the gamedesigninitiative at cornell university

Lecture 19

# Character Behavior

# Classical AI vs. Game AI

- **Classical**: Design of *intelligent agents*
  - Perceives environment, maximizes its success
  - Established area of computer science
  - Subtopics: planning, machine learning
- **Game**: Design of *rational behavior*
  - Does not need to optimize (and often will not)
  - Often about "scripting" a personality
  - More akin to cognitive science

# Role of AI in Games

- **Autonomous Characters** (NPCs)
  - Mimics the "personality" of the character
  - May be opponent or support character
- **Strategic Opponents**
  - AI at the "player level"
  - Closest to classical AI
- **Character Dialog**
  - Intelligent commentary
  - Narrative management (e.g. Façade)

# Role of AI in Games

- **Autonomous Characters** (NPCs)
  - Mimics the "personality" of the character
  - May be opponent or support character
- **Strategic Opponents**
  - AI at the "player level"
  - Closest to classical AI
- **Character Dialog**
  - Intelligent commentary
  - Narrative management (e.g. Façade)

# Review: Sense-Think-Act

- **Sense**:
  - Perceive the world
  - Reading the game state
  - **Example**: enemy near?
- **Think**:
  - Choose an action
  - Often merged with sense
  - **Example**: fight or flee
- **Act**:
  - Update the state
  - Simple and fast
  - **Example**: reduce health

# S-T-A: Separation of Logic

- **Loops** = sensing
  - Read other objects
  - *Aggregate* for thinking
  - **Example**: nearest enemy
- **Conditionals** = thinking
  - Use results of sensing
  - Switch between possibilities
  - **Example**: attack or flee
- **Assignments** = actions
  - Rarely need loops
  - Avoid conditionals

```
move(int direction) {
  switch (direction) {
  case NORTH:
    y -= 1;
    break;
  case EAST:
    x += 1;
    break;
  case SOUTH:
    y += 1;
    break;
  case WEST:
    x -= 1;
    break;
  }
}
```

# S-T-A: Separation of Logic

- **Loops** = sensing
  - Read other objects
  - *Aggregate* for thinking
  - **Example**: nearest enemy
- **Conditionals** = thinking
  - Use results of sensing
  - Switch between possibilities
  - **Example**: attack or flee
- **Assignments** = actions
  - Rarely need loops
  - Avoid conditionals

```
move(int direction) {
  switch (direction) {
  case NORTH:
    y -= 1;
    break;
  case EAST:
    x += 1;
    break;
  case SOUTH:
    y += 1;
    break;
  case WEST:
    x -= 1;
    break;
  }
}
```

# S-T-A: Separation of Logic

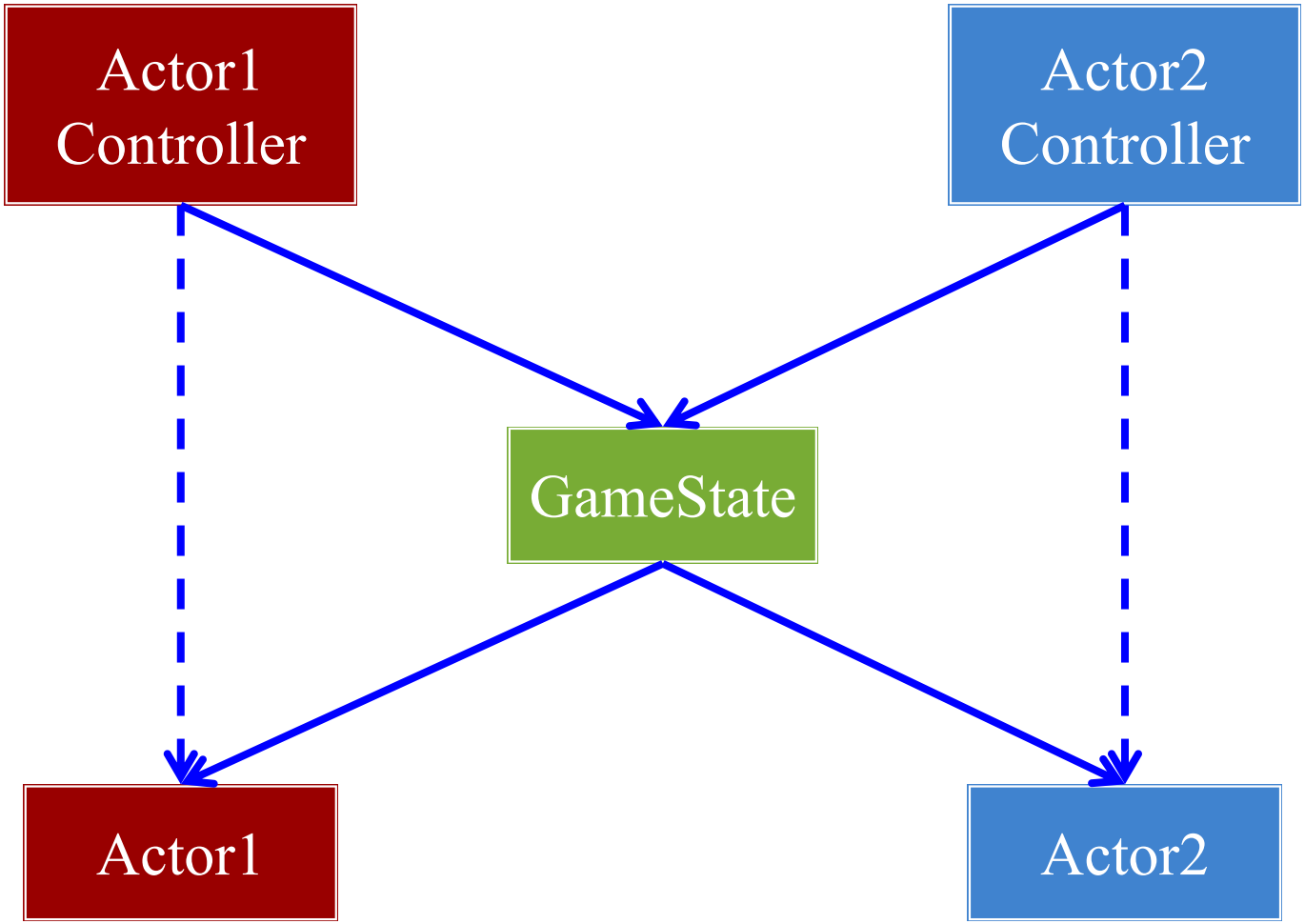
- **Loops** = sensing
  - Read other objects
  - *Aggregate* for thinking
  - **Example**: nearest enemy
- **Conditionals** = thinking
  - Use results of sensing
  - Switch between possibilities
  - **Example**: attack or flee
- **Assignments** = actions
  - Rarely need loops
  - Avoid conditionals

```
move(int direction) {
  switch (direction) {
  case NORTH:
    y -= 1;
    break;
  case EAST:
```

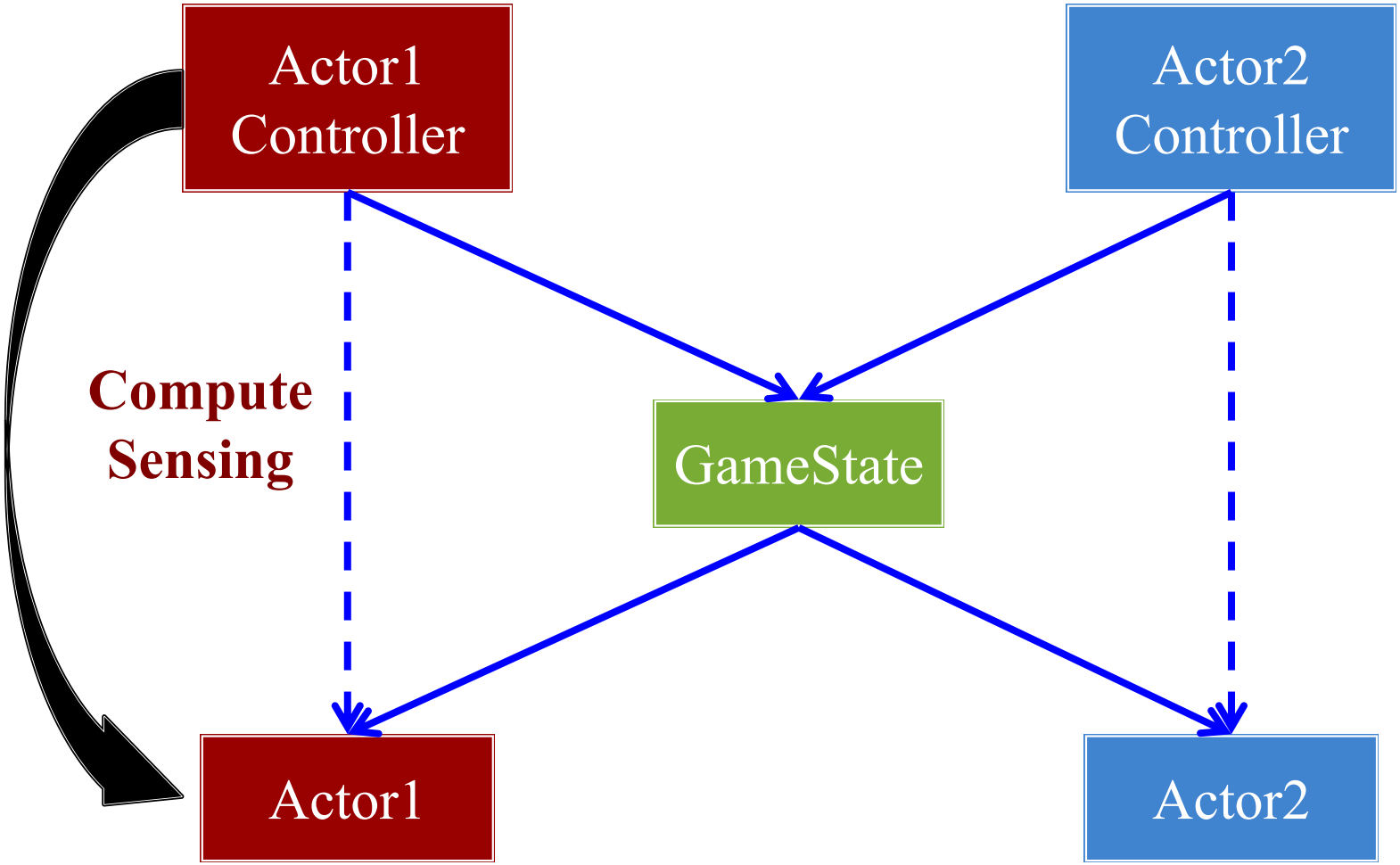
```
    move(int dx, int dy) {
        x += dx;
        y += dy;
    }
```

```
  case WEST:
    x -= 1;
    break;
  }
}
```

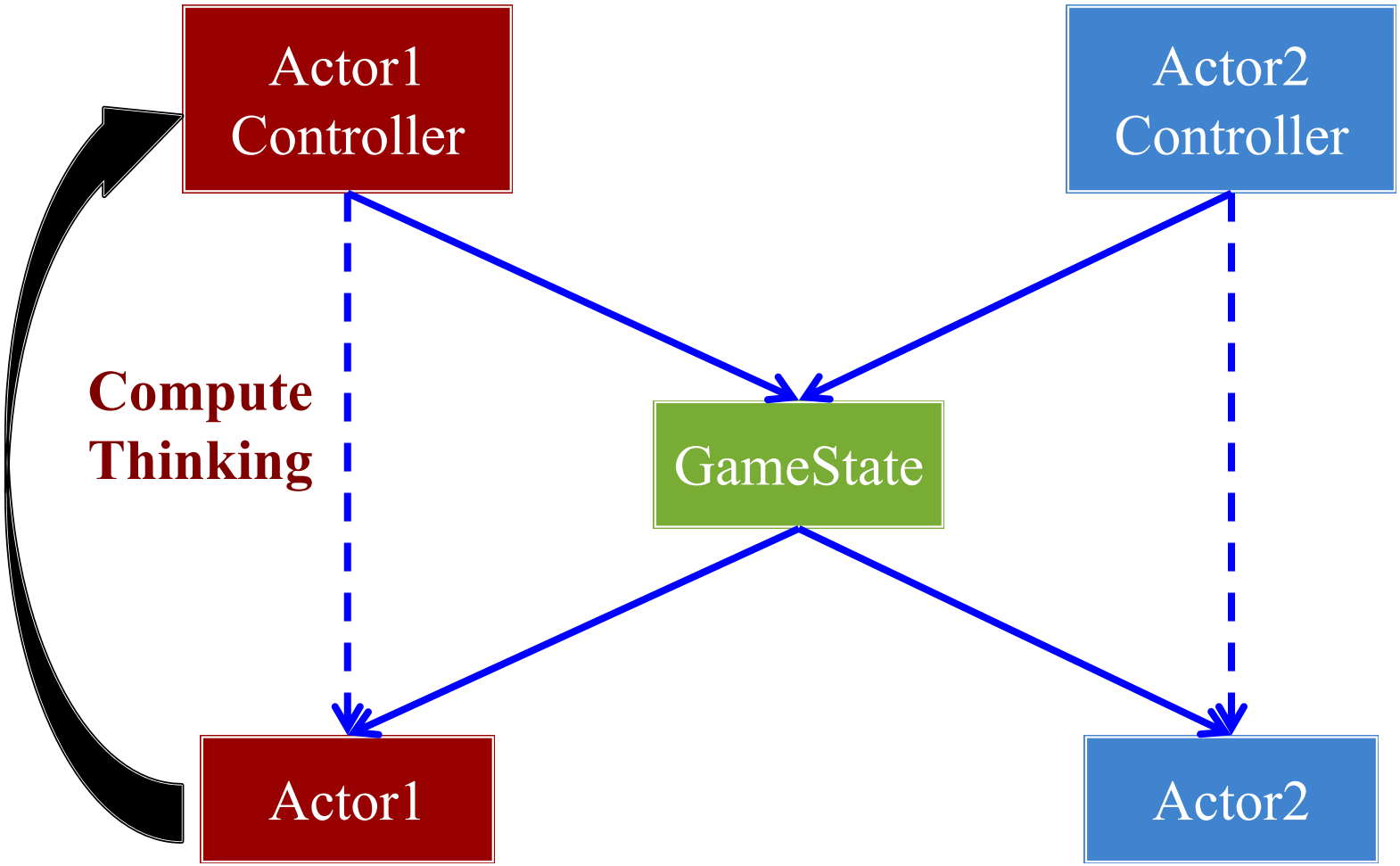
# S-T-A: Reducing Dependencies



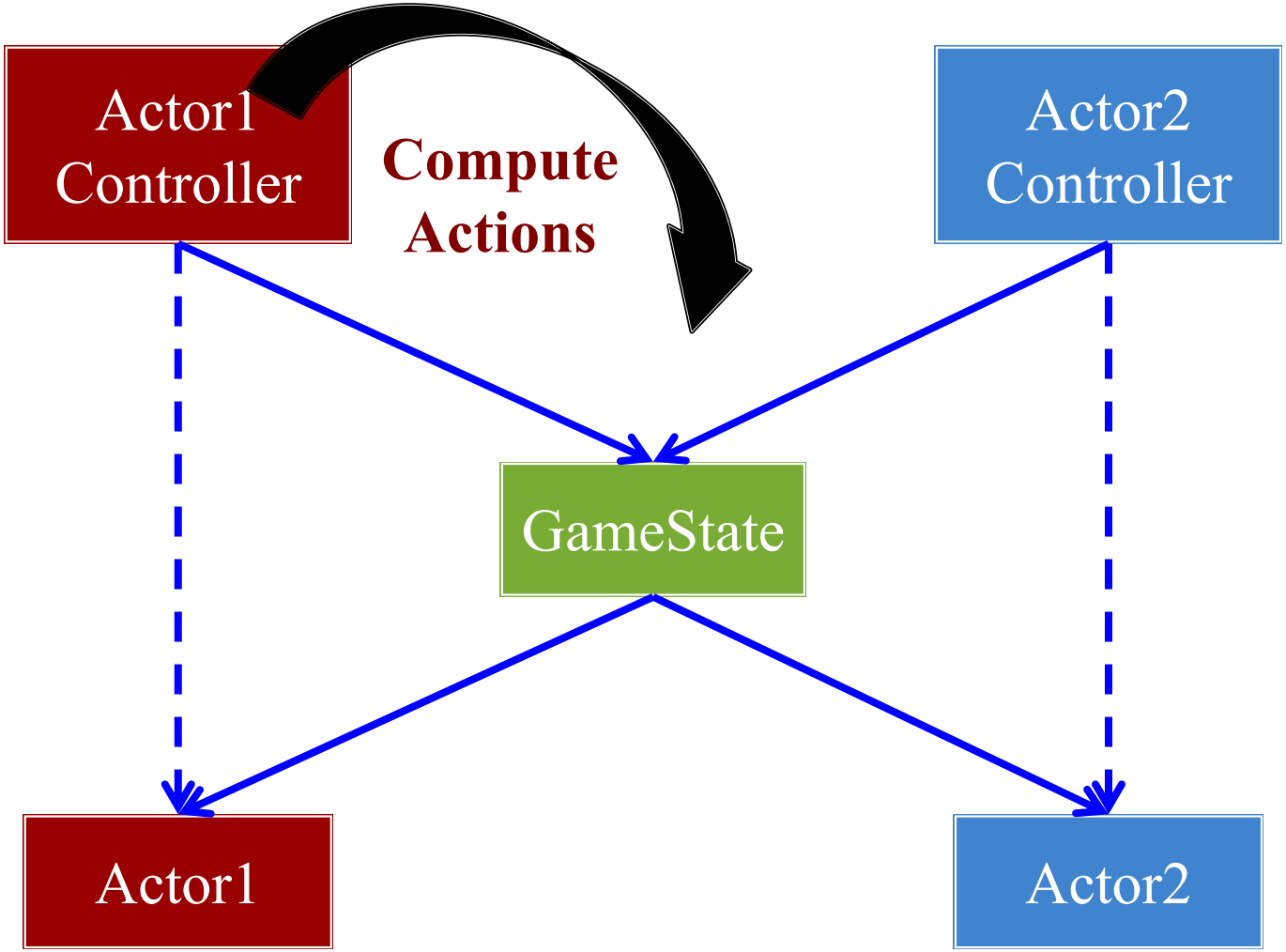
# S-T-A: Reducing Dependencies



# S-T-A: Reducing Dependencies



# S-T-A: Reducing Dependencies

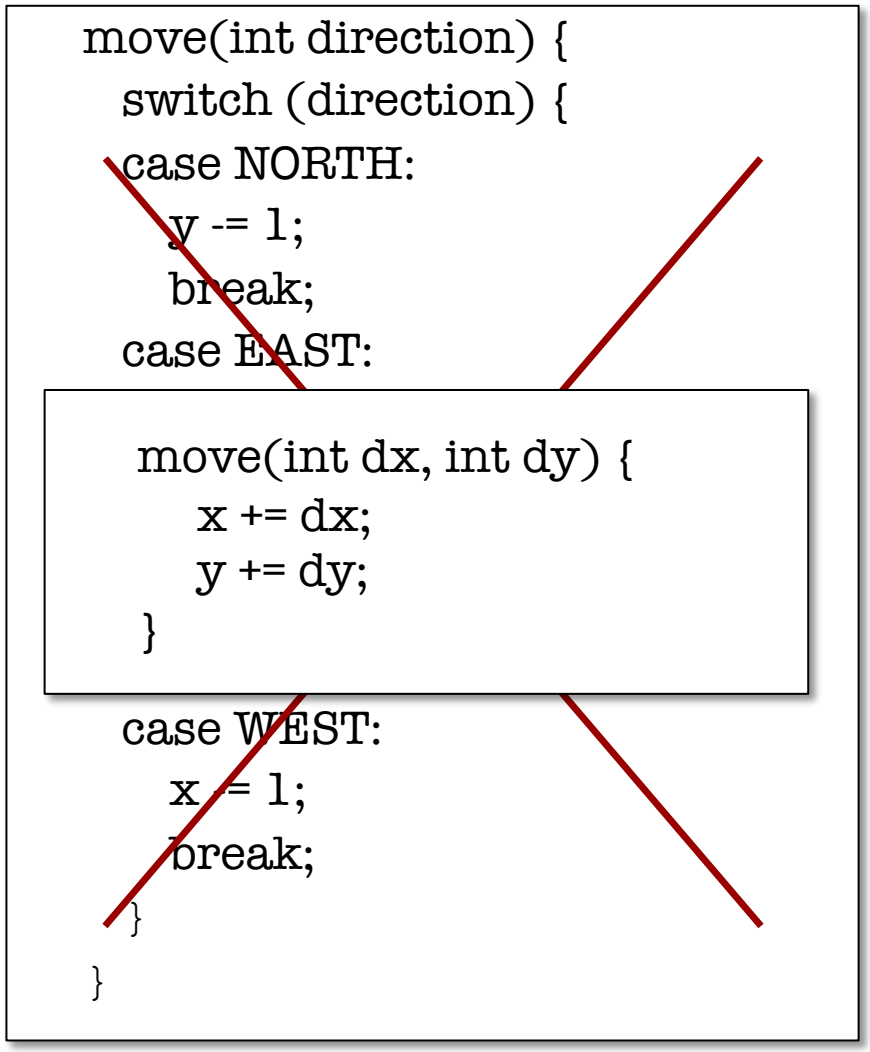


# Review: Sense-Think-Act

- **Sense**:
  - Perceive the world
  - Reading the game state
  - **Example**: enemy near?
- **Think**:
  - Choose an action
  - Often merged with sense
  - **Example**: fight or flee
- **Act**:
  - Update the state
  - Simple and fast
  - **Example**: reduce health

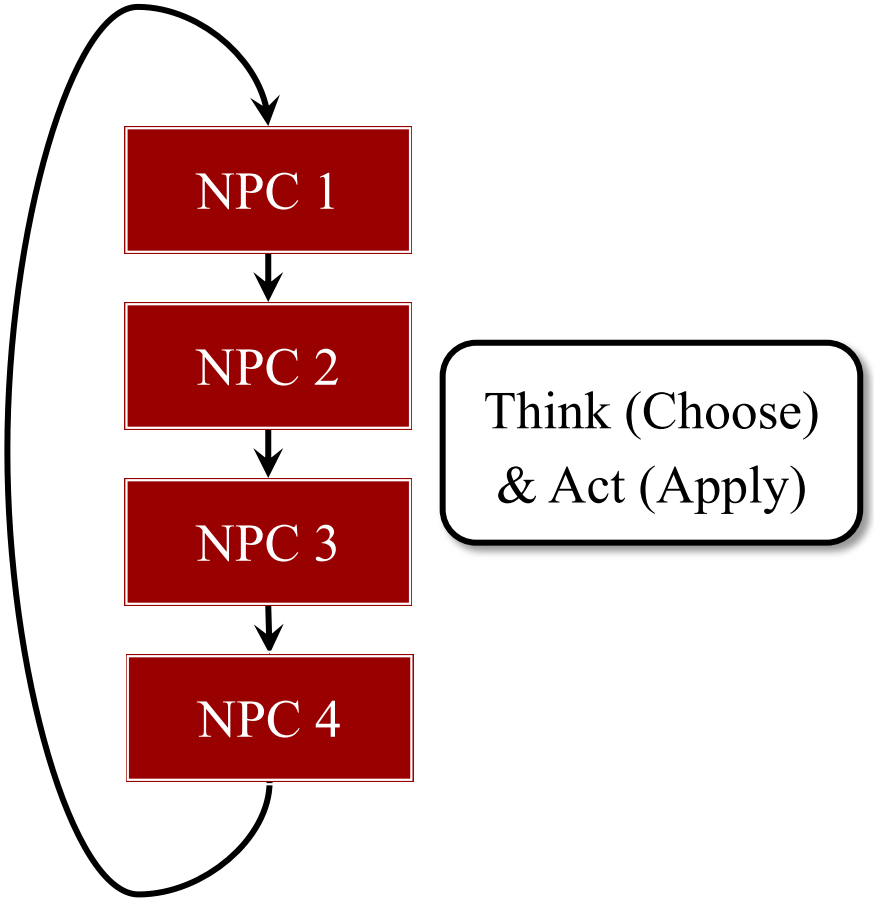
# Actions: Short and Simple

- Mainly use **assignments**
  - Avoid loops, conditionals
  - Similar to getters/setters
  - Complex code in thinking
- Helps with **serializability**
  - Record and undo actions
- Helps with **networking**
  - Keep doing last action
  - See: *dead reckoning*

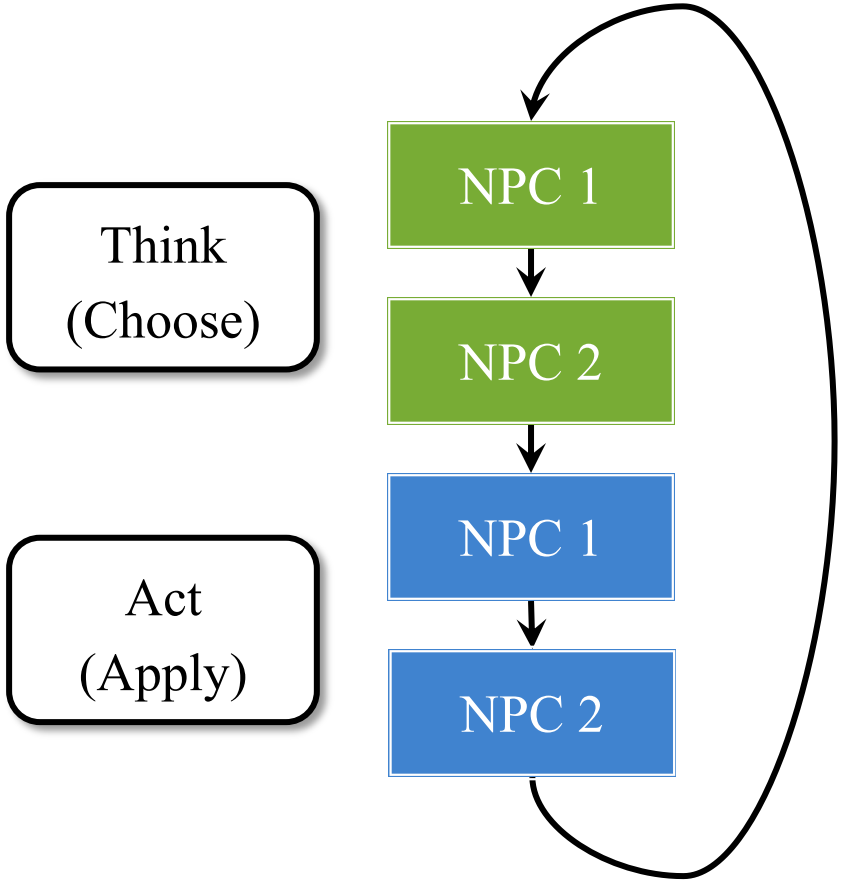


# Delaying Actions

## Sequential Actions are Bad




## Choose Action; Apply Later



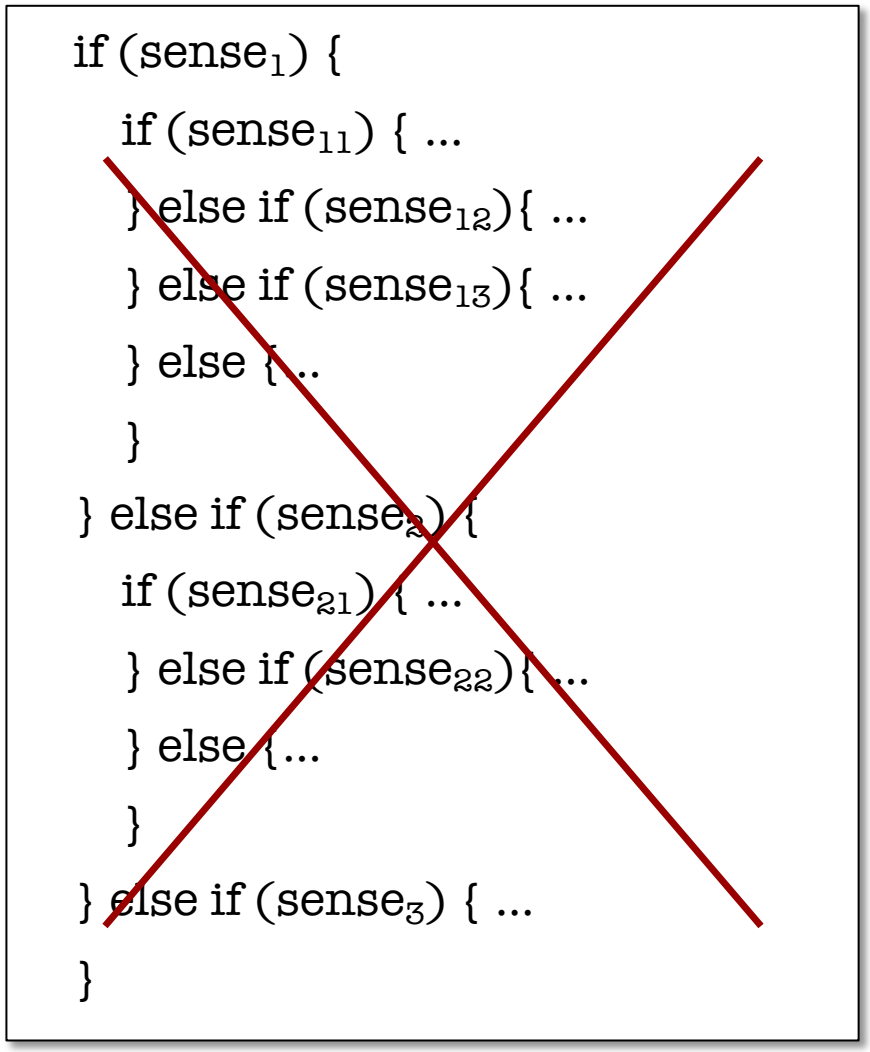
# Thinking: Primary Challenge

- A mess of conditionals
  - "Spaghetti" code
  - Difficult to modify
- Abstraction requirements:
  - Easy to visualize models
  - Mirror "cognitive thought"
- Want to separate talent
  - **Sensing**: Programmers
  - **Thinking**: *Designers*
  - **Actions**: Programmers

```
if (sense1) {
   if (sense11) { ...
   } else if (sense12){ ...
   } else if (sense13){ ...
   } else {...
   }
} else if (sense2) {
   if (sense21) { ...
   } else if (sense22){ ...
   } else {...
   }
} else if (sense3) { ...
}
```

# Thinking: Primary Challenge

- A mess of conditionals
  - "Spaghetti" code
  - Difficult to modify
- Abstraction requirements:
  - Easy to visualize models
  - Mirror "cognitive thought"
- Want to separate talent
  - **Sensing**: Programmers
  - **Thinking**: *Designers*
  - **Actions**: Programmers



# Rule-Based AI

If ***X*** is true, Then do ***Y***

- **Thinking**: Provide a list of several rules
  - But what happens if there is more than one rule?
  - Which rule do we choose?

# Rule-Based AI


- **Thinking**: Provide a list of several rules
  - But what happens if there is more than one rule?
  - Which rule do we choose?

# Simplicity of Rule-Based AI

# Conflict Resolution

- Often **resolve by order**
  - Each rule has a priority
  - Higher priorities go first
  - "Flattening" conditionals
- **Problems**:
  - Predictable
    Same events = same rules
  - Total order
    Sometimes no preference
  - Performance
    On average, go far down list

$R_1$: `if` $event_1$ `then` $act_1$

$R_2$: `if` $event_2$ `then` $act_2$

$R_3$: `if` $event_3$ `then` $act_3$

$R_4$: `if` $event_4$ `then` $act_4$

$R_5$: `if` $event_5$ `then` $act_5$

$R_6$: `if` $event_6$ `then` $act_6$

$R_7$: `if` $event_7$ `then` $act_7$

# Impulses

- Correspond to certain events
  - **Global**: not tied to NPC
  - Must also have duration
- Used to **reorder** rules
  - Event makes rule important
  - Temporarily up the priority
  - Restore when event is over
- Preferred conflict resolution
  - Simple but flexible
  - Used in *Halo* 3 AI.

$R_1$: `if` $event_1$ `then` $act_1$

$R_2$: `if` $event_2$ `then` $act_2$

$R_3$: `if` $event_3$ `then` $act_3$

$R_4$: `if` $event_4$ `then` $act_4$

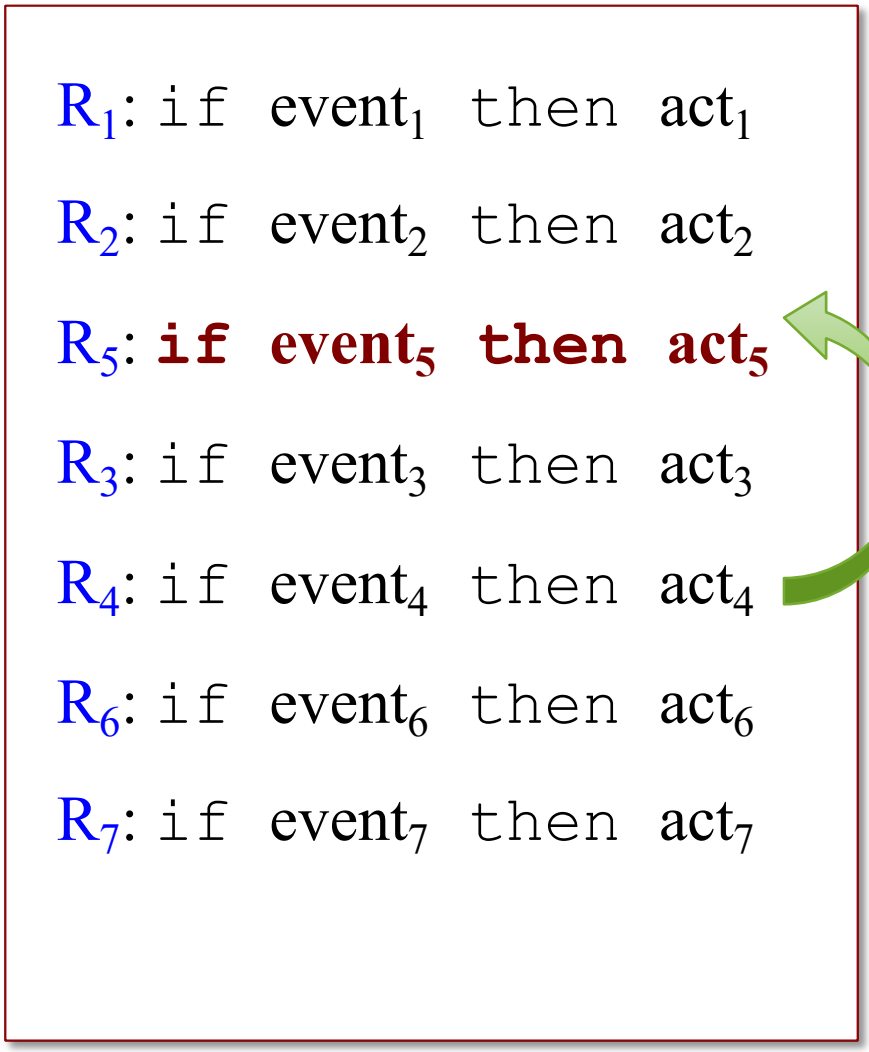
$R_5$: `if` $event_5$ `then` $act_5$

$R_6$: `if` $event_6$ `then` $act_6$

$R_7$: `if` $event_7$ `then` $act_7$

# Impulses

- Correspond to certain events
  - **Global**: not tied to NPC
  - Must also have duration
- Used to **reorder** rules
  - Event makes rule important
  - Temporarily up the priority
  - Restore when event is over
- Preferred conflict resolution
  - Simple but flexible
  - Used in *Halo* 3 AI.



# Impulses

- Correspond to certain events
  - **Global**: not tied to NPC
  - Must also have duration
- Used to **reorder** rules
  - Even
  - Temp
  - Resto
- Preferred conflict resolution
  - Simple but flexible
  - Used in *Halo* 3 AI.

$R_1$: `if` $event_1$ `then` $act_1$

$R_2$: `if` $event_2$ `then` $act_2$

$R_5$: **`if event`** **`then`** **$act_5$**

n $act_3$

en $act_4$

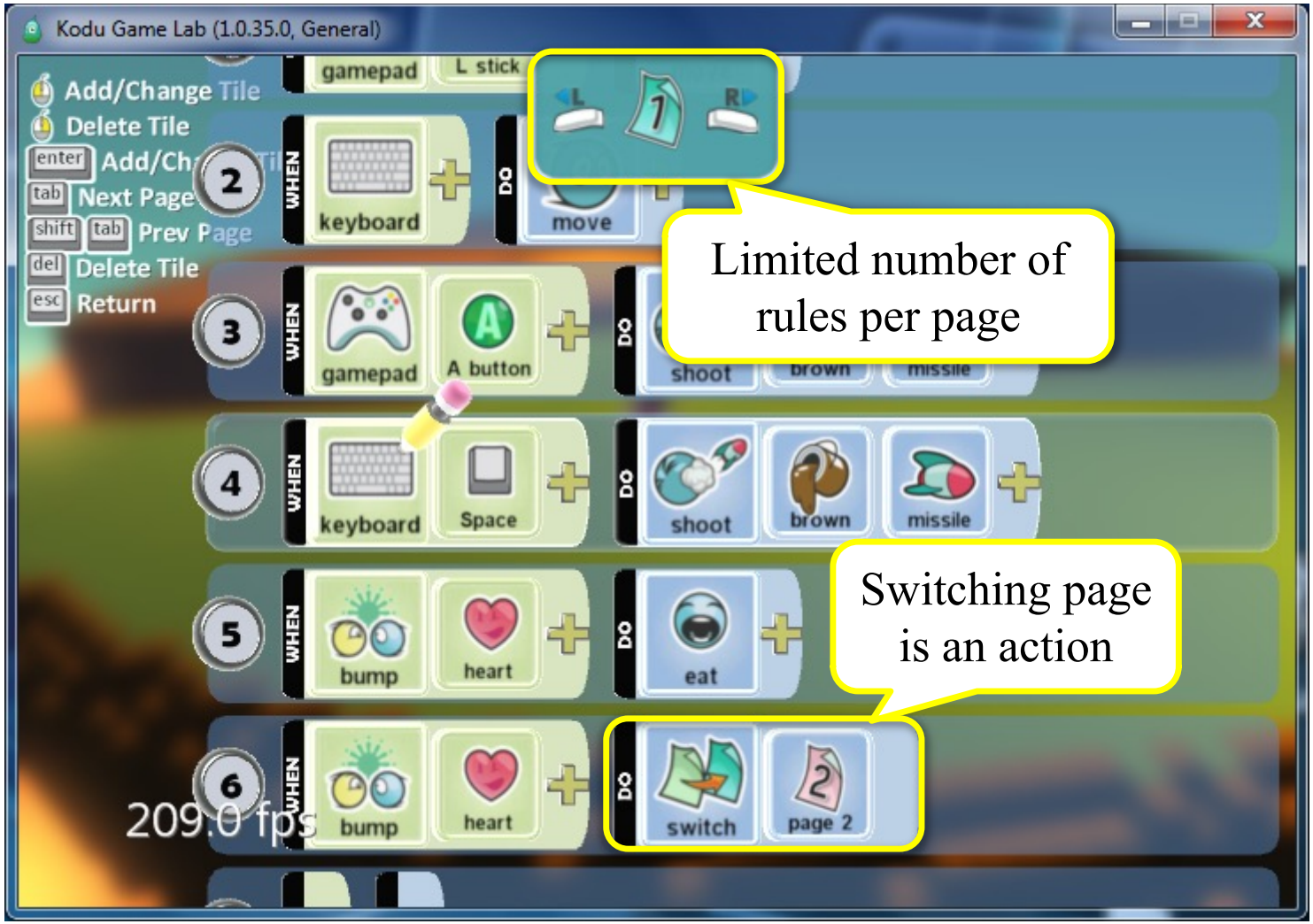
$R_6$: `if` $event_6$ `then` $act_6$

$R_7$: `if` $event_7$ `then` $act_7$

But this is still very limited.
We need a better way to organize.

# Making the Rules Manageable

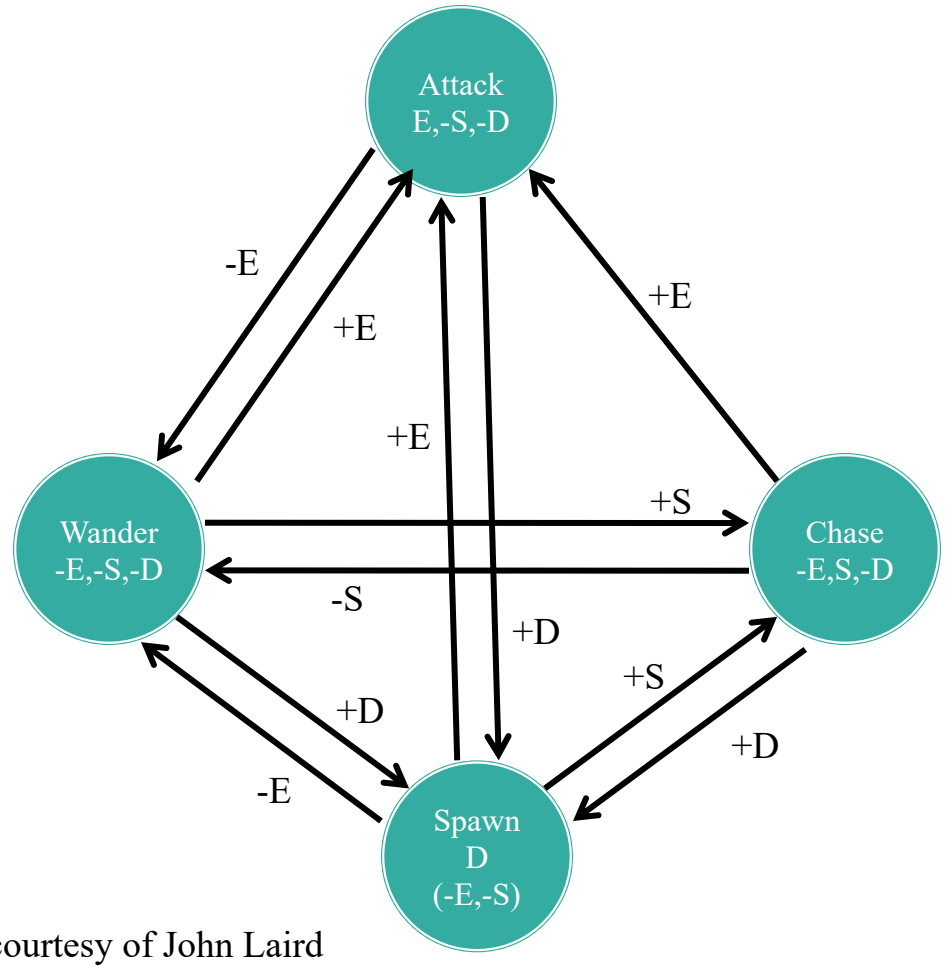
# Making the Rules Manageable



# Managing Rule-Based Behavior

- **Finite State Machines**
  - Group the rules into states (the Kodu example)
  - Effectively what you did in Lab 2
- **Utility Systems**
  - Technique use in the *Sims* games
  - Powerful tool with a lot of emergent behavior
- **Behavior Trees**
  - Popularized by Unreal and Unity
  - Considered the modern standard for AI

# Finite State Machines



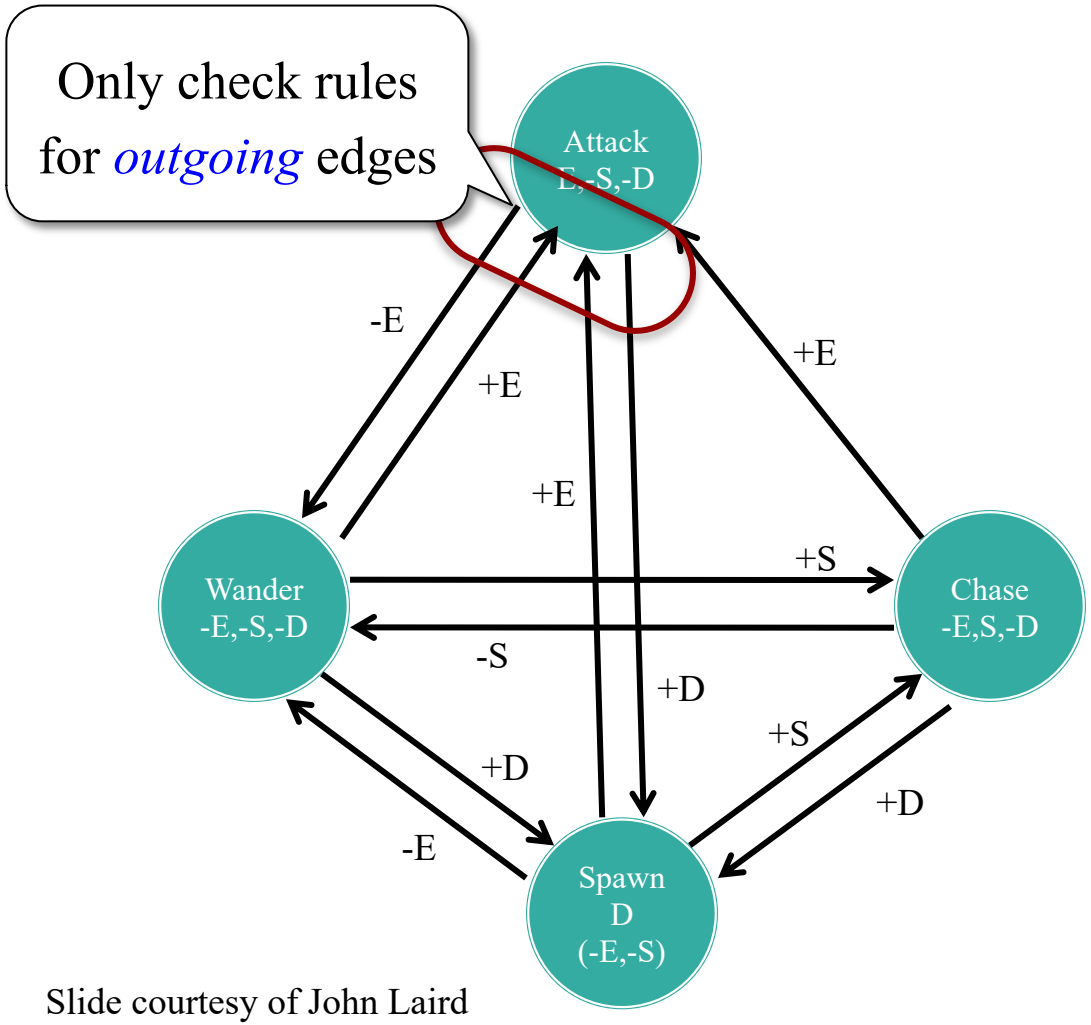

## Events

- **E**=Enemy Seen
- **S**=Sound Heard
- **D**=Die

**Simplify sensing as events**

Slide courtesy of John Laird

# Finite State Machines




## Events

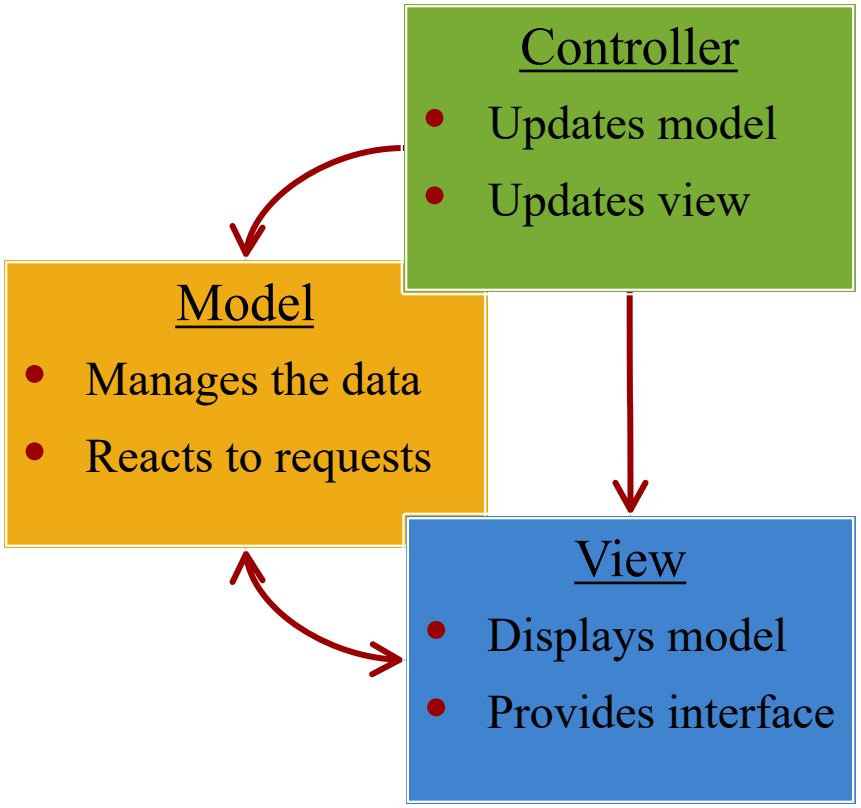
- **E**=Enemy Seen
- **S**=Sound Heard
- **D**=Die

Simplify sensing as **events**

Slide courtesy of John Laird

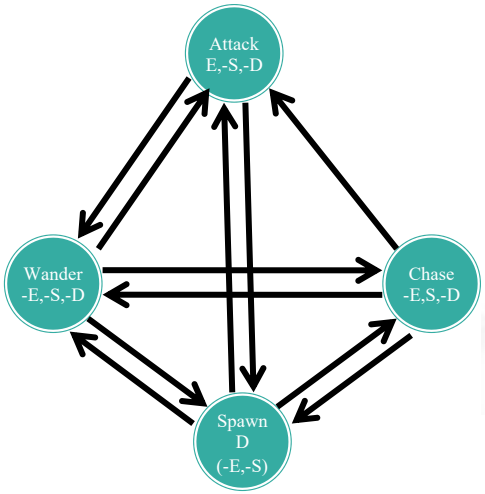
# Implementation: Model-View-Controller

- Games have **thin** models
  - Methods = get/set/update
  - Controllers are heavyweight
- AI is a **controller**
  - Uniform process over NPCs
- But behavior is *personal*
  - Diff. NPCs = diff. behavior
  - Do not want unique code
- What can we do?
  - Data-Driven Design



# Implementation: Model-View-Controller

- **Actions** go in the model
  - Lightweight updates
  - Specific to model or role
- Controller is framework for general **sensing**, **thinking**
  - Standard FSM engine
  - Or FSM alternatives (later)
- **Process** stored in a model
  - Represent thinking as *graph*
  - Controller processes graph




wander

chase

attack

spawn

AI Controller

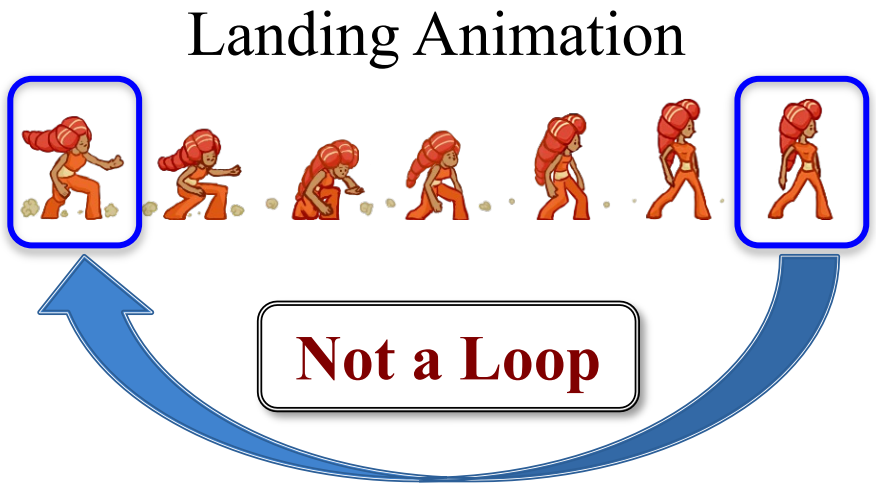
# An Aside: Animations

Landing Animation

Idling Animation

- AI may need many actions
  - Run, jump, duck, slide
  - Fire weapons, cast spells
  - Fidget while idling
- Want animations for all
  - Is loop appropriate for each?
  - How do we transition?
- **Idea**: shared boundaries
  - End of loop = start of another
  - Treat like advancing a frame

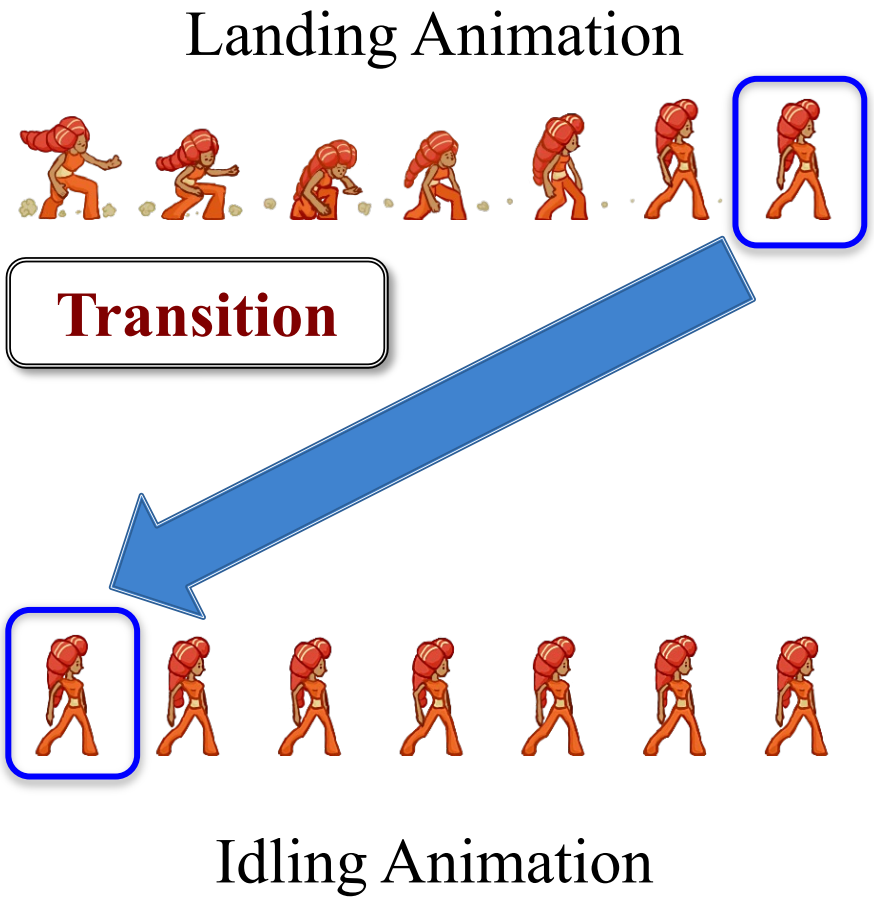
# An Aside: Animations




Idling Animation

- AI may need many actions
  - Run, jump, duck, slide
  - Fire weapons, cast spells
  - Fidget while idling
- Want animations for all
  - Is loop appropriate for each?
  - How do we transition?
- **Idea**: shared boundaries
  - End of loop = start of another
  - Treat like advancing a frame

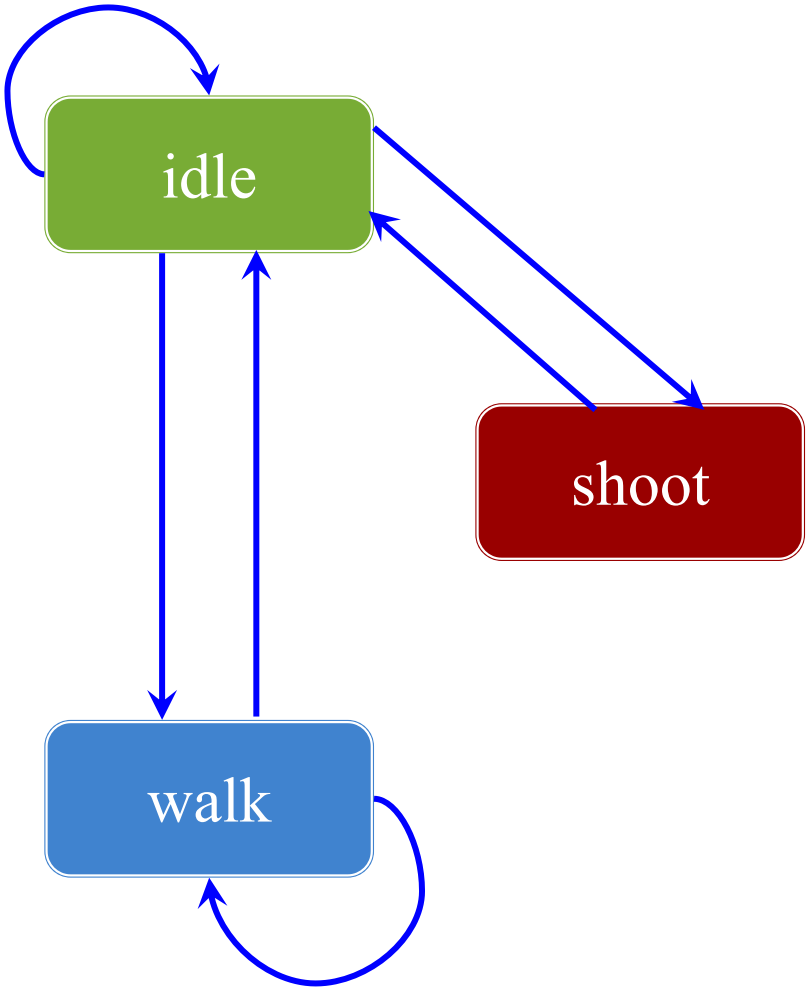
# An Aside: Animations




- AI may need many actions
  - Run, jump, duck, slide
  - Fire weapons, cast spells
  - Fidget while idling
- Want animations for all
  - Is loop appropriate for each?
  - How do we transition?
- **Idea**: shared boundaries
  - End of loop = start of another
  - Treat like advancing a frame

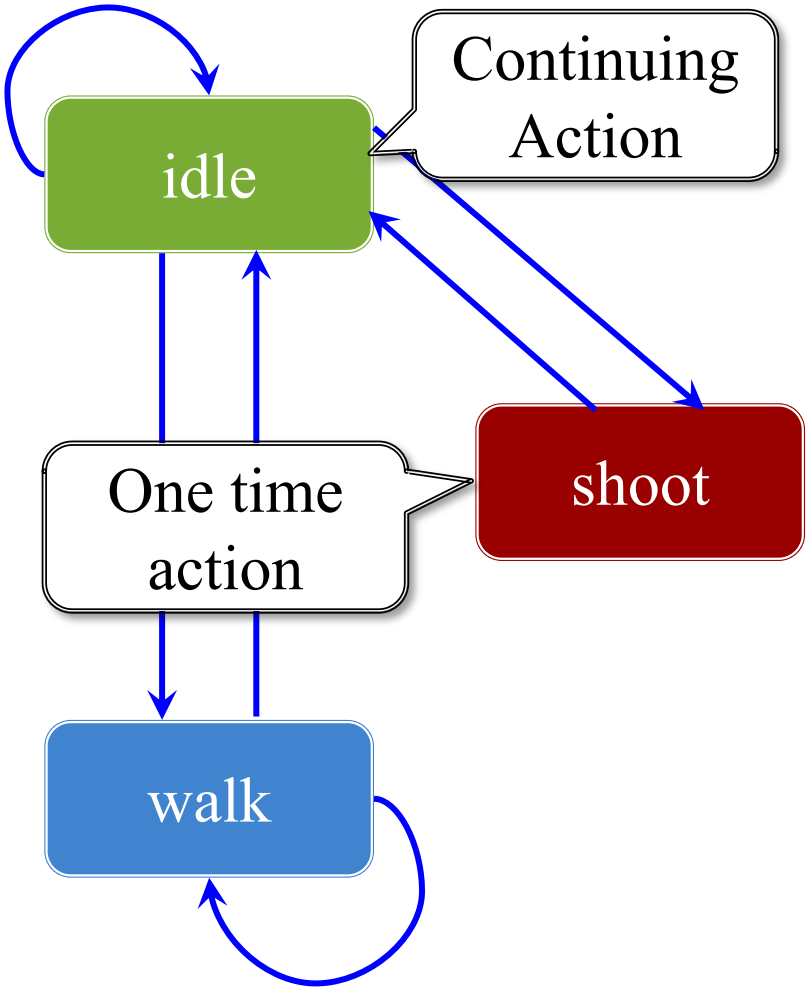
# Animation and State Machines

- **Idea**: Each sequence a state
  - Do sequence while in state
  - Transition when at end
  - Only loop if loop in graph
- A graph edge means…
  - Boundaries match up
  - Transition is allowable
- Similar to data driven AI
  - Created by the designer
  - Implemented by programmer
  - Modern engines have tools

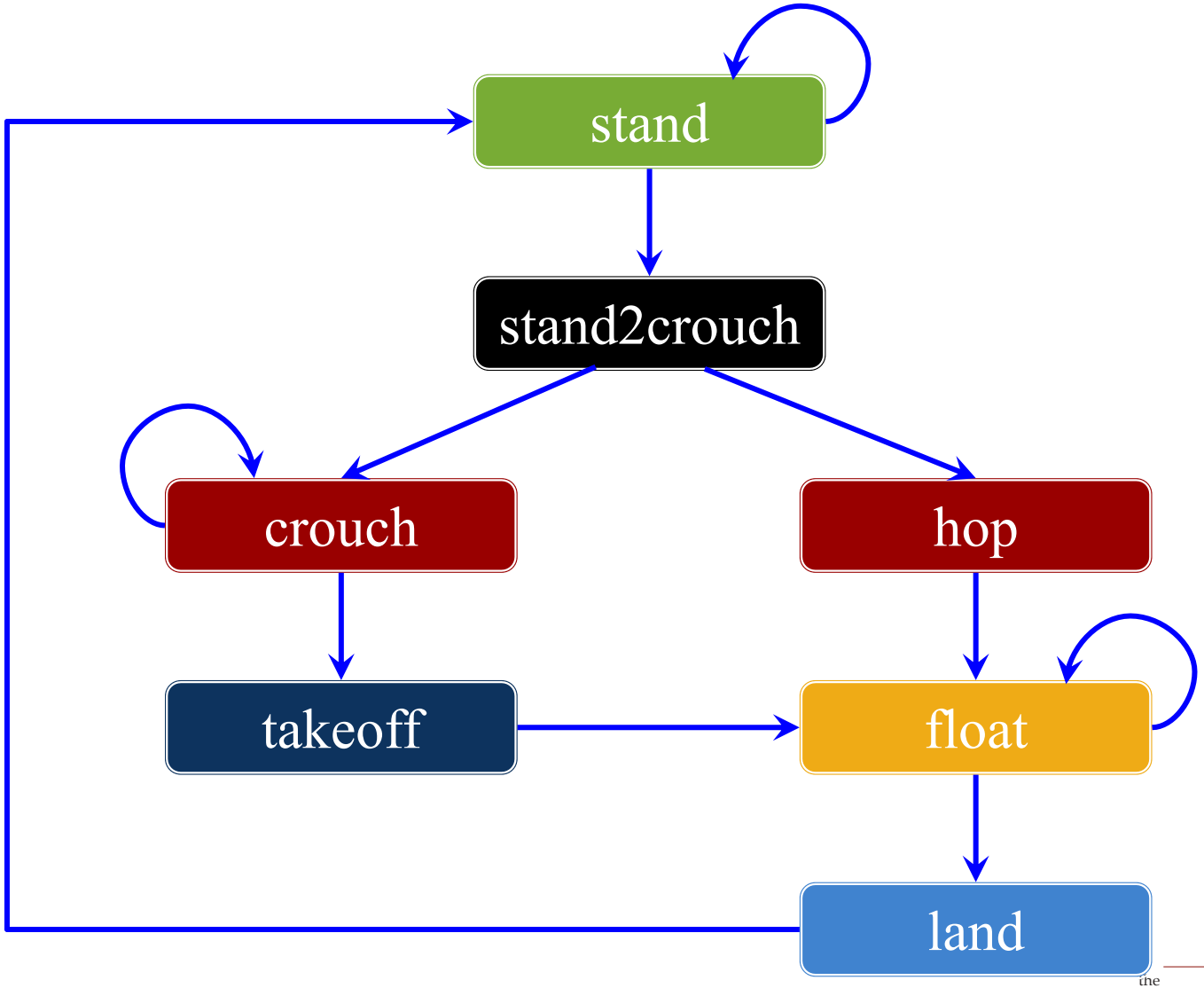


# Animation and State Machines

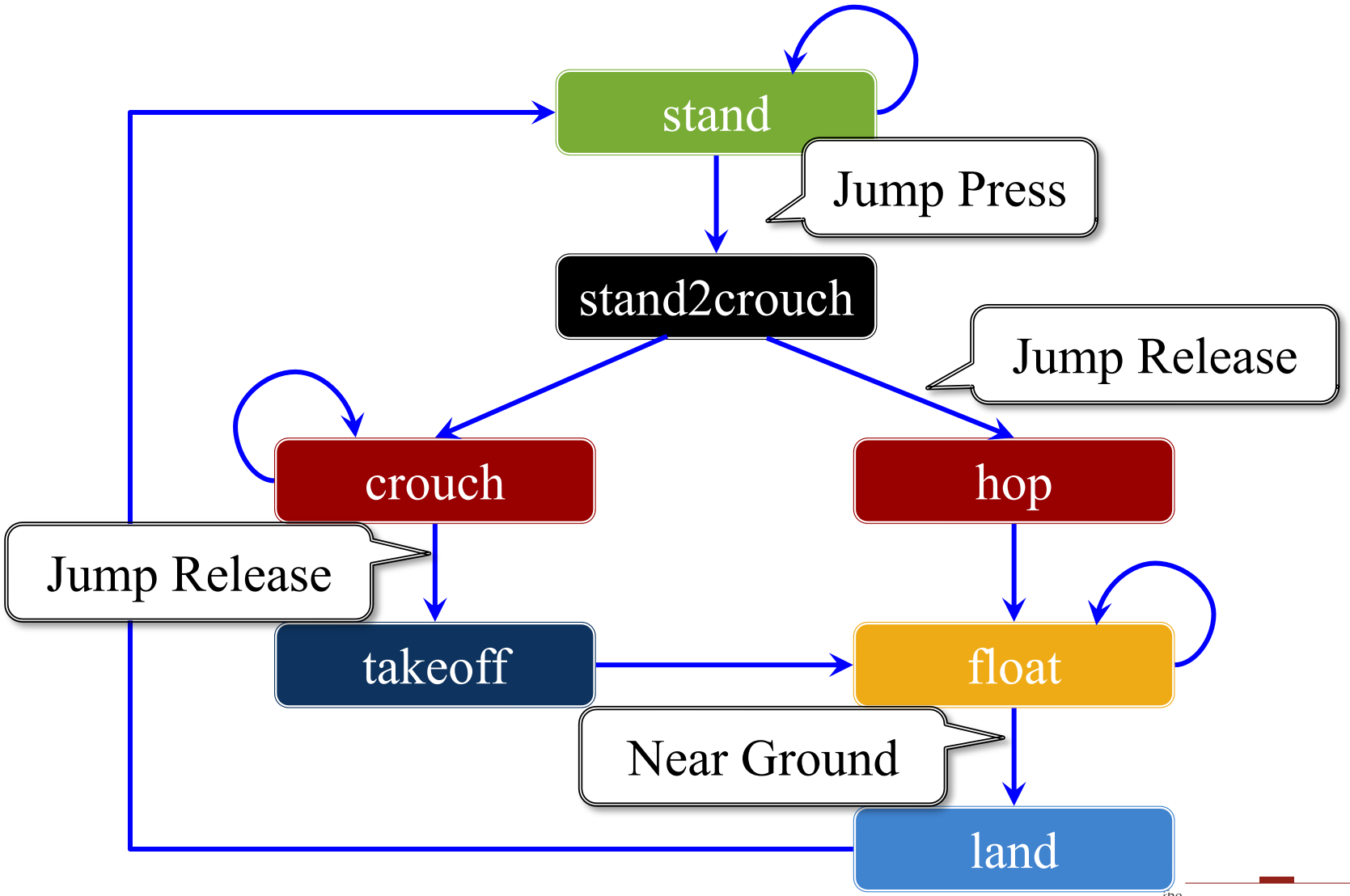
- **Idea**: Each sequence a state
  - Do sequence while in state
  - Transition when at end
  - Only loop if loop in graph
- A graph edge means…
  - Boundaries match up
  - Transition is allowable
- Similar to data driven AI
  - Created by the designer
  - Implemented by programmer
  - Modern engines have tools



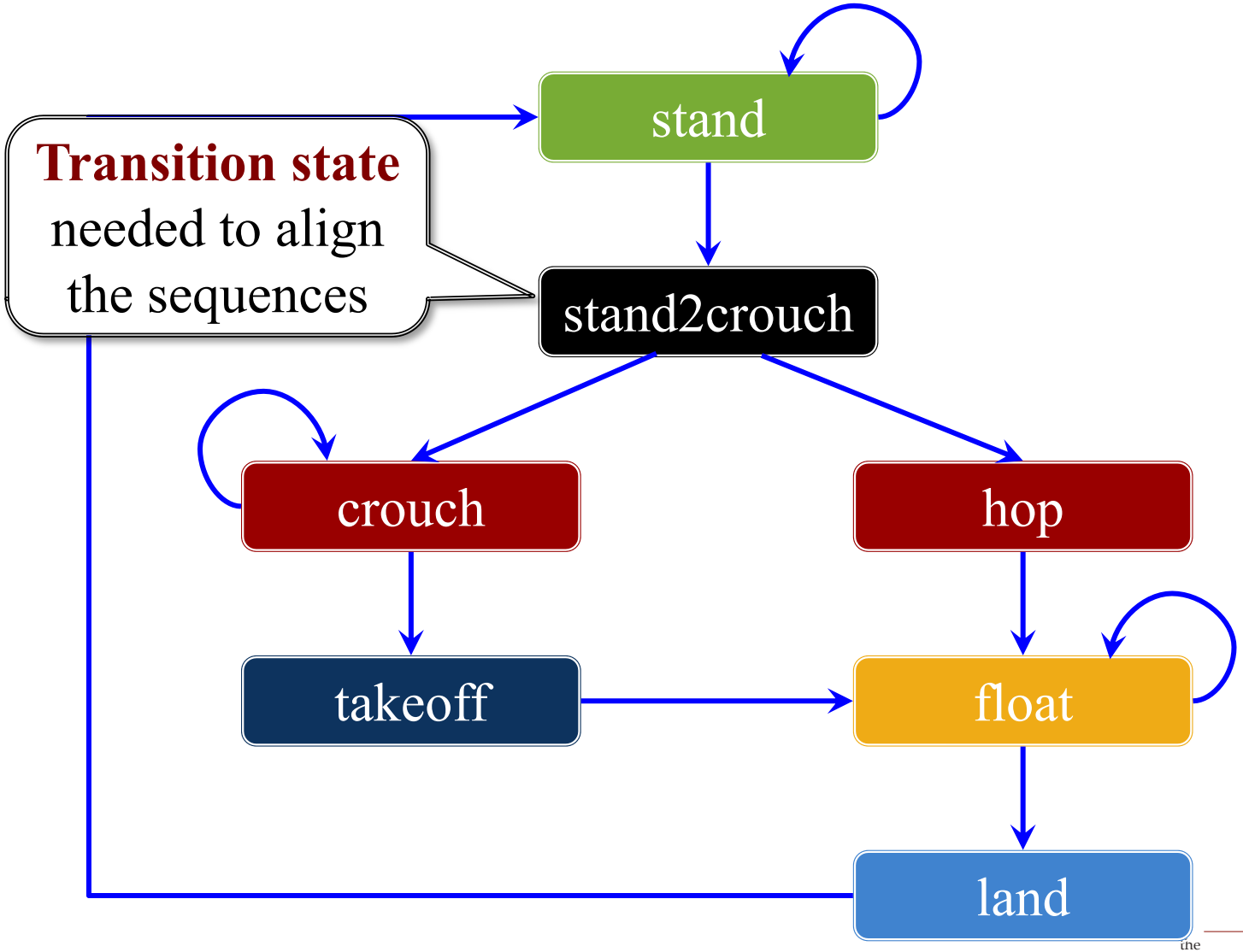
# Complex Example: Jumping



# Complex Example: Jumping



# Complex Example: Jumping



# LibGDX Interfaces

## StateMachine<E>

- Attached to an entity
  - Set the entity in constructor
  - New entity, new state machine
- Must implement methods
  - `update()`
  - `changeState(State<A> state)`
  - `revertToPreviousState()`
  - `getCurrentState()`
  - `isInState(State<A> state)`
- `DefaultStateMachine` provided

## State<E>

- Not attached to an entity
  - `StateMachine` sets state
  - `StateMachine` passes entity
- Must implement methods
  - `enter(E entity)`
    When machine enters state
  - `exit(E entity)`
    When machine exits state
  - `update(E entity)`
    When machine stays in state

# LibGDX Interfaces

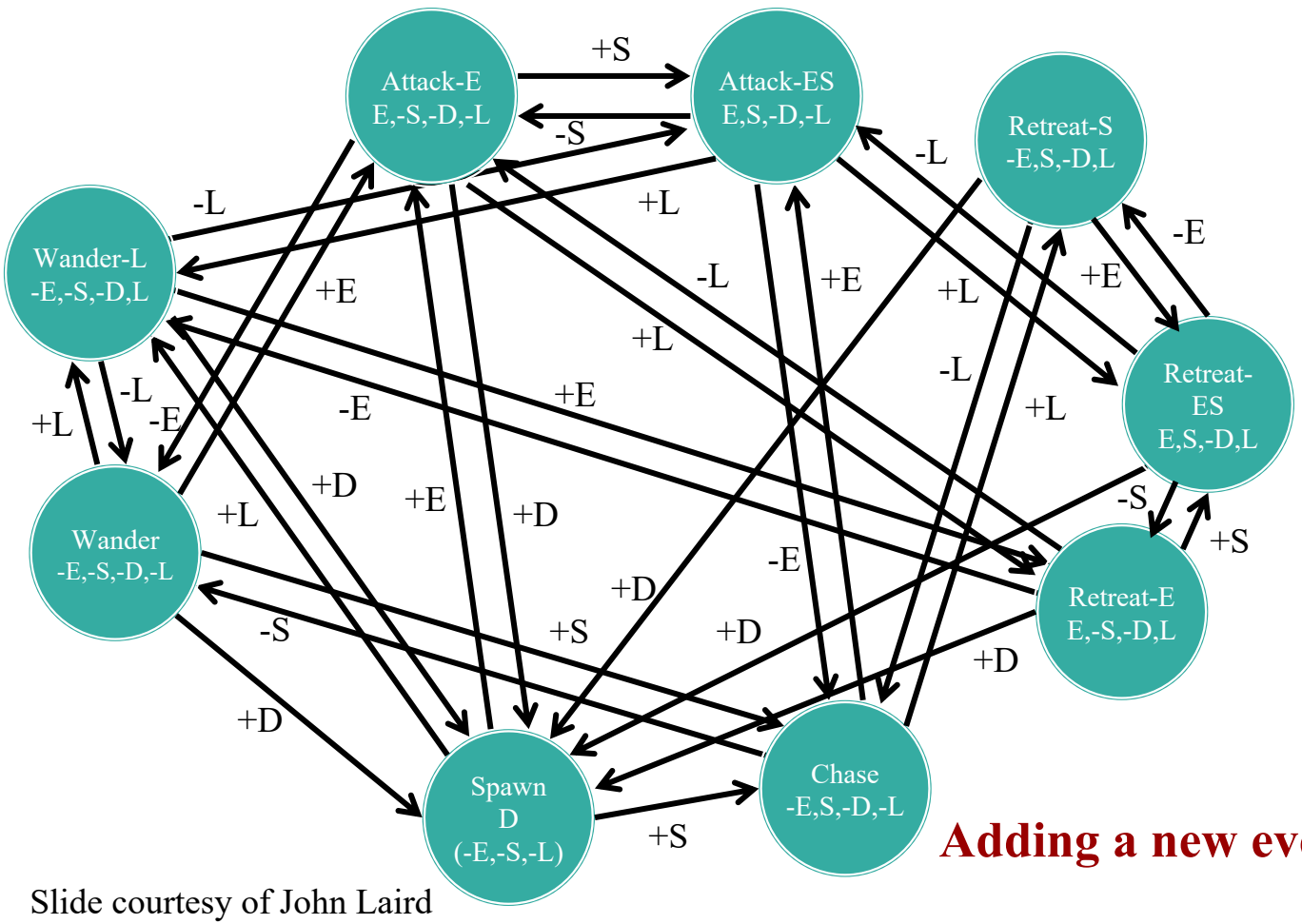
## StateMachine<E>

- Attached to an entity
- Updates current state. Does not transition!
- Transition logic external to the state machine.
- Must implement methods
  - update()
  - changeState(State<A> state)
  - revertToPreviousState()
  - getCurrentState()
  - isInState(State<A> state)
- DefaultStateMachine provided

## State<E>

- Not attached to an entity
  - StateMachine sets state
  - StateMachine passes entity
- Must implement methods
  - enter(E entity)
    When machine enters state
  - exit(E entity)
    When machine exits state
  - update(E entity)
    When machine stays in state

# Problems with FSMs



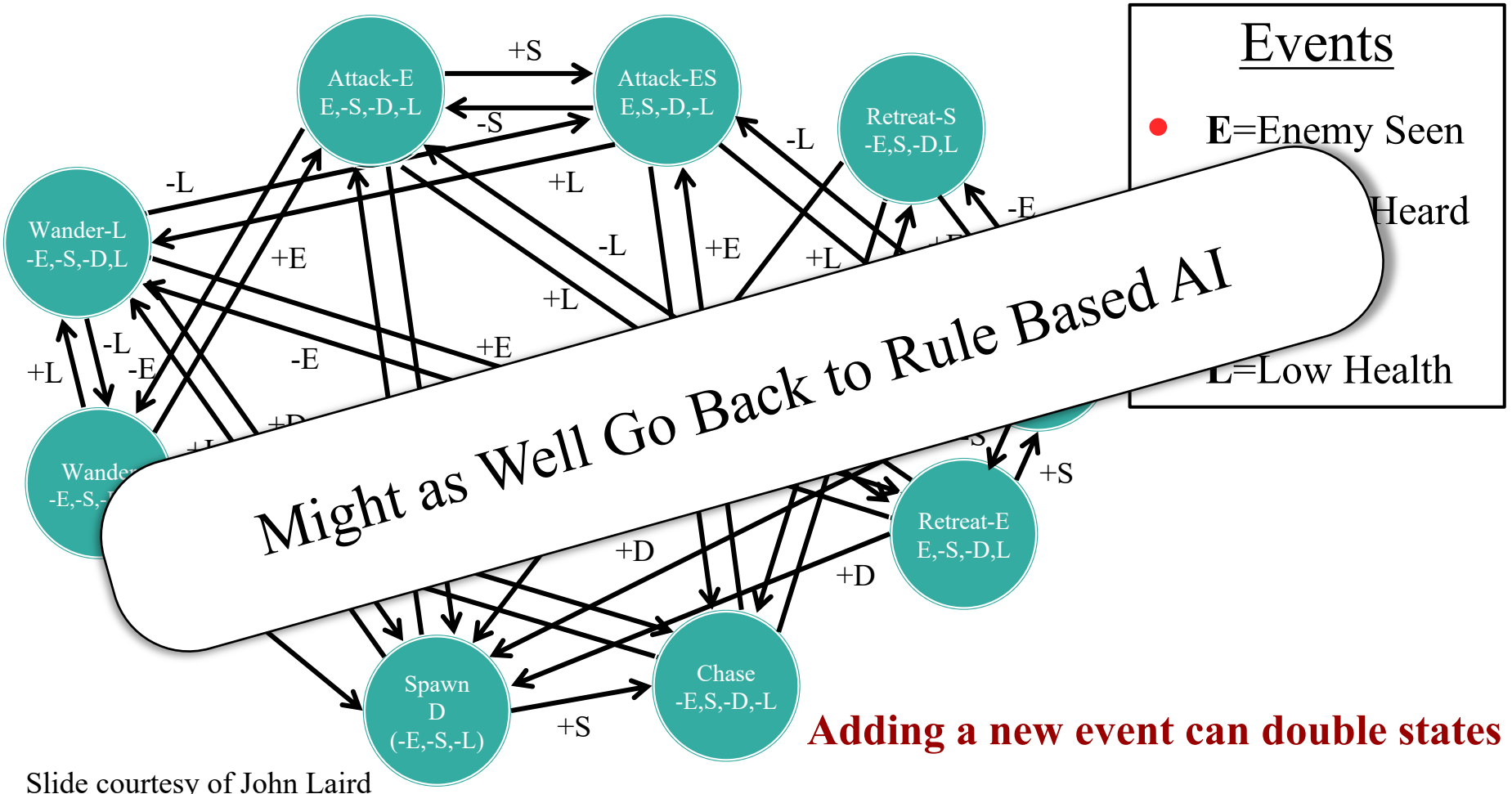

## Events

- **E**=Enemy Seen
- **S**=Sound Heard
- **D**=Die
- **L**=Low Health

**Adding a new event can double states**

Slide courtesy of John Laird

# Problems with FSMs




**Adding a new event can double states**

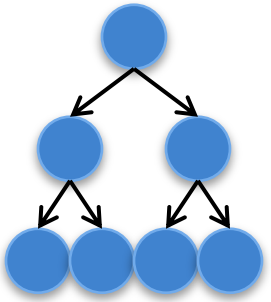
Slide courtesy of John Laird

# An Observation

- Each state has a set of **global attributes**
  - Different attributes may have same actions
  - Reason for redundant behavior
- Example just cared about attributes
  - Not really using the full power of a FSM
  - Why don't we just check attributes directly?
- Attribute-based selection: *decision trees*

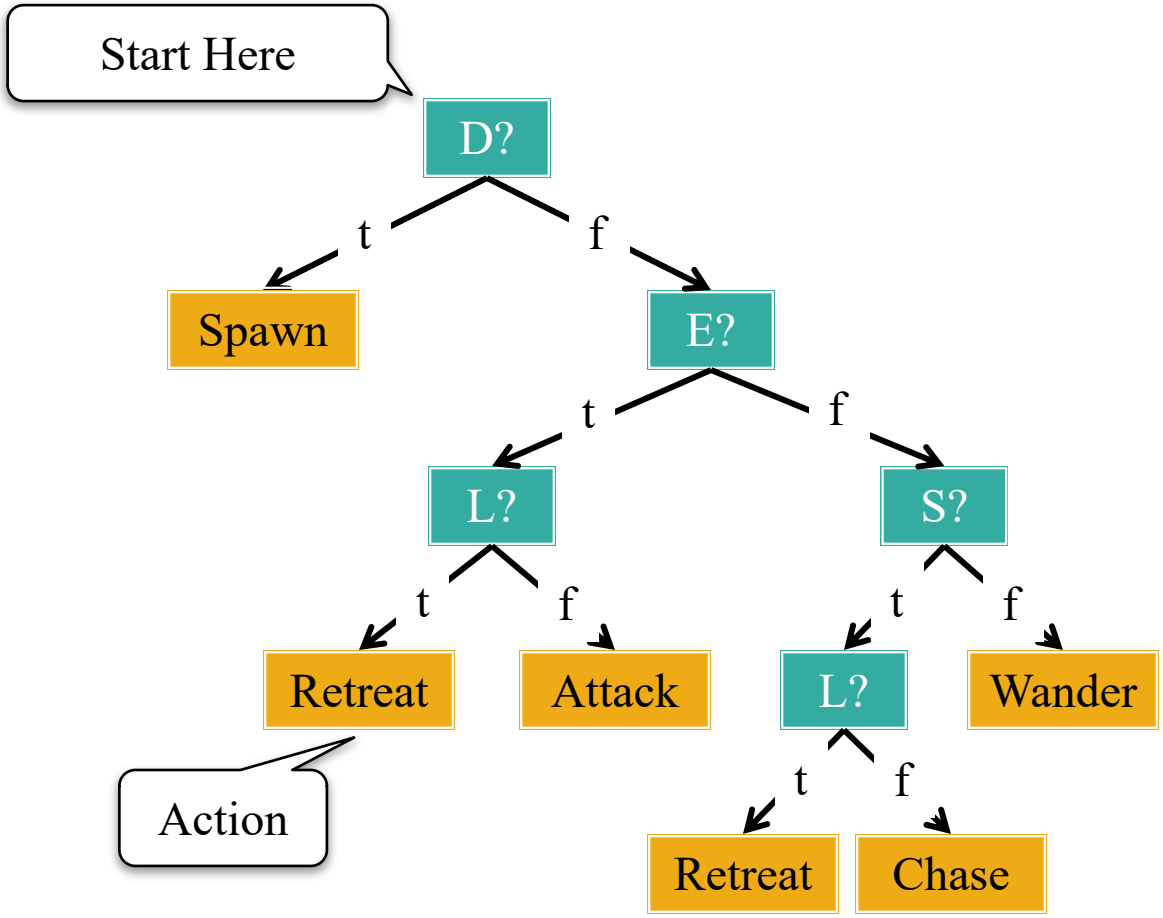
# Decision Trees

- Thinking **encoded as a tree**
  - Attributes = tree nodes
  - Left = true, right = false
  - Actions = leaves (reach from the root)



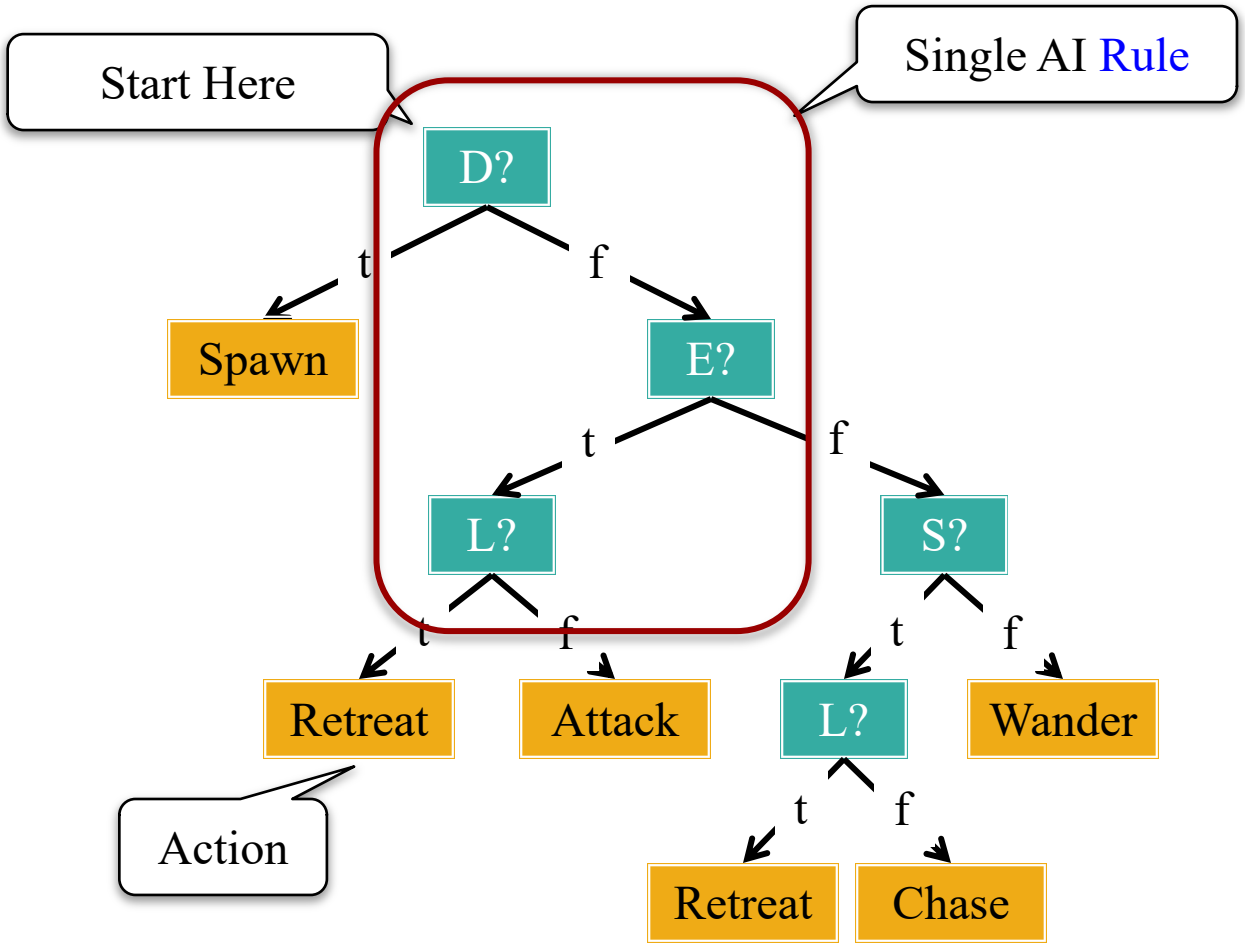
- Classify by **descending** from root to a leaf
  - Start with the test at the root
  - Descend the branch according to the test
  - Repeat until a leaf is reached

# Decision Tree Example



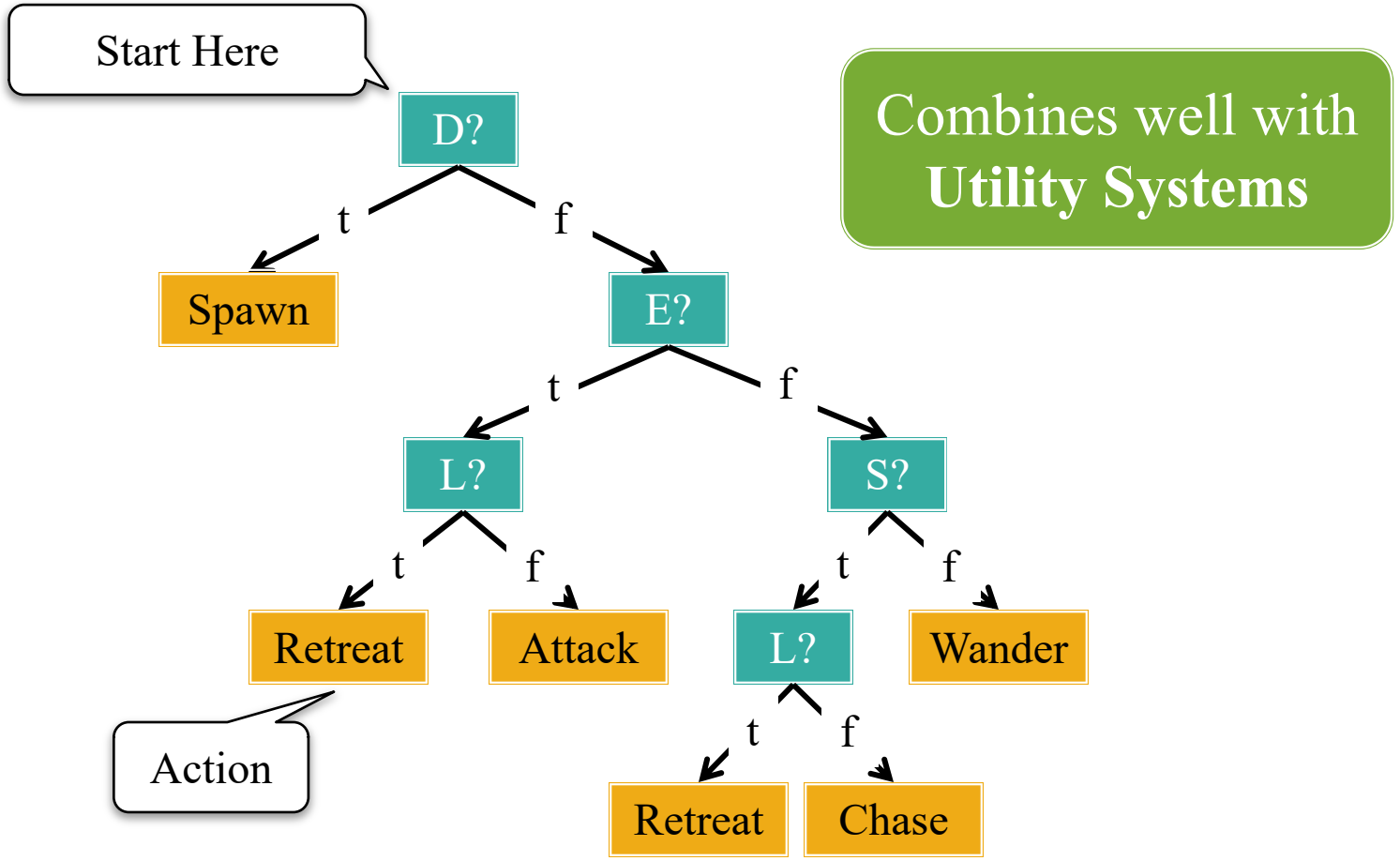

Slide courtesy of John Laird

# Decision Tree Example




Slide courtesy of John Laird

# Decision Tree Example




Slide courtesy of John Laird

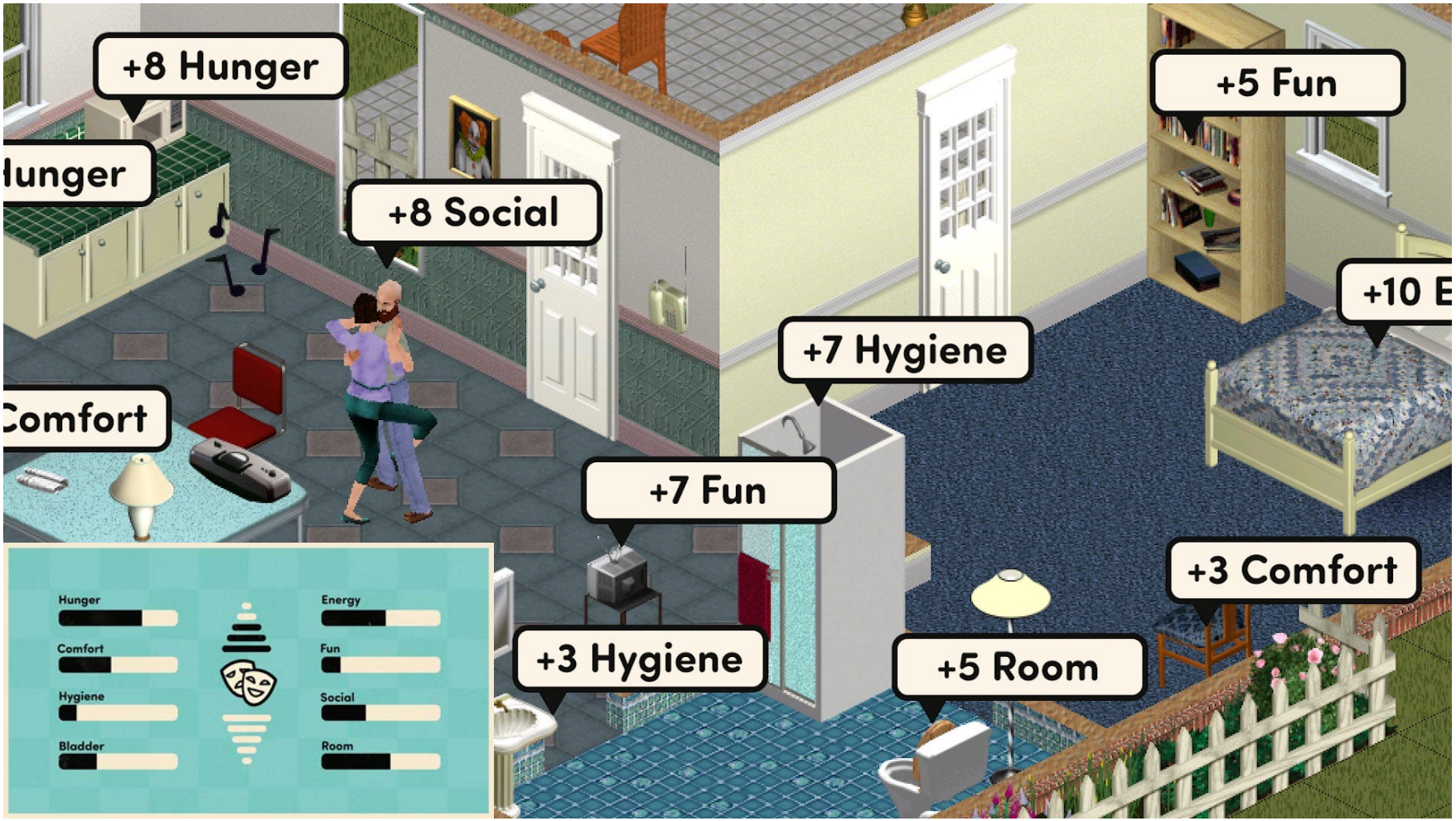
# Managing Rule-Based Behavior

- **Finite State Machines**
  - Group the rules into states (the Kodu example)
  - Effectively what you did in Lab 3
- **Utility Systems**
  - Technique use in the *Sims* games
  - Powerful tool with a lot of emergent behavior
- **Behavior Trees**
  - Popularized by Unreal and Unity
  - Considered the modern standard for AI

# Structure of a Utility System

- Fundamental concept: **a scoring system**
  - Each NPC has a list of possible tasks
  - We give each task a numerical score
  - We pick the task with the highest score
- What types of things go into scoring?
  - How important the NPC feels the task is
  - How easy it is for the NPC to carry out the task
  - Based on desired probabilities/expected values
- Requires custom software

# Utility Systems in the *Sims*




https://gmtk.substack.com/p/the-genius-ai-behind-the-sims

# Structure of a Utility System

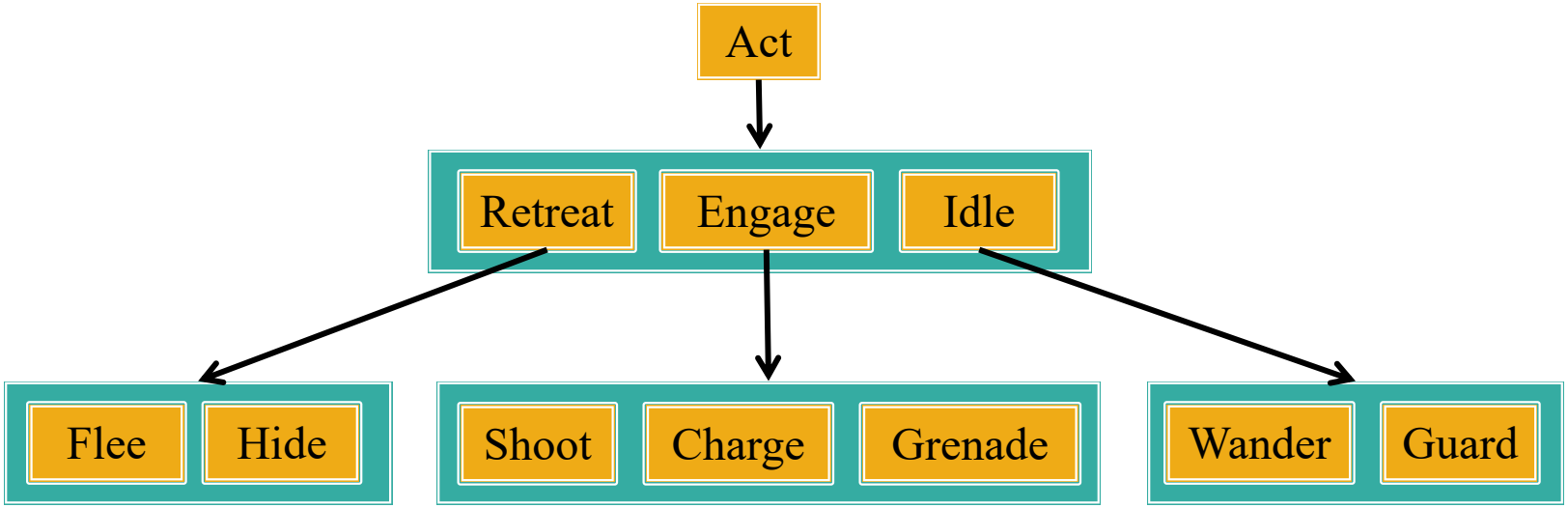
- Fundamental concept: **a scoring system**
  - Each NPC has a list of possible tasks
  - We give each task a numerical score
  - We pick the task with the hig
- What typ
  - How imp
  - How easy it is for the NPC to carry out the task
  - Based on desired probabilities/expected values
- **Requires custom software**

No LibGDX tools.
See today's reading.

# Managing Rule-Based Behavior

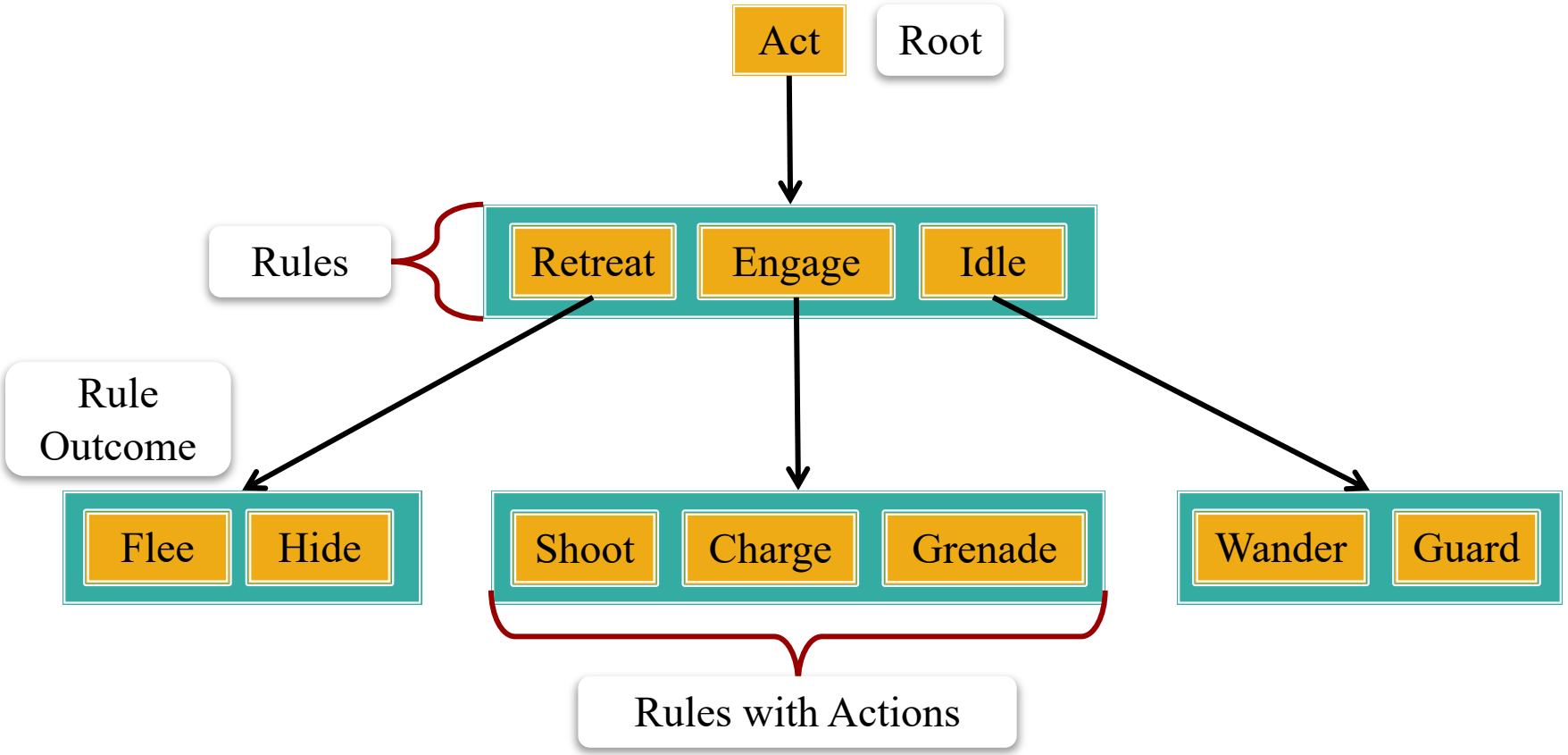
- **Finite State Machines**
  - Group the rules into states (the Kodu example)
  - Effectively what you did in Lab 3
- **Utility Systems**
  - Technique use in the *Sims* games
  - Powerful tool with a lot of emergent behavior
- **Behavior Trees**
  - Popularized by Unreal and Unity
  - Considered the modern standard for AI

# Behavior Trees (*Halo* Version)

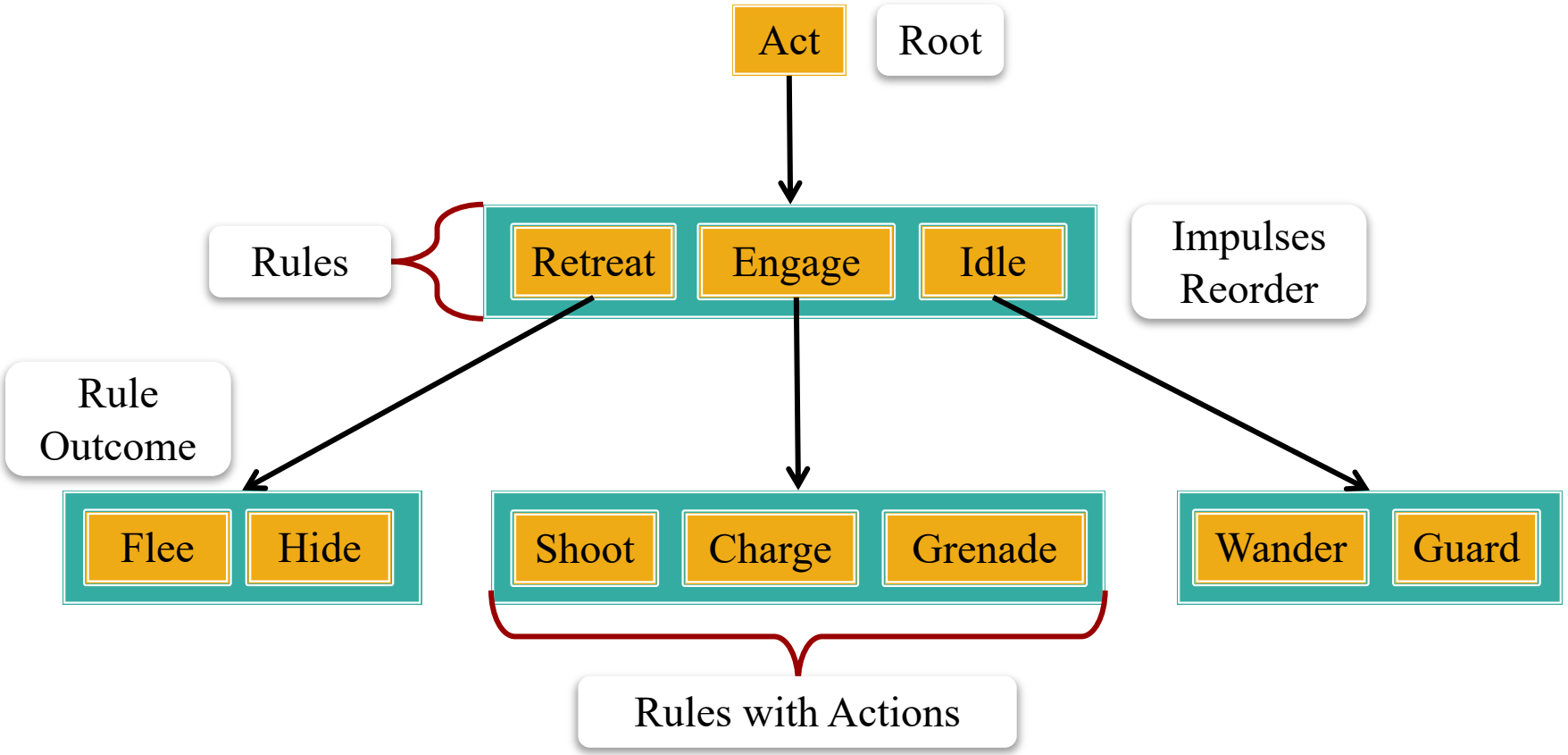



- Node is a list of *actions*
- Select action using *rules*
- Action leads to *subactions*
- Popularized by *Halo 3*
- Each node is rule-based
- Used **impulses** to reorder

# Behavior Trees (*Halo* Version)



# Behavior Trees (*Halo* Version)

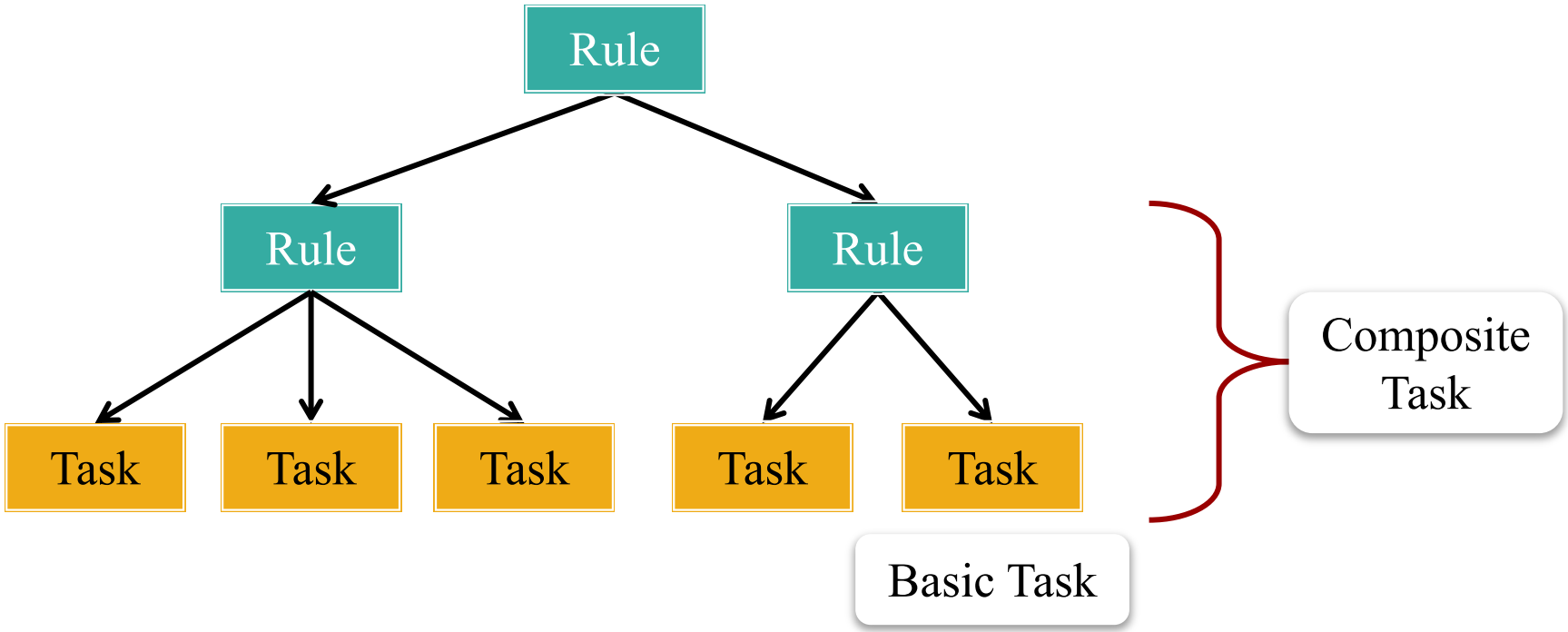


# Modern Behavior Trees

- **Design Goal:** Actions need to have **duration**
  - No longer the same as pressing a button
  - Need to include steps/animations of the action
  - Long-running actions must be **interruptable**
- **Design Goal:** Make **accessible** to designers
  - Allow programmers to specify **leaf actions**
  - Complex actions are **combination** of leaf actions
  - Tree becomes a **visual programming language**
- Modern form popularized by *Unreal*

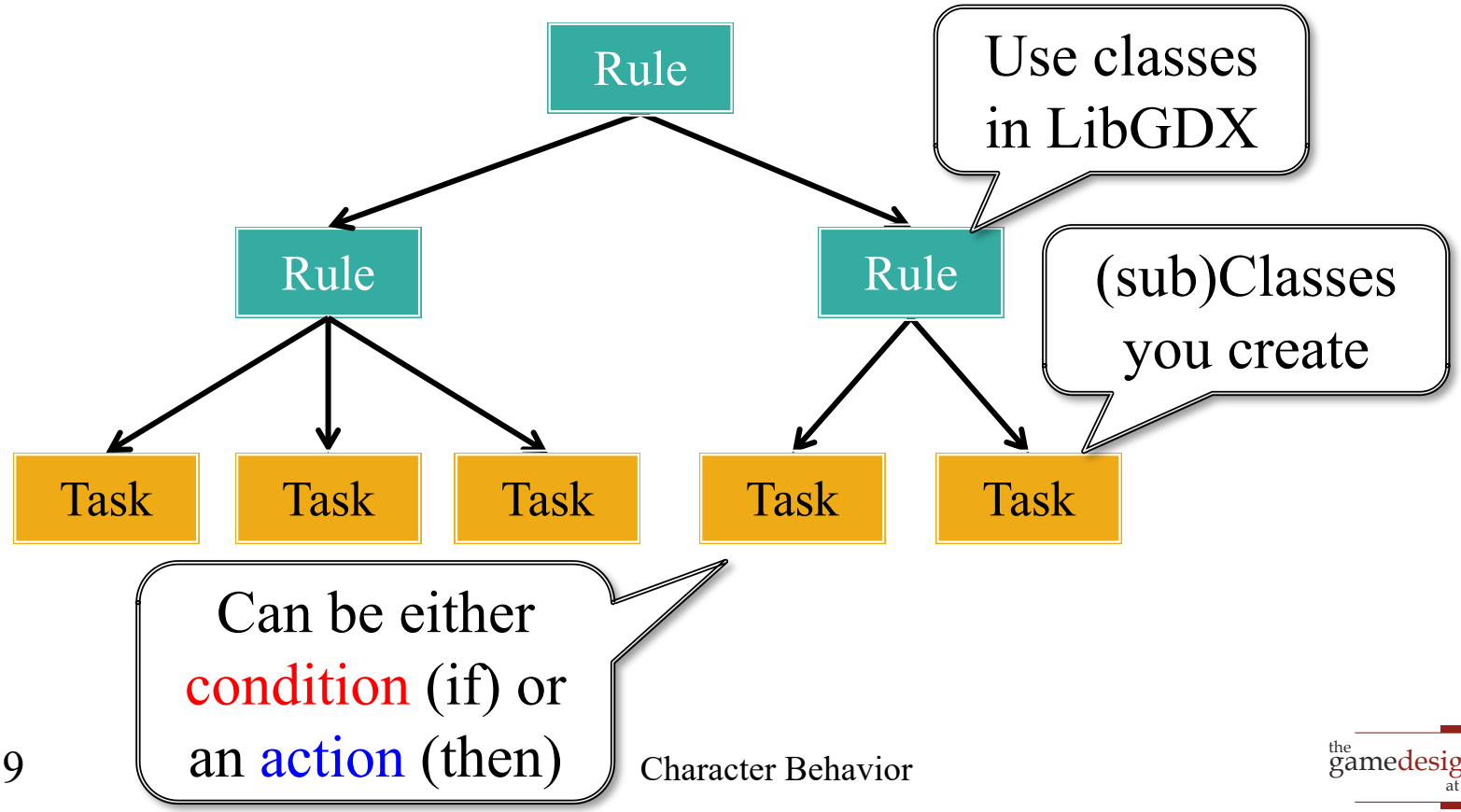
# LibGDX Uses the Unreal Model

- Base actions are defined at the leaves
- Internal nodes to **select** or even **combine** tasks



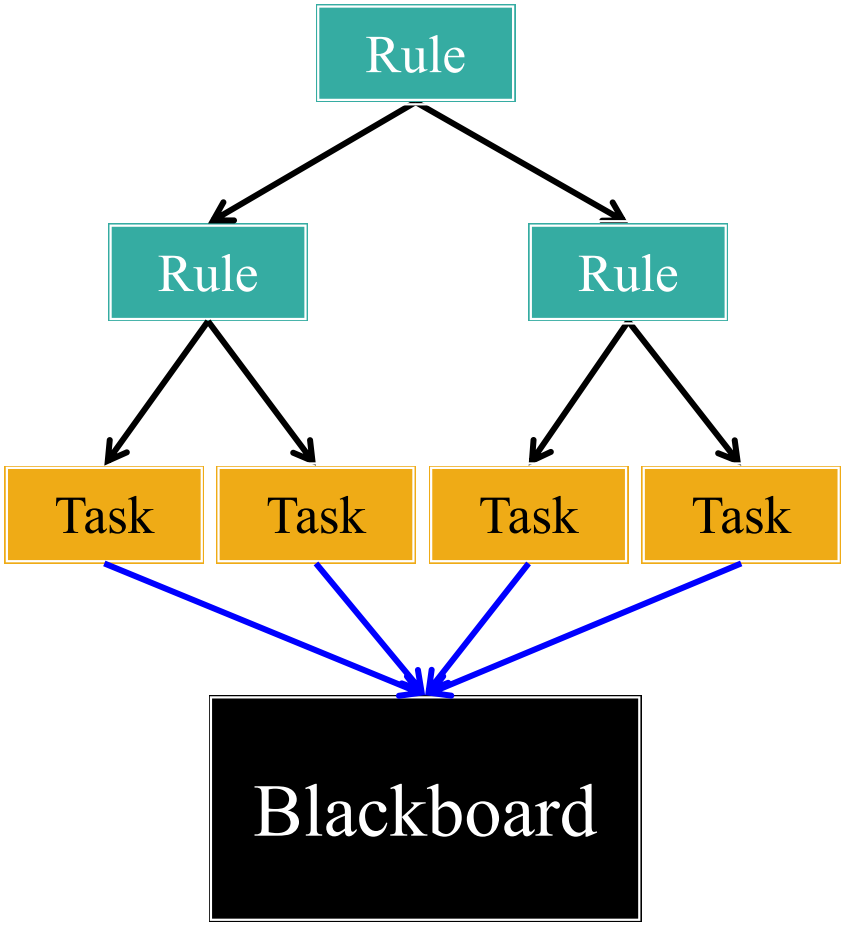
# LibGDX Behavior Trees

- Base actions are defined at the leaves
- Internal nodes to **select** or even **combine** tasks



# The Blackboard Object

- Tree only specifies behavior
  - Can be applied to any NPC
  - May want to reuse behavior
- Tree is attached to an object
  - Called the **blackboard**
  - Object is target of the tasks
  - All task nodes have access
- What is the blackboard?
  - Could just be the NPC
  - Or a portion of **game state**
  - Whatever data the task needs



# LibGDX Tasks

## LeafTask<E>

- `void start()`
  - When task begins
  - Can initialize blackboard
- `Status execute()`
  - Called at each step of task
  - Returns the new `Status`
- `void end()`
  - When complete/cancelled
  - Can "clean up" blackboard

## Status

- `FRESH`
  The task has never run/reset
- `RUNNING`
  The task is still ongoing
- `FAILED`
  The task has failed
- `SUCCEED`
  The task has succeeded
- `CANCELLED`
  The task was cancelled

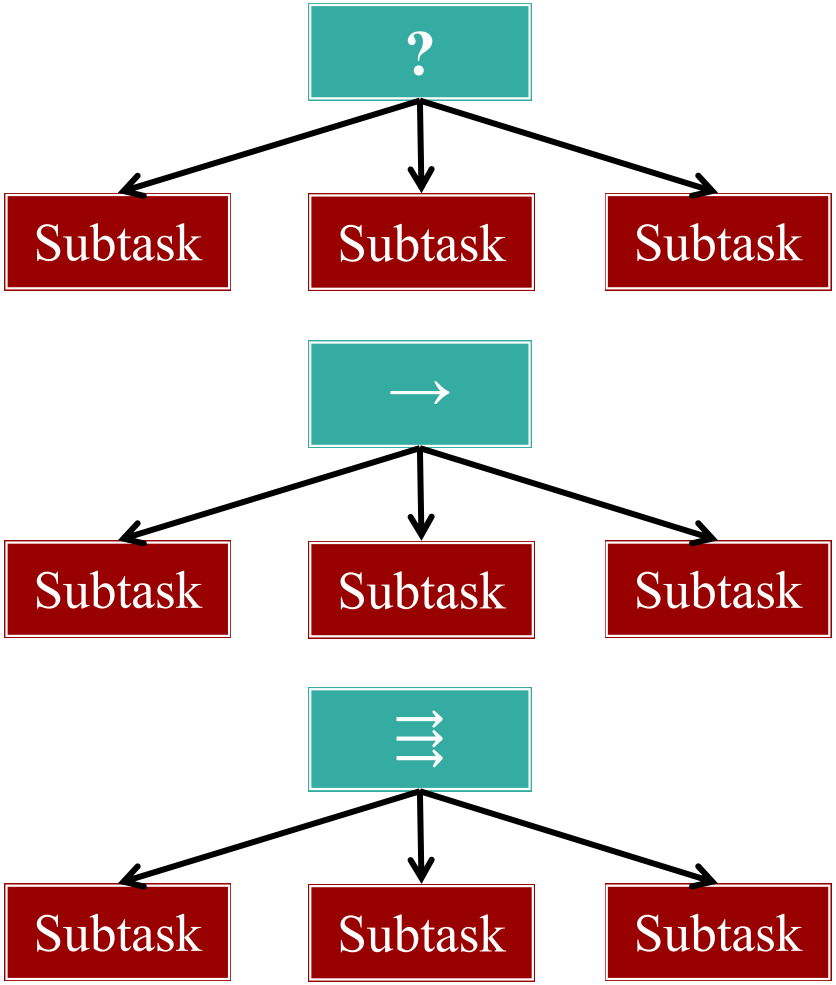
# LibGDX Tasks

## LeafTask<E>

- void start()
  - When task begins
  - Can initialize blackboard
- Status execute()
  - Called at each step of task
  - Returns the new Status
- void end()
  - When complete/cancelled
  - Can "clean up" blackboard

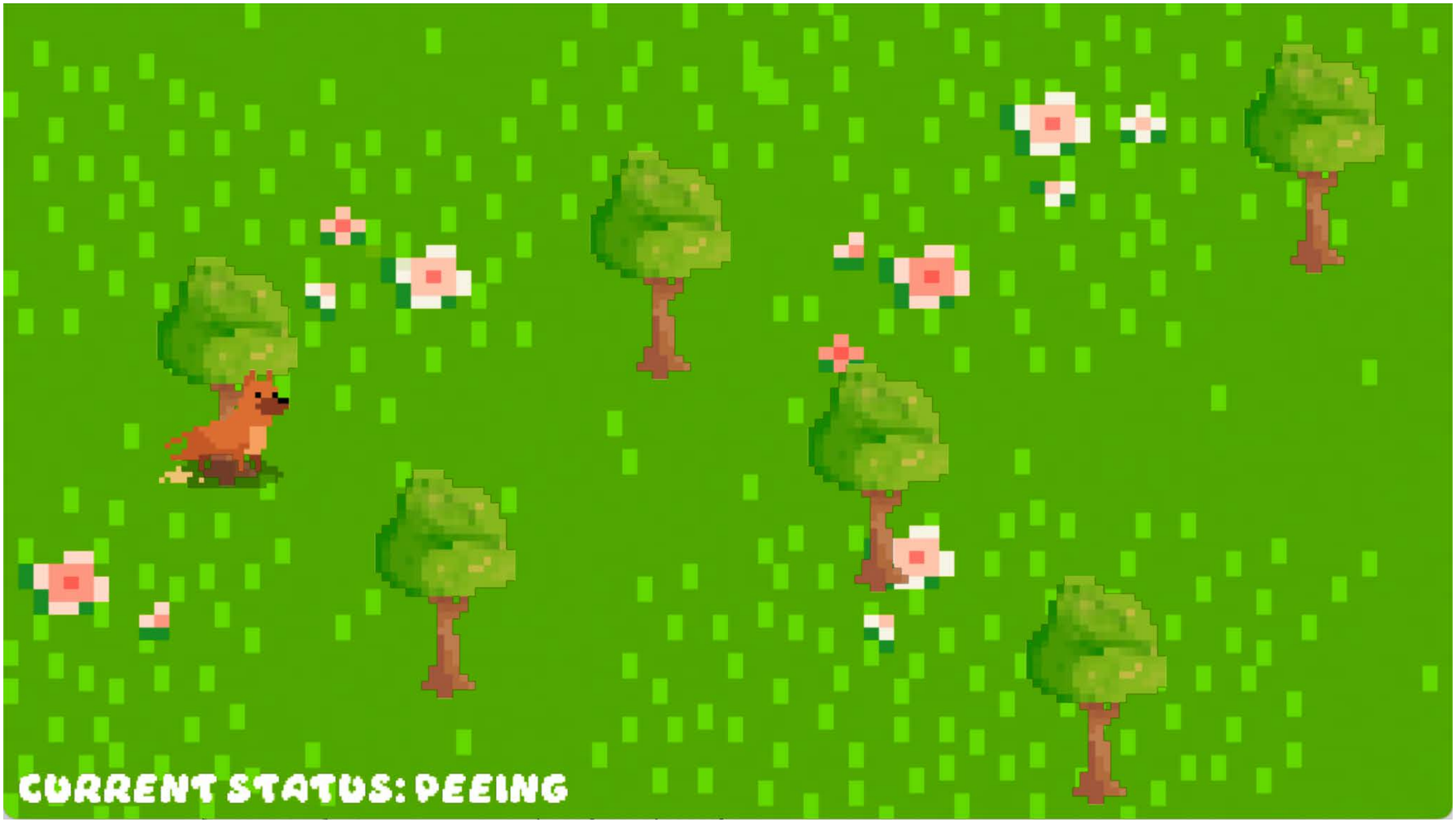
- Use execute to manage Task
  - Update the blackboard
  - Determine the new Status
  - Return new Status for rules
- Select Task based on Status
  - RUNNING ~ SUCCESS
  - CANCELLED ~ FAILURE
- But remember architecture
  - Task should **choose** action
  - It should not **perform** it!

# Making Complex Actions

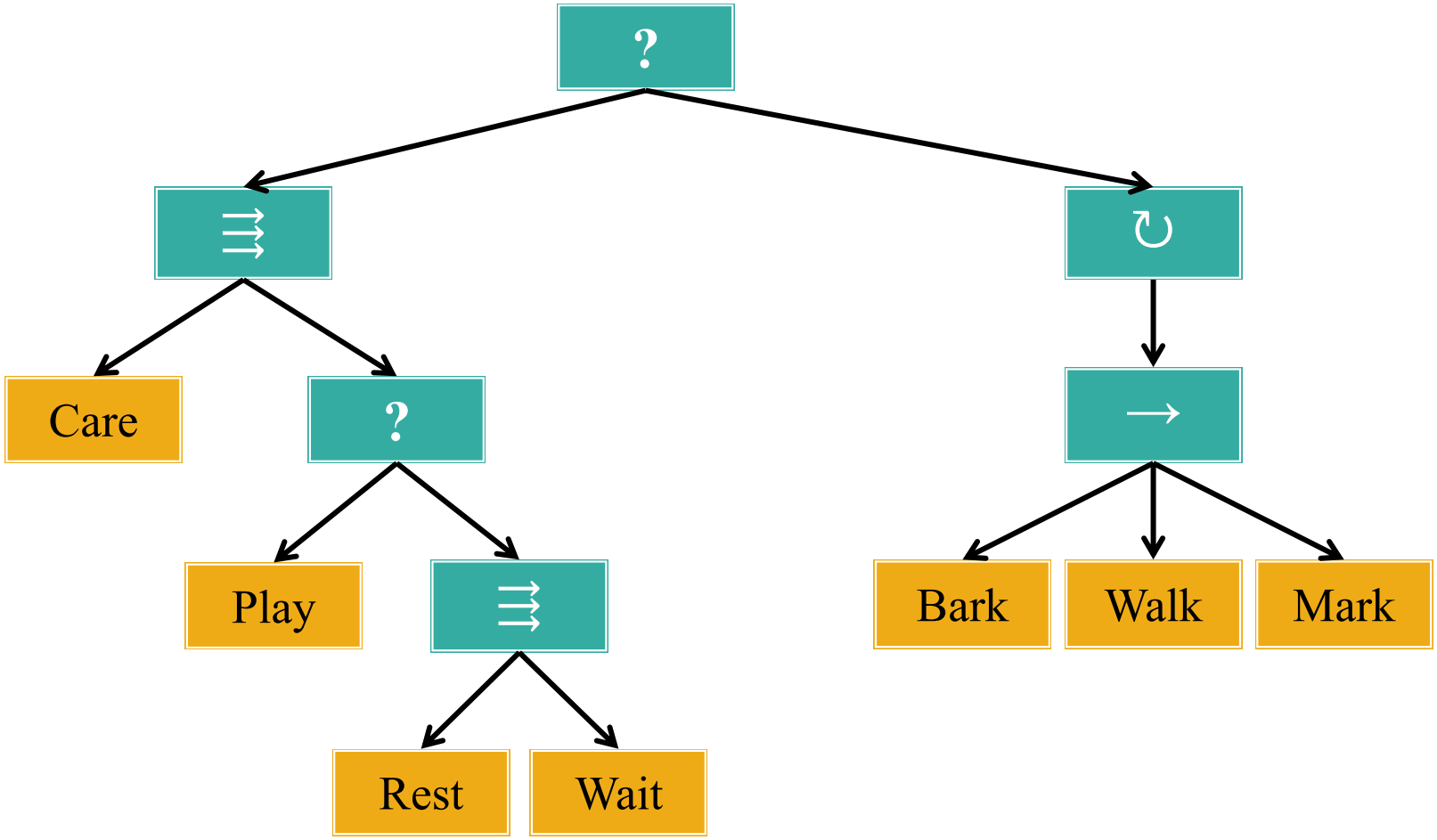



- **Selector** rules
  - Tasks are tried in order
  - Chooses first one to not fail
  - `SUCCESS` only requires **one**
- **Sequence** rules
  - Tasks are tried in order
  - `SUCCESS` needs **all** to succeed
  - `RUNNING` until last task done
- **Parallel** rules
  - Tasks are tried simultaneously
  - `SUCCESS` needs **all** to succeed
  - `RUNNING` if just one task is

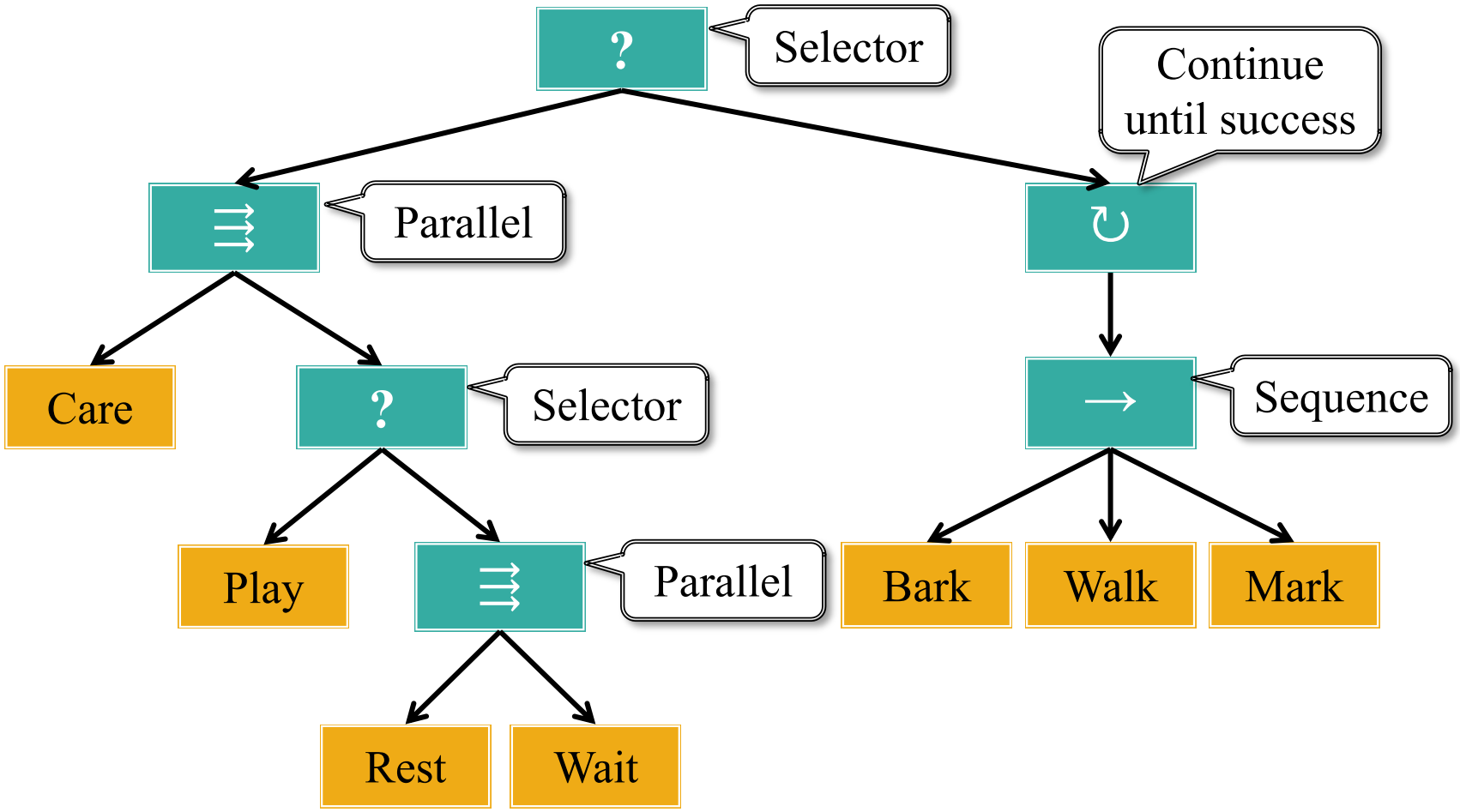
# Dog Park Example



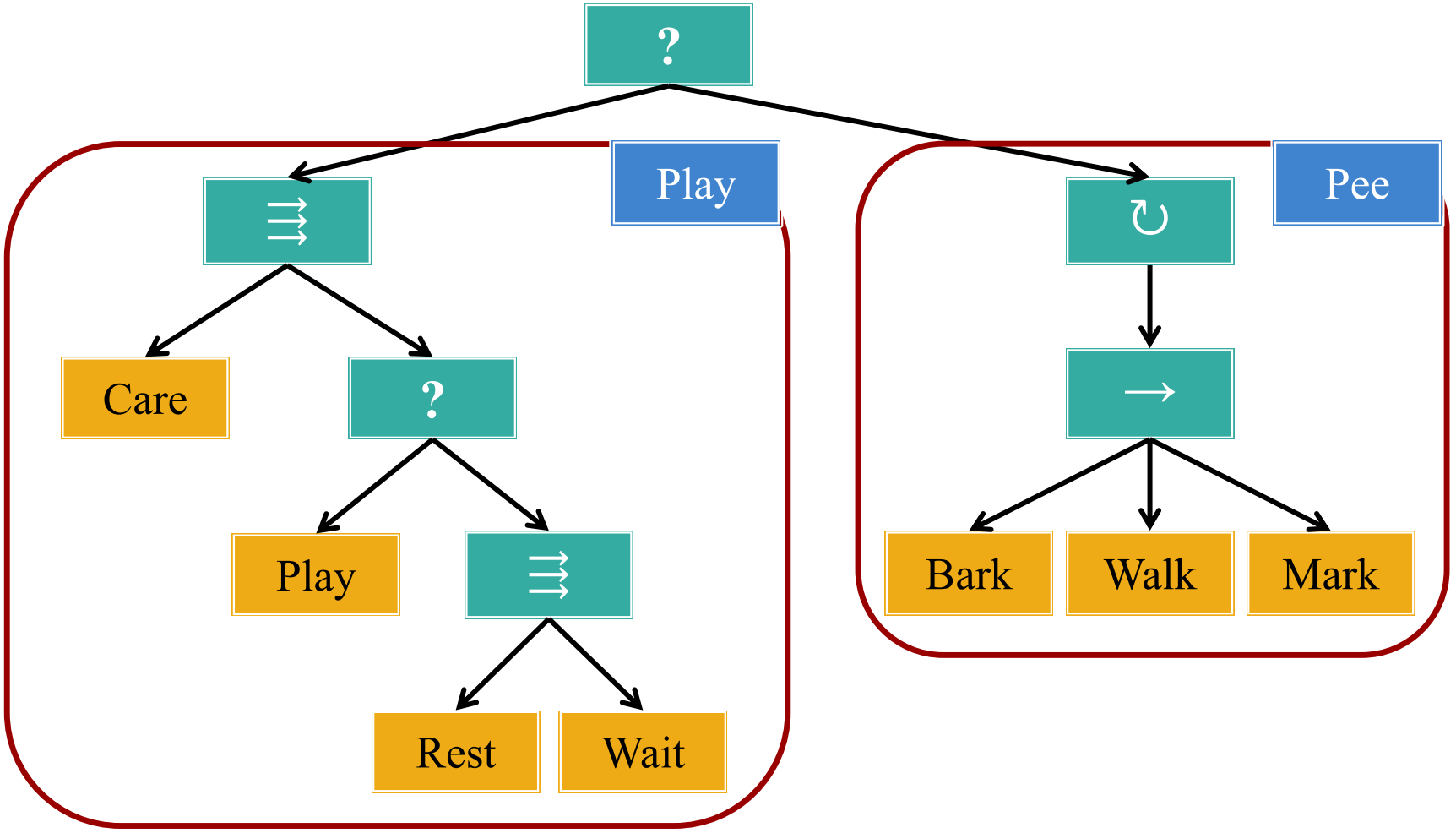
# Dog Park Example



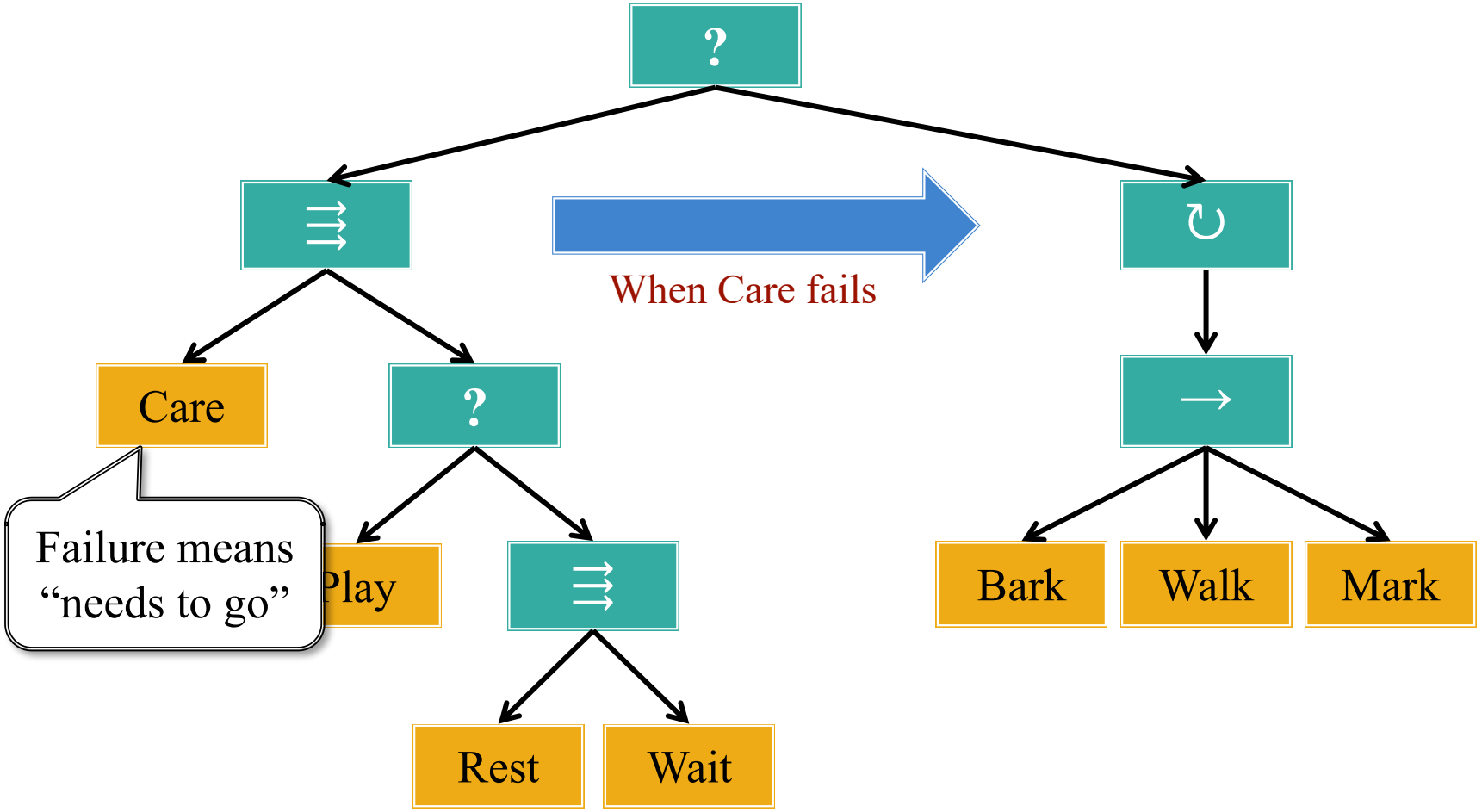
# Dog Park Example



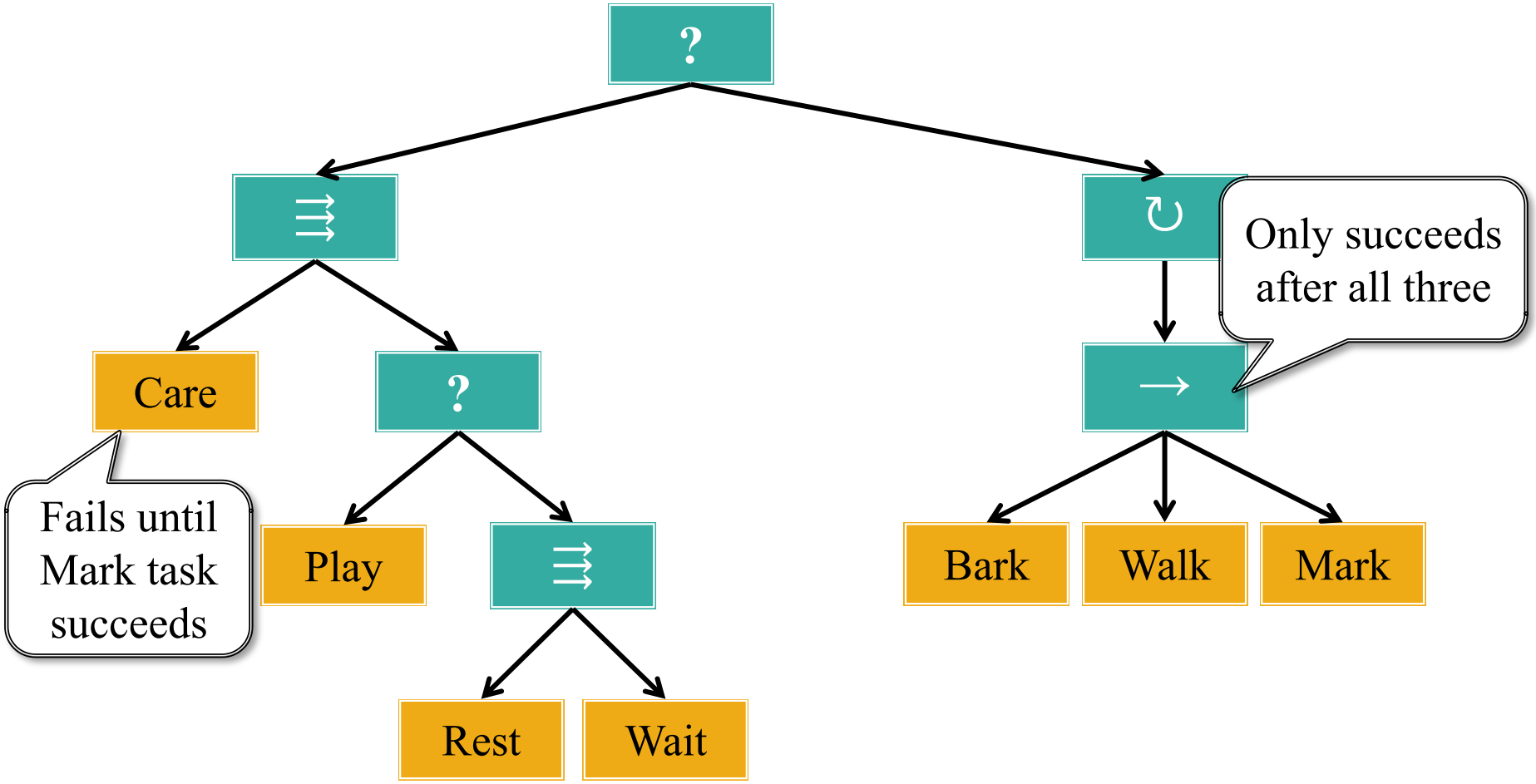
# Dog Park Example



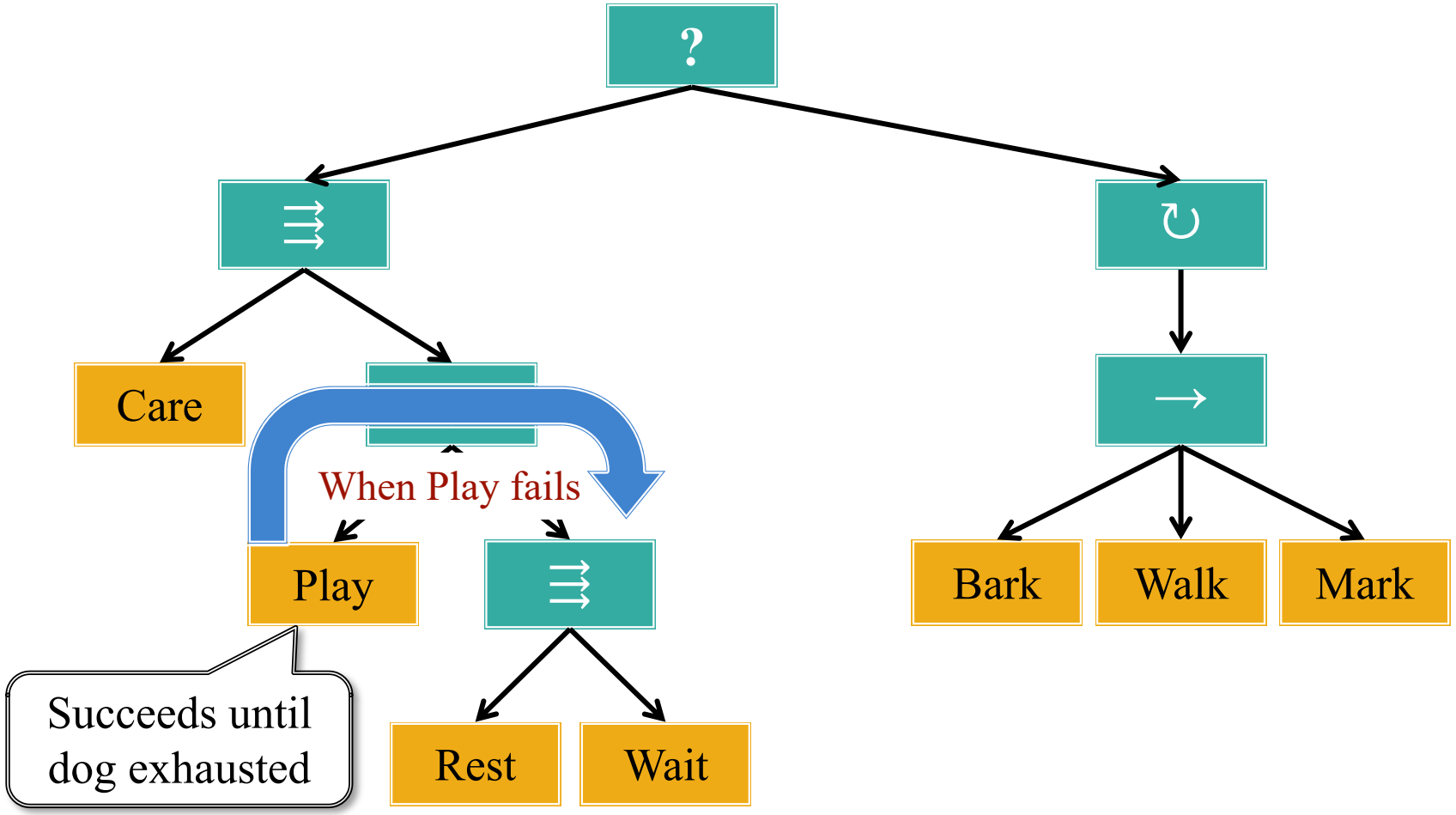
# Dog Park Example



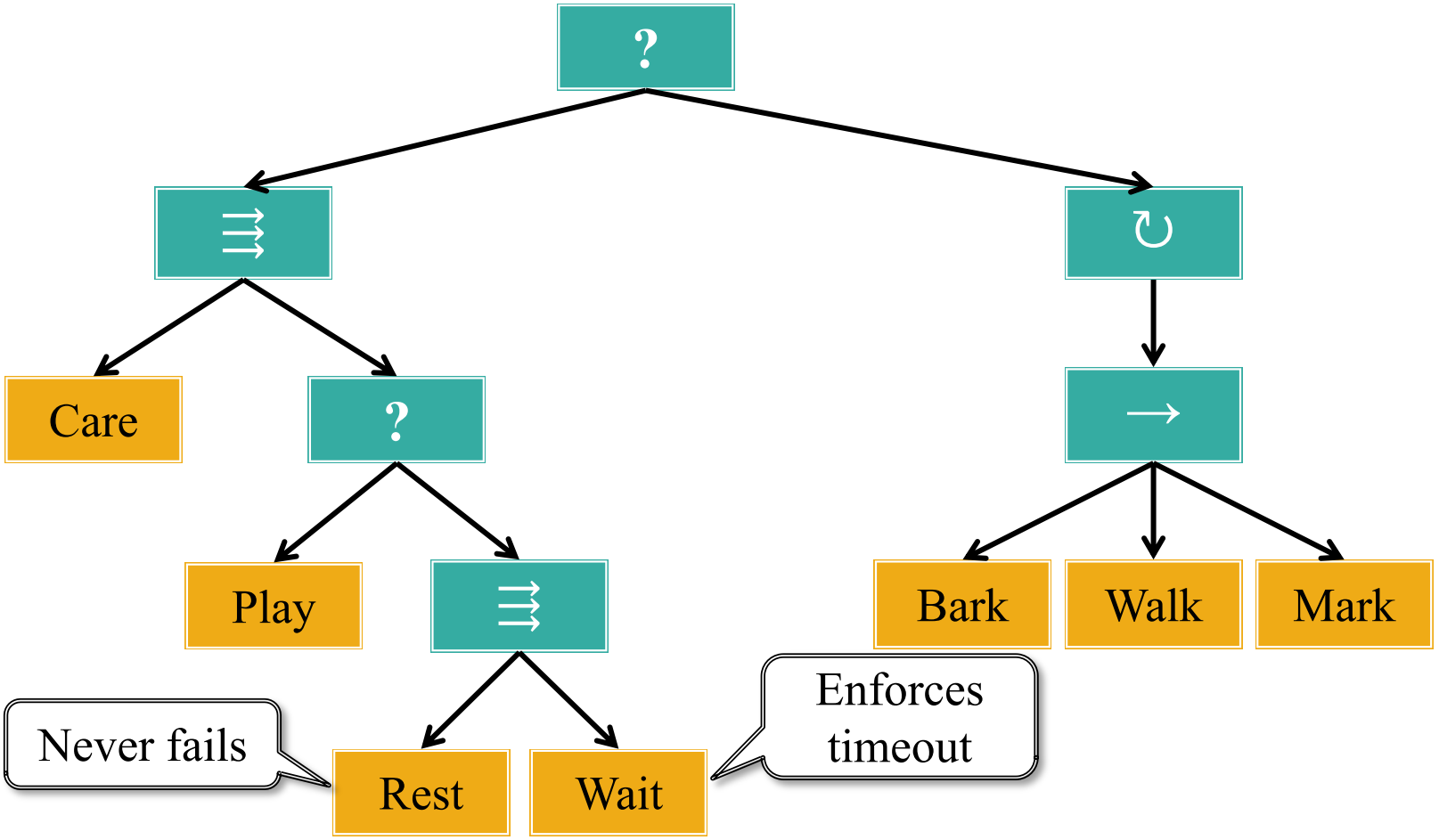
# Dog Park Example



# Dog Park Example



# Dog Park Example



# Behavior Scripts

```
root
    selector                          # Pick first to succeed
        parallel                      # Make sure both succeed
            care urgentProb:0.05      # Check if we need to pee
            selector                  # Pick first to succeed
                play                  # Run until exhausted
                parallel policy:"selector"   # Sleep with timeout
                    wait seconds:"triangular,2.5,5.5"
                    rest
        untilSuccess                  # Finish sequence to completion
            sequence                  # Do these in order
                bark times:"uniform,1,2"     # Bark a few times
                walk                  # Walk to a tree
                mark                  # Mark the tree
```

# Behavior Scripts

```
root
   selector                          # Pick first to succeed
      parallel                          # Make sure both succeed
         care urgentProb:0.05           # Check if we need to pee
         selector                       # Pick first to succeed
            play                                    xhausted
            parallel p                              imeout
               wait se
               rest
      untilSuccess                   # Finish sequence to completion
         sequence                    # Do these in order
            bark times:"uniform,1,2"    # Bark a few times
            walk                        # Walk to a tree
            mark                        # Mark the tree
```

Examine demo for ideas

# Summary

- Character AI is a **software engineering** problem
  - Sense-think-act aids code reuse and ease of design
  - Least standardized aspect of game architecture
- **Rule-based AI** is foundation for all character AI
  - Simplified variation of sense-think-act
  - But managing rules can be difficult
- There are three modern techiques for character AI
  - **Finite State Machines** (Lab 2)
  - **Utility Systems** (*Sims*, simulation games)
  - **Behavior Trees** (Most modern games)