Inteligencia Artificial para Videojuegos – Capítulo 1

Introducción

En esta unidad estudiaremos una de las técnicas de Inteligencia Artificial más utilizadas en los videojuegos. La misma se conoce como Máquinas de Estado Finito o Finite State Machine, FSM de ahora en adelante. Históricamente las FSM se han utilizado en diversos campos de la ciencia. En particular en videojuegos las mismas son generalmente el corazón de todos los personajes y elementos que vemos en pantalla ya que no sólo se usan para modelar comportamientos sino que tienen aplicación en otras áreas como por ejemplo en la animación de los personajes. En esta unidad vamos a estudiarlas enfocándonos en su aplicación para el modelado de personajes y comportamientos, dejando de lado otras aplicaciones que ya han visto en materias anteriores.

Máquinas de Estado Finito

¿Porqué?

Las máquinas de estado finito poseen muchas ventajas. Primero veamos estas ventajas para poder justificar el porqué son nuestra primera unidad del curso:

- Fáciles y rápidas de implementar: Hay muchas formas de programar una y todas son razonablemente sencillas. En este capítulo veremos algunas de ellas.
- Fáciles de depurar: Cómo el comportamiento se divide en partes medianamente independientes es fácil trazar los estados problemáticos e inspeccionarlos.
- Bajo costo computacional: Utilizan muy poco tiempo de procesador ya que esencialmente siguen una serie de reglas pre definidas. Simplemente siguen una serie de cláusulas if-then.
- Intuitivas: Está en la naturaleza humana pensar que las cosas están en un estado y otro. A veces cuando hablamos de nosotros mismos solemos clasificarnos en estados, como cuando decimos: “Estoy contento”, “Mi estado no es el mejor”, etc. De la misma forma podemos describir los estados que atraviesa nuestros personajes virtuales. Además nos dan una herramienta para comunicarnos con otros integrantes del equipo que no son programadores, tales como artista, diseñadores, etc. de una manera mucho más clara.
- Flexibles: Son fácilmente modificadas y ajustadas para proveer el comportamiento requerido por el diseñador. Además es sencillo agregar nuevos estados y comportamientos. Además sirven como base para aplicar otras técnicas más avanzadas como lógica difusa.

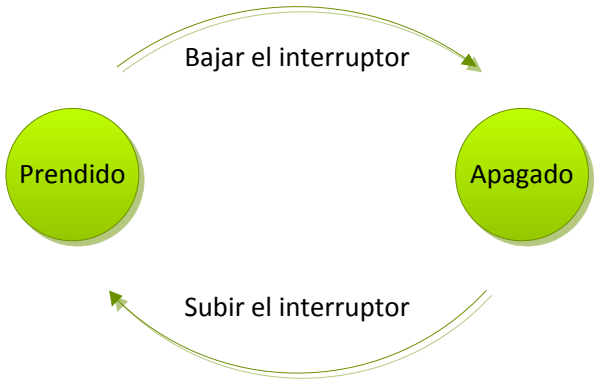
Máquinas de estado… ¿finito?

Empecemos describiendo qué es una máquina de estado finito. Si bien el nombre suena un poco extraño, verás que el concepto detrás es realmente sencillo, y que sin saber lo has venido usando en otras materias.

> ***Una máquina de estado finito (FSM) es un dispositivo o un modelo de un dispositivo que posee un número finito de estados en los cuales puede encontrarse en cualquier momento dado y puede operar en una entrada para realizar transiciones de un estado a otro o para causar una salida o acción.***

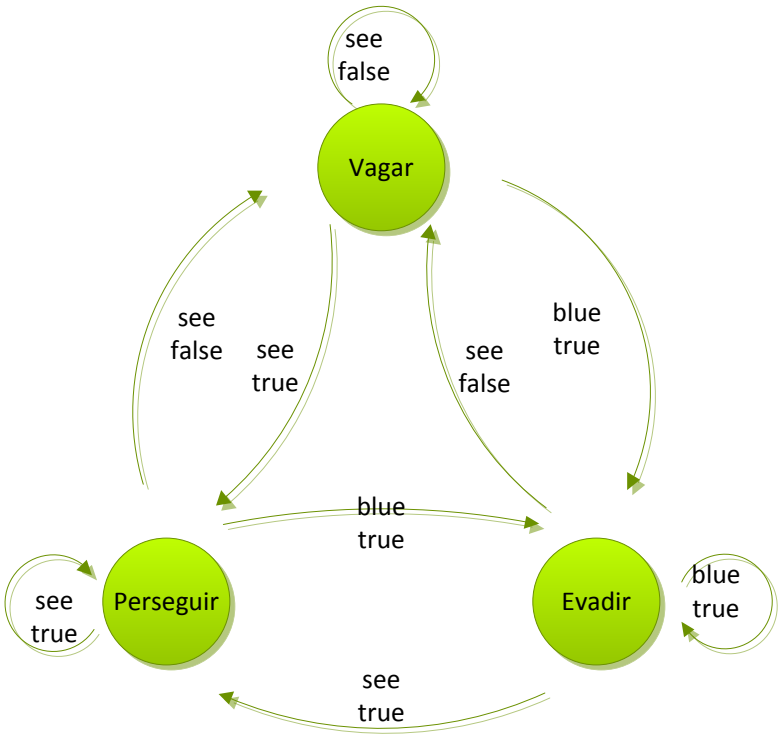
Cabe destacar que una FSM no puede estar en más de un estado a la vez. La idea detrás de una FSM es dividir el comportamiento de un objeto en pedacitos o estados los cuales son más fáciles de manejar por separado. Observa a tu alrededor, y seguramente vas a encontrar un interruptor de luz, ése es un ejemplo extremadamente sencillo de una FSM. El interruptor puede estar

prendido o **apagado.** Nunca puede estar en ambos estados. Las máquinas de estado se suelen representar gráficamente mediante diagramas formados por nodos que indican los estados y arcos que unen los nodos indicando el sentido de recorrido. Por ejemplo para el caso del interruptor de luz:



El interruptor puede estar en dos estados: prendido o apagado. Y para hacerlo pasar de un estado a otro es necesario aplicar una entrada que en este caso sería bajar o subir el interruptor. Este es un ejemplo muy sencillo de una máquina de estados con 2 estados dónde se muestran las funciones o condiciones que llevan a cabo un cambio de estado. Como se puede observar las flechas tienen una dirección que indican en qué sentido pueden ser recorridas. En algunos casos pueden tener dos sentidos.

En los videojuegos las máquinas de estado suelen ser bastante más complejas que lo que te hemos mostrado aquí. Por ejemplo, recordemos un clásico como el Pac-Man. En el mismo los fantasmas se pueden modelar como una FSM de la siguiente forma:



En la misma se pueden observar 3 estados **Vagar**, **Perseguir** y **Evadir.** En **Vagar** el fantasma se mueve libremente sin perseguir un objeto concreto más allá de recorrer el laberinto. El estado **Perseguir** se alcanza cuando el fantasma persigue a Pac-Man y por último **Evadir** es cuando el fantasmita es azul y Pac-Man lo persigue para comerlo. En este caso en los arcos del diagrama se han colocado las condiciones que deben cumplirse para un cambio de estados o para permanecer en el mismo. Se tienen dos posibles banderas *blue* y *see*. La primera es verdadera cuando el fantasma es azul (debido a que Pac-Man comio una pastilla grande) y falsa en todo otro momento. Por otro lado la bandera *see* es verdadera cuando el fantasmita puede ver a Pac-Man. Si miran con detenimiento van a ver que esta máquina de estados describe todo el comportamiento de un fantasma. La máquina siempre arranca en el estado **Vagar** y va cambiando de acuerdo a las entradas o condiciones. Por ejemplo, el fantasma se encuentra en **Vagar**, y mientras see sea falsa, es decir mientras no vea a Pac-Man no va a cambiar de estado. Si luego de dar un par de vueltas el fantasma lo ve a Pac-Man entonces tiene dos posibles acciones: Si el fantasma se encuentra azul, entonces pasa al estado **Evadir** (*blue=true)*, en caso contrario toma el otro arco y pasa al estado **Perseguir.** De esta forma hemos especificado sin ambigüedades cómo se debe comportar el fantasma de nuestro Pac-Man.

De la misma forma en que hemos visto para Pac-Man se utilizan máquinas de estado en la mayoría de los videojuegos, por nombrar algunos:

- Los bots al estilo Quake son implementados mediante FSM. Los mismos tienen estados tales como **BuscarArmadura**, **BuscarEnergia**, **Cubrirse**, **Huir**, etc. Incluso los estados de las armas se suelen modelar de la misma manera

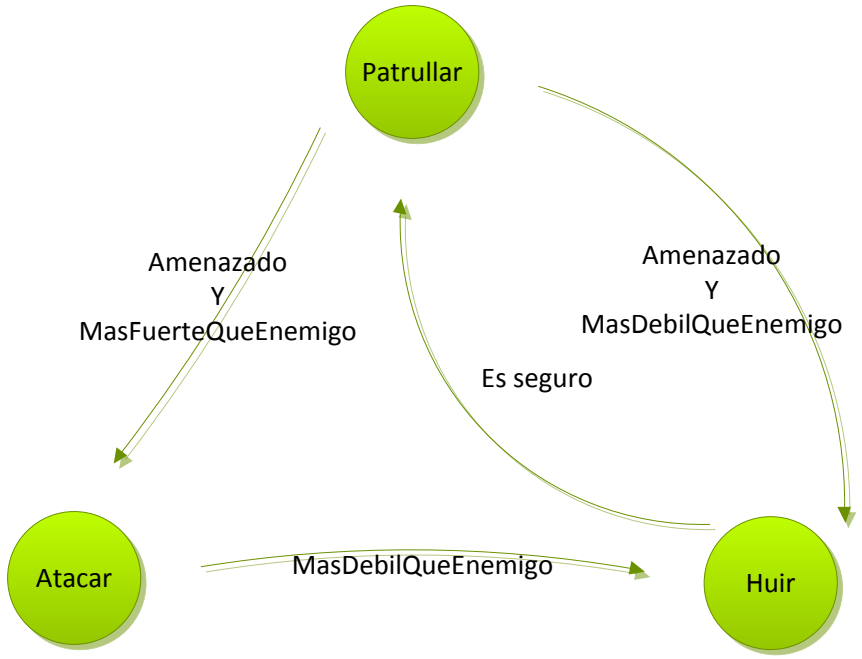
- En los juegos de deportes como en FIFA por ejemplo, cada jugador posee una FSM con estados tales como **Patear**, **Gambetear**, **PerseguirPelota**, **MarcarJugador**, etc.
- Las unidades en los videojuegos de estrategia en tiempo real (RTS) suelen utilizar FSM con estados tales como **MoverseHacia**, **Patrullar**, **SeguirRuta**, etc.

Otra alternativa de representar las FSM es mediante las conocidas **Tablas de Transición de Estados**. Por cada entrada de la misma se enuncia el estado actual, la condición de transición y el estado al cual cambia. Por ejemplo en el caso de representar la FSM del fantasma del Pac-Man vista anteriormente tendriámos:

Estado	Condición	Cambia a
Vagar	see=false	Vagar
Vagar	blue=true	Evadir
Vagar	see=true	Perseguir
Evadir	blue=true	Evadir
Evadir	see=true	Perseguir
Evadir	see=false	Vagar
Perseguir	see=true	Perseguir
Perseguir	blue=true	Evadir
Perseguir	see=false	Vagar

Más allá de los lindos diagramas

Los diagramas son muy bonitos y simpáticos pero no hacen que se mueva un personaje (contrario a lo que muchos creen). Como nosotros somos valientes y nobles vamos a ensuciarnos las manos… bien sucias y vamos a empezar a ver cómo se suelen implementar las FSM. Para ello veamos un ejemplo ahora de un personaje de un RTS el cual es muy similar al del Pac-Man visto anteriormente:



En el mismo tenemos 3 estados **Patrullar**, **Huir** y **Atacar**. Si la unidad se encuentra patrullando y encuentra un enemigo más débil lo ataca, en cambio sí es más fuerte pasa al estado **Huir**. La unidad huye hasta que es seguro luego vuelve a **Patrullar**. Es un comportamiento muy sencillo e incompleto, pero nos servirá para ver cómo lo implementamos. La primera idea que siempre surge es organizar todo con una estructura switch por ejemplo de la siguiente manera:

```
class RTSUnit
{

	enum StateType{RunAway, Patrol, Attack};

	//Condiciones
	bool Safe(){} //esta seguro
	bool Threatened(){} //esta amenazado
	bool StrongerThanEnemy(){} //es mas fuerte que el enemigo
	bool WeakerThanEnemy(){} //es mas debil que el enemigo

	void EvadeEnemy(){} //evadir enemigo
	void BashEnemyOverHead(){} //golpearlo en la cabeza
	void ChangeState(StateType newState){} //cambia el estado actual
	void FollowPatrolPath(){}  //Sigue camino de patrulla

	void RTSUnit::UpdateState(StateType CurrentState)
	{
		switch(CurrentState) //reaccionamos dependiendo del estado
		{
		case StateType::RunAway:  //si esta huyendo
			EvadeEnemy(); //evade al enemigo
			if (Safe()) //si esta seguro
			{
				ChangeState(Patrol); //cambia a patrullar
			}
			break;

		case StateType::Patrol: //si esta patrullando
			FollowPatrolPath(); //que siga el camino
			if (Threatened()) //si algo lo amenaza
			{
				if (StrongerThanEnemy()) //si es mas fuerte
				{
					ChangeState(Attack); //ataca
				}
				else //es mas debil
				{
					ChangeState(RunAway); //cambia a huir
				}
			}
			break;
		case StateType::Attack: //si esta atacando
			if (WeakerThanEnemy()) //y es mas debil que enemigo
			{
				ChangeState(RunAway); //cambia a huir
			}
			else
			{
				BashEnemyOverHead(); //sino golpea en la cabeza
			}
			break;
		}//end switch
	}
};
```

En este caso se construye la FSM con un gran switch sobre el estado actual. Dentro de cada case se pueden observar dos cosas. En primer lugar una condición para el cambio de estado, la cual si es cierta hace la transición. Por otro lado, sino se da la condición para el cambio de estado, entonces debe tomar una acción correspondiente al estado actual. Por ejemplo cuando se encuentra en el estado Attack se fija que no sea más débil que el enemigo, si en algún momento esto se cumple entonces deja de atacar y cambia a huir. En caso contrario llama al método de golpear (*BashEnemyOverHead*).

Si bien implementar una FSM de esta forma puede parecer razonable al principio, sólo es práctica para los objetos más sencillos y con pocos estados. A medida que los comportamientos se vuelven más complejos implementar la FSM usando if-then se torna muy engorroso y difícil de mantener. Sin siquiera mencionar la pesadilla que se puede tornar depurarla. También muchas veces se desea reaccionar al entrar o salir de un estado. Además es muy común durante el desarrollo que los comportamientos cambien y se ajuste, apareciendo nuevos estados y desapareciendo otros. Por estos motivos, debemos buscar una mejor alternativa para implementar una FSM.

Cosas de patos

Vamos a tomar unas ideas de Mat Buckland para la siguiente analogía. Imaginate que tenés un pato robot. Todo amarillo y brillante. El pato por sí mismo no presenta ningún tipo de comportamiento, pero tiene en su panza un slot para tarjetas de memoria. Cada tarjeta de memoria o *estado* posee las instrucciones lógicas para el comportamiento del pato. Por ejemplo hay una tarjeta llamada *nadar* que contiene todas las instrucciones para que el pato nade cuando se le coloca la tarjeta. Así mismo podemos tener tarjetas *comer*, *graznar, volar*, etc. Cada tarjeta representa un estado del pato robot. Supongamos también que el pato es muy habilidoso y puede cambiar él mismo las tarjetas de memoria. De esta forma podría pasar de caminar a volar sin necesidad de ayuda externa.

Ahora para crear comportamientos más elaborados sólo necesitamos indicarle al pato de alguna forma en qué momentos cambiar de tarjetas de memoria. Para realizar esto necesitamos incorporarle algo de lógica en algún lado. Para ello imaginemos que programamos un chip y lo ponemos en la cabeza del pato. Este chip puede leer el estado del pato (por ejemplo si tiene hambre, calor, etc) y puede enviar comandos para que el pato cambie de tarjetas. De esta forma, estamos generando comportamientos más elaborados dependiendo del estado del pato y la lógica que programamos. Si quisiéramos agregar un nuevo estado sólo hace falta agregar una nueva tarjeta de memoria, o si quisiéramos modificar un estado sólo hace falta tocar dicho estado y los demás se mantienen igual. Esto logra una gran independencia entre los estados. Por otro lado también será necesario reprogramar el chip que el pato tiene en su cabeza ante nuevos estados. Sin embargo esta tarea se puede llevar a cabo sin modificar ninguno de los otros circuitos del robot.

Pero… Y si le damos una vuelta más de tuerca?

Sigamos con nuestro elegante pato robot. Ahora imaginemos que en vez de tener la lógica del cambio de tarjetas en un chip en la cabeza, la tienen en las mismas tarjetas de memoria o mejor dicho, estados. De esta forma, en la tarjeta de cada estado, poseemos la lógica de cuándo cambiar a otro estado o tarjeta. Si bien cada estado necesita saber sobre los demás para poder pedir el cambio, esta arquitectura es mucho más independiente que la anterior, ya que no tenemos centralizada la lógica del cambio de estados, haciendo mucho más sencillo cambiar algunos o todos los estados. Por ejemplo podríamos cambiar todas las tarjetas del pato por tarjetas de un perro y el pato seguiría funcionando ahora como un perro sin necesidad de reprogramarle nada adicional.

Esta estructura que hemos expuesto en el párrafo anterior es la que comúnmente se utiliza. Veamos ahora cómo implementamos esta alternativa. Para ello utilizaremos esas características de C++ que tanto adoran: Herencia y Polimorfismo!.

Retomemos el ejemplo de la unidad de un juego de RTS. Primero vamos a definir una clase abstracta que representará un estado de nuestra FSM:

```
class State
{
public:
	//metodo virtual que se ejecutara en cada vuelta
	virtual void Execute (RTSUnit* unit) = 0;

};
```

Esta clase la usaremos de madre para crear los estados deseados. Sólo posee un método virtual que recibe como argumento un objeto del tipo RTSUnit (nuestra unidad) para poder determinar su situación actual y tomar acciones sobre la misma. Veamos ahora cómo declaramos nuestra clase RTSUnit:

```
class RTSUnit
{
	/* Aca van los atributos de clase */
	State* m_pCurrentState;
public:

	void Update()
	{
		m_pCurrentState->Execute(this);
	}
	void ChangeState(State* pNewState)
	{
		delete m_pCurrentState;
		m_pCurrentState = pNewState;
	}
};
```

En la misma se han omitido los atributos de la clase para poder ver claramente lo que nos interesa. Se puede observar que la unidad mantiene un puntero a un objeto de tipo *State* llamado m_pCurrentState. Este puntero representa el estado en el cual se encuentra nuestra unidad. Además nuestra clase posee dos métodos. El método ChangeState borra el estado anterior y configura como actual el que recibe como argumento. Este es el método que nos permitirá luego modificar el comportamiento. Por otro lado tenemos un método Update, en el mismo simplemente se ejecuta el método Execute del estado actual, el cual se encarga de la lógica de nuestra unidad. Este método recibe como argumento la unidad actual para poder obtener su situación y actuar en consecuencia.

Ya tenemos nuestra unidad que sabe cambiar estados y una clase madre que representa un estado. Ahora simplemente debemos crear los estados. Si revisamos nuestra implementación anterior con los if-then vemos 3 estados: *RunAway, Patrol y Attack*. Empecemos implementando el estado *Runaway:*

```
class State_RunAway: public State
{
public:
	void Execute(RTSUnit* unit)
	{
		if (unit->Safe()) //si esta seguro
		{
			unit->ChangeState(new State_Patrol()); //patrullar
		}else{
			unit->EvadeEnemy(); //evade al enemigo
		}
	}
};
```

Dicho estado implementa el mismo comportamiento visto anteriormente pero ahora usando esta nueva metodología. En principio hereda de State e implementa el método virtual Execute. Si la

unidad esta segura (fuera de peligro) entonces cambia su estado a patrullar. Para hacer esto llama al método *ChangeState* y le pasa como argumento un nuevo objeto del tipo del estado deseado. Si no se encuentra segura llama al método *EvadeEnemy* de unit para que siga un plan de escape. Ahora hacemos lo mismo para los otros estados:

```
class State_Patrol: public State
{
public:
	void Execute(RTSUnit* unit)
	{

		if (unit->Threatened()) //si algo lo amenaza
		{
			if (unit->StrongerThanEnemy()) //si es mas fuerte
			{
				unit->ChangeState(new State_Attack()); //ataca
			}
			else //es mas debil
			{
				unit->ChangeState(new State_RunAway()); //huye
			}
		}else{
			unit->FollowPatrolPath(); //que siga el camino
		}
	}
};

class State_Attack: public State
{
public:
	void Execute(RTSUnit* unit)
	{

		if (unit->WeakerThanEnemy()) //y es mas debil que enemigo
		{
			unit->ChangeState(new State_RunAway()); //cambia a huir
		}else{
			unit->BashEnemyOverHead(); //sino golpea en la cabeza
		}
	}
};
```

De esta forma hemos partido el contenido de nuestra estructura switch que presentamos anteriormente en una seria de clases que implementan el mismo comportamiento pero ahora de manera mucho más elegante. Este patrón de diseño se conoce como **patrón de diseño de estados** (***state design pattern***) y es muy utilizado. Como se puede observar el comportamiento se encuentra totalmente contenido en cada objeto State. Esto hace que agregar un estado o eliminar sea relativamente sencillo y nos permite mantener separado el código.

¿Vamos o venimos?

Muchas veces cuando estamos programando nuestros personajes deseamos reaccionar o tener control cuando se produce un cambio de estado. Utilizando el patrón que hemos presentado anteriormente esto se torna muy sencillo, ya que basta con agregar un par de métodos virtuales extra a la clase madre *State*:

```
class State
{
public:
	//método virtual que se ejecutara en cada vuelta
	virtual void Execute (RTSUnit* unit) = 0;
	//al entrar al estado
	virtual void Enter(RTSUnit* unit)=0;
	//al salir del estado
	virtual void Exit(RTSUnit* unit)=0;
};
```

Y por supuesto debemos cambiar el método ChangeState de RTSUnit para que los invoque:

```
class RTSUnit
{
	/* Aca van los atributos de clase */
	State* m_pCurrentState;
public:

	void Update()
	{
		m_pCurrentState->Execute(this);
	}
	void ChangeState(State* pNewState)
	{
		//Le avisamos al estado viejo
		//que lo abandonamos
		m_pCurrentState->Exit(this);
		delete m_pCurrentState;
		//seteamos el nuevo
		m_pCurrentState = pNewState;
		//le avisamos al nuevo estado
		//que estamos entrando
		m_pCurrentState->Enter(this);

	}
};
```

De esta forma podemos programar en Enter o Exit lo que necesitemos para preparar el estado cuando entramos o cuando lo dejamos respectivamente.

Aquella cosa llamada singleton

Todo muy bonito hasta ahora… pero tenemos dos problemas bastante importantes que todavía no hemos abordado. Por un lado nuestra máquina de estados sólo funciona para objetos del tipo RTSUnit, esta es una limitación que luego veremos como superarla. Antes debemos atender a algo un poco más urgente. Si se aprecia con detenimiento el código que hemos venido desarrollando se puede observar que cada vez que hacemos un cambio de estados estamos alocando y liberando memoria. Si bien no vamos a tener *leaks* la operación de alocar memoria es costosa. Además todas las RTSUnit van a compartir los mismos estados normalmente, por lo que tener copias de los estados por cada unidad sería un desperdicio de memoria. Para solucionar este problema vamos a recurrir a un patrón de diseño conocido como Singletons.

Este patrón nos permite crear objetos con las siguientes características:

- Son globalmente accesibles: Los podemos acceder desde cualquier archivo fuente de nuestro programa sin comprometer los preceptos de la orientación a objetos, ya que por definición no son variables globales.
- Sólo puede existir una instancia de los mismos: No se puede crear más de una copia de un objeto que implemente el patrón singleton. Esto es útil si queremos asegurarnos que todos trabajen con el mismo objeto y no lo puedan copiar.

Hay muchas formas de implementar el patrón Singleton, nosotros vamos a pegarnos a una bastante sencilla y eficaz. Veamos cómo declaramos una clase que implemente el patrón:

```
#pragma once
/* Ejemplo de singleton */
class ClaseSingleton
{
private:
	//constructor es privado (IMPORTANTE)
	ClaseSingleton(){}
	//asignacion y constructor por copia privados (IMPORTANTE)
	ClaseSingleton(const ClaseSingleton &);
	ClaseSingleton& operator=(const ClaseSingleton &);

public:
	//Para ser estrictos el destructor deberia ser privado
	//pero algunos compiladores tienen problemas con esto
	//por ello lo declaramos publico
	~ClaseSingleton();

	//Este método devuelve un puntero
	//a la unica instancia de la clase
	static ClaseSingleton* Instance();

};
```

Aquí declaramos una clase *ClaseSingleton* para ejemplificar el patrón. En la misma vemos que sus constructores son privados de manera que nadie pueda crear objetos de la clase utilizando el operador *new* o instanciandolos como variables en el *stack*. También vemos un método que se llama Instance, el mismo nos va a devolver un puntero a la única instancia que existe de la clase. Ahora veamos cómo implementamos *Instance* para conseguir que sólo exista una sola instancia. Para ello siempre en el .cpp:

```
#include "ClaseSingleton.h"


ClaseSingleton* ClaseSingleton::Instance()
{
	static ClaseSingleton instance;
	return &instance;
}
```

Aquí podemos ver que declaramos una instancia de ClaseSingleton como estática, lo cual significa que existe a lo largo de toda la vida de la aplicación (no importa que el scope sea local en este caso) y devolvemos dicha variable. No debe pensarse que en este caso se crea una instancia nueva cada vez que se invoca *Instance* sino que se devuelve la instancia estática que es siempre la misma. De esta forma podemos utilizar nuestra clase Singleton desde cualquier parte de nuestra aplicación haciendo:

```
ClaseSingleton::Instance()->[MiembroDeseado]();
```

De esta forma invocamos *Instance* que es un miembro estático y éste nos devuelve un puntero a la instancia estática de ClaseSingleton. Luego podemos invocar sobre la misma el miembro que deseemos. Debe notarse que aquí no creamos nunca una instancia de ClaseSingleton sino que usamos un miembro estático. Por esto mismo tampoco nunca debemos hacer el delete de la misma.

Entonces ahora, utilizando estas ideas que hemos expuesto nuestros estados se verán de la siguiente forma, si declaramos los estados en una archivo states.h :

```
class State_RunAway: public State
{

private:
	//constructores vacios y por copia privados
	//para que nadie pueda crear la instancia
	State_RunAway(){}
	State_RunAway(const State_RunAway* &other){}
	State_RunAway& operator=(const State_RunAway &){}


public:

	static State_RunAway* Instance();
	void Execute(RTSUnit* unit);
	void Enter(RTSUnit* unit){}
	void Exit(RTSUnit* unit){}
};

class State_Patrol: public State
{
private:
	//constructores vacios y por copia privados
	//para que nadie pueda crear la instancia
	State_Patrol(){}
	State_Patrol(const State_Patrol* &other){}
	State_Patrol& operator=(const State_Patrol &){}

public:
	static State_Patrol* Instance();
	void Execute(RTSUnit* unit);
	void Enter(RTSUnit* unit){}
	void Exit(RTSUnit* unit){}
};

class State_Attack: public State
{
private:
	//constructores vacios y por copia privados
	//para que nadie pueda crear la instancia
	State_Attack(){}
	State_Attack(const State_Attack* &other){}
	State_Attack& operator=(const State_Attack &){}

public:
	static State_Attack* Instance();

	void Execute(RTSUnit* unit);
	void Enter(RTSUnit* unit){}
	void Exit(RTSUnit* unit){}
};
```

Podemos ver cómo hemos incorporado el patrón singleton a cada estado. Ahora veamos en el archivo de definiciones state.cpp cómo cambian los métodos *Execute* de cada estado:

```
State_RunAway*  State_RunAway::Instance()
{
	static State_RunAway instance;
	return &instance;
}

void State_RunAway::Execute(RTSUnit* unit)
{
	if (unit->Safe()) //si esta seguro
	{
		unit->ChangeState(State_Patrol::Instance()); //cambia a patrullar
	}else{
		unit->EvadeEnemy(); //evade al enemigo
	}
}

State_Patrol*  State_Patrol::Instance()
{
	static State_Patrol instance;
	return &instance;
}

void State_Patrol::Execute(RTSUnit* unit)
{
	if (unit->Threatened()) //si algo lo amenaza
	{
		if (unit->StrongerThanEnemy()) //si es mas fuerte que enemigo
		{
			unit->ChangeState(State_Attack::Instance()); //ataca
		}
		else{ //es mas debil
			unit->ChangeState(State_RunAway::Instance()); //huye
		}
	}else{
		unit->FollowPatrolPath(); //que siga el camino
	}
}


State_Attack*  State_Attack::Instance()
{
	static State_Attack instance;
	return &instance;
}

void State_Attack::Execute(RTSUnit* unit)
{
	if (unit->WeakerThanEnemy()) //y es mas debil que enemigo
	{
		unit->ChangeState(State_RunAway::Instance()); //cambia a huir
	}
	else{
		unit->BashEnemyOverHead(); //sino golpea en la cabeza
	}
}
```

Como se puede observar el código es casi idéntico al que presentamos anteriormente, la única diferencia la hemos resaltado en amarillo. En vez de crear un estado nuevo usando el operador new, simplemente recuperamos la única instancia del estado gracias al patrón singleton. De esta forma no estamos creando ni borrando memoria constantemente, y nos evitamos el impacto de performance que esto tiene especialmente en consolas.

También cambia un poco el método ChangeState de RTSUnit ya que ahora no debemos eliminar el estado que borramos. En definitiva terminamos obteniendo:

```
void ChangeState(State* pNewState)
{
	//Le avisamos al estado viejo
	//que lo abandonamos
	m_pCurrentState->Exit(this);
	//seteamos el nuevo
	m_pCurrentState = pNewState;
	//le avisamos al nuevo estado
	//que estamos entrando
	m_pCurrentState->Enter(this);

}
```

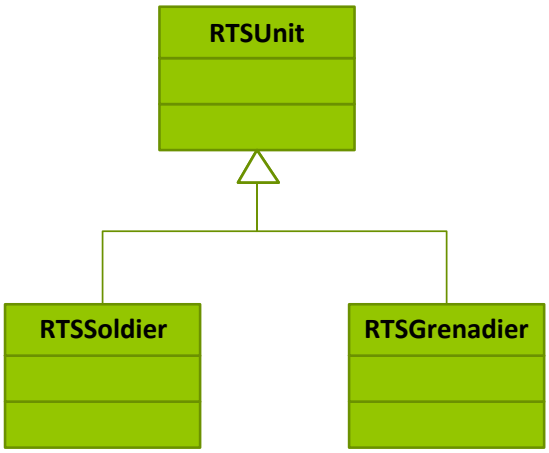
De esta forma tenemos estados que existen a lo largo de nuestra aplicación y no nos representan un overhead importante a la hora de usarlos. Sólo nos falta resolver el segundo problema que habíamos mencionado: la falta de generalidad.

Combatiendo la falta de generalidad

Nuestras máquinas de estado hasta ahora fueron diseñadas para funcionar con un tipo de datos (RTSUnit en el ejemplo que vimos). Imaginemos ahora que necesitamos una máquina de estados para una nueva unidad llamada RTSSoldier. Si tomamos el camino poco feliz vamos a tener que volver a implementar lo que acabamos de hacer para este nuevo tipo, ya que la máquina de estados depende del mismo para poder actuar (No sigas sino estas convencido de esto!) . Esto es muy poco práctico y tira por el piso las ventajas que perseguíamos al utilizar máquinas de estado. Es por ello que debemos buscar otras alternativas. En particular hay dos formas muy utilizadas para sobrellevar este problema. La primera es utilizando polimorfismo y la segunda utilizando templates. Veremos los dos casos a continuación.

La primer cara de la moneda… O breve historia sobre la aplicación del polimorfismo

La primera forma que vamos a ver de resolver el problema de la generalidad será mediante las relaciones entre clases y la capacidad de polimorfismo que nos brinda el lenguaje C++. La idea es que todos los objetos de nuestro videojuego que necesitan implementar una máquina de estados hereden de una clase madre especificada y que nuestra máquina de estados reciba como argumentos esa clase madre. En nuestro caso vamos a suponer que queremos implementar dos unidades de un juego de RTS, un soldado y un granadero. Entonces podemos plantear una clase madre llamada RTSUnit de la cual ambas heredan:



Con esta jerarquía luego podemos plantear nuestra máquina de estados para que reciba como argumento un RTSUnit, y de ser necesario adentro realizar el cast de tipos para obtener punteros al RTSSolider o al RTSGrenadier. Por ejemplo supongamos que el soldado y el granadero comparten el mismo comportamiento excepto cuando están patrullando. El soldado no huye nunca y el granadero si lo hará si el enemigo es más fuerte. El estado madre se mantiene igual:

```
class State
{
public:
	//metodo virtual que se ejecutara en cada vuelta
	virtual void Execute (RTSUnit* unit) = 0;
	//al entrar al estado
	virtual void Enter(RTSUnit* unit)=0;
	//al salir del estado
	virtual void Exit(RTSUnit* unit)=0;
};
```

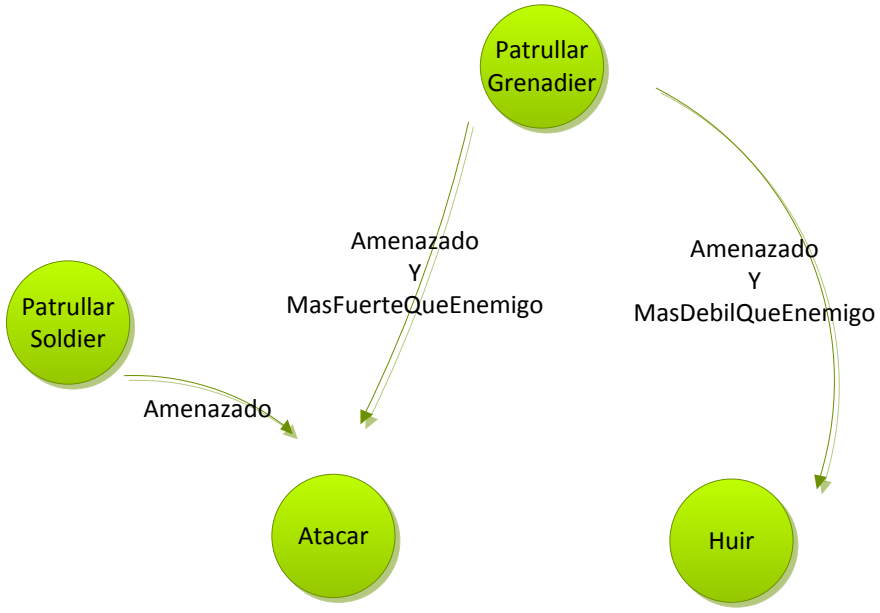
Mientras que ahora tendremos dos estados para patrullar:

```
class State_Patrol_Soldier: public State
{
private:
	//constructores vacios y por copia privados
	//para que nadie pueda crear la instancia
	State_Patrol_Soldier(){}
	State_Patrol_Soldier(const State_Patrol_Soldier* &other){}
	State_Patrol_Soldier& operator=(const State_Patrol_Soldier &){}

public:
	static State_Patrol_Soldier* Instance();
	void Execute(RTSUnit* unit);
	void Enter(RTSUnit* unit){}
	void Exit(RTSUnit* unit){}
};

class State_Patrol_Grenadier: public State
{
private:
	//constructores vacios y por copia privados
	//para que nadie pueda crear la instancia
	State_Patrol_Grenadier(){}
	State_Patrol_Grenadier(const State_Patrol_Grenadier* &other){}
	State_Patrol_Grenadier& operator=(const State_Patrol_Grenadier &){}

public:
	static State_Patrol_Grenadier* Instance();
	void Execute(RTSUnit* unit);
	void Enter(RTSUnit* unit){}
	void Exit(RTSUnit* unit){}
};
```

Y las implementaciones de sus métodos:

```
void State_Patrol_Grenadier::Execute(RTSUnit* unit)
{
	if (unit->Threatened()) //si algo lo amenaza
	{
		if (unit->StrongerThanEnemy()) //si es mas fuerte que enemigo
		{
			unit->ChangeState(State_Attack::Instance()); //ataca
		}
		else{ //es mas débil
			//cambia a huir
			unit->ChangeState(State_RunAway::Instance());
		}
	}else{
		unit->FollowPatrolPath(); //que siga el camino
	}
}


void State_Patrol_Soldier::Execute(RTSUnit* unit)
{
	if (unit->Threatened()) //si algo lo amenaza
	{
		//siempre ataca
		unit->ChangeState(State_Attack::Instance());
	}else{
		unit->FollowPatrolPath(); //que siga el camino
	}
}
```

Si hacemos un diagrama de nuestra nueva máquina de estados entonces veremos lo siguiente:



En este caso estamos suponiendo que el estado Atacar y Huir de ambos son iguales por lo que lo comparten. Esto nos servirá para ilustrar un problema y su posible solución. ¿Qué sucede cuando queremos volver a los estados Patrullar Soldier y Patrullar Grenadier? En el caso de huir no hay

peligro de equivocación ya que el único estado que lo lleva a Huir es Patrullar Grenadier. Pero en el caso de atacar esto no es tan obvio, ya que podemos estar en el estado Atacar porque llegamos siendo Soldado o Granadero como lo muestran las dos flechas que convergen al mismo. La forma más sencilla de solucionar esto es creando dos estados Atacar, en vez de unos solo. Esto es válido, pero si ambos estados atacar son idénticos es un poco engorroso, entonces lo que se puede hacer es mantener un solo estado Atacar y allí dentro determinar el tipo de unidad y en base a eso hacer el cambio de estado. Por ejemplo, supongamos que para pasar de Atacar a patrullar debe darse la condición de que hemos derrotado al enemigo:

```
void State_Attack::Execute(RTSUnit* unit)
{
	if(unit->EnemyDefeated())
	{
		if(unit->tipo==0)
			unit->ChangeState(State_Patrol_Soldier::Instance());
		else
			unit->ChangeState(State_Patrol_Grenadier::Instance());
	}
}
```

Si se da la condición entonces preguntamos por el tipo de unit, ya que lo que tenemos en realidad es un puntero a la clase madre y no a las hijas. Por eso siempre necesitamos tener en la clase madre algún campo que nos permita diferenciar las hijas. En este caso usamos un entero denominado tipo, cada clase hija se encarga de inicializar dicho entero a los valores correctos. De esta forma podemos hacer el cambio de estado condicionados al tipo de la unidad.

La otra cara de la moneda…

Generalmente los templates suelen ser sinónimos de “cosas que hacen a la sintaxis compleja y difícil de leer”, pero en realidad vamos a ver que son sinónimo de “cosas que nos simplifican la vida y nos hacen programadores felices”.

El tema de templates ya ha sido abordado en materias anteriores por lo que nos vamos a limitar a hacer un breve repaso para hacerles un poco más sencilla la vida. Los templates incorporan en C++ el concepto de programación genérica.

La programación genérica nos permite escribir funciones y clases que funcionen con varios tipos de datos de manera automática. Para lograr esto lo que se hace es escribir unas plantillas de donde se deja un comodín en el lugar donde debería ir el tipo de datos. Por ejemplo si tenemos una función que suma 2 números flotantes, normalmente la escribiríamos:

```
float sumar(float dato1, float dato2)
{
	return dato1+dato2;
}
```

Pero si luego necesitamos una función que sume 2 enteros, entonces deberíamos escribir otra función:

```
int sumar(int dato1, int dato2)
{
	return dato1+dato2;
}
```

Y así para cada tipo de datos numérico que deseemos sumar. Como se puede observar la función es idéntica, sólo cambia el tipo de datos que manipula. Para evitar todo este trabajo repetido, podemos declarar una plantilla de función sin decirle cuál será el tipo de datos a sumar. Para hacer eso hacemos:

```
template <class T>
T sumar(T dato1, T dato2){

	return dato1+dato2;
}
```

Como podemos observar en la primera línea usamos la cláusula *template* que indica que a continuación le pasaremos una lista de los "comodines" o parámetros templates que utilizaremos. Luego escribimos la función normalmente pero en vez de poner un tipo de datos fijo en las declaraciones ponemos el parámetro template T. De esta forma tenemos nuestra plantilla de función. Ahora sólo resta indicarle cuando la llamemos qué es T:

```
float resf=sumar<float>(2.0f, 3.0f);

int resi=sumar<int>(2, 3);
```

Como se muestra luego del nombre de la función y entre "<>" le indicamos con qué debe reemplazar T. El compilador al ver esto busca la plantilla reemplaza T por el tipo indicado y compila la función. Hace esta tarea por cada tipo de datos distintos que ve. En este ejemplo el compilador termina generando dos funciones idénticas a las que planteamos al principio, pero nosotros sólo escribimos una plantilla. Esta es una de las grandes ventajas de la programación genérica. En muchos casos el compilador puede inferir qué es T a partir de los parámetros por lo que no siempre es necesario indicarle el tipo de datos.

De la misma forma que podemos utilizar parámetros genéricos en funciones lo podemos hacer en clases. Por ejemplo si tenemos una clase que representa un rectángulo alineado con los

ejes(AABB) para detectar colisiones que necesitamos que funcione con coordenadas enteras y flotantes podemos escribirla:

```
template <class T>
class AARect{

	T centroX;
	T centroY;
	T ancho;
	T largo;
	//demas miembros, etc...
}
```

Luego podemos crear AARect para coordenadas flotantes o enteras:

```
AARect<float> RectF;
AARect<int> RectI;
```

De esta forma programamos una plantilla que puede ser reutilizada para distintos tipos de datos. Esta idea es la que usaremos en nuestras máquinas de estado.

A continuación vamos a definir nuestro estado madre de la siguiente forma:

```
template <class entity_type>
class State
{
public:
	virtual void Enter(entity_type*)=0;
	virtual void Execute(entity_type*)=0;
	virtual void Exit(entity_type*)=0;
	virtual ~State(){}
};
```

La definición es idéntica a la que habíamos visto antes, sólo que ahora las funciones reciben un tipo genérico. De esta forma podemos usar esta clase como base para máquinas de estado de las distintas unidades que tengamos. Por ejemplo si tenemos las dos unidades de la sección anterior, un soldado y un granadero, entonces podemos definir sus estados patrullar como:

```
class State_Patrol_Soldier: public State<RTSSoldier>
{
private:
	//constructores vacios y por copia privados
	//para que nadie pueda crear la instancia
	State_Patrol_Soldier(){}
	State_Patrol_Soldier(const State_Patrol_Soldier* &other){}
	State_Patrol_Soldier& operator=(const State_Patrol_Soldier &){}

public:
	static State_Patrol_Soldier* Instance();
	void Execute(RTSSoldier * unit);
	void Enter(RTSSoldier * unit){}
	void Exit(RTSSoldier * unit){}
};

class State_Patrol_Grenadier: public State<RTSGrenadier>
{
private:
	//constructores vacios y por copia privados
	//para que nadie pueda crear la instancia
	State_Patrol_Grenadier(){}
	State_Patrol_Grenadier(const State_Patrol_Grenadier* &other){}
	State_Patrol_Grenadier& operator=(const State_Patrol_Grenadier &){}

public:
	static State_Patrol_Grenadier* Instance();
	void Execute(RTSGrenadier * unit);
	void Enter(RTSGrenadier * unit){}
	void Exit(RTSGrenadier * unit){}
};
```

Sistemas de mensajes

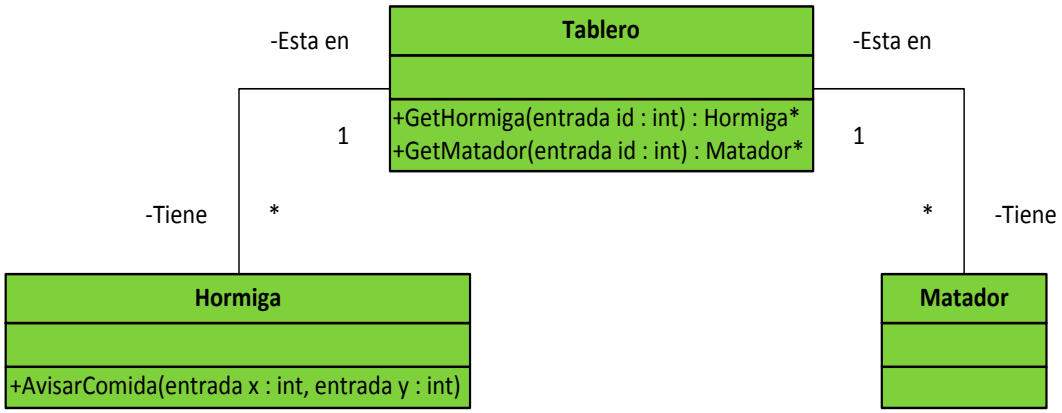
Los elementos de un videojuego (personajes, enemigos, etc) suelen necesitar enviarse información y notificarse de ciertos eventos. Para llevar a cabo esta tarea existen principalmente dos formas.

- La primera consiste en que los objetos adquieran un puntero a los objetos que necesitan y luego llamar sus métodos (lo cual requiere que el objeto que invoca conozca el funcionamiento interno del llamado).
- La segunda consiste en tener un sistema de paso de mensajes para que los objetos puedan enviarse notificaciones entre sí.

Tomemos como ejemplo un juego sencillo el cual consiste en un tablero sobre el cuál se encuentran hormigas (personajes buenos) y matadores (enemigos). Las hormigas deben recolectar la comida del tablero evitando a los matadores, mientras que éstos últimos sólo tienen como objetivo exterminar las hormigas. A continuación se puede observar un ejemplo de dicho juego, dónde los puntos amarillos son las hormigas, los rojos los matadores y los cuadrados negros la comida:

Primera alternativa

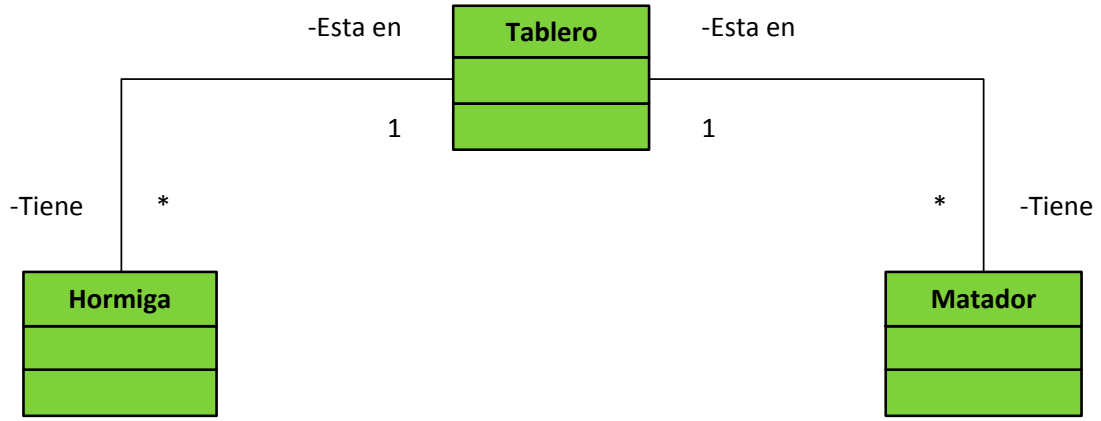
La primera alternativa que planteamos es que todas las comunicaciones entre objetos de juego pasen por alguna clase de mayor jerarquía que los contenga a todos y nos permita obtener mediante funciones obtener punteros a los objetos que nos interesan para poder invocar sus métodos. Si hacemos un diagrama de clases de los objetos que hemos descripto tendríamos:

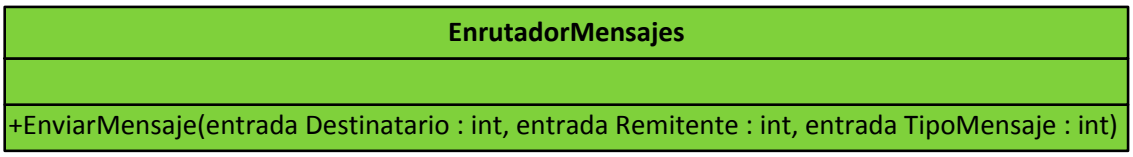


En este diagrama la clase Tablero contiene punteros a todas las hormigas y matadores y contiene métodos para que los objetos puedan obtener punteros a una hormiga o un matador dado un identificador entero único de cada objeto. Además en este ejemplo la hormiga tiene métodos para que otra le avise dónde encontró comida. Ahora para ver como esto funciona, imaginemos que tenemos dos hormigas: Penny y Lane con ids 1 y 2 respectivamente. Penny se las ingenia para llegar a una posición dónde encontró comida, y dese avisarle a Lane para que vaya al mismo punto. Usando este diagrama de clases lo primero que debe hacer Penny es invocar el método GetHormiga() de la clase Tablero con el parámetro 2 para recuperar un puntero a la Hormiga Lane. Una vez que tiene el puntero debe invocar el método AvisarComida pasándole la posición. De esta forma logramos que se comuniquen dos hormigas, pero todas las comunicaciones se realizan pasando por la clase de mayor jerarquía Tablero.

~~Segunda alternativa~~ Sistema de mensajes

La alternativa a lo planteado anteriormente es implementar algún sistema de paso de mensajes entre objetos. En este caso los objetos crean un mensaje y lo entregan a una clase dedicada a administrarlos. Dicha clase se encarga de entregarlos a los destinatarios. Esto hace que él envió de notificaciones sea más sencillo y permite separar la lógica del "aviso" en una clase especializada. Veamos cómo sería un diagrama de clases para el ejemplo anterior:





En este caso tenemos una clase singleton *EnrutadorMensajes* que posee un método *EnviarMensaje*. El mismo permite enviar mensajes entre entidades de juego de manera sencilla identificándolas mediante su id. En este caso ya no es necesario obtener un puntero a la hormiga ni invocar su método ya que el enrutador de mensajes se debe encargar. Y por supuesto debemos programar la reacción de la hormiga al recibir cada tipo de mensaje.

¿Entonces?

Tenemos dos alternativas válidas y levemente diferentes para implementar la comunicación entre nuestros objetos de juego. El sistema de mensajes suele ser más general y flexible que la primer alternativa, pero esto depende en gran medida del proyecto. No hay una forma “óptima” ni ideal para la comunicación sólo recomendaciones basadas en casos generalmente particulares. Por ello es cuestión de experiencia y horas silla poder elegir correctamente una alternativa u otra.

Bibliografía

"Programming Game AI by Example", Mat Buckland. Wordware Publishing, 2005.

"AI for Game Developers", David M. Bourg, Glenn Seeman. O'Reilly,2004.

"Artificial Intelligence: A Modern Approach"*, Stuart Russell, Peter Norvig.*

"The C++ Programming Language", Bjarne Stroustrup. Addison Wesley.

Web: http://www.ai-junkie.com/

Web: http://es.wikipedia.org/wiki/Singleton