# Finite state machine manual



## Package Overview

This package contains the following scripts:

IFiniteStateMachine.cs

FiniteStateMachine.cs

StateMachineState.cs

StateMachineTransition.cs

### Introduction

The state pattern is on of the fundamental design patterns any aspiring developer should know and is widely used in software programming. It is part of the 23 original software patterns described by the Gang of Four in their book "*Design Patterns: Elements of Reusable Object-Oriented Software*". The state pattern allows us to separate large codebases into states rather than contain them in one large script. This can

greatly simplify code design and debugging. A program continues to stay in a state until a condition tells it to transition to another state. A state machine consists of 5 fundamental elements. This implementation focuses on a table driven approach for state transitions.

### Owner

The owner is the object owning a state machine. The owner is also often called the context. The owner contains the information needed for the state machine to run. The owner contains the state machine, the states and the conditions for the transitions.

### State machine

The state machine is the brain of the state pattern. It is responsible for holding the transitions, transitioning between states and updating them.

### States

A state is where the custom logic is implemented in the program.

### Transitions

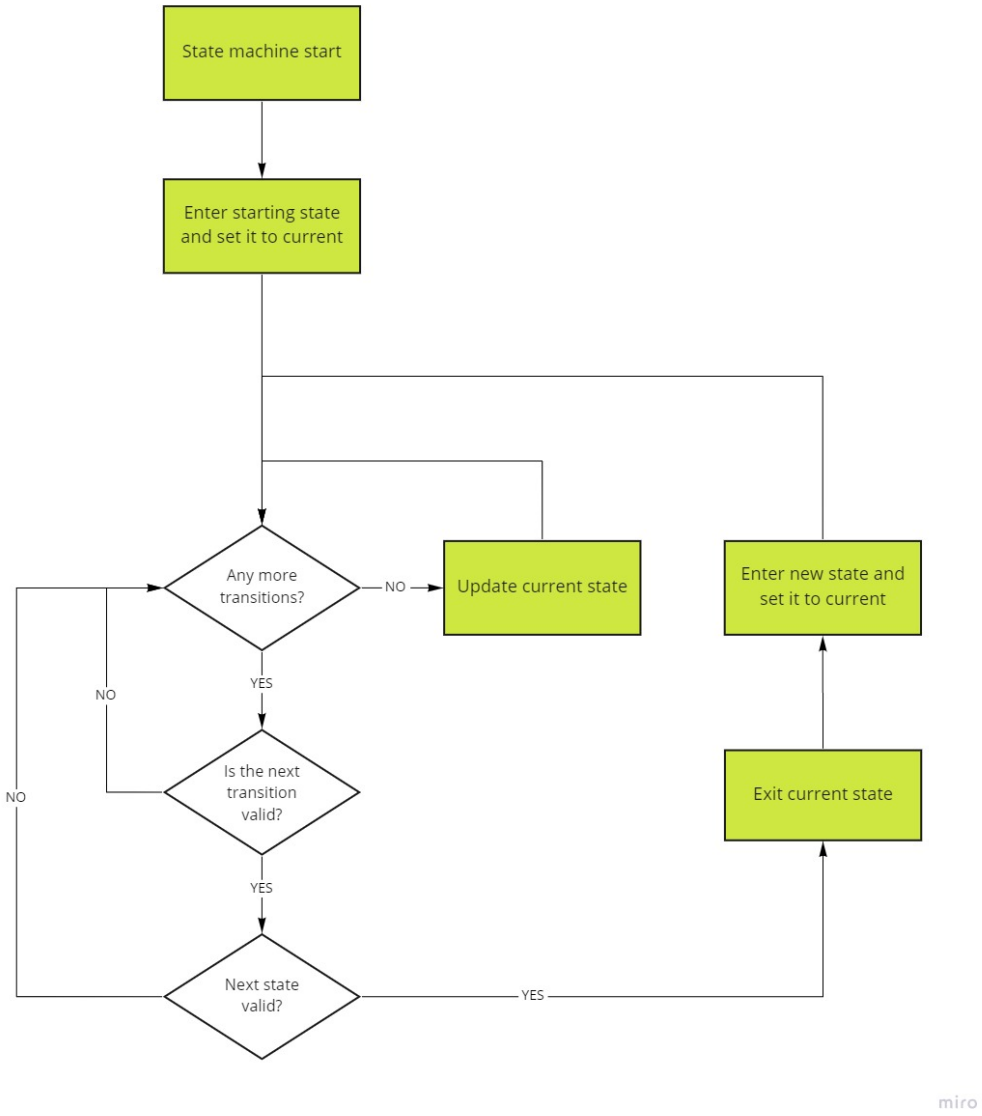
A transition allows one state to transition to another. It contains information about the origin state, the target state and the condition required to allow the move between them.

### Conditions

A condition is added to a transition, and decides if the transition is valid or not.

## State Machine Flow

Following is the flow of this finite state machine logic. A state machine is initialized and moves into a starting state, which becomes the current state. The state machine will stay in the current state until a transition becomes active, in which case it will exit the current state and move to the new one.




A state machine flow diagram.

# IFiniteStateMachine

This is the main interface class for the state machine. It defines the contract that all state machine implementations use.

## Public methods

### Start

```
public void Start(StateMachineState<T> state);
```

| | |
|---|---|
| state | The starting state of the machine |

Initializes a state machine with the given starting state

### Update

```
public void Update();
```

Updates the finite state machine

# FiniteStateMachine

The finite state machine class is the brain of the state machine. It is a container for transitions and is responsible for updating states and transitioning between them.

## Constructors

```
public FiniteStateMachine(T newOwner)
```

| | |
|---|---|
| newOwner | The owner of the Finite state machine |

Initializes a new finite state machine with it's owner.

## Public methods

### AddTransition

```
public void AddTransition(StateMachineState<T> originState, StateMachineState<T> targetState, (Func<bool> condition, bool conditionMod
```

| | |
|---|---|
| originState | The State this transition originates from |
| targetState | The State this transition is targeted towards |
| conditions | Array of condition and condition mode pairs. Each pair is put into a tuple with the syntax (Func<bool> condition, bool conditionMode). |

This is a concrete implementation of a finite state machine implementing the IFiniteStateMachine interface. A Finite state machine contains the following methods:

**IMPORTANT: Transitions must be added to the finite state machine before calling Start().**

### AddGlobalTransition

```
public void AddGlobalTransition(StateMachineState<T> targetState, (Func<bool> condition, bool conditionMode)[] conditions)
```

| | |
|---|---|
| targetState | The State this transition is targeted towards |
| conditions | Array of condition and condition mode pairs. Each pair is put into a tuple with the syntax (Func<bool> condition, bool conditionMode). |

Adds a global transition to the state machine. A global transition is evaluated each fram no matter what, and is evaluated before other transitions.

### Start

```
public void Start(StateMachineState<T> startingState)
```

| | |
|---|---|
| startingState | The starting state of the machine |

Starts the machine by transitioning to the starting state. In a monobehaviour, This method should be called in either Awake, OnEnable or Start. This method is required to get the state machine running. Must be called before update.

**IMPORTANT: Start() must be called before the state machine can update.**

### Update

```
public void Update()
```

Ticks the machine. In a monobehaviour, this method is usually called in Update, FixedUpdate or LateUpdate. Each update will first check all the transitions for the current state and make the first valid transition. If no transitions are valid, it will update the machines current state.

# StateMachineState

This is the base class for all StateMachineStates. A state is the place where the custom logic can be run by the state machine. A new state can be defined by inheriting from this base class. It is not possible to use this base class directly. It must be inherited by other classes.

## Public methods

### OnStateEnter

```
public abstract void OnStateEnter(T owner);
```

| | |
|---|---|
| owner | The owner of the state |

Called on the first frame that the state becomes active, much like Awake, OnEnable and Start in Unity. This method is typically used for initialization code like GetComponent or setting/resetting values.

### OnStateUpdate

```
public abstract void OnStateUpdate(T owner);
```

| | |
|---|---|
| owner | The owner of the state |

Updates the machine. This method is called if no transitions are valid. In Unity, this method is typically run in Update/FixedUpdate/LateUpdate, but can work for other implementations as well.

### OnStateExit

```
public abstract void OnStateExit(T owner);
```

| | |
|---|---|
| owner | The owner of the state |

Called when a transition becomes valid and this state exits. This method is called in the same frame and before the next state's OnStateEnter method. This method is typically used for cleanup, much like OnEnable and OnDestroy in Unity.

# StateMachineTransition

This is a class containing the logic for making a transition to a new state.

## Constructor

```
public StateMachineTransition(StateMachineState<T> originState, StateMachineState<T> targetState, (Func<bool> condition, bool conditio
```

| | |
|---|---|
| originState | The State this transition originates from |
| targetState | The State this transition is targeted towards |
| conditions | Array of condition and condition mode pairs. Each pair is put into a tuple with the syntax (Func<bool> condition, bool conditionMode). |

Constructs a new StateMachineTransition

## Public methods

### Evaluate

```
public bool Evaluate()
```

Evaluates the transition by comparing the supplied condition method to the conditionMode. If they are equal, the method will return true. If not, it will return false.

## Properties

### TargetState

```
public StateMachineState<T> TargetState;
```

Returns the target state of this transition

### TargetState

```
public StateMachineState<T> OriginState;
```

Returns the origin state of this transition

# Example Usage

Below is a use case example with code to highlight how to use the state machine classes.

## Choosing and implementing a context

A context is the class owning the machine. Typically, it would be a gameObject, player controller, game manager or any other class that we would like to turn into a state machine. The important thing about the context is that it contains the state specific information (Ground checks, Movement speed, ammo, etc) or the means of getting these.

Create a class “StateMachineOwner” with the following code. It has two state specific variables, “isAngry” and “isHappy” that it will use to change it’s state. These variables will be changed by the players key presses in the update loop.

```
public class StateMachineOwner : MonoBehaviour
{
    [SerializeField] private KeyCode toggleHappy;
    [SerializeField] private KeyCode toggleAngry;
    [SerializeField] private KeyCode toggleDead;

    // State parameters
    private bool isAngry;
    private bool isHappy;
    private bool isDead;

    private void Update
    {
        if(Input.GetKeyDown(toggleAngry))
        {
            isAngry = !isAngry;
        }

        if(Input.GetKeyDown(toggleHappy))
        {
            isHappy = !isHappy;
        }

        if(Input.GetKeyDown(toggledead))
        {
            isDead = !isDead;
        }
    }
}
```

## Implementing the FiniteStateMachine

The finite state machine is the class controlling the flow of the machine. It Updates the states, makes the transitions and does the general bookkeeping for us. In the StateMachineOwner class we add a new variable for the machine and initialize it in the

Awake.

```
private FiniteStateMachine<StateMachineOwner> machine;

private void Awake()
{
    machine = new FiniteStateMachine<StateMachineOwner>();
}
```

## Creating States

A state is a container for all the logic that needs to be run while in that state, like moving, particle effects, animations, sounds, etc. A new state is created by inheriting from the StateMachineState base class. Create 3 classes named NormalState.cs, HappyState.cs and AngryState.cs. Let them all inherit from StateMachineState. The base class will enforce the use of the 3 methods, OnStateEnter, OnStateUpdate and OnStateExit so that we don’t forget them.

```
public class NormalState : StateMachineState<StateMachineOwner>
{
    public override void OnStateEnter(StateMachineOwner owner)
    {
        Debug.Log("I am Normal");
    }

    public override void OnStateUpdate(StateMachineOwner owner);
    {
    }

    public override void OnStateExit(StateMachineOwner owner)
    {
        Debug.Log("I am no longer Normal");
    }
```

```
public class HappyState : StateMachineState<StateMachineOwner>
{
    public override void OnStateEnter(StateMachineOwner owner)
    {
        Debug.Log("I am Happy");
    }

    public override void OnStateUpdate(StateMachineOwner owner);
    {
    }

    public override void OnStateExit(StateMachineOwner owner)
    {
        Debug.Log("I am no longer Happy");
    }
```

```
public class AngryState : StateMachineState<StateMachineOwner>
{
    public override void OnStateEnter(StateMachineOwner owner)
    {
        Debug.Log("I am Angry, Grrr");
    }

    public override void OnStateUpdate(StateMachineOwner owner);
    {
    }

    public override void OnStateExit(StateMachineOwner owner)
    {
        Debug.Log("I am no longer Angry");
    }
```

```
public class DeadState : StateMachineState<StateMachineOwner>
{
    public override void OnStateEnter(StateMachineOwner owner)
    {
        Debug.Log("I am dead, shit");
    }

    public override void OnStateUpdate(StateMachineOwner owner);
    {
```

```
    }

    public override void OnStateExit(StateMachineOwner owner)
    {
        Debug.Log("There is no coming back from death");
    }
```

We add references to these states directly in our StateMachineOwner class

```
protected FiniteStateMachine<StateMachineOwner> machine;
protected normalState = new NormalState();
protected happyState = new HappyState();
protected angryState = new AngryState();
protected deadState = new DeadState();
```

## Adding transitions

A transition contains the logic for moving from one state to the next. For the transition to occur, a series of conditions need to be met. In this system, conditions must be methods with a boolean return value and no parameters. The state machine uses C# delegates to store the methods as pointers in the transition class.

Create three methods in the StateMachineOwner class called “IsHappy”, “IsAngry” and “IsDead”. As stated above, both methods must have no parameters and a boolean return value. The state machine will call these methods internaly.

```
private bool IsHappy()
{
    return isHappy;
}

private bool IsAngry()
{
    return isAngry;
}

private bool IsDead()
{
    return isDead;
}
```

A state machine transition can be added with the following code:

```
machine.AddTransition(normalState, happyState, (IsHappy, true));
```

This consitutes a transition from normalState to happyState using the IsHappy function. Notice that there are no brackets behind IsHappy! The last boolean parameter implies that we want IsHappy to be true for the transition to be valid. In the code below, more transitions have been added in the Awake method.

```
private void Awake()
{
    machine = new FiniteStateMachine<StateMachineOwner>(this);
    machine.AddTransition(normalState, happyState, (IsHappy, true));
    machine.AddTransition(happyState, normalState, (IsHappy, false));
    machine.AddTransition(normalState, angryState, (IsAngry, true));
    machine.AddTransition(angryState, normalState, (IsAngry, false));
}
```

We can also add any number of transition conditions using the syntax

```
machine.AddTransition(normalState, happyState, (IsHappy, true), (IsAngry, false))
```

## Adding Global transitions

A global transition is a special kind of transition that can be called any time. (Like the animator AnyState node). Global transitions are checked every frame no matter what, and they are evaluated before regular transitions.

```
machine.AddGlobalTransition(deadState, (IsDead, true));
```

Add it together with the other transitions.

```
private void Awake()
{
    machine = new FiniteStateMachine<StateMachineOwner>(this);
    machine.AddTransition(normalState, happyState, (IsHappy, true));
    machine.AddTransition(happyState, normalState, (IsHappy, false));
    machine.AddTransition(normalState, angryState, (IsAngry, true));
    machine.AddTransition(angryState, normalState, (IsAngry, false));
    machine.AddGlobalTransition(deadState, (IsDead, true));
}
```

## Initializing and running the machine

The machine is initialized by calling it's Start method. We will call this method in the Start() method of the StateMachineOwner class, but it can also be called in Awake(After adding the transitions!) or OnEnable.

```
private void Start()
{
    machine.Start(normalState);
}
```

This will initialize the machine to start in the normalstate. After initializing the machine, we must update it each frame. We will put the code in the StateMachineOwner's Update method after the input checks.

```
private void Update()
{
    if(Input.GetKeyDown(toggleAngry))
    {
        isAngry = !isAngry;
    }

    if(Input.GetKeyDown(toggleHappy))
    {
        isHappy = !isHappy;
    }

    if(Input.GetKeyDown(toggleDead))
    {
        isDead = !isDead;
    }

    machine.Update();
}
```

## Final Code

The final code for the component is given below. putting it on a gameObject will allow the gamObject to change states based on player inputs.

```
using UnityEngine;

public class StateMachineOwner : MonoBehaviour
{
    [SerializeField] private KeyCode toggleHappy;
    [SerializeField] private KeyCode toggleAngry;
    [SerializeField] private KeyCode toggleDead;

    // State parameters
    private bool isAngry;
    private bool isHappy;
    private bool isDead;

    private FiniteStateMachine<StateMachineOwner> machine;
    protected NormalState normalState = new NormalState();
    protected HappyState happyState = new HappyState();
    protected AngryState angryState = new AngryState();
    protected DeadState deadState = new DeadState();
```

```
    private void Awake()
    {
        machine = new FiniteStateMachine<StateMachineOwner>(this);
        machine.AddTransition(normalState, happyState, (IsHappy, true));
        machine.AddTransition(happyState, normalState, (IsHappy, false));
        machine.AddTransition(normalState, angryState, (IsAngry, true));
        machine.AddTransition(angryState, normalState, (IsAngry, false));
        machine.AddGlobalTransition(deadState, (IsDead, true));
    }

    private void Start()
    {
        machine.Start(normalState);
    }

    private void Update()
    {
        if(Input.GetKeyDown(toggleAngry))
        {
            isAngry = !isAngry;
        }

        if (Input.GetKeyDown(toggleHappy))
        {
            isHappy = !isHappy;
        }

        if (Input.GetKeyDown(toggleDead))
        {
            isDead = !isDead;
        }

        machine.Update();
    }

    private bool IsHappy()
    {
        return isHappy;
    }

    private bool IsAngry()
    {
        return isAngry;
    }

    private bool IsDead()
    {
        return isDead;
    }
}
```