Basic Game AI

Technical Game Development II

Professor Charles Rich
Computer Science Department
rich@wpi.edu

With material from: Millington and Funge, *Artificial Intelligence for Games*, Morgan Kaufmann 2009. (Chapter 5)

Definitions?

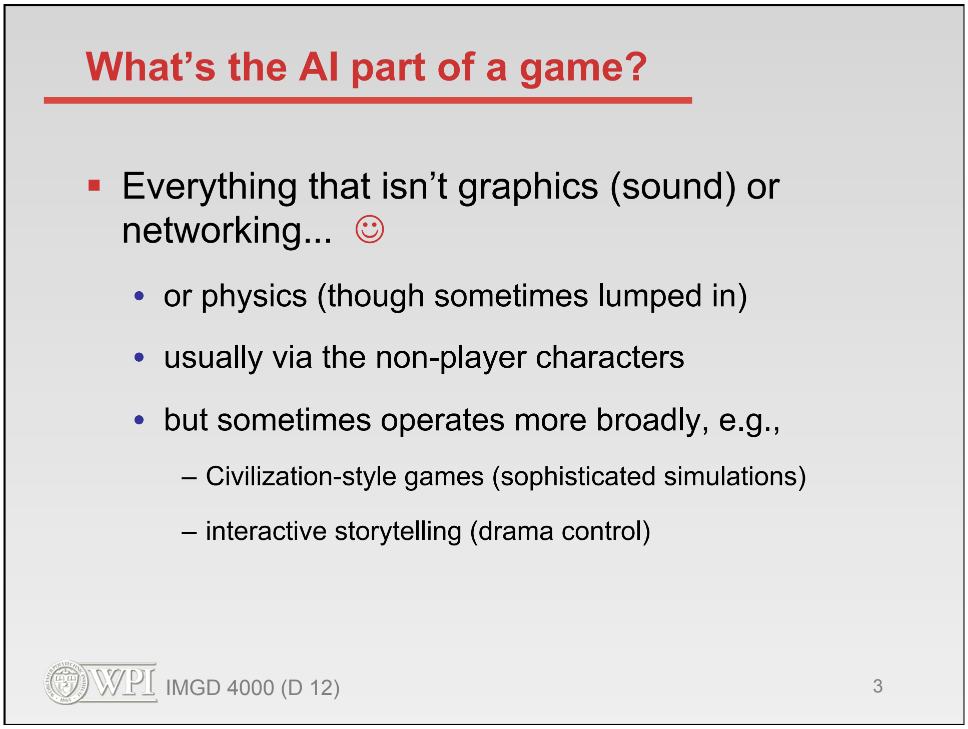
- What is artificial intelligence (AI) ?
  - subfield of computer science ?
  - subfield of cognitive science ?
- What is “AI for Games” ?
  - versus “academic AI” ?

In games, ***everything*** (including the AI) is in service of the ***player’s*** experience (“fun”)

- What does it mean for a game AI to “cheat”?

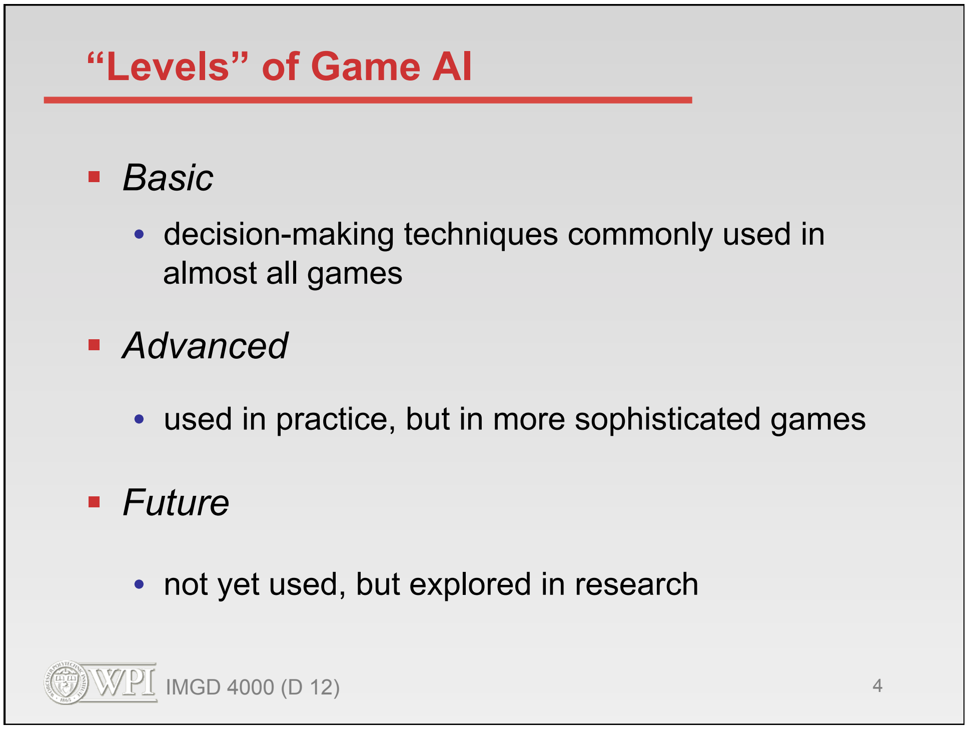
Resources: introduction to Buckland, www.gameai.com, aigamedev.com, www.aiwisdom.com, www.ai4games.org

IMGD 4100, B Term 2013



What’s the AI part of a game?

- Everything that isn’t graphics (sound) or networking... ☺
  - or physics (though sometimes lumped in)
  - usually via the non-player characters
  - but sometimes operates more broadly, e.g.,
    - Civilization-style games (sophisticated simulations)
    - interactive storytelling (drama control)



“Levels” of Game AI

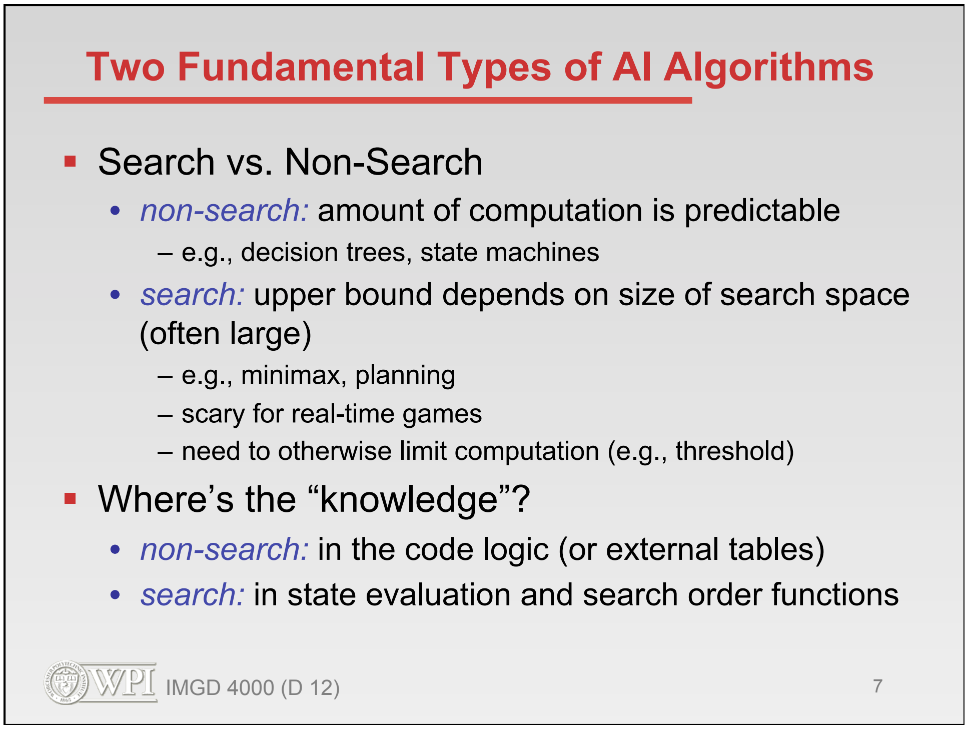
- *Basic*
  - decision-making techniques commonly used in almost all games
- *Advanced*
  - used in practice, but in more sophisticated games
- *Future*
  - not yet used, but explored in research

This course

- ***Basic*** game AI
  - decision-making techniques commonly used in almost all games
    - – basic pathfinding (A*) *(IMGD 3000)*
    - – decision trees *(today)*
    - – (hierarchical) state machines *(today)*
- ***Advanced*** game AI
  - used in practice, but in more sophisticated games
    - – advanced pathfinding *(Tues)*
    - – behavior trees (in Halo 3) *(Thurs, Fri)*

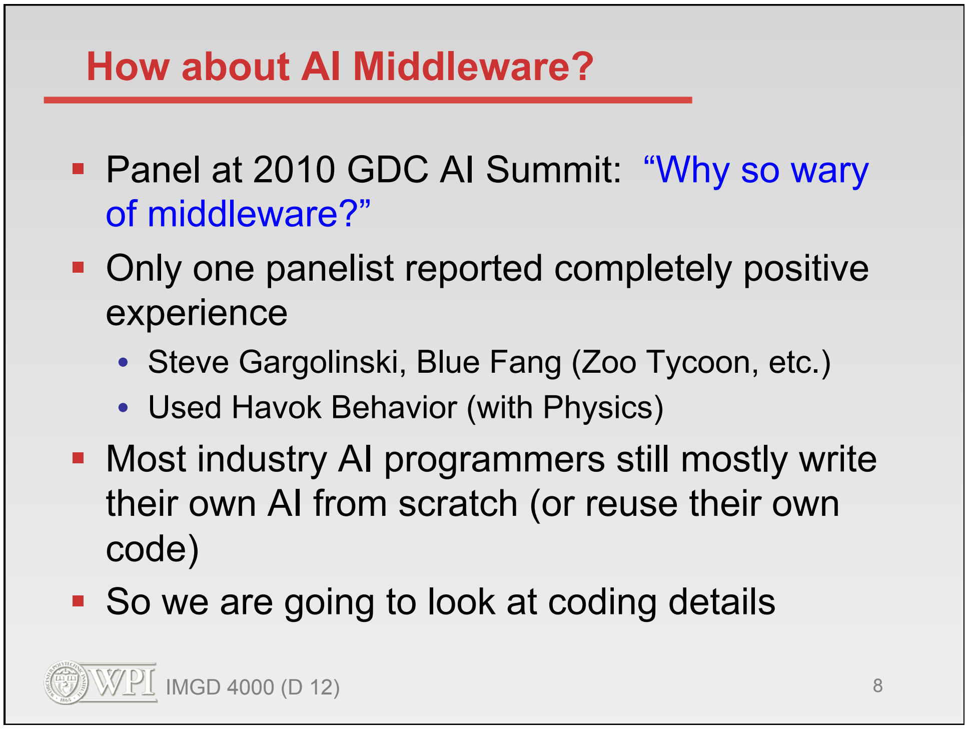
Future Game AI ?

- Take IMGD 4100 in 2013 (B) [alt yr course] “AI for Interactive Media and Games”
  - fuzzy logic
  - more goal-driven agent behavior
- Take CS 4341 “Artificial Intelligence”
  - machine learning
  - planning



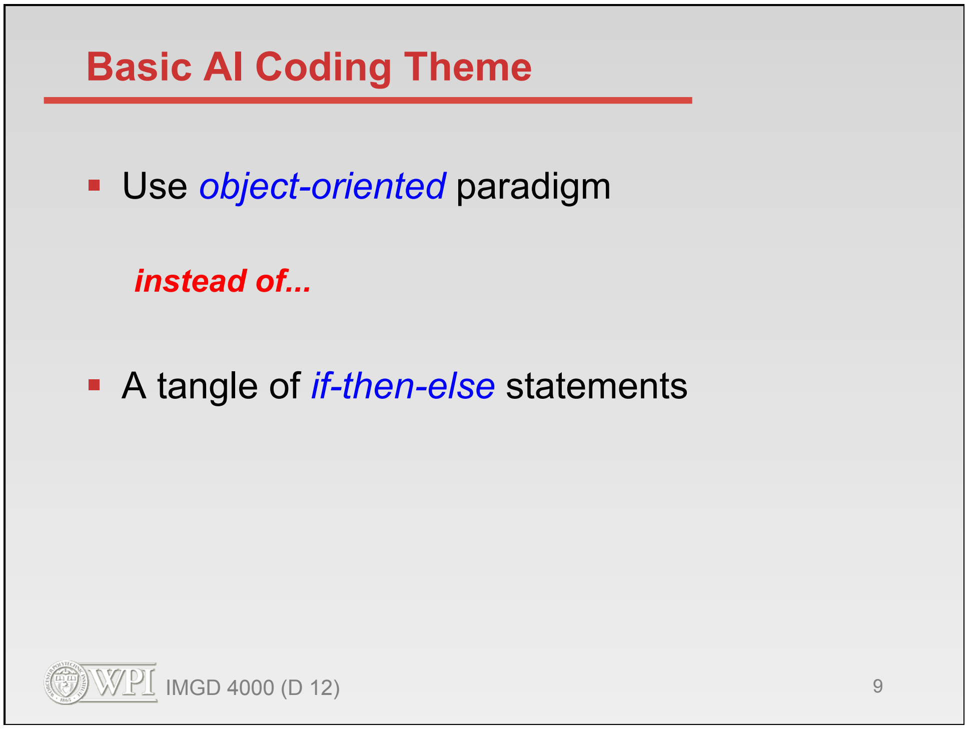
Two Fundamental Types of AI Algorithms

- Search vs. Non-Search
  - *non-search:* amount of computation is predictable
    - e.g., decision trees, state machines
  - *search:* upper bound depends on size of search space (often large)
    - e.g., minimax, planning
    - scary for real-time games
    - need to otherwise limit computation (e.g., threshold)
- Where's the "knowledge"?
  - *non-search:* in the code logic (or external tables)
  - *search:* in state evaluation and search order functions



How about AI Middleware?

- Panel at 2010 GDC AI Summit: "Why so wary of middleware?"
- Only one panelist reported completely positive experience
  - Steve Gargolinski, Blue Fang (Zoo Tycoon, etc.)
  - Used Havok Behavior (with Physics)
- Most industry AI programmers still mostly write their own AI from scratch (or reuse their own code)
- So we are going to look at coding details

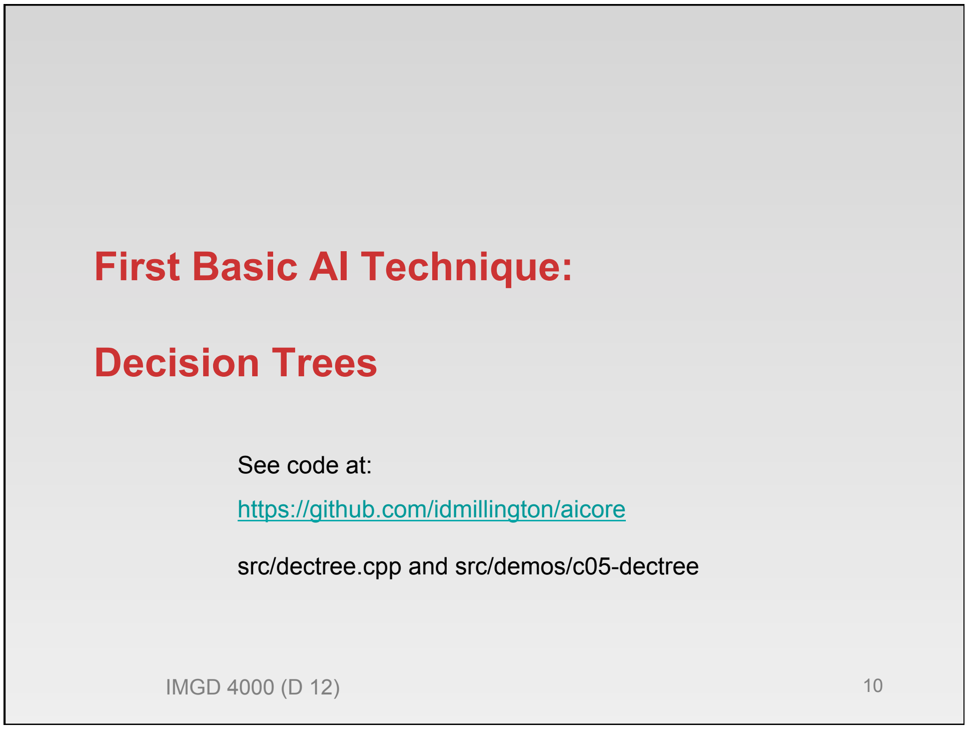


Basic AI Coding Theme

- Use *object-oriented* paradigm

instead of...

- A tangle of *if-then-else* statements



First Basic AI Technique:

Decision Trees

See code at:

https://github.com/idmillington/aicore

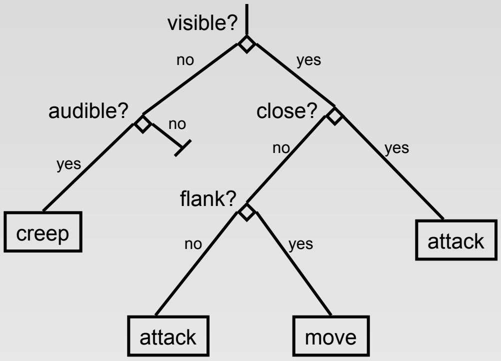
src/dectree.cpp and src/demos/c05-dectree

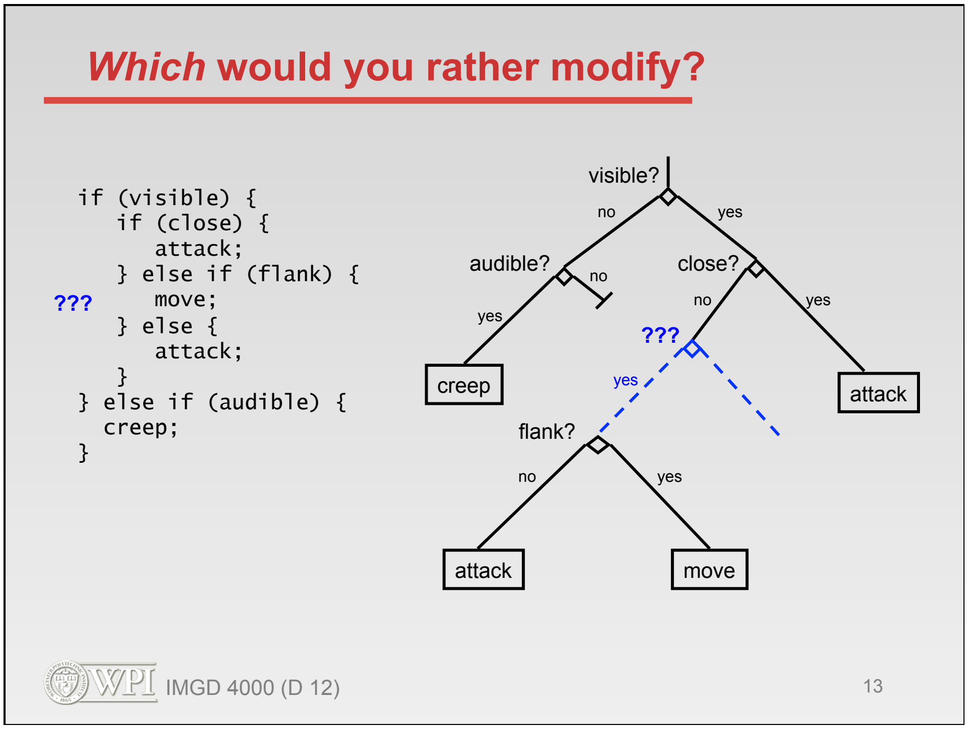
Decision Trees

- The most basic of the basic AI techniques
- Easy to implement
- Fast execution
- Simple to understand

Deciding how to respond to an enemy

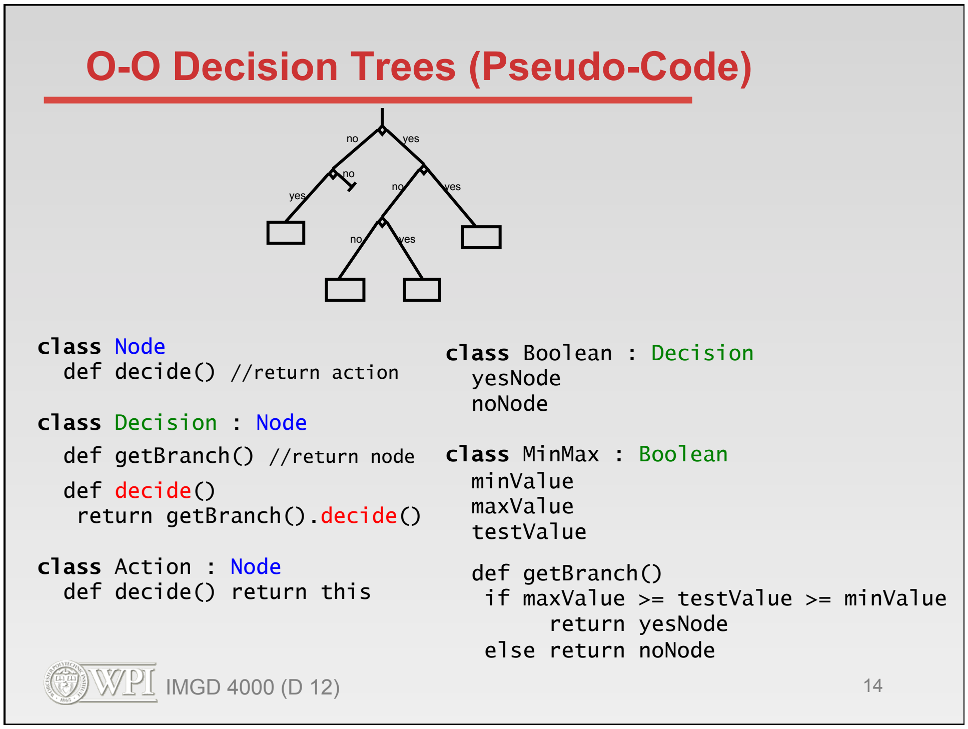
```
if (visible) {
   if (close) {
      attack;
   } else {
      if (flank) {
         move;
      } else {
         attack;
      }
   }
} else {
   if (audible) {
     creep;
   }
}
```





Which would you rather modify?

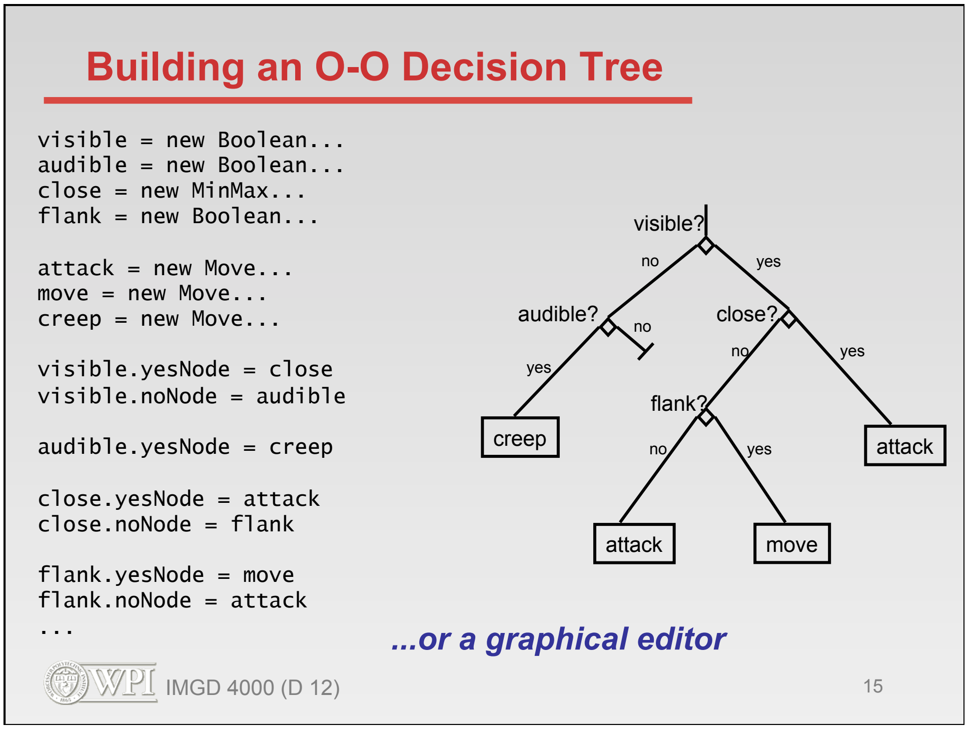
```
    if (visible) {
       if (close) {
          attack;
       } else if (flank) {
???         move;
       } else {
          attack;
       }
    } else if (audible) {
       creep;
    }
```



O-O Decision Trees (Pseudo-Code)

```
class Node
  def decide() //return action

class Decision : Node
  def getBranch() //return node
  def decide()
   return getBranch().decide()

class Action : Node
  def decide() return this
```

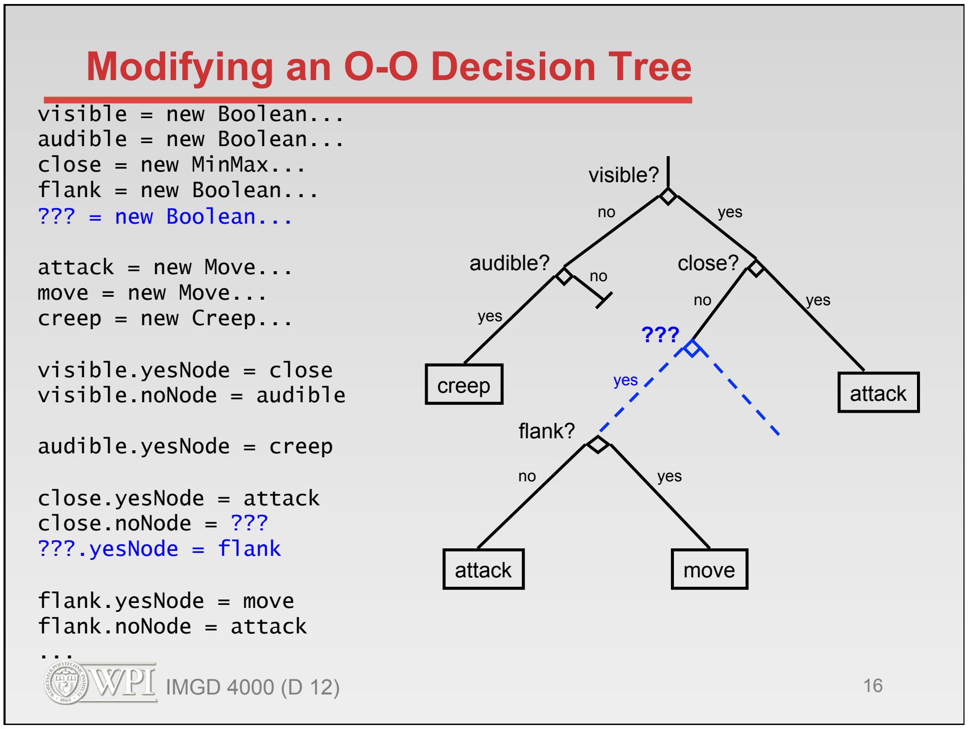
```
class Boolean : Decision
  yesNode
  noNode

class MinMax : Boolean
  minValue
  maxValue
  testValue

  def getBranch()
   if maxValue >= testValue >= minValue
        return yesNode
   else return noNode
```



Building an O-O Decision Tree

```
visible = new Boolean...
audible = new Boolean...
close = new MinMax...
flank = new Boolean...

attack = new Move...
move = new Move...
creep = new Move...

visible.yesNode = close
visible.noNode = audible

audible.yesNode = creep

close.yesNode = attack
close.noNode = flank

flank.yesNode = move
flank.noNode = attack
...
```

...or a graphical editor

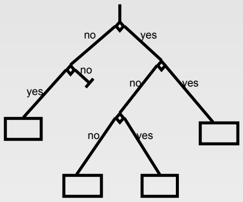


Modifying an O-O Decision Tree

```
visible = new Boolean...
audible = new Boolean...
close = new MinMax...
flank = new Boolean...
??? = new Boolean...

attack = new Move...
move = new Move...
creep = new Creep...

visible.yesNode = close
visible.noNode = audible

audible.yesNode = creep

close.yesNode = attack
close.noNode = ???
???.yesNode = flank

flank.yesNode = move
flank.noNode = attack
...
```

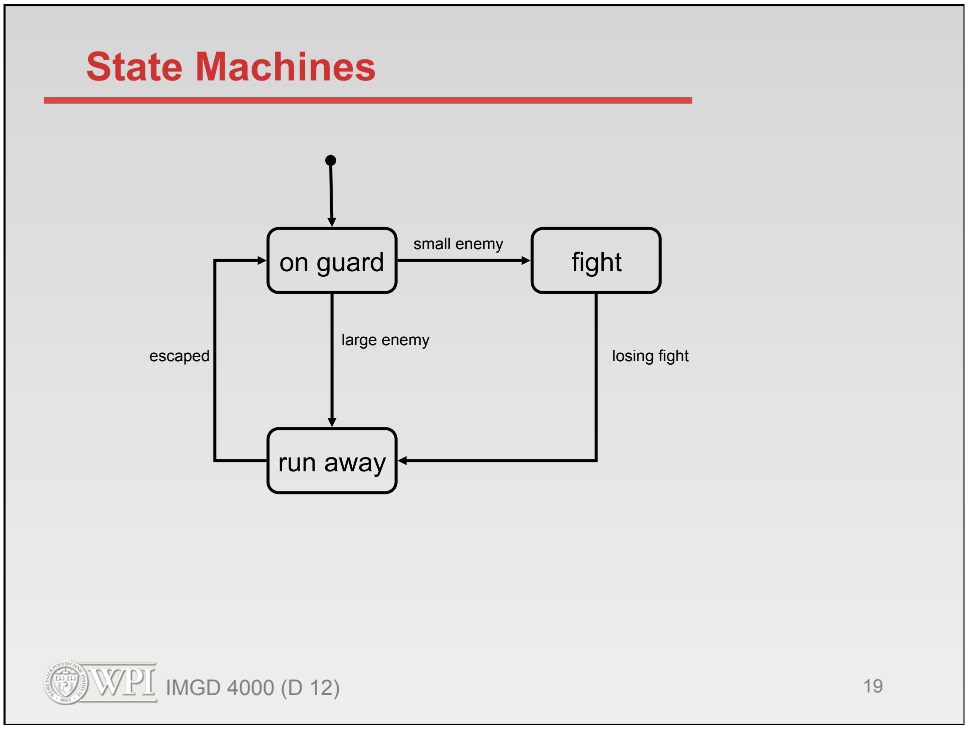
Performance Issues

- individual node tests (`getBranch`) typically constant time (and *fast*)
- worst case behavior depends on *depth* of tree
  - longest path from root to action
- roughly “balance” tree (when possible)
  - not too deep, not too wide
  - make commonly used paths shorter
  - put most expensive decisions late

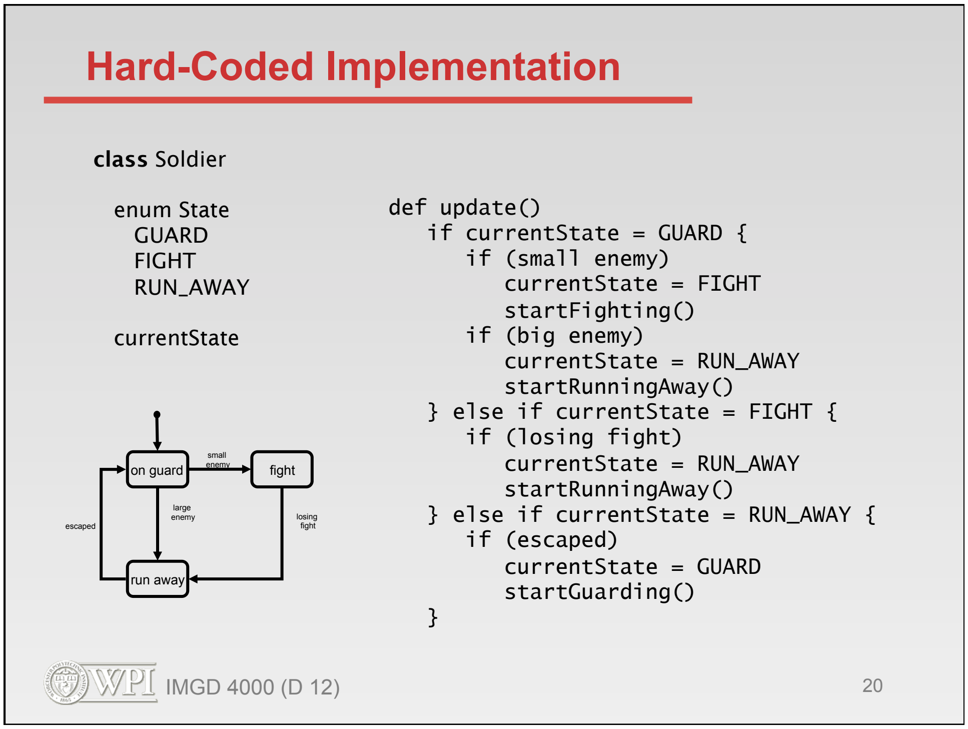


Second Basic AI Technique:

(Hierarchical) State Machines



State Machines



Hard-Coded Implementation

```
class Soldier

  enum State
    GUARD
    FIGHT
    RUN_AWAY

  currentState
```

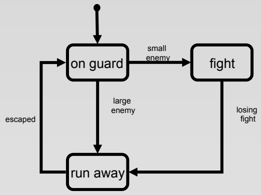
```
def update()
   if currentState = GUARD {
      if (small enemy)
         currentState = FIGHT
         startFighting()
      if (big enemy)
         currentState = RUN_AWAY
         startRunningAway()
   } else if currentState = FIGHT {
      if (losing fight)
         currentState = RUN_AWAY
         startRunningAway()
   } else if currentState = RUN_AWAY {
      if (escaped)
         currentState = GUARD
         startGuarding()
   }
```

Hard-Coded State Machines

- Easy to write (at the start)
- Very efficient
- Notoriously hard to maintain (e.g., debug)

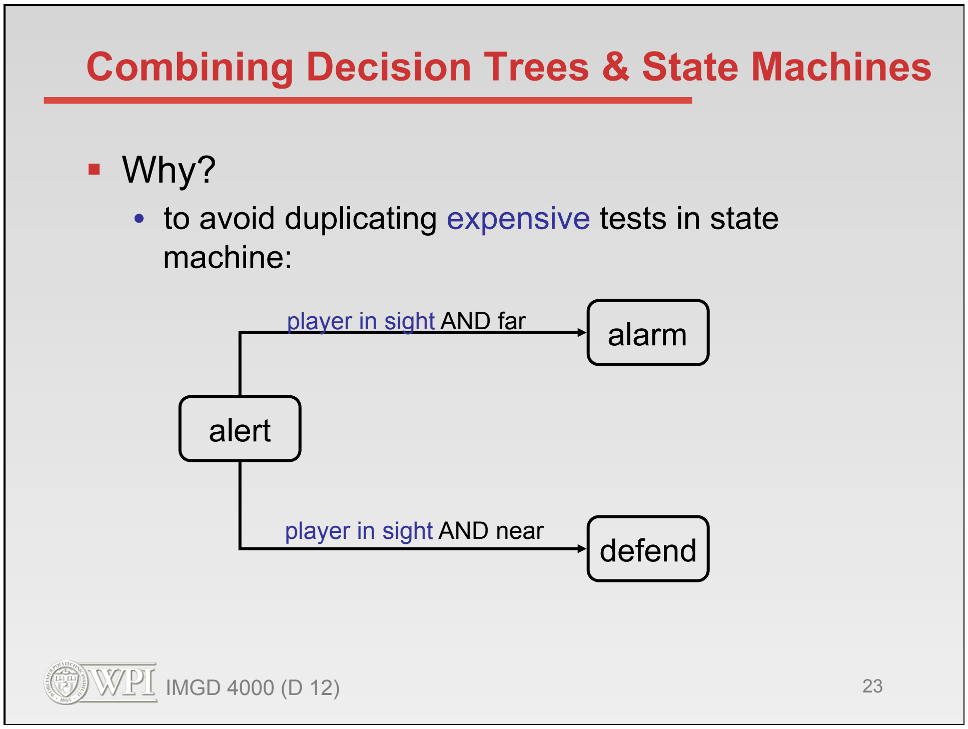
Cleaner & More Flexible O-O Implementation

```
class State
   def getAction()
   def getEntryAction()
   def getExitAction()
   def getTransitions()

class Transition
   def isTriggered()
   def getTargetState()
   def getAction()
```



```
class StateMachine

   states
   initialState
   currentState = initialState

   def update()

      triggeredTransition = null

      for transition in currentState.getTransitions()
         if transition.isTriggered()
            triggeredTransition = transition
            break

      if triggeredTransition
         targetState = triggeredTransition.getTargetState()
         actions = currentState.getExitAction()
         actions += triggeredTransition.getAction()
         actions += targetState.getEntryAction()
         currentState = targetState
         return actions
      else
         return currentState.getAction()
```

...add tracing



Combining Decision Trees & State Machines

- Why?
  - to avoid duplicating expensive tests in state machine:

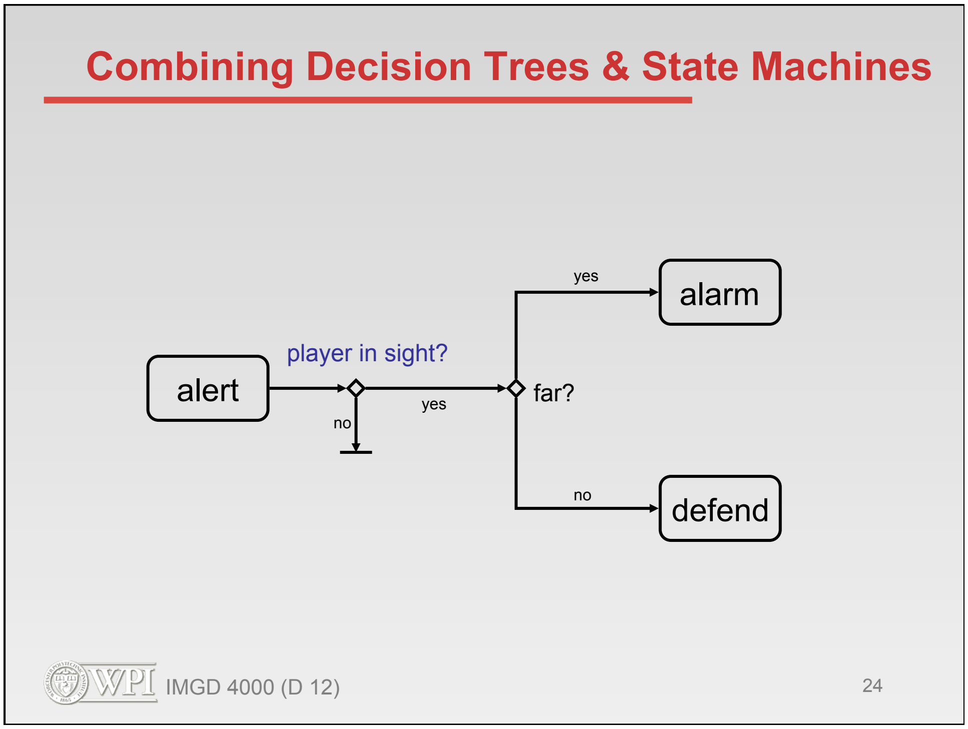
player in sight AND far

alarm

alert

player in sight AND near

defend



Combining Decision Trees & State Machines

yes

alarm

alert

player in sight?

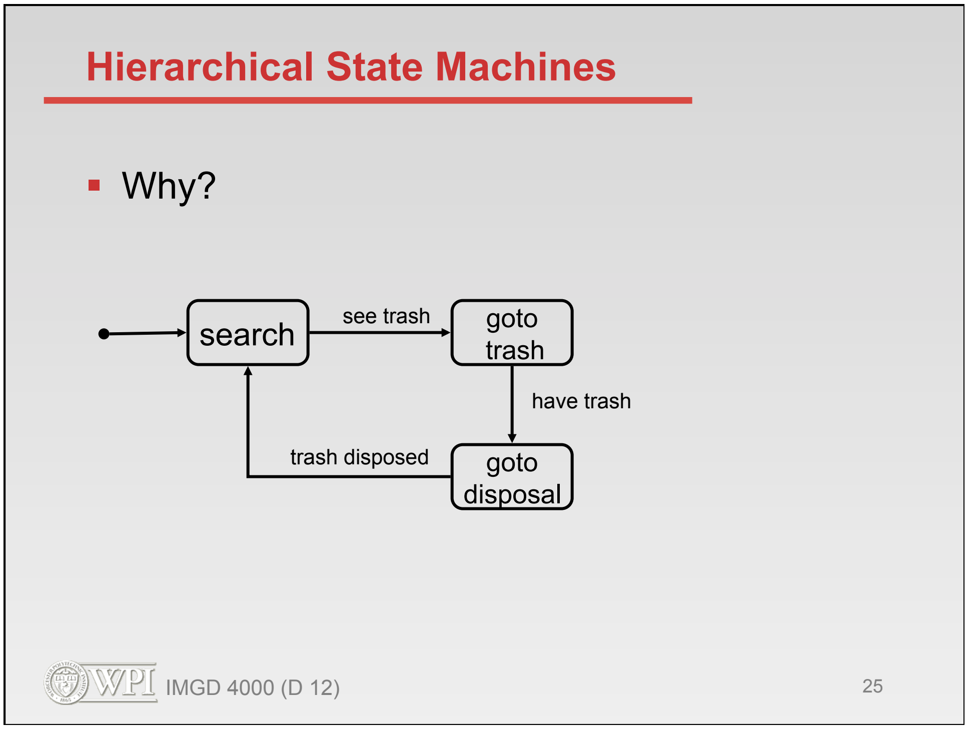
yes

far?

no

no

defend



Hierarchical State Machines

- Why?

search

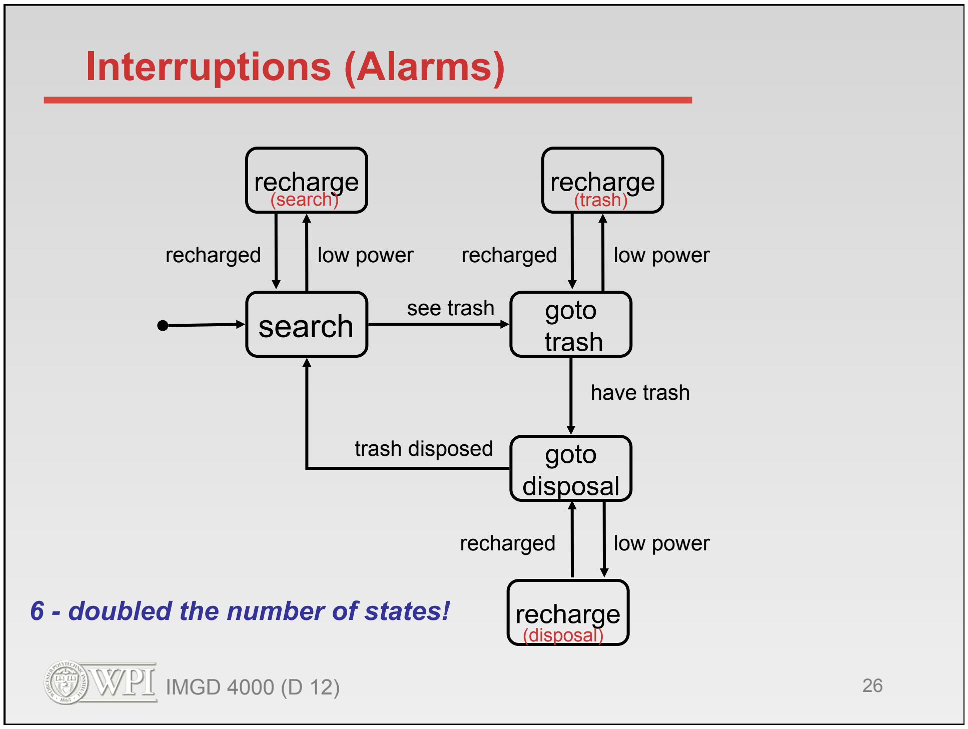
see trash

goto trash

have trash

goto disposal

trash disposed



Interruptions (Alarms)

recharge (search)

recharged

low power

recharge (trash)

recharged

low power

search

see trash

goto trash

have trash

trash disposed

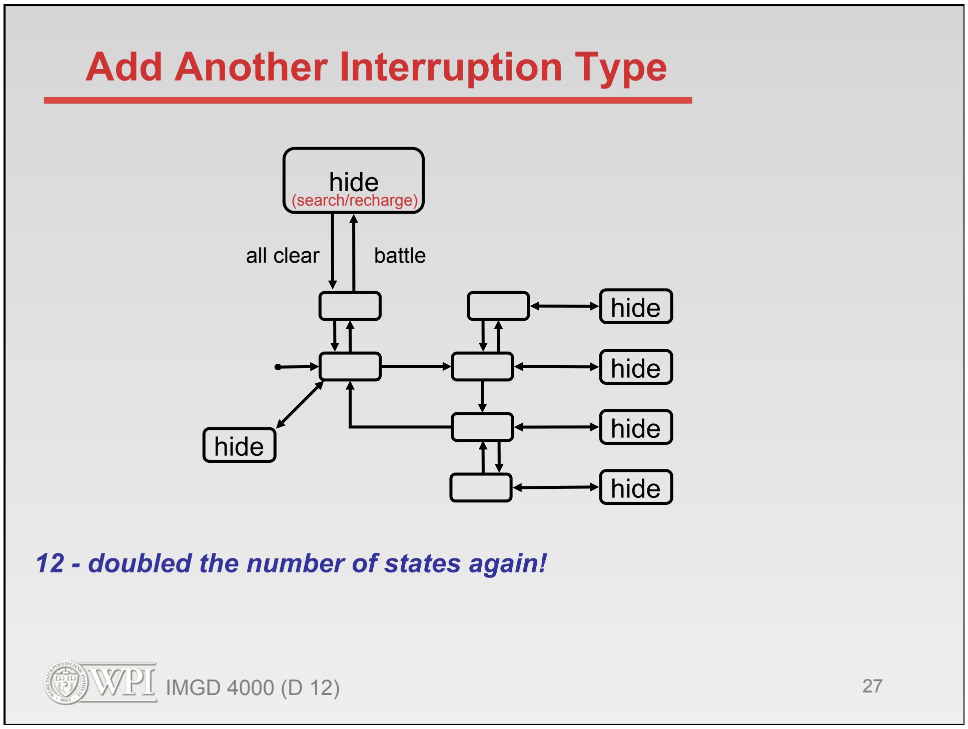
goto disposal

recharged

low power

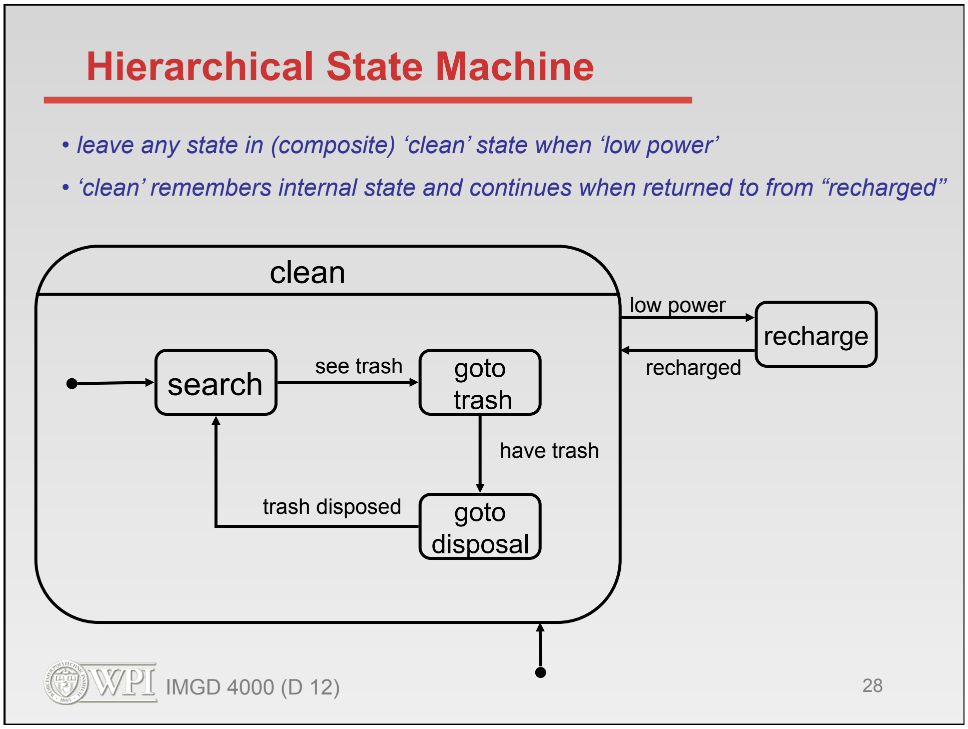
recharge (disposal)

6 - doubled the number of states!



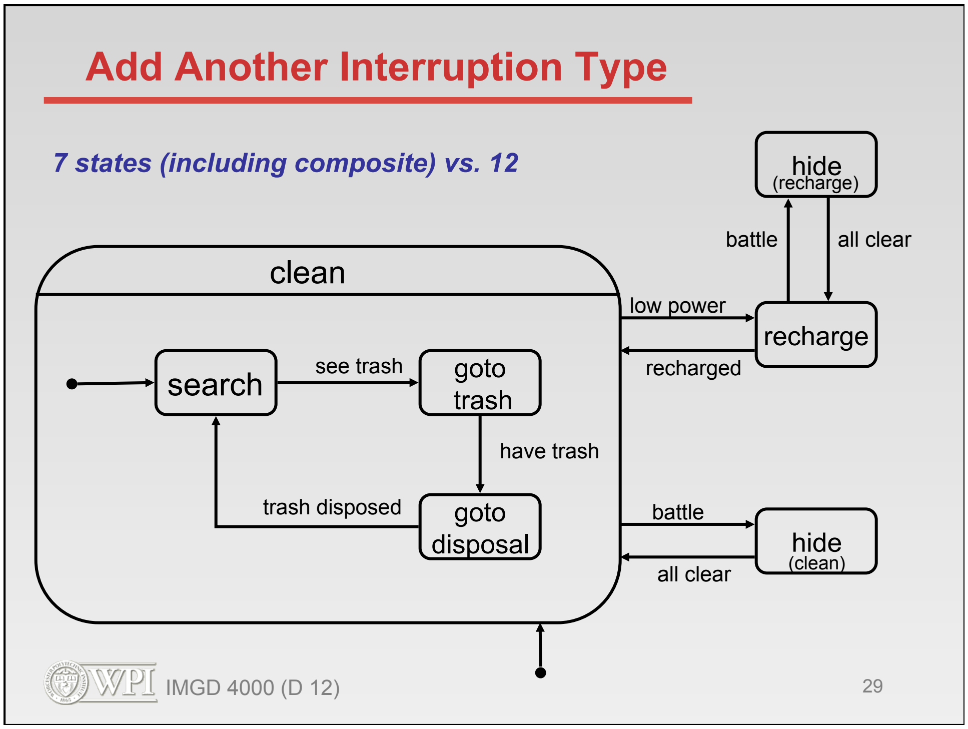
Add Another Interruption Type

12 - doubled the number of states again!



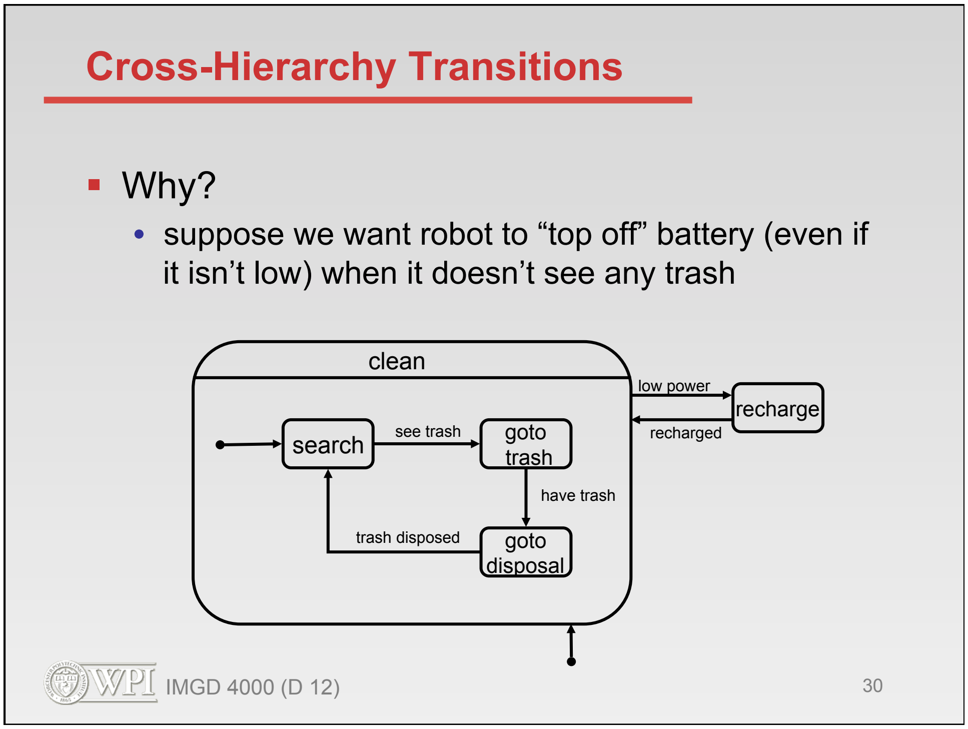
Hierarchical State Machine

- *leave any state in (composite) ‘clean’ state when ‘low power’*
- *‘clean’ remembers internal state and continues when returned to from “recharged”*



Add Another Interruption Type

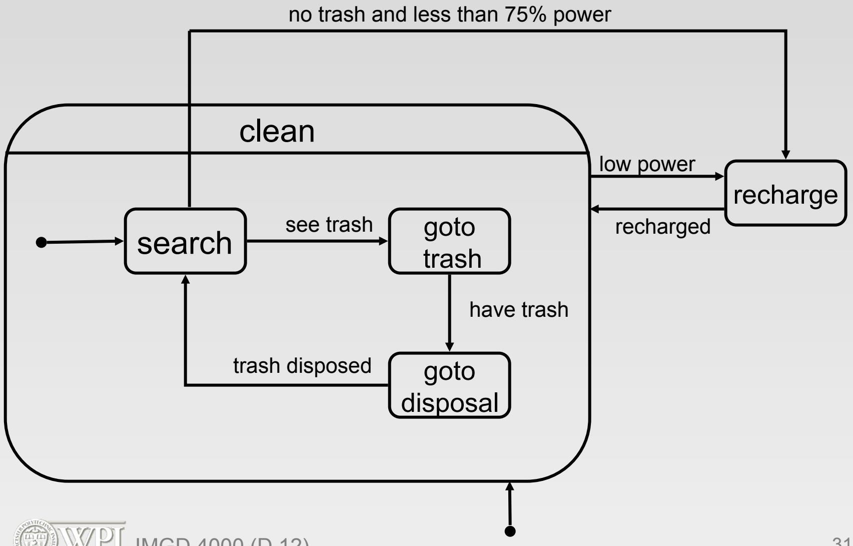
7 states (including composite) vs. 12



Cross-Hierarchy Transitions

- Why?
  - suppose we want robot to “top off” battery (even if it isn’t low) when it doesn’t see any trash

Cross-Hierarchy Transitions



Implementation Sketch

```
class State

   # stack of return states
   def getStates() return [this]

   # recursive update
   def update()

   # rest same as flat machine

class Transition

   # how deep this transition is
   def getLevel()

   # rest same as flat machine

struct UpdateResult # returned from update
   transition
   level
   actions # same as flat machine
```

```
class HierarchicalStateMachine

   # same state variables as flat machine

   # complicated recursive algorithm*
   def update ()


class SubMachine : HierarchicalStateMachine,
                  State

   def getStates()
      push this onto currentState.getStates()
```

See full pseudo-code at http://www.cs.wpi.edu/~rich/courses/imgd4000-d12/hsm.pdf

Add tracing/debug code!!