Artificial Intelligence

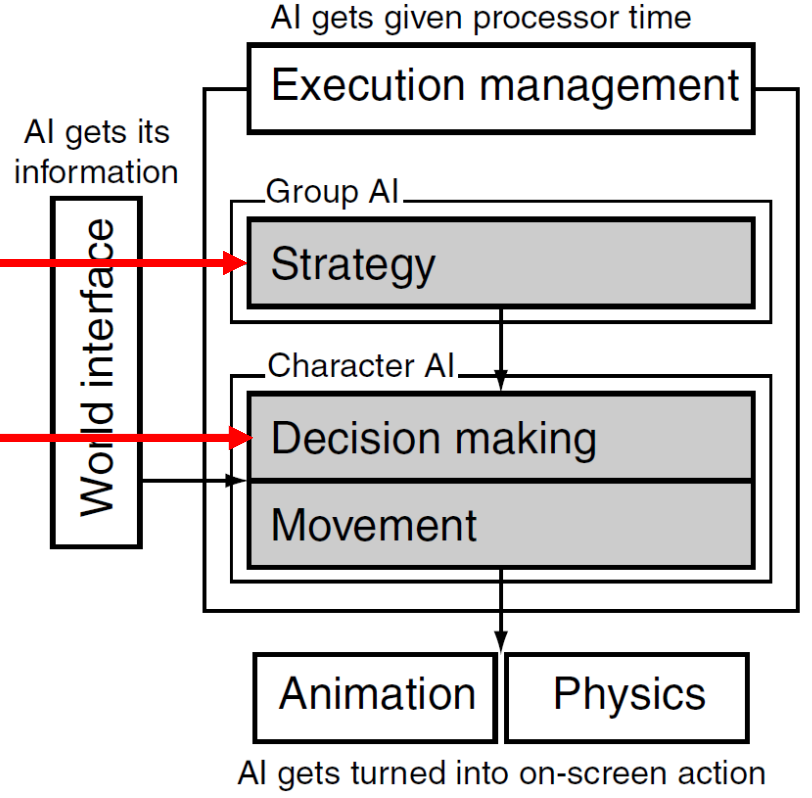
Lecture 03 – Finite State Machines

Edirlei Soares de Lima

<edirlei.lima@universidadeeuropeia.pt>

Game AI – Model

- Pathfinding
- Steering behaviours
- **Finite state machines**
- Automated planning
- Behaviour trees
- Randomness
- Sensor systems
- Machine learning



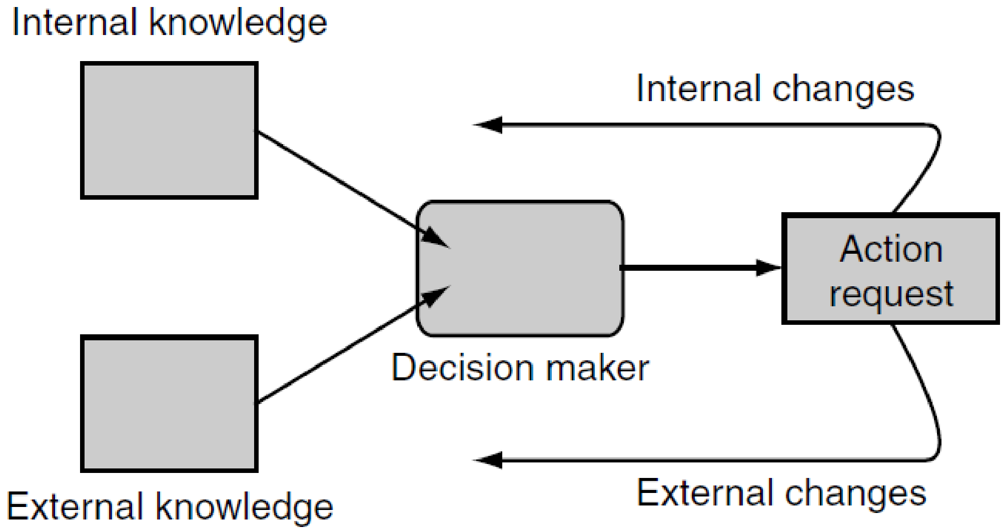
Decision Making

- In game AI, decision making is the ability of a character/agent to decide what to do.

- The agent processes a set of information that it uses to generate an action that it wants to carry out.
  - **Input:** agent’s knowledge about the world;
  - **Output:** an action request;

Decision Making

- The knowledge can be broken down into external and internal knowledge.
  - **External knowledge:** information about the game environment (e.g. characters' positions, level layout, noise direction).
  - **Internal knowledge:** information about the character's internal state (e.g. health, goals, last actions).



Finite State Machines

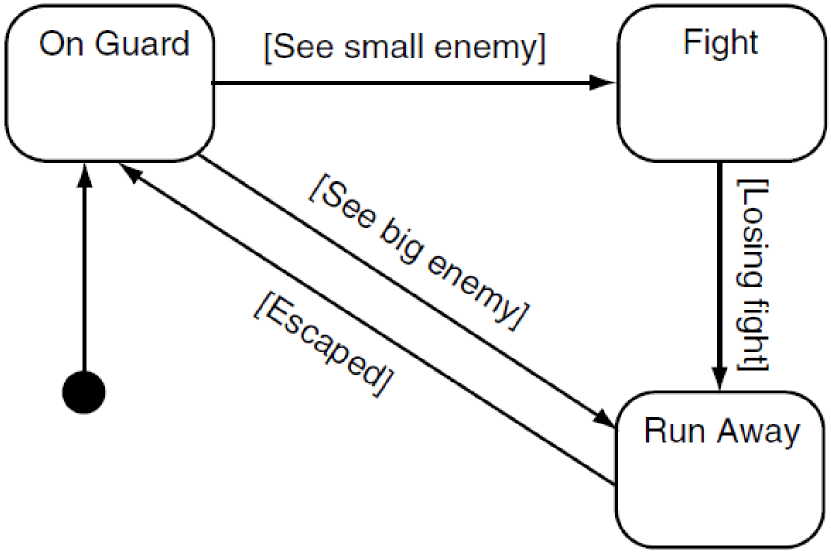
- Usually, game characters have a limited set of possible behaviors. They carry on doing the same thing until some event or influence makes them change.
  - Example: a guard will stand at its post until it notices the player, then it will switch into attack mode, taking cover and firing.

- State machines are the technique most often used for this kind of decision making process in games.

- What is a state machine?

Finite State Machines

- Actions or behaviors are associated with each state.
- Each transition leads from one state to another, and each has a set of associated conditions.
- When the conditions of a transition are met, then the character changes state to the transition's target state.
- Each character is controlled by one state machine and they have a current state.

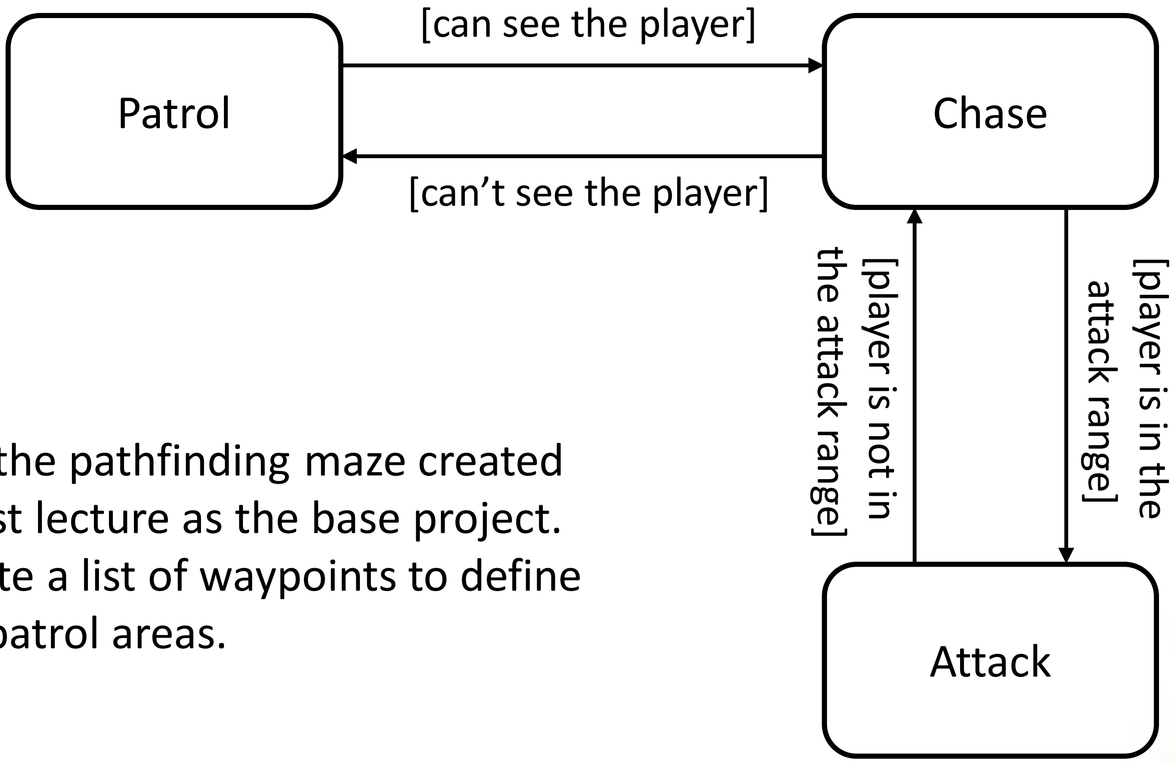


Hard-Coded Finite State Machines

```
enum State {PATROL, DEFEND, SLEEP};
State myState;

function update(){
  if (myState == PATROL){
    if (canSeePlayer())
      myState = DEFEND;
    if (tired())
      myState = SLEEP;
  }
  elseif (myState == DEFEND){
    if not canSeePlayer()
      myState = PATROL;
  }
  elseif (myState == SLEEP){
    if (not tired())
      myState = PATROL;
  }
}
```

Exercise 1

1) Implement a hard-coded finite state machine to control an NPC based on the following diagram:



Hints:

- Use the pathfinding maze created in last lecture as the base project.
- Create a list of waypoints to define the patrol areas.

Hard-Coded Finite State Machines

- Although hard-coded state machines are <u>easy to write</u> and are <u>very fast</u>, they are notoriously <u>difficult to maintain</u>.

- Complex finite states machines require thousands of lines of code.

- Another weaknesses:
  - Programmers are responsible for writing the AI behaviors of each character.
  - The game has to be recompiled each time the behavior changes.

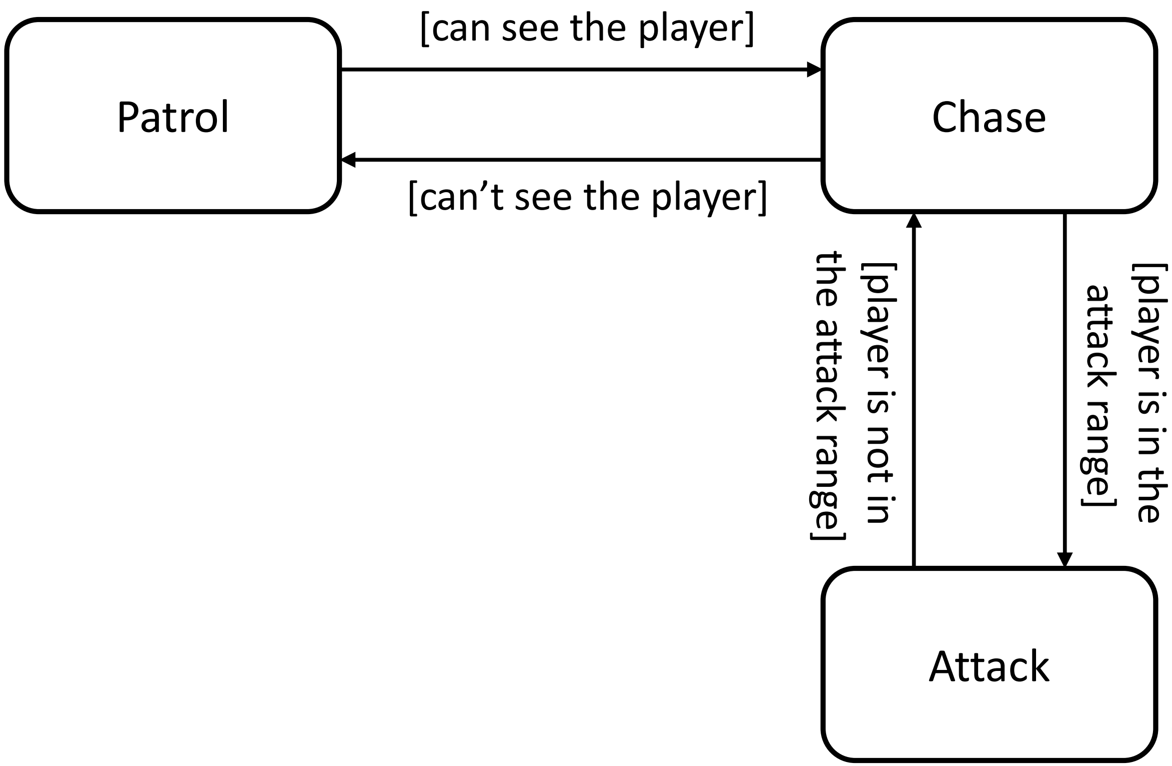
Class-Based Finite State Machines

```
class StateMachine{

  private List<State> states;
  private State initialState;
  private State currentState = initialState;

  List<Action> update(){

    triggeredTransition = Transition.None;
    for each Transition t in currentState.getTransitions(){
      if (t.isTriggered()){
        triggeredTransition = t;
        break;
      }
    }

    ...
```

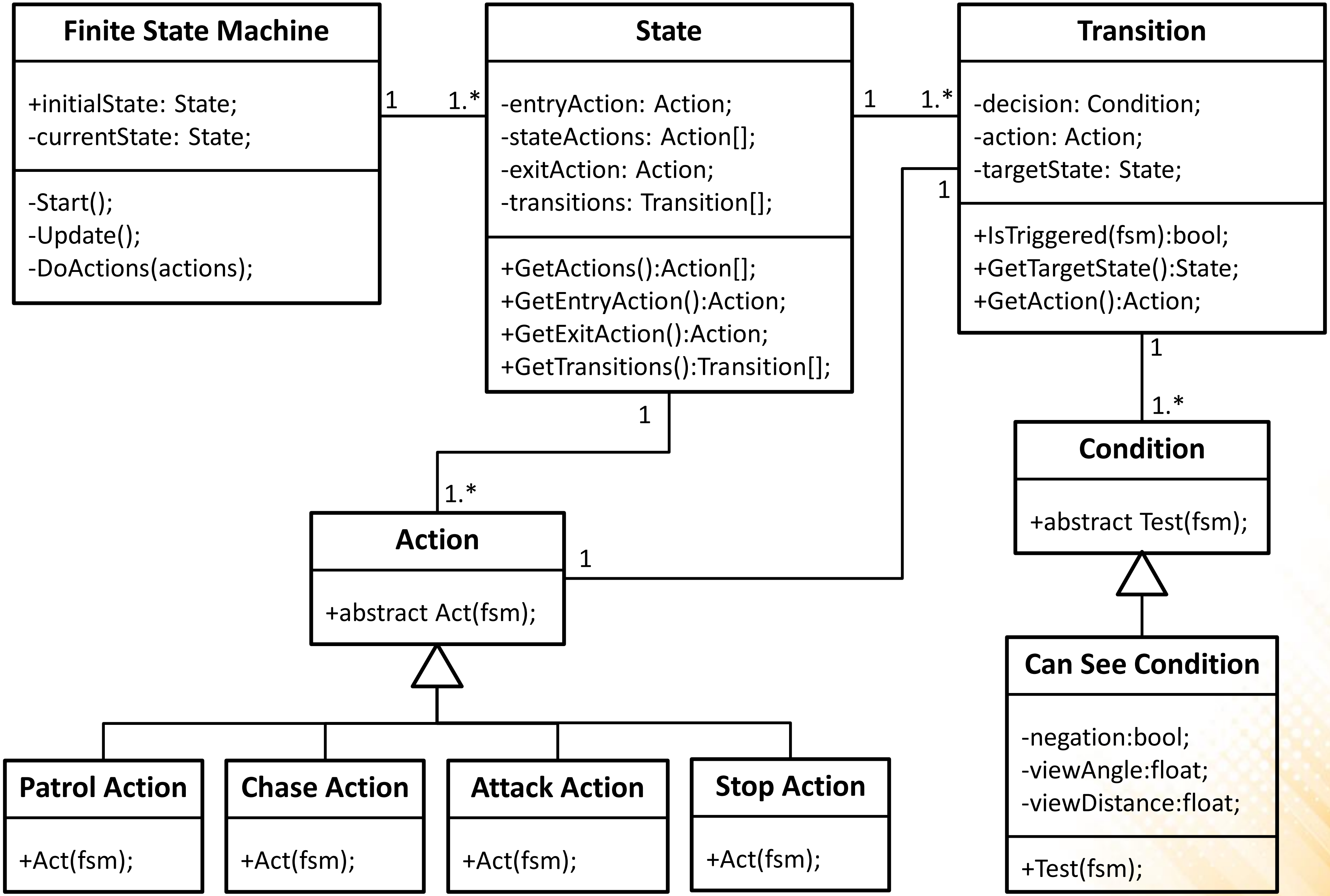
Class-Based Finite State Machines

```
    ...

    if (triggeredTransition)
    {
      targetState = triggeredTransition.getTargetState();
      List<Action> actions = new List<Action>();
      actions.Add(currentState.getExitAction());
      actions.Add(triggeredTransition.getAction());
      actions.Add(targetState.getEntryAction());
      currentState = targetState;
      return actions;
    }
  }
  else
  {
    return currentState.getAction();
  }
}
```

Unity – Implementation



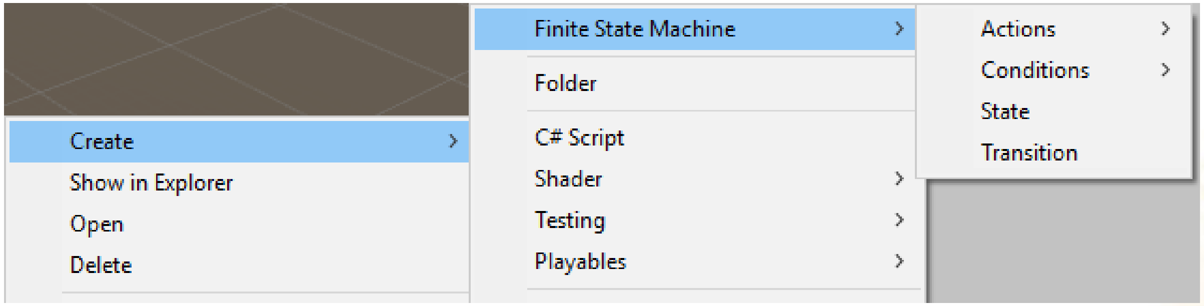
Class Diagram



Unity – ScriptableObject

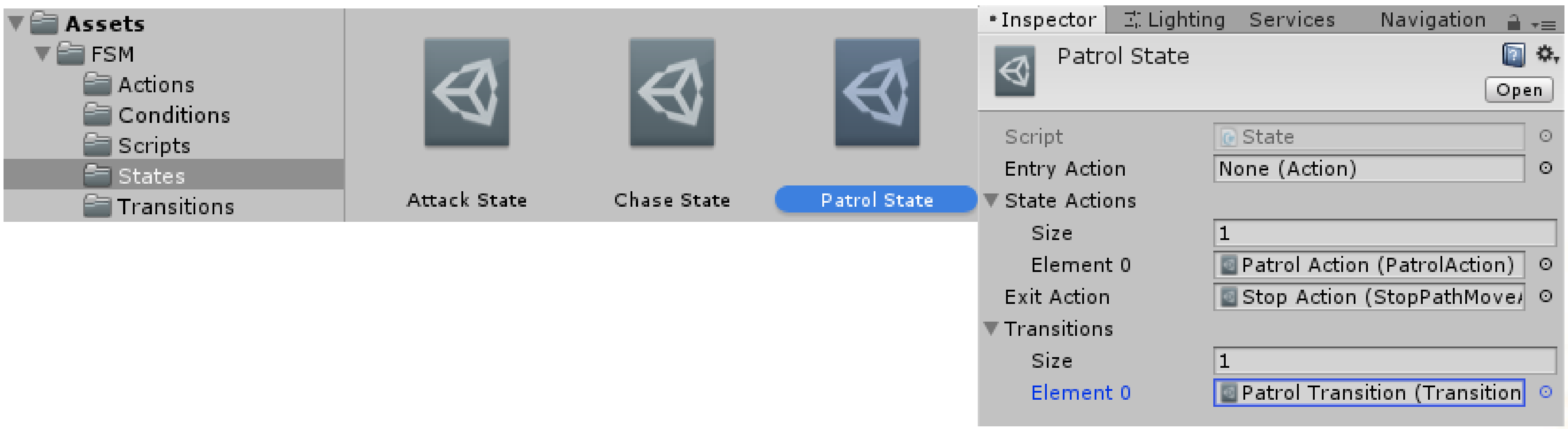
- In Unity, a ScriptableObject is a class that allows you to store data and execute code independent from script instances. They can also be used to create pluggable data sets.
  - They work like the MonoBehaviour class, but they don't need to be attached to GameObjects.

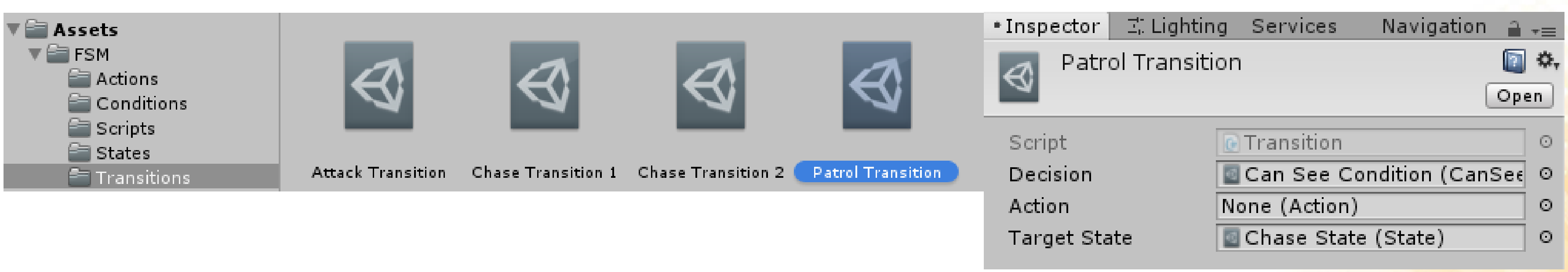
- Once a ScriptableObject-derived class have been defined, is possible to use the CreateAssetMenu attribute to make it easy to create custom assets of the class.



Unity – ScriptableObject

- ScriptableObjects allow us to create a <u>pluggable finite state machine system</u>.





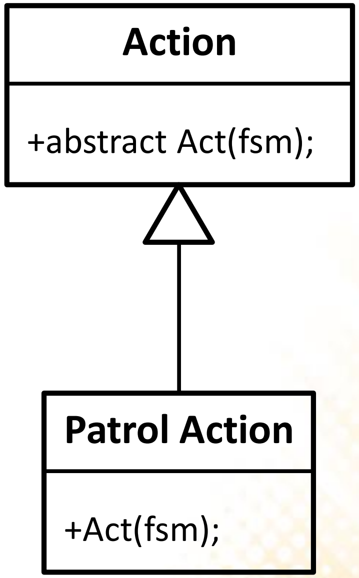
Action Classes

- **Action Class:**

```
public abstract class Action : ScriptableObject
{
  public abstract void Act(FiniteStateMachine fsm);
}
```

- **Patrol Action Class:**

```
[CreateAssetMenu(menuName = "Finite State Machine
                             /Actions/Patrol")]
public class PatrolAction : Action
{
  public override void Act(FiniteStateMachine fsm)
  {
    if (fsm.GetNavMeshAgent().IsAtDestionation())
      fsm.GetNavMeshAgent().GoToNextWaypoint();
  }
}
```



Action Classes

- **Chase Action Class:**

```
[CreateAssetMenu(menuName = "Finite State Machine/Actions/Chase")]
public class ChaseAction : Action
{
  public override void Act(FiniteStateMachine fsm)
  {
    if (fsm.GetNavMeshAgent().IsAtDestionation())
      fsm.GetNavMeshAgent().GoToTarget();
  }
}
```

- **Stop Action Class:**

```
[CreateAssetMenu(menuName = "Finite State Machine/Actions/Stop")]
public class StopAction : Action{
    public override void Act(FiniteStateMachine fsm)
    {
        fsm.GetNavMeshAgent().StopAgent();
    }
}
```

Action Classes

- **Attack Action Class:**

```
[CreateAssetMenu(menuName = "Finite State Machine/Actions/Attack")]
public class AttackAction : Action {
  public GameObject shootPrefab;
  public float shootTimeInverval = 2;
  private float shootTime = float.PositiveInfinity;

  public override void Act(FiniteStateMachine fsm)
  {
    shootTime += Time.deltaTime;
    if (shootTime > shootTimeInverval){
      shootTime = 0;
      GameObject bullet = Instantiate(shootPrefab,
                    fsm.transform.position, fsm.transform.rotation);
      bullet.GetComponent<Rigidbody>().velocity =
            fsm.transform.TransformDirection(Vector3.forward * 10);
    }
  }
}
```

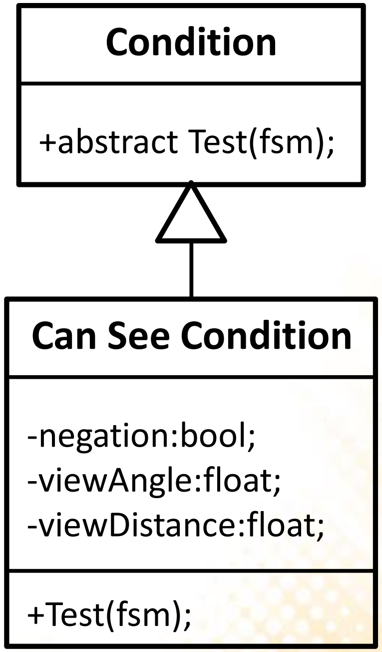
Action Classes

- **Condition Class:**

```
public abstract class Condition : ScriptableObject
{
  public abstract bool Test(FiniteStateMachine fsm);
}
```

- **Can See Condition Class:**

```
[CreateAssetMenu(menuName = "Finite State Machine
                              /Conditions/Can See")]
public class CanSeeCondition : Condition {
   [SerializeField]
  private bool negation;
   [SerializeField]
  private float viewAngle;
   [SerializeField]
  private float viewDistance;
   ...
```



Condition Classes

```
  ...
  public override bool Test(FiniteStateMachine fsm){
    Transform target = fsm.GetNavMeshAgent().target;
    Vector3 targetDir = target.position - fsm.transform.position;
    float angle = Vector3.Angle(targetDir, fsm.transform.forward);
    float dist = Vector3.Distance(target.position,
                                  fsm.transform.position);
    if ((angle < viewAngle) && (dist < viewDistance)){
      if (negation)
        return false;
      else
        return true;
    }else{
      if (negation)
        return true;
      else
        return false;
     }
  }
}
```

Transition Class

```
[CreateAssetMenu(menuName = "Finite State Machine
                              /Transition")]
public class Transition : ScriptableObject{
  [SerializeField]
  private Condition decision;
  [SerializeField]
  private Action action;
  [SerializeField]
  private State targetState;
  public bool IsTriggered(FiniteStateMachine fsm){
    return decision.Test(fsm);
  }
  public State GetTargetState(){
    return targetState;
  }
  public Action GetAction(){
    return action;
  }
}
```

Transition
-decision: Condition; -action: Action; -targetState: State;
+IsTriggered(fsm):bool; +GetTargetState():State; +GetAction():Action;

State Class

```
[CreateAssetMenu(menuName = "Finite State Machine/State")]
public class State : ScriptableObject{
   [SerializeField]
  private Action entryAction;
   [SerializeField]
  private Action[] stateActions;
   [SerializeField]
  private Action exitAction;
   [SerializeField]
  private Transition[] transitions;
  public Action[] GetActions(){
    return stateActions;
  }
  public Action GetEntryAction(){
    return entryAction;
  }
  public Action GetExitAction(){
    return exitAction;
  }
  public Transition[] GetTransitions(){
    return transitions;
  }
}
```

State
-entryAction: Action; -stateActions: Action[]; -exitAction: Action; -transitions: Transition[];
+GetActions():Action[]; +GetEntryAction():Action; +GetExitAction():Action; +GetTransitions():Transition[];

Finite State Machine Class

```
public class FiniteStateMachine : MonoBehaviour {
  public State initialState;
  private State currentState;
  private MyNavMeshAgent navMeshAgent;

  void Start(){
    currentState = initialState;
    navMeshAgent = GetComponent<MyNavMeshAgent>();
  }

  void Update(){
    Transition triggeredTransition = null;
    foreach (Transition t in currentState.GetTransitions()){
      if (t.IsTriggered(this)){
        triggeredTransition = t;
        break;
      }
    }
    ...
```

Finite State Machine
+initialState: State; -currentState: State;
-Start(); -Update(); -DoActions(actions);

Finite State Machine Class

```
    List<Action> actions = new List<Action>();
    if (triggeredTransition){
      State targetState = triggeredTransition.GetTargetState();
      actions.Add(currentState.GetExitAction());
      actions.Add(triggeredTransition.GetAction());
      actions.Add(targetState.GetEntryAction());
      currentState = targetState;
    }
    else{
      foreach (Action a in currentState.GetActions())
        actions.Add(a);
    }
    DoActions(actions);
  }
  void DoActions(List<Action> actions){
    foreach (Action a in actions){
      if (a != null)
        a.Act(this);
    }
  }
}
```

Nav Mesh Agent

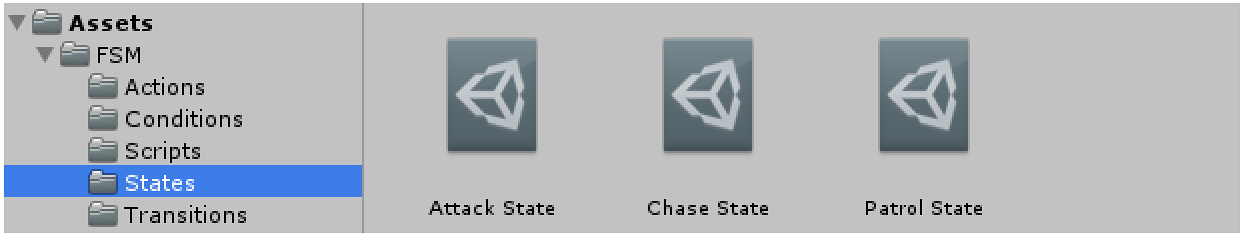
```
public class MyNavMeshAgent : MonoBehaviour {
  public Transform target;
  public Transform[] waypoints;
  private int currentWaypoint;
  private NavMeshAgent agent;

  void Start(){
    currentWaypoint = 0;
    agent = GetComponent<NavMeshAgent>();
  }

  public void GoToNextWaypoint(){
    agent.destination = waypoints[currentWaypoint].position;
    currentWaypoint++;
    if (currentWaypoint >= waypoints.Length)
      currentWaypoint = 0;
  }

  ...
```

Nav Mesh Agent

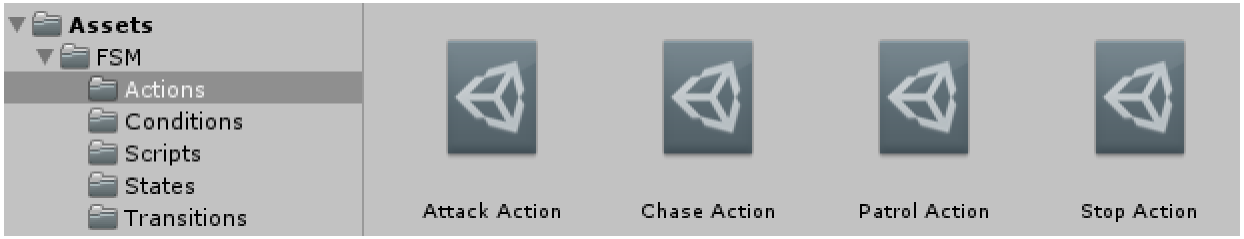
```
  public void GoToTarget(){
    agent.destination = target.position;
  }

  public void StopAgent(){
    agent.isStopped = true;
    agent.ResetPath();
  }

  public bool IsAtDestionation(){
    if (!agent.pathPending){
      if (agent.remainingDistance <= agent.stoppingDistance){
        if (!agent.hasPath || agent.velocity.sqrMagnitude == 0f){
          return true;
        }
      }
    }
    return false;
  }
}
```

Finite State Machine – Objects

- States:



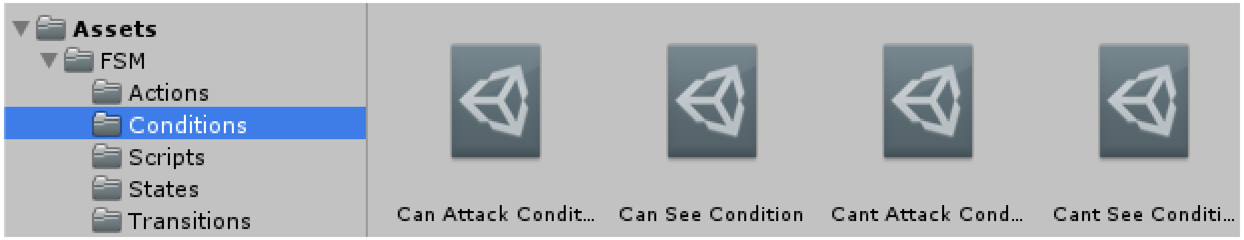
- Actions:



Finite State Machine – Objects

- Transitions:

- Conditions:



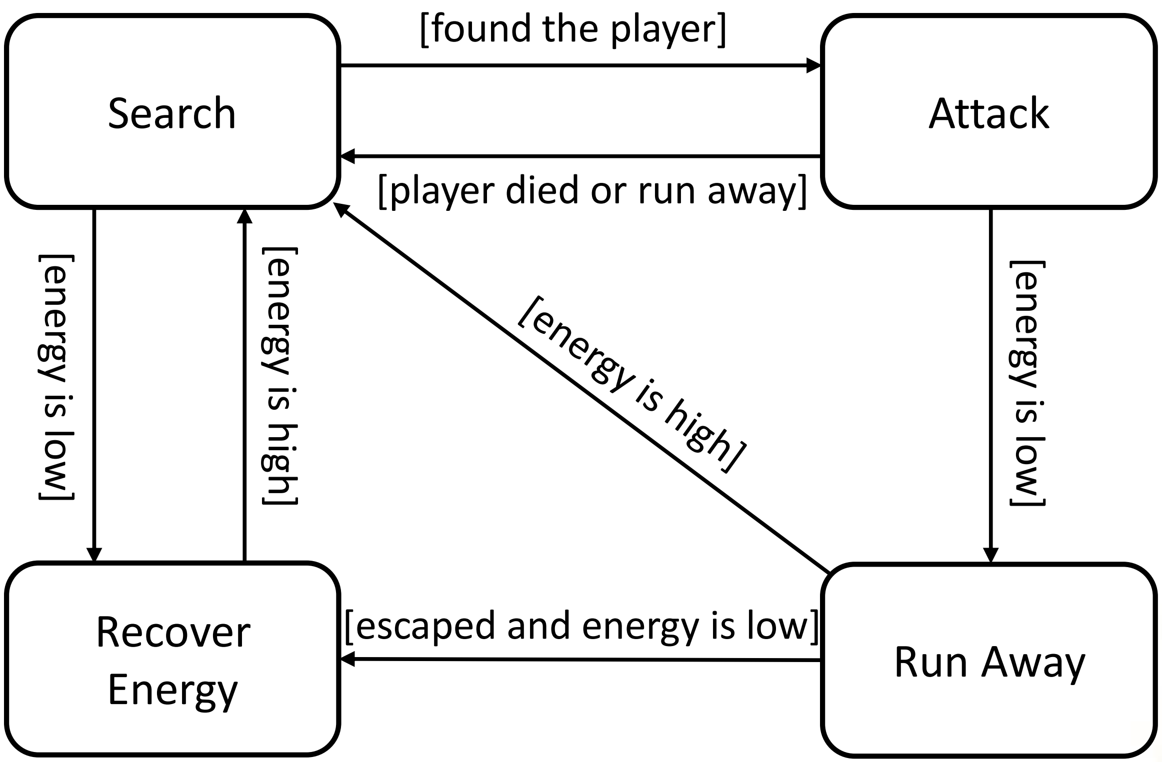
Class-Based Finite State Machines

- The class-based approach gives a lot of flexibility to the Finite States Machines, but reduces its performance due to the large number of method calls.

- Another alternative: Script-Based Finite States Machines
  - Scripting languages: Lua, Pawn, GameMonkey, ...
  - Allows designers to create the state machine rules but can be slightly more efficient.
  - However, interpreting a script is at least as time consuming as executing a large number of method calls.

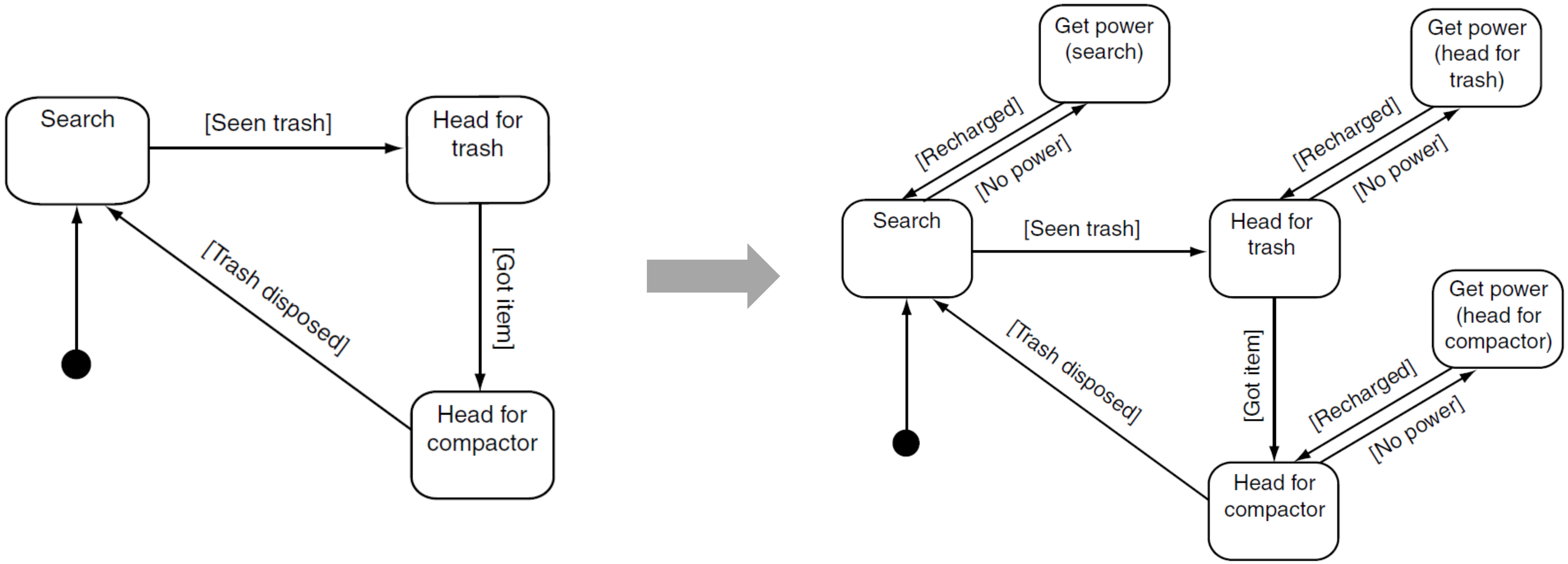
Exercise 2

2) Implement the AI of an NPC using the following finite state machine and the pluggable FSM system:



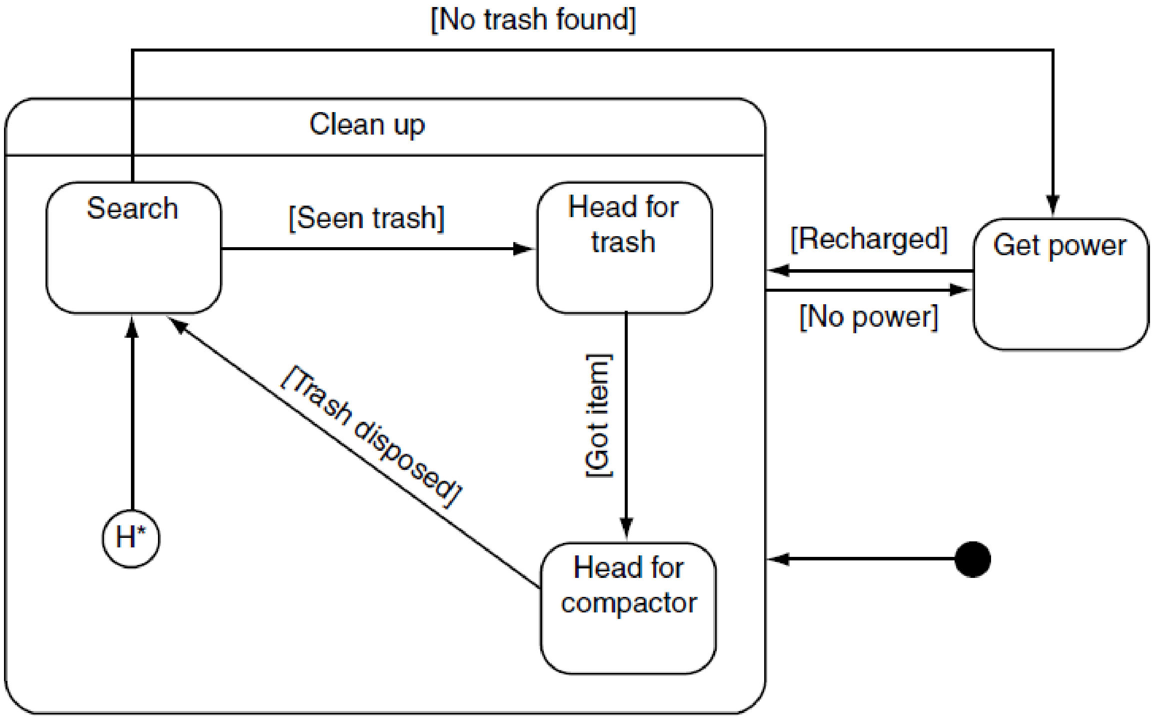
Finite State Machines

- On its own, a state machine is a powerful tool, but as the complexity of agent behavior increases, the state machine can grow uncontrollably.
  - Even the visual representation becomes complex.
  - It can also be difficult to express composed behaviors (e.g. a recharge behavior that can occur at any state).



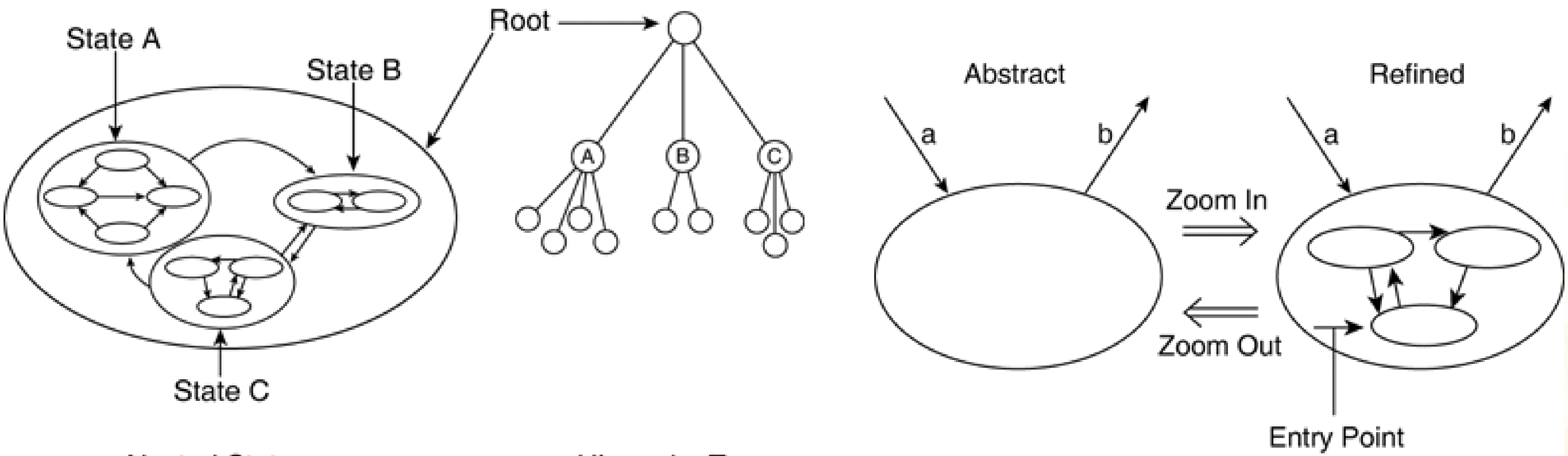
Hierarchical State Machines

- Solution to reduce the complexity of the finite state machines: Hierarchical State Machines
  - Rather than combining all the logic into a single state machine, we can separate it into several state machines arranged in a hierarchy.



Hierarchical State Machines

- While high level states represent abstract actions, low level states represent concrete actions.

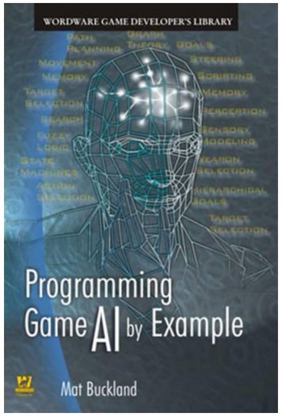


Nested States

Hierarchy Tree

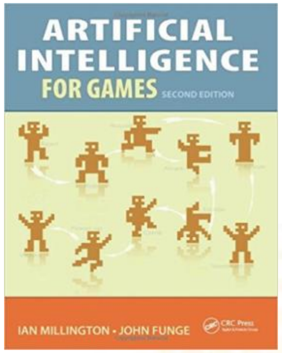
Further Reading

- Buckland, M. (2004). **Programming Game AI by Example**. Jones & Bartlett Learning. ISBN: 978-1-55622-078-4.



  – **Chapter 2: State-Driven Agent Design**

- Millington, I., Funge, J. (2009). **Artificial Intelligence for Games** (2nd ed.). CRC Press. ISBN: 978-0123747310.



  – **Chapter 5.3: State Machines**

- Web:
  - https://unity3d.com/pt/learn/tutorials/topics/navigation/finite-state-ai-delegate-pattern