AI for Games

State Machines

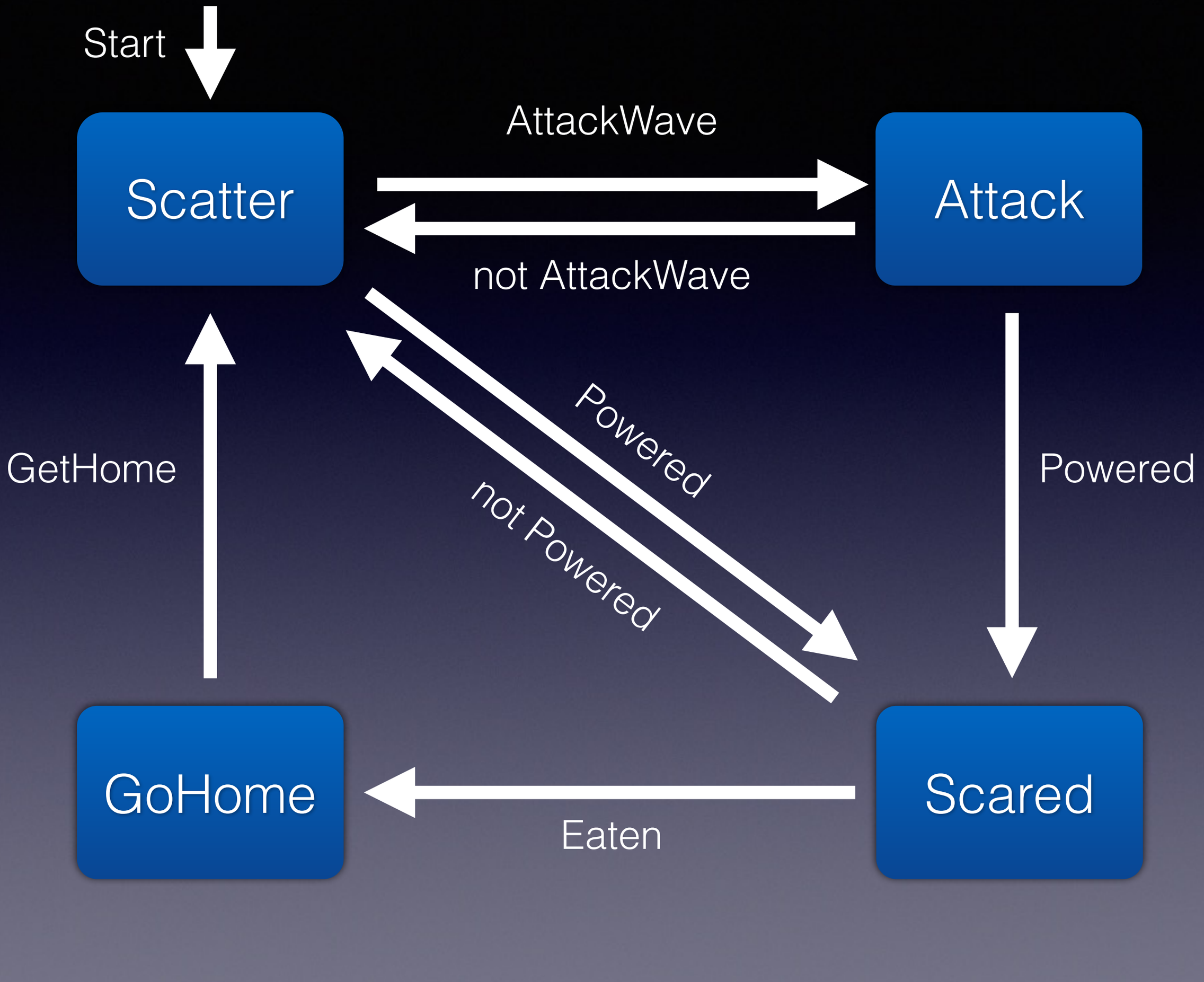
Jeremy Gow
j.gow@gold.ac.uk

https://learn.gold.ac.uk/course/view.php?id=3109

Ghost AI

States
Attack
Scatter
Scared
GoHome

Conditions
AttackWave
Powered
Eaten
GetHome


Start
Scatter
AttackWave
not AttackWave
Attack
GetHome
Powered
not Powered
Powered
GoHome
Eaten
Scared

State Machines

A Definition

- States $S = \{s_1, s_2, \ldots, s_n\}$
- Current state $s \in S$
- Start state $s_{start} \in S$
- Conditions $\{c_1, c_2, \ldots, c_m\}$
- Actions $\{a_1, a_2, \ldots a_k\}$
- Transition rules for changing states
- "**If** $s==s_1$ and c_3 **then** $s=s_2$ and do a_5"

State Machines

Terminology

- **Game AI**: any state-driven approach is called a *Finite State Machine* (FSM)
- **Comp. Sci.**: FSM has a specific meaning
- Conditions = tests, inputs, events, triggers, …
- Actions = outputs, …

AttackRanged, AttackClose, SeekCover, RunAway, …

Strike, Dribble, ChaseBall, MarkPlayer, …

Implementing FSMs

1. switch/if-else

2. Transition tables

3. State objects

switch/if-else FSM

- Every AI tick…
- Execute current state (switch)
    - Execute state behaviour
    - Test transition conditions (if-else)

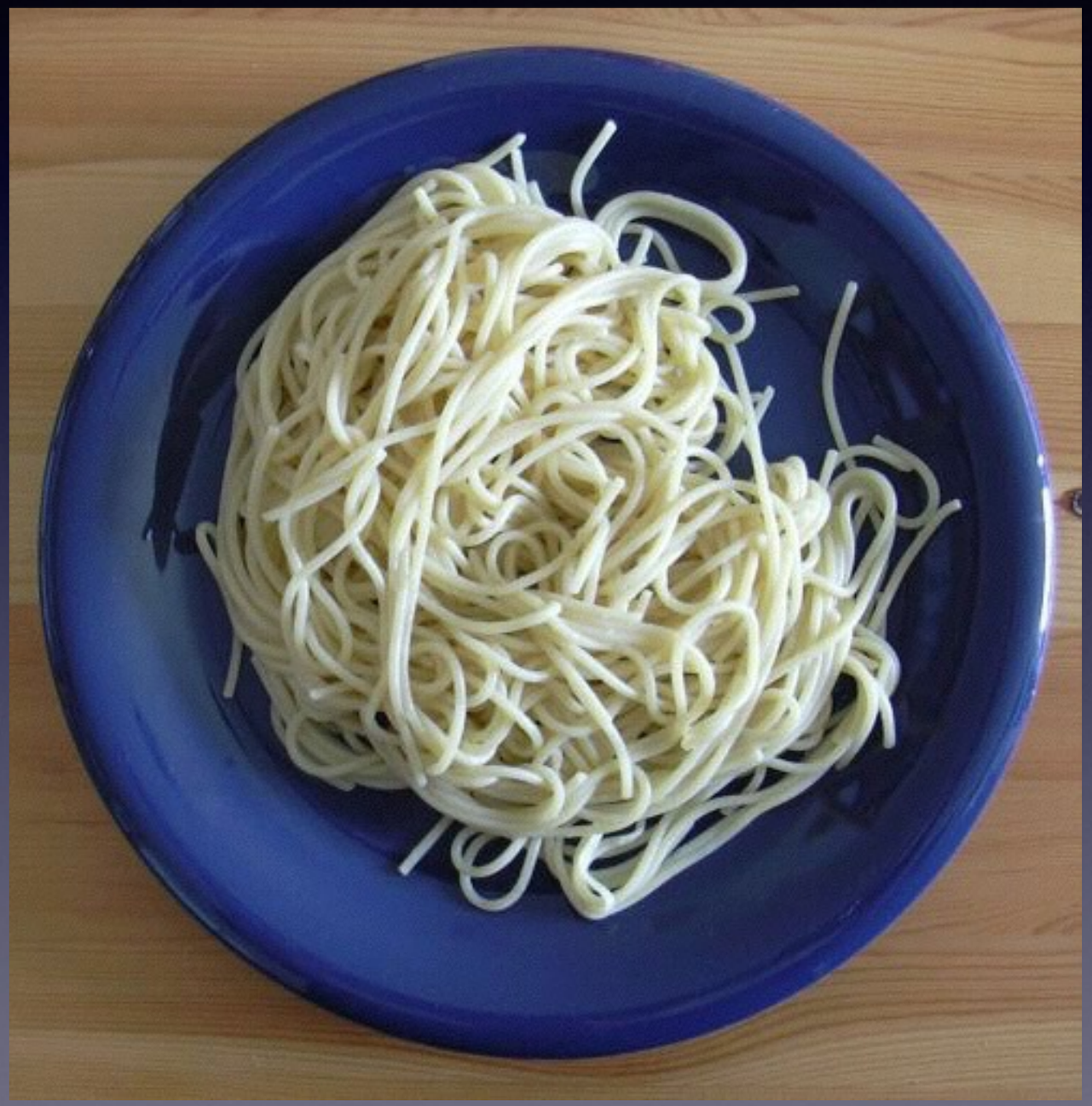
switch/if-else FSM

```
public class GhostAI : MonoBehaviour {

    enum State {scatter, attack, scared, gohome}
    State state = State.scatter;

    void Update () {

        switch (state) {

            case State.scatter:
                moveToHomeCorner();

                if (isAttackWave()) {
                    state = State.attack;
                } else if (seePoweredPacMan()) {
                    state = State.scared;
                }
                break;
```

v1

switch/if-else FSM

- Hard-coded AI
- Doesn’t scale well
- Spaghetti code
- Hard to change
- Hard to debug



Transition Tables

Current	Condition	Transition
Scatter	AttackWave	Attack
Attack	not AttackWave	Scatter
Scatter	Powered	Scared
Attack	Powered	Scared
Scared	not Powered	Scatter
Scared	Eaten	GoHome
GoHome	AtHome	Scatter

Transition Tables

- Store transitions as **data, not code**
- Implement states independently
- Generic FSM code to run agent
- Clean separation of states and transitions
- Table can be edited, e.g. by design tools

State Objects

- Encapsulate a state as an object
- Only state knows its behaviour and transitions
- Benefits of OOP…
    - complete separation of states
    - can add/replace states easily
    - subclass states to refine behaviours

v2

```
public class GhostAI : MonoBehaviour {

    State current = new ScatterState();

    void Update() {
        current.execute(this);
    }

    void changeState(State next) {
        current = next;
    }

    void reverseDirection() { ... }
    void moveToHomeCorner() { ... }
    bool isAttackWave() { ... }
    bool seePoweredPacMan { ... }
}
```

v2

```
public interface State {
    void execute(GhostAI ghost);
}
```

```
public class ScatterState : State {

    void execute(GhostAI ghost) {
        ghost.moveToHomeCorner();

        if (ghost.isAttackWave()) {
            ghost.changeState(new AttackState());

        } else if (ghost.seePoweredPacMan()) {
            ghost.changeState(new ScaredState());
        }
    }
}
```

v2

Enter/Exit Behaviours

```
public class ScatterState : State {

    void enter(GhostAI ghost) {
        ghost.reverseDirection();
    }

    void execute(GhostAI ghost) {

        ghost.moveToHomeCorner();

        if (ghost.isAttackWave()) {
            ghost.changeState(new AttackState());

        } else if (ghost.seePoweredPacMan()) {
            ghost.changeState(new ScaredState());
        }
    }

    void exit(GhostAI ghost) {}
}
```

```
public interface State {
    void enter(GhostAI ghost);
    void execute(GhostAI ghost);
    void exit(GhostAI ghost);
}
```

Define actions to be taken when state is entered and exited

v3

Enter/Exit Behaviours

```
public class GhostAI : MonoBehaviour {

    State current = new ScatterState();

    void Update() {
        current.execute(this);
    }

    void changeState(State next) {
        current.exit(this);
        current = next;
        current.enter(this);
    }

    void reverseDirection() { ... }
    void moveToHomeCorner() { ... }
    bool isAttackWave() { ... }
    bool seePoweredPacMan { ... }
}
```

v3

Generic States

```
public interface State {
    void enter(GhostAI ghost);
    void execute(GhostAI ghost);
    void exit(GhostAI ghost);
}
```

```
public interface State<A> {
    void enter(A agent);
    void execute(A agent);
    void exit(A agent);
}
```

Allow states to be used by different AIs.

v4

Generic States

```
public class GhostAI : MonoBehaviour {

    State<GhostAI> current = new ScatterState();

    void Update() {
        current.execute(this);
    }

    void changeState(State<GhostAI> next) {
        current.exit(this);
        current = next;
        current.enter(this);
    }

    void reverseDirection() { ... }
    void moveToHomeCorner() { ... }
    bool isAttackWave() { ... }
    bool seePoweredPacMan { ... }
}
```

v4

A State Machine Class

```
public class StateMachine<A> {

    A agent;
    State<A> current;

    public StateMachine(A a) { agent = a; }

    void getAgent() { return agent; }
    State getState() { return current; }
    void setState(State<A> s) { current = s; }

    void changeState(State<A> next) {
        current.exit(this);
        current = next;
        current.enter(this);
    }

    void update() {
        current.execute(agent, this);
    }
}
```

Separate FSM class used by multiple AIs.

v5

A State Machine Class

```
public class GhostAI : MonoBehaviour {

    StateMachine<GhostAI> fsm;

    public GhostAI() {
        fsm = new StateMachine(this);
        fsm.setState(new ScatterState());
    }

    public Update() {
        fsm.update();
    }

    void reverseDirection() { ... }
    void moveToHomeCorner() { ... }
    bool isAttackWave() { ... }
    bool seePoweredPacMan { ... }
}
```

v5

A State Machine Class

```
public class ScatterState : State<GhostAI> {

    void enter(GhostAI ghost) {
        ghost.reverseDirection();
    }

    void execute(GhostAI ghost, StateMachine fsm) {

        ghost.moveToHomeCorner();

        if (ghost.isAttackWave()) {
            fsm.changeState(new AttackState());

        } else if (ghost.seePoweredPacMan()) {
            fsm.changeState(new ScaredState());
        }
    }

    void exit(GhostAI ghost) {}
}
```

Both agent and FSM passed to state.

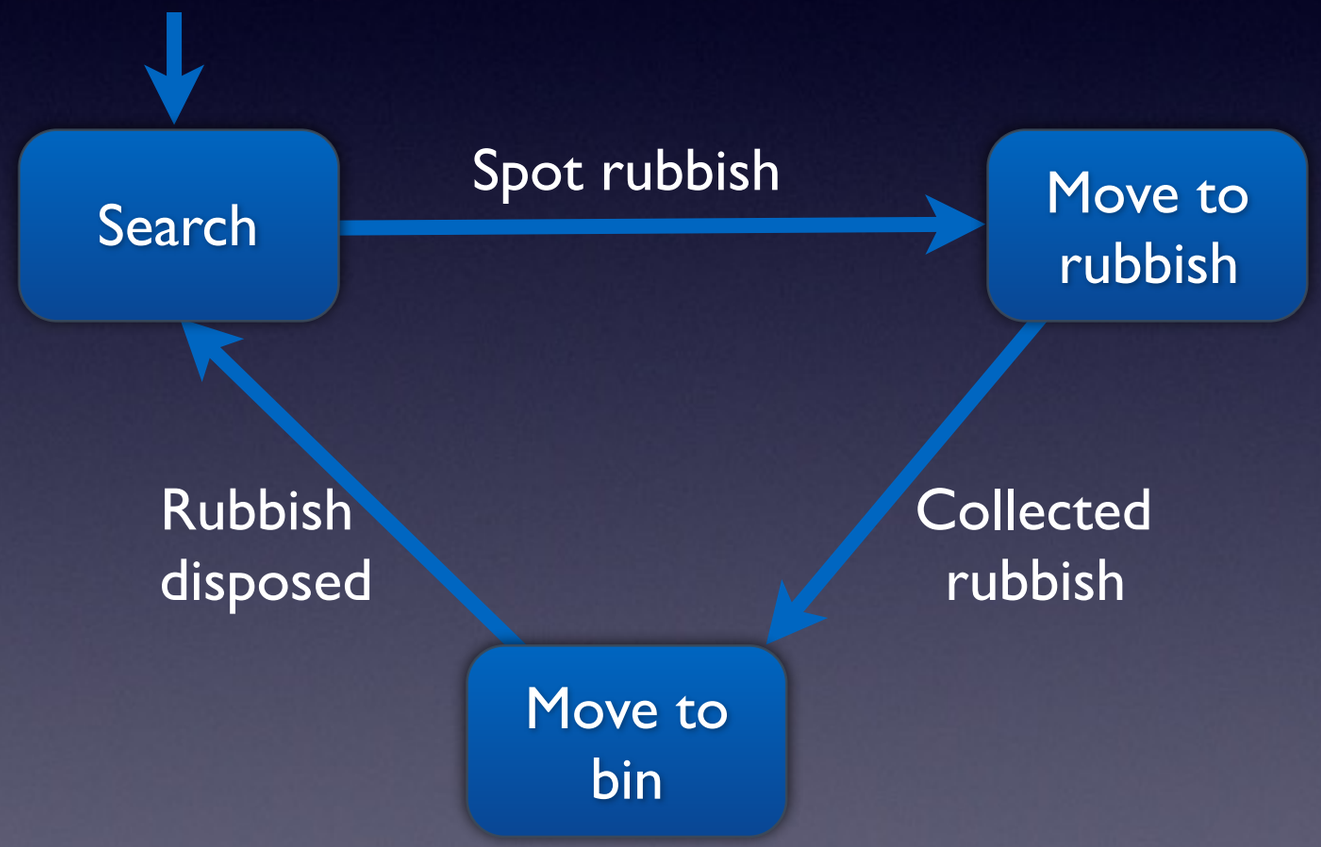
```
public interface State<A> {
    void enter(A agent);
    void execute(A agent, StateMachine s);
    void exit(A agent);
}
```

v5

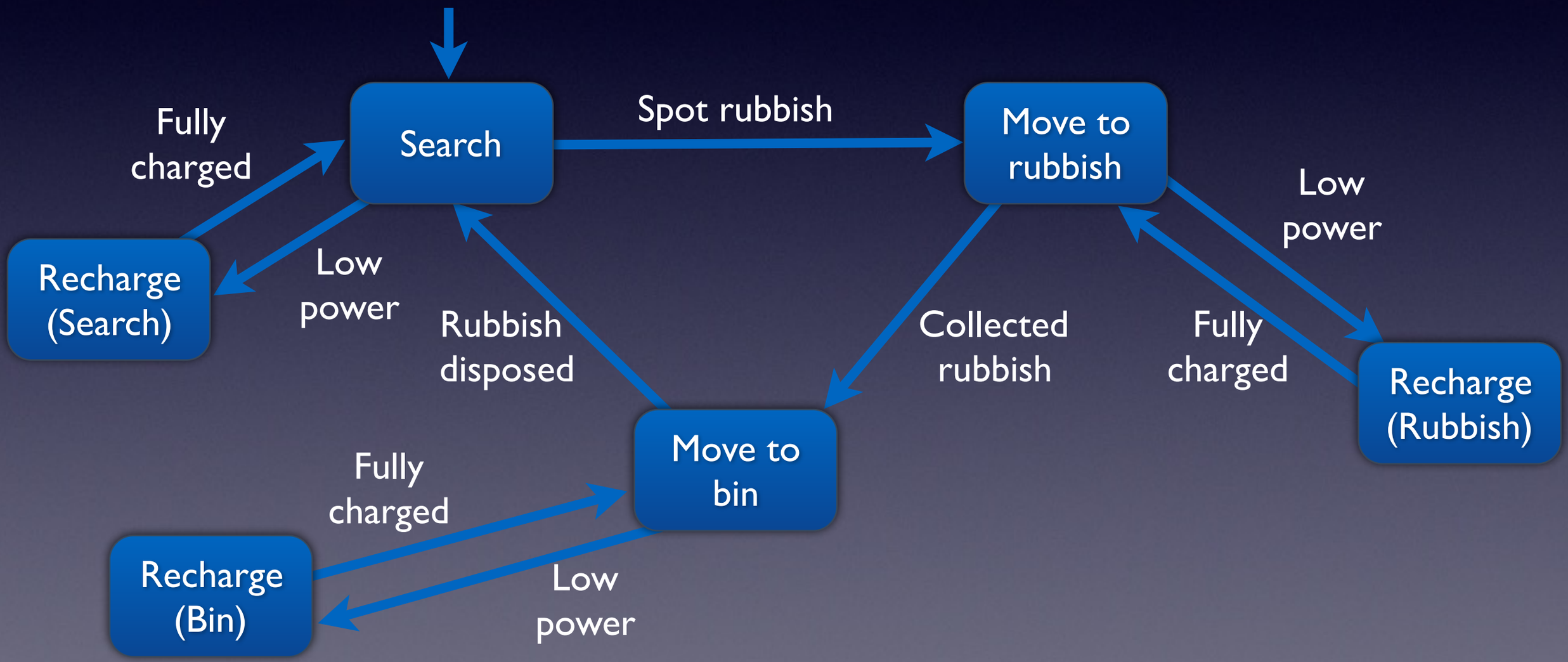
Global States

- Some agents will have transitions they always make, regardless of state
    - e.g. ghosts all stop when PacMan dies
- FSM can have a single global state
    - Never changes
    - Executed every AI tick

Robot Cleaner



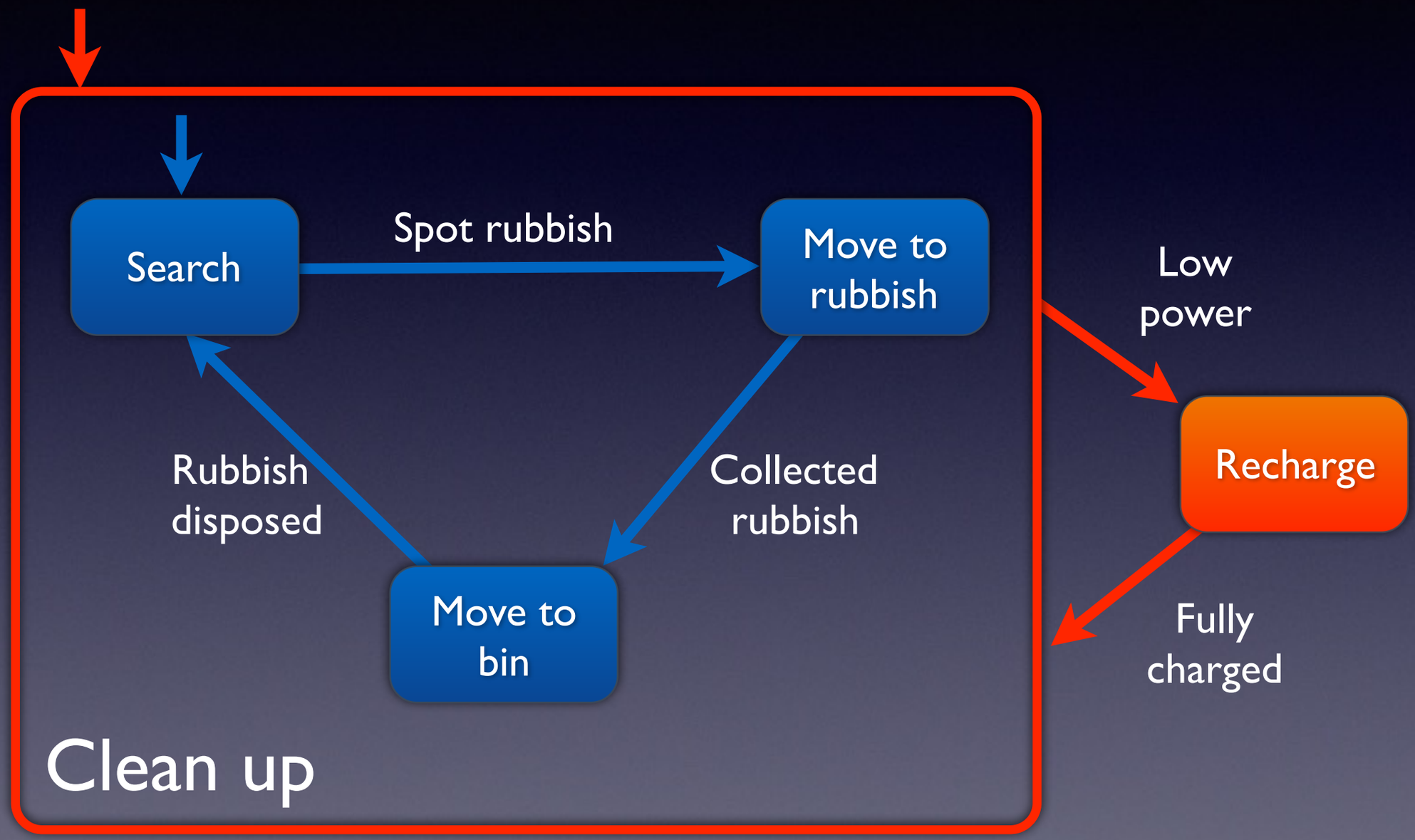
Robot Cleaner



Interruptions

- Need to resume leads to **duplicate states**
- Instead give FSM a memory
    - Store last state
    - Interrupt/resume methods
- For multiple nested interruptions use a stack

Hierarchical FSMs



Event-Driven FSMs

- Expensive to test conditions every AI tick
- Cheaper to react to messages passed from another agent or game object
- Soldier sends “shot you” to Target (who dies)
- Power pill times out and sends “done!” to ghost (who resumes attack)

Further Reading

- Buckland, Chap. 2: *State-Driven Agent Design*
- Millington, Section 5.3: *State Machines*
- PacMan AI: http://home.comcast.net/~jpittman2/pacman/pacmandossier.html