
void fire1(){
   action1();
   currentState = State_2;
}
```

```
void action2(){
   std::cout << "action 2 " << std::endl;
}
void fire2(){
   action2();
   currentState = State_1;
}
void idle(){}
```

```
void error(){
      throw std::domain_error("this event is not expected");
}
```

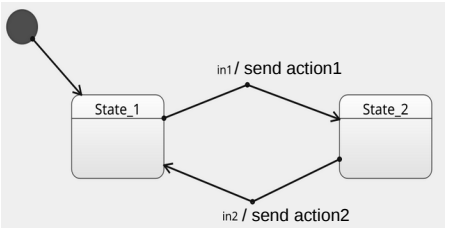


State-Event table encoding

	in1	in2
State_1	State_2 (send action1)	X
State_2	X	State_1 (send action2)

```
void action2(){
    std::cout << "action 2 " << std::endl;
}
void fire2(){
    action2();
    currentState = State_1;
}
void idle(){}
```

```
enum State {State_1, State_2};
enum Event {in1, in2};
const unsigned int nbState = 2;
const unsigned int nbEvent = 2;

State currentState = State_1;

void action1(){
    std::cout << "action 1 " << std::endl;
}
void fire1(){
    action1();
    currentState = State_2;
}
```



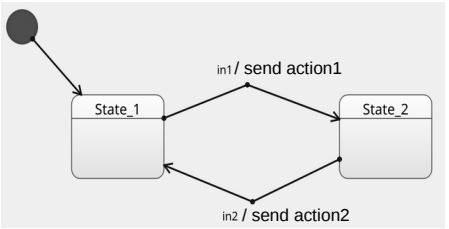
State-Event table encoding

	in1	in2
State_1	State_2 (send action1)	X
State_2	X	State_1 (send action2)

```
enum State {State_1, State_2};
enum Event {in1, in2};
const unsigned int nbState = 2;
const unsigned int nbEvent = 2;

State currentState = State_1;

void action1(){
    std::cout << "action 1 " << std::endl;
}
void fire1(){
    action1();
    currentState = State_2;
}
```

```
void action2(){
    std::cout << "action 2 " << std::endl;
}
void fire2(){
    action2();
    currentState = State_1;
}
void idle(){}
```

```
using FunctionPtr = void (*)();
using FSM = std::array<std::array<FunctionPtr,nbEvent>,nbState>;
```



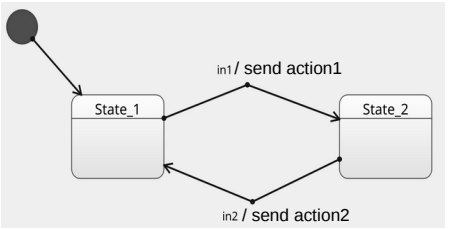
State-Event table encoding

	in1	in2
State_1	State_2 (send action1)	X
State_2	X	State_1 (send action2)

```
enum State {State_1, State_2};
enum Event {in1, in2};
const unsigned int nbState = 2;
const unsigned int nbEvent = 2;

State currentState = State_1;

void action1(){
    std::cout << "action 1 " << std::endl;
}
void fire1(){
    action1();
    currentState = State_2;
}
```

```
void action2(){
    std::cout << "action 2 " << std::endl;
}
void fire2(){
    action2();
    currentState = State_1;
}
void idle(){}
```

```
using FunctionPtr = void (*)();
using FSM = std::array<std::array<FunctionPtr,nbEvent>,nbState>;

FSM fsm =
{{
    {fire1,idle},
    {idle,fire2}
}};
```



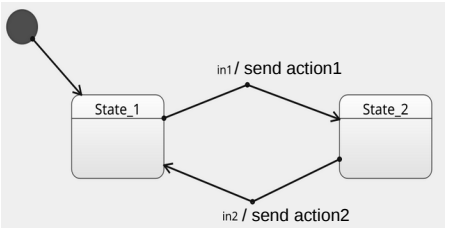
State-Event table encoding

	in1	in2
State_1	State_2 (send action1)	X
State_2	X	State_1 (send action2)

```
enum State {State_1, State_2};
enum Event {in1, in2};
const unsigned int nbState = 2;
const unsigned int nbEvent = 2;

State currentState = State_1;

void action1(){
   std::cout << "action 1 " << std::endl;
}
void fire1(){
   action1();
   currentState = State_2;
}
```

```
void action2(){
   std::cout << "action 2 " << std::endl;
}
void fire2(){
   action2();
   currentState = State_1;
}
void idle(){}
```

```
using FunctionPtr = void (*)();
using FSM = std::array<std::array<FunctionPtr,nbEvent>,nbState>;

FSM fsm =
{{
   {fire1,idle},
   {idle,fire2}
}};

int activate(Event newEvent){
   fsm[currentState][newEvent]();
}
```



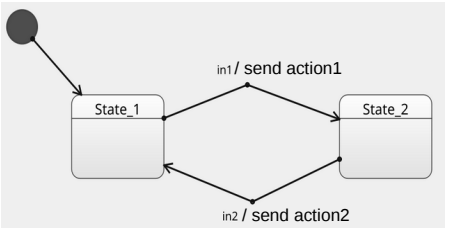
State-Event table encoding

	in1	in2
State_1	State_2 (send action1)	X
State_2	X	State_1 (send action2)

```
enum State {State_1, State_2};
enum Event {in1, in2};
const unsigned int nbState = 2;
const unsigned int nbEvent = 2;

State currentState = State_1;

void action1(){
   std::cout << "action 1 " << std::endl;
}
void fire1(){
   action1();
   currentState = State_2;
}
```

```
void action2(){
   std::cout << "action 2 " << std::endl;
}
void fire2(){
   action2();
   currentState = State_1;
}
void idle(){}
```

```
using FunctionPtr = void (*)();
using FSM = std::array<std::array<FunctionPtr,nbEvent>,nbState>;

FSM fsm =
{{
   {fire1,idle},
   {idle,fire2}
}};

int activate(Event newEvent){
  fsm.at(currentState).at(newEvent)();
}
```



State-Event table encoding

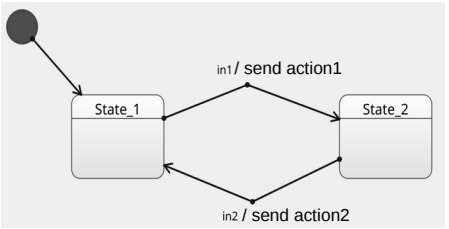
	in1	in2
State_1	State_2 (send action1)	X
State_2	X	State_1 (send action2)

```
enum State {State_1, State_2};
enum Event {in1, in2};
const unsigned int nbState = 2;
const unsigned int nbEvent = 2;

State currentState = State_1;

void action1(){
   std::cout << "action 1 " << std::endl;
}
void fire1(){
   action1();
   currentState = State_2;
}
```

```
void action2(){
   std::cout << "action 2 " << std::endl;
}
void fire2(){
   action2();
   currentState = State_1;
}
void idle(){}
```

```
using FunctionPtr = void (*)();
using FSM = std::array<std::array<FunctionPtr,nbEvent>,nbState>;

FSM fsm =
{{
   {fire1,idle},
   {idle,fire2}
}};

int activate(Event newEvent){
  fsm[currentState][newEvent]();
}
```

```
int main(){
  activate(in1);
  activate(in1);
  activate(in2);
}
```



State-Event table encoding

	in1	in2
State_1	State_2 (send action1)	X
State_2	X	State_1 (send action2)

```
enum State {State_1, State_2};
enum Event {in1, in2};
const unsigned int nbState = 2;
const unsigned int nbEvent = 2;

State currentState = State_1;

void action1(){
   std::cout << "action 1 " << std::endl;
}
void fire1(){
   action1();
   currentState = State_2;
}
```

```
void action2(){
   std::cout << "action 2 " << std::endl;
}
void fire2(){
   action2();
   currentState = State_1;
}
void idle(){}
```

```
using FunctionPtr = void (*)();
using FSM = std::array<std::array<FunctionPtr,nbEvent>,nbState>;

FSM fsm =
{{
   {fire1,idle},
   {idle,fire2}
}};

int activate(Event newEvent){
  fsm[currentState][newEvent]();
}
```

```
int main(){
  activate(in1);
  activate(in1);
  activate(in2);
}
```

Peut bien sûr être encapsulé dans une classe !

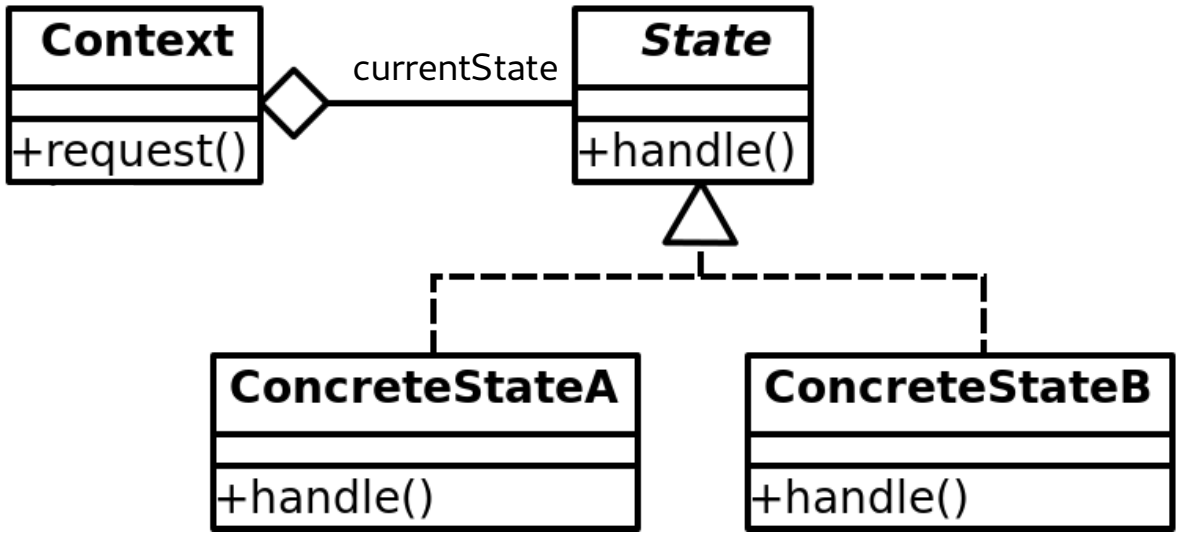
Écrire/générer du code implémentant un State chart

- Un state chart “simple” s’implémente simplement.
- On trouve généralement deux manières d’implémenter un state chart[1]:
  - Une implémentation procédurale “à base de Switch” ou chaque “case” correspond à un état
  - Une implémentation basée sur une représentation tabulaire des state charts.
  - Une implémentation orientée objet tirant partie du polymorphisme et du typage dynamique

[1] *Il en existe une infinité de variante et de croisement, mais ce sont les plus fréquentes*

Écrire du code implémentant un State chart

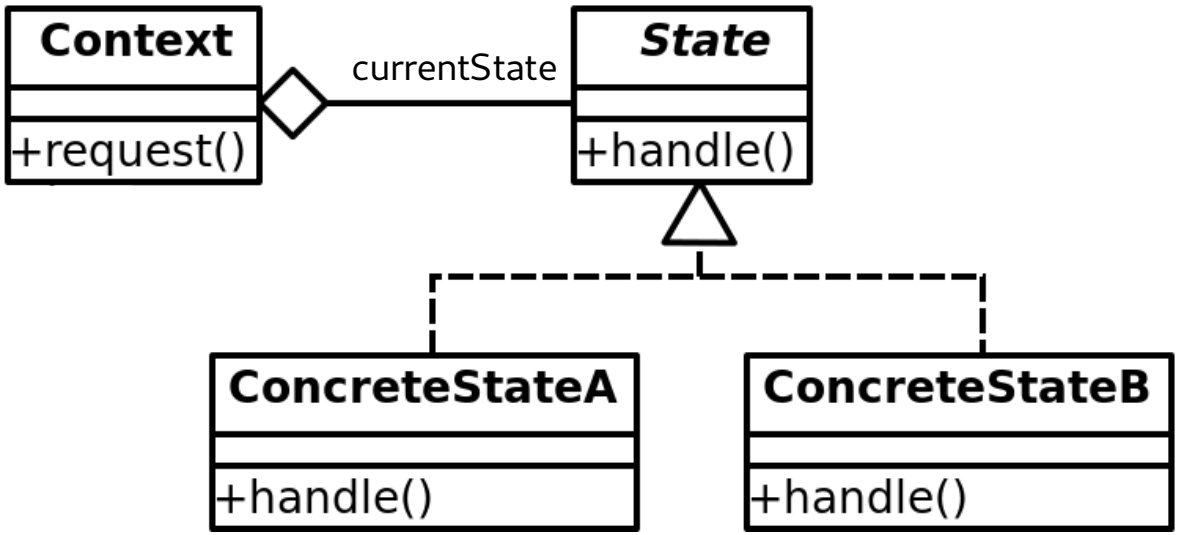
State Pattern



Modified from wikipedia

Écrire du code implémentant un State chart

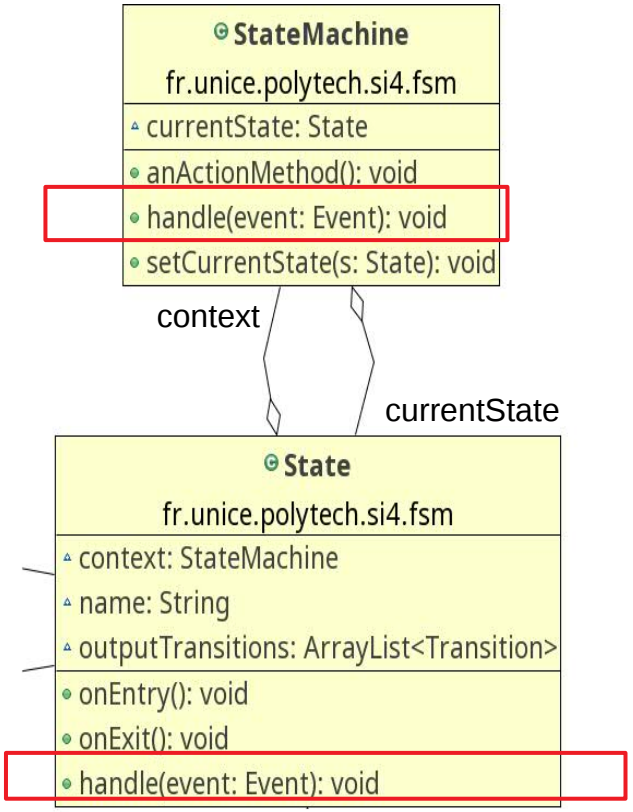
State Pattern



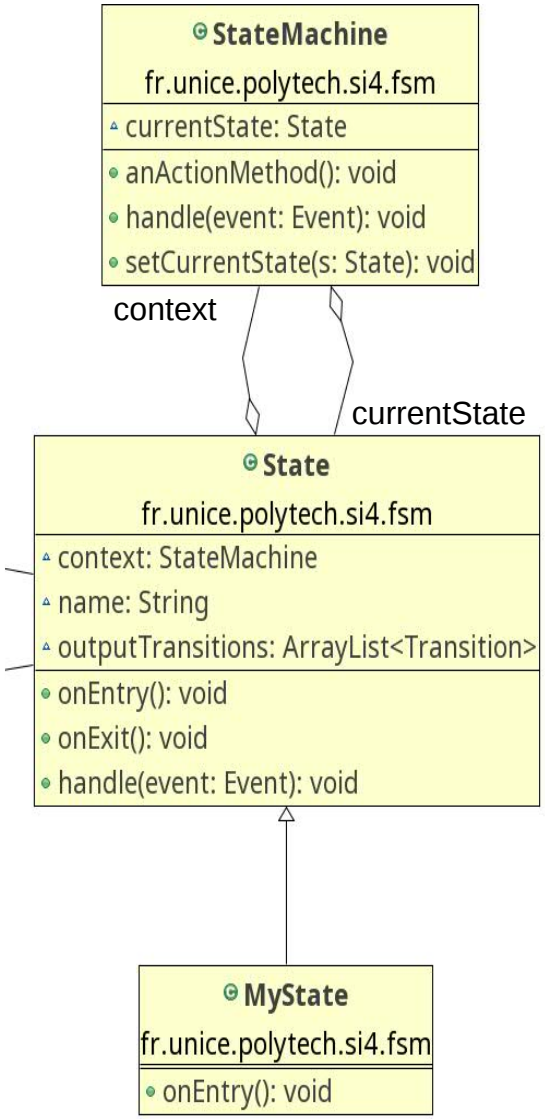
Modified from wikipedia

```
public void request(){
   this.currentState.handle()
}
```

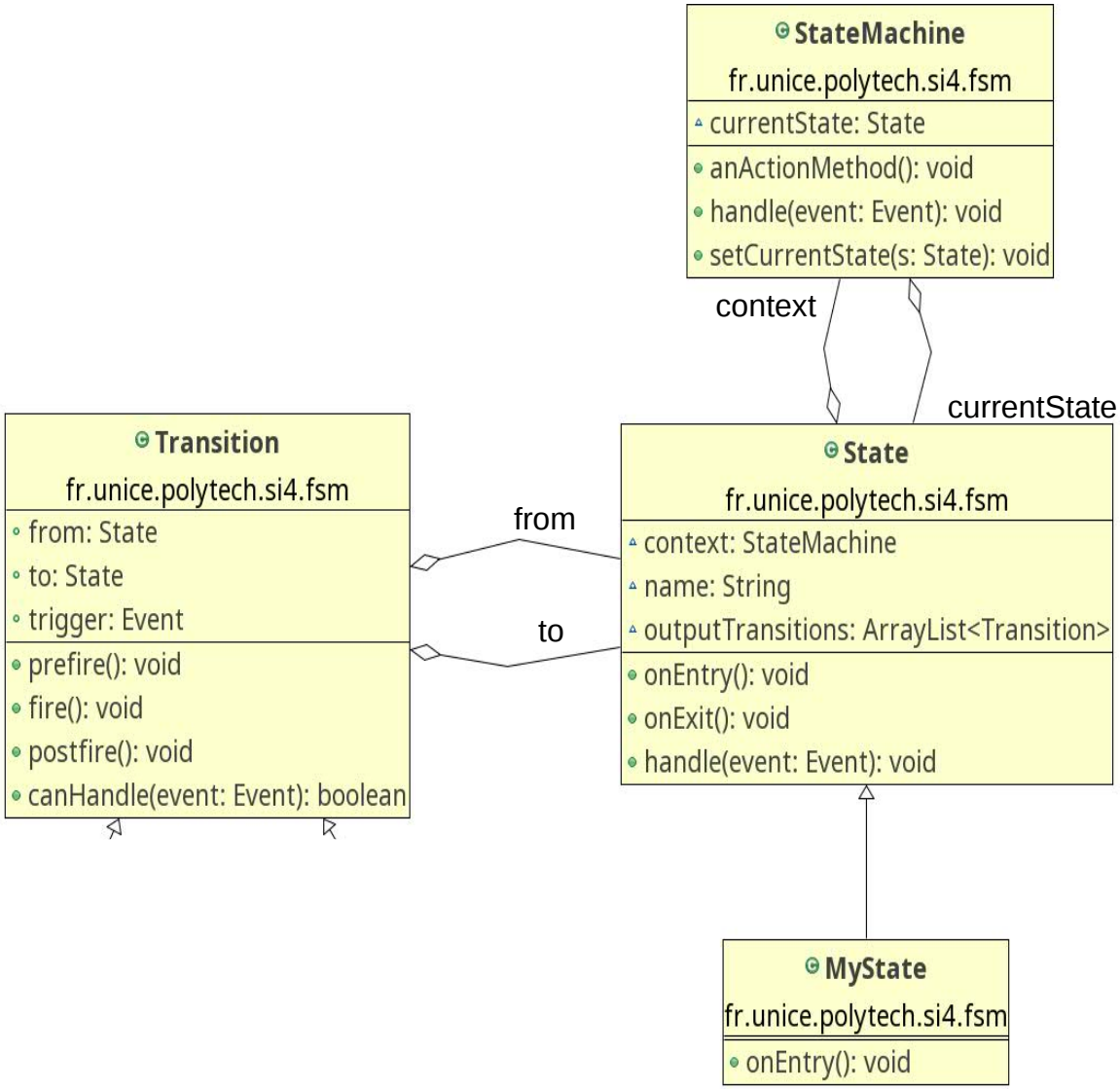
Adapted State Pattern



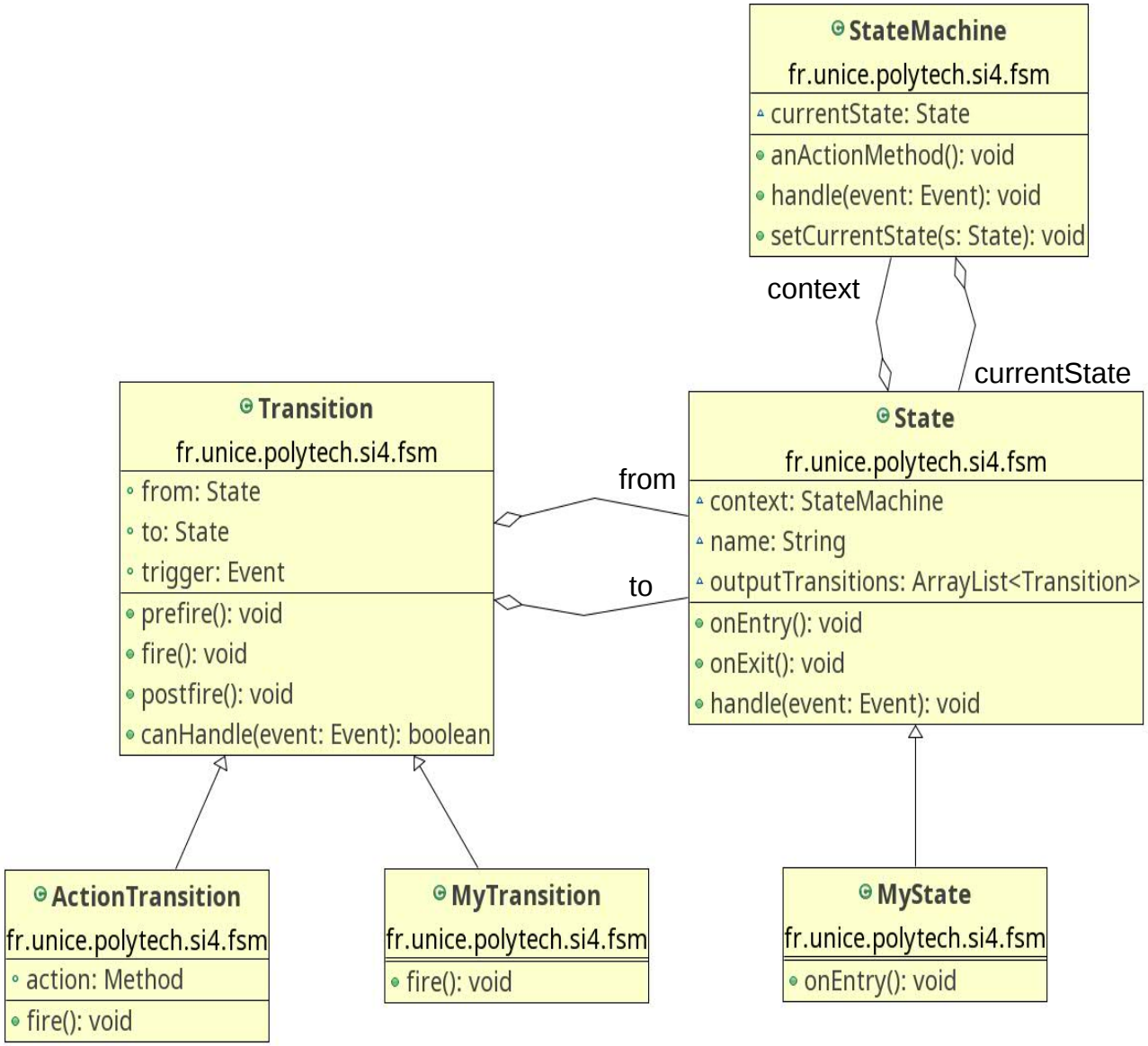
Adapted State Pattern



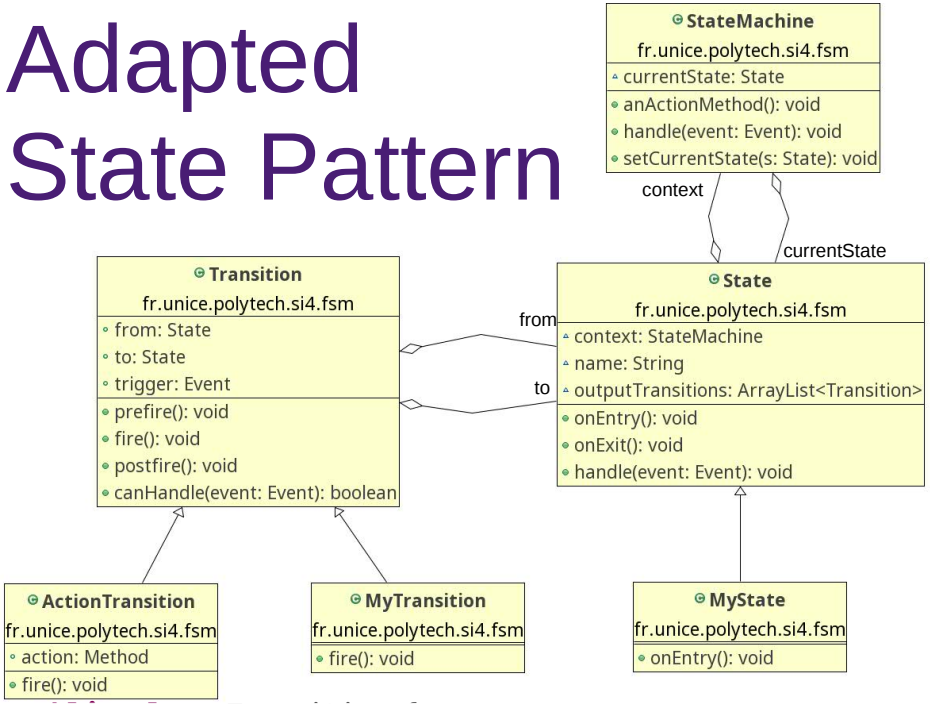
Adapted State Pattern



Adapted State Pattern



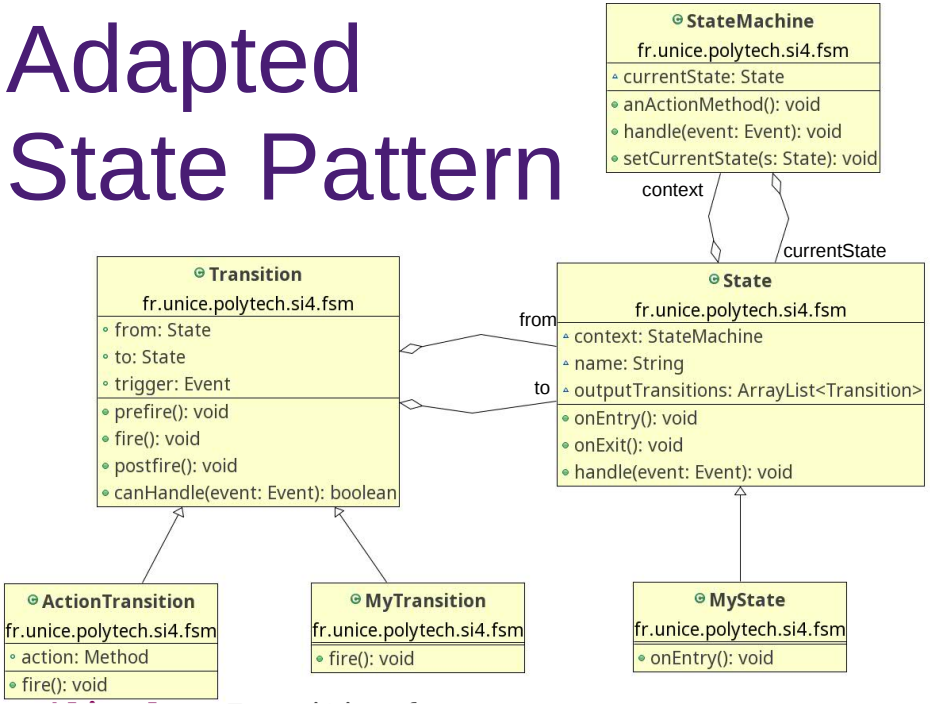
Adapted State Pattern




```
public class StateMachine {
	State currentState = null;
	public void handle(Event event){
		currentState.handle(event);
	}
}
```

!

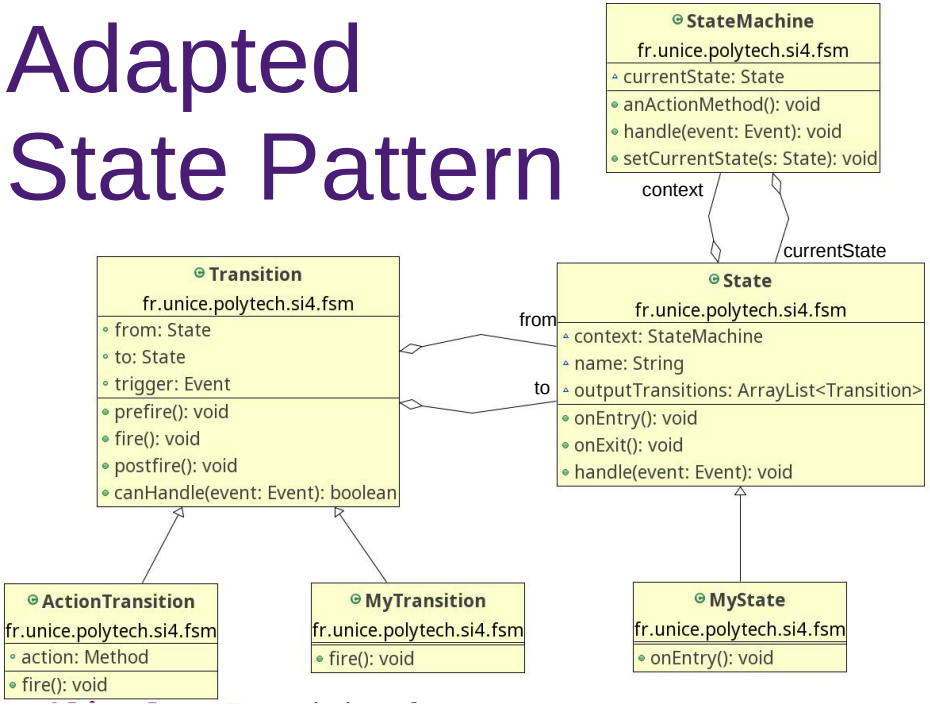
Adapted State Pattern




```
public class StateMachine {
	State currentState = null;
	public void handle(Event event){
		currentState.handle(event);
	}
}
```

```
public class State {
	public void handle(Event event){
		System.out.println("receiving "+event);
		for(Transition t : outputTransitions){
			if (t.canHandle(event)){
				t.prefire();
				t.fire();
				t.postfire();
				context.currentState = t.to;
				return;
			}
		}
	}
}
```

Adapted State Pattern

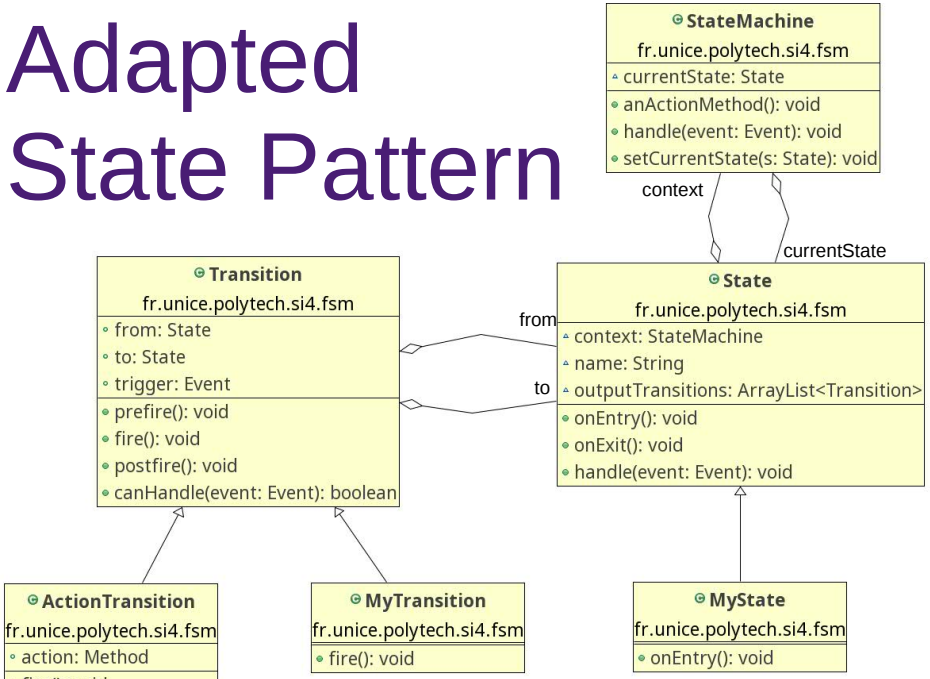



```
public class StateMachine {
	State currentState = null;
	public void handle(Event event){
		currentState.handle(event);
	}
}
```

```
public class State {
	public void handle(Event event){
		System.out.println("receiving "+event);
		for(Transition t : outputTransitions){
			if (t.canHandle(event)){
				t.prefire();
				t.fire();
				t.postfire();
				context.currentState = t.to;
				return;
			}
		}
	}
}
```

Impact of implementation !

Adapted State Pattern

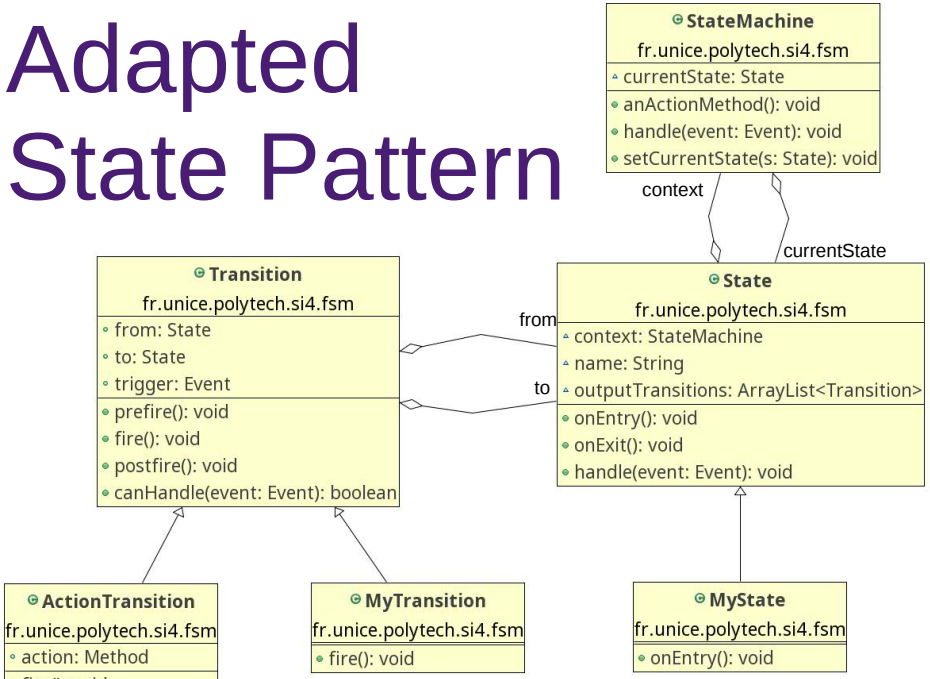



```
public class Transition {
	public State from = null;
	public State to = null;
	public Event trigger;

	public void prefire() {
		this.from.onExit();
	}
	public void fire(){
		System.out.println(
			"  -> fire transition from "+
			from.name +" to "+to.name);
	}
	public void postfire(){
		this.to.onEntry();
	}
	public boolean canHandle(Event event){
		return trigger == event;
	}
}
```

```
public class StateMachine {
	State currentState = null;
	public void handle(Event event){
		currentState.handle(event);
	}
}
```

```
public class State {
	public void handle(Event event){
		System.out.println("receiving "+event);
		for(Transition t : outputTransitions){
			if (t.canHandle(event)){
				t.prefire();
				t.fire();
				t.postfire();
				context.currentState = t.to;
				return;
			}
		}
	}
}
```

Adapted State Pattern



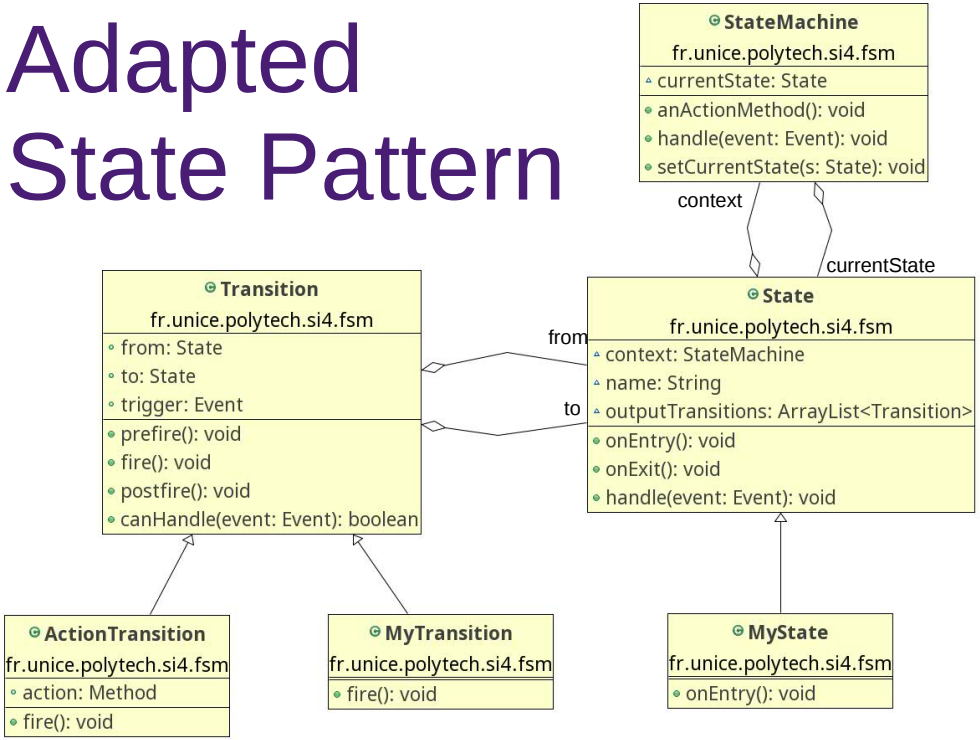

```
public class Transition {
	public State from = null;
	public State to = null;
	public Event trigger;

	public void prefire() {
		this.from.onExit();
	}
	public void fire(){
		System.out.println(
			"  -> fire transition from "+
			from.name +" to "+to.name);
	}
	public void postfire(){
		this.to.onEntry();
	}
	public boolean canHandle(Event event){
		return trigger == event;
	}
}
```

```
public class StateMachine {
	State currentState = null;
	public void handle(Event event){
		currentState.handle(event);
	}
}
```

```
public class State {
	public void handle(Event event){
		System.out.println("receiving "+event);
		for(Transition t : outputTransitions){
			if (t.canHandle(event)){
				t.prefire();
				t.fire();
				t.postfire();
				context.currentState = t.to;
				return;
			}
		}
	}
}
```

```
public class ActionTransition extends Transition{
	public Method action;
	public void fire() {
		super.fire();
		action.invoke(from.context);
	}
}
```

Adapted State Pattern



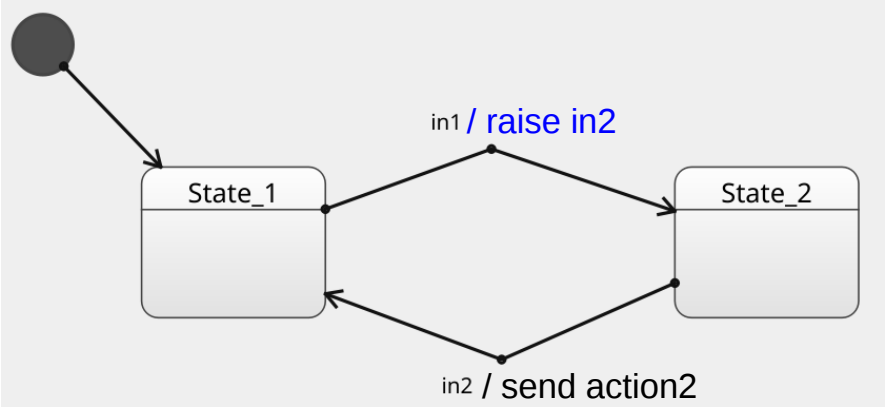

```
public class Test1 {

public static void main(String[] args) {

  StateMachine fsm = new StateMachine();

  State s1 = new State(fsm,"s1");
  State s2 = new MyState(fsm,"s2");

 Transition t1 = new MyTransition(s1, s2, Event.e1);
 Transition t2 = new Transition(s2, s1, Event.e2);
 ActionTransition t3 = new ActionTransition(s1, s1, Event.e3,
             fsm.getClass().getMethod("anActionMethod"));

 s1.addOutputTransition(t1);
 s2.addOutputTransition(t2);
 s1.addOutputTransition(t3);


 fsm.setCurrentState(s1);
 fsm.handle(Event.e1);
 fsm.handle(Event.e1);
 fsm.handle(Event.e2);
 fsm.handle(Event.e3);
 fsm.handle(Event.e3);
 fsm.handle(Event.e3);
 return ;
 }
}
```

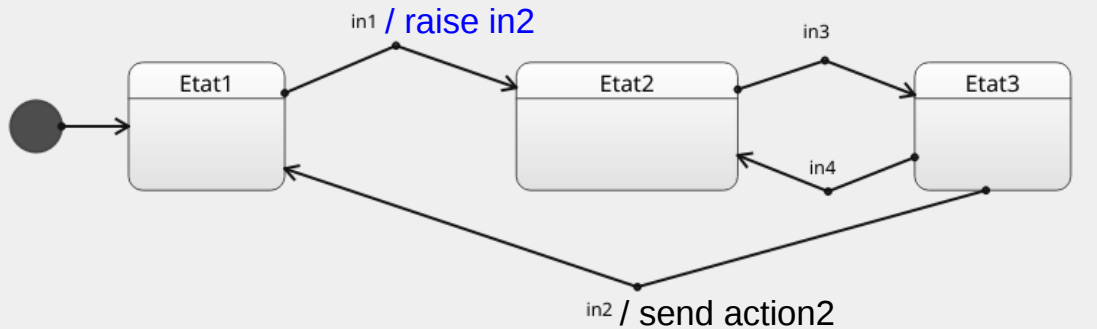
Écrire du code implémentant un State chart

- Un state chart “simple” s’implémente simplement.
- Il peut être nécessaire de faire des choix de “simplification” (exemple, sending action1 implique d’appeler la fonction action1())
- SCXML propose de nombreuses fonctionnalités de modélisation et de nombreuses subtilités dans la sémantique sous jacente.
- Il est important de savoir quelles sont les constructions supportées par le code que l’on veut écrire/générer.

Gestion de raise/send

- Choice: raise is internal only, send is external only and call a function whose name is the one of the event

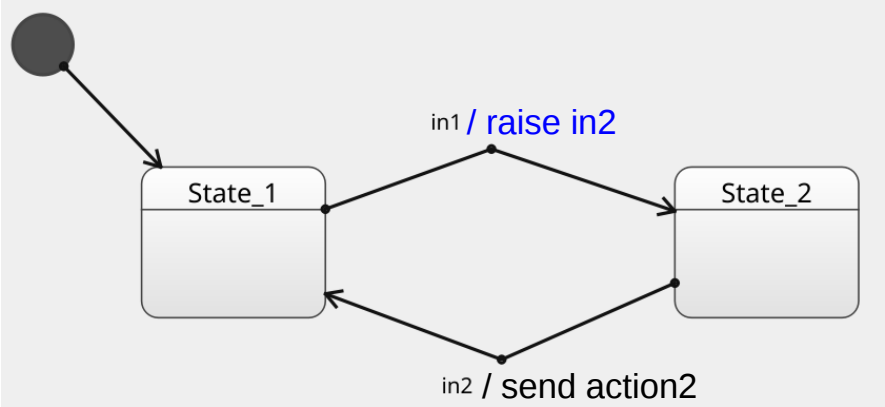




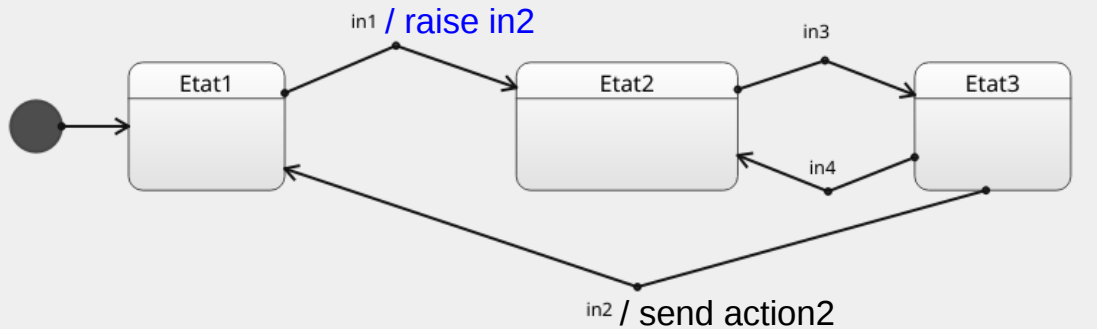
Gestion de raise/send

- Choice: raise is internal only, send is external only and call a function whose name is the one of the event

- Need to deal with an event queue
- Need to be sure about the semantics (*i.e.*, how to manage the queue)




```
t0.start();
this->t0.submitEvent("in1");
```

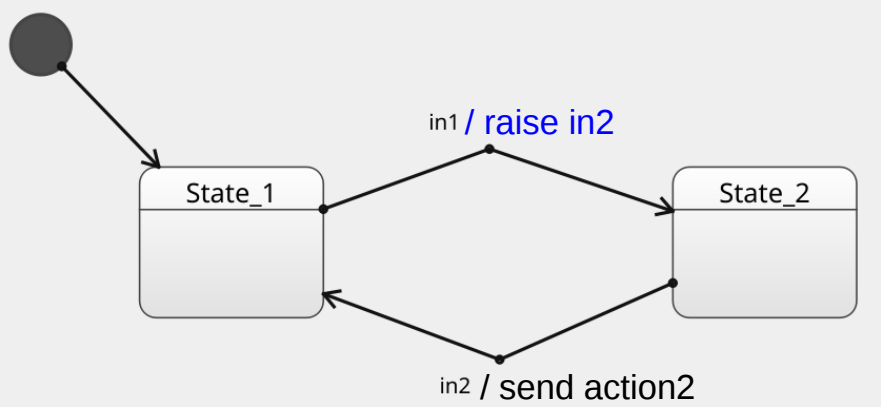



```
t1.start();
this->t1.submitEvent("in1");
this->t1.submitEvent("in3");
this->t1.submitEvent("in4");
```

Gestion de raise/send

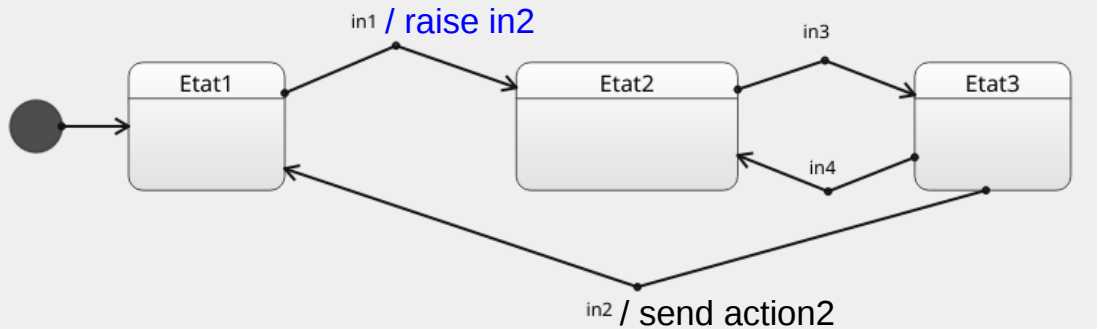
- Choice: raise is internal only, send is external only and call a function whose name is the one of the event

- Need to deal with an event queue
- Need to be sure about the semantics (*i.e.*, how to manage the queue)




```
t0.start();
this->t0.submitEvent("in1");
```

```
scxml.statemachine: "" : "enter etat1"
scxml.statemachine: "" : "fire transition in1"
scxml.statemachine: "" : "enter etat2"
scxml.statemachine: "" : "fire transition in2"
scxml.statemachine: "" : "enter etat1"
```

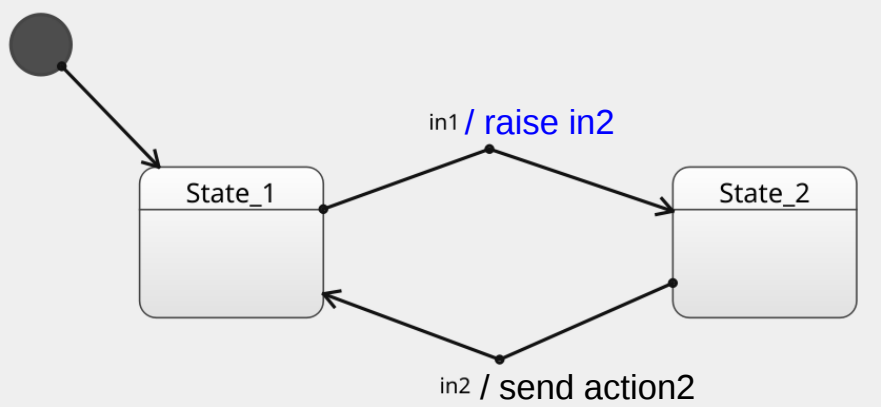



```
t1.start();
this->t1.submitEvent("in1");
this->t1.submitEvent("in3");
this->t1.submitEvent("in4");
```

Gestion de raise/send

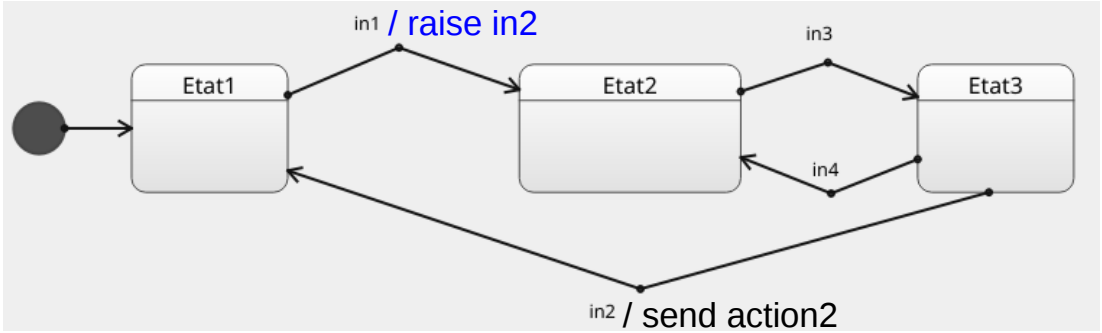
- Choice: raise is internal only, send is external only and call a function whose name is the one of the event

- Need to deal with an event queue
- Need to be sure about the semantics (*i.e.*, how to manage the queue)




```
t0.start();
this->t0.submitEvent("in1");
```

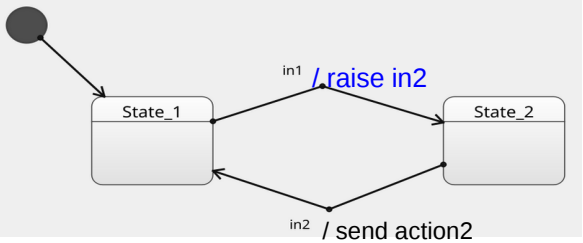
```
scxml.statemachine: "" : "enter etat1"
scxml.statemachine: "" : "fire transition in1"
scxml.statemachine: "" : "enter etat2"
scxml.statemachine: "" : "fire transition in2"
scxml.statemachine: "" : "enter etat1"
```




```
t1.start();
this->t1.submitEvent("in1");
this->t1.submitEvent("in3");
this->t1.submitEvent("in4");
```

```
scxml.statemachine: "" : "enter etat1"
scxml.statemachine: "" : "fire transition in1"
scxml.statemachine: "" : "enter etat2"
scxml.statemachine: "" : "fire transition in3"
scxml.statemachine: "" : "enter etat 3"
scxml.statemachine: "" : "fire transition in4"
scxml.statemachine: "" : "enter etat2"
```

Gestion de raise/send



- Choice: raise is internal only, send is external only and call a function whose name is the one of the event
- Need to deal with an event queue
- Need to be sure about the semantics (*i.e.*, how to manage the queue)

```
void activate(){
   while(! internalQueue.empty()){
      Event newEvent = internalQueue.front();
      bool consummed = false;
      switch(currentState){
         case State_1:
            consummed = handleState_1(newEvent);
            break;
         case State_2:
            if (newEvent == in2){
               consummed = true;
               action2();
               currentState = State_1;
               std::cout << "enter State_1" << std::endl;
            }
            break;
      }
      internalQueue.pop();
      if (!consummed){
         std::cout << "unconsummed event: " << newEvent << std::endl;
      }

   }
}
```

```
bool handleState_1(Event newEvent){
switch(newEvent){
      //transition 1 (t1)
      case in1:
         //exit source of t1
         internalQueue.push(in2);
         currentState = State_2;
         //enter the target of t1
        return true;
   }
   return false;
}
```

Gestion de raise/send

- Choice: raise is internal only, send is external only and call a function whose name is the one of the event

- Need to deal with an event queue
- **Need to be sure about the semantics (*i.e.*, how to manage the queue)**

external event: An SCXML event appearing in the external event queue. Such events are either sent by external sources or generated with the <send> element.

internal event: An event appearing in the internal event queue. Such events are either raised automatically by the platform or generated with the <raise> or <send> elements.

Microstep: A microstep involves the processing of a single transition (or, in the case of parallel states, a single set of transitions.) A microstep may change the current configuration, update the data model and/or generate new (internal and/or external) events. This, by causality, may in turn enable additional transitions which will be handled in the next microstep in the sequence, and so on.

Macrostep: A macrostep consists of a sequence (a chain) of microsteps, at the end of which the state machine is in a stable state and ready to process an external event. Each external event causes an SCXML state machine to take exactly one macrostep. However, if the external event does not enable any transitions, no microstep will be taken, and the corresponding macrostep will be empty

Run to completion: SCXML adheres to a run to completion semantics in the sense that an external event can only be processed when the processing of the previous external event has completed, i.e. when all microsteps (involving all triggered transitions) have been completely taken..

Gestion de raise/send

- Choice: raise is internal only, send is external only and call a function whose name is the one of the event

- Need to deal with an event queue
- **Need to be sure about the semantics (*i.e.*, how to manage the queue)**

external event: An SCXML event appearing in the external event queue. Such events are either sent by external sources or generated with the <send> element.

internal event: An event appearing in the internal event queue. Such events are either raised automatically by the platform or generated with the <raise> or <send> elements.

Microstep: A microstep involves the processing of a single transition (or, in the case of parallel states, a single set of transitions.) A microstep may change the current configuration, update the data model and/or generate new (internal and/or external) events. This, by causality, may in turn enable additional transitions which will be handled in the next microstep in the sequence, and so on.

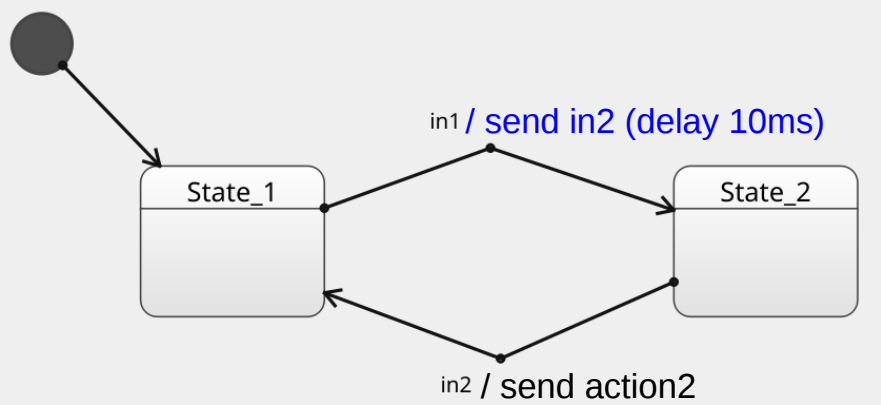
Macrostep: A macrostep consists of a sequence (a chain) of microsteps, at the end of which the state machine is in a stable state and ready to process an external event. Each external event causes an SCXML state machine to take exactly one macrostep. However, if the external event does not enable any transitions, no microstep will be taken, and the corresponding macrostep will be empty

Run to completion: SCXML adheres to a run to completion semantics in the sense that an external event can only be processed when the processing of the previous external event has completed, i.e. when all microsteps (involving all triggered transitions) have been completely taken..

More generally: https://www.w3.org/TR/scxml/#AlgorithmforSCXMLInterpretation

Écrire du code implémentant un State chart

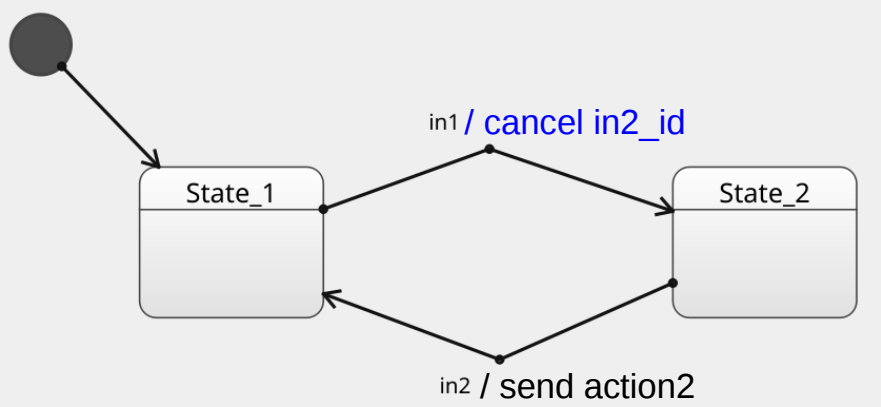
- Un state chart “simple” s’implémente simplement.
- SCXML propose de nombreuse fonctionnalités de modélisation et de nombreuses subtilités dans la sémantique sous jacente.
- Il est important de savoir quelles sont les constructions supportées par le code que l’on veut écrire/générer.
- Il peut être nécessaire de faire des choix de “simplification” (exemple, sending action1 implique d’appeler la fonction action1())



Ici par exemple la difficulté dépend
- du langage cible (mise en place de call back)
- Du choix du type de code
  - Execution symbolic
  - Execution en temps “réel” (wall clock)

Écrire du code implémentant un State chart

- Un state chart “simple” s’implémente simplement.
- SCXML propose de nombreuse fonctionnalités de modélisation et de nombreuses subtilités dans la sémantique sous jacente.
- Il est important de savoir quelles sont les constructions supportées par le code que l’on veut écrire/générer.
- Il peut être nécessaire de faire des choix de “simplification” (exemple, sending action1 implique d’appeler la fonction action1())



Écrire du code implémentant un State chart

- Supporter toutes les constructions offertes par SCXML peut s'avérer compliqué (vous pouvez aller voir le code générer par *Yakindu* si vous êtes curieux)