# Decision Making: Finite State Machines

Matthew Guzdial

guzdial@ualberta.ca

UNIVERSITY OF ALBERTA

## Announcements

- Quiz 1 nearly completely graded (Tomorrow at latest)
- Quiz 2 on Friday!
  - APSP, Trees, MCTS, Rules Systems, and FSMs (Today's lecture) all fair game. Basic concepts from Wednesday's lecture as well.
  - Worth remembering stuff about graphs, A*, and other things that have come up in those lectures.
- HW2 due tonight, HW3 released

## Last Class

- A nightmare of technical issues
- Rule systems
- Benefits to rule systems/issues with rule systems

## Most Actions Require More Than One Rule

- Shoot
- Patrol
- Farm
- Sleep
- Hide
- Hunt
- …

We will call these “states”

# Most game agents go through more than one state

Game agents will have different sets of states, and different ways of moving through those states based on their purpose in the game:

- Guard: Patrol -> Attack -> Hunt
- Farmer/Villager: Farm -> Patrol -> Sleep

## Decision Making

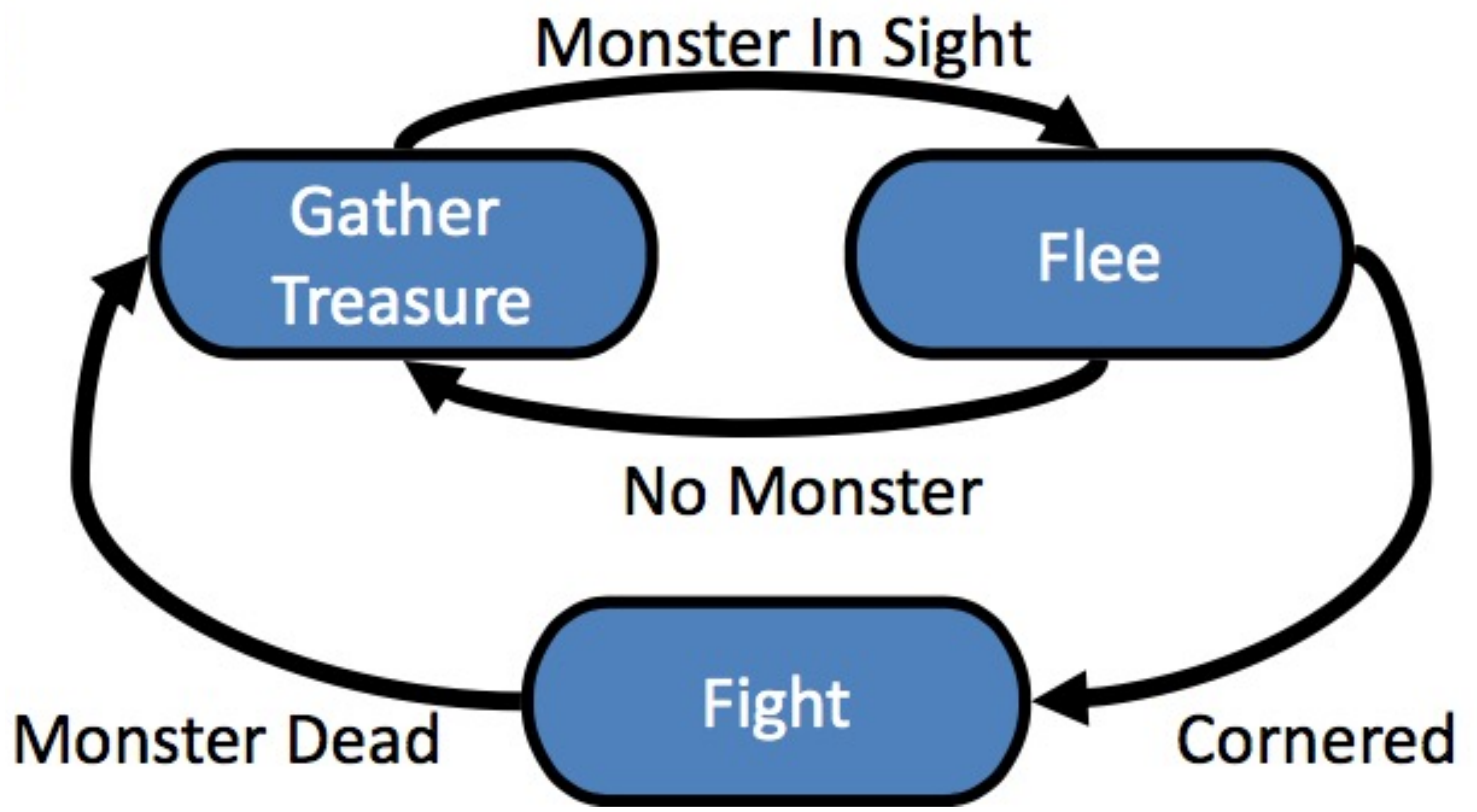
- Standard AI
  - Making the *optimal* choice of action that *maximizes* the chance of achieving a goal
- Game AI
  - Choosing the right state to support the experience

- State: Combination of animation and mechanical effect in the game engine
  - Example: “Attack” playing an animation where the agent swings at the player, moves towards the player, checks to see if the player is hit, and does damage if so: https://youtu.be/9-SOZtjf4T8

## Finite-state Machine (FSM)

- Historically a “mathematical model of computation” – Wikipedia
- General Game AI model for agent behavior with:
  - A finite number of states (S)
  - An initial state (I)
  - A transition function T(s) -> s’
  - Zero or more final states $S_{Final}$

## FSM Example


Monster In Sight
Gather Treasure
Flee
No Monster
Monster Dead
Fight
Cornered

# FSMs also represented as a table

| Current State | Condition | State Transition |
|---|---|---|
| Gather Treasure | Monster | Flee |
| Flee | Cornered | Fight |
| Flee | No Monster | Gather Treasure |
| Fight | Monster Dead | Gather Treasure |

## PQ1

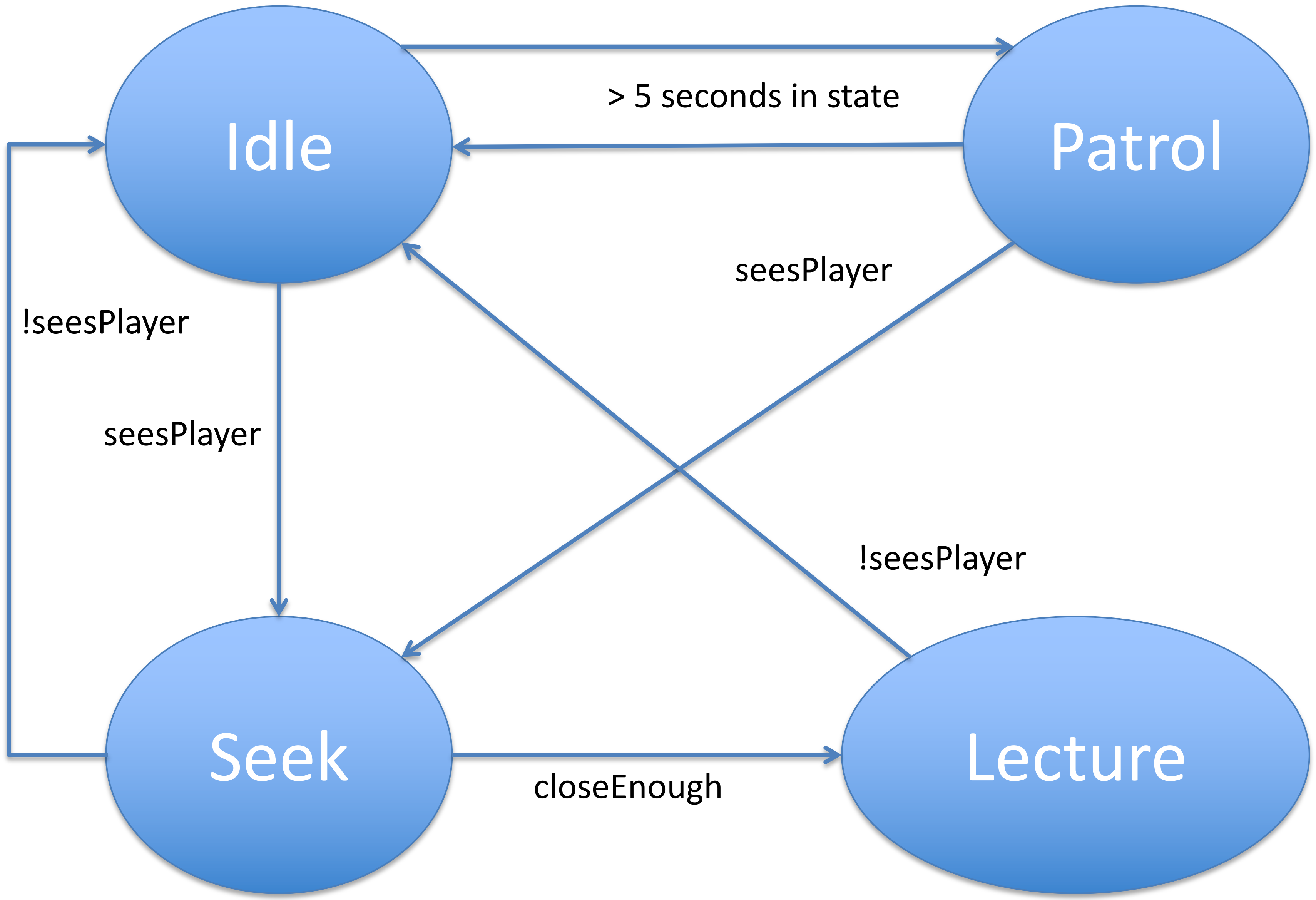
**https://forms.gle/13CcPynPYKmAsGta6**

**https://tinyurl.com/guz-pq14**

Design an FSM for a security guard that patrols a school after hours. If it sees the player it needs to seek the player and give them a lecture till the player can slip away. Otherwise switch between Idle and Patrol.

States = {Idle, Patrol, Seek, Lecture}

Feel free to make up variables/conditions.


Idle
Patrol
Seek
Lecture
> 5 seconds in state
seesPlayer
!seesPlayer
seesPlayer
!seesPlayer
closeEnough

## Advantages

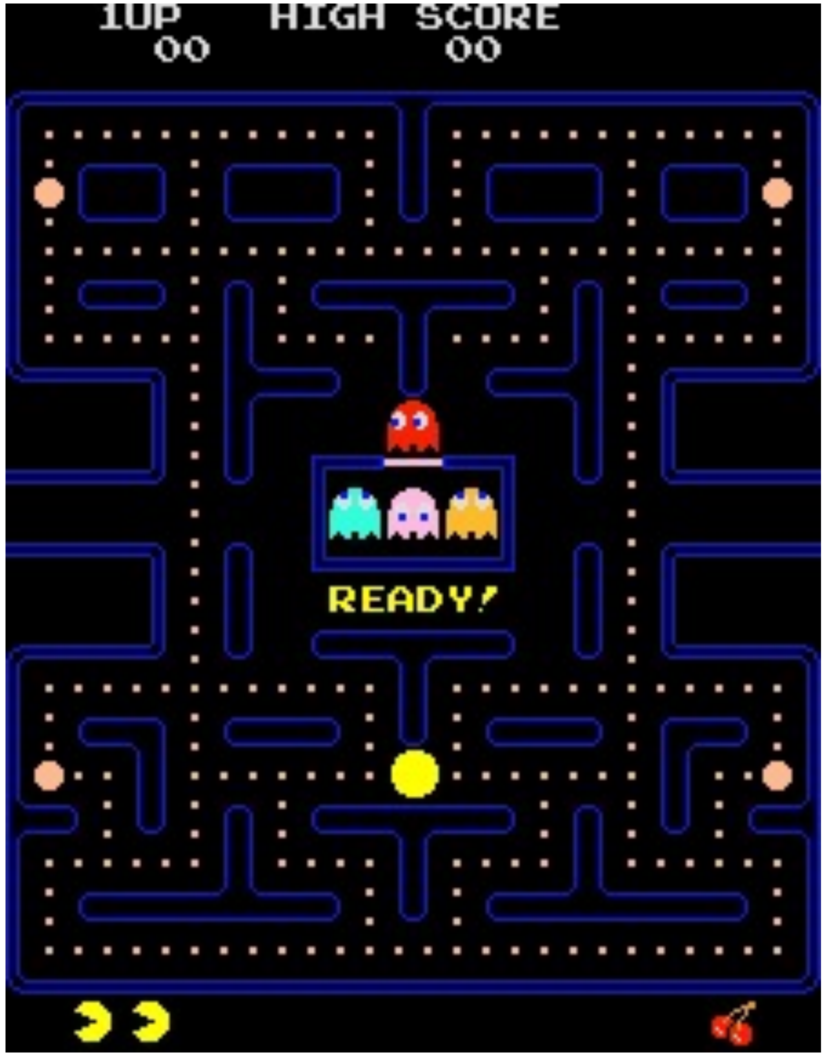
- Ubiquitous (not only in digital games)
- Quick and simple to code
- Easy to debug*
- Fast: small computational overhead
- Intuitive
- Flexible
- Easy for designers without coding knowledge

## Disadvantages

- When it fails, fails hard:
  - A transition from one state to another requires forethought (get stuck in a state or can’t do the “correct” next action) https://youtu.be/OjgZw4s-EOA
- Number of states can grow fast
  - Exponentially with number of events in world (multiple ways to react to same event given other varaibles)
- Number of transitions can grow even faster
- Doesn’t work with sequences of actions/memory

# Example: Pacman

https://www.webpacman.com/pacman-html5.php



# Implementations

1. Throw it all in a switch statement
   – Simple, but not extendable

```
void RunLogic( int state ) {
        switch( state ) {
                case 0: //Wander
                        Wander();
                        if( SeeEnemy() )
                                state = 1;
                        if( Dead() )
                                state = 2;
                break;
                case 1: //Attack
                        Attack();
                        state = 0;
                        if( Dead() )
                                state = 2;
                        break;
                case 2: //Dead
                        SlowlyRot()
                         break;
 }
```

# Implementations

1. Throw it all in a switch statement
   - Simple, but not extendable
2. State as class
   - Agent carries reference to current state. Extendable

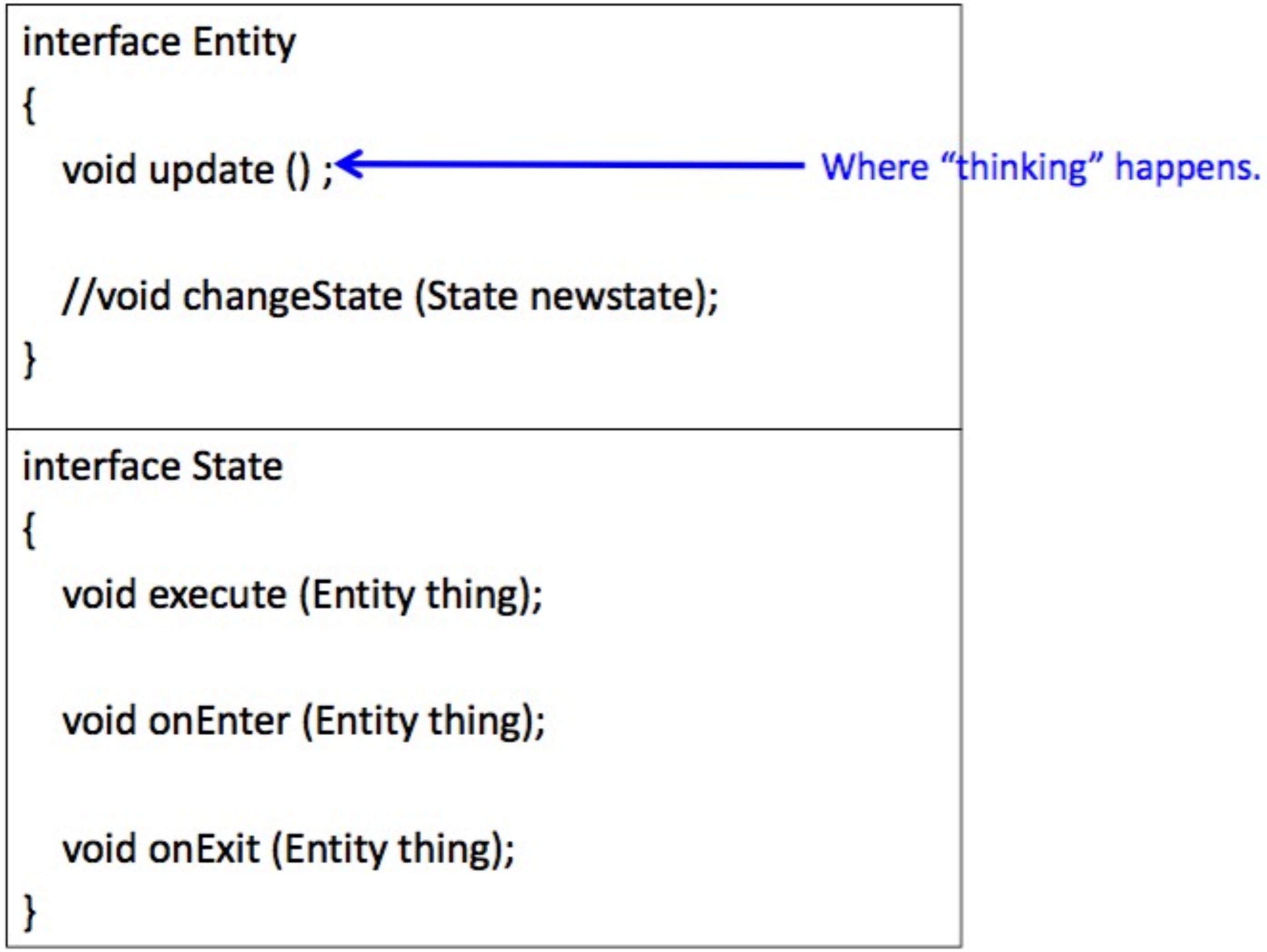
```
void RunLogic( int state ) {
        switch( state ) {
                case 0: //Wander
                        Wander();
                        if( SeeEnemy() )
                                state = 1;
                        if( Dead() )
                                state = 2;
                break;
                case 1: //Attack
                        Attack();
                        state = 0;
                        if( Dead() )
                                state = 2;
                        break;
                case 2: //Dead
                        SlowlyRot()
                         break;
}
```

# Implementations

1. Throw it all in a switch statement
   - Simple, but not extendable
2. State as class
   - Agent carries reference to current state. Extendable
3. State as table
   - Can be stored separately, easier for designers

```
void RunLogic( int state ) {
        switch( state ) {
                case 0: //Wander
                        Wander();
                        if( SeeEnemy() )
                                state = 1;
                        if( Dead() )
                                state = 2;
                break;
                case 1: //Attack
                        Attack();
                        state = 0;
                        if( Dead() )
                                state = 2;
                        break;
                case 2: //Dead
                        SlowlyRot()
                         break;
}
```

# State as class



```
interface Entity
{
   void update () ;

   //void changeState (State newstate);
}
```

Where “thinking” happens.

```
interface State
{
   void execute (Entity thing);

   void onEnter (Entity thing);

   void onExit (Entity thing);
}
```

## **PQ2**: How can we avoid getting stuck in a state?

https://forms.gle/UnYXanScf2iiRp8H6

https://tinyurl.com/guz-pq14b

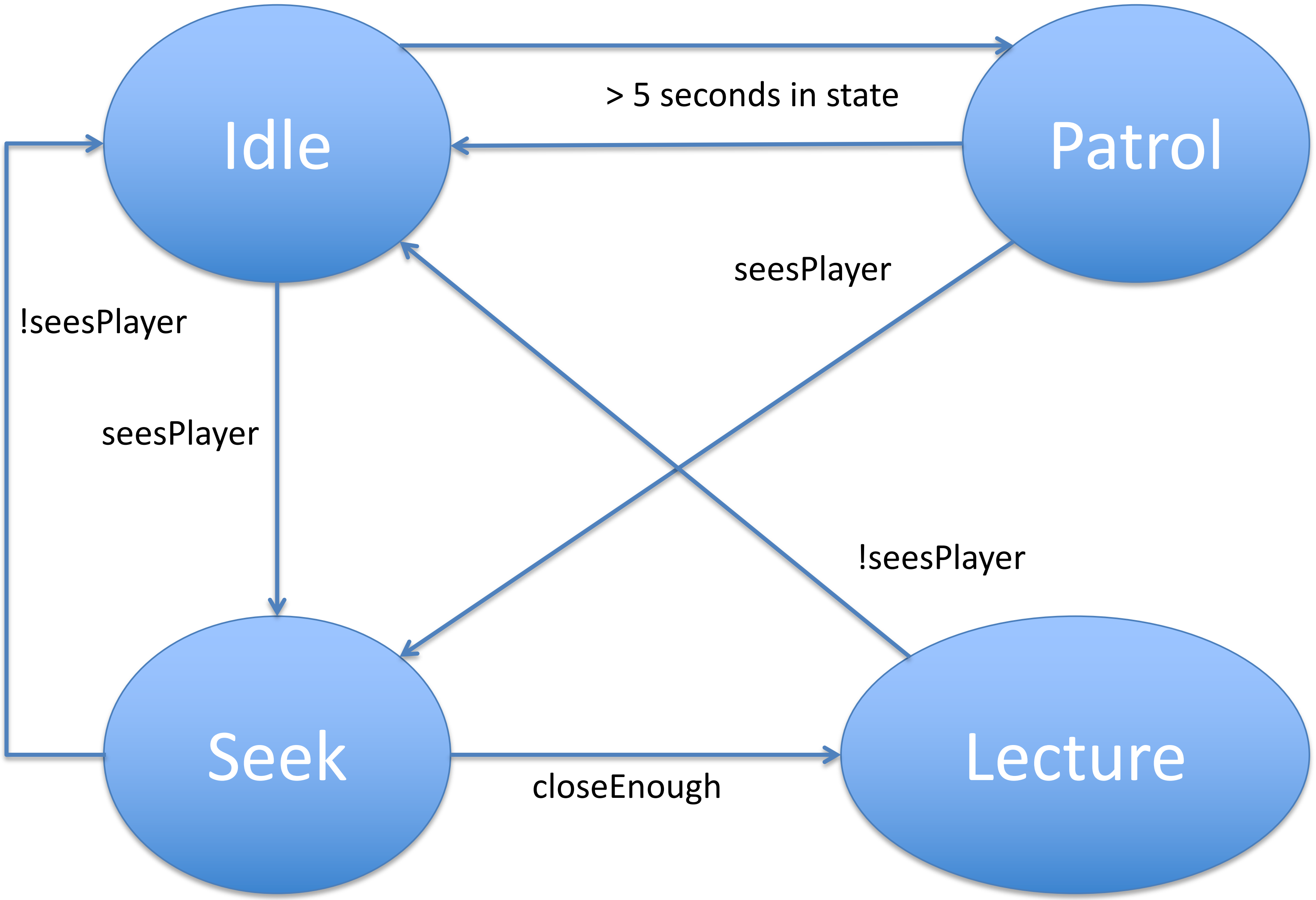
Naïve: Think ahead and author/design all possible transitions we might need between states.

What are alternatives that cost less design effort?

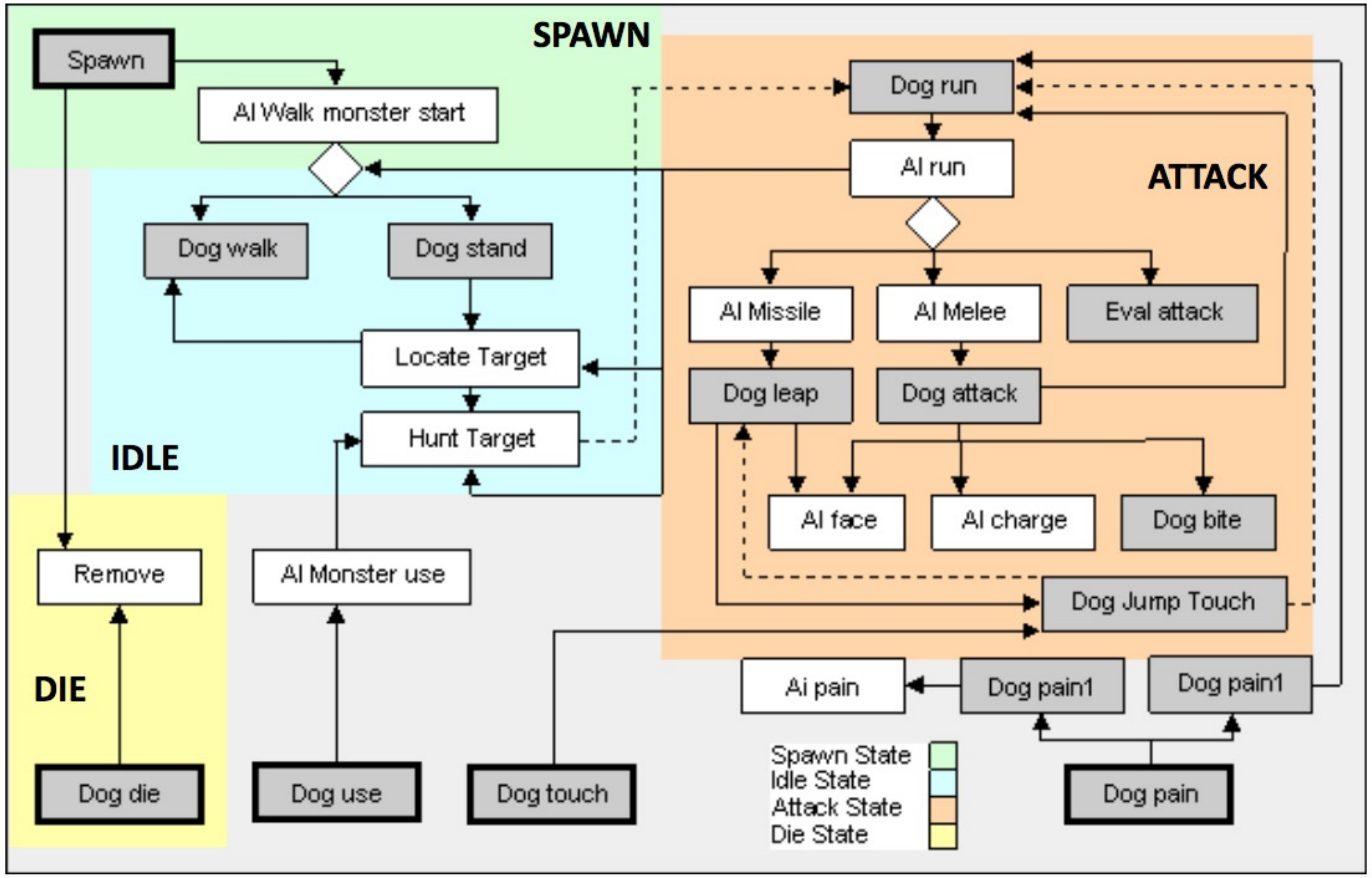
# Extending FSMs

- Stack FSMs
  - Push new state onto stack, when it’s done pop stack for next state
- Hierarchical FSMs
  - Two FSM “levels”. Abstract FSM that moves between low-level FSM “states”

# Motivating Stack FSMs



# Motivating Hierarchical FSMs



http://ai-depot.com/FiniteStateMachines/FSM-Framework.html

# References

- AI Game Programming Wisdom 2
- Web
  - http://ai-depot.com/FiniteStateMachines
  - http://www.gamasutra.com/view/feature/130279/creating_all_humans_a_data driven_.php
  - https://en.wikipedia.org/wiki/Virtual_finite-state_machine
- Buckland Ch 2
  - http://www.ai-junkie.com/architecture/state_driven/tut_state1.html
- Millington Ch 5
- Jarret Raim’s slides (Dr. Munoz-Avila’s GAI class 2005)
  - http://www.cse.lehigh.edu/~munoz/CSE497/classes/FSM_In_Games.ppt

# Disadvantages (now)

- When it fails, fails hard:
  - A transition from one state to another requires forethought (~~get stuck in a state~~ or can't do the "correct" next action)
- Number of states can grow fast
- ~~Number of transitions can grow even faster~~
- ~Doesn't work with sequences of actions/memory

## What if…

- We could fail gracefully?
  - If “confused” enter more and more general states

- Encode sequences of states?
  - Without having to bog each state down tracking more variables/conditions

# Behavior Trees (B trees)

- Popular in industry since 2004
- Easy to design
- Easy to alter
- Fail gracefully

- Coming to a lecture hall near you (Wednesday)

## Assignment 3 Review