Richard Lord
Screenwriter, Choreographer,
Game Developer

What is an Entity Component System architecture for game development?

Posted on **19 January 2012**

Last week I released Ash, an entity component system framework for Actionscript game development, and a number of people have asked me the question "What is an entity component system framework?". This is my rather long answer.

Entity systems are growing in popularity, with well-known examples like Unity, and lesser known frameworks like Actionscript frameworks Ember2, Xember and my own Ash. There's a very good reason for this; they simplify game architecture, encourage clean separation of responsibilities in your code, and are fun to use.

In this post I will walk you through how an entity based architecture evolves from the old fashioned game loop. This may take a while. The examples will be in Actionscript because that happens to be what I'm using at the moment, but the architecture applies to all programming language.

Note that the naming of things within this post is based on

- How they were named as I discovered these archtectures over the past twenty years of my game development life
- How they are usually named in modern entity component system architectures

This is different, for example, to how they are named in Unity, which is an entity architecture but is not an entity component system architecture.

This is based on a presentation I gave at try{harder} in 2011.

The examples

Throughout this post, I'll be using a simple Asteroids game as an example. I like to use Asteroids as an example because it involves simplified versions of many of the systems required in larger games - rendering, physics, ai, user control of a character, non-player characters.

The game loop

To understand why we use entity systems, you really need to understand the old-fashioned game loop. A game loop for Asteroids might look something like this

```
function update( time:Number ):void
{
  game.update( time );
  spaceship.updateInputs( time );
  for each( var flyingSaucer:FlyingSaucer in flyingSaucers )
  {
    flyingSaucer.updateAI( time );
  }
  spaceship.update( time );
  for each( var flyingSaucer:FlyingSaucer in flyingSaucers )
  {
    flyingSaucer.update( time );
  }
  for each( var asteroid:Asteroid in asteroids )
  {
    asteroid.update( time );
  }
  for each( var bullet:Bullet in bullets )
  {
    bullet.update( time );
  }
  collisionManager.update( time );
  spaceship.render();
  for each( var flyingSaucer:FlyingSaucer in flyingSaucers )
  {
    flyingSaucer.render();
```

```
  }
  for each( var asteroid:Asteroid in asteroids )
  {
    asteroid.render();
  }
  for each( var bullet:Bullet in bullets )
  {
    bullet.render();
  }
}
```

This game loop is called on a regular interval, usually every 60th of a second or every 30th of a second, to update the game. The order of operations in the loop is important as we update various game objects, check for collisions between them, and then draw them all. Every frame.

This is a very simple game loop. It's simple because

1. The game is simple
2. The game has only one state

In the past, I have worked on console games where the game loop, a single function, was over 3,000 lines of code. It wasn't pretty, and it wasn't clever. That's the way games were built and we had to live with it.

Entity system architecture derives from an attempt to resolve the problems with the game loop. It addresses the game loop as the core of the game, and pre-supposes that simplifying the game loop is more important than anything else in modern game architecture. More important than separation of the view from the controller, for example.

Processes

The first step in this evolution is to think about objects called processes. These are objects that can be initialised, updated on a regular basis, and destroyed. The interface for a process looks something like this.

```
interface IProcess
{
  function start():Boolean;
  function update( time:Number ):void;
```

```
  function end():void;
}
```

We can simplify our game loop if we break it into a number of processes to handle, for example, rendering, movement, collision resolution. To manage those processes we create a process manager.

```
class ProcessManager
{
  private var processes:PrioritisedList;

  public function addProcess( process:IProcess, priority:int ):Boolean
  {
    if( process.start() )
    {
      processes.add( process, priority );
      return true;
    }
    return false;
  }

  public function update( time:Number ):void
  {
    for each( var process:IProcess in processes )
    {
      process.update( time );
    }
  }

  public function removeProcess( process:IProcess ):void
  {
    process.end();
    processes.remove( process );
  }
}
```

This is a somewhat simplified version of a process manager. In particular, we should ensure we update the processes in the correct order (identified by the priority parameter in the add method) and we should handle the situation where a process is removed during the update loop. But you get the idea. If our game loop is broken into multiple processes, then the

update method of our process manager is our new game loop and the processes become the core of the game.

The render process

Lets look at the render process as an example. We could just pull the render code out of the original game loop and place it in a process, giving us something like this

```
class RenderProcess implements IProcess
{
  public function start() : Boolean
  {
    // initialise render system
    return true;
  }

  public function update( time:Number ):void
  {
    spaceship.render();
    for each( var flyingSaucer:FlyingSaucer in flyingSaucers )
    {
      flyingSaucer.render();
    }
    for each( var asteroid:Asteroid in asteroids )
    {
      asteroid.render();
    }
    for each( var bullet:Bullet in bullets )
    {
      bullet.render();
    }
  }

  public function end() : void
  {
    // clean-up render system
  }
}
```

Using an interface

But this isn't very efficient. We still have to manually render all the different types of game object. If we have a common interface for all renderable objects, we can simplify matters a lot.

```
interface IRenderable
{
  function render();
}
```

```
class RenderProcess implements IProcess
{
  private var targets:Vector.<IRenderable>;

  public function start() : Boolean
  {
    // initialise render system
    return true;
  }

  public function update( time:Number ):void
  {
    for each( var target:IRenderable in targets )
    {
      target.render();
    }
  }

  public function end() : void
  {
    // clean-up render system
  }
}
```

Then our spaceship class might contain some code like this

```
class Spaceship implements IRenderable
{
  public var view:DisplayObject;
```

```
  public var position:Point;
  public var rotation:Number;

  public function render():void
  {
    view.x = position.x;
    view.y = position.y;
    view.rotation = rotation;
  }
}
```

This code is based on the flash display list. If we were blitting, or using stage3d, it would be different, but the principles would be the same. We need the image to be rendered, and the position and rotation for rendering it. And the render function does the rendering.

Using a base class and inheritance

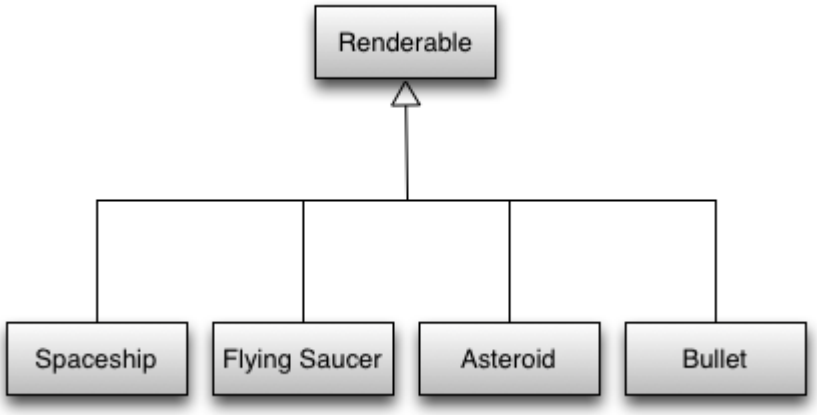
In fact, there's nothing in this code that makes it unique to a spaceship. All the code could be shared by all renderable objects. The only thing that makes them different is which display object is assigned to the view property, and what the position and rotation are. So lets wrap this in a base class and use inheritance.

```
class Renderable implements IRenderable
{
  public var view:DisplayObject;
  public var position:Point;
  public var rotation:Number;

  public function render():void
  {
    view.x = position.x;
    view.y = position.y;
    view.rotation = rotation;
  }
}
```

```
class Spaceship extends Renderable
{
}
```

Of course, all renderable items will extend the renderable class, so we get a simple class heirarchy like this



The move process

To understand the next step, we first need to look at another process and the class it works on. So lets try the move process, which updates the position of the objects.

```
interface IMoveable
{
  function move( time:Number );
}
```

```
class MoveProcess implements IProcess
{
  private var targets:Vector.<IMoveable>;

  public function start():Boolean
  {
    return true;
  }

  public function update( time:Number ):void
  {
    for each( var target:IMoveable in targets )
    {
      target.move( time );
    }
  }
```

```
  public function end():void
  {
  }
}
```

```
class Moveable implements IMoveable
{
  public var position:Point;
  public var rotation:Number;
  public var velocity:Point;
  public var angularVelocity:Number;

  public function move( time:Number ):void
  {
    position.x += velocity.x * time;
    position.y += velocity.y * time;
    rotation += angularVelocity * time;
  }
}
```

```
class Spaceship extends Moveable
{
}
```

Multiple inheritance

That's almost good, but unfortunately we want our spaceship to be both moveable and renderable, and many modern programming languages don't allow multiple inheritance.

Even in those languages that do permit multiple inheritance, we have the problem that the position and rotation in the Moveable class should be the same as the position and rotation in the Renderable class.

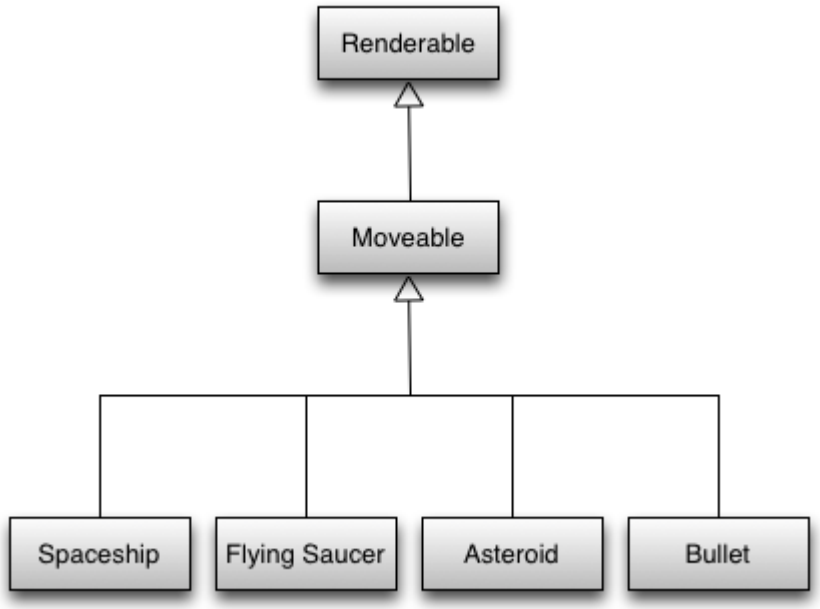
One common solution is to use an inheritance chain, so that Moveable extends Renderable.

```
class Moveable extends Renderable implements IMoveable
{
  public var velocity:Point;
  public var angularVelocity:Number;
```

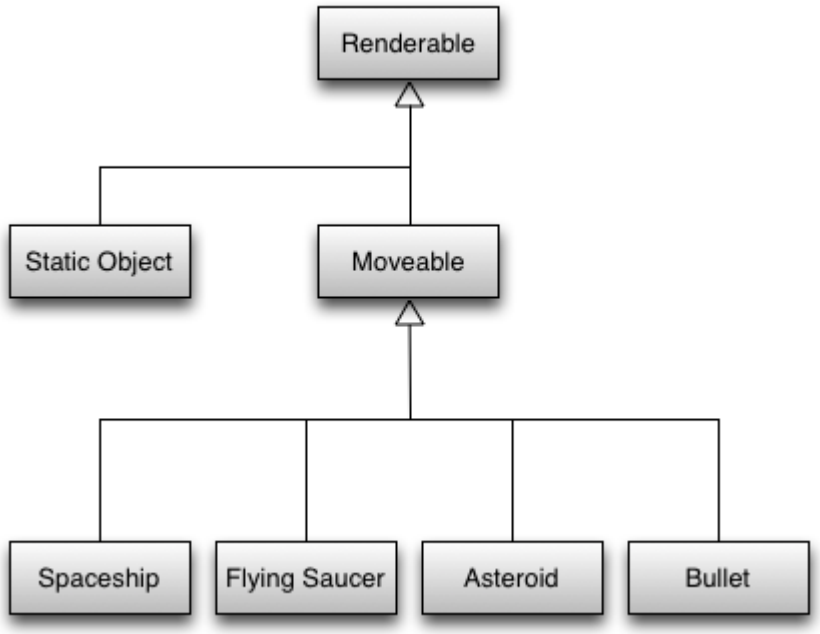
```
  public function move( time:Number ):void
  {
    position.x += velocity.x * time;
    position.y += velocity.y * time;
    rotation += angularVelocity * time;
  }
}
```

```
class Spaceship extends Moveable
{
}
```

Now the spaceship is both moveable and renderable. We can apply the same principles to the other game objects to get this class hierarchy.



We can even have static objects that just extend Renderable.

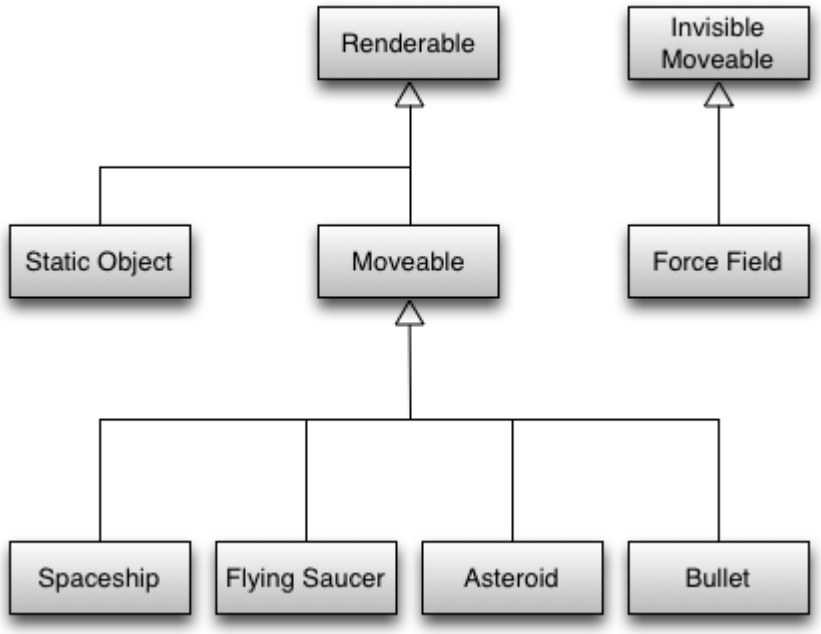


Moveable but not Renderable

But what if we want a Moveable object that isn't Renderable? An invisible game object, for example? Now our class hierarchy breaks down and we need an alternative implementation of the Moveable interface that doesn't extend Renderable.

```
class InvisibleMoveable implements IMoveable
{
  public var position:Point;
  public var rotation:Number;
  public var velocity:Point;
  public var angularVelocity:Number;

  public function move( time:Number ):void
  {
    position.x += velocity.x * time;
    position.y += velocity.y * time;
    rotation += angularVelocity * time;
  }
}
```



In a simple game, this is clumsy but manageable, but in a complex game using inheritance to apply the processes to objects rapidly becomes unmanageable as you'll soon discover items in your game that don't fit into a simple linear inheritance tree, as with the force-field above.

Favour composition over inheritance

It's long been a sound principle of object-oriented programming to favour composition over inheritance. Applying that principle here can rescue us from this potential inheritance mess.

We'll still need Renderable and Moveable classes, but rather than extending these classes to create the spaceship class, we will create a spaceship class that contains an instance of each of these classes.

```
class Renderable implements IRenderable
{
  public var view:DisplayObject;
  public var position:Point;
  public var rotation:Number;

  public function render():void
  {
    view.x = position.x;
    view.y = position.y;
    view.rotation = rotation;
```

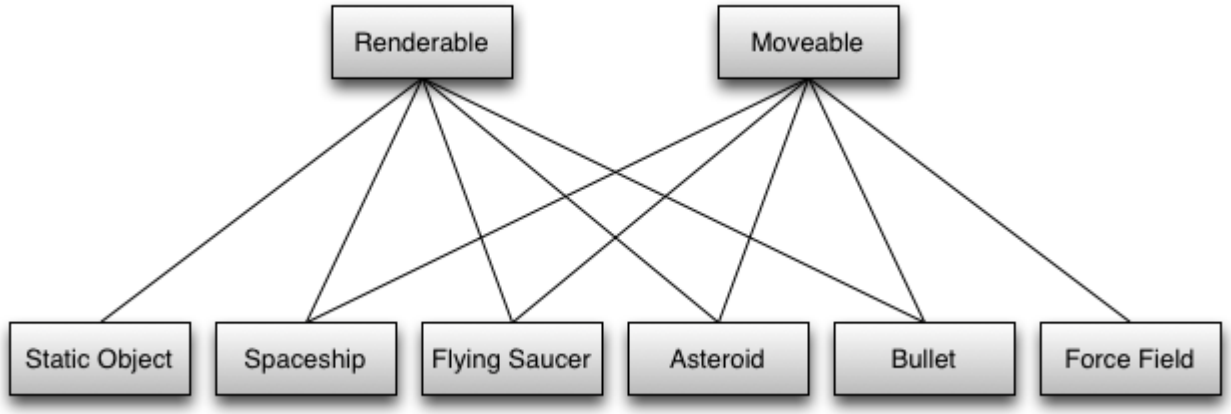
```
  }
}
```

```
class Moveable implements IMoveable
{
  public var position:Point;
  public var rotation:Number;
  public var velocity:Point;
  public var angularVelocity:Number;

  public function move( time:Number ):void
  {
    position.x += velocity.x * time;
    position.y += velocity.y * time;
    rotation += angularVelocity * time;
  }
}
```

```
class Spaceship
{
  public var renderData:IRenderable;
  public var moveData:IMoveable;
}
```

This way, we can combine the various behaviours in any way we like without running into inheritance problems.



The objects made by this composition, the Static Object, Spaceship, Flying Saucer, Asteroid, Bullet and Force Field, are collectively called entities.

Our processes remain unchanged.

```
interface IRenderable
{
  function render();
}
```

```
class RenderProcess implements IProcess
{
  private var targets:Vector.<IRenderable>;

  public function update(time:Number):void
  {
    for each(var target:IRenderable in targets)
    {
      target.render();
    }
  }
}
```

```
interface IMoveable
{
  function move();
}
```

```
class MoveProcess implements IProcess
{
  private var targets:Vector.<IMoveable>;

  public function update(time:Number):void
  {
    for each(var target:IMoveable in targets)
    {
      target.move( time );
    }
  }
}
```

But we don't add the spaceship entity to each process, we add it's components. So when we create the spaceship we do something like this

```
public function createSpaceship():Spaceship
{
  var spaceship:Spaceship = new Spaceship();
  ...
  renderProcess.addItem( spaceship.renderData );
  moveProcess.addItem( spaceship.moveData );
  ...
  return spaceship;
}
```

This approach looks good. It gives us the freedom to mix and match process support between different game objects without getting into spagetti inheritance chains or repeating ourselves. But there's one problem.

What about the shared data?

The position and rotation properties in the Renderable class instance need to have the same values as the position and rotation properties in the Moveable class instance, since the Move process will change the values in the Moveable instance and the Render process will use the values in the Renderable instance.

```
class Renderable implements IRenderable
{
  public var view:DisplayObject;
  public var position:Point;
  public var rotation:Number;

  public function render():void
  {
    view.x = position.x;
    view.y = position.y;
    view.rotation = rotation;
  }
}
```

```
class Moveable implements IMoveable
{
  public var position:Point;
  public var rotation:Number;
```

```
  public var velocity:Point;
  public var angularVelocity:Number;

  public function move( time:Number ):void
  {
    position.x += velocity.x * time;
    position.y += velocity.y * time;
    rotation += angularVelocity * time;
  }
}
```

```
class Spaceship
{
  public var renderData:IRenderable;
  public var moveData:IMoveable;
}
```

To solve this, we need to ensure that both class instances reference the same instances of these properties. In Actionscript that means these properties must be objects, because objects can be passed by reference while primitives are passed by value.

So we introduce another set of classes, which we'll call components. These components are just value objects that wrap properties into objects for sharing between processes.

```
class PositionComponent
{
  public var x:Number;
  public var y:Number;
  public var rotation:Number;
}
```

```
class VelocityComponent
{
  public var velocityX:Number;
  public var velocityY:Number;
  public var angularVelocity:Number;
}
```

```
class DisplayComponent
{
```

```
  public var view:DisplayObject;
}
```

```
class Renderable implements IRenderable
{
  public var display:DisplayComponent;
  public var position:PositionComponent;

  public function render():void
  {
    display.view.x = position.x;
    display.view.y = position.y;
    display.view.rotation = position.rotation;
  }
}
```

```
class Moveable implements IMoveable
{
  public var position:PositionComponent;
  public var velocity:VelocityComponent;

  public function move( time:Number ):void
  {
    position.x += velocity.velocityX * time;
    position.y += velocity.velocityY * time;
    position.rotation += velocity.angularVelocity * time;
  }
}
```

When we create the spaceship we ensure the Moveable and Renderable instances share the same instance of the PositionComponent.

```
class Spaceship
{
  public function Spaceship()
  {
    moveData = new Moveable();
    renderData = new Renderable();
    moveData.position = new PositionComponent();
    moveData.velocity = new VelocityComponent();
    renderData.position = moveData.position;
```

```
    renderData.display = new DisplayComponent();
  }
}
```

The processes remain unaffected by this change.

A good place to pause

At this point we have a neat separation of tasks. The game loop cycles through the processes, calling the update method on each one. Each process contains a collection of objects that implement the interface it operates on, and will call the appropriate method of those objects. Those objects each do a single important task on their data. Through the system of components, those objects are able to share data and thus the combination of multiple processes can produce complex updates in the game entities, while keeping each process relatively simple.

This architecture is similar to a number of entity systems in game development. The architecture follows good object-oriented principles and it works. But there's more to come, starting with a moment of madness.

Abandoning good object-oriented practice

The current architecture uses good object-oriented practices like encapsulation and single responsibility - the IRenderable and IMoveable implementations encapsulate the data and logic for single responsibilities in the updating of game entities every frame - and composition - the Spaceship entity is created by combining implementations of the IRenderable and IMoveable interfaces. Through the system of components we ensured that, where appropriate, data is shared between the different data classes of the entities.

The next step in this evolution of entity systems is somewhat counter-intuitive, breaking one of the core tenets of object-oriented programming. We break the encapsulation of the data and logic in the Renderable and Moveable implementations. Specifically, we remove the logic from these classes and place it in the processes instead.

So this

```
interface IRenderable
{
```

```
  function render();
}
```

```
class Renderable implements IRenderable
{
  public var display:DisplayComponent;
  public var position:PositionComponent;

  public function render():void
  {
    display.view.x = position.x;
    display.view.y = position.y;
    display.view.rotation = position.rotation;
  }
}
```

```
class RenderProcess implements IProcess
{
  private var targets:Vector.<IRenderable>;

  public function update( time:Number ):void
  {
    for each( var target:IRenderable in targets )
    {
      target.render();
    }
  }
}
```

Becomes this

```
class RenderData
{
  public var display:DisplayComponent;
  public var position:PositionComponent;
}
```

```
class RenderProcess implements IProcess
{
  private var targets:Vector.<RenderData>;
```

```
  public function update( time:Number ):void
  {
    for each( var target:RenderData in targets )
    {
      target.display.view.x = target.position.x;
      target.display.view.y = target.position.y;
      target.display.view.rotation = target.position.rotation;
    }
  }
}
```

And this

```
interface IMoveable
{
  function move( time:Number );
}
```

```
class Moveable implements IMoveable
{
  public var position:PositionComponent;
  public var velocity:VelocityComponent;

  public function move( time:Number ):void
  {
    position.x += velocity.velocityX * time;
    position.y += velocity.velocityY * time;
    position.rotation += velocity.angularVelocity * time;
  }
}
```

```
class MoveProcess implements IProcess
{
  private var targets:Vector.<IMoveable>;

  public function move( time:Number ):void
  {
    for each( var target:Moveable in targets )
    {
      target.move( time );
```

```
    }
  }
}
```

Becomes this

```
class MoveData
{
  public var position:PositionComponent;
  public var velocity:VelocityComponent;
}
```

```
class MoveProcess implements IProcess
{
  private var targets:Vector.<MoveData>;

  public function move( time:Number ):void
  {
    for each( var target:MoveData in targets )
    {
      target.position.x += target.velocity.velocityX * time;
      target.position.y += target.velocity.velocityY * time;
      target.position.rotation += target.velocity.angularVelocity * time;
    }
  }
}
```

It's not immediately clear why we'd do this, but bear with me. On the surface, we've removed the need for the interface, and we've given the process something more important to do - rather than simply delegate its work to the IRenderable or IMoveable implementations, it does the work itself.

The first apparent consequence of this is that all entities must use the same rendering method, since the render code is now in the RenderProcess. But that's not actually the case. We could, for example, have two processes, RenderMovieClip and RenderBitmap for example, and they could operate on different sets of entities. So we haven't lost any flexibility.

What we gain is the ability to refactor our entities significantly to produce an architecture with clearer separation and simpler configuration. The refactoring starts with a question.

Do we need the data classes?

Currently, our entity

```
class Spaceship
{
  public var moveData:MoveData;
  public var renderData:RenderData;
}
```

Contains two data classes

```
class MoveData
{
  public var position:PositionComponent;
  public var velocity:VelocityComponent;
}
```

```
class RenderData
{
  public var display:DisplayComponent;
  public var position:PositionComponent;
}
```

These data classes in turn contain three components

```
class PositionComponent
{
  public var x:Number;
  public var y:Number;
  public var rotation:Number;
}
```

```
class VelocityComponent
{
  public var velocityX:Number;
  public var velocityY:Number;
  public var angularVelocity:Number;
}
```

```
class DisplayComponent
{
  public var view:DisplayObject;
}
```

And the data classes are used by the two processes

```
class MoveProcess implements IProcess
{
  private var targets:Vector.<MoveData>;

  public function move( time:Number ):void
  {
    for each( var target:MoveData in targets )
    {
      target.position.x += target.velocity.velocityX * time;
      target.position.y += target.velocity.velocityY * time;
      target.position.rotation += target.velocity.angularVelocity * time;
    }
  }
}
```

```
class RenderProcess implements IProcess
{
  private var targets:Vector.<RenderData>;

  public function update( time:Number ):void
  {
    for each( var target:RenderData in targets )
    {
      target.display.view.x = target.position.x;
      target.display.view.y = target.position.y;
      target.display.view.rotation = target.position.rotation;
    }
  }
}
```

But the entity shouldn't care about the data classes. The components collectively contain the state of the entity. The data classes exist for the convenience of the processes. So we

refactor the code so the spaceship entity contains the components rather than the data classes.

```
class Spaceship
{
  public var position:PositionComponent;
  public var velocity:VelocityComponent;
  public var display:DisplayComponent;
}
```

```
class PositionComponent
{
  public var x:Number;
  public var y:Number;
  public var rotation:Number;
}
```

```
class VelocityComponent
{
  public var velocityX:Number;
  public var velocityY:Number;
  public var angularVelocity:Number;
}
```

```
class DisplayComponent
{
  public var view:DisplayObject;
}
```

By removing the data classes, and using the constituent components instead to define the spaceship, we have removed any need for the spaceship entity to know what processes may act on it. The spaceship now contains the components that define its state. Any requirement to combine these components into other data classes for the processes is some other class's responsibility.

Systems and Nodes

Some core code within the entity system framework (which we'll get to in a minute) will dynamically create these data objects as they are required by the processes. In this reduced context, the data classes will be mere nodes in the collections (arrays, linked-lists, or otherwise, depending on the implementation) used by the processes. So to clarify this we'll rename them as nodes.

```
class MoveNode
{
  public var position:PositionComponent;
  public var velocity:VelocityComponent;
}
```

```
class RenderNode
{
  public var display:DisplayComponent;
  public var position:PositionComponent;
}
```

The processes are unchanged, but in keeping with the more common naming I'll also change their name and call them systems.

```
class MoveSystem implements ISystem
{
  private var targets:Vector.<MoveNode>;

  public function update( time:Number ):void
  {
    for each( var target:MoveNode in targets )
    {
      target.position.x += target.velocity.velocityX * time;
      target.position.y += target.velocity.velocityY * time;
      target.position.rotation += target.velocity.angularVelocity * time;
    }
  }
}
```

```
class RenderSystem implements ISystem
{
  private var targets:Vector.<RenderNode>;
```

```
  public function update( time:Number ):void
  {
    for each( var target:RenderNode in targets )
    {
      target.display.view.x = target.position.x;
      target.display.view.y = target.position.y;
      target.display.view.rotation = target.position.rotation;
    }
  }
}
```

```
interface ISystem
{
  function update( time:Number ):void;
}
```

And what is an entity?

One last change - there's nothing special about the Spaceship class. It's just a container for components. So we'll just call it Entity and give it a collection of components. We'll access those components based on their class type.

```
class Entity
{
  private var components : Dictionary;

  public function add( component:Object ):void
  {
    var componentClass : Class = component.constructor;
    components[ componentClass ] = component'
  }

  public function remove( componentClass:Class ):void
  {
    delete components[ componentClass ];
  }

  public function get( componentClass:Class ):Object
  {
    return components[ componentClass ];
```

```
  }
}
```

So we'll create our spaceship like this

```
public function createSpaceship():void
{
  var spaceship:Entity = new Entity();
  var position:PositionComponent = new PositionComponent();
  position.x = Stage.stageWidth / 2;
  position.y = Stage.stageHeight / 2;
  position.rotation = 0;
  spaceship.add( position );
  var display:DisplayComponent = new DisplayComponent();
  display.view = new SpaceshipImage();
  spaceship.add( display );
  engine.add( spaceship );
}
```

The core Engine class

We mustn't forget the system manager, formerly called the process manager.

```
class SystemManager
{
  private var systems:PrioritisedList;

  public function addSystem( system:ISystem, priority:int ):void
  {
    systems.add( system, priority );
    system.start();
  }

  public function update( time:Number ):void
  {
    for each( var system:ISystem in systemes )
    {
      system.update( time );
    }
  }
```

```
  public function removeSystem( system:ISystem ):void
  {
    system.end();
    systems.remove( system );
  }
}
```

This will be enhanced and will sit at the heart of our entity component system framework. We'll add to it the functionality mentioned above to dynamically create nodes for the systems.

The entities only care about components, and the systems only care about nodes. So to complete the entity component system framework, we need code to watch the entities and, as they change, add and remove their components to the node collections used by the systems. Because this is the one bit of code that knows about both entities and systems, we might consider it central to the game. In Ash, I call this the Engine class, and it is an enhanced version of the system manager.

Every entity and every system is added to and removed from the Engine class when you start using it and stop using it. The Engine class keeps track of the components on the entities and creates and destroys nodes as necessary, adding those nodes to the node collections. The Engine class also provides a way for the systems to get the collections they require.

```
public class Engine
{
  private var entities:EntityList;
  private var systems:SystemList;
  private var nodeLists:Dictionary;

  public function addEntity( entity:Entity ):void
  {
    entities.add( entity );
    // create nodes from this entity's components and add them to node lists
    // also watch for later addition and removal of components from the entity so
    // you can adjust its derived nodes accordingly
  }

  public function removeEntity( entity:Entity ):void
  {
    // destroy nodes containing this entity's components
    // and remove them from the node lists
```

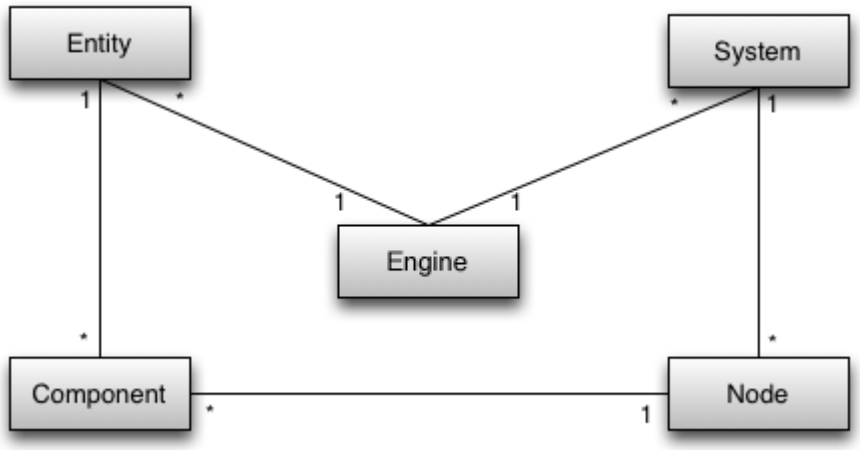
```
    entities.remove( entity );
  }

  public function addSystem( system:System, priority:int ):void
  {
    systems.add( system, priority );
    system.start();
  }

  public function removeSystem( system:System ):void
  {
    system.end();
    systems.remove( system );
  }

  public function getNodeList( nodeClass:Class ):NodeList
  {
    var nodes:NodeList = new NodeList();
    nodeLists[ nodeClass ] = nodes;
    // create the nodes from the current set of entities
    // and populate the node list
    return nodes;
  }

  public function update( time:Number ):void
  {
    for each( var system:ISystem in systemes )
    {
      system.update( time );
    }
  }
}
```



To see one implementation of this architecture, checkout the Ash entity system framework, and see the example Asteroids implementation there too.

A step further

In Actionscript, the Node and Entity classes are necessary for efficiently managing the Components and passing them to the Systems. But note that these classes are just glue, the game is defined in the Systems and the Components. The Entity class provides a means to find and manage the components for each entity and the Node classes provide a means to group components into collections for use in the Systems. In other languages and runtime environments it may be more efficient to manage this glue differently.

For example, in a large server-based game we might store the components in a database - they are just data after all - with each record (i.e. each component) having a field for the unique id of the entity it belongs to along with fields for the other component data. Then we pull the components for an entity directly from the database when needed, using the entity id to find it, and we create collections of data for the systems to operate on by doing joined queries across the appropriate databases. For example, for the move system we would pull records from the postion components table and the movement components table where entity ids match and a record exists in both tables (i.e. the entity has both a position and a movement component). In this instance the Entity and Node classes are not required and the only presence for the entity is the unique id that is used in the data tables.

Similarly, if you have control over the memory allocation for your game it is often more efficient to take a similar approach for local game code too, creating components in native arrays of data and looking-up the components for an entity based on an id. Some aspects of the game code become more complex and slower (e.g. finding the components for a specific

entity) but others become much faster (e.g. iterating through the component data collections inside a system) because the data is efficiently laid out in memory to minimise cache misses and maximise speed.

The important elements of this architecture are the components and the systems. Everything else is configuration and glue. And note that components are data and systems are functions, so we don't even need object oriented code to do this.

Conclusion

So, to summarise, entity component systems originate from a desire to simplify the game loop. From that comes an architecture of components, which represent the state of the game, and systems, which operate on the state of the game. Systems are updated every frame - this is the game loop. Components are combined into entities, and systems operate on the entities that have all the components they are interested in. The engine monitors the systems and the components and ensures each system has access to a collection of all the components it needs.

An **entity component system framework** like Ash provides the basic scaffolding and core management for this architecture, without providing any actual component or system classes. You create your game by creating the appropriate components and systems.

An **entity component system game engine** will provide many standard systems and components on top of the basic framework.

Three entity component system frameworks for Actionscript are my own Ash, Ember2 by Tom Davies and Xember by Alec McEachran. Artemis is an entity system framework for Java, that has also been ported to C#.

My next post covers some of the reasons why I like using an entity system framework for my game development projects.

Share this post or a comment online -

Also in the collection **Entity-Component-System architecture**

- Ash - a new entity system framework for Actionscript games

- Why use an Entity Component System architecture for game development?
- Finite State Machines with Ash entity component system framework

---

Richard Lord - Screenwriter, Choreographer, Game Developer - www.richardlord.net