

SÉANCE 4

MACHINES D'ÉTATS UML

SUJETS

États

Transitions

Gardes

Effets

Actions d'État

Décisions

États composés

Entrée et sortie alternatives

États historique

Étude de cas

DIAGRAMME D'ACTIVITÉ VS MACHINES D'ÉTATS

Pour les diagrammes activités

- Sommet représente les Actions
- Arrêt (flèche) représente une transition qui va avoir lieu après la complétion d'une action et avant le début d'une autre (flux de contrôle)

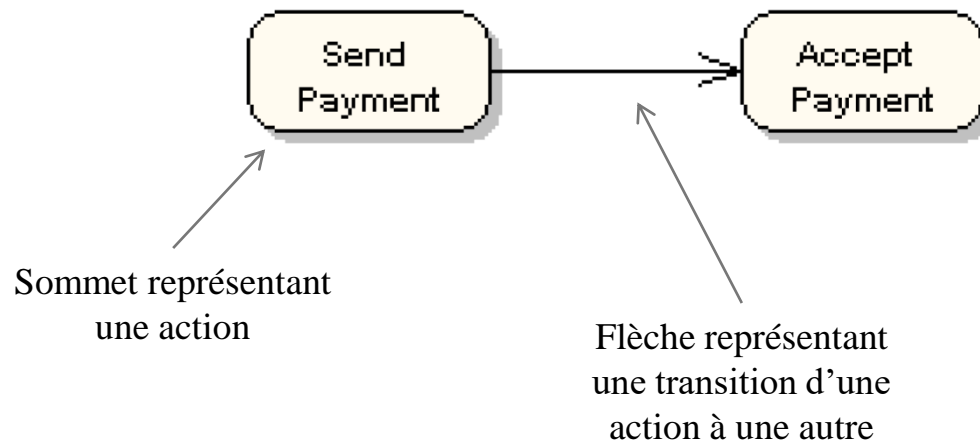
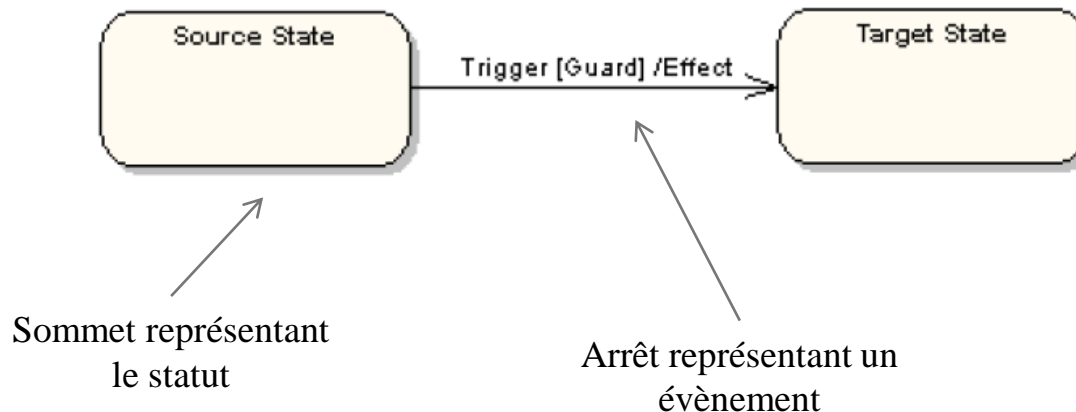


DIAGRAMME D'ACTIVITÉ VS MACHINES D'ÉTATS

Pour les machines d'états

- Sommet représente le statut (état) d'un processus
- Arrêt (flèche) représente un évènement



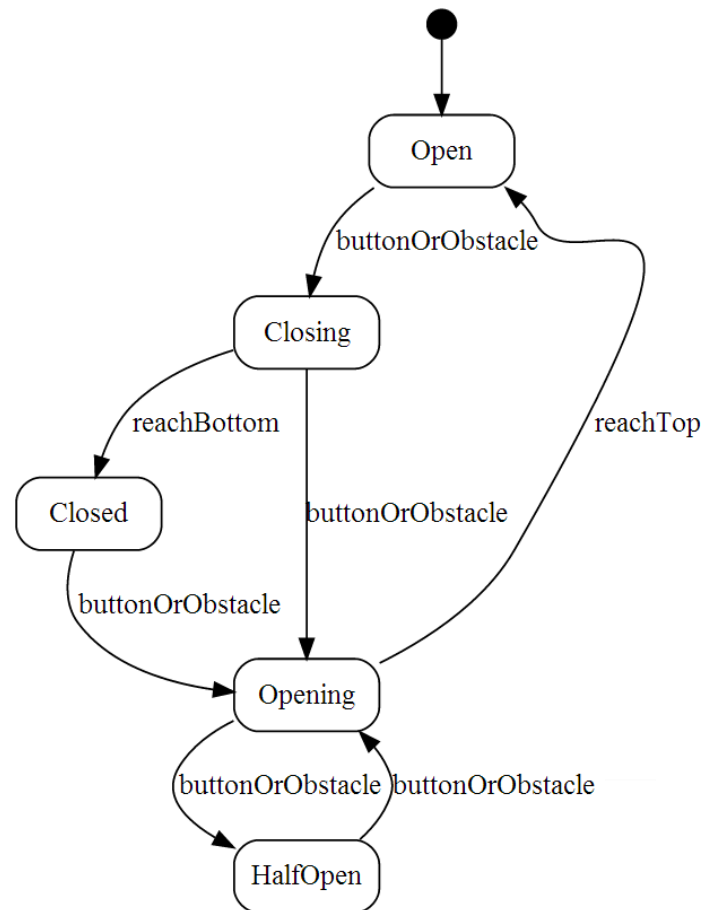
MACHINES D'ÉTATS UML

Utilisé pour modéliser le comportement dynamique d'un processus

- Peut être utilisé pour modéliser de haut niveau le comportement général d'un système entier
- Peut être utilisé pour modéliser le comportement détaillé d'un seul objet
- Tout autre niveau de détail entre ces deux extrêmes est possible

EXEMPLE D'UNE MACHINE D'ÉTATS

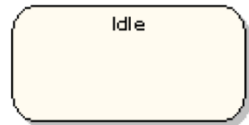
Exemple Pour Modéliser les états d'une porte de garage



[Cliquer ici](#)

ÉTATS

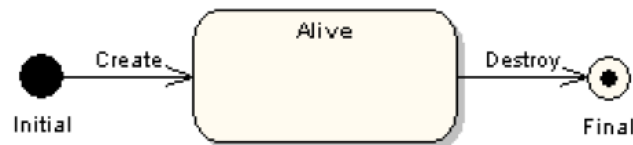
Symbole pour un état



Un système dans un état va rester dans cet état jusqu'à l'occurrence d'un évènement qui lui cause de changer d'état

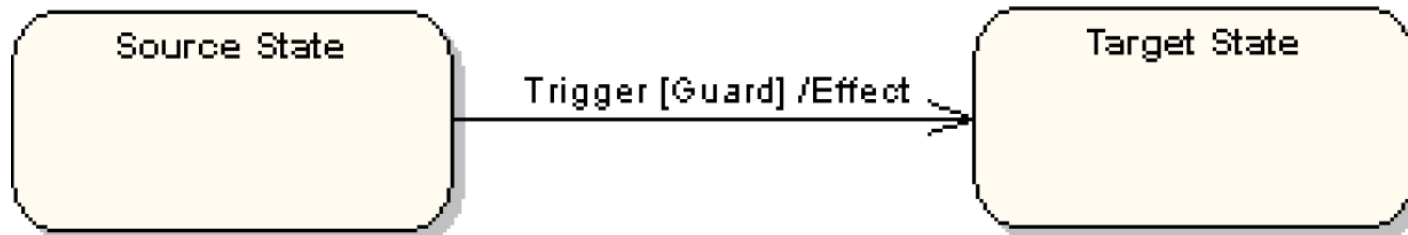
- Être dans un état veut dire que le système va se comporter d'une façon spécifique en réponse à un évènement qui se produit

Symboles pour les états initial et final



TRANSITIONS

Transitions sont représentées par des flèches



TRANSITIONS

Une transition représente un changement d'état en réponse à un évènement

- Il est supposé se produire d'une façon instantanée (ça ne prend pas du temps)

Une transition peut avoir

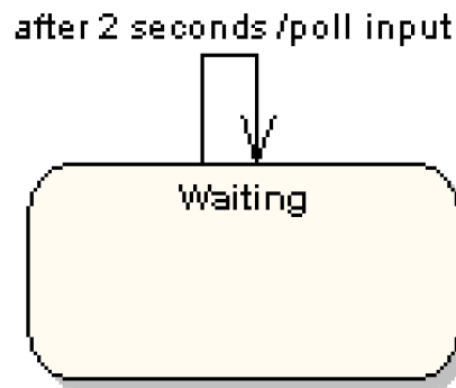
- **Un Déclencheur (Trigger):** la cause de la transition qui peut être un évènement
- **Une Garde (Guard):** une condition qui doit être vraie pour que le déclencheur cause la transition
- **Un Effet (Effect):** une action qui va être invoquée directement sur le système ou l'objet modélisé lorsque la transition se produit

AUTO TRANSITION

Un état peut avoir une transition qui est dirigée vers lui-même: auto transition

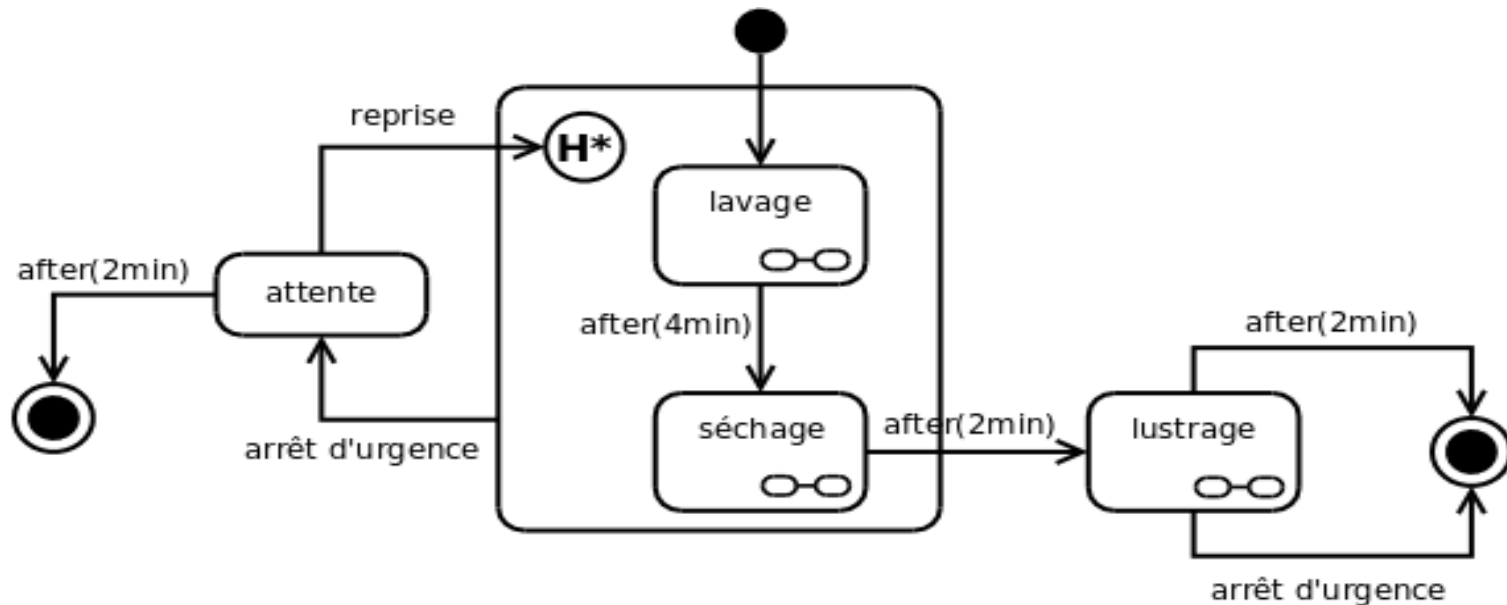
- Ces auto-transitions sont plus utiles quand elles sont associées avec un effet

Ceci est plus utile quand un effet est associé avec la transition





TRANSITIONS D'ACHÈVEMENT



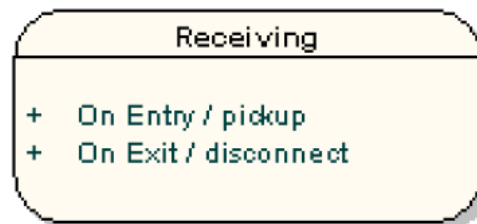
Transition d'achèvement: transition sans déclencheur initié implicitement lorsque l'état source a complété son comportement

ACTIONS D'ÉTATS

Un effet peut être associé avec une transition

Si l'état de destination est associé avec plusieurs transitions d'arrivée (*plusieurs transitions arrive a cet état*), et chaque transition possède le même effet:

- Ce serait mieux d'associer l'effet avec les états (au lieu des transitions)
- On peut réaliser cela en utilisant un effet d'entrée
- On peut aussi ajouter un effet de sortie



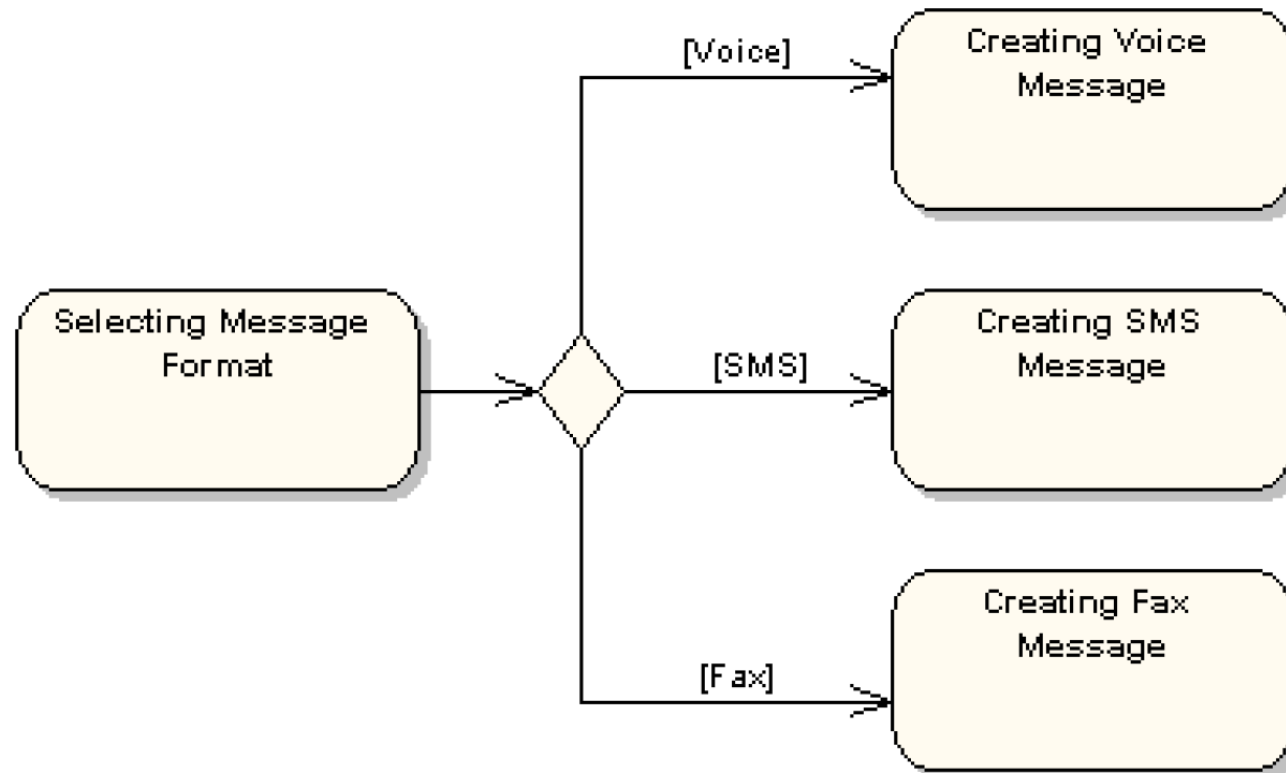
DÉCISIONS

Comme dans les diagrammes d'activité, on peut utiliser des nœuds de décisions (on les appelle pseudo-états)

Les nœuds de décisions sont représentés par des symboles de diamants

- On a toujours une transition d'arrivée et deux ou plus transitions de départ
- La branche d'exécution est décidée par la condition (garde) associée avec les transitions qui sortent du nœud de décision

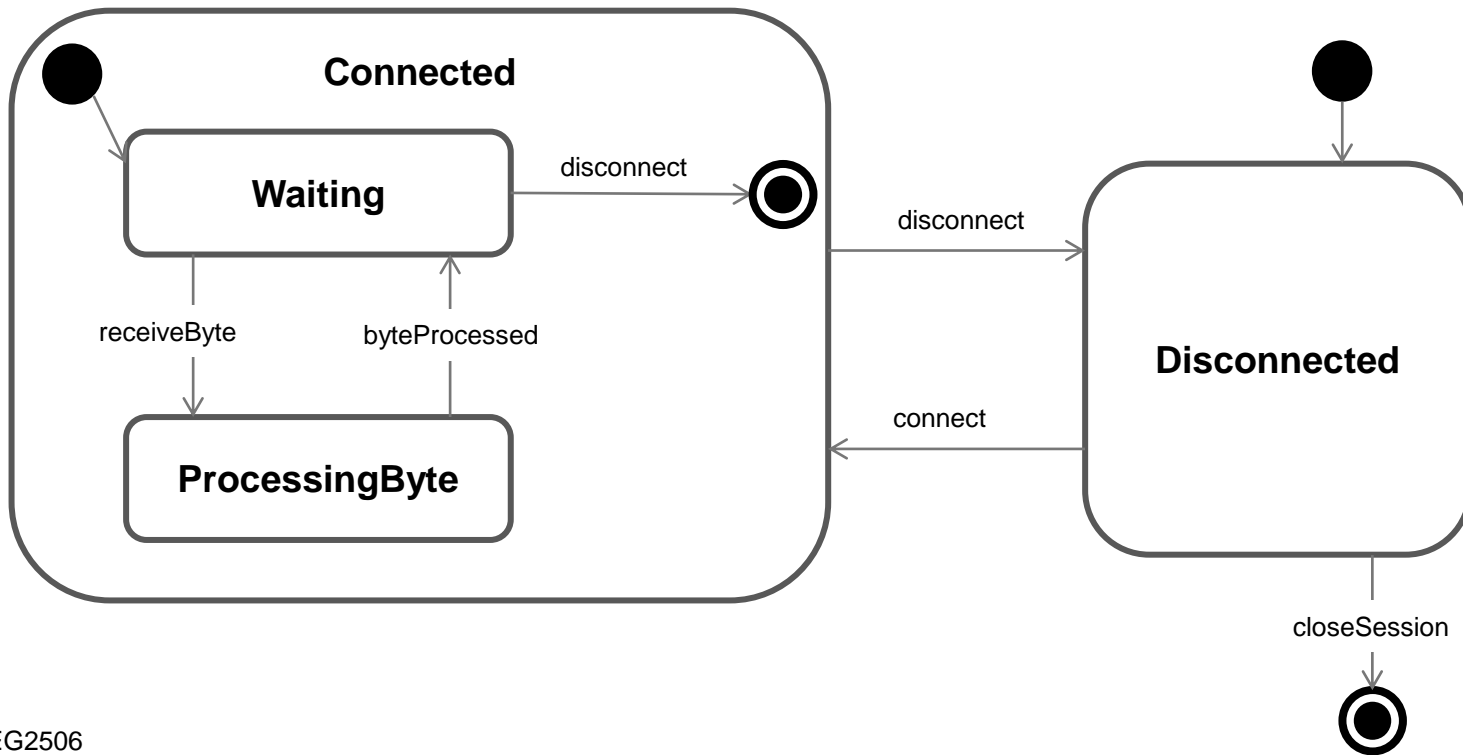
DÉCISIONS



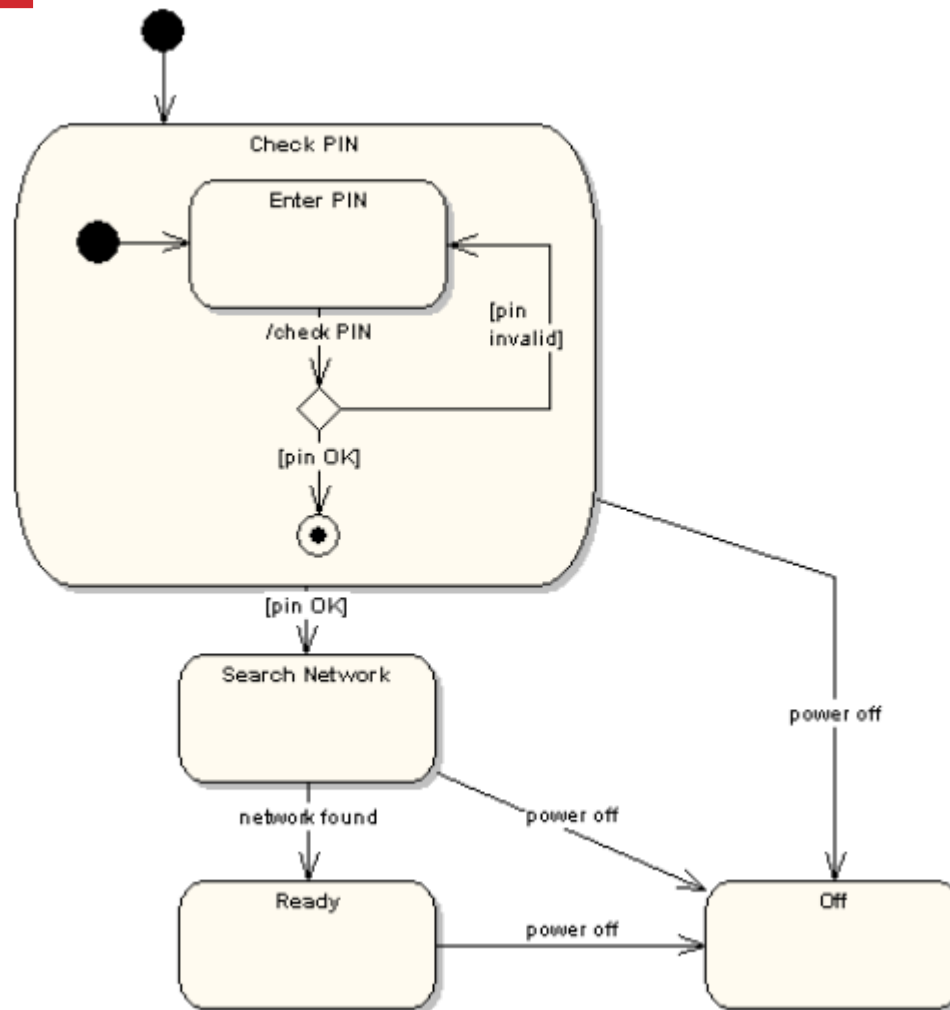
ÉTATS COMPOSÉS

Une machine d'état peut inclure des diagrammes de sous-machine

Exemple d'une application de communication simple:



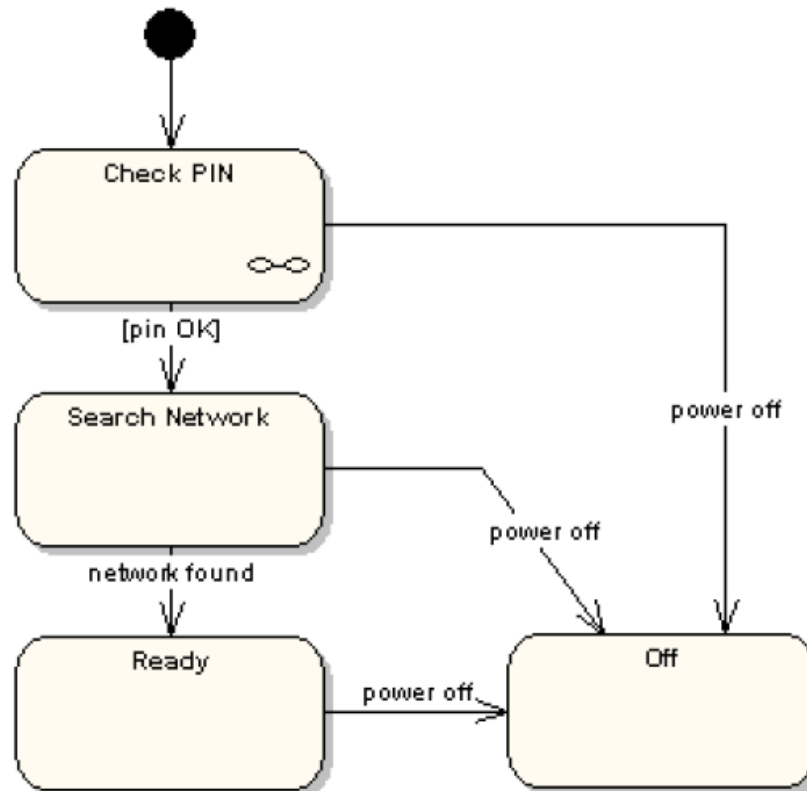
ÉTATS COMPOSÉS - EXEMPLE



ÉTATS COMPOSÉS - EXEMPLE

Même exemple, mais avec une représentation alternative

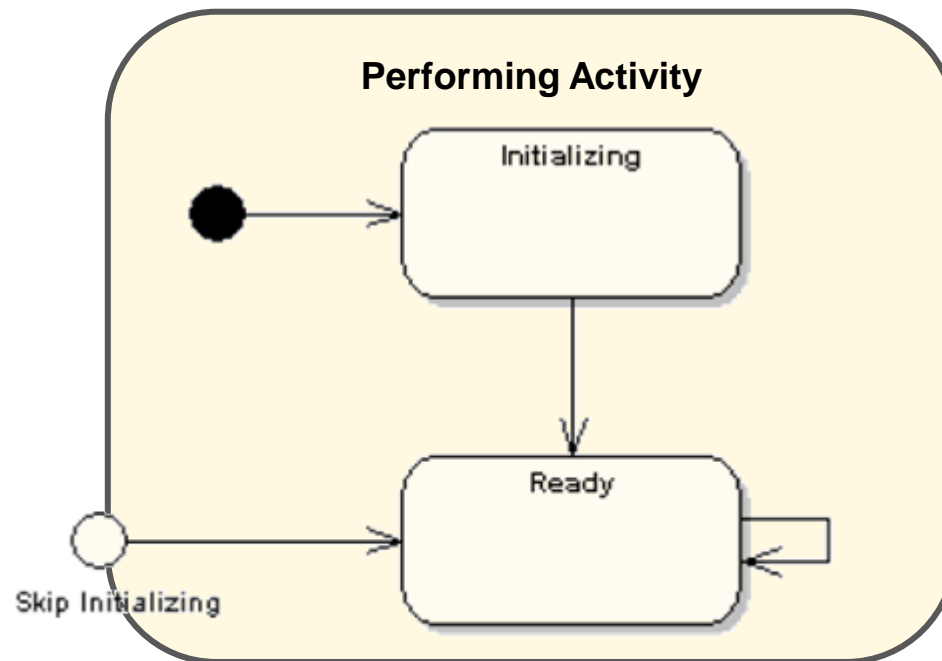
- Le symbole dans l'état "Check Pin" indique que les détails de la sous-machine sont spécifiés dans une autre machine d'état



POINTS D'ENTRÉE

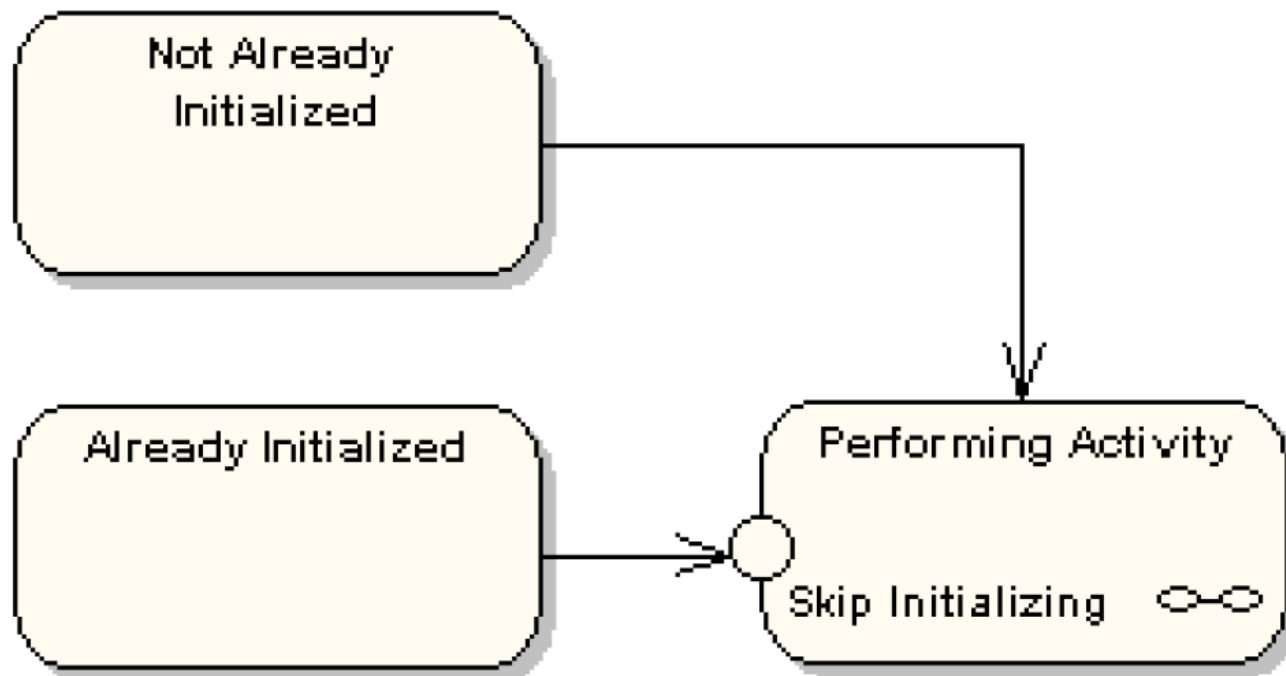
Parfois, on n'a pas besoin de commencer l'exécution par l'état initial

- Il arrive des fois qu'on veut commencer l'exécution à partir d'un point alternatif



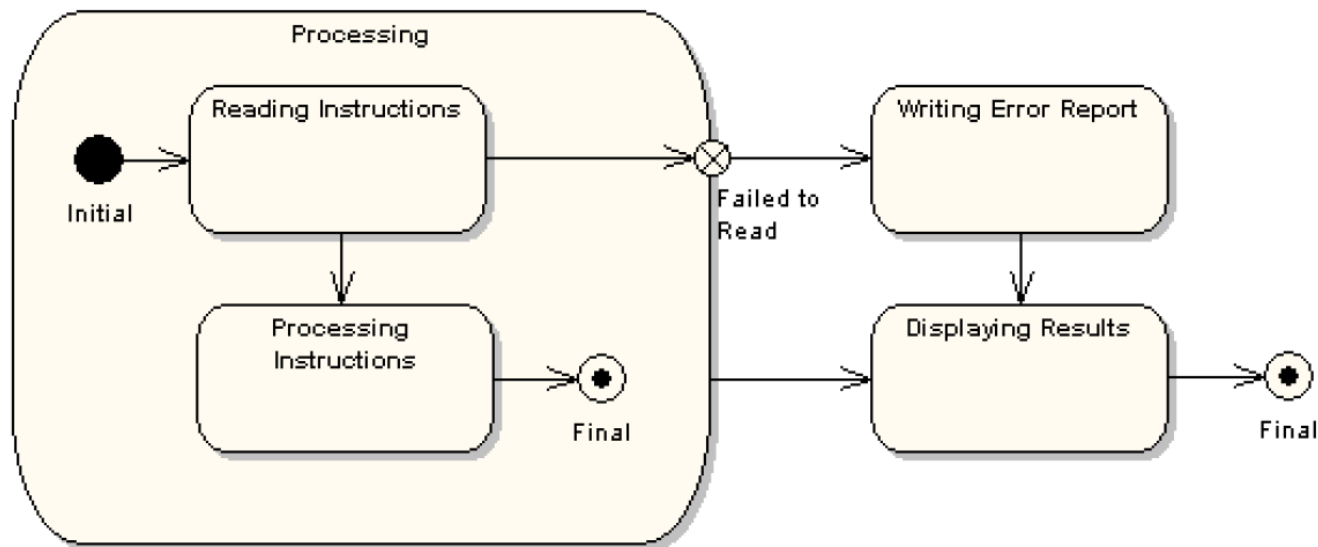
POINTS D'ENTRÉE

Voici le même système qu'on vient de voir, d'un niveau plus haut



POINTS DE SORTIE ALTERNATIVE

Il est aussi possible d'avoir des points de sortie alternative d'un état composé



ÉTATS D'HISTOIRE

Une machine d'état décrit les aspects dynamiques d'un processus dont le comportement actuel dépend de son passé

Une machine d'état en vigueur spécifie l'ordre légal des états dont un processus peut passer au cours de sa vie

Quand une transition entre dans un état composé, l'action de la sous machine d'état recommence à son état initial

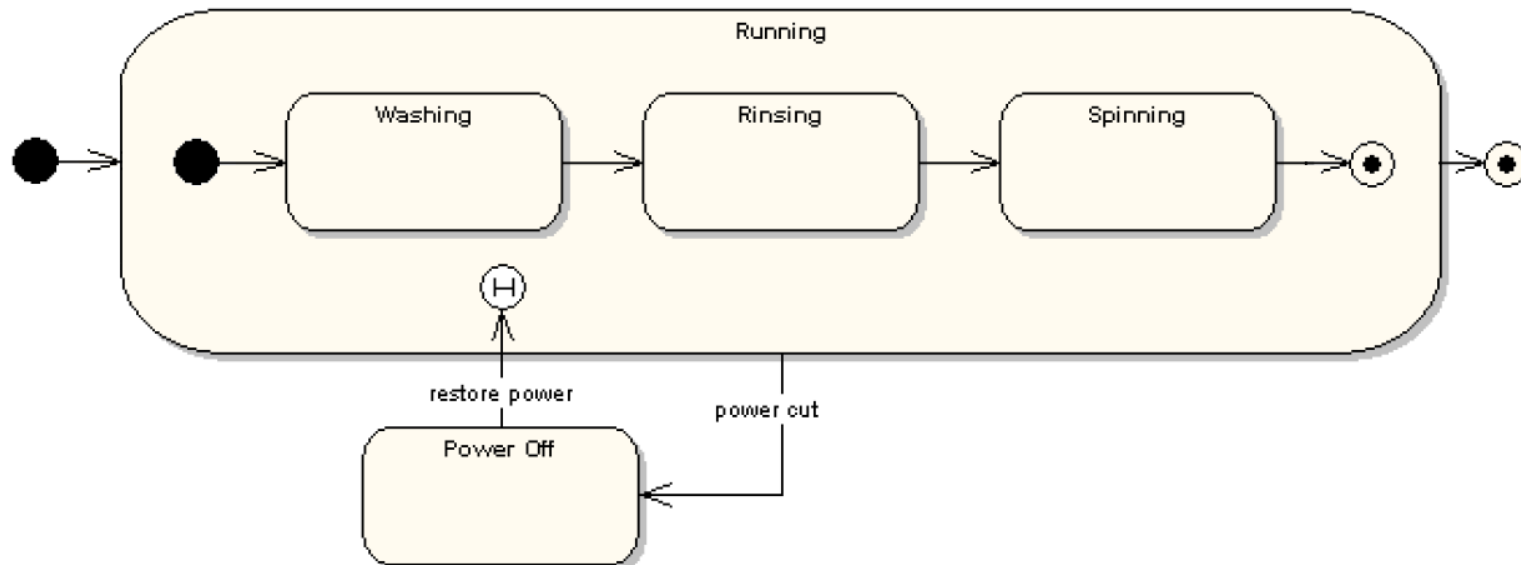
- Sauf si un point d'entrée alternatif est spécifié

Il y a des moments où vous souhaitez modéliser un processus de sorte qu'il se souvienne du dernier sous-état qui était actif avant de quitter l'état composé

ÉTATS D'HISTOIRE

Diagramme d'état simple d'une machine à laver:

- Événement coupure du courant électrique: transition à l'état "Power Off"
- Événement courant restauré: transition à l'état actif avant la coupure du courant pour continuer le cycle

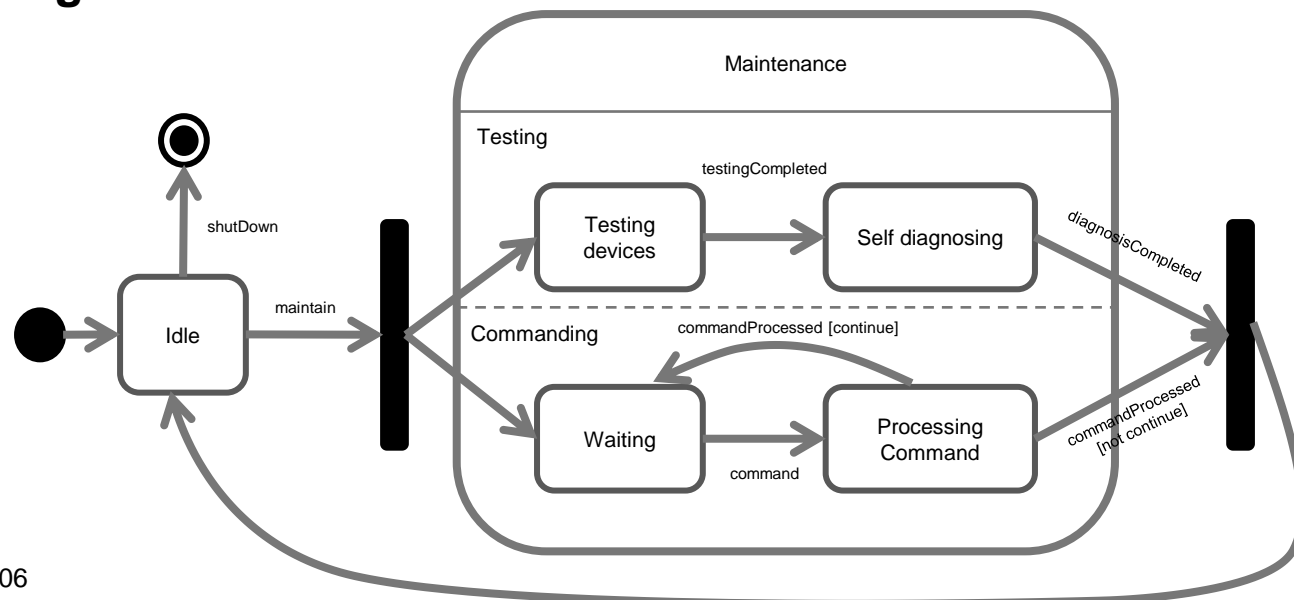


RÉGIONS CONCURRENTIELLES

Les machines de sous-état séquentielles sont le genre de sous-machines les plus connues

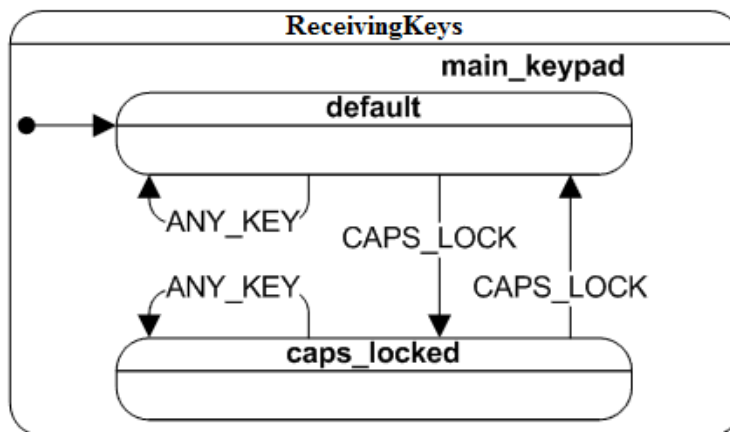
- Dans certaines situations de modélisation, on fait appel à des sous machines concurrentielles (deux machines de sous-état ou plus qui travaillent simultanément)

Exemple de modélisation d'un système de maintenance qui utilise des régions concurrentielles



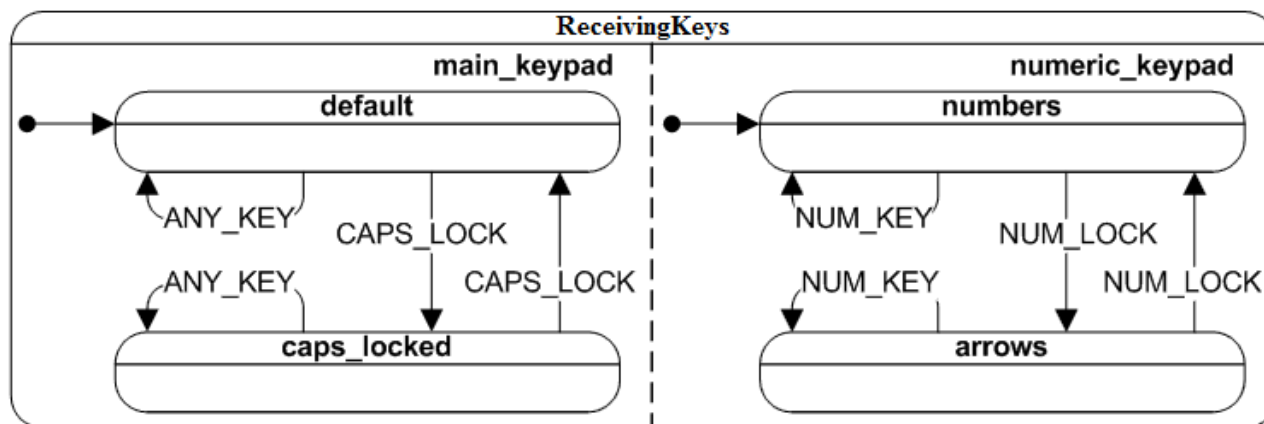
EXEMPLE DU CLAVIER (1)

Exemple du clavier sans régions orthogonales



EXEMPLE DU CLAVIER (2)

Exemple du clavier avec régions orthogonales



PORTE DU GARAGE – ÉTUDE DE CAS

Contextes

- La compagnie DOORS inc. fabrique des composantes de portes de garage
- Cependant, ils ont des problèmes avec le logiciel intégré (embedded) qui contrôle le moteur (motor unit) de l'ouvre-porte automatique du garage
 - Ils ont développé ce logiciel eux-mêmes
 - Ceci leur cause la perte de clients
- Ils ont décidé de jeter le logiciel existant et embaucher une compagnie de logiciel professionnelle pour livrer un logiciel sans problèmes (bugs)

EXIGENCES DU CLIENT

Exigences (informelles) du client:

- **Exigence 1:** Quand la porte du garage est fermée, elle doit s'ouvrir lorsque l'utilisateur pèse le bouton sur le système de contrôle mural ou sur la télécommande
- **Exigence 2:** Quand la porte du garage est ouverte, elle doit se fermer lorsque l'utilisateur pèse le bouton sur le système de contrôle mural ou sur la télécommande
- **Exigence 3:** La porte du garage ne doit pas se fermer sur un obstacle
- **Exigence 4:** Il doit y avoir un moyen pour avoir la porte du garage semi-ouverte
- **Exigence 5:** Le système doit accomplir un test auto-diagnostic avant d'effectuer une commande (ouvrir ou fermer) afin de s'assurer que toutes ses composantes sont fonctionnelles



uOttawa

L'Université canadienne
Canada's university

EXIGENCES DU CLIENT

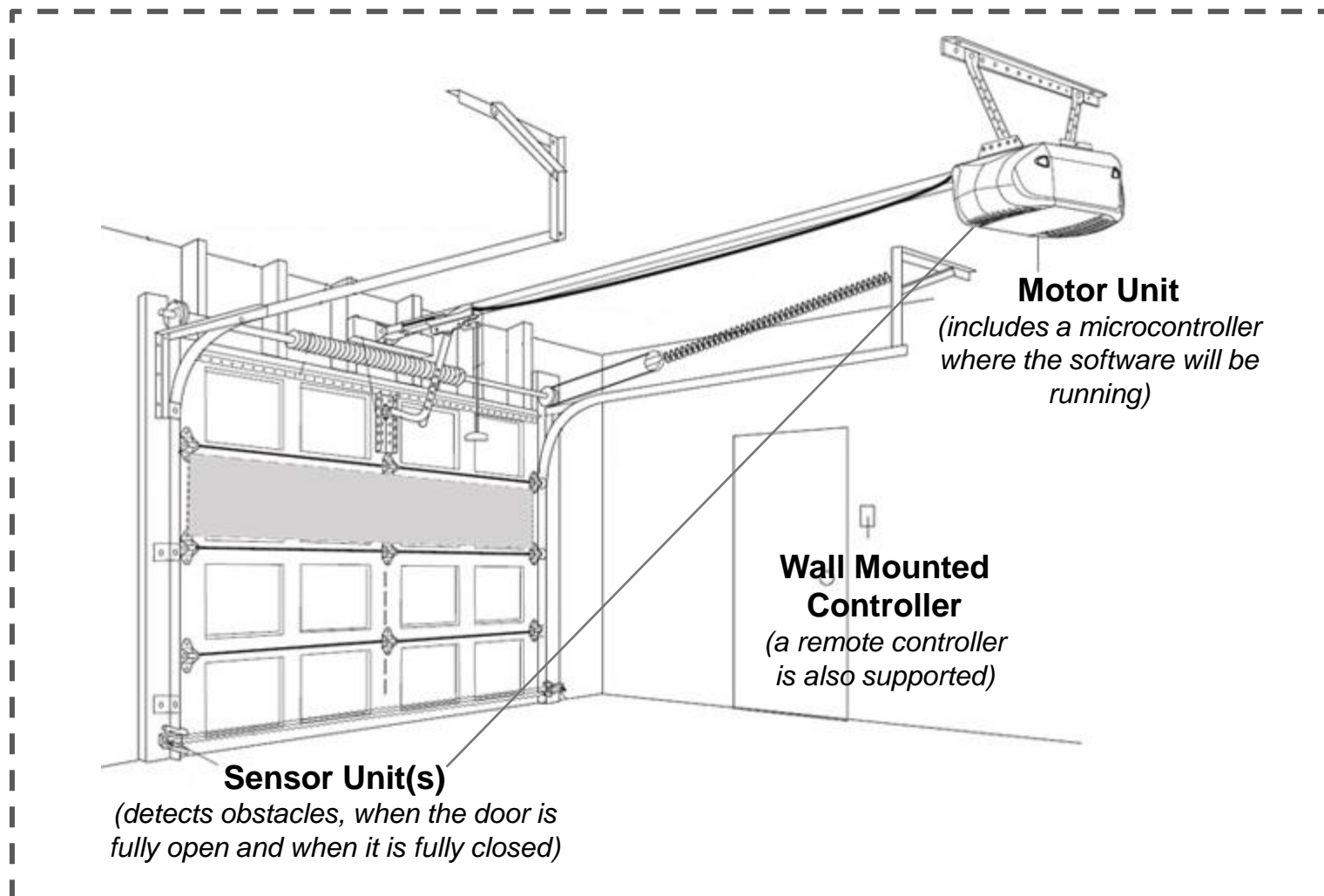
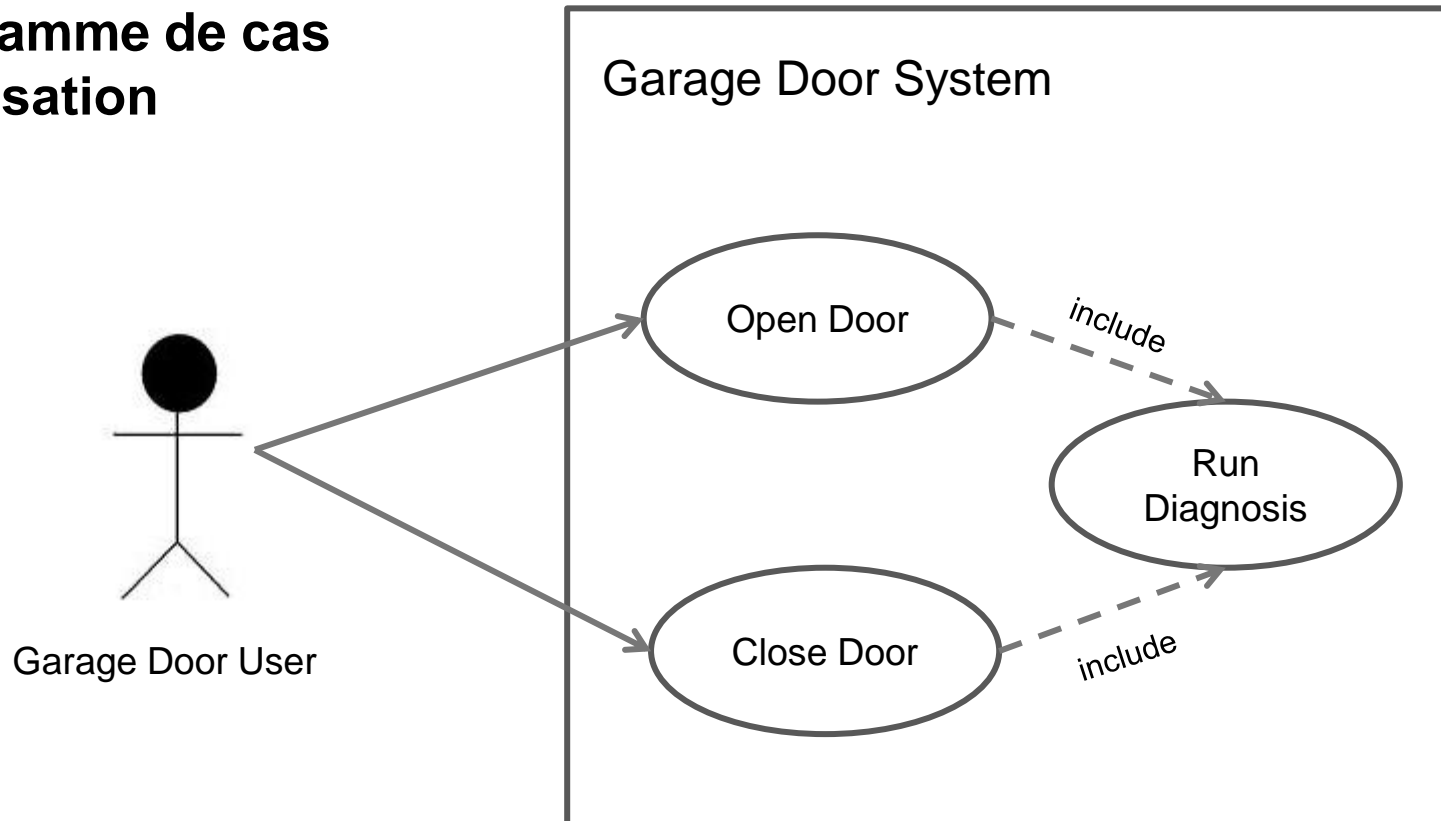


DIAGRAMME DE CAS D'UTILISATION

Diagramme de cas d'utilisation



CAS D'UTILISATION: EFFECTUER UN DIAGNOSTIC

Nom du cas d'utilisation: Effectuer un Diagnostic

Sommaire: Le système effectue un test autodiagnostic

Acteur: L'utilisateur de la porte de garage

Précondition: L'utilisateur a pressé sur la télécommande ou sur le bouton du système de contrôle mural

Séquence:

1. Vérifier si le détecteur fonctionne correctement
2. Vérifier si l'unité moteur fonctionne correctement
3. Si toutes les composantes sont vérifiées avec succès, le système autorise l'exécution de la commande

Séquence alternative:

Étape 3: L'un des contrôles échoue et le système attend 3 minutes et déclenche une vérification de nouveau

Post-condition: L'autodiagnostic détermine que le système est opérationnel

CAS D'UTILISATION: OUVRIR LA PORTE

Nom du cas d'utilisation: Ouvrir la porte

Sommaire: Ouvrir la porte du garage

Acteur: L'utilisateur de la porte du garage

Dépendance: Inclure le cas d'utilisation d'effectuer un diagnostic

Précondition: Le système de la porte du garage est opérationnel et prêt à accepter une commande

Séquence:

1. L'utilisateur pèse la télécommande ou le bouton du système de contrôle mural
2. Inclure le cas d'utilisation d'effectuer un diagnostic
3. Si la porte est actuellement en train de se fermer ou elle est déjà fermée, le système ouvre la porte

Séquence alternative:

Étape 3: Si la porte est ouverte, le système ferme la porte

Étape 3: Si la porte est en train de s'ouvrir, le système arrête la porte (la gardant semi-ouverte)

Poste-condition: La porte du garage est ouverte

CAS D'UTILISATION: FERMER LA PORTE

Nom du cas d'utilisation: Fermer la porte

Sommaire: Fermer la porte du garage

Acteur: L'utilisateur de la porte du garage

Dépendance: Inclure le cas d'utilisation d'effectuer un diagnostic

Précondition: Le système de la porte du garage est opérationnel et prêt à accepter une commande

Séquence:

1. L'utilisateur pèse la télécommande ou le bouton du système de contrôle mural
2. Inclure le cas d'utilisation d'effectuer un diagnostic
3. Si la porte est actuellement en train de s'ouvrir ou elle est déjà ouverte, le système ferme la porte

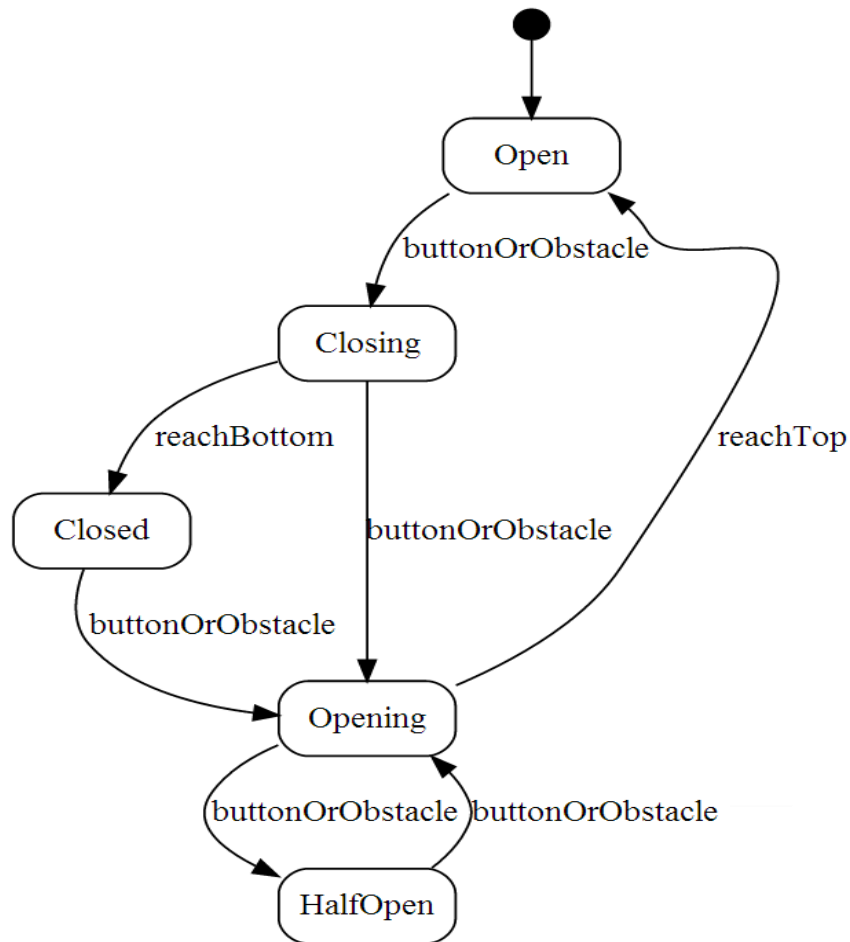
Séquence alternative:

Étape 3: Si la porte est en train de se fermer ou elle est déjà fermée, le système ouvre la porte

Étape 3: Si la porte est en train de s'ouvrir, le système arrête la porte (la gardant semi-ouverte)

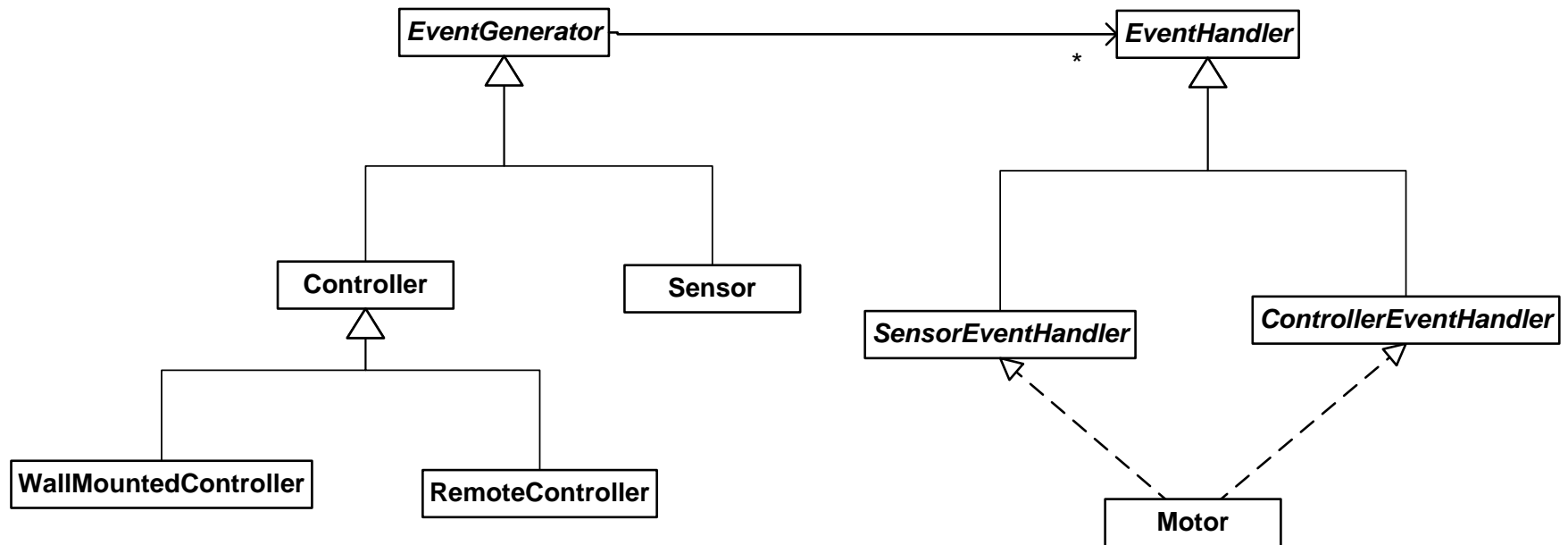
Poste-condition: La porte du garage est fermée

MODÉLISATION COMPORTEMENTALE DE HAUT NIVEAU

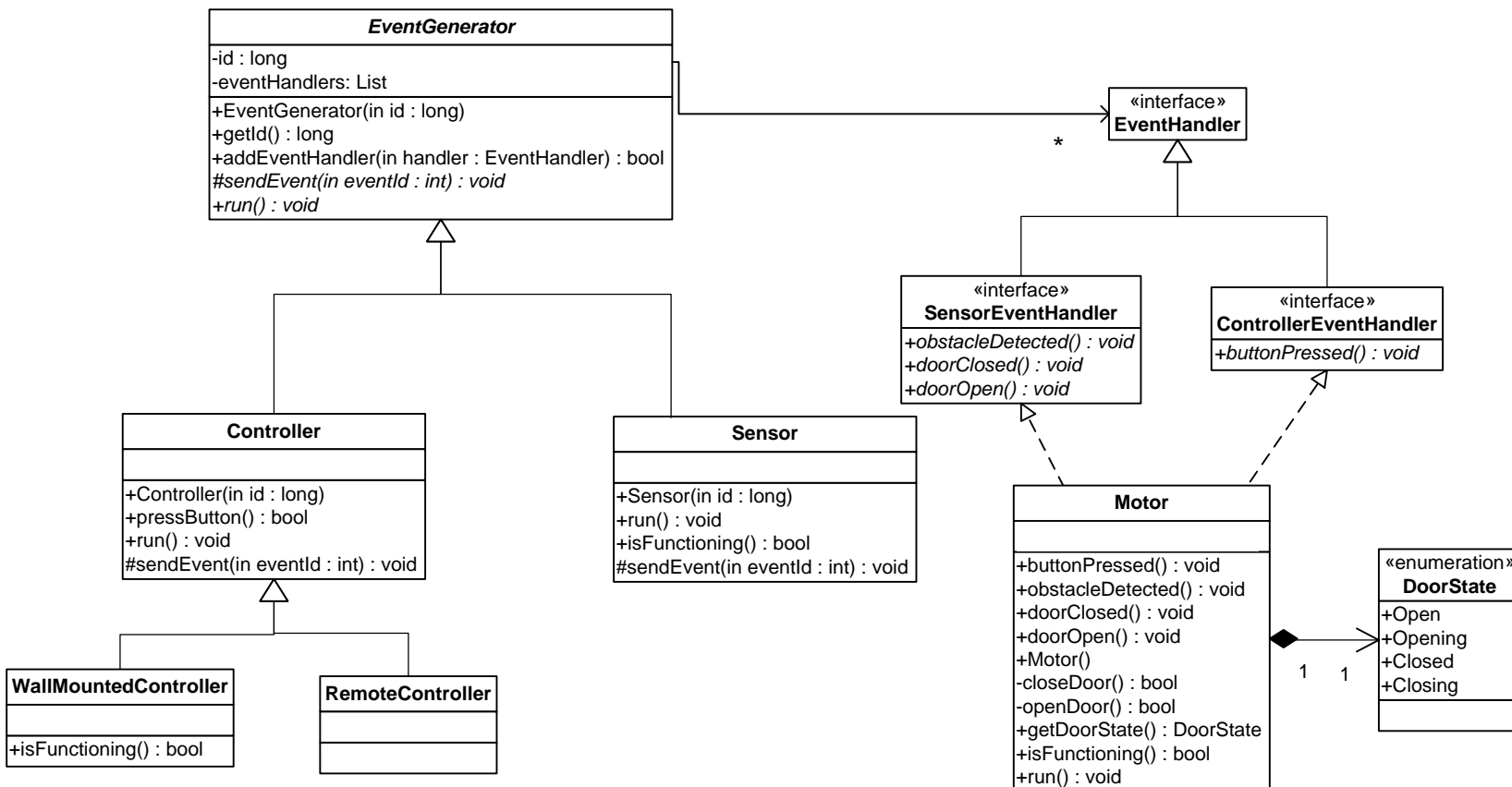


Cliquer ici

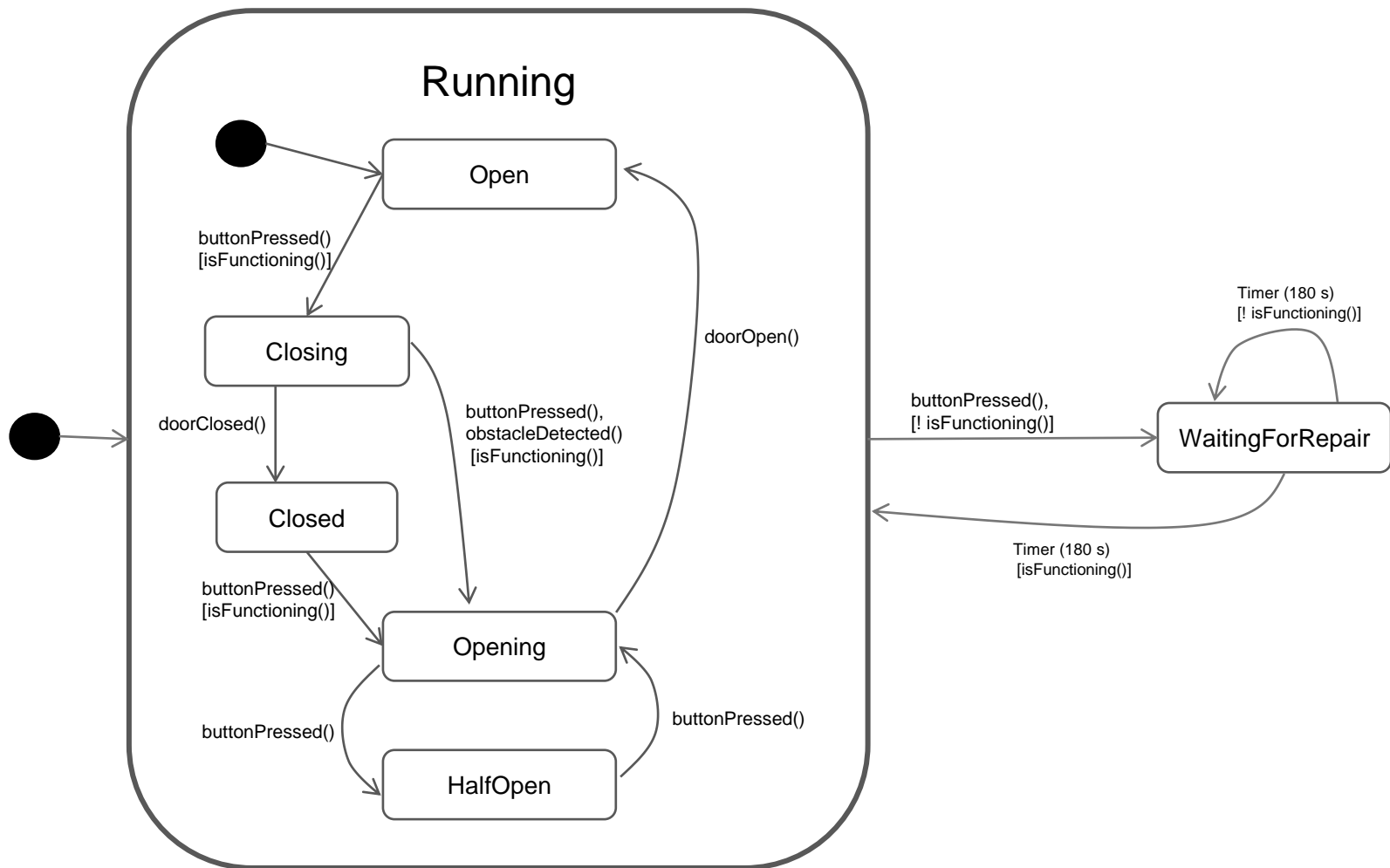
MODÈLE STRUCTURAL DE HAUT NIVEAU



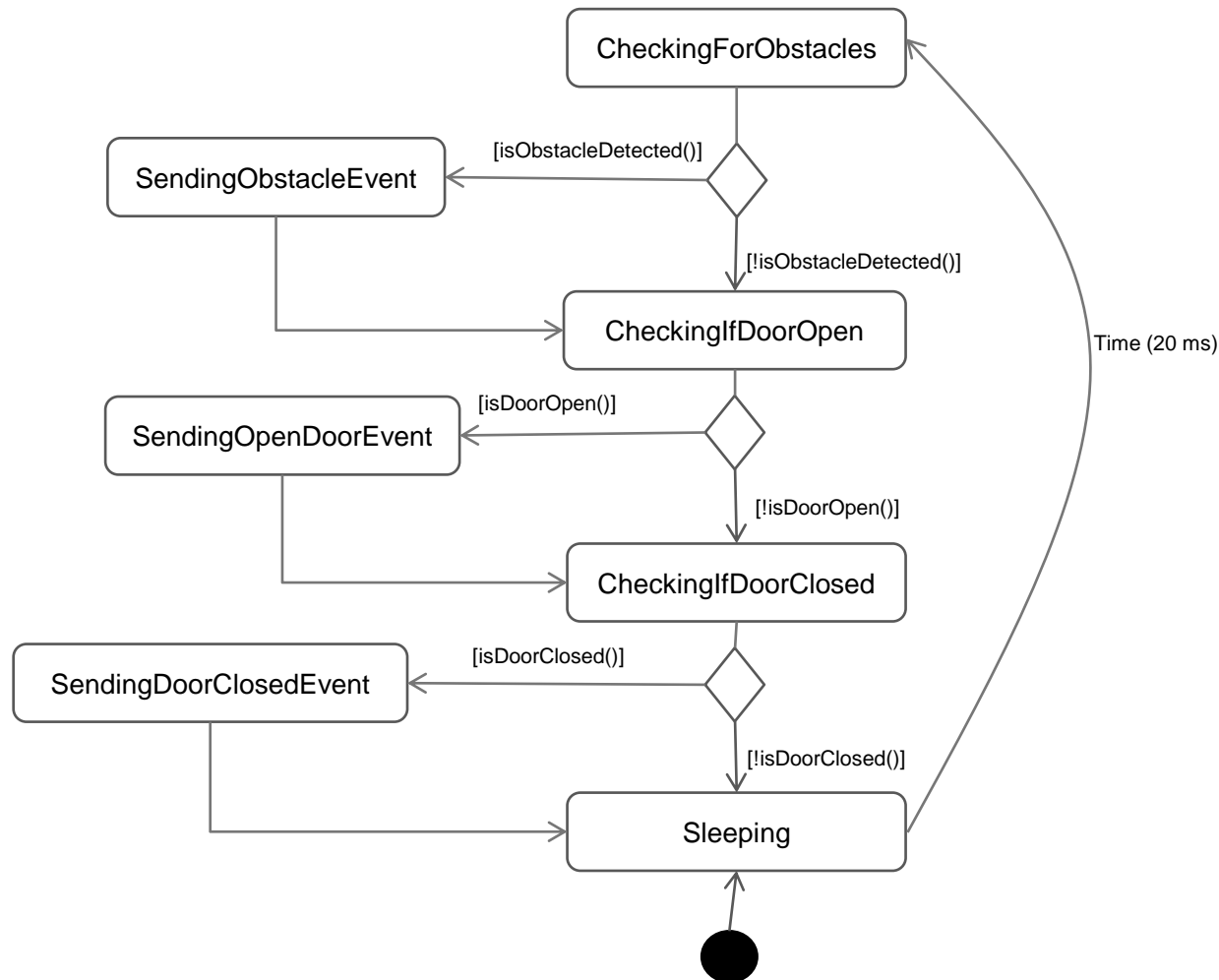
MODÈLE STRUCTURAL RAFFINÉ



MODÈLE COMPORTEMENTAL RAFFINÉ – UNITÉ MOTEUR



MODÈLE COMPORTEMENTAL RAFFINÉ – UNITÉ DÉTECTEUR



PROGRAMMATION

Lorsque nous sommes satisfaits du niveau de détail dans notre modèle comportemental, nous sommes prêts à programmer

Quelques parties du code peuvent être générées directement par des outils du modèle comportemental

- Quelques modifications peuvent être nécessaires (bien que généralement découragés par les fournisseurs d'outils)
 - Les Humains sont toujours les programmeurs les plus intelligents

CLASSE EVENT GENERATOR

```
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();
}
```



CLASSE SENSOR

```
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();
                }
            }
        }
    }
}
```



CLASSE SENSOR

```
@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

DÉMO UMPLE

UMPLE est un outil de modélisation pour permettre ce qu'on appelle la programmation modèle-orientée

- C'est ce qu'on fait dans ce cours

Vous pouvez l'utiliser pour créer des diagrammes de classe (modèles structuraux) et des machines d'état (modèles comportementaux)

Cet outil a été développé à l'Université d'Ottawa

- Une version en ligne existe sur ce lien:
<http://cruise.eecs.uottawa.ca/umpleonline/>
- Il existe aussi un eclipse plugin pour cet outil

CODE UML POUR MACHINE D'ÉTAT À UNITÉ MOTEUR

```
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;}  
        }  
    }  
}
```

FRAGMENTS DE CODE DE LA CLASSE MOTEUR



uOttawa

L'Université canadienne
Canada's university

```
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

MERCI!

QUESTIONS?