Datastructures and Algorithms GD

HearthStats

HVA

7 oktober 2017

Sjors Gielen (500765899)

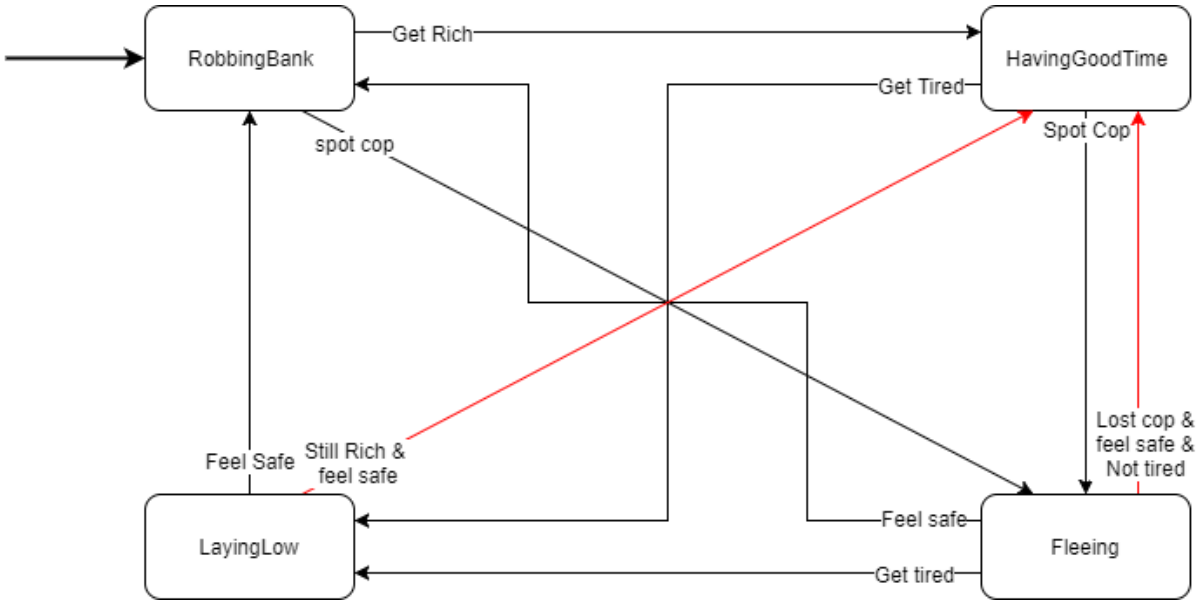
Datastructures and Algorithms GD

HearthStats

Question 1:

Add two new state transitions to the diagram above. Name these transitions and draw out the new diagram:

The two new state transitions are found in red.



Question 2:

Implement your new state machine using switch statements, a state transition table or the state design pattern. Build it in such a way that the user can change the NPC's state by writing a string (get rich, spot cop, etc.) in the Console.

I made use of a state design pattern. It works but I feel that there are some ways to make the design better.

I'll first show the interfaces of the relevant underlying classes. Followed by the classes themselves and afterwards I will show the implementation of the classes.

StateInterface:

```
namespace Datastruct_and_algo_excersizes.StateMananger
{
    interface StateInterface<T>
    {
        string _stateName { get; }

        /*
         * Called if the state is being evaulted.
         * Use the Agent param to acces variables from the object this state belongs
too
         * if a stateChange needs to happen, return true
```

```
         * Use the changeStateToo param to tell to which state needs to be switched
         */
        bool EvaluateAgent(T agent, out State<T> changeStateToo);

        /*
         * Called when entering this state
         * Use the prevState param to custimize behavior based on the prevState
         */
        void OnEnterState(State<T> prevState);

        /*
         * Called when leaving this state, end state's don't have to implement this
         * Use the nextState param to custimize behavior based on the nextState
         */
        void OnExitState(State<T> nextState);

        /*
         * Called if no state transsition occurs on this evaluation
         * Passes in the agent so it may be modified.
         */
        void OnStayInState(T agent);
    }
}
```

StateManagerInterface:

```
using System.Collections.Generic;

namespace Datastruct_and_algo_excersizes.StateMananger
{
    interface StateManagerInterface<T>
    {
        /*
         * Whether or not the stateMachine has reached an end-state state
         */
        bool _isInEndState { get ; }

        /*
         * Add a state into the state machine, required for validating the state
machine.
         * Returns true if adding was succesfull
         * Returns false if the state._stateName already exsists in the state machine
         * Can throw StateManagerAlreadyActiveException
         */
        bool AddState(State<T> newState, bool isStartState = false);


        /*
         * Changes the state too the param state.
         * Cals the current states OnExitState method and the new states OnEnterState
method
         */
        void ChangeState(State<T> stateToChangeToo);

        /*
         * Disable's the state machine to allow for changes.
         */
        void DisableStateMachine();

        /*
         * Validates the state machine. If validation was succesfull returns true
         * A valid state machine has a startState and has every state be reachable AND
         * has no reachable states not be added too the stateManager
```

```
         * Returns true if the stateMachine is valid
         * Returns false if the stateMachine is invalid
         * Note: Does NOT reset the currentState too startState if the stateMachine is
already valid!
         */
        bool EnableStateMachine();

        /*
         * Execute the current state's logic via it's EvalueteAgent method
         * If Evaluate Agent returns true the currentState of the StateManager will
change via the ChangeState method
         * Note: If a state change is enacted the currentState when the method is
called will NOT execute it's RemainInState method.
         * Can throw StateManagerNotValidatedException
         */
        void ExecuteCurrentState();

        /*
         * Returns all states included in the state manager
         */
        List<State<T>> GetAllStates();

        /*
         * Returns all states without an exit state
         */
        List<State<T>> GetEndStates();

        /*
         * Returns all reachable states(including states that may not be included in
the state mannager)
         */
        List<State<T>> GetReachableStates(State<T> startState);

        /*
         * Returns all unreachable states, great for debugging a state machine
         */
        List<State<T>> GetUnreachableStates(State<T> StartState);
    }
}
```

State<T> Class

```
namespace Datastruct_and_algo_excersizes.StateMananger
{
    /* Improvements:
     * Alter the state class to no longer require instance of T
     * and exitStates
     */
    abstract class State<T> : StateInterface<T>
    {
        public Dictionary<string, State<T>> exitStates;
        private string stateName;

        public string _stateName
        {
            get { return this.stateName; }
        }

        public State(string name)
        {
            this.exitStates = new Dictionary<string, State<T>>();
```

```
            this.stateName = name;
        }

        public virtual bool EvaluateAgent(T agent, out State<T> changeStateToo)
        {
            changeStateToo = null;
            return false;
        }

        public virtual void OnExitState(State<T> nextState)
        {

        }

        public virtual void OnEnterState(State<T> prevState)
        {

        }

        public virtual void OnStayInState(T agent)
        {

        }
        public void AddExitState(State<T> stateToAdd)
        {
            this.exitStates.Add(stateToAdd._stateName, stateToAdd);
        }

    }
}
```

StateMananger Class

```
namespace Datastruct_and_algo_excersizes.StateMananger
{

    /* Improvements:
     * Alter the StateMananger to evaluate the agents variables and then dictate state
changes.
     * Every evaluation a stateChange should occur
     * Keep track of which state can lead into which within the mananger
     */
    class StateManager<T> : StateManagerInterface<T>
    {
        Dictionary<string, State<T>> myStates;
        State<T> startState;
        State<T> currentState;
        T agent;
        private List<State<T>> endStates;
        private bool isInEndState;
        bool isValidStateMachine = false;

        public bool _isInEndState {  get { return isInEndState; } }

        internal State<T> _currentState { get { return currentState; } }

        public StateManager(T agent)
        {
            myStates = new Dictionary<string, State<T>>();
            this.agent = agent;
        }
```

```
        public bool AddState(State<T> newState, bool isStartState = false)
        {
            if(isValidStateMachine)
            {
                throw new StateManagerAlreadyActiveException("State machine has
already been validated and may not be edited anymore, " +
                    "If intended please disable the stateMachine first");
            }

            if(myStates.ContainsKey(newState._stateName))
            {
                return false;
            }

             myStates.Add(newState._stateName, newState);

            if (isStartState)
            {
                startState = newState;
            }
            return true;
        }

        public void ExecuteCurrentState()
        {
            if(!isValidStateMachine)
            {
                throw new StateManagerNotValidatedException("State machine has not
been validated yet. Did you forget to validate the statemachine?");
            }

            State<T> changeState;
            if (this.currentState.EvaluateAgent(this.agent, out changeState))
            {
                this.ChangeState(changeState);
            }
            else
            {
                if(!this.isInEndState)//once we know we are in an endState we won't be
able to move out of it anyway
                    this.isInEndState = this.endStates.Contains(currentState);
                currentState.OnStayInState(this.agent);
            }

        }

        public void ChangeState(State<T> stateToChangeToo)
        {
            var prevState = currentState;
            currentState.OnExitState(stateToChangeToo);
            currentState = stateToChangeToo;
            currentState.OnEnterState(prevState);
        }


        public List<State<T>> GetAllStates()
        {
            List<State<T>> allStates = new List<State<T>>();
            foreach(KeyValuePair<string, State<T>> keyValuePair in myStates)
            {
                allStates.Add(keyValuePair.Value);
```

```
            }
            return allStates;
        }

        public List<State<T>> GetUnreachableStates(State<T> StartState)
        {
            List<State<T>> unreachableStates = GetAllStates();

            List<State<T>> discoveredStates = new List<State<T>>
            {
                StartState
            };
            if (!unreachableStates.Contains(StartState))
            {
                throw new StateNotIncludedException("The state " + StartState + " is
not a part of this StateManager");
            }
            unreachableStates.Remove(StartState);


            while (discoveredStates.Count != 0)
            {
                State<T> investigatingState = discoveredStates[0];//get a state we
know off
                discoveredStates.Remove(investigatingState);
                foreach(KeyValuePair<string, State<T>> foundState in
investigatingState.exitStates)
                {
                    if(unreachableStates.Contains(foundState.Value))//if the state
found is still considered unreachable
                    {
                        unreachableStates.Remove(foundState.Value);//make it no longer
unreachalble
                        discoveredStates.Add(foundState.Value);//and add it too the
discovered states
                    }
                }
            }

            return unreachableStates;

        }

        public List<State<T>> GetReachableStates(State<T> startState)
        {
            List<State<T>> reachedStates = new List<State<T>>
            {
                startState
            };
            List<State<T>> discoveredStates = new List<State<T>>
            {
                startState
            };
            while (discoveredStates.Count != 0)
            {
                State<T> investigatingState = discoveredStates[0];//get a state we
know off
                discoveredStates.Remove(investigatingState);
                foreach(KeyValuePair<string,State<T>> foundState in
investigatingState.exitStates)
                {
```

```
                    if(!reachedStates.Contains(foundState.Value))//if the state is not
in the reached states yet
                    {
                        reachedStates.Add(foundState.Value);//Add it too the reached
states
                        discoveredStates.Add(foundState.Value);//and add it too the
discovered states
                    }
                }
            }
            return reachedStates;
        }

        public List<State<T>> GetEndStates()
        {
            List<State<T>> endStates = new List<State<T>>();
            foreach(KeyValuePair<string, State<T>> stateNamePair in myStates)
            {
                if(stateNamePair.Value.exitStates.Count == 0)
                {
                    endStates.Add(stateNamePair.Value);
                }
            }
            return endStates;
        }

        private bool AreAllReachableStatesInStateMachine()
        {
            List<State<T>> reachableStates = GetReachableStates(this.startState);
            foreach(State<T> reachableState in reachableStates)
            {
                if (!this.myStates.ContainsKey(reachableState._stateName))//if any
reachable state doesn't exsist in the dictionary, return false.
                {
                    return false;
                }
            }
            return true;
        }

        private bool AreAllStatesReachable()
        {
            return GetUnreachableStates(this.startState).Count < 1;
        }

        public bool EnableStateMachine()
        {
            if(this.isValidStateMachine)//already validated, don't do it again.
            {
                return this.isValidStateMachine;
            }

            if(this.AreAllStatesReachable() &&
this.AreAllReachableStatesInStateMachine())
            {
                this.currentState = this.startState;
                this.endStates = GetEndStates();//cache the result in a list so we
don't need to run this expensive method over and over again.
                this.isValidStateMachine = true;
            }
            return this.isValidStateMachine;
        }
```

```
        public void DisableStateMachine()
        {
            this.isValidStateMachine = false;
            this.currentState = this.startState;
        }


    }
}
```

And now to implement the stateMachine using these base-classes. First I made all the states and their internal logic:

```
namespace Datastruct_and_algo_excersizes
{
    class InputRobbinBankState<Robber> : State<Robber>, StateInterface<Robber>
        where Robber : Datastruct_and_algo_excersizes.InputRobber
    {
        public InputRobbinBankState() : base("RobbinBank")
        {

        }

        public override bool EvaluateAgent(Robber agent, out State<Robber>
changeStateToo)
        {
            changeStateToo = null;
            if(agent.agentString.Equals("Do nothing"))
            {
                return false;
            }
            foreach(KeyValuePair<string, State<Robber>> state in this.exitStates)
            {
                if (state.Value._stateName.Equals(agent.agentString))
                {
                    changeStateToo = state.Value;
                    return true;
                }
            }
            Console.WriteLine("Error: input fell through, something went wrong in the
statemanager debug info:\n" +
                "{0} State, {1} agentString", this, agent.agentString);
            return false;
        }

        public override void OnEnterState(State<Robber> prevState)
        {
            Console.WriteLine("It's robbin time");
        }

        public override void OnExitState(State<Robber> nextState)
        {
            Console.WriteLine("I aint robbin no more");
        }

        public override void OnStayInState(Robber agent)
        {
            Console.WriteLine("I'm still robbin");
        }
```

```
    }

    class InputFleeingState<Robber> : State<Robber>, StateInterface<Robber>
        where Robber : Datastruct_and_algo_excersizes.InputRobber
    {
        public InputFleeingState() : base("Fleeing")
        {

        }

        public override bool EvaluateAgent(Robber agent, out State<Robber>
changeStateToo)
        {
            changeStateToo = null;
            if (agent.agentString.Equals("Do nothing"))
            {
                return false;
            }
            foreach (KeyValuePair<string, State<Robber>> state in this.exitStates)
            {
                if (state.Value._stateName.Equals(agent.agentString))
                {
                    changeStateToo = state.Value;
                    return true;
                }
            }
            Console.WriteLine("Error: input fell through, something went wrong in the
statemanager debug info:\n" +
                "{0} State, {1} agentString", this, agent.agentString);
            return false;
        }

        public override void OnEnterState(State<Robber> prevState)
        {
            Console.WriteLine("Fuck this, time to bolt");
        }

        public override void OnExitState(State<Robber> nextState)
        {
            Console.WriteLine("I aint runnin no more");
        }

        public override void OnStayInState(Robber agent)
        {
            Console.WriteLine("Gotta go fast");
        }
    }

    class InputHavingGoodTimeState<Robber> : State<Robber>, StateInterface<Robber>
        where Robber : Datastruct_and_algo_excersizes.InputRobber
    {
        public InputHavingGoodTimeState() : base("HavingGoodTime")
        {

        }

        public override bool EvaluateAgent(Robber agent, out State<Robber>
changeStateToo)
        {
            changeStateToo = null;
            if (agent.agentString.Equals("Do nothing"))
            {
```

```
                return false;
            }
            foreach (KeyValuePair<string, State<Robber>> state in this.exitStates)
            {
                if (state.Value._stateName.Equals(agent.agentString))
                {
                    changeStateToo = state.Value;
                    return true;
                }
            }
            Console.WriteLine("Error: input fell through, something went wrong in the
statemanager debug info:\n" +
                "{0} State, {1} agentString", this, agent.agentString);
            return false;
        }

        public override void OnEnterState(State<Robber> prevState)
        {
            Console.WriteLine("Time to have a good time");
        }

        public override void OnExitState(State<Robber> nextState)
        {
            Console.WriteLine("I aint havin a good time no more");
        }

        public override void OnStayInState(Robber agent)
        {
            Console.WriteLine("Keep the good times goin!");
        }
    }

    class InputLayingLowState<Robber> : State<Robber>, StateInterface<Robber>
        where Robber : Datastruct_and_algo_excersizes.InputRobber
    {
        public InputLayingLowState() : base("LayingLow")
        {

        }

        public override bool EvaluateAgent(Robber agent, out State<Robber>
changeStateToo)
        {
            changeStateToo = null;
            if (agent.agentString.Equals("Do nothing"))
            {
                return false;
            }
            foreach (KeyValuePair<string, State<Robber>> state in this.exitStates)
            {
                if (state.Value._stateName.Equals(agent.agentString))
                {
                    changeStateToo = state.Value;
                    return true;
                }
            }
            Console.WriteLine("Error: input fell through, something went wrong in the
statemanager, debug info:\n" +
                "{0} State, {1} agentString", this, agent.agentString);
            return false;
        }
```

```
        public override void OnEnterState(State<Robber> prevState)
        {
            Console.WriteLine("A shit, time to Duck down");
        }

        public override void OnExitState(State<Robber> nextState)
        {
            Console.WriteLine("I aint chillin no more");
        }

        public override void OnStayInState(Robber agent)
        {
            Console.WriteLine("Ima chill a bit longer");
        }
    }
}
```

This was then followed by creating the robber class(ofcourse it already exsisted but it wasn't fully implemented yet. Otherwise the where line would not work)

```
namespace Datastruct_and_algo_excersizes
{
    /*
     * Class to contain the robber's variables and any references he may need to other
objects
     * aswell as his respective stateMachine
     */
    class InputRobber
    {
        StateManager<InputRobber> myStateMachine;
        public string agentString;

        public InputRobber()
        {
            myStateMachine = new StateManager<InputRobber>(this);
            var robbingBankState = new InputRobbinBankState<InputRobber>();
            var fleeingState = new InputFleeingState<InputRobber>();
            var goodTimeState = new InputHavingGoodTimeState<InputRobber>();
            var layingLowState = new InputLayingLowState<InputRobber>();

            //connect the states with one another
            robbingBankState.AddExitState(goodTimeState);
            robbingBankState.AddExitState(fleeingState);
            layingLowState.AddExitState(robbingBankState);
            goodTimeState.AddExitState(fleeingState);
            goodTimeState.AddExitState(layingLowState);
            fleeingState.AddExitState(layingLowState);
            fleeingState.AddExitState(robbingBankState);

            //add my new state transitions
            layingLowState.AddExitState(goodTimeState);
            fleeingState.AddExitState(goodTimeState);

            //add states too the manager
            myStateMachine.AddState(fleeingState);
            myStateMachine.AddState(goodTimeState);
            myStateMachine.AddState(layingLowState);
            myStateMachine.AddState(robbingBankState, true);//this is our starting
state, so pass true for the optional paramater
            try
            {
                if (!myStateMachine.EnableStateMachine())
```

```
                {
                    Console.WriteLine("State machine failed to enalbe");
                }
            }
            catch (StateNotIncludedException e)
            {
                Console.WriteLine(e.Message + "\n" + e.StackTrace);
            }
        }

        public void StateInput()
        {
            string[] options = new
string[myStateMachine._currentState.exitStates.Count + 1];
            {
                int i = 0;
                foreach (KeyValuePair<string, State<InputRobber>> state in
myStateMachine._currentState.exitStates)
                {
                    options[i++] = state.Value._stateName;
                }
                options[i] = "Do nothing";
            }//we no longer need the i value so toss it.
            string input;
            var validInput = false;
            do
            {
                Console.WriteLine("Please tell the Robber what action to take. The
possible options are(No leading or trailing spaces, caps mattern \'|\' are
seperators):\n" +
                    "Current action is: " +
this.myStateMachine._currentState._stateName);
                foreach(string option in options)
                {
                    Console.Write(option + " | ");
                }
                Console.WriteLine();
                input = Console.ReadLine();
                if (options.Contains(input))
                {
                    validInput = true;
                }
                else
                {
                    Console.WriteLine("Boy, that input was wrong, get your act
toghther...");
                }
                Console.WriteLine();
            } while (!validInput);
            this.agentString = input;
            this.myStateMachine.ExecuteCurrentState();
        }
    }
}
```

Question 3

Add output to your program by showing a line of text that indicates the current state or action.

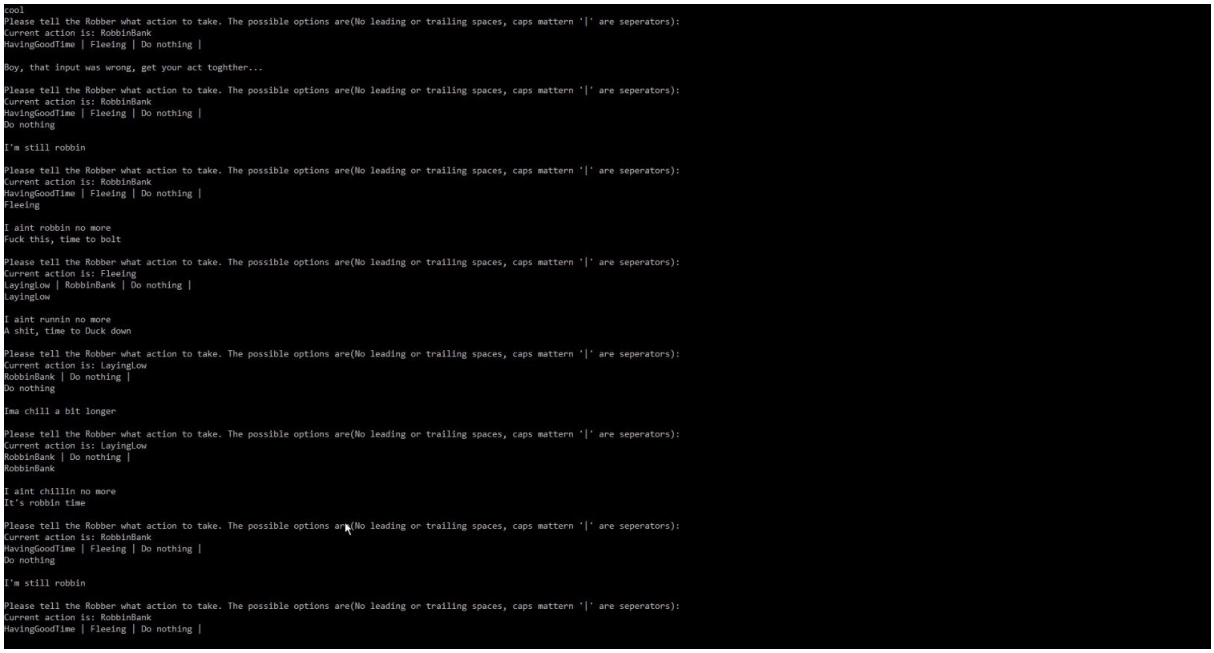
This was already implemented during Question 2. Find the output lines within the OnExitState OnEnterState and OnRemainInState methods.

Excersize3_Q3 file looks like this:

```
class Excersize3_Q3 : Excersize
    {
        public override int run()
        {
            //Create agent
            var robber = new InputRobber();

            for (int i = this._n; i > 0; i--) {//not a fan of forever so lets do this
N times
                //run the input robber's get input method.
                robber.StateInput();
                Console.WriteLine("You still have " + i + " turns left");
            }

            return base.run();
        }
    }
```

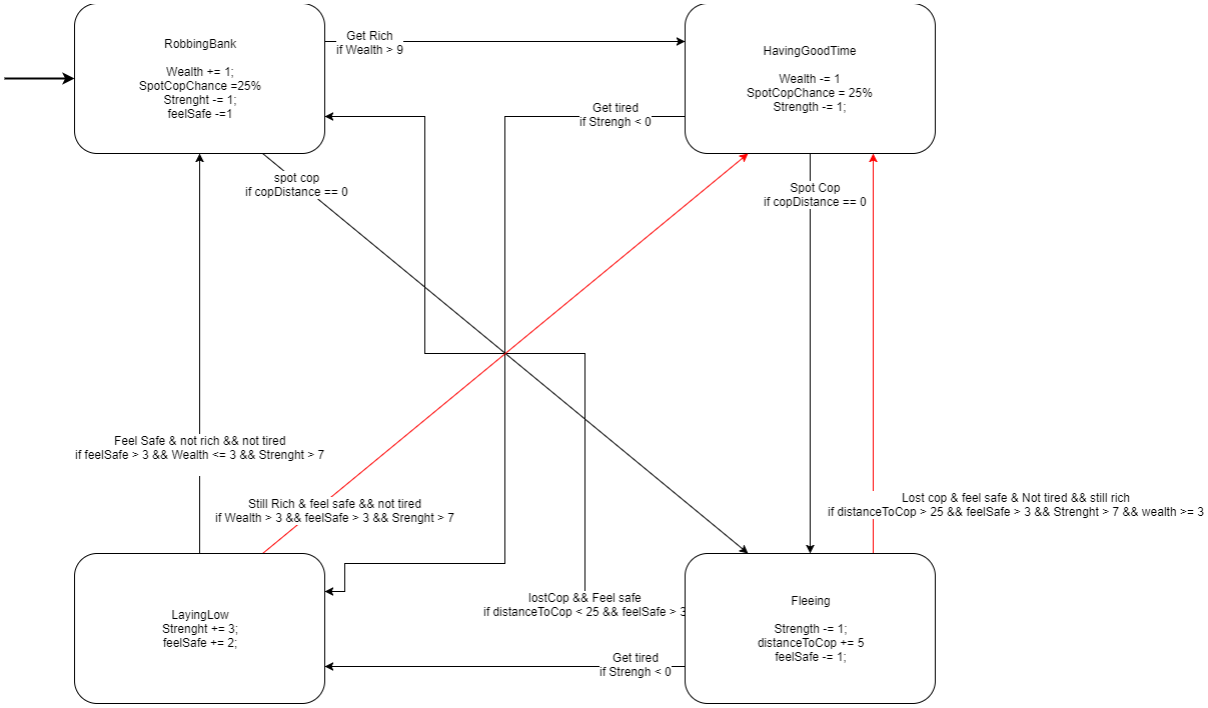
Note that I am still using the same program.cs file from exercise 1 and 2.

Example output:



Question 4

Add variables to your NPC for wealth, distanceToCop and strenght. Wealth increases while the NPC is robbing banks, but decreases while he is having a good time or fleeing. DistanceToCop can suddenly change to 0 during robbing a bank or having a good time, which causes the NPC to start fleeing. Strenght decreases during robbery, having a good time and fleeing, but increases during laying low. Continue this way and make sure that each state transition depends on one or more of these variables, and draw out your new diagram



Question 5

Implement the variables and conditions state transitions that you chose in your NPC. Add a loop to the program, such state it updates the variables and looks at the current state at each iteration; it then determines whether a state transition should occur. Also implement a one second pause at each iteration to your program. You can now remove the user input function, as the new states is set only by the variable's values which are updated at each iteration. Adjust your program so that this works and show the output of an example run.

The excersize3_Q5 file looks like this:

```
class Excersize3_Q5 : Excersize
    {
        public override int run()
        {
            //Create agent
            var robber = new AutomatedRobber();

            for (int i = this._n; i > 0; i--)
            {//not a fan of forever so lets do this N times
                //run the input robber's get input method.
                robber.EvaluateStateMachine();
                Console.WriteLine("You still have " + i + " turns left");
                System.Threading.Thread.Sleep(500);//sleep for half a second
            }

            return base.run();
        }
    }
```

I'm still somewhat limiting the amount of times the program executes the evaluateStateMachine method as I am not a fan of infinite loops.

I've put the thread to sleep for 500 ms as I don't like waiting.

The automated robber class and his respective states look like this:

```
namespace Datastruct_and_algo_excersizes
{
    class AutomatedRobber
    {
        StateManager<AutomatedRobber> myStateMachine;
        //create the agents variables.
        public float distanceToCop = 10, wealth = 2, strength = 5, feelSafe = 0;

        public AutomatedRobber()
        {
            myStateMachine = new StateManager<AutomatedRobber>(this);
            var robbingBankState = new AutomatedRobbinBankState<AutomatedRobber>();
            var fleeingState = new AutomatedFleeingState<AutomatedRobber>();
            var goodTimeState = new AutomatedHavingGoodTimeState<AutomatedRobber>();
            var layingLowState = new AutomatedLayingLowState<AutomatedRobber>();

            //connect the states with one another
            robbingBankState.AddExitState(goodTimeState);
            robbingBankState.AddExitState(fleeingState);
            layingLowState.AddExitState(robbingBankState);
            goodTimeState.AddExitState(fleeingState);
            goodTimeState.AddExitState(layingLowState);
            fleeingState.AddExitState(layingLowState);
            fleeingState.AddExitState(robbingBankState);

            //add my new state transitions
            layingLowState.AddExitState(goodTimeState);
            fleeingState.AddExitState(goodTimeState);

            //add states too the manager
            myStateMachine.AddState(fleeingState);
            myStateMachine.AddState(goodTimeState);
            myStateMachine.AddState(layingLowState);
            myStateMachine.AddState(robbingBankState, true);//this is our starting
state, so pass true for the optional paramater
            try
            {
                if (!myStateMachine.EnableStateMachine())
                {
                    Console.WriteLine("State machine failed to enalbe");
                }
            }
            catch (StateNotIncludedException e)
            {
                Console.WriteLine(e.Message + "\n" + e.StackTrace);
            }
        }

        public bool _lostCop
        {
            get { return this.distanceToCop > 25; }
        }

        public bool _feelSafe
        {
            get { return this.feelSafe > 3;  }
        }

        public bool _spotCop
        {
```

```
            get { return this.distanceToCop == 0; }
        }

        public bool _isRich
        {
            get { return this.wealth > 2; }
        }

        public bool _gotRich
        {
            get { return this.wealth > 9; }
        }

        public bool _notTired
        {
            get { return this.strength > 7; }
        }

        public bool _stillRich
        {
            get { return this.wealth >= 3; }
        }

        public bool _tired
        {
            get { return this.strength < 0; }
        }

        public void EvaluateStateMachine()
        {
            this.myStateMachine.ExecuteCurrentState();
        }


    }

    //Becuase I am lazy I am just going to toss these states here. Lazyness ftw

    class AutomatedRobbinBankState<Robber> : State<Robber>, StateInterface<Robber>
        where Robber : Datastruct_and_algo_excersizes.AutomatedRobber
    {
        public AutomatedRobbinBankState() : base("RobbinBank")
        {

        }

        public override bool EvaluateAgent(Robber agent, out State<Robber>
changeStateToo)
        {
            changeStateToo = null;
            if (agent._gotRich)
            {
                changeStateToo = this.exitStates["HavingGoodTime"];
                return true;
            }
            if (agent._spotCop)
            {
                changeStateToo = this.exitStates["Fleeing"];
                return true;
            }
            return false;
        }
```

```
        public override void OnEnterState(State<Robber> prevState)
        {
            Console.WriteLine("It's robbin time");
        }

        public override void OnExitState(State<Robber> nextState)
        {
            Console.WriteLine("I aint robbin no more");
        }

        public override void OnStayInState(Robber agent)
        {
            Console.WriteLine("I'm still robbin");
            agent.strength -= 1;
            agent.feelSafe -= 1;
            agent.wealth += 1;
        }
    }

    class AutomatedFleeingState<Robber> : State<Robber>, StateInterface<Robber>
        where Robber : Datastruct_and_algo_excersizes.AutomatedRobber
    {
        public AutomatedFleeingState() : base("Fleeing")
        {

        }

        public override bool EvaluateAgent(Robber agent, out State<Robber>
changeStateToo)
        {
            changeStateToo = null;

            if (agent._tired)
            {
                changeStateToo = this.exitStates["LayingLow"];
                return true;
            }
            if(agent._lostCop && agent._feelSafe
            && agent._notTired && agent._stillRich)
            {
                changeStateToo = this.exitStates["HavingGoodTime"];
                return true;
            }
            if(agent._lostCop && agent._feelSafe)
            {
                changeStateToo = this.exitStates["RobbinBank"];
                return true;
            }
            return false;
        }

        public override void OnEnterState(State<Robber> prevState)
        {
            Console.WriteLine("Fuck this, time to bolt");
        }

        public override void OnExitState(State<Robber> nextState)
        {
            Console.WriteLine("I aint runnin no more");
        }
```

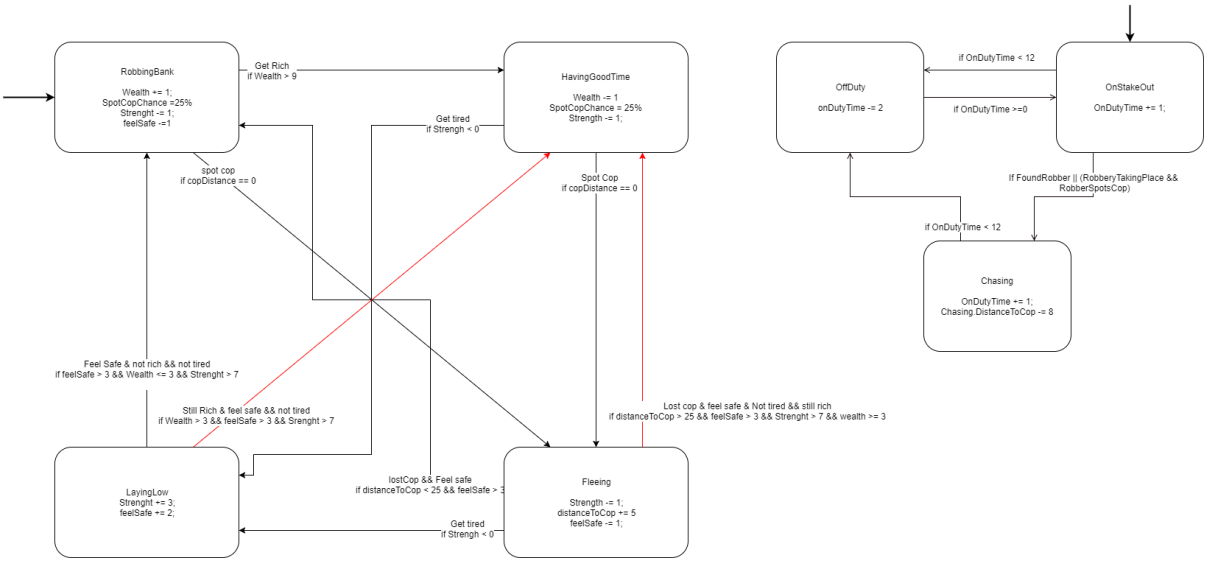
```
        public override void OnStayInState(Robber agent)
        {
            Console.WriteLine("Gotta go fast");
            agent.strength -= 1;
            agent.distanceToCop += 5;
            agent.feelSafe -= 1;
        }
    }

    class AutomatedHavingGoodTimeState<Robber> : State<Robber>, StateInterface<Robber>
        where Robber : Datastruct_and_algo_excersizes.AutomatedRobber
    {
        public AutomatedHavingGoodTimeState() : base("HavingGoodTime")
        {

        }

        public override bool EvaluateAgent(Robber agent, out State<Robber>
changeStateToo)
        {
            changeStateToo = null;
            if (agent._spotCop)
            {
                changeStateToo = this.exitStates["Fleeing"];
                return true;
            }
            if (agent._tired)
            {
                changeStateToo = this.exitStates["LayingLow"];
                return true;
            }
            return false;
        }

        public override void OnEnterState(State<Robber> prevState)
        {
            Console.WriteLine("Time to have a good time");
        }

        public override void OnExitState(State<Robber> nextState)
        {
            Console.WriteLine("I aint havin a good time no more");
        }

        public override void OnStayInState(Robber agent)
        {
            Console.WriteLine("Keep the good times goin!");
            agent.strength -= 1;
            agent.wealth -= 1;
            Random r = new Random(Guid.NewGuid().GetHashCode());
            if (r.Next(100) > 25)
            {
                agent.distanceToCop = 0;
            }
        }
    }

    class AutomatedLayingLowState<Robber> : State<Robber>, StateInterface<Robber>
        where Robber : Datastruct_and_algo_excersizes.AutomatedRobber
    {
        public AutomatedLayingLowState() : base("LayingLow")
        {
```

```
        }

        public override bool EvaluateAgent(Robber agent, out State<Robber>
changeStateToo)
        {
            changeStateToo = null;
            if(agent._feelSafe && !agent._isRich && agent._notTired)
            {
                changeStateToo = this.exitStates["RobbinBank"];
                return true;
            }
            if(agent._stillRich && agent._feelSafe && agent._notTired)
            {
                changeStateToo = this.exitStates["HavingGoodTime"];
                return true;
            }
            return false;
        }

        public override void OnEnterState(State<Robber> prevState)
        {
            Console.WriteLine("A shit, time to Duck down");
        }

        public override void OnExitState(State<Robber> nextState)
        {
            Console.WriteLine("I aint chillin no more");
        }

        public override void OnStayInState(Robber agent)
        {
            Console.WriteLine("Ima chill a bit longer");
            agent.strength += 3;
            agent.feelSafe += 2;
        }
    }
}
```

Question 6

Design a state diagram for a second NPC (Cop) with three possible states: OffDuty, OnStakeOut and Chasing. Design logical state transitions that depend on the variable 'dutyTime': the value of this variable increases during OnStakeOut and Chasing until it reaches a certain value, at which point the state transitions to OffDuty, when it starts decreasing again until 0. The Cop initially starts in the OnStakeOut state. When the former NPC starts robbing banks, the Cop will at some point start Chasing. The first NPC then starts fleeing, which changes the distanceToCop value. Draw out the full Cop state diagram and design logical transitions between states such that he is able to catch the first NPC and the program terminates.

To maintain a sense of logic I also added a “captured state”. I’ve also assumend that the robber is captured if the cop is right next to him, and I have limited the maximum distance between the robber and the cop to 50. So the entire new state diagram now looks like this:



Question 7

Implement the Cop NPC in your existing program loop. Make sure the Cop also outputs a line indicating its current state or actions.

This was rather simple at this point. His respective class and state's are as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Datastruct_and_algo_excersizes.StateMananger;

namespace Datastruct_and_algo_excersizes
{
    class AutomatedCop
    {
        public CapturableAutomatedRobber robber;
        public CapturableAutomatedRobber chasing;
        public float onDutyTime = 0;
        StateManager<AutomatedCop> myStateMachine;

        public bool _goOffDuty
        {
            get { return this.onDutyTime > 12; }
        }

        public bool _goOnnDuty
        {
            get { return this.onDutyTime <= 0; }
        }

        public bool _capturedTarget
        {
            get { return this.chasing._isCaptured; }
        }

        public AutomatedCop(CapturableAutomatedRobber robber)
        {
            this.robber = robber;
```

```
            myStateMachine = new StateManager<AutomatedCop>(this);

            //build states
            var stakeOutState = new AutomatedCopStakeOutState<AutomatedCop>();
            var offDutyState = new AutomatedCopOffDutyState<AutomatedCop>();
            var chasingState = new AutomatedCopChasingState<AutomatedCop>();

            //connect states
            stakeOutState.AddExitState(chasingState);
            stakeOutState.AddExitState(offDutyState);
            offDutyState.AddExitState(stakeOutState);
            chasingState.AddExitState(stakeOutState);
            chasingState.AddExitState(offDutyState);

            //add states too stateMachine
            myStateMachine.AddState(stakeOutState, true);
            myStateMachine.AddState(offDutyState);
            myStateMachine.AddState(chasingState);

            try
            {
                if (!myStateMachine.EnableStateMachine())
                {
                    Console.WriteLine("State machine failed to enalbe");
                }
            }
            catch (StateNotIncludedException e)
            {
                Console.WriteLine(e.Message + "\n" + e.StackTrace);
            }

        }

        public void EvaluateStateMachine()
        {
            this.myStateMachine.ExecuteCurrentState();
        }
    }

    class AutomatedCopStakeOutState<Cop> : State<Cop>, StateInterface<Cop>
        where Cop : Datastruct_and_algo_excersizes.AutomatedCop
    {
        public AutomatedCopStakeOutState() : base("StakeOut")
        {
        }

        public override bool EvaluateAgent(Cop agent, out State<Cop> changeStateToo)
        {
            changeStateToo = null;
            if (agent.chasing != null)
            {
                changeStateToo = this.exitStates["Chasing"];
                return true;
            }
            if (agent._goOffDuty)
            {
                changeStateToo = this.exitStates["OffDuty"];
                return true;
            }
            return false;
        }
```

```
        public override void OnEnterState(State<Cop> prevState)
        {
            Console.WriteLine("Time for active duty");
        }

        public override void OnExitState(State<Cop> nextState)
        {
            if(nextState._stateName.Equals("OffDuty"))
                Console.WriteLine("I've been copping too long");
            if (nextState._stateName.Equals("Chasing"))
                Console.WriteLine("Suspect spotted, engaging");
        }

        public override void OnStayInState(Cop agent)
        {
            Console.WriteLine("Stakin out places, trying to find trouble");
            agent.onDutyTime += 1;
            if (agent.robber._getCurrentState._stateName.Equals("RobbinBank") ||
agent.robber._getCurrentState._stateName.Equals("HavingGoodTime"))
            {
                Random r = new Random(Guid.NewGuid().GetHashCode());
                if (r.Next(100) > 25)
                {
                    agent.chasing = agent.robber;
                }
            }
        }
    }

    class AutomatedCopOffDutyState<Cop> : State<Cop>, StateInterface<Cop>
        where Cop : Datastruct_and_algo_excersizes.AutomatedCop
    {
        public AutomatedCopOffDutyState() : base("OffDuty")
        {
        }

        public override bool EvaluateAgent(Cop agent, out State<Cop> changeStateToo)
        {
            changeStateToo = null;
            if (agent._goOnnDuty)
            {
                changeStateToo = this.exitStates["StakeOut"];
                return true;
            }
            return false;
        }

        public override void OnEnterState(State<Cop> prevState)
        {
            Console.WriteLine("Active duty no longer, headed home");
        }

        public override void OnExitState(State<Cop> nextState)
        {
            Console.WriteLine("Back to Active duty...");
        }

        public override void OnStayInState(Cop agent)
        {
            Console.WriteLine("The cop drama shows on TV nowadays are so
unrealistic....");
```

```
                agent.onDutyTime -= 2;
                Random r = new Random(Guid.NewGuid().GetHashCode());
            }
        }

        class AutomatedCopChasingState<Cop> : State<Cop>, StateInterface<Cop>
            where Cop : Datastruct_and_algo_excersizes.AutomatedCop
        {
            public AutomatedCopChasingState() : base("Chasing")
            {
            }

            public override bool EvaluateAgent(Cop agent, out State<Cop> changeStateToo)
            {
                changeStateToo = null;
                if (agent._goOffDuty)
                {
                    changeStateToo = this.exitStates["OffDuty"];
                    agent.chasing = null;
                    return true;
                }
                if (agent.chasing._isCaptured)
                {
                    changeStateToo = this.exitStates["OffDuty"];
                    agent.chasing = null;
                    return true;
                }
                return false;
            }

            public override void OnEnterState(State<Cop> prevState)
            {
                Console.WriteLine("Time for active duty");
            }

            public override void OnExitState(State<Cop> nextState)
            {
                if (nextState._stateName.Equals("OffDuty"))
                    Console.WriteLine("I've been copping too long");
                if (nextState._stateName.Equals("Chasing"))
                    Console.WriteLine("Suspect spotted, engaging");
            }

            public override void OnStayInState(Cop agent)
            {
                Console.WriteLine("I'm getting closer too the purp " +
agent.chasing.distanceToCop);
                if (agent.chasing._getCurrentState._stateName.Equals("Fleeing"))
                {
                    Console.WriteLine("I am in HOT pursuit, I do not need backup as I am a
badass!");
                }
                else
                {
                    Console.WriteLine("The dummy doesn't even know I am on his tail");
                }
                agent.onDutyTime += 1;
                agent.chasing.distanceToCop -= 8;
            }
        }
    }
```

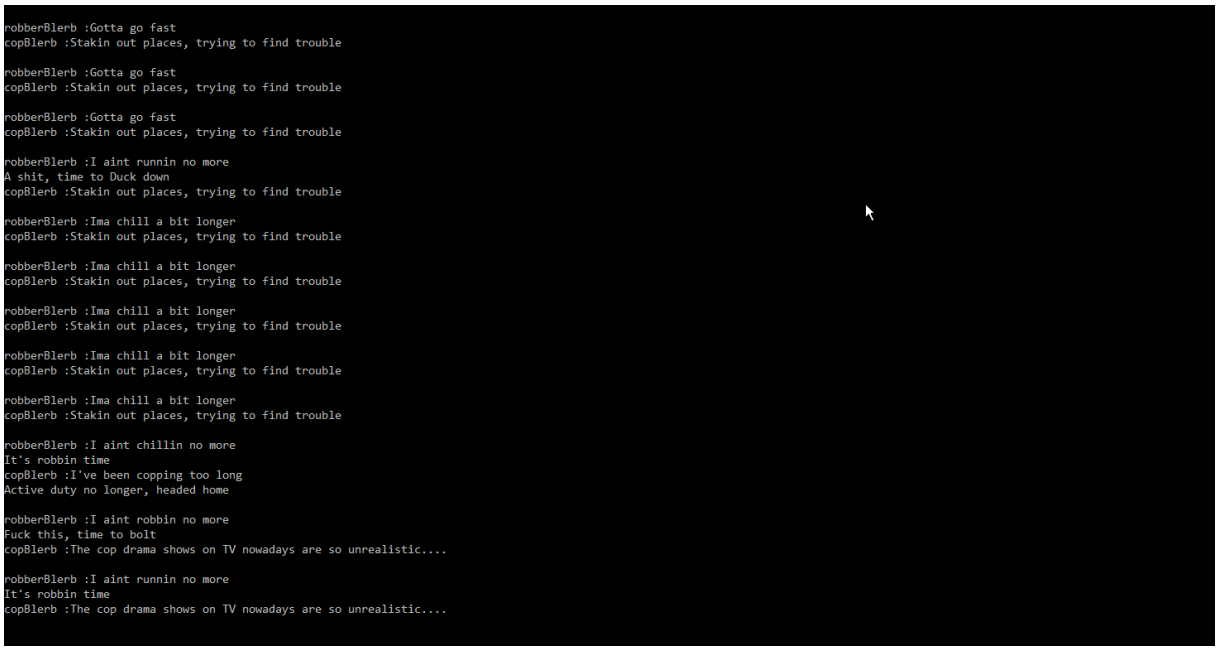
Question 8

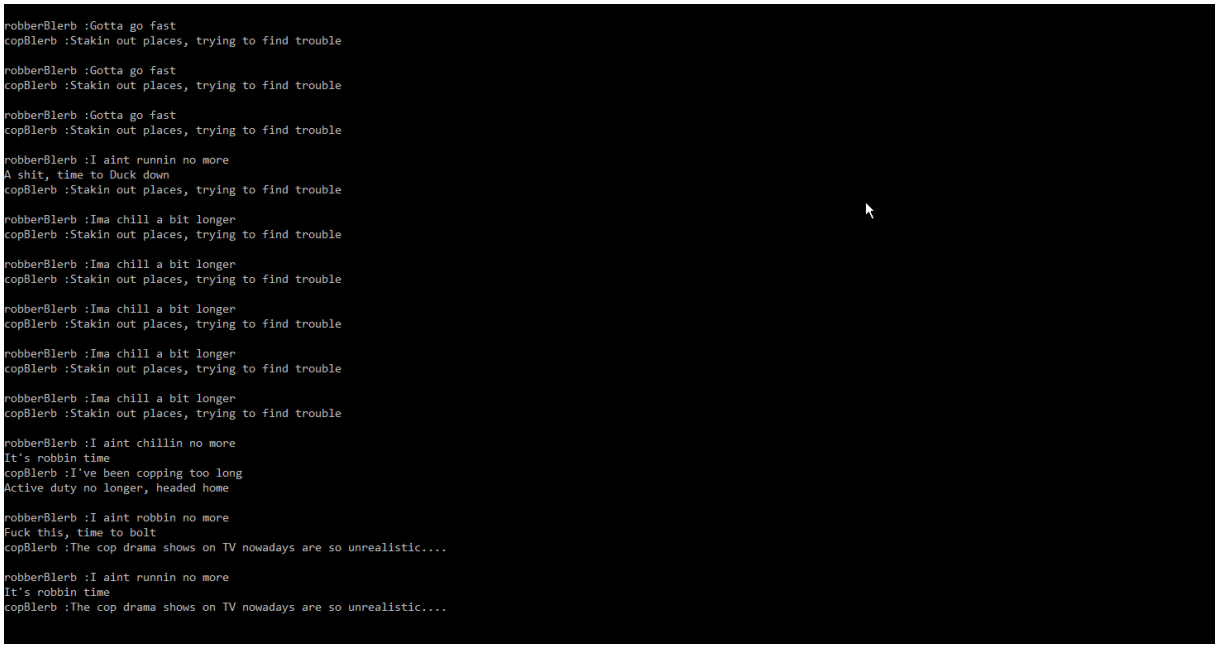
Show the output of an example run that terminates by itself, and showing alternating lines said by both NPC's.

For outputting and testing I added another excersize file, notable excersize3_Q7 which looks as follows:

```
namespace Datastruct_and_algo_excersizes
{
    class Excersize3_Q7 : Excersize
    {
        public override int run()
        {
            var robber = new CapturableAutomatedRobber();
            var cop = new AutomatedCop(robber);
            while (!robber._isCaptured)
            {
                Console.Write("robberBlerb :");
                robber.EvaluateStateMachine();
                Console.Write("copBlerb :");
                cop.EvaluateStateMachine();
                Console.WriteLine();
                System.Threading.Thread.Sleep(500);
            }
            return base.run();
        }
    }
}
```

And here are some example outputs of this program.




robberBlerb :Gotta go fast
copBlerb :Stakin out places, trying to find trouble
robberBlerb :Gotta go fast
copBlerb :Stakin out places, trying to find trouble
robberBlerb :Gotta go fast
copBlerb :Stakin out places, trying to find trouble
robberBlerb :I aint runnin no more
A shit, time to Duck down
copBlerb :Stakin out places, trying to find trouble
robberBlerb :Ima chill a bit longer
copBlerb :Stakin out places, trying to find trouble
robberBlerb :Ima chill a bit longer
copBlerb :Stakin out places, trying to find trouble
robberBlerb :Ima chill a bit longer
copBlerb :Stakin out places, trying to find trouble
robberBlerb :Ima chill a bit longer
copBlerb :Stakin out places, trying to find trouble
robberBlerb :Ima chill a bit longer
copBlerb :Stakin out places, trying to find trouble
robberBlerb :I aint chillin no more
It's robbin time
copBlerb :I've been copping too long
Active duty no longer, headed home
robberBlerb :I aint robbin no more
Fuck this, time to bolt
copBlerb :The cop drama shows on TV nowadays are so unrealistic....
robberBlerb :I aint runnin no more
It's robbin time
copBlerb :The cop drama shows on TV nowadays are so unrealistic....