VGP332 – Artificial Intelligence

Instructor: Peter Chan

Agenda

- Assignment 2 Redux
- Finite State Machines
- State Design Pattern
- Messaging
- Assignment 3 Overview

Agenda

- Assignment 2 Redux
- Finite State Machines
- State Design Pattern
- Messaging
- Assignment 3 Overview

Assignment 1 Review

- Implicit vs. explicit graphs

Assignment 2 Redux

- Questions?

Agenda

- Assignment 2 Redux
- Finite State Machines
- State Design Pattern
- Messaging
- Assignment 3 Overview

States

States

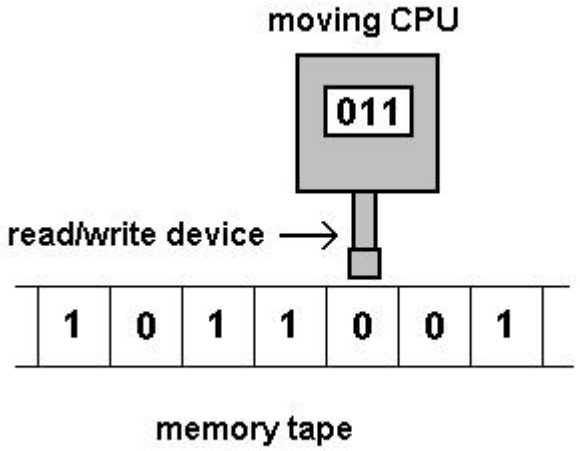
- A state is a unique configuration of information
  - Lightbulb on
  - Lightbulb off
- In games:
  - Doom?
  - Warcraft?
  - Others?

Finite State Machines

- Also called finite state automata
- Many applications outside computers:
  - Biology, math, logic, etc.
- Within comp. sci.:
  - Hardware design
  - Software engineering
  - Algorithm behaviour
  - Artificial intelligence
  - …

Turing Machine

- Finite state machine
- Operates on an infinitely long tape
- Has a tape head that can:
  - Move left
  - Move right
  - Read a symbol
  - Write a symbol

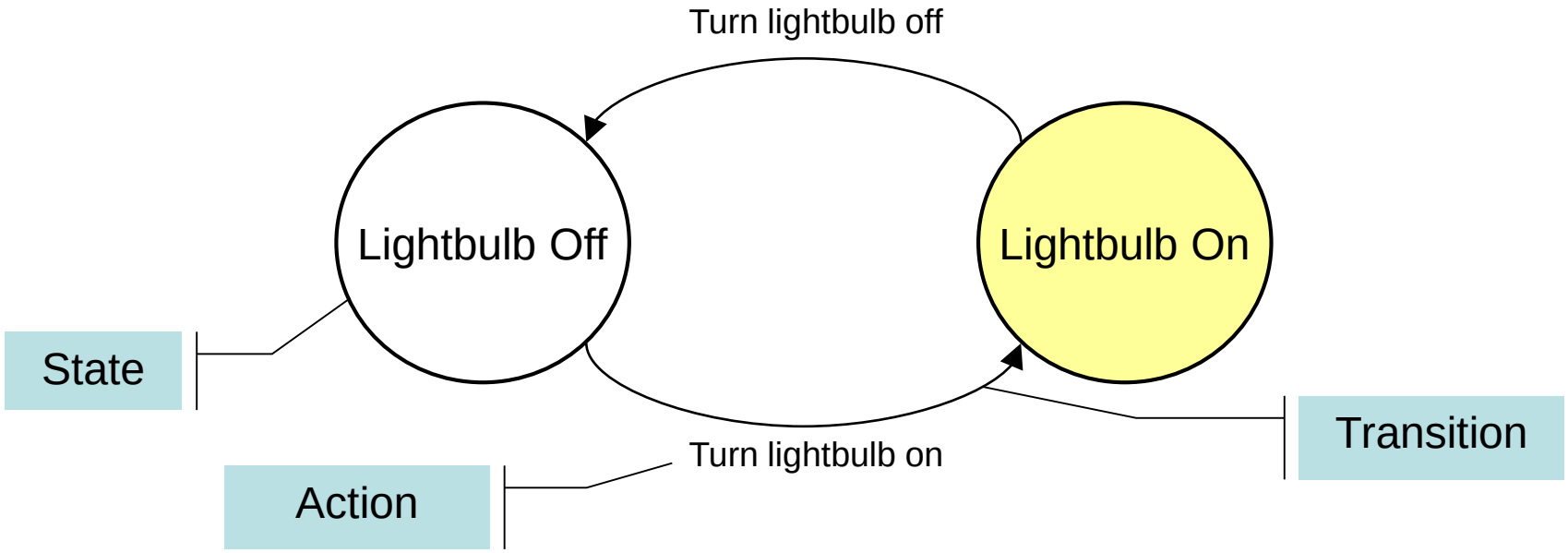


Turing Machine

- Can simulate *any* computer algorithm
- FSMs are a good model for AI behaviours

Finite State Machines

- Composed of:
  - A finite number of states
  - Transitions between those states
  - Actions

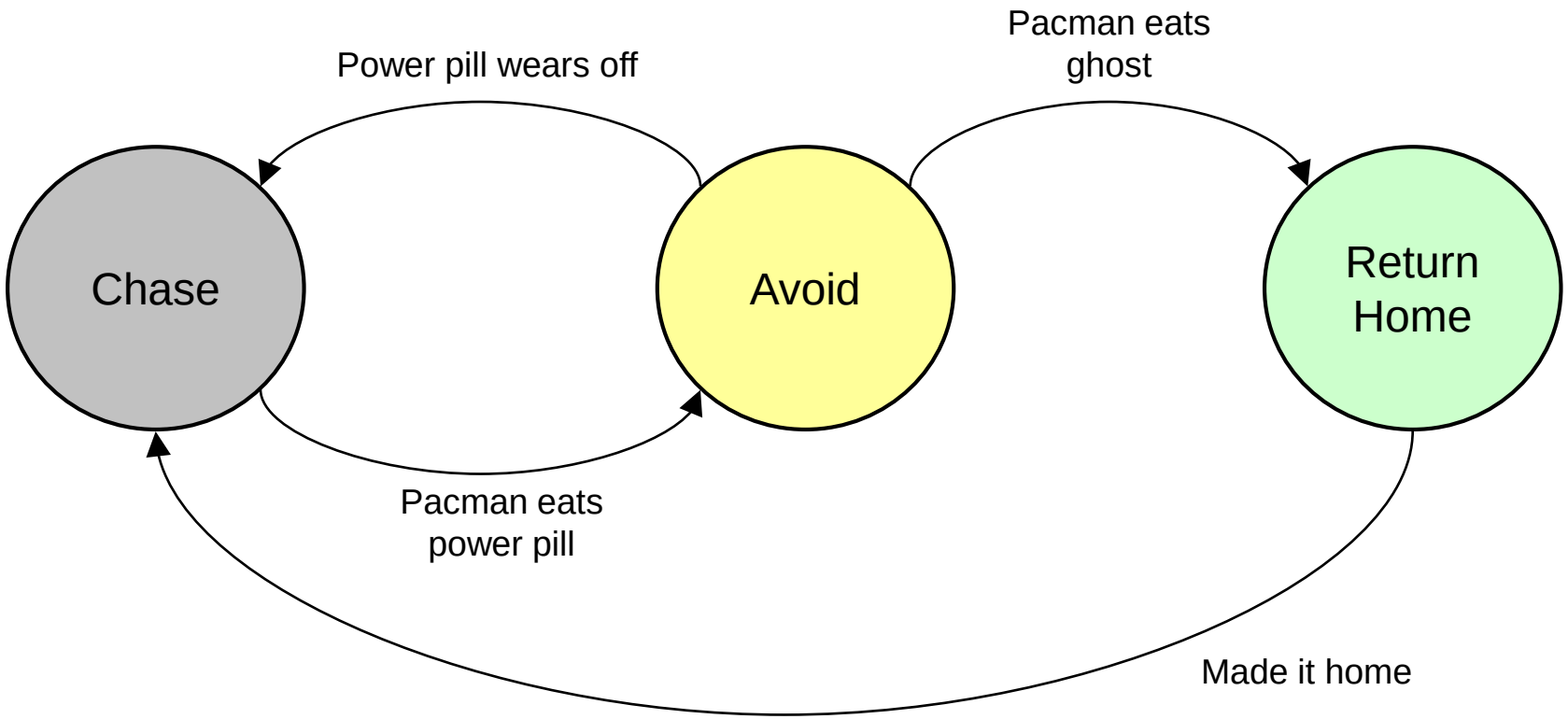


Play Pacman

http://www.thepcmanwebsite.com/media/pacman_flash/

- How many states do the ghosts have?

FSM for Ghosts in Pacman



switch-case Implementation

```
enum StateType { Chase, Avoid, ReturnHome };

void Ghost::UpdateState( StateType CurrentState )
{
    switch ( CurrentState )
    {
        case Chase:
            ChasePacman();
            if ( PowerPillEaten() )
                ChangeState( Avoid );
            break;
        case Avoid:
            AvoidPacman();
            if ( GhostEaten() )
                ChangeState( ReturnHome );
            if ( PowerPillTime() < 0 )
                ChangeState( Chase );
            break;
        case ReturnHome:
            ReturnToCenter();
            if ( GhostPosition() == HomePosition )
                ChangeState( Chase );
            break;
    }
}
```

switch-case Implementation

- Pros?
- Cons?

switch-case Implementation

- Pro: Quick to code up
- Pro: Handles simple cases sufficiently

- Con: Not extensible
- Con: Gets messy very fast

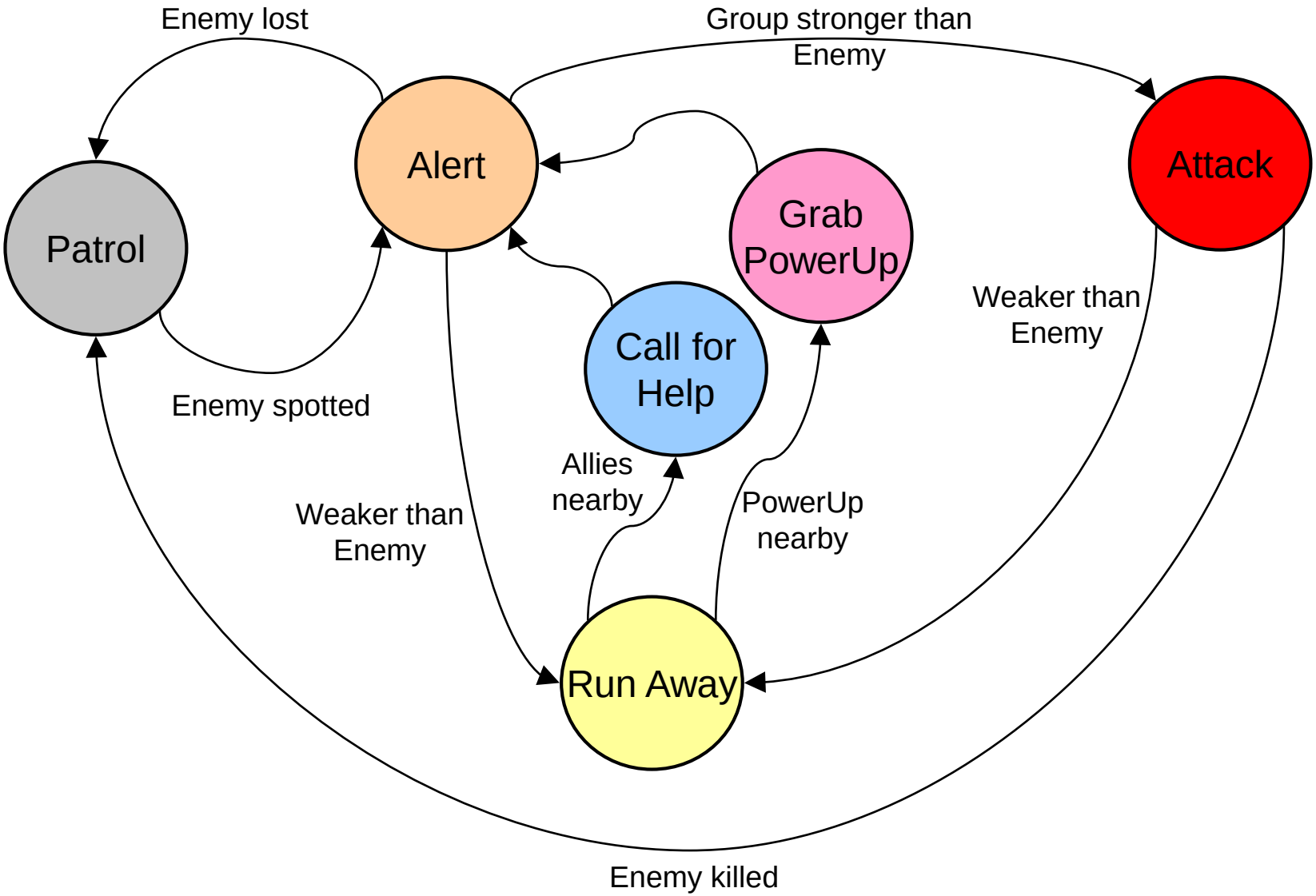
switch-case Implementation

- Pro: Quick to code up
- Pro: Handles simple cases sufficiently

- Con: Not extensible
- Con: Gets messy very fast

Don’t use in any but the simplest of games … and maybe not even then

FSM for Guard Patrol



State Transition Table

Current State	Condition	State Transition
Patrol	Enemy spotted	Alert
Alert	Enemy lost	Patrol
Alert	Stronger than Enemy	Attack
Alert	Weaker than Enemy	Run Away
Attack	Enemy killed	Patrol
Attack	Weaker than Enemy	Run Away
Run Away	Allies nearby	Call for Help
Run Away	PowerUp nearby	Grab PowerUp
Call for Help	-	Alert
Grab PowerUp	-	Alert

State Transition Table

- Can be queried by agent at regular intervals
- State transitions made based on table
- Conditions can encompass agent, other agents, environment, etc.
- Each state can be its own object external to the agent

State Transition Table

- Pros?
- Cons?

State Transition Table

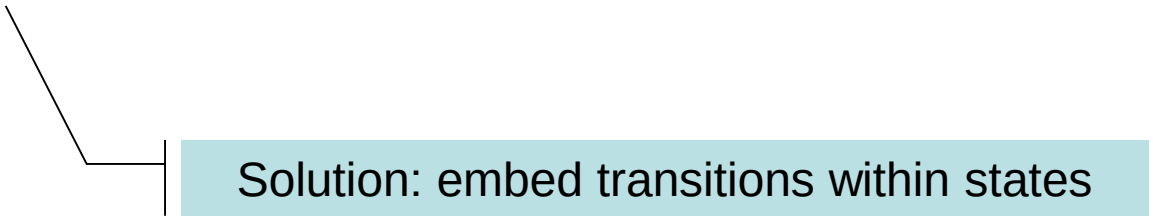
- Pros: Extensible, flexible architecture
- Pros: States can be external to the agent

- Cons: State transition table is internal to the agent

State Transition Table

- Pros: Extensible, flexible architecture
- Pros: States can be external to the agent

- Cons: State transition table is internal to the agent



Agenda

- Assignment 2 Redux
- Finite State Machines
- State Design Pattern
- Messaging
- Assignment 3 Overview

Embedded Rules

- All State objects derived from a pure virtual **`State`** base class:

```
class State
{
public:
    virtual void Execute( Agent *agent ) = 0;
};
```

- Derived State objects can be triggered by calling the **`Execute`** pure virtual method

Embedded Rules

- An Agent needs:
  - a pointer to its current state
  - a way to change its state

```
class Agent
{
private:
    State *m_pCurrentState;
public:
    void Update()
    {
        m_pCurrentState->Execute( this );
    }
    void ChangeState( const State *pNewState )
    {
        delete m_pCurrentState;
        m_pCurrentState = pNewState;
    }
};
```

Note: I'm omitting the usual attributes and methods an Agent class would have (e.g., health, position, etc.)

Embedded Rules

- An Agent needs:
  - a pointer to its current state
  - a way to change its state

Passing the Agent's pointer to the State allows the State to monitor the Agent's attributes and modify the Agent with specific behaviour

```
class Agent
{
private:
    State *m_pCurrentState;
public:
    void Update()
    {
        m_pCurrentState->Execute( this );
    }
    void ChangeState( const State *pNewState )
    {
        delete m_pCurrentState;
        m_pCurrentState = pNewState;
    }
};
```

Embedded Rules

- States can be completely customizable:

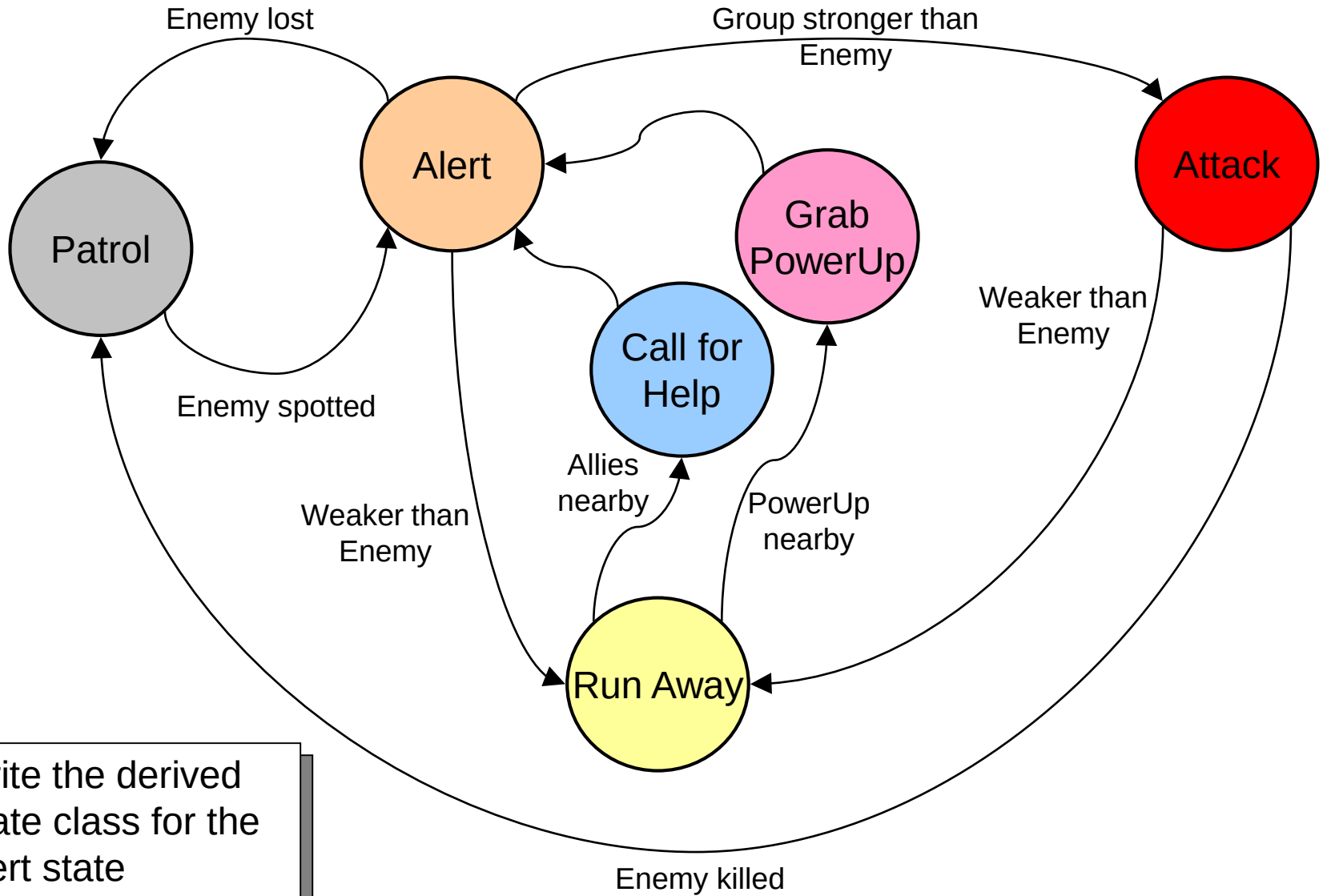
```
class State_RunAway : public State
{
public:
    void Execute( Agent *agent )
    {
        if ( agent->isCloseToAllies() )
        {
            agent->ChangeState( new State_CallForHelp );
        }
        else if ( agent->isCloseToPowerUp() )
        {
            agent->ChangeState( new State_GrabPowerUp );
        }
        else
        {
            agent->MoveAwayFromEnemy();
        }
    }
};
```

Embedded Rules

- States can be completely customizable:

```
class State_Attack : public State
{
public:
    void Execute( Agent *agent )
    {
        if ( agent->groupWeakerThanEnemy() )
        {
            agent->ChangeState( new State_RunAway );
        }
        else if ( agent->enemyKilled() )
        {
            agent->ChangeState( new State_Patrol );
        }
        else
        {
            agent->AttackEnemy();
        }
    }
};
```

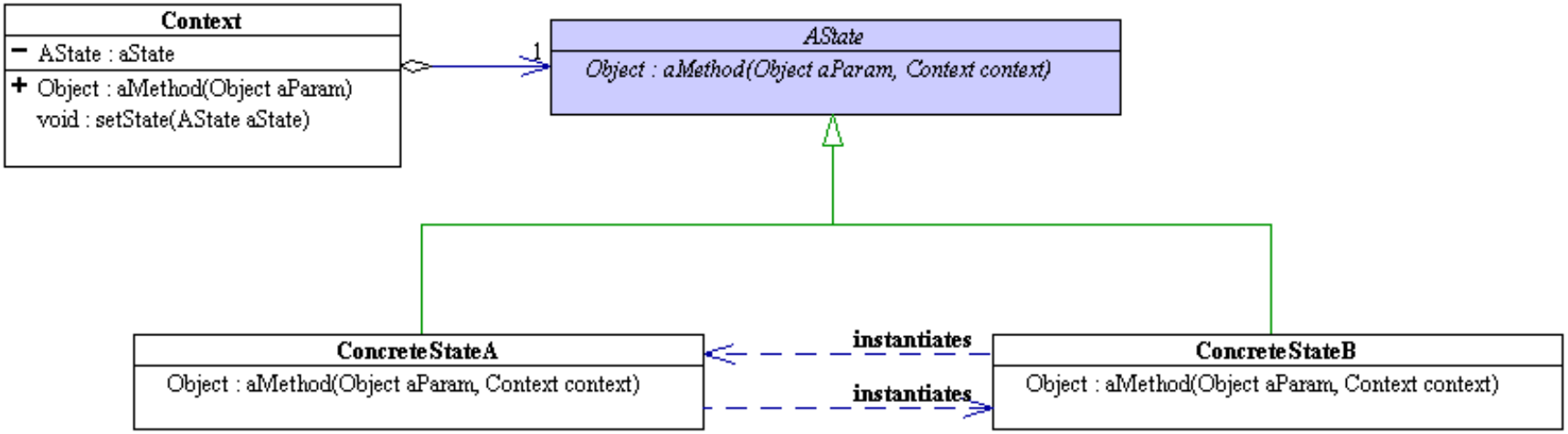
FSM for Guard Patrol



Write the derived State class for the Alert state

State Design Pattern

- This is the state design pattern



WestWorld Example

- Wild West setting
- Four locations
- Miner Bob is our agent

Gold Mine

Saloon

Miner Bob

Bank

Home

- From Programming Game AI by Example, Mat Buckland

WestWorld Example

- States = game locations

Gold Mine

EnterMineAndDigForNugget

Saloon

QuenchThirst

Bank

VisitBankAndDepositGold

Home

GoHomeAndSleepTillRested

WestWorld Example

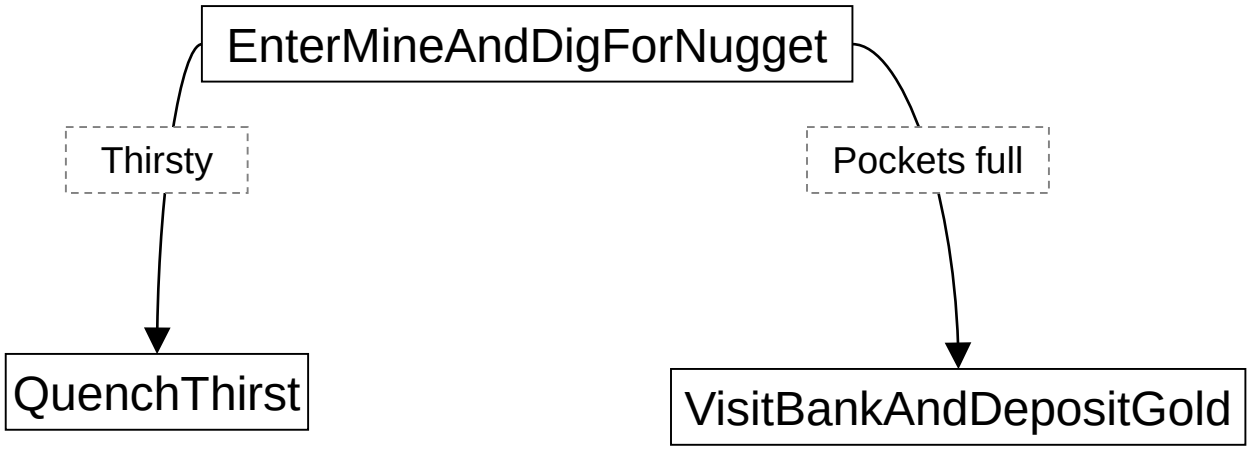
EnterMineAndDigForNugget

QuenchThirst

VisitBankAndDepositGold

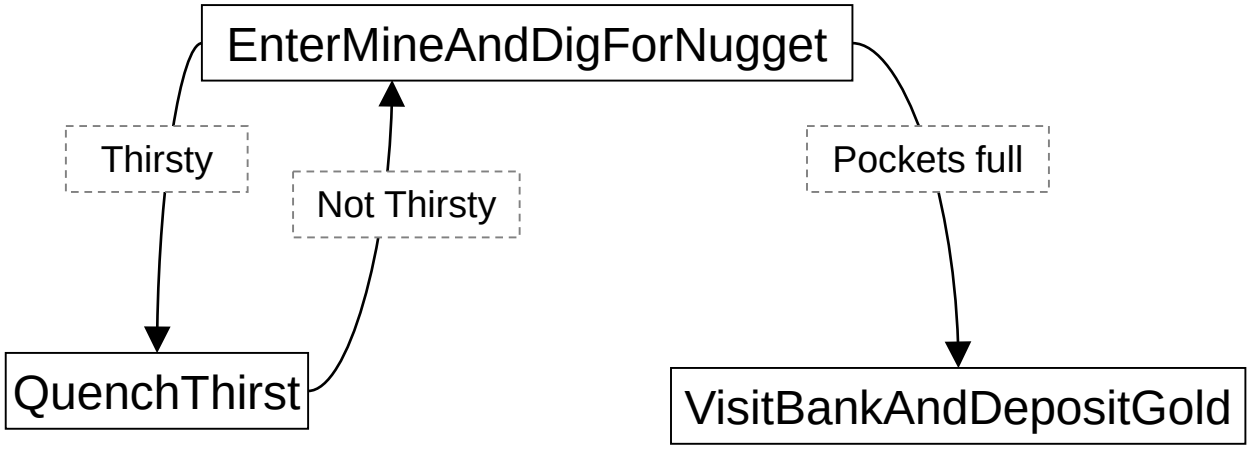
GoHomeAndSleepTillRested

WestWorld Example



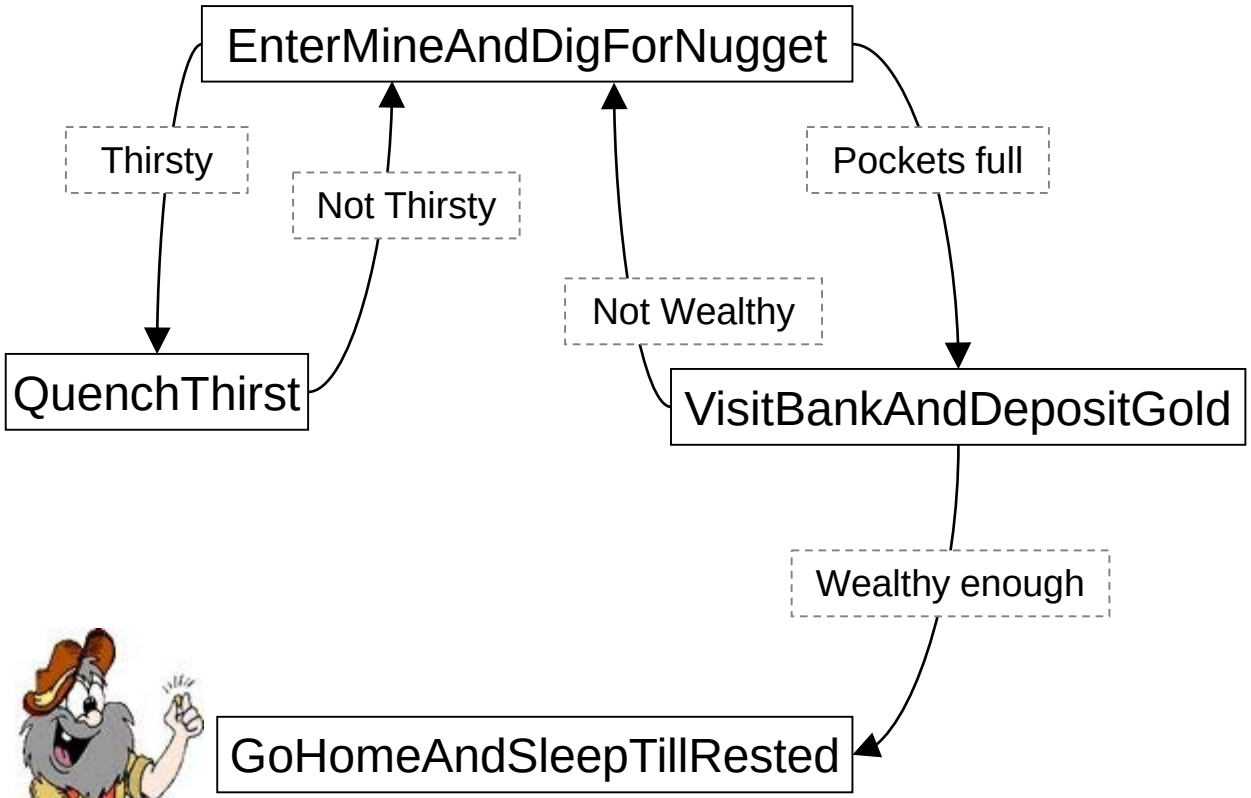
GoHomeAndSleepTillRested

WestWorld Example

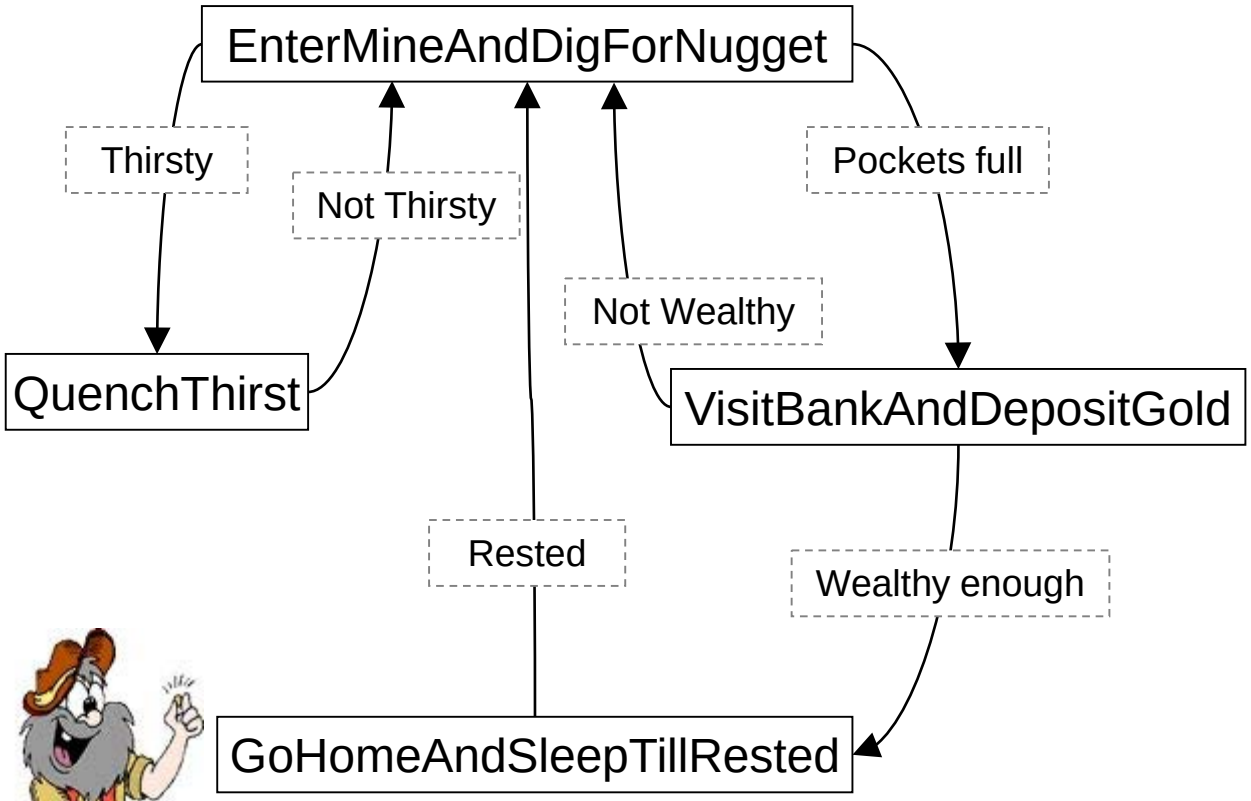


GoHomeAndSleepTillRested

WestWorld Example



WestWorld Example



WestWorld Example

- All WestWorld inhabitants have base class

```
class BaseGameEntity
{
private:
    int m_ID;    // Every entity has a unique identifying number
    static int m_iNextValidID;    // Next valid ID

    // Verifies that value passed to the method >= next valid ID
    // If so, set ID and increment to next valid ID
    void SetID( int val );

public:
    BaseGameEntity( int id ) { SetID( id ); }
    virtual ~BaseGameEntity() {}

    // All entities must implement an update function
    virtual void Update() = 0;

    int ID() const { return m_ID; }
};
```

WestWorld Example

- Miner class, derived from **`BaseGameEntity`**

```
class Miner : public BaseGameEntity
{
private:
    State * m_pCurrentState;    // Pointer to an instance of a State
    location_type m_Location;   // Miner's current location
    int m_iGoldCarried;         // # of nuggets in miner's pocket
    int m_iMoneyInBank;         // Amount of money in bank
    int m_iThirst;              // Higher value = thirstier miner
    int m_iFatigue;             // Higher value = miner more tired

public:
    Miner( int ID );
    void Update();
    void ChangeState( State *pNewState );  // Change to new state

    /* rest of interface omitted */
};
```

WestWorld Example

- **`Update`** class, called every “frame”

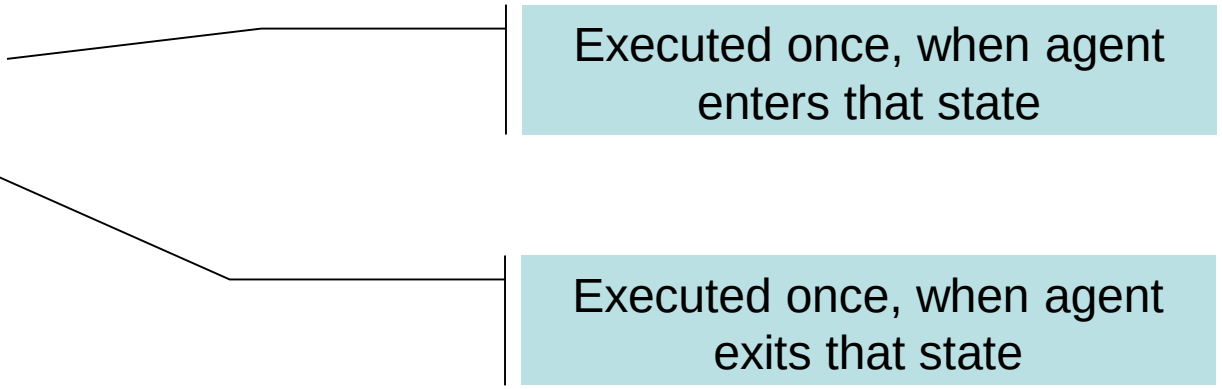
```
void Miner::Update()
{
    // Miner’s always getting thirsty
    m_iThirst += 1;

    // Execute state logic
    if ( m_pCurrentState )
    {
        m_pCurrentState->Execute( this );
    }
}
```

State Design Pattern

- Pros?
- Cons?

State Improvements

- States can be made more flexible with additional action states:
  - Enter
  - Exit



- Examples?

State Improvements

- Add those methods to the **`State`** base class:

```
class State
{
public:
    virtual ~State() {}

    // This will execute when the state is entered
    virtual void Enter( Miner * ) = 0;

    // This is called by the miner's update function
    virtual void Execute( Miner * ) = 0;

    // This will execute when the state is exited
    virtual void Exit( Miner * ) = 0;
};
```

Back to the Miner Class

- Implement the **`ChangeState`** method

```
void Miner::ChangeState( State *pNewState )
{
    // Both states need to be valid
    assert ( m_pCurrentState && pNewState );

    // Call the exit method of the existing state
    m_pCurrentState->Exit( this );

    // Change state to the new state
    m_pCurrentState = pNewState;

    // Call the entry method of the new state
    m_pCurrentState->Enter( this );
}
```

EnterMineAndDigForNugget

- Miner should change location to gold mine
- Once at gold mine, dig for gold
- When pockets full, change state to VisitBankAndDepositNugget
- If miner gets thirsty, change state to QuenchThirst

EnterMineAndDigForNugget

```
class EnterMineAndDigForNugget : public State
{
private:
    EnterMineAndDigForNugget() {}
    /* copy ctor and assignment op omitted */

public:
    // This is a singleton
    static EnterMineAndDigForNugget *Instance();

    virtual void Enter( Miner *pMiner );

    virtual void Execute( Miner *pMiner );

    virtual void Exit( Miner *pMiner );
};
```

Singleton Pattern

- Used when you want only one instance of a class
- Usually this object coordinates actions across the application
- Less messy than global variables

Singleton Pattern

- Examples in games:
  - Render manager
  - NIS manager
  - Load scheduler
  - Front End
  - …
  - …

Singleton Pattern

- Singleton class declaration

```
class Singleton
{
public:
   static Singleton* Instance();
protected:
   Singleton();
   Singleton(const Singleton&);
   Singleton& operator= (const Singleton&);
private:
   static Singleton* pinstance;
};
```

From http://www.inquiry.com/techtips/cpp_pro/10min/10min0200.asp

Singleton Pattern

- Singleton class implementation

```
Singleton* Singleton::pinstance = 0;  // initialize pointer
Singleton* Singleton::Instance ()
{
   if (pinstance == 0) // is it the first call?
   {
      pinstance = new Singleton; // create sole instance
   }
   return pinstance; // address of sole instance
}

Singleton::Singleton()
{
   //... perform necessary instance initializations
}
```

From http://www.inquiry.com/techtips/cpp_pro/10min/10min0200.asp

Singleton Pattern

- All of the following return the same instance:

```
Singleton *p1 = Singleton::Instance();
Singleton *p2 = p1->Instance();
Singleton & ref = * Singleton::Instance();
```

EnterMineAndDigForNugget

```
class EnterMineAndDigForNugget : public State
{
private:
    EnterMineAndDigForNugget() {}
    /* copy ctor and assignment op omitted */

public:
    // This is a singleton
    static EnterMineAndDigForNugget *Instance();

    virtual void Enter( Miner *pMiner );

    virtual void Execute( Miner *pMiner );

    virtual void Exit( Miner *pMiner );
};
```

EnterMineAndDigForNugget

```
void EnterMineAndDigForNugget::Enter( Miner *pMiner )
{
    // If the miner is not already located at the gold mine,
    // he must change location to the gold mine
    if ( pMiner->Location() != goldmine )
    {
        cout << “\n” << GetNameOfEntity( pMiner->ID() ) << “: “
             << “Walkin’ to the gold mine”;

        pMiner->ChangeLocation( goldmine );
    }
}
```

EnterMineAndDigForNugget

```
void EnterMineAndDigForNugget::Execute( Miner *pMiner )
{
    // Dig one nugget of gold
    pMiner->AddToGoldCarried(1);

    // diggin' is hard work
    pMiner->IncreaseFatigue();

    cout << "\n" << GetNameOfEntity( pMiner->ID() ) << ": "
         << "Pickin' up a nugget";

    // if enough gold mined, go and put it in the bank
    if ( pMiner->PocketsFull() )
    {
        pMiner->ChangeState( VisitBankAndDepositGold::Instance() );
    }

    // if thirsty go and get a whiskey
    if ( pMiner->Thirsty() )
    {
        pMiner->ChangeState( QuenchThirst::Instance() );
    }
}
```

EnterMineAndDigForNugget

```
void EnterMineAndDigForNugget::Exit( Miner *pMiner )
{
    cout << “\n” << GetNameOfEntity( pMiner->ID() ) << “: “
         << “Ah’m leavin’ the gold mine with mah pockets full o’ gold”;
}
```

WestWorld Code

- Browse through the code to see how this all fits together
- Try it out

State Improvements

```
class State
{
public:
    virtual ~State() {}

    // This will execute when the state is entered
    virtual void Enter( Miner * ) = 0;

    // This is called by the miner's update function
    virtual void Execute( Miner * ) = 0;

    // This will execute when the state is exited
    virtual void Exit( Miner * ) = 0;
};
```

What if you had other inhabitants, like a bandit and a sheriff?

State Improvements

- Use a templated class:

```
template <class entity_type>
class State
{
public:
    virtual ~State() {}

    // This will execute when the state is entered
    virtual void Enter( entity_type * ) = 0;

    // This is called by the entity's update function
    virtual void Execute( entity_type * ) = 0;

    // This will execute when the state is exited
    virtual void Exit( entity_type * ) = 0;
};
```

EnterMineAndDigForNugget

- Each state for the miner entity is now derived from an appropriately templated **`State`**

```
class EnterMineAndDigForNugget : public State<Miner>
{
    /* OMITTED */
};
```

- The **`State`** base class is now very reusable

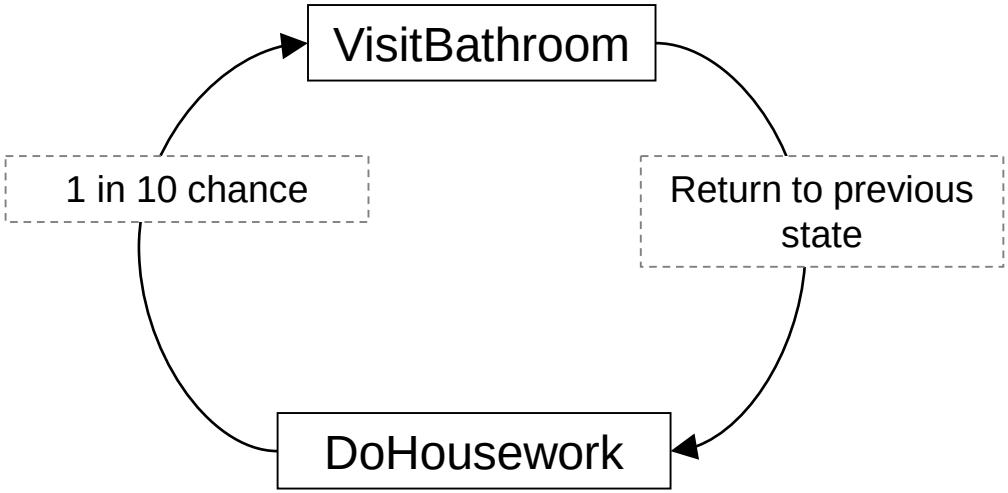
Global States

- Usually a number of states need to be accessible from every state
  - Sims: going to the bathroom
  - Bots: needing health
  - Others?
- Rather than duplicate code, use a global state
- And usually, after finishing with the global state, the agent returns to previous task

Adding Global State to Miner

```
class Miner : public BaseGameEntity
{
private:
    State<Miner> * m_pCurrentState;
    State<Miner> * m_pPreviousState;
    State<Miner> * m_pGlobalState;

    /* ... */

public:
    void ChangeState( State<Miner> * pNewState );
    void RevertToPreviousState();

    /* ... */
};
```

Miner’s Wife



- Global state not shown
- Triggers the 1 in 10 chance from any state
- VisitBathroom reverts to previous state

One More Improvement

- Create a **`StateMachine`** class template
  - Delegates state machine management from agent to separate class

Look at code!

WestWorld with Two Agents

- Try it out

Finite State Machines

- Pros?
- Cons?

Finite State Machines

- Pro: Quick to implement
- Pro: Predictable, i.e., easy to debug
- Pro: Relatively flexible
- Pro: Easy mapping from design to implementation

- Con: Deterministic, i.e., predictable
- Con: Large, complex systems can be messy
- Con: States are relatively distinct

Agenda

- Assignment 2 Redux
- Finite State Machines
- State Design Pattern
- Messaging
- Assignment 3 Overview

Communication in Code

- How do sections of code communicate to each other?
  - Control flow
  - Callbacks
  - Polling
  - Events

Messaging

- Have you used / seen messaging before?

Messaging

- Also called event-handling or event-driven architecture
- In AI:
  - Game agents can use messages to communicate with the world
  - Game agents can use messages to communicate with each other

WestWorld Messaging

- Miner sends “Honey, I’m Home” when he returns home
- Elsa receives “Honey, I’m Home”, stops what she’s doing, changes state to CookStew
- In the CookStew state, Elsa sends a delayed “Stew Ready” message to herself
- Elsa receives the “Stew Ready” message some ‘frames’ later
- Elsa sends the “Stew Ready” message to Miner Bob
- If Miner Bob is in the GoHomeAndSleepTillRested state, he changes state to EatStew
- Otherwise message is dismissed

New WestWorld FSM Diagrams

- What would the new WestWorld FSM diagrams look like?

Message Classes

- Telegram
  - Standard message class

Look at code!

Message Classes

- EntityManager
  - “Phone book” of game entities to enable message passing

Look at code!

Message Classes

- MessageDispatcher
  - Routes messages between entities correctly

Look at code!

Message Classes

- Handling messages
  - Try current state first
  - If message not handled, try global state
- Code modified:
  - BaseGameEntity
  - State
  - StateMachine

Look at code!

WestWorld Messaging

- Try it out

Messaging

- Pros?
- Cons?

Agenda

- Assignment 2 Redux
- Finite State Machines
- State Design Pattern
- Messaging
- Assignment 3 Overview