**Game Architecture Overview**

The game is segmented into five main groups

* The entities in the game
* The components used by the entities
* The entity managers
* The state machines
* AI

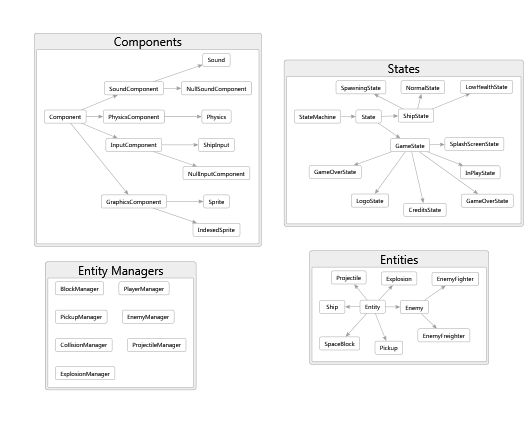


Figure 1 The non-AI groups and the classes they contain

**The Entities**

The entity classes represent the moving visible objects on the screen. The entity classes are Ship, SpaceBlock, Projectile, Enemy, EnemyFighter, EnemyFreighter, Explosion and Pickup. These classes are subclasses of the Entity class.

Information common to all entities is stored in the Entity class. This is information such as:

* Velocity
* Position
* Height
* Width
* Rotation

The Entity class is extended / sub-classed by every entity. The Entity class constructor takes several abstract components for drawing, playing sound, handling user input, etc. The Entity base class calls update on all of these components when its Update method is called. It calls draw on its graphics component when its Draw method is called.

The specific components to be used by the base class are created by the entity subclass.

An entity subclass fills in the components used by the Entity base class. An entity subclass builds itself by creating components and passing them to the base class constructor.

One way to look at it, is that Entity is a shell of a car, just the frame. The actual entity creates a new frame and populates it with features (doors, windows, etc).

**The Components**

The abstract component types are:

* GraphicsComponent (draws the entity)
* InputComponent (handles input)
* PhysicsComponent (updates the position of the entity)
* SoundComponent (plays sounds (if any) for the entity)

In otherwords, the entities are composed of data and the components which operate on that data.

Figure 2 An Entity is Composed of Components

This approach is called the Component pattern and it has become popular in recent years for game programming.

See <http://gameprogrammingpatterns.com/component.html> for more information about this pattern.

**The Entity Managers**

To avoid having lots of code in the main Game class dealing with explosions and lasers and ships, each entity has an associated class called a manager. Instead of looping through each of these entities in the update and draw methods, the game calls Update and Draw on each manager.

The manager class keeps track of instances of its associated entity.

ExplosionManager for example has methods for adding and removing explosions from the screen. The block manager can spawn a number of blocks to the screen.

These managers are used to decouple the main Update method from all of the individual entities.

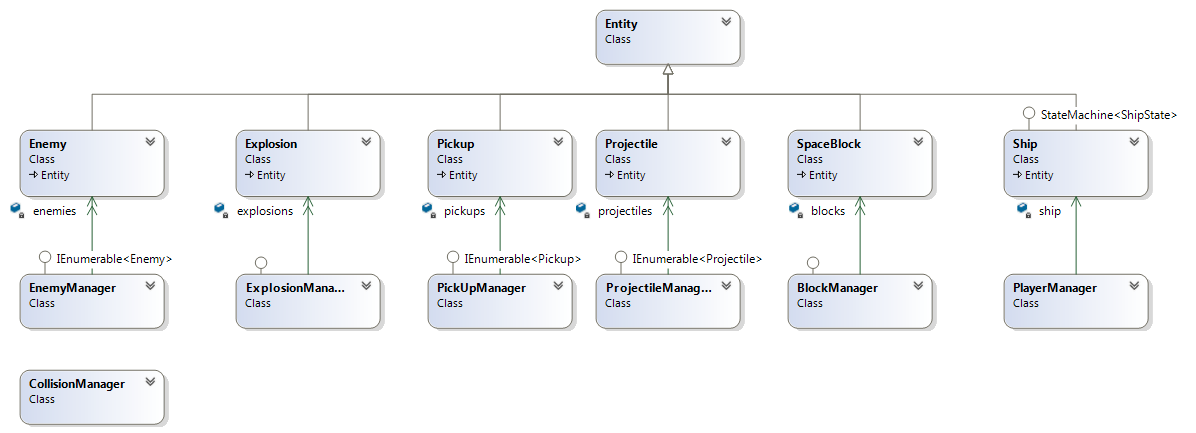


Figure 3 The Managers and the Entities they Manage

The managers are an example of the "update method" pattern: <http://gameprogrammingpatterns.com/update-method.html>

**The Finite State Machines**

The finite state machines are currently Game and Ship. Game uses states to transition between the splash screen, the main game, and the game over screen.

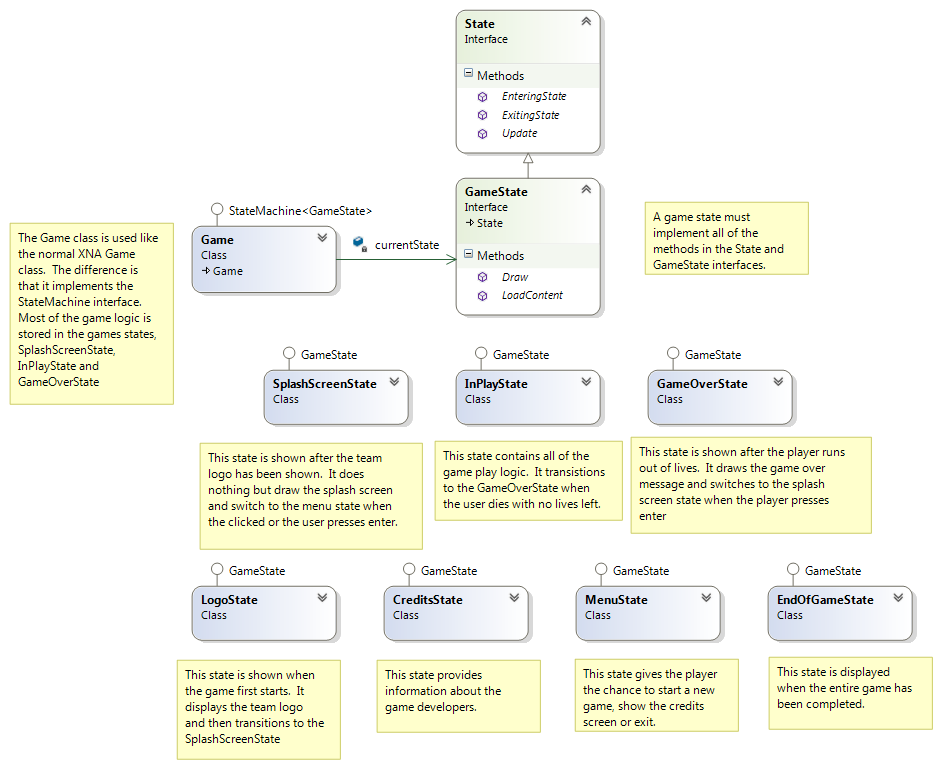
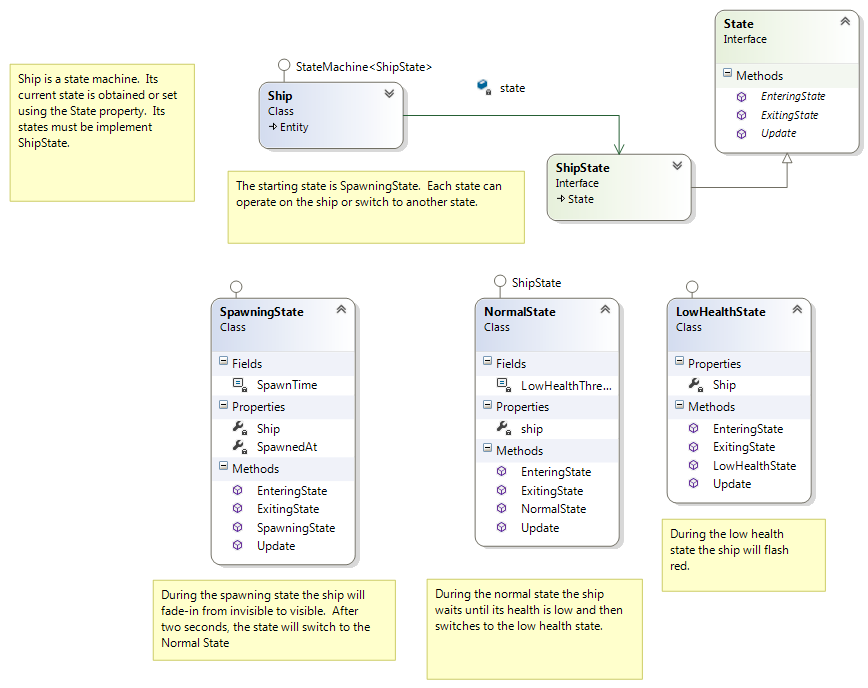


Figure 4 Game States

The ship uses states to transition between spawning, having its normal appearance, and being near death (blinking red).

Figure 5 Ship States

This is an example of the State pattern.

[http://gameprogrammingpatterns.com/state.html#the-state-pattern](http://gameprogrammingpatterns.com/state.html" \l "the-state-pattern)

**The StateMachine and State Interfaces**

StateMachine and State are interfaces which are only used when creating a new state machine or adding a new state to an existing state machine.

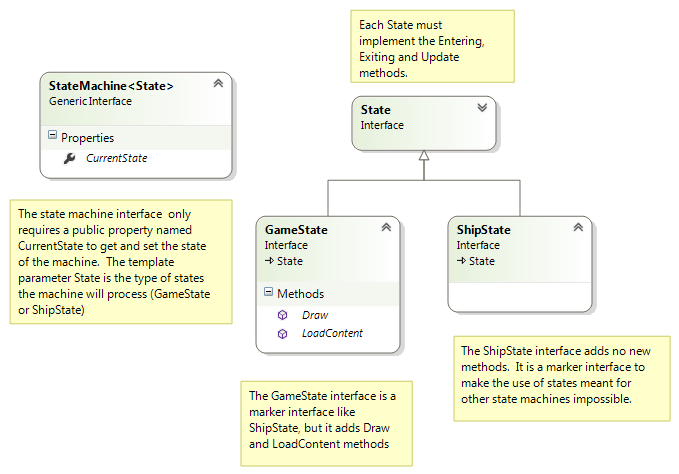
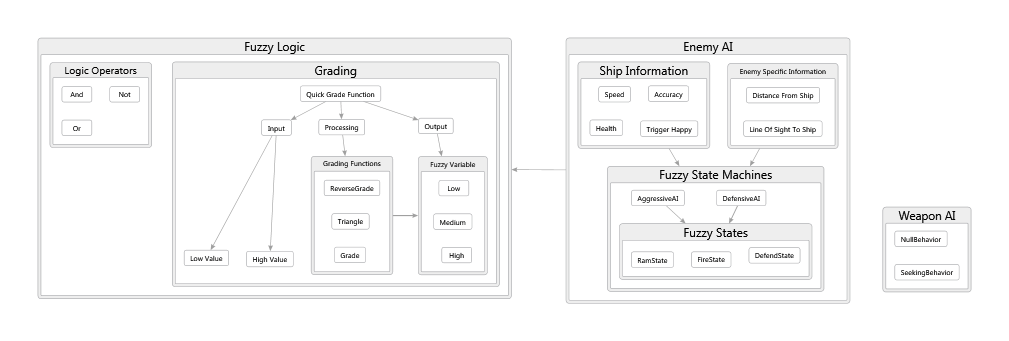


Figure 6 State / StateMachine Interfaces

Illustration 1: AI

The AI in the game consists of three components:

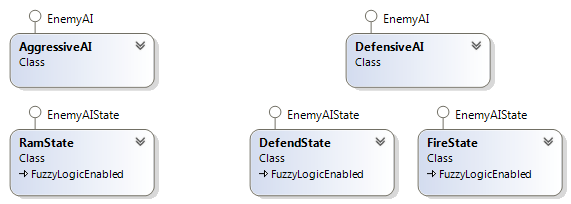
* Classes containing raw and computed information
* Behavior classes which consume information and operate on specific entity types
* Entities containing a read / write property allowing their behavior to be set

In this game, every fuzzy variable consists of three fuzzy sets(low, medium, high).

A quick grade function was created which takes minimum, maximum and actual values for crisp input and returns a FuzzyVariable class instance. A FuzzyVariable instance holds the degree to which the crisp input is low, medium or high.

**Enemy AI**

The enemy AI behaviors are implemented as fuzzy state machines. Each behavior has one or more fuzzy states which are all active to various degrees all the time.

Figure 7: Enemy AI behaviors

The fuzzy states use the fuzzy variables in the information classes along with fuzzy logic operators to come up with one fuzzy output variable, the degree to which that fuzzy state is active.

This membership is then used in the state as a multiplier to apply to a constant value resulting in crisp output.

The distance the ship must be to be in attack range is overlooked if the player is doing too well (high accuracy and high health). This is accomplished by using a logical or on the “too well” and “in range” variables.

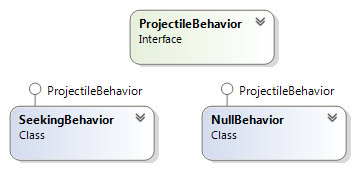
**Examples**

* The rate at which a defensive AI fires at the player is also dependent on the fuzzy distance between the player and the enemy.
* The aggressive AI's Ram state plots way-points of varying accuracy and moves towards them at a varying speed and with a varying rotation based on the states membership. The state membership is a logical or of the ships distance and how well the player is doing. If the player is doing “too well” (high health and a high accuracy) the AI will attack the ship from further away.

**Weapon AI**

Weapon AI uses the information classes to operate on a projectile.

* The default projectile behavior has the projectile move in the direction it was fired.
* The seeking AI uses line of sight, interception and a steering behavior to smoothly intercept its target.

Figure 8: Projectile behaviors

Unlike the enemy AI behaviors, the projectile behaviors are not implemented using state machines.

**Design Philosophy**

* Many small files are better than several large files
* Having many specific interfaces is better than having a couple of very general interfaces.
* Composition is better than inheritance.
* Use patterns where appropriate (this project uses the component, state and update method patterns)