estudio de caso: el patrón de diseño State

par Rodrigo Pons (home)

Date de publication : 24/09/2009

Dernière mise à jour : 05/07/2010

El *design pattern State*, o **patrón de diseño** Estado, es uno de los más utilizados. En su definición inicial, es muy sencillo, pero su implementación concreta puede variar mucho según el contexto. Les propongo, en este artículo, analizar varias implementaciones posibles.

volver a la página inicial

I - Introducción

I-A - Generalidades

El *pattern State* (patrón Estado) es un *behavioural design pattern* (patrón de comportamiento).
Design patterns proportionan modelos teóricos que permiten resolver problemas recurrentes. Han sido pensados para responder a numerosos problemas, de manera que puedan ser aplicados con éxito en los contextos más diferentes posible. De hecho, son utilizados regularmente por los desarrolladores que les gusta el paradigma objeto, y los encontramos en numerosos programas.
En lo que concirene a los design pattern, pienso que es inútil aprenderlos de memoria. Ya que a fuerza de utilizarlos acabamos por conocerlos sin tener que hacer el esfuerzo de aprenderlos. En cambio, creo que es importante conocerlos, justo para saber que existan y saber donde encontrar informaciones sobre ellos cuando queremos utilizarlos concretamente. Ádemas, la comprensión de estos diseño pattern es un ejercicio excelente que permite comprender un cierto número de problemáticas relativas al paradigma objeto y aprehender soluciones eficaces, generales y elegantes.
EL UML define varios tipos diagramas, que permiten representar diferentes etapas de la realización de un software, y de diferentes maneras. Para comprender este artículo, solo necesitará saber leer un diagrama de clase, y saber lo que es un diagrama de estado.
Aquí utilizaré sólo diagramas muy simplificados, limpiados de todo lo que no esta directamente relacionado con el sujeto.

I-B - Definición del patrón Estado

La idea de este patrón es de obtener una clase, que llamare *contexto* (context), que tendrá un comportamiento circunstanciado al *estado* corriente, es decir un comportamiento diferente según su *estado*. . Vamos a crear una clase abstracta que define la interfaz pública de los estados. En resumen, vamos a poner en ella las funciones del *contexto* cuyo comportamiento puede variar. Es el *estado abstracto* (abstract state). . Luego hay que implementar los *estados concretos*, que heredarán del *estado abstracto*
. . Después creamos un puntero sobre el *estado corriente*, en una variable miembro del *contexto*. . Cuando se desea que el *contexto* cambie de estado, solo hay que modificar el *estado corriente*.

Una implementación de un patrón estado está considerada como mejor si respeta los criterios siguientes: . El **LSP**[en] es respetado (entre él *estado abstracto* y los *estados concretos*). Esto facilita la manipulación de los estados y puede evitar situaciones complejas y\o ambiguas.
. Los *estados concretos* no poseen datos. Los datos deben estar en el *contexto* o en él *estado abstracto*.

I-C - El diagrama de estados

El diagrama de estados es generalmente menos conocido que el diagrama de clases, entonces voy a hablar un poco de esto. Este diagrama sirve para representar un autómata de estado terminado (o grafo). Es extremadamente sencillo: hay unos estados - los nudos del grafo - y eventos (o transiciones) - los arcos del grafo - que permiten irse de un estado a un otro.

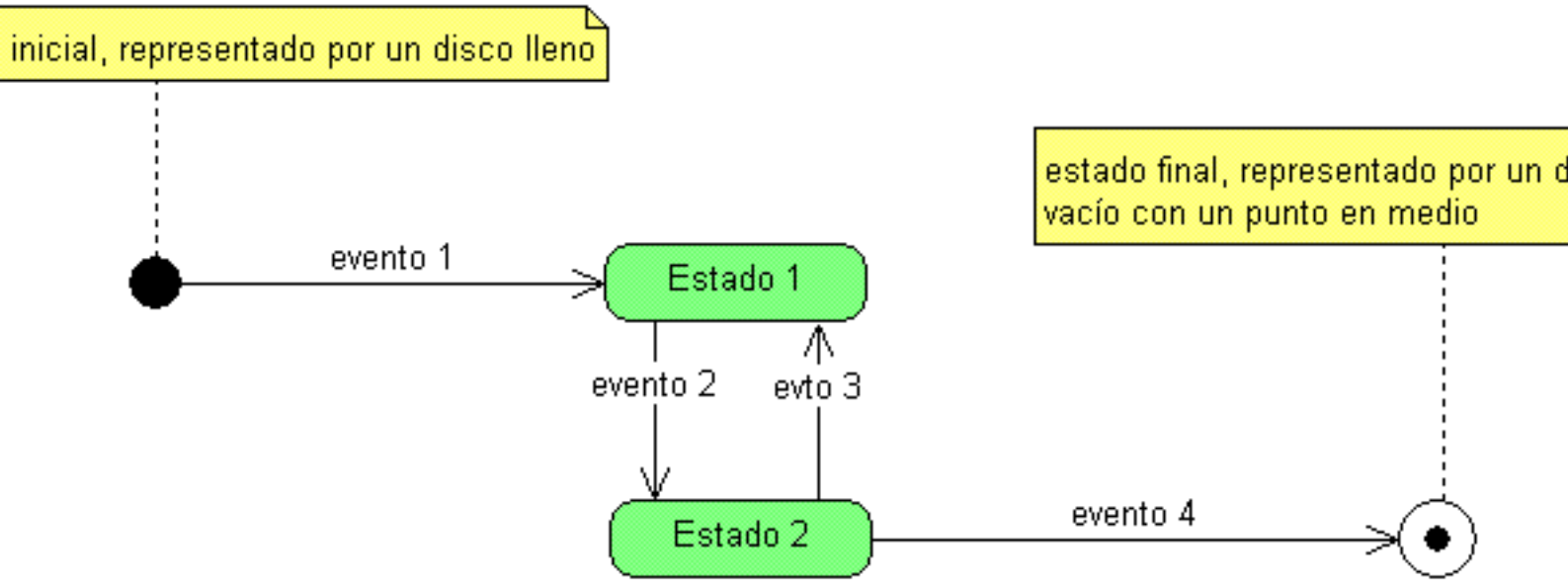


diagrama de estados

Vamos a tomar un ejemplo didáctico sencillo y sin relación con el desarrollo software (UML es concebido para representar todo tipo de problemas, no únicamente de programación). Tomemos pues el ejemplo de un grabador de CD. Podrá encontrarse en tres estados diferentes: **stand-by** (no hace nada), **lectura**, y **grabación**. Para pasar de uno a otro de estos estados, tenemos 3 eventos: **play**, **stop**, y **record**, que corresponden por ejemplo a la activación por el usuario de la tecla correspondiente del telemando.

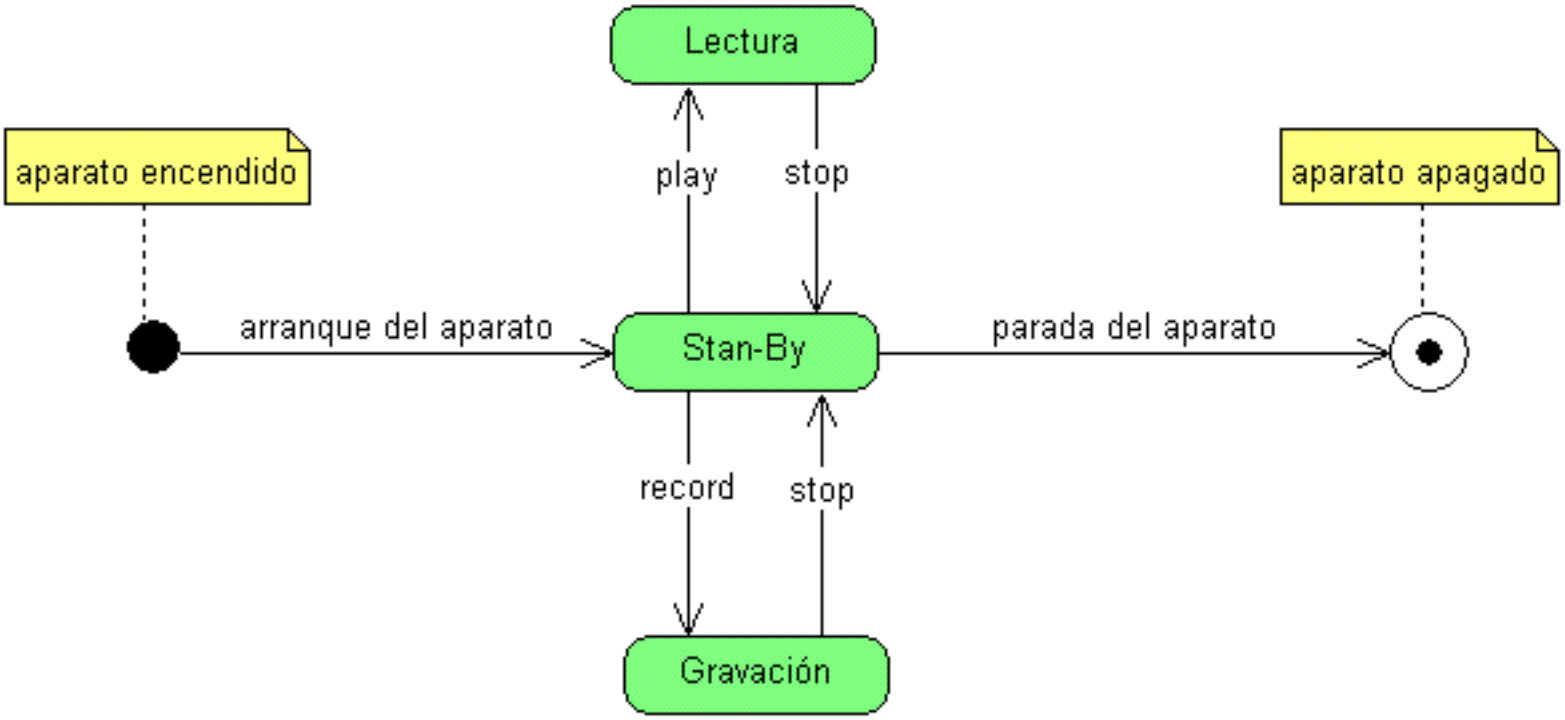
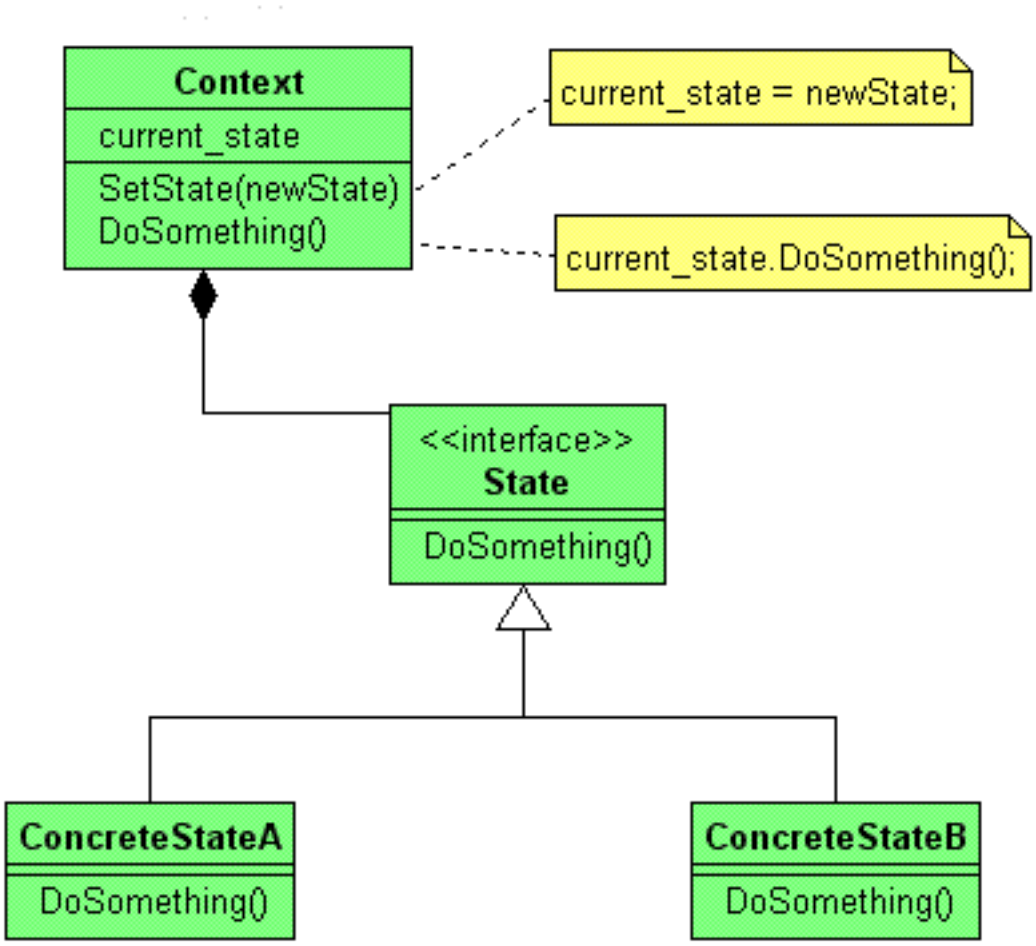


diagrama de estados

II - Modelo inicial

El patrón Estado el más sencillo es el de **la página wikipedia sobre el pattern State**[en].



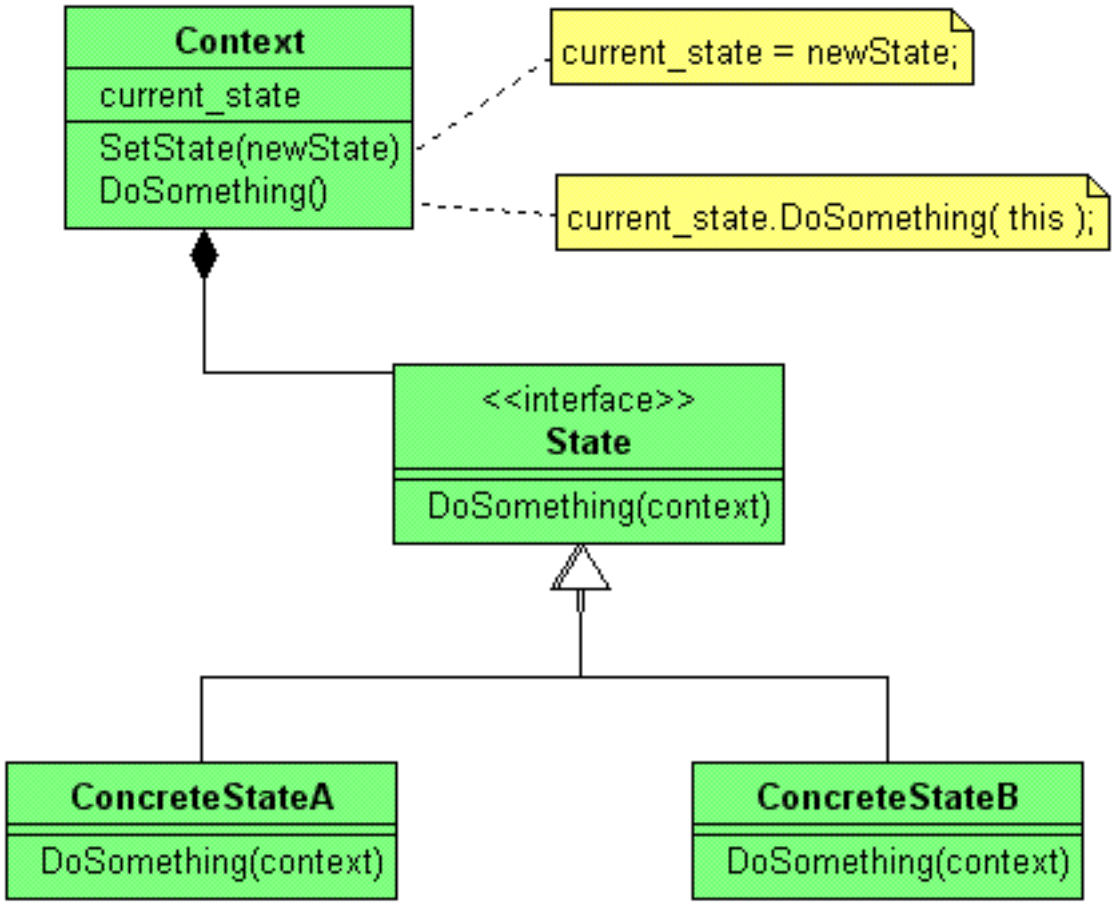
patrón Estado el más sencillo

Diagrama que puede dar el código que sigue por ejemplo:

```
#include <iostream>
#include <boost/shared_ptr.hpp> // por boost::shared_ptr
using namespace std;

class State
{
public:
virtual ~State() {}
 void Action() { DoSomething(); }
private:
 virtual void DoSomething() = 0;
};

class ConcreteStateA : public State
{
private:
 void DoSomething() { cout << "ConcreteStateA: DoSomething" << endl; }
};

class ConcreteStateB : public State
{
private:
 void DoSomething() { cout << "ConcreteStateB: DoSomething" << endl; }
```

```
};

class Context
{
public:
 Context():currentState( new ConcreteStateA ) {}

 void SetState( State * newState )
 {
  currentState.reset( newState );
 }

 void DoSomething() { currentState->Action(); }

private:
 boost::shared_ptr<State> currentState;
};

int main()
{
 Context context;
 context.SetState( new ConcreteStateA );
 context.DoSomething();
 context.SetState( new ConcreteStateB );
 context.DoSomething();
 // ...

 cin.get();
}
```

En el código más arriba, hay una utilización de un puntero inteligente (*boost::shared_ptr*). Los paradigmas del C++ moderno aconsejan la utilización sistemática de los punteros inteligentes. Para saber más sobre eso, ver:
- **La página de Zator sobre los apuntadores inteligentes**
- **La página de boost sobre los punteros inteligentes**
El del GoF[en] (GoF=**Gang Of Four**) no es fundamentalmente distinto, solo añade el pasaje del contexto a la acción efectuada por el estado.



patrón Estado del GoF

El código es casi igual que el primero:

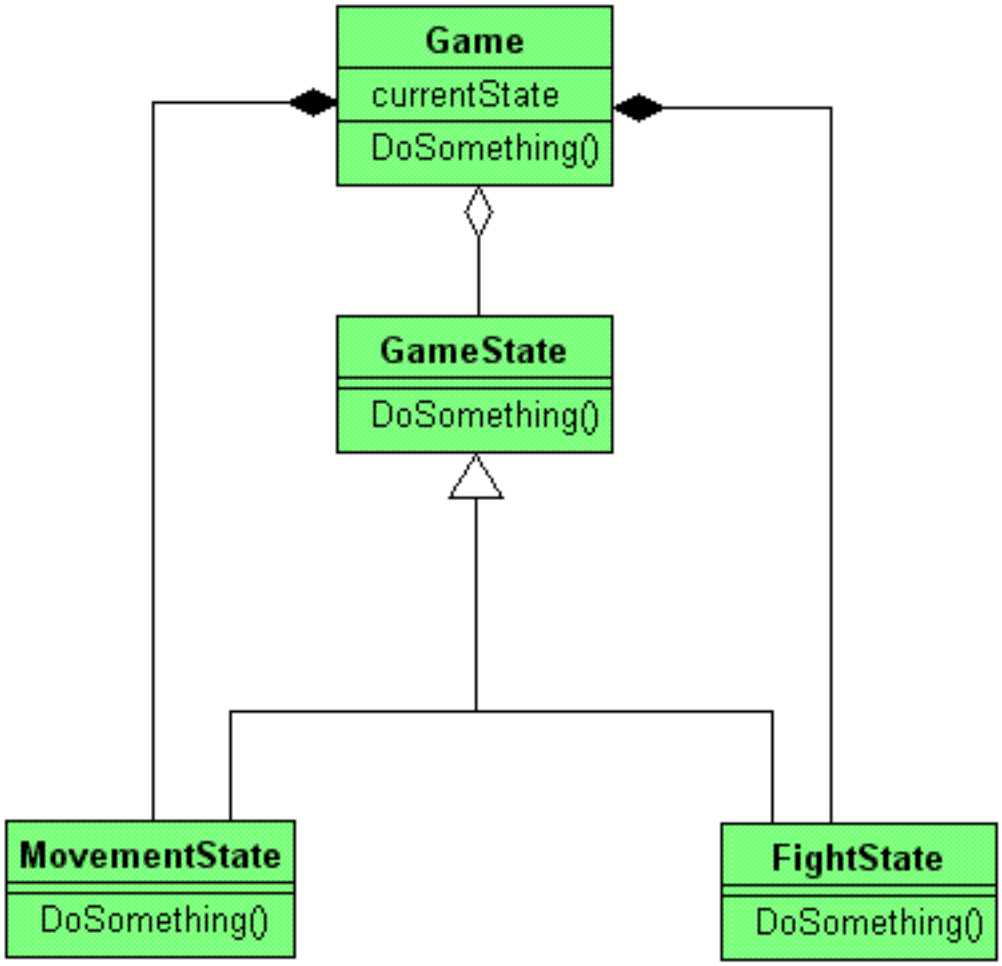
```
class State
{
public:
 void Action( Context context) { DoSomething( context ); }
private:
 virtual void DoSomething( Context ) = 0;
};
```

Esta implementación permite evitar tener que almacenar datos en los estados, almacenando todos estos datos en el contexto. En cambio, añade una dependencia de los estados hacia el contexto que no me gusta mucho. Otra razón para la cual no me gusta demasiado esta implementación, es que plantea un problema (un caso de conciencia por lo menos) si se desea que los estados puedan acceder a los datos/funciones no públicos del contexto.

III - Implementaciones en c++

III-A - Implementación "YaTuS"

Llamé esta implementación así porque es la que utilizo varias veces en **YaTuS**. Esta implementación me parece buena cuando tenemos pocos estados y cuando los estados concretos son gordos.
Tomemos por ejemplo los estados de la clase Game. Esta clase representa la partida corriente, de un punto de vista del "comportamiento". Posee 2 estados, que respectivamente representan la fase de movimiento (MovementState) y la fase de combate (FightState). Su diagrama de clase es el siguiente:



patrón Estado 'YaTuS'

Lo que cambia conr tespecto al patrón calsico (ver **capítulo II**), es que aquí el contexto (Game) posee una instancia de cada estado. En esta implementación, el contexto (Game) posee también un puntero que apuntará sucesivamente en cada una de estas instancias.
Esta implementación deja la responsabilidad de las transiciones a la clase que posee el contexto.

```
#include <iostream>
#include <string>
using namespace std;

class GameState
{
public:
 void Action() { DoSomething(); }
private:
 virtual void DoSomething() = 0 ;
};

class MovementState : public GameState
{
private:
 void DoSomething() { cout << "MovementState::DoSomething" << endl; }
};

class FightState : public GameState
{
private:
 void DoSomething() { cout << "FightState::DoSomething" << endl; }
};
```

```
class Game
{
public:
 Game() : m_currentState( &m_mvtState ) {}

 void DoSomething()
 {
  m_currentState->Action();
 }

 void UpdateCurPhase( const std::string & stateName )
 {
  if ( stateName == "movement" )
   m_currentState = &m_mvtState;
  else
   m_currentState = &m_fightState;
 }

private:
 MovementState m_mvtState;
 FightState m_fightState;
 GameState * m_currentState;
};

int main()
{
 Game game;
 game.DoSomething();
 game.UpdateCurPhase( "fight" );
 game.DoSomething();
 game.UpdateCurPhase( "movement" );
 game.DoSomething();

 cin.get();
 return 0;
}
```

Ventajas:
- Los estados son creados sólo una vez, y el estado corriente es sólo un puntero que va a apuntar sobre uno de estos estados ya instanciados. Entonces, no hay ninguna manipulación de memoria (new/delete), lo que es a menudo una buena cosa cuando las clases estados concretos son un poco gordas.
- Este método no impone limitaciones relacionadas con la responsabilidad de las transiciones. En efecto, puede ser implementada en el contexto (es el caso en el código más arriba), o dejada en los estados concretos.
Inconvenientes:
- Esta implementación no es deseable cuando hay muchos estados de tipo diferente.
- Esta implementación implica una gestión bastante especializada de los estados (el contexto debe conocer los diferentes tipos de estados). Esto puede hacer que sea más difícil la modificación del diagrama de estado.

III-B - FSM State

La definición inicial del patrón Estado no especifica quien debe encargarse de las transiciones (el *contexto*, los estados, otro...).
La implementación FSM[en] es un ejemplo de implementación del patrón Estado que nos propone Vince Huston. Es concebida de manera que las transiciones sean definidas en el *contexto*, y no en los estados.
La principal ventaja de esta implementación es que todas las transiciones son centralizadas (en el contexto), facilitando así la comprensión y el mantenimiento.

```
class FSMstate { public:
 virtual void on()  { cout << "undefined combo" << endl; }
 virtual void off() { cout << "undefined combo" << endl; }
 virtual void ack() { cout << "undefined combo" << endl; } };

class FSM {
public:
```

```
 FSM();
 void on()   { states[current]->on();  current = next[current][0]; }
 void off()  { states[current]->off(); current = next[current][1]; }
 void ack()  { states[current]->ack(); current = next[current][2]; }
private:
 FSMstate*  states[3];
 int        current;
 int        next[3][3];
};

class A : public FSMstate { public:
 void on()  { cout << "A, on ==> A" << endl; }
 void off() { cout << "A, off ==> B" << endl; }
 void ack() { cout << "A, ack ==> C" << endl; }
};
class B : public FSMstate { public:
 void off() { cout << "B, off ==> A" << endl; }
 void ack() { cout << "B, ack ==> C" << endl; }
};
class C : public FSMstate { public:
 void ack() { cout << "C, ack ==> B" << endl; }
};

FSM::FSM() {
 states[0] = new A; states[1] = new B; states[2] = new C;
 current = 1;
 next[0][0] = 0; next[0][1] = 1; next[0][2] = 2;
 next[1][0] = 1; next[1][1] = 0; next[1][2] = 2;
 next[2][0] = 2; next[2][1] = 2; next[2][2] = 1; }

enum     Message { On, Off, Ack };
Message  messageArray[10] = { On,Off,Off,Ack,Ack,Ack,Ack,On,Off,Off };

int main() {
 FSM  fsm;
 for (int i = 0; i < 10; i++) {
  if (messageArray[i] == On)        fsm.on();
  else if (messageArray[i] == Off)  fsm.off();
  else if (messageArray[i] == Ack)  fsm.ack(); }
}
```

Esta implementación es muy orientada "grafo". FSM es el contexto. El array *next*, variable miembro de FSM, no es nada más que el grafo de transiciones, o STT (state transición cuenta).
Un buen estudio sobre este modo de proceder es propuesto por Robert C. Martín **aquí**[en].

III-C - Dejar el control a los estados concretos

En los implementaciones que hemos visto hasta ahora, es el contexto, o aun el dueño del contexto, que tiene la responsabilidad de las transiciones. Pero suele suceder, según el diseño del programa, que sea preferible dejar está responsabilidad a los estados concretos. En este caso, hay que optar por un mecanismo que no hace new/delete del estado corriente mientras las transiciones. La solución más utilizada consiste en pasar el contexto en parametro de las funciones de los estados.
Existe una implementación corriente que me gusta mucho, porque es muy modular. He aquí al que se parece:

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

class State
{
public:
 const string Action( int i ) { return DoSomething(i); }
private:
 virtual const string DoSomething( int i ) = 0;
```

```
};

class StateA: public State
{
private:
 const string DoSomething( int i ) {
  cout << "A -> ";
  return (i%2==0) ? "B" : "C";
 }
};

class StateB: public State
{
private:
 const string DoSomething( int i ) {
  cout << "B -> ";
  return (i%3==0) ? "C" : "A";
 }
};

class StateC: public State
{
private:
 const string DoSomething( int i ) {
  cout << "C -> ";
  return (i%4==0) ? "B" : "A";
 }
};

class Graph
{
public:
 Graph() {
  states_["A"] = new StateA;
  states_["B"] = new StateB;
  states_["C"] = new StateC;
  currentState_ = states_["A"];
 }

 void DoSomething( int i ){
  string nextState = currentState_->Action(i);
  currentState_ = states_[nextState];
 }

private:
 map<string, State*> states_;
 State * currentState_;
};


int main()
{
 Graph graph;
 for (int i=0; i<20 ; i++)
  graph.DoSomething(i);

 cout << "end" << endl;
 cin.get();
 return 0;
}
```

Salida del programa de arriba:

```
A -> B -> A -> B -> C -> B -> A -> B -> A -> B -> C -> A -> C -> B -> A -> B ->
C -> B -> A -> B -> end
```

El principio de esta implementación es que no hay función *changeState()*, la gestión de los estados se hace automáticamente: las llamadas a las funciones implementadas por los estados concretos deben devolver el nuevo estado, y el contexto se actualiaz automaticamente.
La principal ventaja de esta implementación es que permite la construcción de un grafo, tan complejo sea, de modo bastante intuitivo.
Otra ventaja es que permite el añadido de nuevos estados con poco gasto. En efecto, para añadir un nuevo estado, basta con crear la clase de este estado, y declararlo en el constructor del contexto (*states_["identificador del estado"] = new StateX;*) y ya esta.

Vemos en esta implementación que, aunque son los estados concretos que deciden el estado siguiente, el código que efectúa la transición es en el contexto (es la línea: *currentState_=states_[nextState];*).
Es posible ir más lejos y dejar esta responsabilidad en los estados concretos. Para hacerlo, hay que pasar el contexto en parámetro a las funciones que van a modificar este estado, como dicho en el precedente párrafo.

III-D - Los estados en forma de singleton

Una implementación frecuente del patrón Estado es implementar los estados en forma de **singleton**. La encontramos por ejemplo en **este artículo de Stephen B. Morris** .
Hay varias ventajas en proceder así. En primer lugar, ya que cada estado es único, es bastante lógico de hacerlo en forma de singleton. Luego, esto evita hacer *new* y *delete* durante la ejecución.

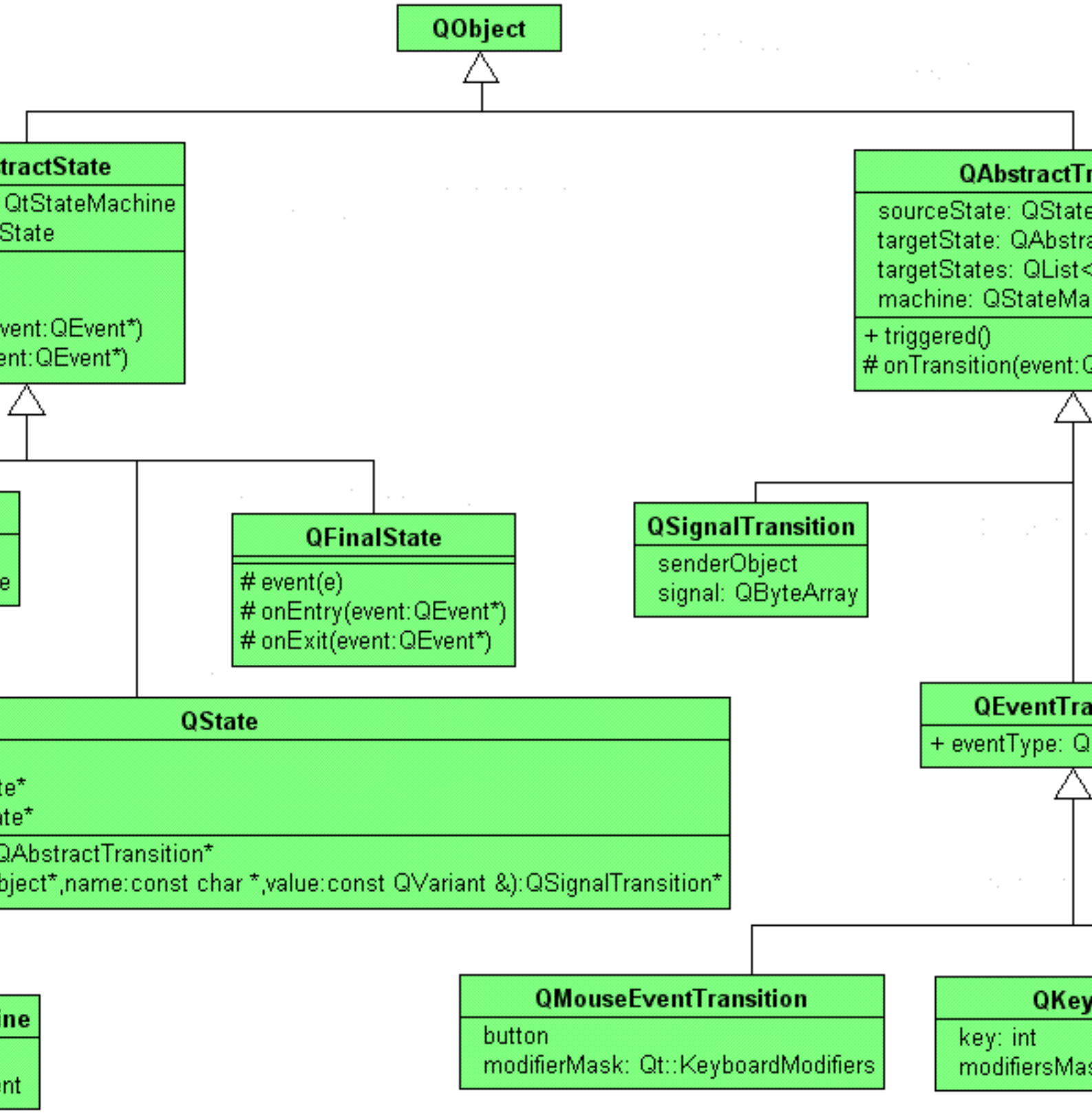
Aplicando esta técnica al ejemplo del párrafo III-a, daría algo como esto:

```
#include <iostream>
#include <string>
using namespace std;

class GameState
{
public:
 void Action() { DoSomething(); }
private:
 virtual void DoSomething() = 0 ;
};

class MovementState : public GameState
{
public:
 static MovementState& GetInstance() { return instance_; }
private:
 void DoSomething() { cout << "MovementState::DoSomething" << endl; }
private:
 static MovementState instance_;
 MovementState(){}
 ~MovementState(){}
};

class FightState : public GameState
{
public:
 static FightState& GetInstance() { return instance_; }
private:
 void DoSomething() { cout << "FightState::DoSomething" << endl; }
private:
 static FightState instance_;
 FightState(){}
 ~FightState(){}
};

MovementState MovementState::instance_;
FightState  FightState::instance_;

class Game
{
```

```
public:
 Game()
 : m_mvtState( MovementState::GetInstance() )
 , m_fightState( FightState::GetInstance() )
 , m_currentState( &m_mvtState )
 {}

 void DoSomething()
 {
  m_currentState->Action();
 }

 void UpdateCurPhase( const std::string & stateName )
 {
  if ( stateName == "movement" )
   m_currentState = &m_mvtState;
  else
   m_currentState = &m_fightState;
 }

private:
 MovementState& m_mvtState;
 FightState& m_fightState;
 GameState * m_currentState;
};

int main()
{
 Game game;
 game.DoSomething();
 game.UpdateCurPhase( "fight" );
 game.DoSomething();
 game.UpdateCurPhase( "movement" );
 game.DoSomething();

 cin.get();
 return 0;
}
```

III-E - La máquina de estado según Qt

Qt propone una interfaz muy elaborada para la creación de nuestra propia máquina de estado: **el framework "state machine" de Qt.**
Le propongo aquí un diagrama de clase muy simplificado de la máquina de estado de Qt:



Qt state

Todavía no he hablado mucho de eso, pero la noción de transición es primordial en una máquina de estado. Entonces es legítimo dedicar una clase a esta noción de transición. Es (entre otras cosas) lo que hace Qt: vemos claramente en el diagrama más arriba que la máquina de estado contiene 2 tipos de objetos: los estados y las transiciones.

IV - Enlaces

. **Patrón de diseño en wikipedia**
. **El patrón Estado en wikipedia**
. **Diagrama de estados en wikipedia**