LECTURE 4

UML STATE MACHINES

SUBJECTS

uOttawa
L'Université canadienne
Canada's university

States

Transitions

Guards

Effects

State actions

Decisions

Compound states

Alternative entry and exit

History states

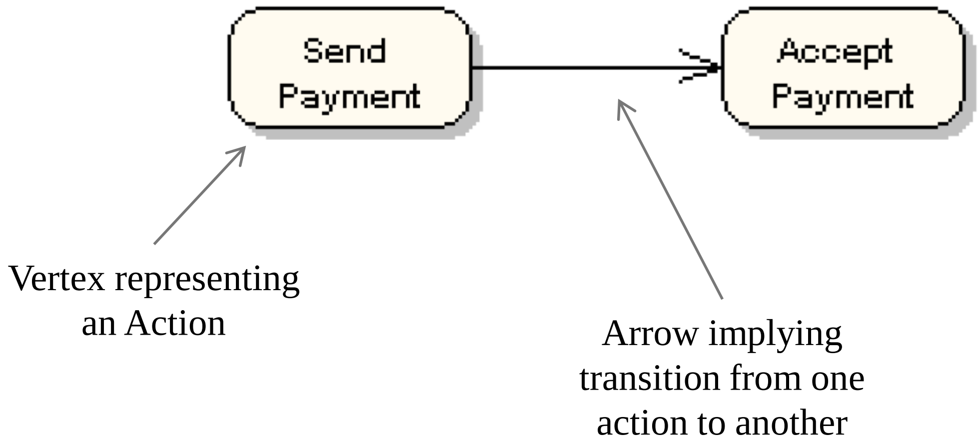
Case study

ACTIVITY DIAGRAMS VS STATE MACHINES

uOttawa
L'Université canadienne
Canada's university

In Activity Diagrams

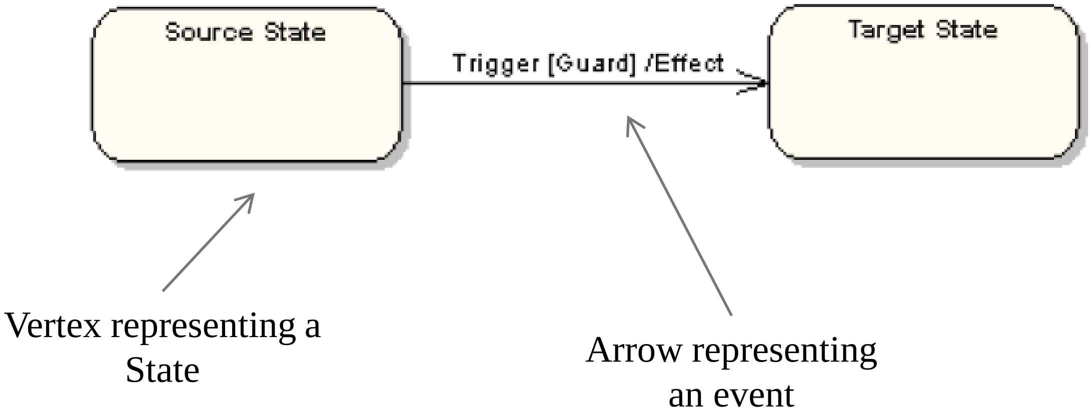
- Vertices represent Actions
- Edges (arrows) represent transition that occurs at the completion of one action and before the start of another one (control flow)



ACTIVITY DIAGRAMS VS STATE MACHINES

In State Machines

- Vertices represent states of a process
- Edges (arrows) represent occurrences of events



UML STATE MACHINES

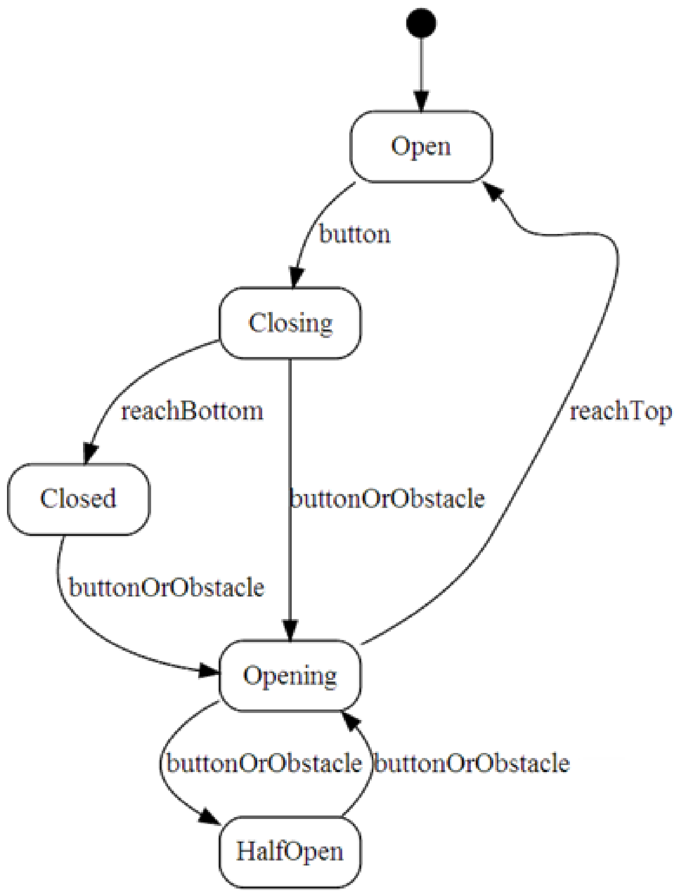
Used to model the dynamic behaviour of a process

- Can be used to model a high level behaviour of an entire system
- Can be used to model the detailed behaviour of a single object
- All other possible levels of detail in between these extremes are also possible

UML STATE MACHINE EXAMPLE

uOttawa
L'Université canadienne
Canada's university

Example of a garage door state machine



STATES

Symbol for a state 

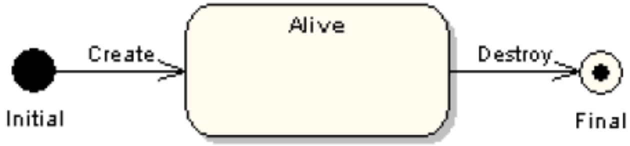
A system in a state will remain in it until the occurrence of an event that will cause it to transition to another one

- Being in a state means that a system will behave in a predetermined way in response to a given event

Names for states are usually chosen as:

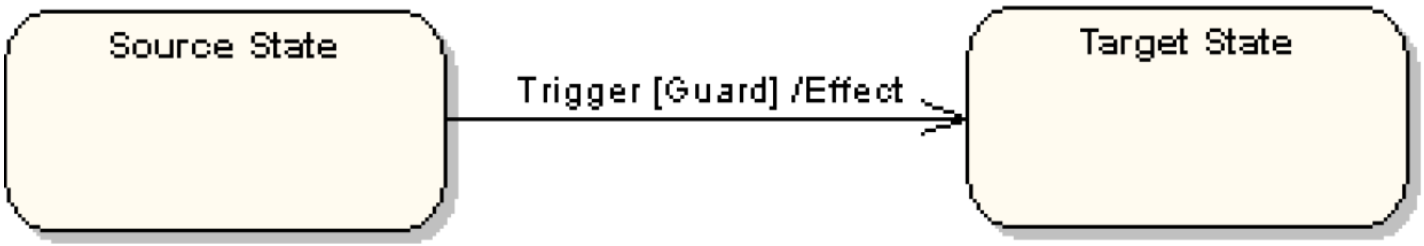
- Adjectives: open, closed, ready…
- Present continuous verbs: opening, closing, waiting…

Symbols for the initial and final states



TRANSITIONS

Transitions are represented with arrows



TRANSITIONS

Transitions represent a change in a state in response to an event

- Theoretically, it is supposed to occur in a instantaneous manner (it does not take time to execute)

A transition can have:

- **Trigger:** causes the transition; can be any type of event
- **Guard:** a condition that must evaluate to true for the transition to occur
- **Effect:** an action that will be invoked directly on the system
    - If we are modeling an object, the effect would correspond to a specific method call

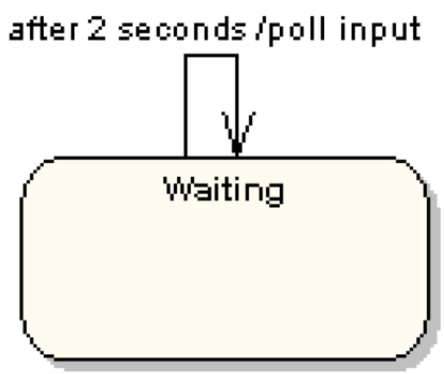
SELF TRANSITION

State can also have self transitions

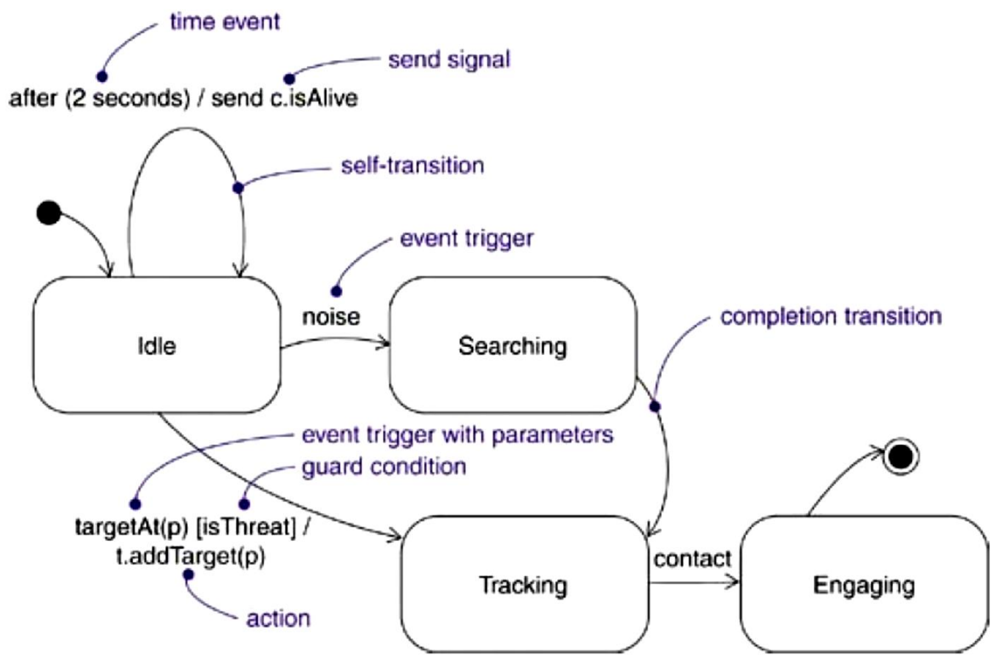
- These self transition are more useful when they have an effect associated with them

Timer events are usually popular with self transitions

Below is a typical example:



COMPLETION TRANSITIONS



Completion transition: transition with no trigger that is initiated implicitly when its source state has completed its behavior, if any

GUARD CONDITION

A guard condition is a Boolean expression enclosed in square brackets and placed after the trigger event

The guard condition is only evaluated after the trigger event for its transition occurs

- It is possible to have multiple transitions from the same source state and with the same event trigger, as long as the guard conditions do not overlap

EFFECT

An effect is a behavior that is executed when a transition fires

Effects may include

- Inline computation
- Operation calls
- Creation and destruction of another object
- Sending a signal to an object

To indicate sending of a signal, you can prefix the signal name with the keyword *send*

STATE ACTIONS

An effect can also be associated with a state

If a destination state is associated with numerous incident transitions *(transitions arriving a that state)*, and every transition defines the same effect:

- The effect can therefore be associated with the state instead of the transitions (avoid duplications)
- This can be achieved using an “On Entry” effect (we can have multiple entry effects)
- We can also add one or more “On Exit” effect

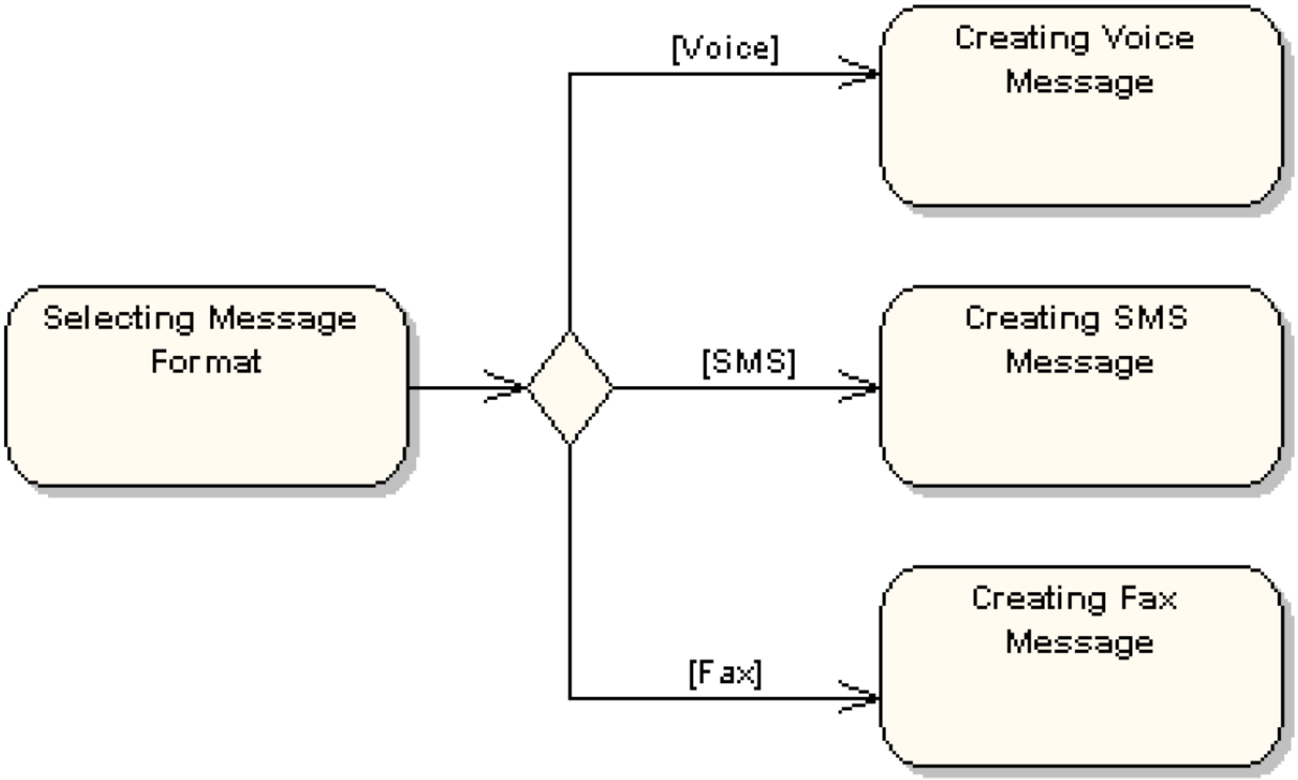
DECISIONS

Just like activity diagrams, we can use decisions nodes (although we usually call them decision pseudo-states)

Decision pseudo-states are represented with a diamond

- We always have one input transition and multiple outputs
- The branch of execution is decided by the guards associated with the transitions coming out of the decision pseudo-state

DECISIONS

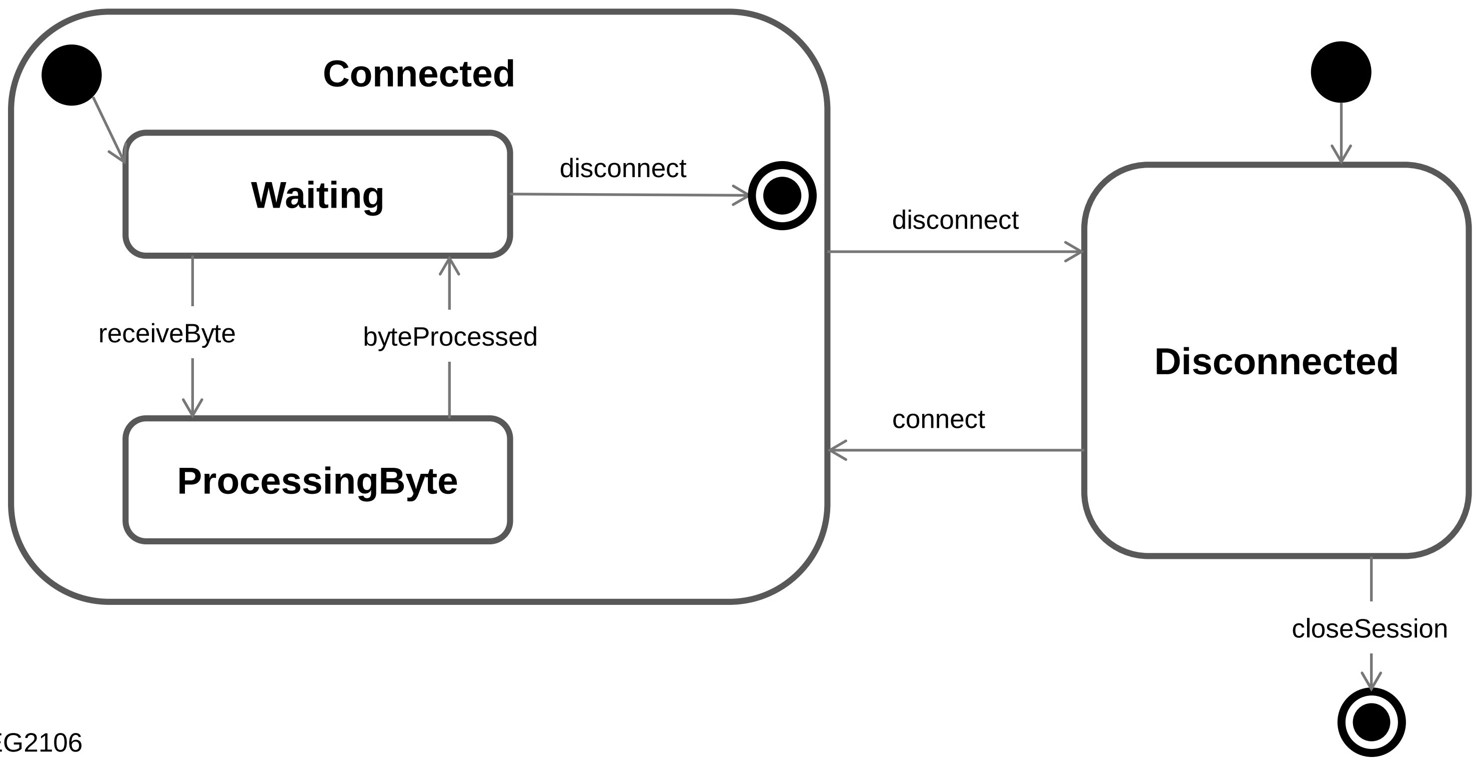


Selecting Message Format
[Voice]
[SMS]
[Fax]
Creating Voice Message
Creating SMS Message
Creating Fax Message

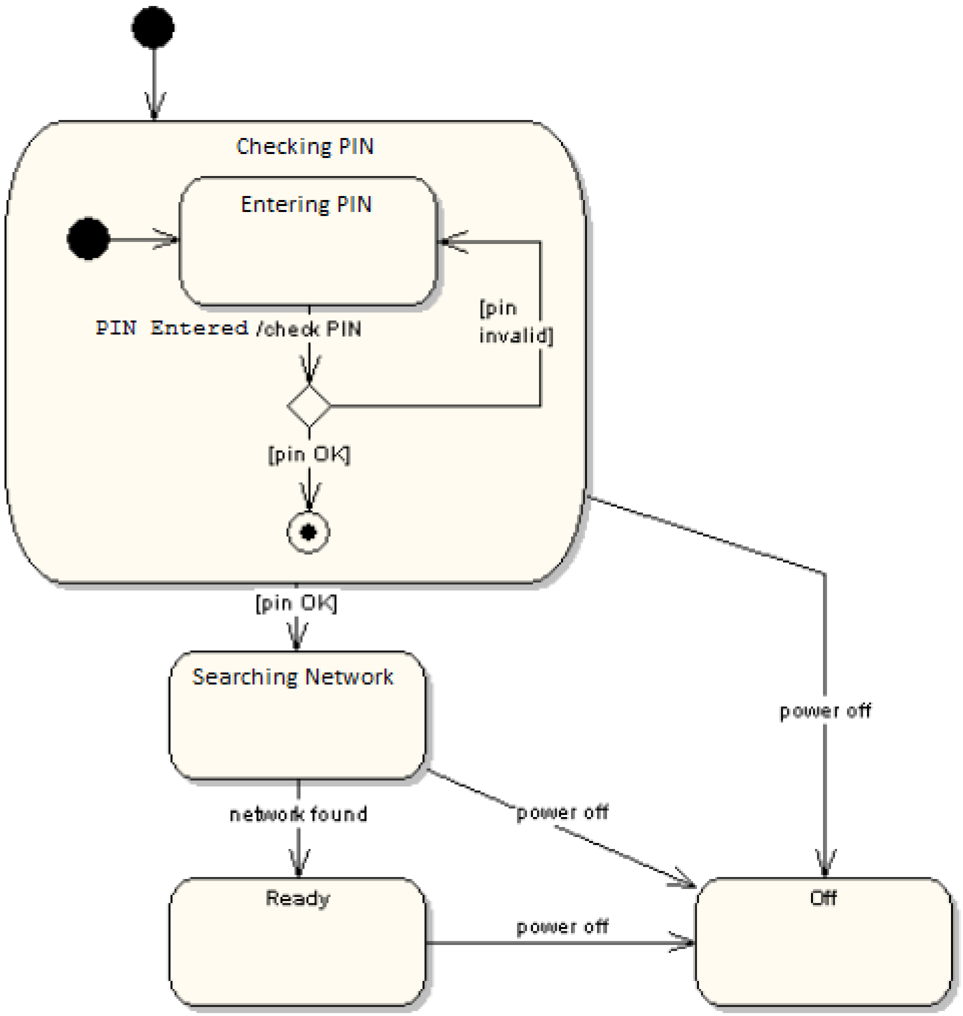
COMPOUND STATES

A state machine can include several sub-machines

Below is an example of a sub-machine included in the compound state “Connected”



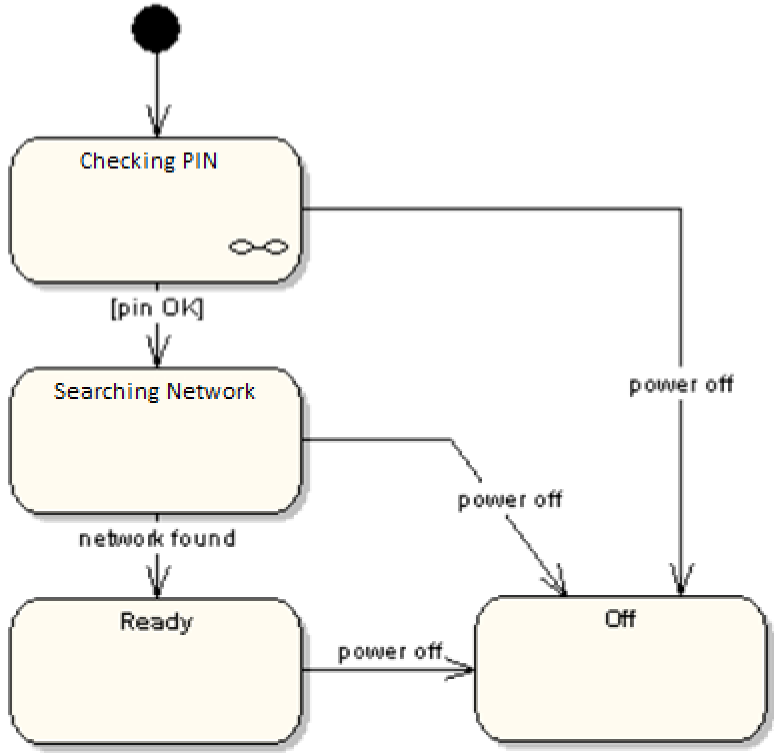
COMPOUND STATES EXAMPLE



COMPOUND STATES EXAMPLE

Same example, with an alternative notation

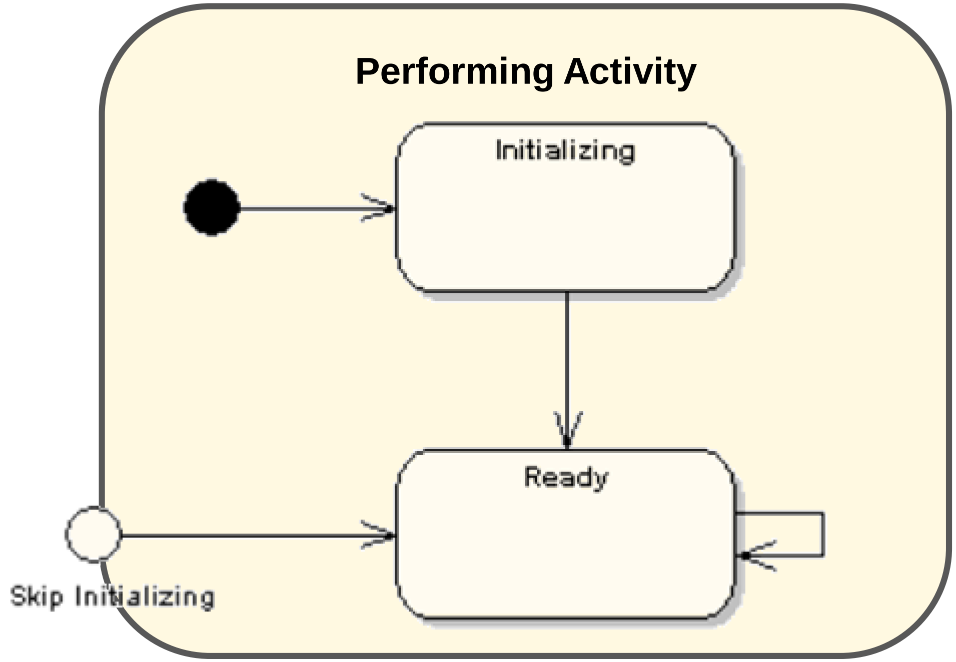
- The link symbol in the "Check Pin" state indicates that the details of the sub-machine associated with "Check Pin" are specified in an another state machine



ALTERNATIVE ENTRY POINTS

Sometimes, in a sub-machine, we do not want to start the execution from the initial state

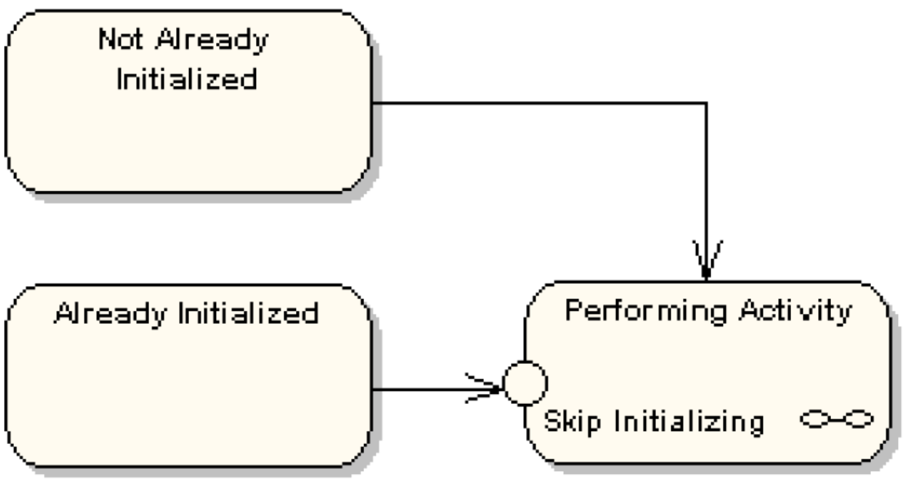
- We want to start the execution from a "name alternative entry point"



ALTERNATIVE ENTRY POINTS

Here's the same system, from a higher level

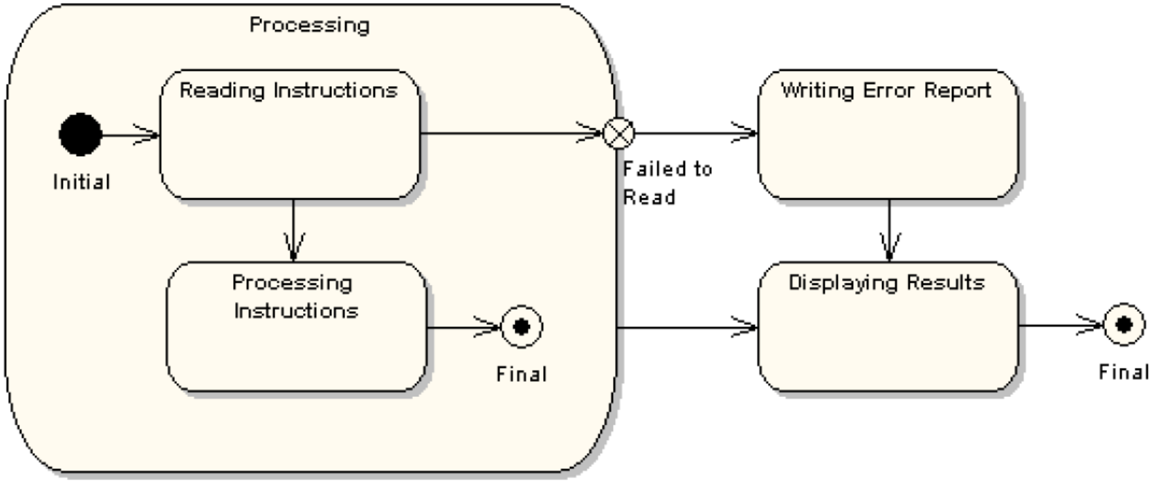
- Transition from the "No Already Initialized" state leads to the standard initial state in the sub-machine
- Transition from the "Already Initialized" state is connected to the named alternative entry point "Skip Initializing"



ALTERNATIVE EXIT POINTS

It is also possible to have alternative exit points for a compound state

- Transition from “Processing Instructions” state takes the regular exit
- Transition from the “Reading Instructions” state takes an "alternative named exit point



HISTORY STATES

A state machine describes the dynamic aspects of a process whose current behavior depends on its past

A state machine in effect specifies the legal ordering of states a process may go through during its lifetime

When a transition enters a compound state, the action of the nested state machine starts over again at its initial state

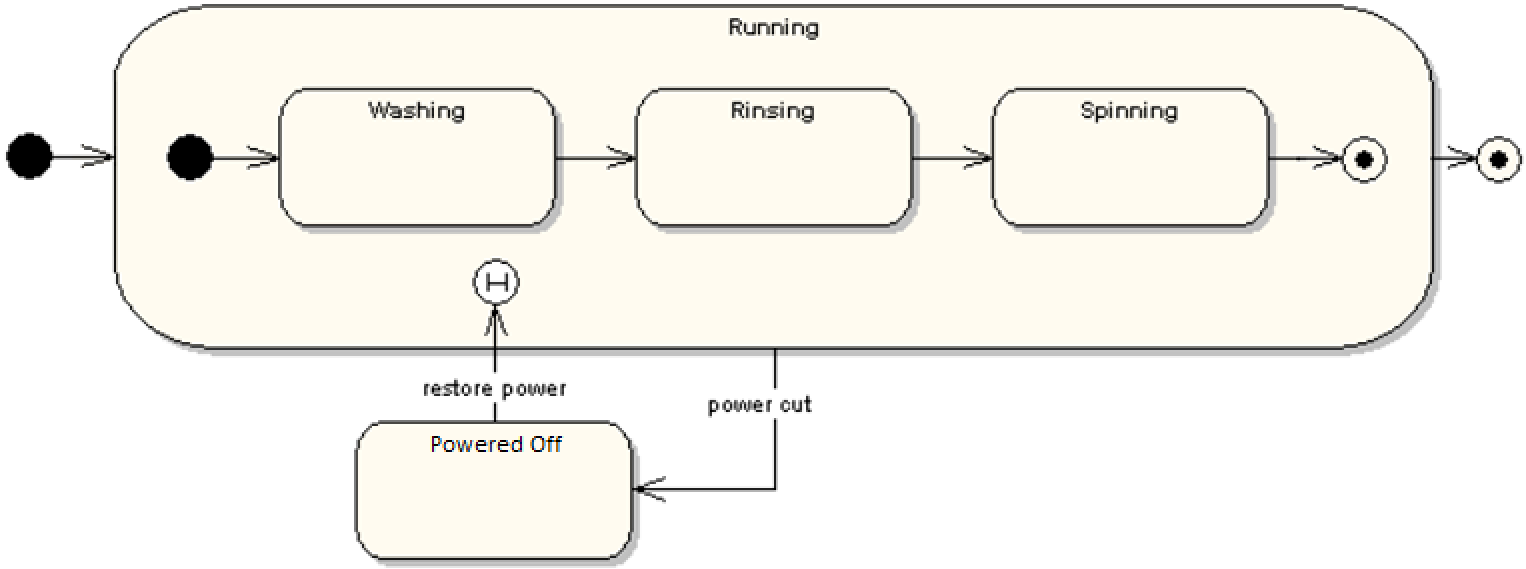
- Unless an alternative entry point is specified

There are times you'd like to model a process so that it remembers the last substate that was active prior to leaving the compound state

HISTORY STATES

Simple washing machine state diagram:

- Power Cut event: transition to the “Power Off” state
- Restore Power event: transition to the active state before the power was cut off to proceed in the cycle

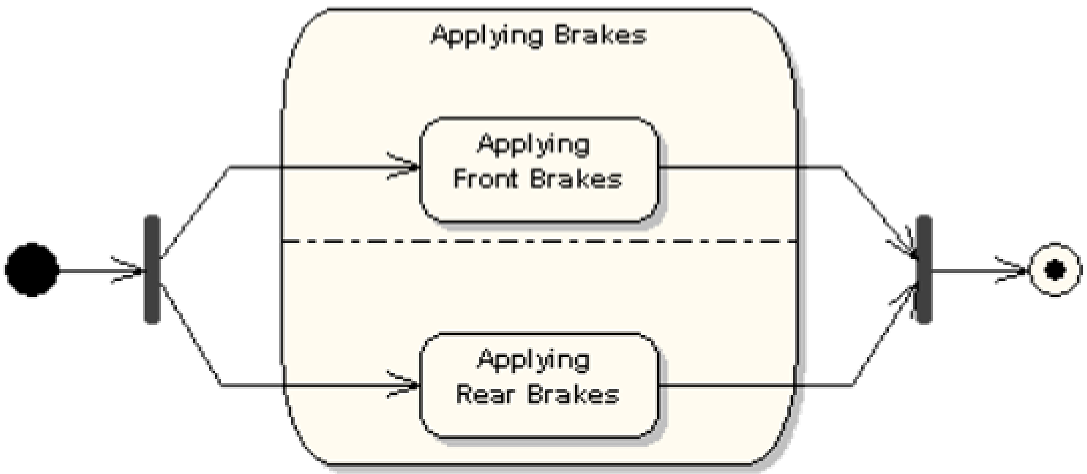


CONCURRENT REGIONS

Sequential sub state machines are the most common kind of sub machines

- In certain modeling situations, concurrent sub machines might be needed (two or more sub state machines executing in parallel)

Brakes example:



CONCURRENT REGIONS

uOttawa
L'Université canadienne
Canada's university

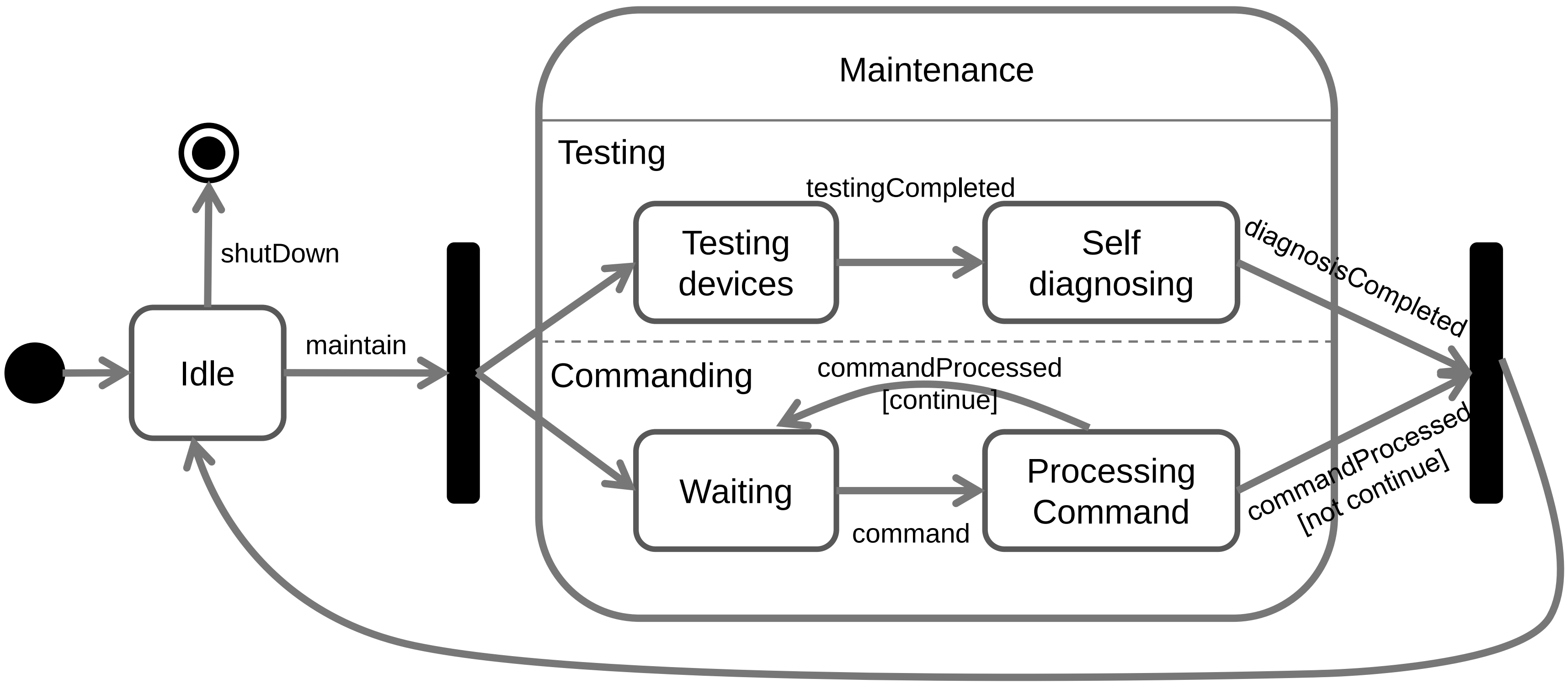
Concurrent Regions are also called Orthogonal Regions

These regions allow us to model a relationship of "And" between states (as opposed to the default "or" relationship)

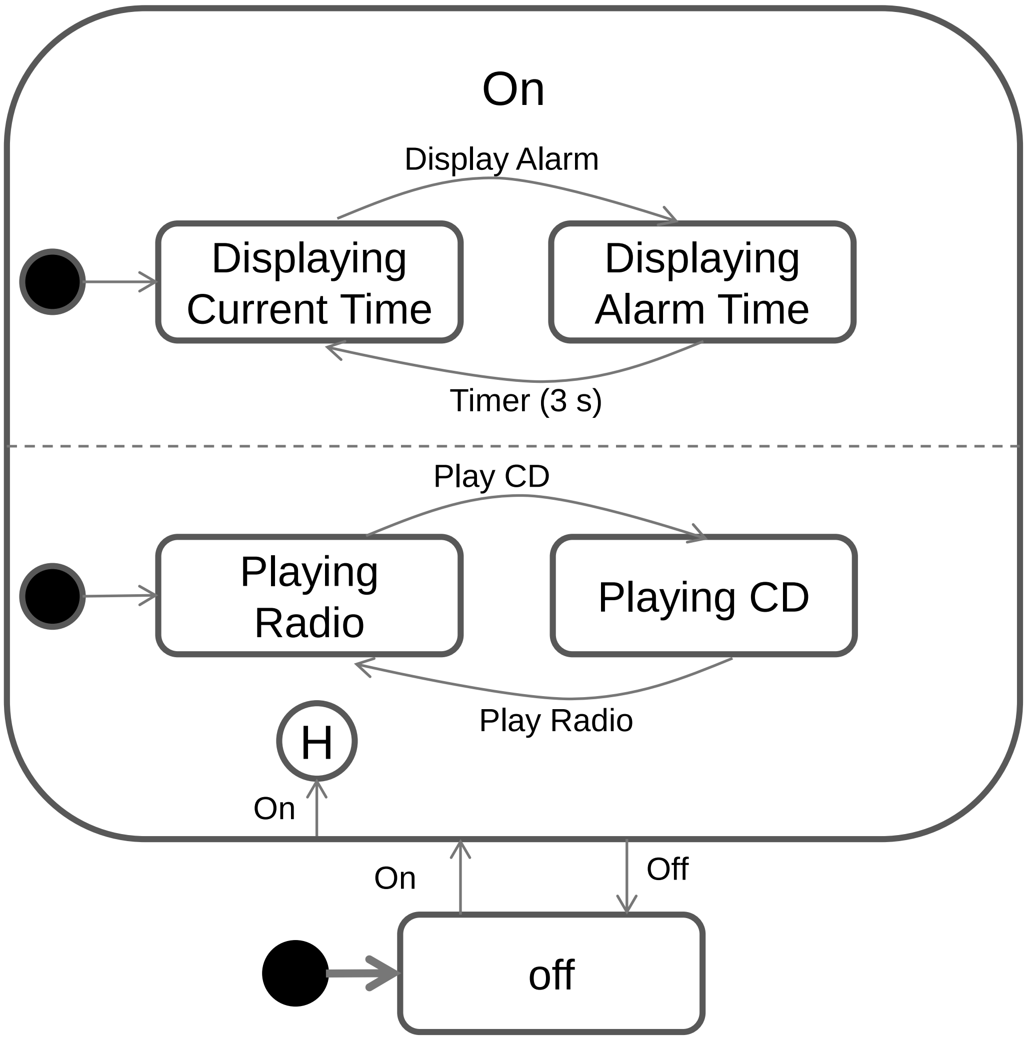
- This means that in a sub state machine, the system can be in several states simultaneously

CONCURRENT REGIONS

Example of modeling system maintenance using concurrent regions



EXAMPLE OF A CD PLAYER WITH A RADIO



GARAGE DOOR – CASE STUDY

Background

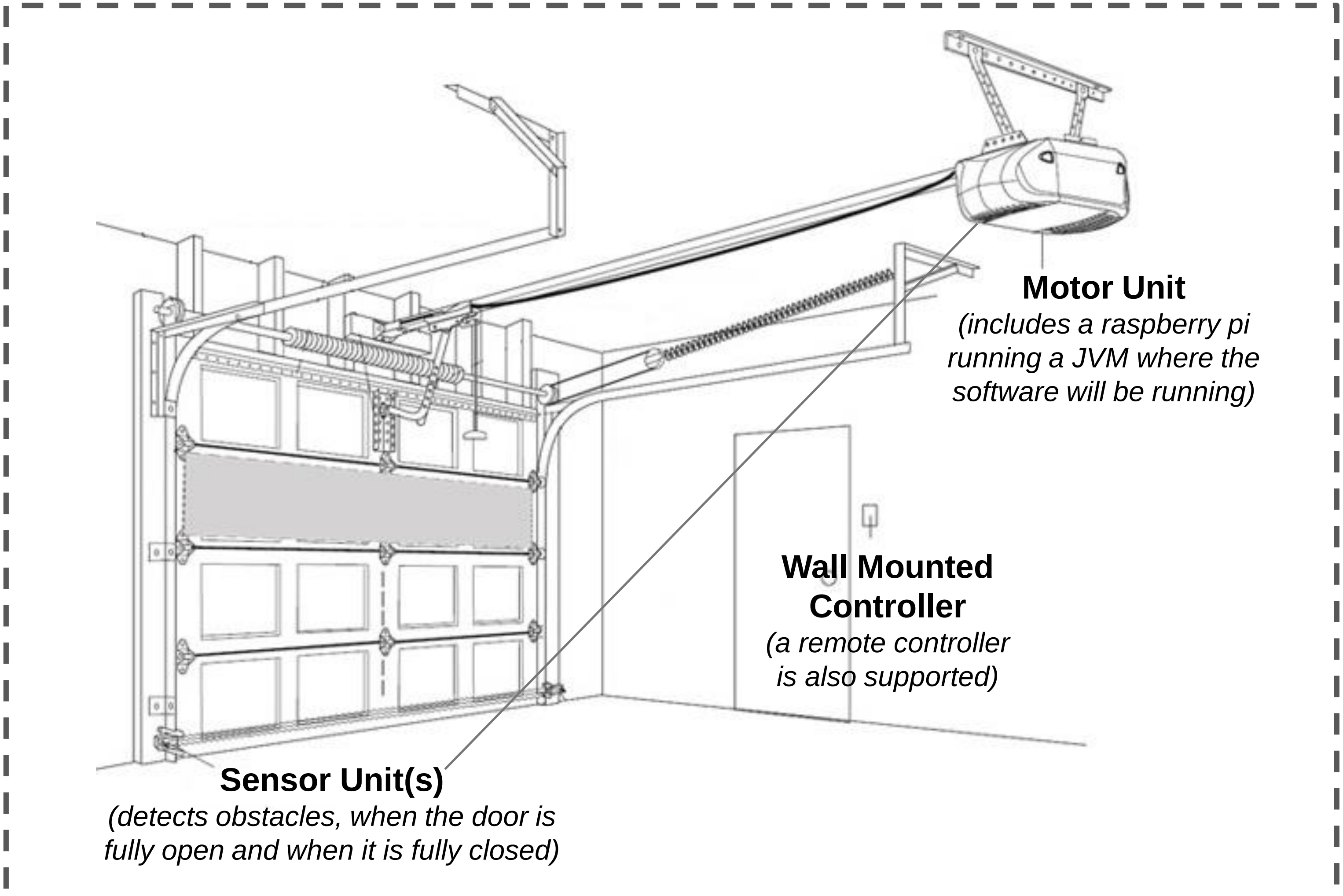
- Company DOORS inc. manufactures garage door components
- Nonetheless, they have been struggling with the embedded software running on their automated garage opener Motor Unit that they developed in house
  - This is causing them to lose business
- They decided to scrap the existing software and hire a professional software company to deliver “bug free” software

CLIENT REQUIREMENTS

Client (informal) requirements:

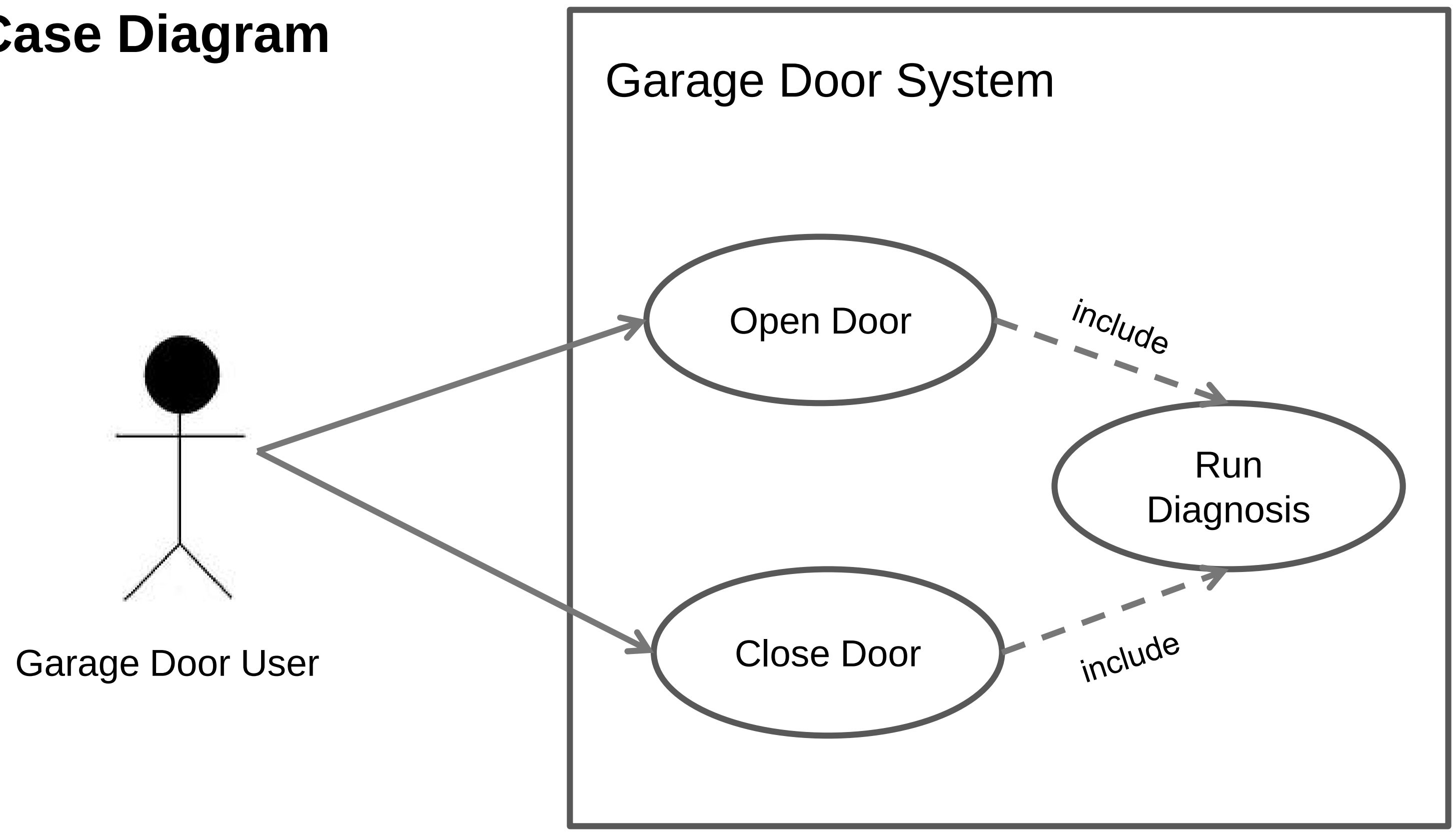
- **Requirement 1:** When the garage door is closed, it must open whenever the user presses on the button of the wall mounted door control or the remote control
- **Requirement 2:** When the garage door is open, it must close whenever the user presses on the button of the wall mounted door control or the remote control
- **Requirement 3:** The garage door should not close on an obstacle
- **Requirement 4:** There should be a way to leave the garage door half open
- **Requirement 5:** System should run a self diagnosis test before performing any command (open or close) to make sure all components are functional

CLIENT REQUIREMENTS



USE CASE DIAGRAM

Use Case Diagram



RUN DIAGNOSIS USE CASE

Use Case Name: Run Diagnosis

Summary: The system runs a self diagnosis procedure

Actor: Garage door user

Pre-Condition: User has pressed the remote or wall mounted control button

Sequence:

1. Check if the sensors are operating correctly
2. Check if the motor unit is operating correctly
3. If all checks are successful, system authorizes the command to be executed

Alternative Sequence:

Step 3: One of the checks fails and the system waits for 3 minutes and initiates a check again

Postcondition: Self diagnosis ensured that the system is operational

OPEN DOOR USE CASE

Use Case Name: Open Door

Summary: Open the garage the door

Actor: Garage door user

Dependency: Include Run Diagnosis use case

Pre-Condition: Garage door system is operational and ready to take a command

Sequence:

1. User presses the remote or wall mounted control button
2. Include Run Diagnosis use case
3. If the door is currently closing or is already closed, system opens the door

Alternative Sequence:

Step 3: If the door is open, system closes door

Step 3: If the door is currently opening, system stops the door (leaving it half open)

Postcondition: Garage door is open

CLOSE DOOR USE CASE

Use Case Name: Close Door

Summary: Close the garage the door

Actor: Garage door user

Dependency: Include Run Diagnosis use case

Pre-Condition: Garage door system is operational and ready to take a command

Sequence:

1. User presses the remote or wall mounted control button
2. Include Run Diagnosis use case
3. If the door is currently open, system closes the door

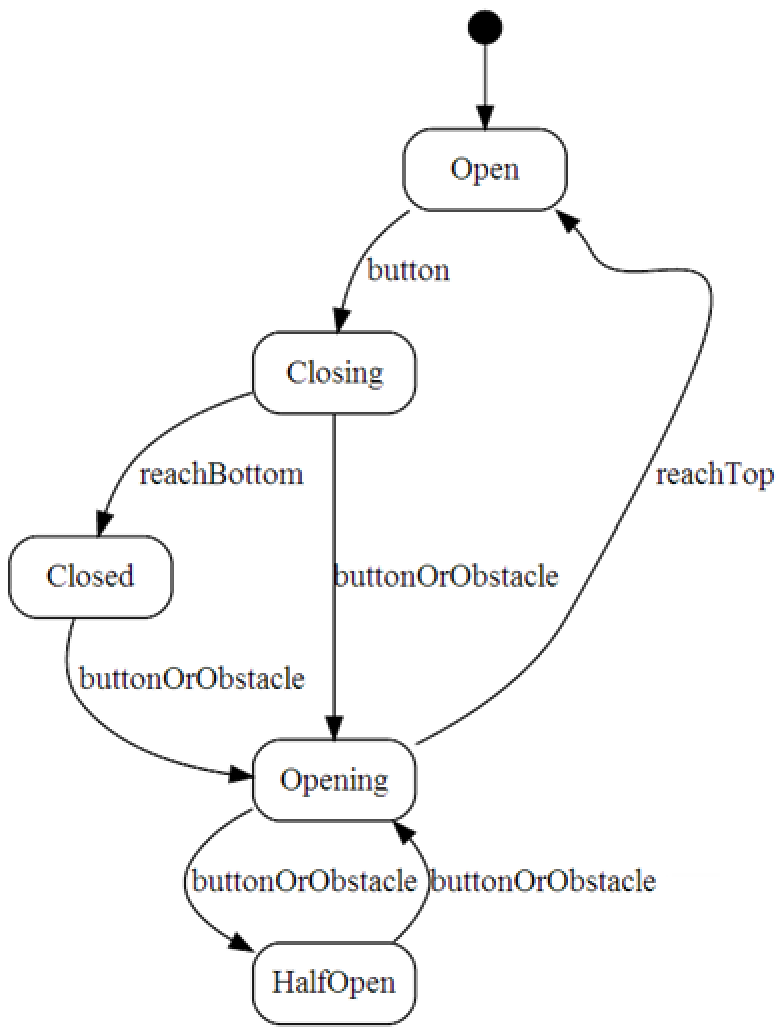
Alternative Sequence:

Step 3: If the door is currently closing or is already closed, system opens the door

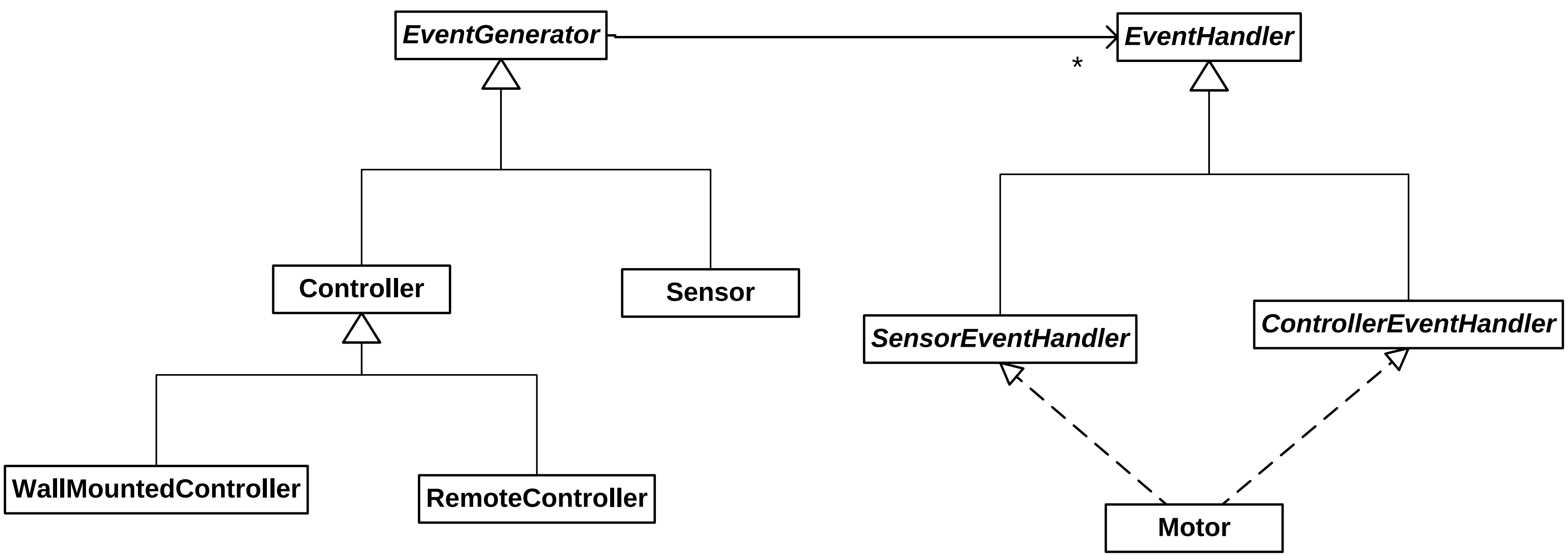
Step 3: If the door is currently opening, system stops the door (leaving it half open)

Postcondition: Garage door is closed

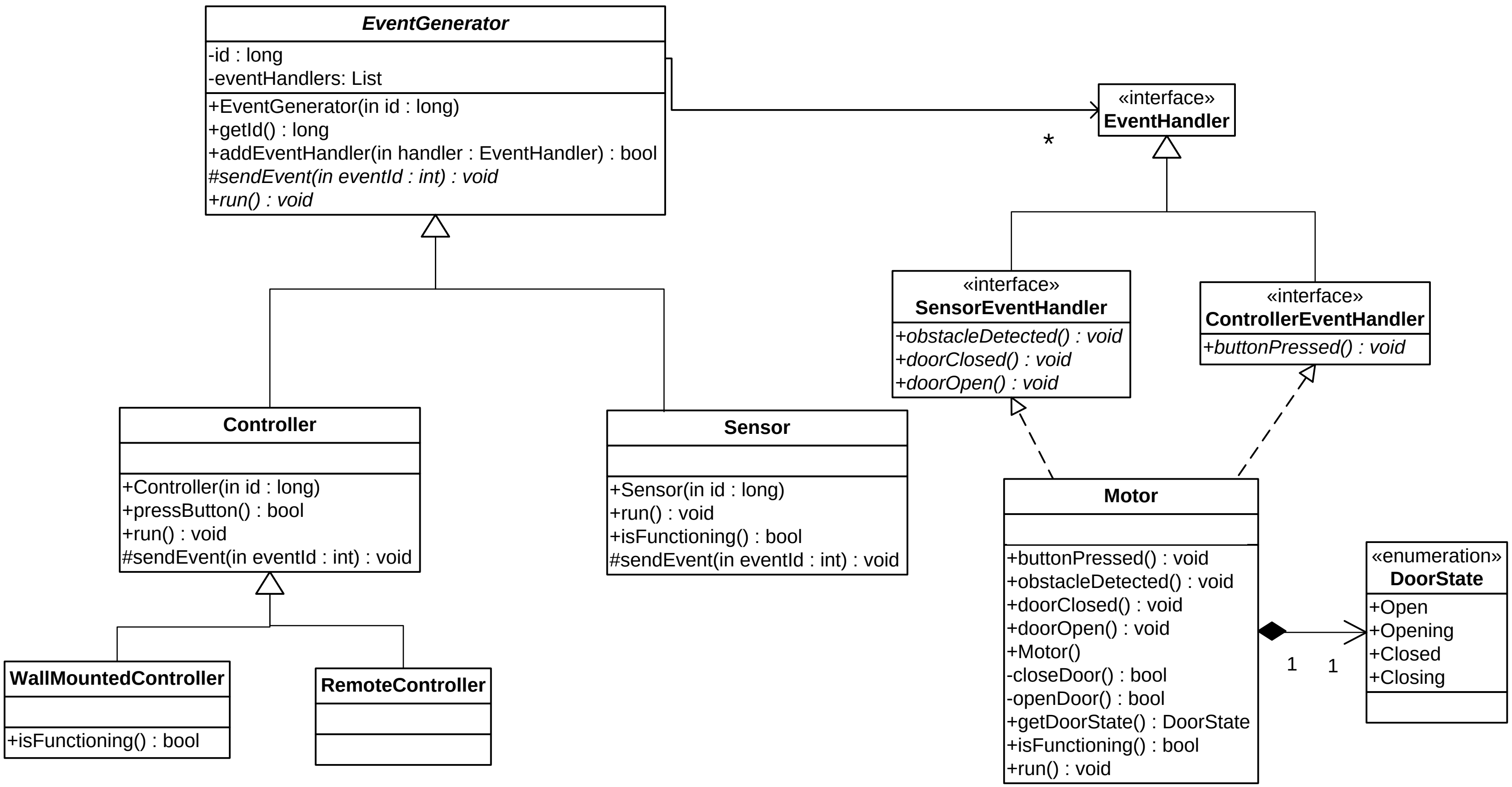
HIGH LEVEL BEHAVIORAL MODELING



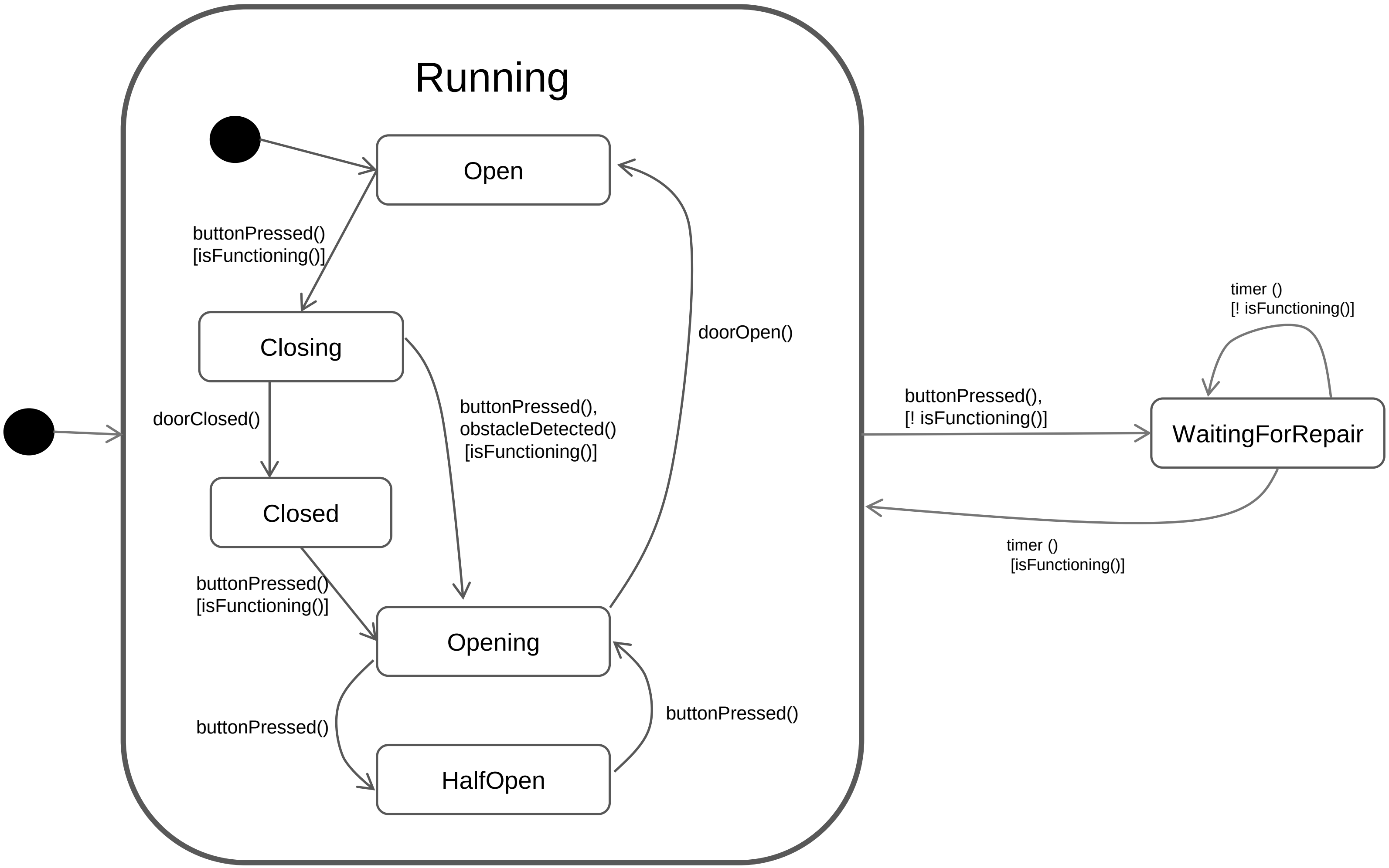
HIGH LEVEL STRUCTURAL MODEL



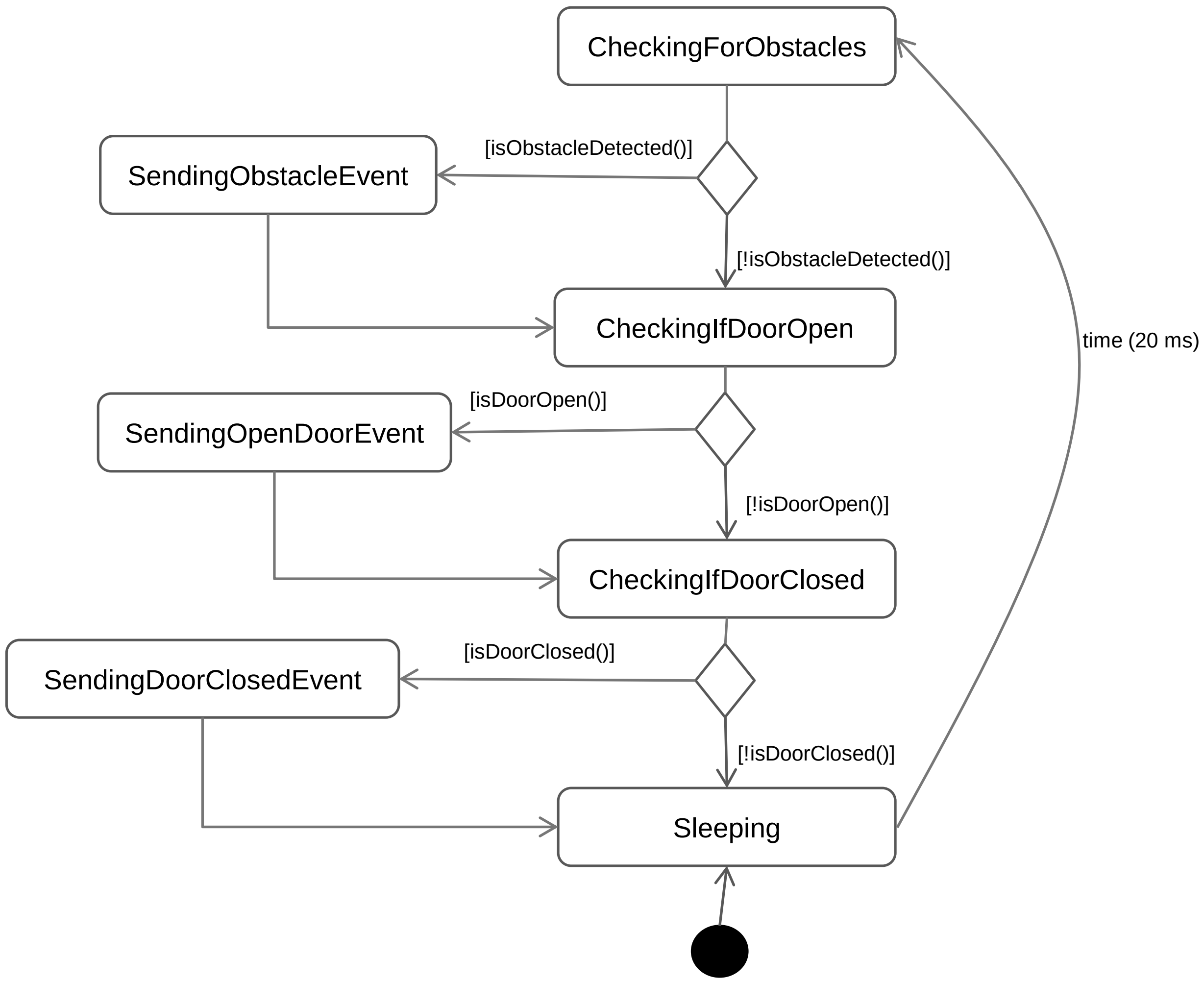
REFINED STRUCTURAL MODEL



REFINE BEHAVIORAL MODEL – MOTOR UNIT



REFINE BEHAVIORAL MODEL – SENSOR UNIT



CODING

Whenever we are satisfied with the level of detail in our behavioral models, we can proceed to coding

Some of the code can be generated directly by tools from the behavioral model

- Some tweaking might be necessary (although usually discouraged by tool providers)

EVENT GENERATOR CLASS

uOttawa
L'Université canadienne
Canada's university

```
public abstract class EventGenerator {

    // Declaring some constants
    protected static final int EVENT_HANDLERS_LIMIT = 100;
    protected static final int BUTTON_EVENT_ID = 1;
    protected static final int DOOR_CLOSED_EVENT_ID = 2;
    protected static final int DOOR_OPEN_EVENT_ID = 3;
    protected static final int OBSTACLE_EVENT_ID = 4;

    protected long id;
    protected List<EventHandler> eventHandlers;

    public EventGenerator (long id){
        this.id = id;

        eventHandlers = new LinkedList<EventHandler>();
    }

    public long getId(){
        return id;
    }

    public boolean addEventHandler(EventHandler handler){
        if (eventHandlers.size() < EVENT_HANDLERS_LIMIT){
            eventHandlers.add(handler);
            return true;
        }
        else {
            return false;
        }
    }

    protected abstract void sendEvent(int eventId);

    public abstract void run();
}
```

SENSOR CLASS

```
public class Sensor extends EventGenerator implements Runnable{
    private static final int POLLING_INTERVAL = 20; // 20 milliseconds
    private boolean running;
    private Thread pollingThread;

    public Sensor(long id){
        super (id);
        running = true;

        pollingThread = new Thread(this);

        pollingThread.start();
    }

    @Override
    protected void sendEvent(int eventId) {
        for (EventHandler handler: eventHandlers){

            if (handler instanceof SensorEventHandler){
                SensorEventHandler sensorHandler = (SensorEventHandler)handler;

                if (eventId == OBSTACLE_EVENT_ID){
                    sensorHandler.obstacleDetected();
                }
                else if (eventId == DOOR_CLOSED_EVENT_ID){
                    sensorHandler.doorClosed();
                }
                else if (eventId == DOOR_OPEN_EVENT_ID){
                    sensorHandler.doorOpen();
                }
            }
        }
    }
```

SENSOR CLASS

```
@Override
public void run() {
    // poll sensor hardware continuously in order to check
    // whether there is an obstacle, the door has reached the top (is open)
    // or the door has reached the bottom (is closed)
    while (running) {
        try {
            Thread.sleep(POLLING_INTERVAL);
            // Check for an obstacle
            boolean obstacleDetected = HardwareSensorInterface.isObstacleDetected();
            if (obstacleDetected){
                sendEvent(OBSTACLE_EVENT_ID);
            }
            // Check if door is open
            //(this call returns true only the first time it is called after the door is open)
            boolean doorOpen = HardwareSensorInterface.isDoorOpen();
            if (doorOpen){
                sendEvent(DOOR_OPEN_EVENT_ID);
            }
            // Check if door is closed
            //(this call returns true only the first time it is called after the door is closed)
            boolean doorClosed = HardwareSensorInterface.isDoorClosed();

            if (doorClosed){
                sendEvent(DOOR_CLOSED_EVENT_ID);
            }

        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
}
```

Sensor
State machine
Implementation

UMPLE ONLINE DEMO

UMPLE is a modeling tool to enable what we call Model-Oriented Programming

You can use it to create class diagrams (structural models) and state machines (behavioral models)

The tool was developed at the university of Ottawa

- Online version can be found at:

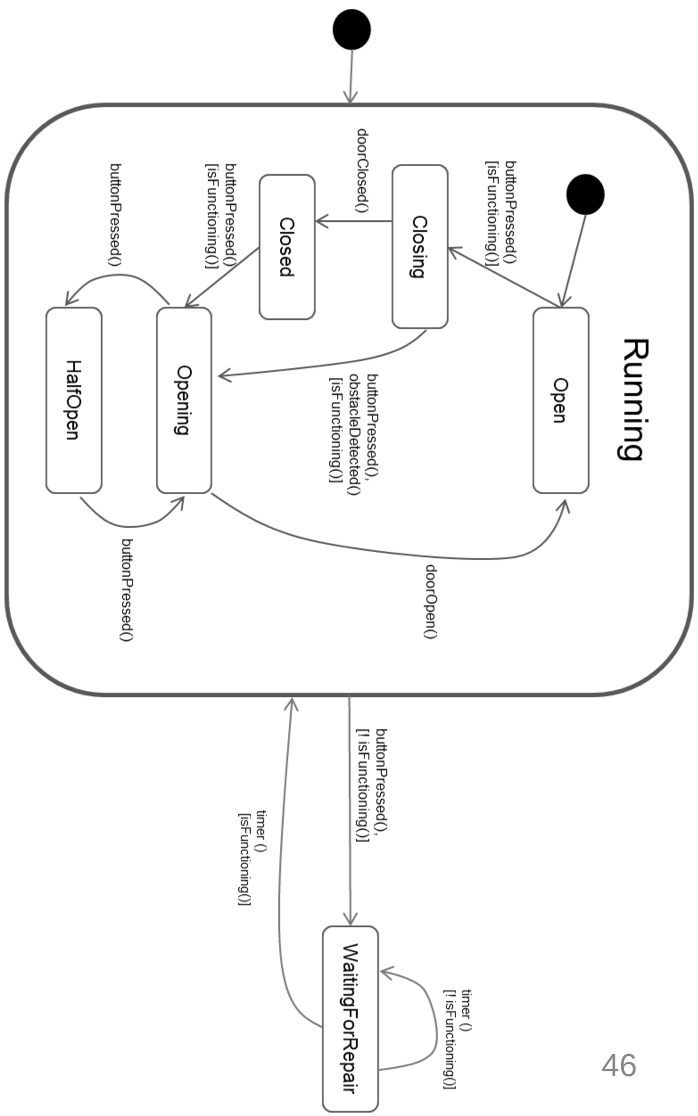
http://cruise.eecs.uottawa.ca/umpleonline/

- There’s also an eclipse plugin for the tool

UMPLE CODE FOR MOTOR UNIT STATE MACHINE

```
class Motor {
    status {
        Running {
            Open {buttonPressed[isFunctioning()]->Closing; }
            Closing { buttonPressed()[isFunctioning()]->Opening;
                ObstacleDetected()[isFunctioning()]->Opening;
                doorClosed()->Closed;}
            Closed { buttonPressed()[isFunctioning()]->Opening; }
            Opening { buttonPressed()->HalfOpen;
                doorOpen()->Open; }
            HalfOpen{buttonPressed()->Opening;}

            buttonPressed()[!isFunctioning()]->WaitingForRepair;
    }
        WaitingForRepair{
            timer()[!isFunctioning()]->WaitingForRepair;
            timer()[isFunctioning()]->Running;}
    }
}
```



MOTOR CLASS SNIPPETS

```
public boolean buttonPressed(){
  boolean wasEventProcessed = false;
  Status aStatus = status;
  StatusRunning aStatusRunning = statusRunning;
  switch (aStatus){
    case Running:
      if (!isFunctioning()){
        exitStatus();
        setStatus(Status.WaitingForRepair);
        wasEventProcessed = true;
        break;
      }
      break;
    default: //Other states do respond to this event
  }
  switch (aStatusRunning){
    case Open:
      if (isFunctioning())
      {
        setStatusRunning(StatusRunning.Closing);
        wasEventProcessed = true;
        break;
      }
      break;
    case Closing:
      if (isFunctioning()){
        setStatusRunning(StatusRunning.Opening);
        wasEventProcessed = true;
        break;
      }
      break;
    case Closed:
      if (isFunctioning()){
        setStatusRunning(StatusRunning.Opening);
        wasEventProcessed = true;
        break;
      }
      break;
    case Opening:
      setStatusRunning(StatusRunning.HalfOpen);
      wasEventProcessed = true;
      break;
    case HalfOpen:
      setStatusRunning(StatusRunning.Opening);
      wasEventProcessed = true;
      break;
    default:// Other states do respond to this event
  }
  return wasEventProcessed;
}
```

Switching between high level states

Switching between nest states inside the Running compound state

uOttawa

L'Université canadienne
Canada's university

THANK YOU!

QUESTIONS?