

Principles of Computer Game Design and Implementation

Lecture 23

We already learned

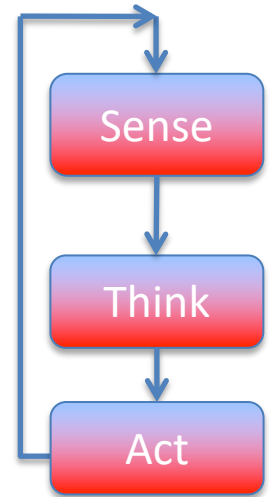
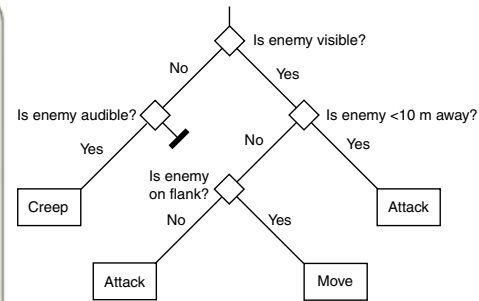
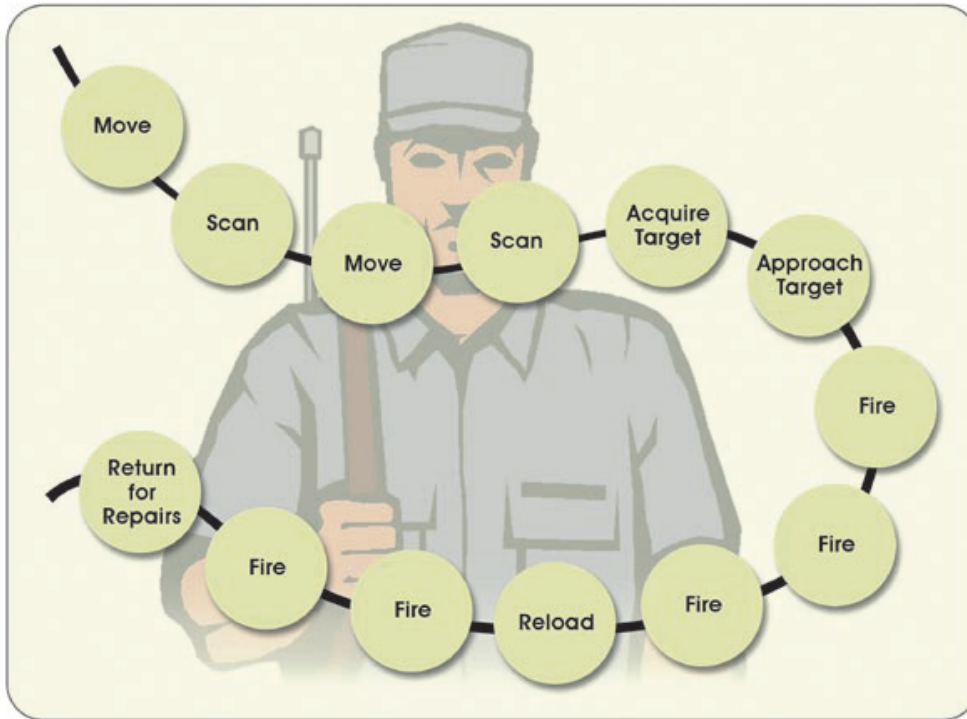
- Decision Tree

Outline for today

- Finite state machine

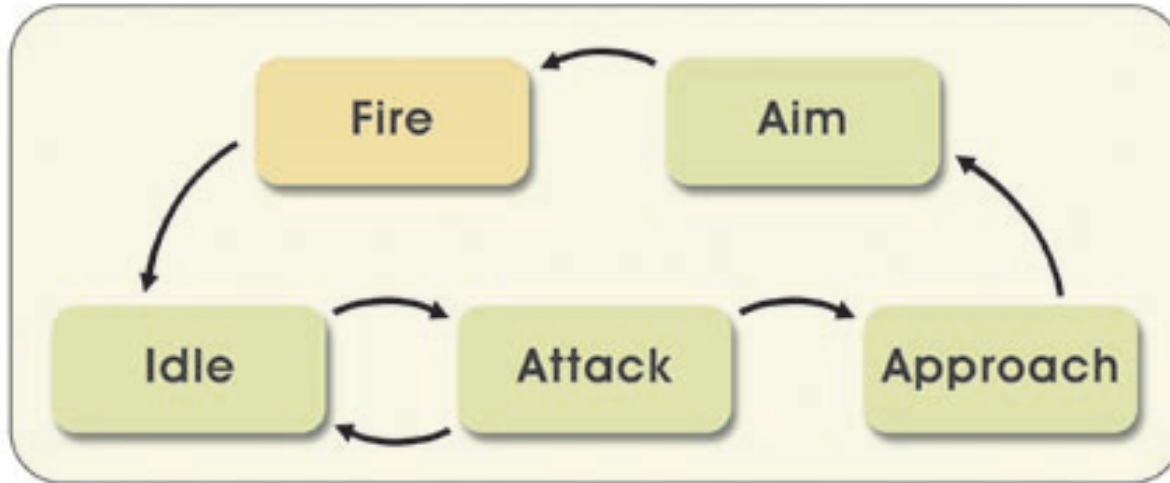
Creating & Controlling AI Behaviors

Behavior: A Sequence of Actions



The patrol and guard behavior is defined as a sequence of actions

So, Basically...



- An agent goes through a sequence of *states*
- Arrows indicate *transitions*

Finite-State Machine (FSMs)

- Abstract model of computation
 - Formally:
 - Set of states
 - A starting state
 - An input vocabulary
 - A transition function that maps inputs and the current state to a next state

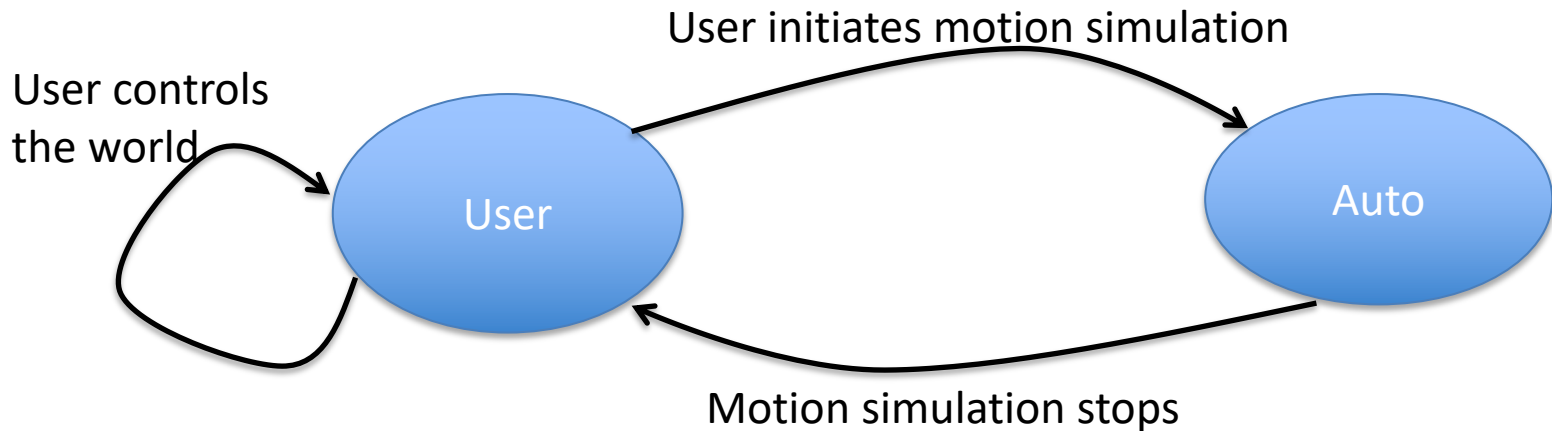
FSMs In Game Development

Deviate from formal definition

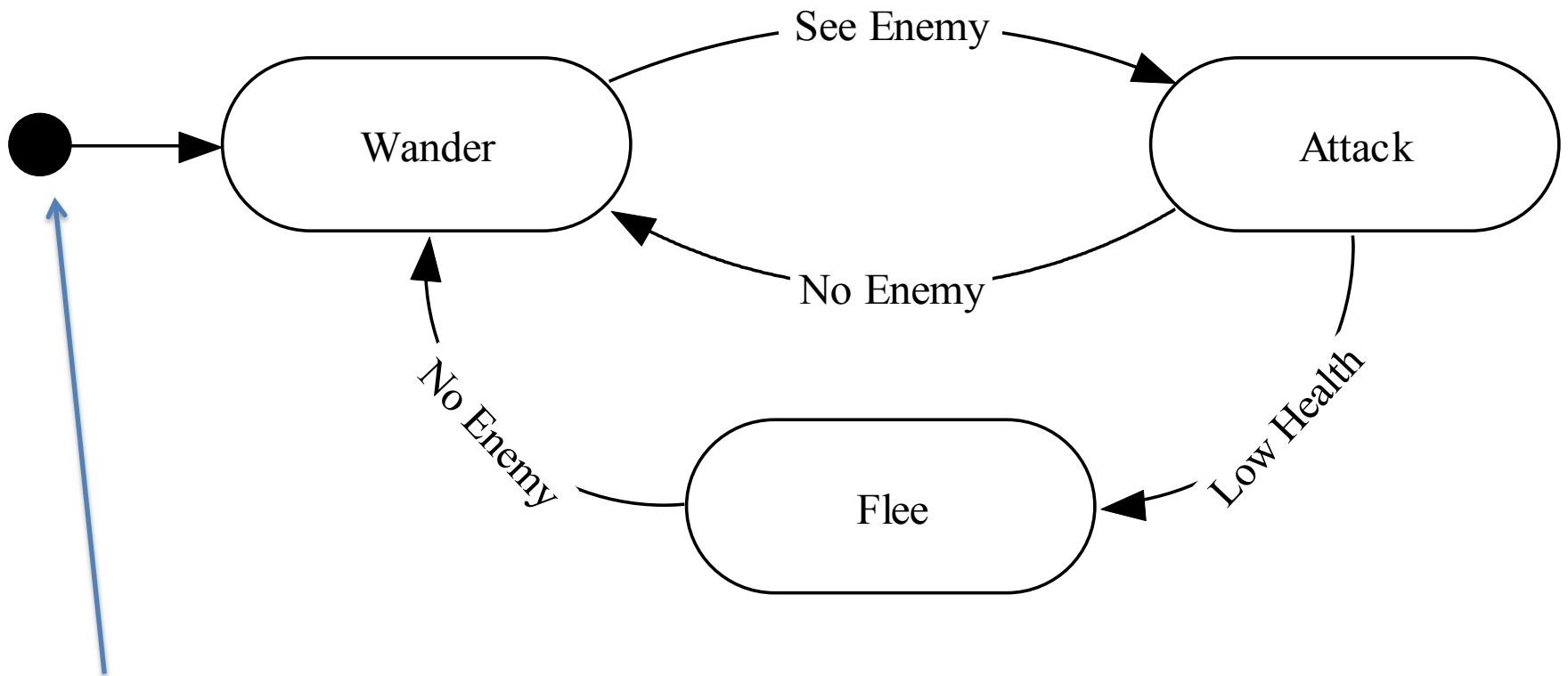
1. States define behaviors (containing code)
 - Wander, Attack, Flee
 - As longer as an agent stays in a state, it carries on the same action
2. Transition function divided among states
 - Keeps relation clear
3. Extra state information
 - For example, health

Recall: User Control V Modelling

- In these examples, user controlled completely the state of the world or there was no user input.
 - How to mix user control and physical modelling?
 - Game states



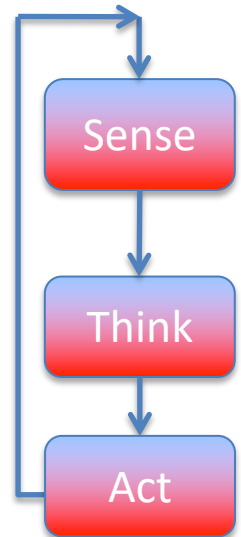
Finite-State Machine: UML Diagram



Initial state

State Actions

- Actions is what player sees
 - Movement
 - Animation
- Instead of one action can consider
 - onEntry
 - Executed when FSM enters the state
 - onExit
 - onUpdate
 - Runs *every tick* while FSM is in the state



Finite-State Machine: Approaches

- Three approaches
 - Hardcoded (switch statement)
 - Scripted
 - Hybrid Approach

Hard-Coded FSM

```
enum State {wander, attack, flee};
State state;
...
switch (state )
{
    case wander:
        Wander();
        if( SeeEnemy() )      { state = State.attack; }
        break;

    case attack:
        Attack();
        if( LowOnHealth() ) { state = State.flee; }
        if( NoEnemy() )     { state = State.wander; }
        break;

    case flee:
        Flee();
        if( NoEnemy() )     { state = State.wander; }
        break;
}
```

Hard-Coded FSM: Weaknesses

- Maintainability
 - Language doesn't enforce structure
 - Can't determine 1st time state is entered
- FSM change -> recompilation
 - Critical for large projects
 - Cannot be changed by game designers / players
- Harder to extend
 - Hierarchical FSMs
 - Probabilistic / fuzzy FSMs

Finite-State Machine: Scripted with alternative language

```
BeginFSM
  State( STATE_Wander )
    OnEnter
      Java code
    OnUpdate
      Java code

      if(seeEnemy()) ChangeState(STATE_Attack);

    OnExit
      Java code
  State( STATE_Attack )
    OnEnter
      Java code
    OnUpdate
      Java code to execute every tick
    OnExit
EndFSM
```

Finite-State Machine: Scripting Advantages

1. Structure enforced
2. Events can be handed as well as polling
3. OnEnter and OnExit concept exists
4. Can be authored by game designers
 - Easier learning curve than straight C/C++

Finite-State Machine: Scripting Disadvantages

- Not trivial to implement
- Several months of development
 - Custom compiler
 - With good compile-time error feedback
 - Bytecode interpreter
 - With good debugging hooks and support
- Scripting languages often disliked by users
 - Can never approach polish and robustness of commercial compilers/debuggers

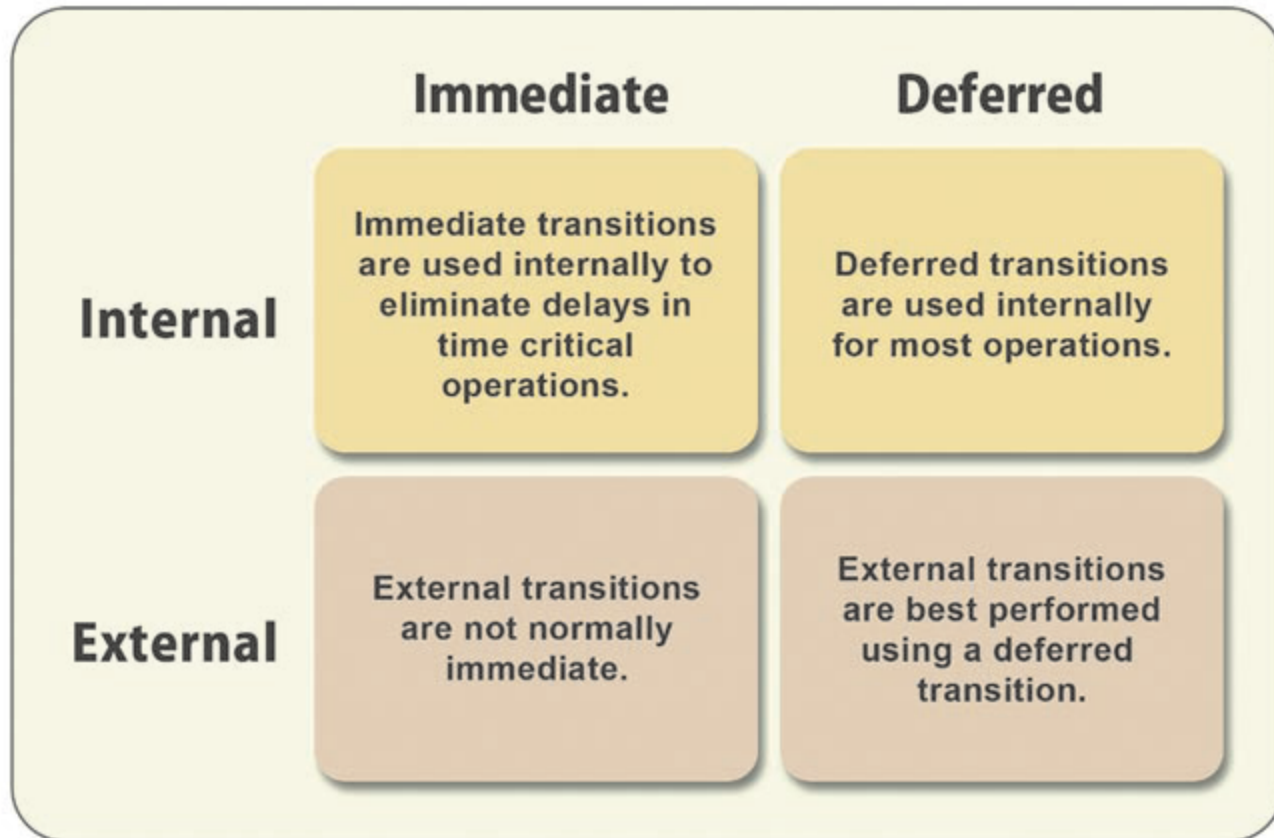
Finite-State Machine: Hybrid Approach

- Use a class and C-style macros to approximate a scripting language
- Allows FSM to be written completely in C++ leveraging existing compiler/debugger
- Capture important features/extensions
 - OnEnter, OnExit
 - Timers
 - Handle events
 - Consistent regulated structure
 - Ability to log history
 - Modular, flexible, stack-based
 - Multiple FSMs, Concurrent FSMs
- Can't be edited by designers or players

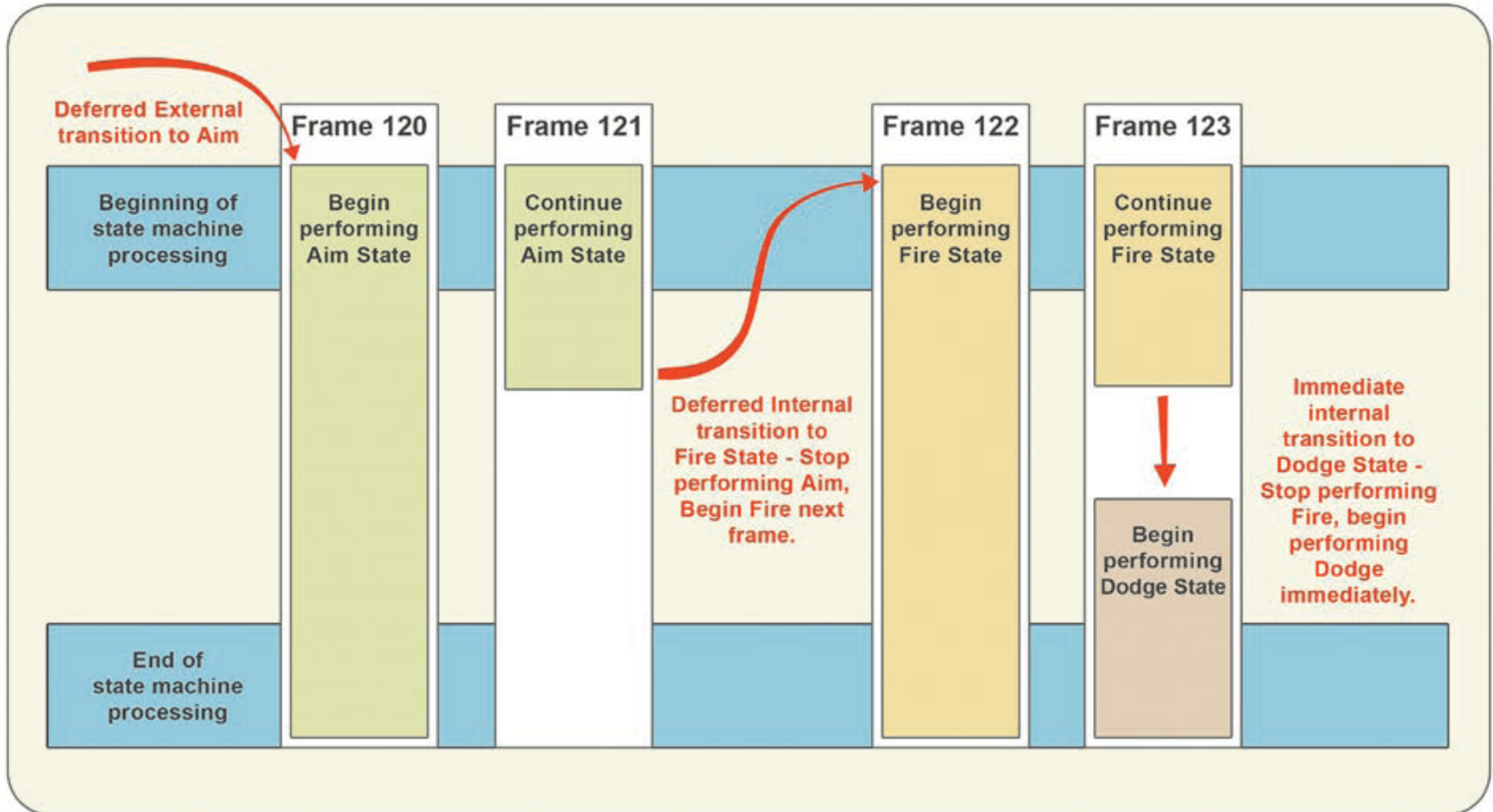
Transitions

- Internal
 - Independent of environment
 - E.g. out of ammo
- External
 - Event-driven
- Immediate
- Deferred
 - E.g. to wait till animation sequence stops

Transitions



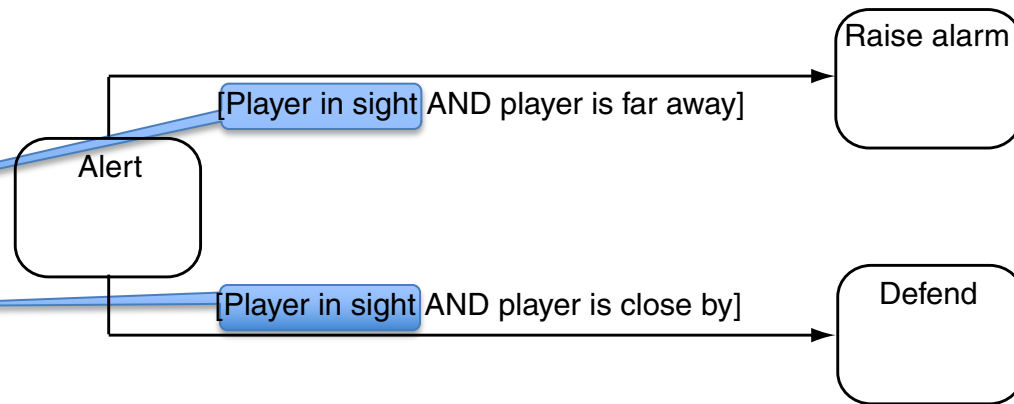
Transitions



Decision Trees in Transitions

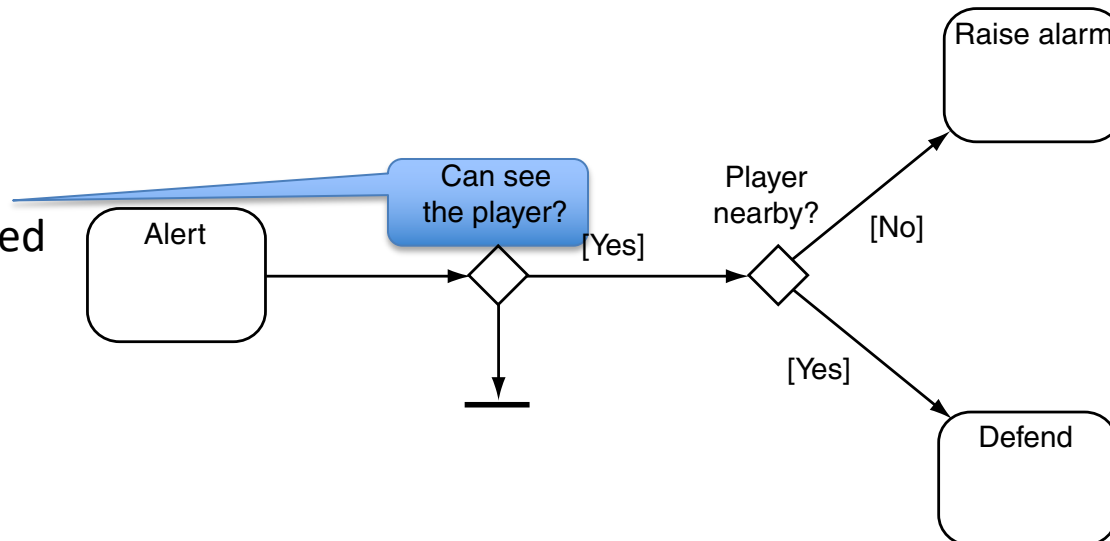
- Compare

Computationally-expensive test performed **twice**



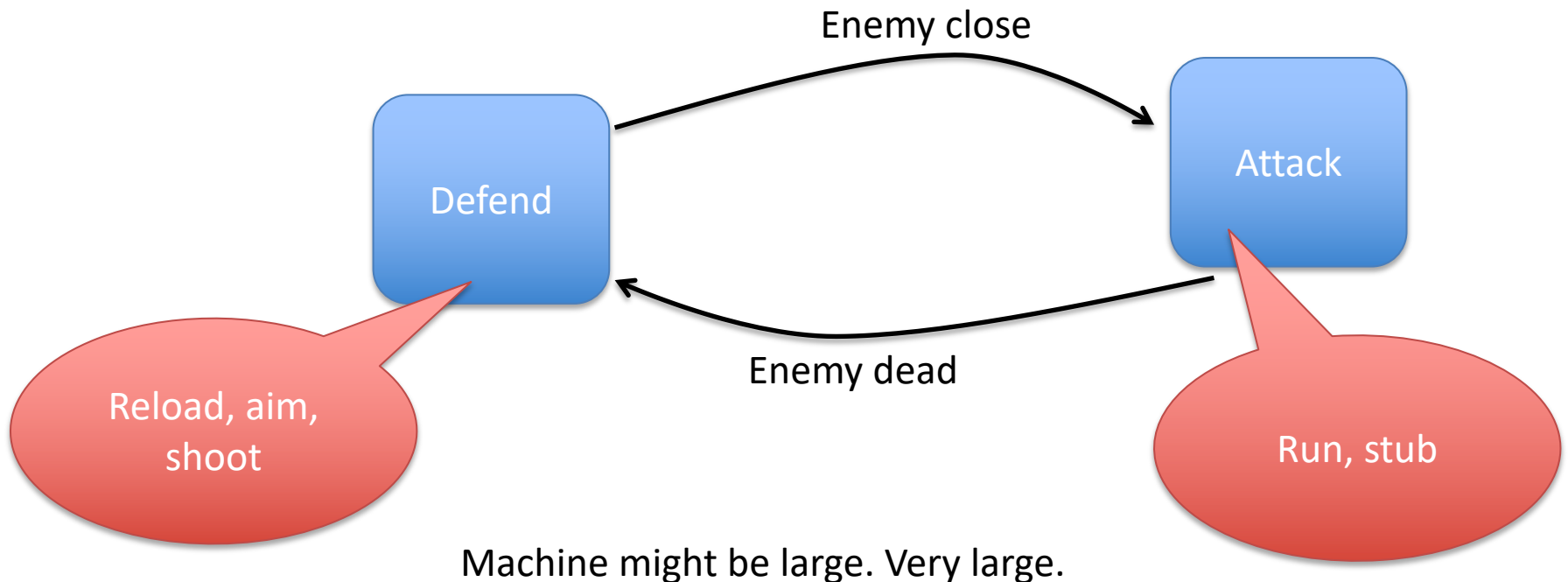
- With

Computationally-expensive test performed **once**



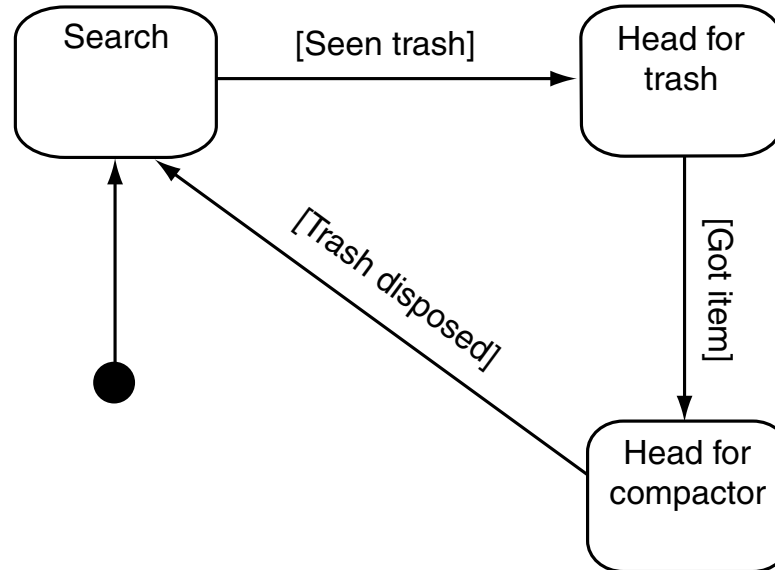
Generalisation: Hierarchical FSM

- Often, there are several “levels” of behaviour
 - Complications from “insignificant details”



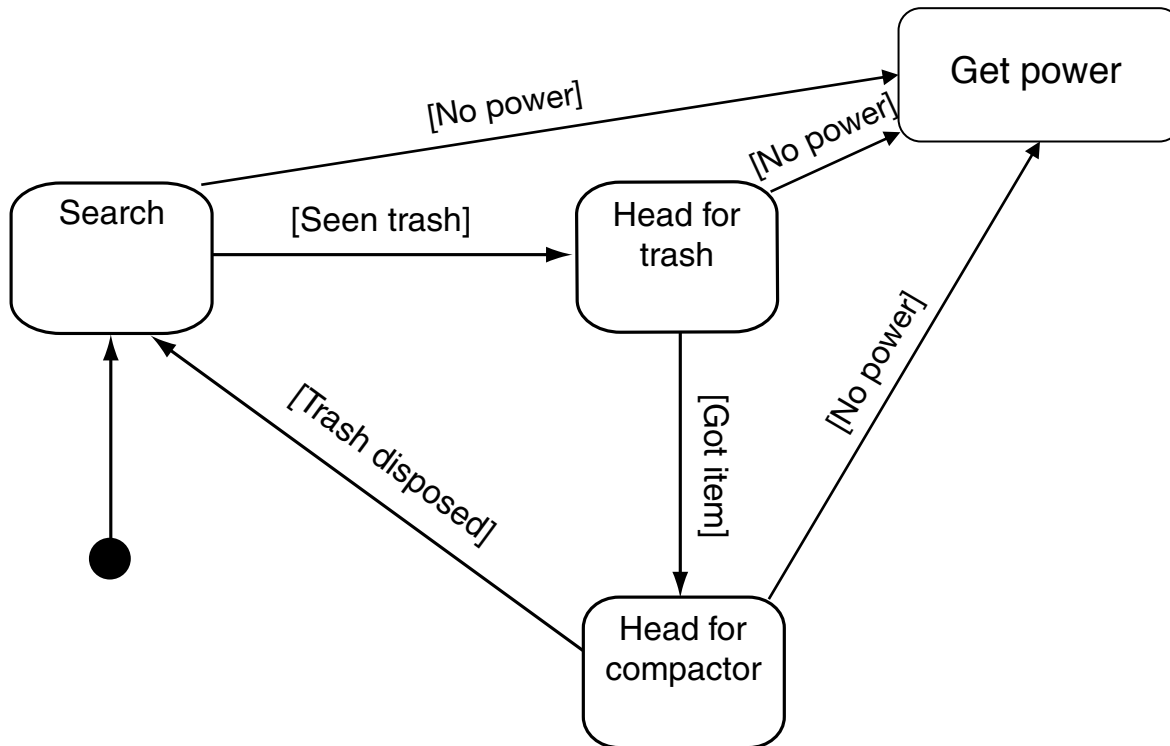
Clean Up FSM Example

- A robot cleans a floor space



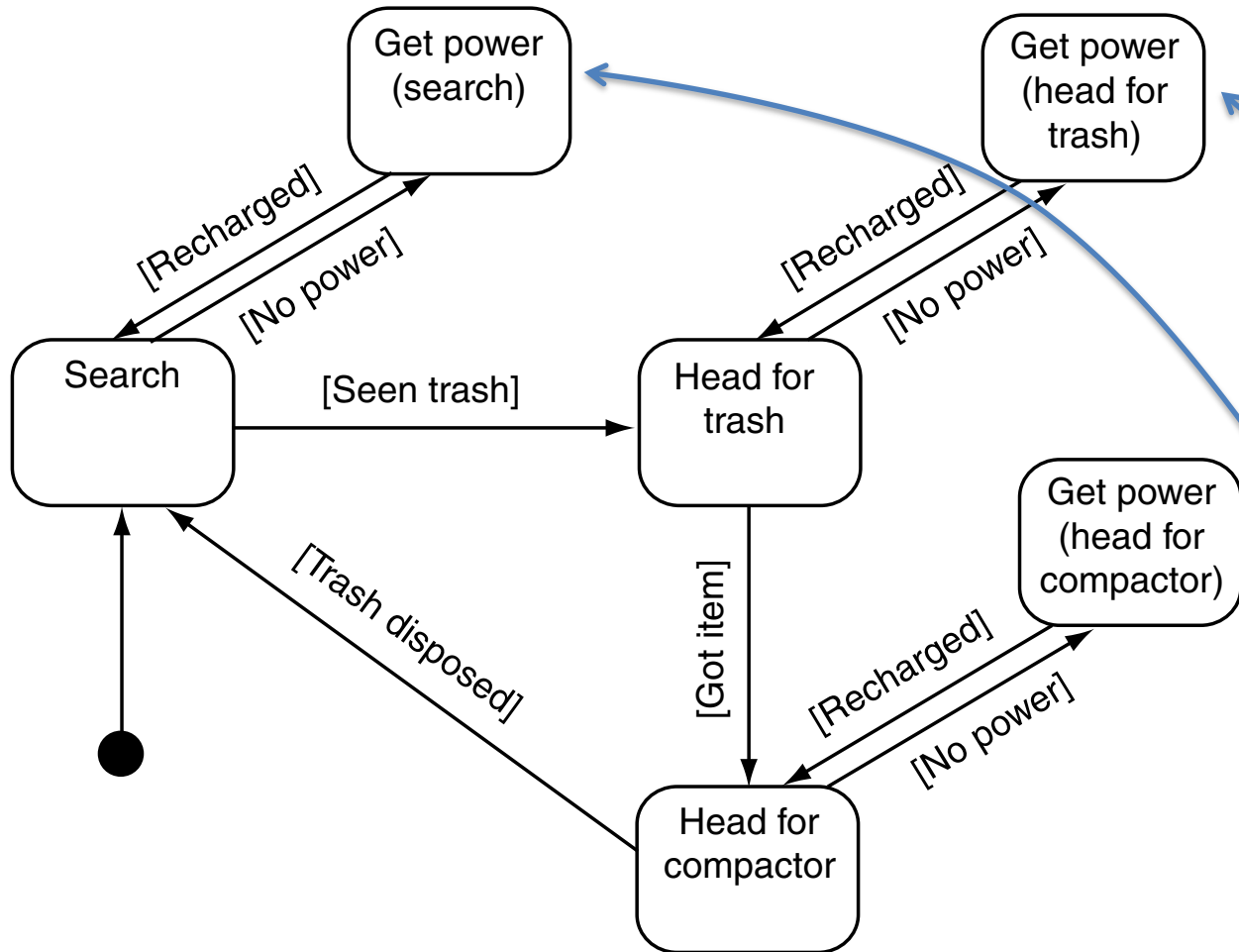
- Unless it recharges, it breaks

Recharging Clean Up FSM Example



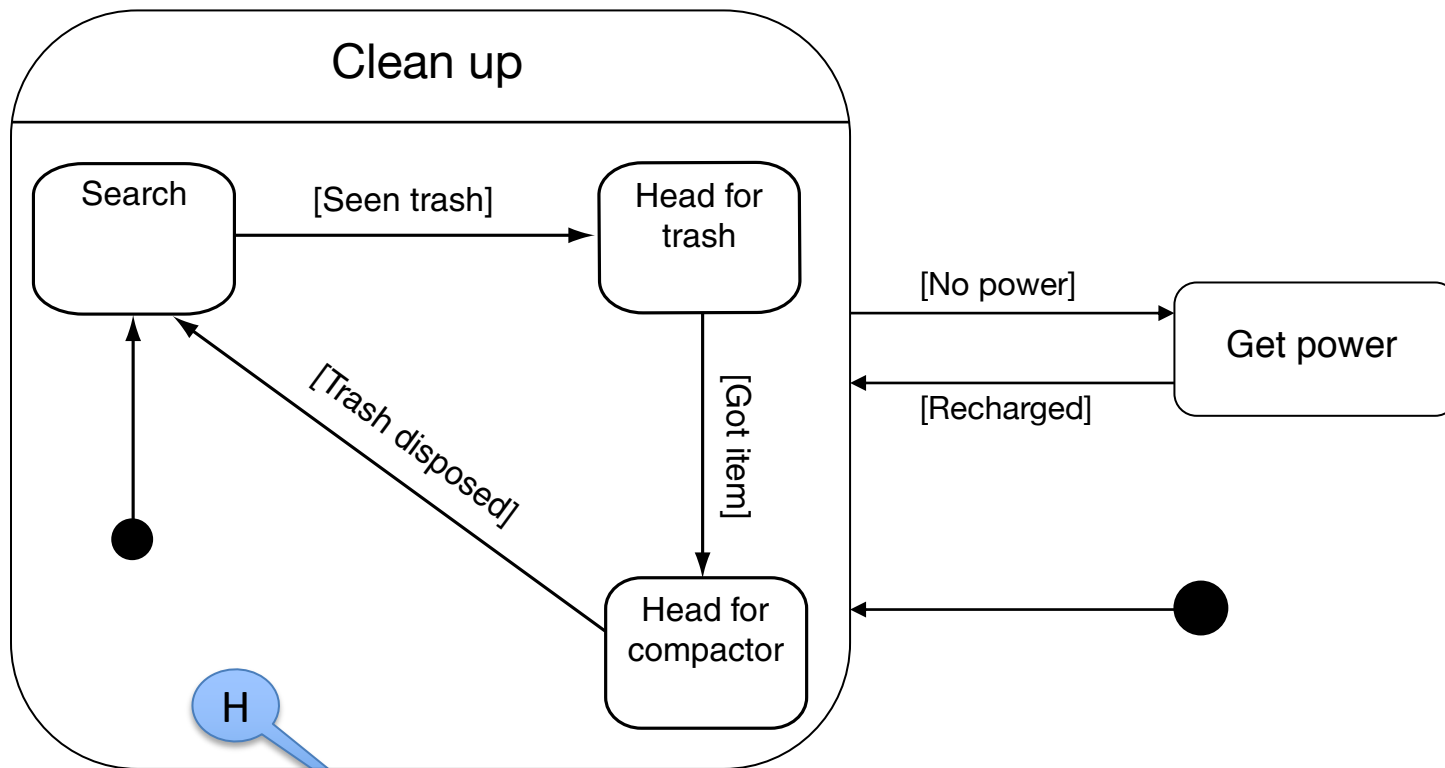
But what to do **after** charging???

Recharging Cleaner FSM



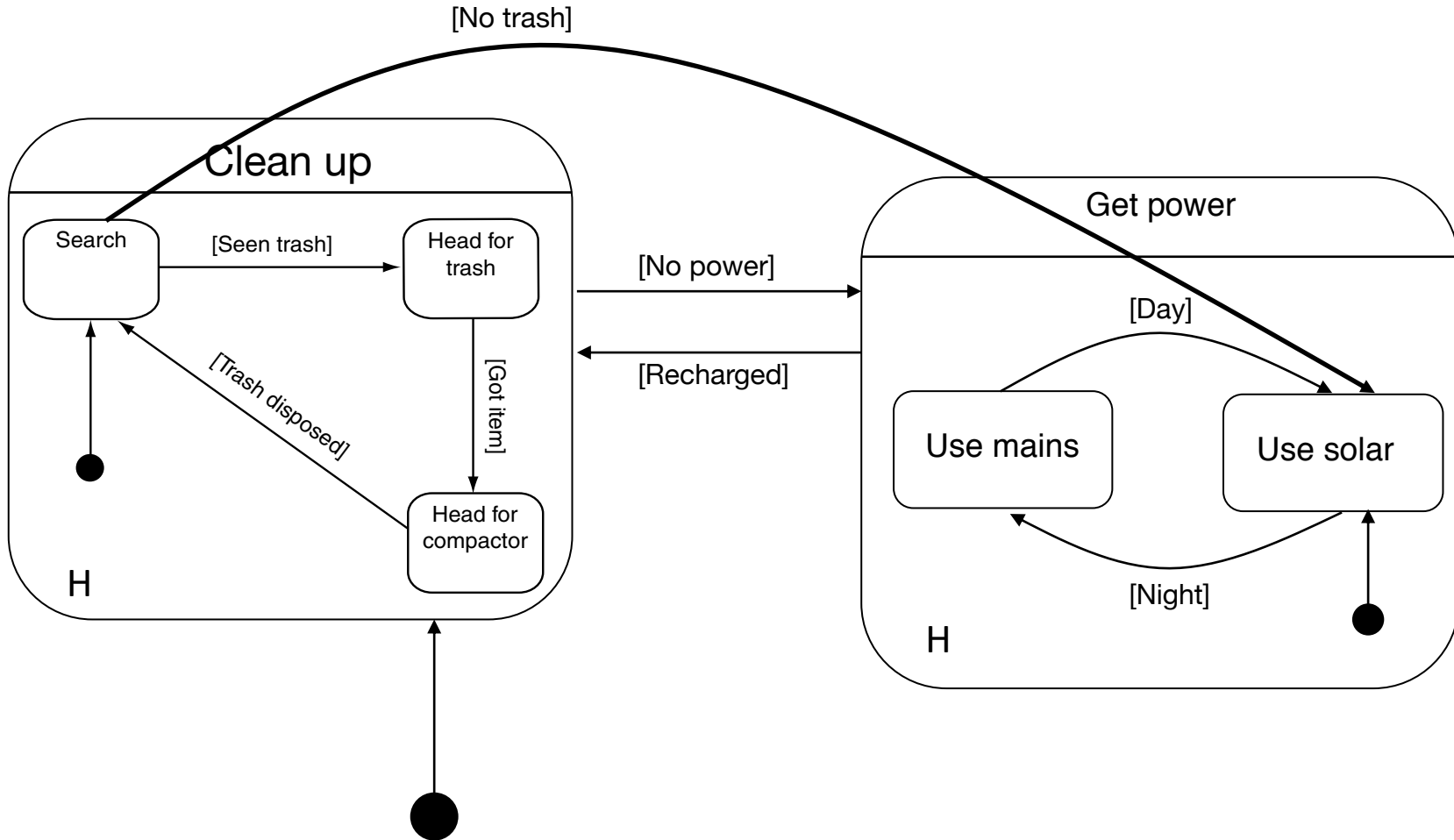
Three states just to remember where to come back

Hierarchical Approach



Hierarchical state

Hierarchical Recharge



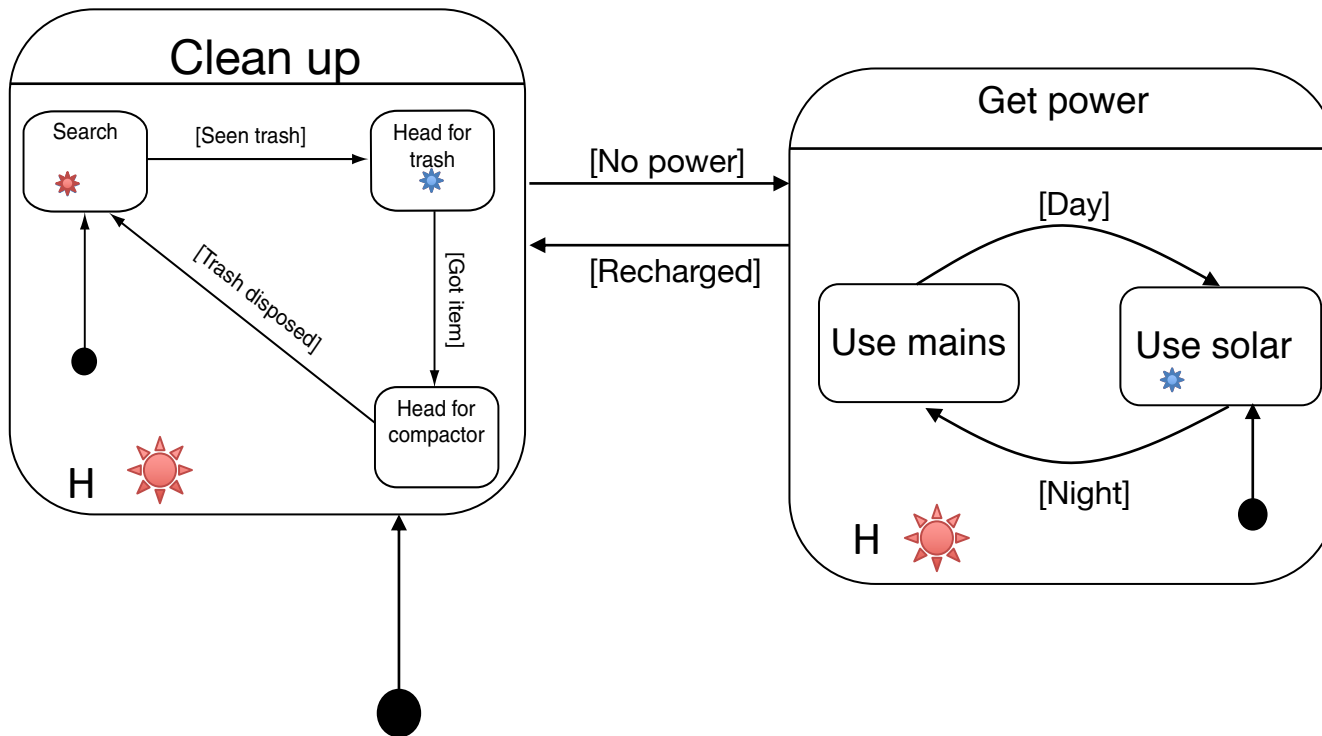
Algorithm

- Based on the notion of a *current state*
 - Every state stores the current state of its sub FSM
- Hierarchical evaluation
 - If transition is applicable to higher-level current state
 - Change state
 - Else
 - Execute the OnStay method
 - Apply transition to the sub FSM

Example

Events:

- No power
- Recharged
- Seen trash
- No power
- ...



Stack-Based FSMs

- This idea can be extended to allow storing past states using a stack
- Every time a machine is “suspended” the current state is pushed into the stack
- Every time it is “resumed” the state is popped from the stack
 - E.g. several machines and a switch between them

Finite-State Machine In Game Development: Summary

- Most common game AI software pattern
 - Natural correspondence between states and behaviors
 - Easy to diagram
 - Easy to program
 - *Easy to debug*
 - Completely general to any problem
- Problems
 - Explosion of states
 - Too predictable
 - Often created with ad hoc structure