Actionscript 3.0 Basics

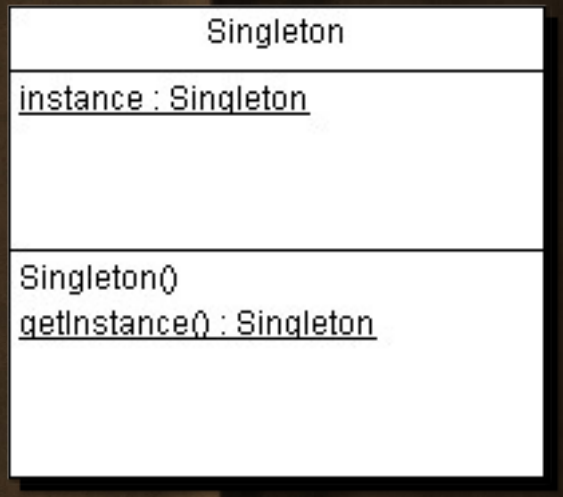
8 – Patterns

Patterns

In software engineering, a pattern is a **general reusable solution** to a **commonly occurring problem** that can be transformed directly into code.

It is a description or template for how to solve a problem that can be used in many different situations.

Singleton



The singleton pattern is used to implement the mathematical concept of a singleton, by **restricting the instantiation** of a class to **one object**.

Advantages:
- Single point of control
- Global Access

Singleton: Managers

Example: Sound Manager

```
1  package sound.manager {
2
3      public class SoundManager extends  EventDispatcher {
4
5          // SINGLETON
6          private static var instance:SoundManager
7
8          public static function get manager():SoundManager
9          {
10             if(!instance){
11                 instance = new SoundManager()
12             }
13             return instance
14         }
15
16         public function SoundManager():void {
17
18         }
19
20         public function addSound(key:String, sound:Sound):Sound {
21
22         }
23
24         public function playSound(key:String, startTime:Number = 0, loops:int = 0, sndTransform:SoundTransform = null):SSound {
25
26         }
27
28         public function stopSound(key:String):void {
29
30         }
31
32         public function unmute():void {
33
34         }
35
36         public function mute():void {
37
38         }
```

Singleton: Managers

Example: Sound Manager Usage

```
1  import sound.manager.SoundManager;
2  [Embed(source="shoot.mp3")]
3  private static const SOUND_SHOOT:Class;
4  [Embed(source="run.mp3")]
5  private static const SOUND_RUN:Class;
6  [Embed(source="jump.mp3")]
7  private static const SOUND_JUMP:Class;
8
9  SoundManager.manager.addSound("shoot",SOUND_SHOOT);
10 SoundManager.manager.addSound("run",SOUND_RUN);
11 SoundManager.manager.addSound("jump",SOUND_JUMP);
12
13 SoundManager.manager.playSound("jump");
14
15 //Still can do this:
16 var anotherManager:SoundManager = new SoundManager();
```

Singleton: Managers

Strict Singleton:

```
1  package sound.manager {
2
3      public class SoundManager extends  EventDispatcher {
4
5          // SINGLETON
6          private static var instance:SoundManager
7          private static var singleton_lock:Boolean = true;
8
9          public static function get manager():SoundManager
10         {
11             if(!instance){
12                 singleton_lock = false;
13                 instance = new SoundManager()
14                 singleton_lock = true;
15             }
16             return instance
17         }
18
19         public function SoundManager():void {
20             if(singleton_lock){
21                 throw new Error("Cannot instantiate directly, use singleton SoundManager.manager")
22             }
23             else{
24                 //construct...
25             }
26         }
27
28         public function addSound(key:String, sound:Sound):Sound {
29
30         }
31
32         public function playSound(key:String, startTime:Number = 0, loops:int = 0, sndTransform:Sou
```

Singleton

General code:

```
1  public class Singleton {
2      private static const instance:Singleton
3      private static var singleton_lock:Boolean = true
4
5      public static get singleton():Singleton
6      {
7        if(!instance)
8        {
9          singleton_lock = false;
10         instance = new Singleton()
11         singleton_lock = true;
12       }
13
14       return instance
15     }
16
17     public Singleton()
18     {
19       if(singleton_lock)
20       {
21         throw new Error("Cannot instantiate directly, use singleton");
22       }
23
24       //construct ...
25     }
26 }
```

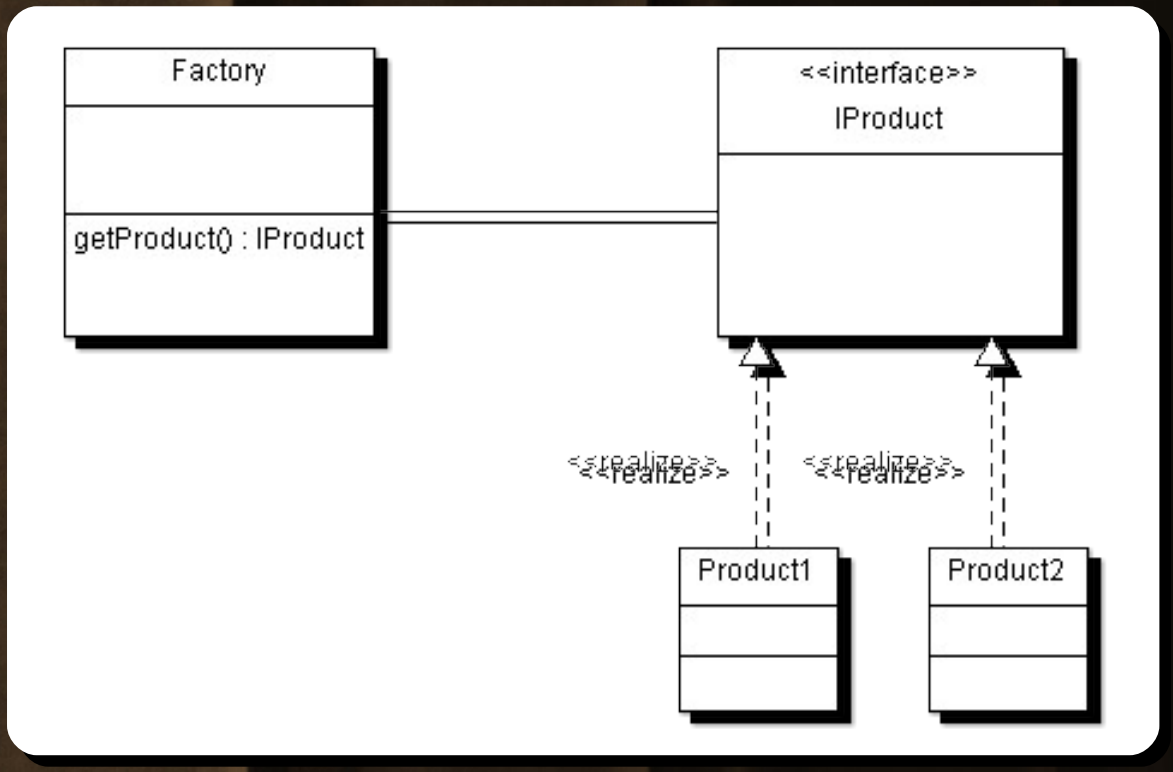
Singleton

Some people consider Singleton as an **anti-pattern**.

Reasons:
- Introduces Global Environments
- Promotes bad design practices when abused.
- Most of the time “singletonised” objects are not actually required to be unique.

Example:
Why is SoundManager a singleton?
Can we have a SoundManager per layer?
Can we have a SoundManager per level?

Factory



A factory is an **object for creating other objects**. It is an **abstraction of a constructor**, and can be used to implement various allocation schemes.

Factory: Asset Factory

Simple Asset Factory

```
1  package {
2      public class Actor extends Sprite
3      {
4          public function startAnimation():void {...}
5      }
6
7      public class Mario extends Actor
8      {
9          override public function startAnimation():void {...}
10     }
11
12     public class Koopa extends Actor
13     {
14         override public function startAnimation():void {...}
15     }
16
17     public class Goomba extends Actor
18     {
19         override public function startAnimation():void {...}
20     }
21
22     public class ActorFactory
23     {
24         public static function getActor(type:String):Actor
25         {
26             switch(type)
27             {
28                 case "mario":
29                 case "character": return new Mario();
30                     break;
31                 case "koopa": return new Koopa();
32                     break;
33                 case "goomba": return new Goomba();
34                     break;
35                 case "enemy":
36                     return new ([Koopa,Goomba][Math.round(Math.random())]) as Actor;
37                     break;
38             }
```

Factory: Asset Factory

Simple Asset Factory Usage

```
1  //create mario
2  var mario:Actor = ActorFactory.getActor("mario");
3  addChild(mario);
4  mario.startAnimation();
5
6  // add random enemies
7  var enemy:Actor;
8  for (var i:int = 0; i<5;i++ )
9  {
10     enemy = ActorFactory.getActor("enemy");
11     addChild(enemy);
12     enemy.startAnimation();
13 }
```

Pool

A pool is a **use case of the Factory Pattern**.
It is used mainly for object **lazy instantiation** and **re-utilization**.

Pool: Asset Pool

Simple asset pool:

```
1  package
2  {
3      public class Actor
4      {...}
5      public class Enemy extends Actor
6      {...}
7      public class Koopa extends Enemy
8      {...}
9      public class Goomba extends Enemy
10     {...}
11
12     public class EnemyPool
13     {
14         private static const POOL_SIZE:int = 10;
15         private static var pool:Vector.<Enemy> = new <Enemy>[];
16
17         public static function getEnemy():Enemy
18         {
19             var enemy:Enemy;
20             if(pool.length < POOL_SIZE)
21             {
22                 enemy = new ([Koopa,Goomba][Math.round(Math.random())]) as Enemy;
23             }
24             else
25             {
26                 enemy = pool.shift();
27             }
28
29             pool.push(enemy);
30             return enemy
31         }
32     }
33 }
```

Pool: Asset Pool

Resetable Assets:

```
package
{
    public interface IResetable
    {
        function reset():void;
    }

    public class Actor implements IResetable
    {
        public function reset():void
        {
            x = y = z = rotation = 0;
            scaleX = scaleY = scaleZ = 1;
            //...
        }
    }
    public class Enemy extends Actor
    {...}
    public class Koopa extends Enemy
    {...}
    public class Goomba extends Enemy
    {...}

    public class EnemyPool
    {
        private static const POOL_SIZE:int = 10;
        private static var pool:Vector.<Enemy> = new <Enemy>[];

        public static function getEnemy():Enemy
        {
            var enemy:Enemy;
            if(pool.length < POOL_SIZE)
            {
                enemy = new ([Koopa,Goomba][Math.round(Math.random())]) as Enemy;
            }
            else
            {
                enemy = pool.shift();
            }

            pool.push(enemy);
            enemy.reset();
            return enemy
        }
    }
}
```

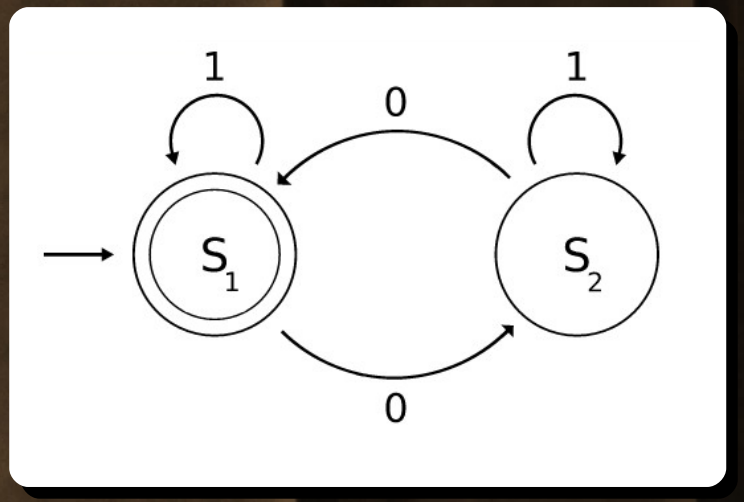
Pool: General Purpose Pool

```
1  package
2  {
3      public interface IPoolable
4      {
5          function init(...args):void;
6          function reset(...args):void;
7      }
8
9      public class Pool
10     {
11         public var size:int = 10;
12
13         private var type:Class;
14         private var pool:Vector.<IPoolable>;
15
16         public function Pool(type:Class)
17         {
18             this.type = type;
19             this.pool = new <IPoolable>[];
20         }
21
22         public function getInstance(...args):IPoolable
23         {
24             var item:IPoolable;
25             if(pool.length < size)
26             {
27                 item = new (type) as IPoolable;
28                 if(!item) throw new Error("Class provided is not IPoolable!")
29
30                 item.init.apply(item,args);
31             }
32             else
33             {
34                 enemy = pool.shift();
35                 item.reset.apply(item,args);
36             }
37
38             pool.push(item);
39             return item
40         }
41     }
42 }
```

Pool: General Purpose Pool

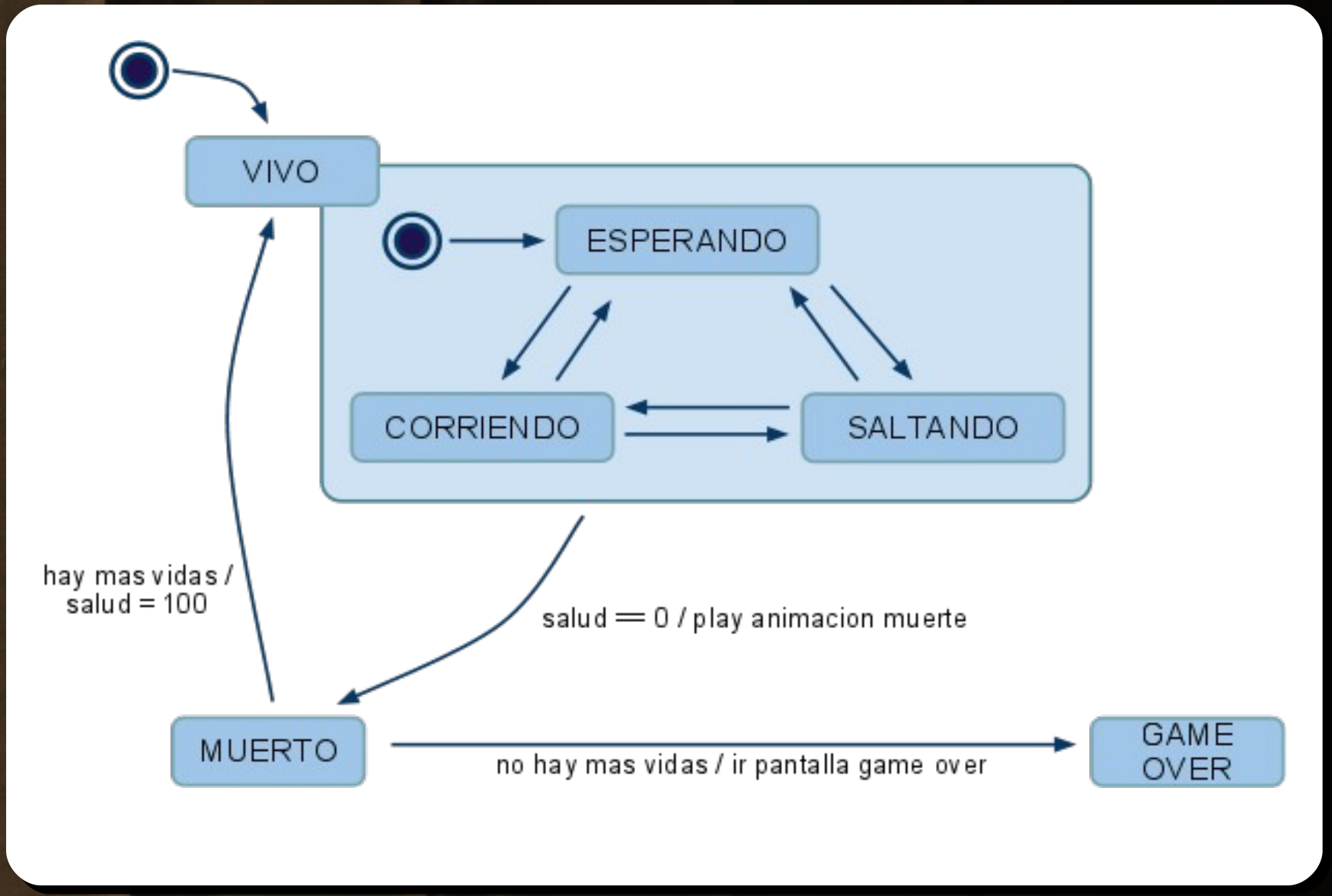
```
1   public class Actor extends Sprite implements IPoolable
2   {
3       public function init(..args):void
4       {
5           x = args[0];
6           y = args[1];
7       }
8
9       public function reset(..args):void
10      {
11          //x = args[0];
12          //y = args[1];
13          init.apply(this,args);
14
15          z = rotation = 0;
16          scaleX = scaleY = scaleZ = 1;
17          //...
18      }
19  }
20
21  public class Koopa extends Actor
22  {...}
23
24  var koopaPool:Pool = new Pool(Koopa);
25  var koopa:Koopa;
26
27  for(var i:int = 0; i<10; i++)
28  {
29      koopa = koopaPool.getInstance(i*10,10) as Koopa;
30      addChild(koopa);
31  }
```

State Machines

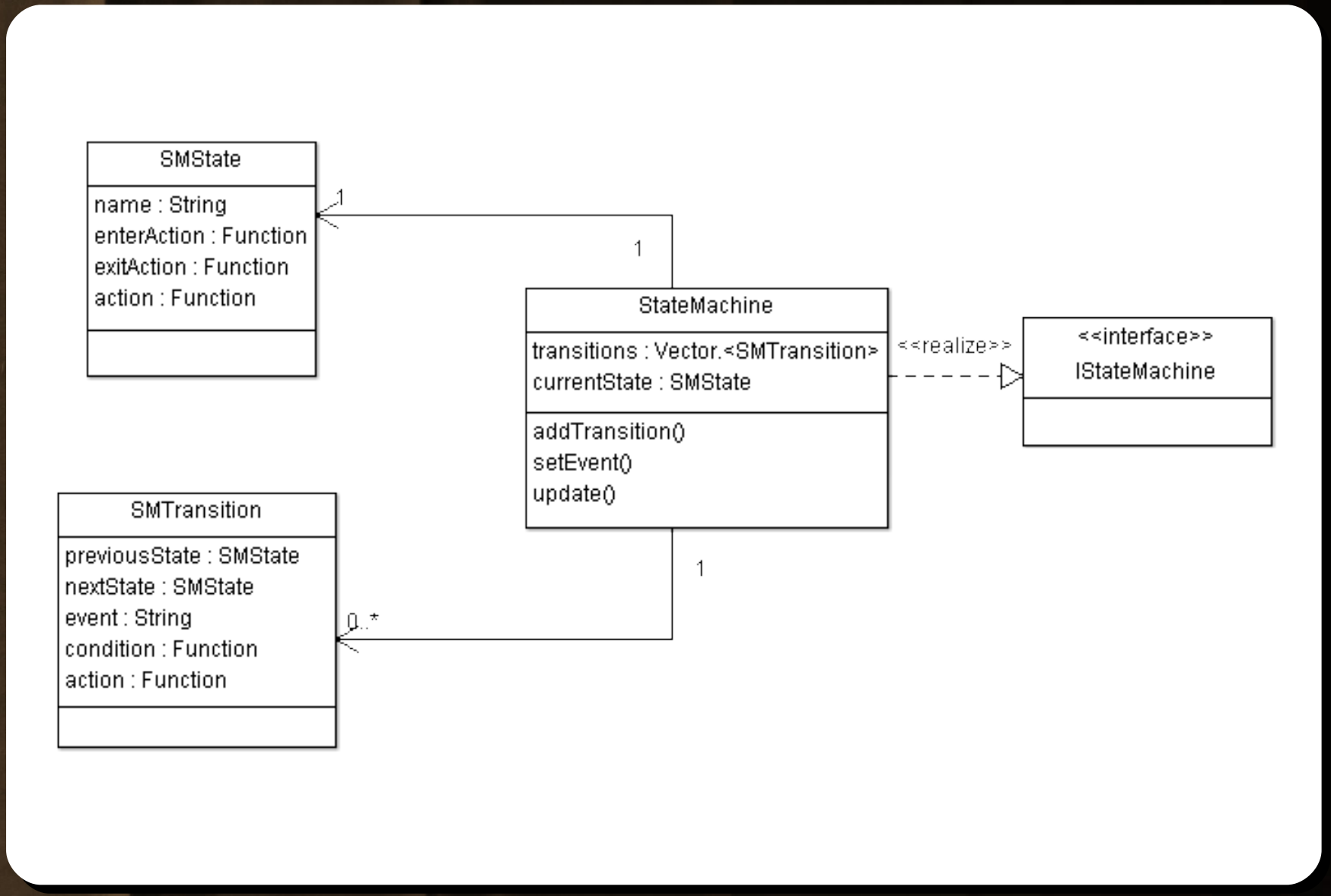


A state machine, is a **behavioral model** used to design computer programs.
It is composed of a finite number of **states** associated to **transitions**. A transition is a set of actions that starts from one state and ends in another (or the same) state. A transition is started by a **trigger**, and a trigger can be an **event** or a **condition**.

State Machines



State Machines



State Machines: State

```
1  package
2  {
3      public class SMState
4      {
5          private var _name:String;
6          private var _action:Function;
7          private var _enterAction:Function;
8          private var _exitAction:Function;
9
10         public function SMState(name:String,
11                                 action:Function = null,
12                                 enterAction:Function = null,
13                                 exitAction:Function = null)
14         {
15             _name = name;
16             _action = action;
17             _enterAction = enterAction;
18             _exitAction = exitAction;
19         }
20
21         public function get name():String { return _name; }
22
23         public function get action():Function { return _action; }
24
25         public function get enterAction():Function { return _enterAction; }
26
27         public function get exitAction():Function { return _exitAction; }
28
29     }
30
31 }
```

State Machines: Transition

```
1  package
2  {
3      public class SMTransition
4      {
5          private var _nextState:SMState;
6          private var _previousState:SMState;
7          private var _event:String;
8          private var _description:String;
9          private var _condition:Function;
10         private var _action:Function;
11
12         public function SMTransition(previousState:SMState,
13                                      nextState:SMState,
14                                      event:String,
15                                      condition:Function = null,
16                                      action:Function = null)
17         {
18             _previousState = previousState;
19             _nextState = nextState;
20             _condition = condition;
21             _action = action;
22             _event = event;
23         }
24
25         public function get nextState():SMState { return _nextState; }
26
27         public function get previousState():SMState { return _previousState; }
28
29         public function get event():String { return _event; }
30
31         public function get condition():Function { return _condition; }
32
33         public function get action():Function { return _action; }
34
35     }
36 }
```

State Machines: Machine

```
1  package
2  {
3      public interface IStateMachine
4      {
5          function addTransition(previousState:SMState,
6                                 nextState:SMState,
7                                 event:String,
8                                 condition:Function = null,
9                                 action:Function = null):void;
10
11         function update():void;
12
13         function setEvent(event:String):void;
14
15         function get state():SMState;
16     }
17 }
```

State Machines: Machine

```
1  package
2  {
3      public class StateMachine implements IStateMachine {
4
5          private var transitions:Vector.<SMTransition>;
6          private var currentState:SMState;
7
8          public function StateMachine(firstState:SMState) {
9              currentState = firstState;
10             transitions = new <SMTransition>[];
11         }
12
13         public function addTransition(previousState:SMState, nextState:SMState, event:String, condition:Function = null, action:Function = null):void {
14             transitions.push(new SMTransition(previousState, nextState, event, condition, action));
15         }
16
17         public function update():void {
18             for each (var transition:SMTransition in transitions) {
19                 evaluateTransition(transition);
20             }
21             if (currentState.action) currentState.action();
22         }
23
24         public function setEvent(event:String):void {
25             for each (var transition:SMTransition in transitions) {
26                 if (transition.event == event) evaluateTransition(transition);
27             }
28         }
29
30         private function evaluateTransition(transition:SMTransition):void {
31             if(transition.previousState === currentState){
32                 if (transition.condition) {
33                     if (transition.condition()) setNextState(transition);
34                 }
35                 else setNextState(transition);
36             }
37         }
38
39         private function setNextState(transition:SMTransition):void {
40             if (currentState.exitAction) currentState.exitAction();
41             if (transition.action) transition.action();
42             if (transition.nextState.enterAction) transition.nextState.enterAction();
43             currentState = transition.nextState;
44         }
45
46         public function get state():SMState {return currentState;}
47     }
```

State Machines

```
1  public class Mario extends StateMachine
2  {
3      public function Mario():void
4      {
5          var normal:SMState = new SMState("normal");
6          var running:SMState = new SMState("running",whenRunning);
7          var jumping:SMState = new SMState("jumping",whenJumping);
8          var dead:SMState = new SMState("dead",onDead);
9
10         super(normal);
11
12         addTransition(normal,running,"",KeyboardManager.manager.leftIsPressed);
13         addTransition(normal,jumping,"",KeyboardManager.manager.upIsPressed);
14         addTransition(running,jumping,"",KeyboardManager.manager.upIsPressed);
15         // ...
16         addTransition(running,dead,"touched-enemy");
17     }
18
19     private function whenRunning():void
20     {
21         //animate running
22         this.x += 10;
23         for each(var enemy:Enemy in Game.manager.enemies)
24         {
25             if(this.hitTest(enemy)) setEvent("touched-enemy")
26         }
27     }
28
29     private function whenJumping():void
30     {/*animate jumping*/}
31
32     private function onDead():void
33     {
34         if(this.lifes==0) {/*GAME OVER*/}
35         else {/*restart level... */}
36     }
37
38 }
```

State Machines

On display Objects:

```
1  public class Mario extends Sprite implements IStateMachine
2  {
3      private var machine:StateMachine;
4
5      public function Mario():void
6      {
7          var normal:SMState = new SMState("normal");
8          var running:SMState = new SMState("running",whenRunning);
9          var jumping:SMState = new SMState("jumping",whenJumping);
10         var dead:SMState = new SMState("dead",onDead);
11
12         machine = new StateMachine(normal);
13
14         addTransition(normal,running,"",KeyboardManager.manager.leftIsPressed);
15         addTransition(normal,jumping,"",KeyboardManager.manager.upIsPressed);
16         addTransition(running,jumping,"",KeyboardManager.manager.upIsPressed);
17         // ...
18         addTransition(running,dead,"touched-enemy");
19     }
20
21     //....
25
26     function addTransition(previousState:SMState,
27                            nextState:SMState,
28                            event:String,
29                            condition:Function = null,
30                            action:Function = null):void
31    {machine.addTransition(previousState,nextState,event,condition,action)}
32
33     function update():void {machine.update()}
34
35     function setEvent(event:String):void {machine.setEvent(event)}
36
37     function get state():SMState {return machine.state}
38 }
```

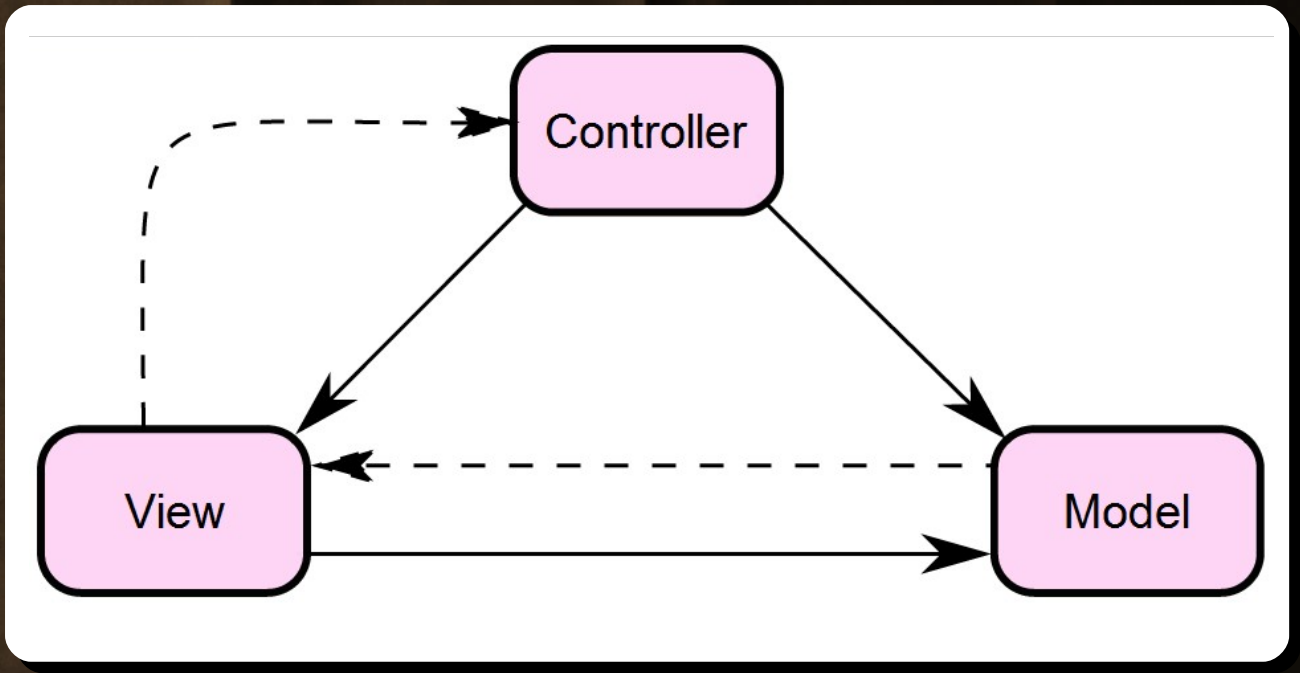
State Machines

Add to main loop:

```
var mario:Mario = new Mario();
addEventlistener(Event.ENTER_FRAME, gameLoop);

function gameLoop(e:Event):void
{
    mario.update();
}
```

Model-View-Controller



MVC is an architectural pattern that isolates "**domain logic**" from the **user interface** (input and presentation), permitting **independent development**, testing and maintenance of each (**separation of concerns**).

Model-View-Controller

```as3
public class MarioAsset extends Sprite
{
    public function animateRun():void{}
    public function animateJump():void{}
}

public class MarioData
{
    public var health:Number = 100;
    public var lifes:Number = 3;

    public function hurt():void{
        health-=10;
        if(health==0) {
            lifes--;
            health = 100;
        }
    }
}

public class MarioController extends AbstractActorController implements IStateMachine
{
    private var machine:StateMachine;
    private var data:MarioData;
    private var asset:MarioAsset;
    //...

    private function whenRunning():void
    {
        //animate running
        asset.x += 10;
        for each(var enemy:Enemy in Game.manager.enemies)
        {
            if(asset.hitTest(enemy)) setEvent("touched-enemy")
        }
    }

    private function onDead():void
    {
        if(data.lifes==0) {/*GAME OVER*/}
        else {/*restart level... */}
    }
}
```

Generally...

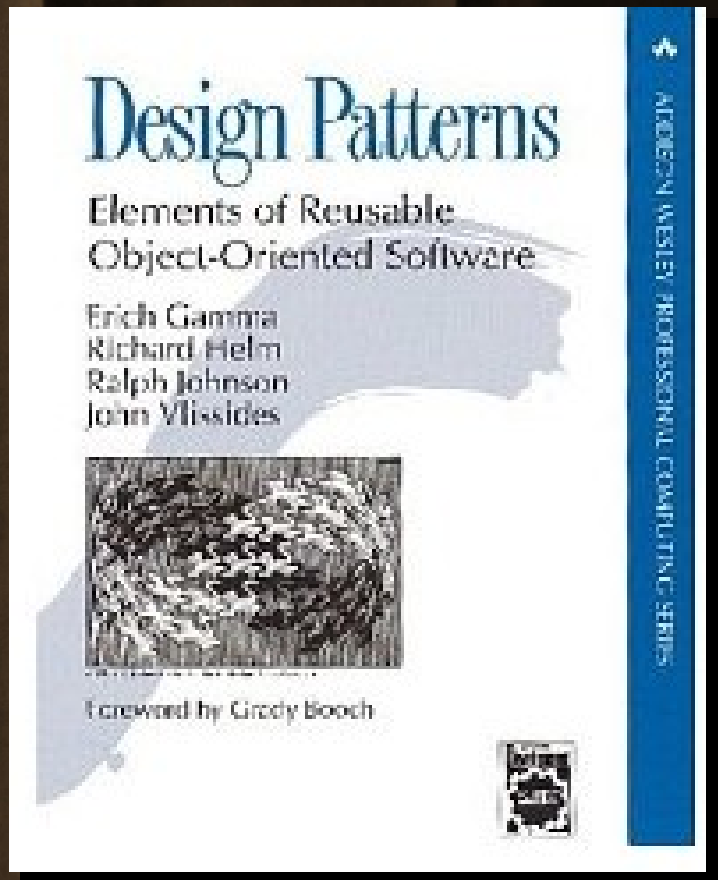
KISS
Keep It Simple & Stupid

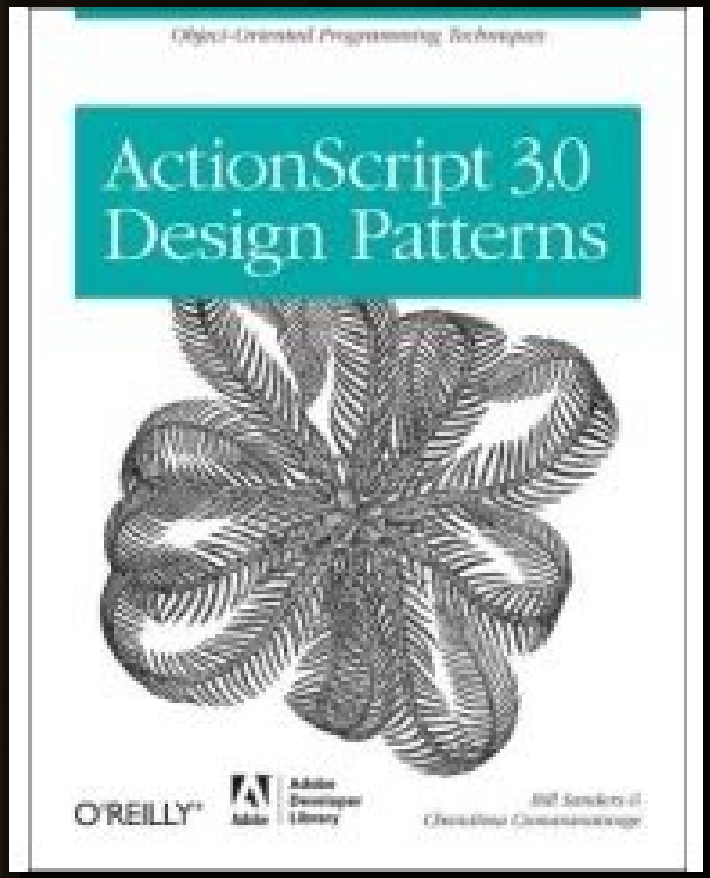
COC
Convention Over Configuration

DRY
Don't Repeat Yourself


KEEP
CALM
AND
CODE
ON

Read This!


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


ActionScript 3.0
Design Patterns
O'REILLY

Thanks!

github.com/matix/as3basics

matias.figueroa@globant.com
@matixfigueroa