Library of Congress Cataloging-in-Publication Data

Buckland, Mat.
Programming game AI by example / by Mat Buckland.
p. cm.
Includes index.
ISBN 1-55622-078-2 (pbk.)
1. Computer games—Design. 2. Computer games—Programming. 3. Computer graphics. I. Title.
QA76.76.C672B85 2004
794.8'1526—dc22 2004015103

2320 Los Rios Boulevard
Plano, Texas 75074

Printed in the United States of America

ISBN 1-55622-078-2

10 9 8 7 6 5 4 3
0409

All inquiries for volume purchases of this book should be addressed to Wordware Publishing, Inc., at the above address. Telephone inquiries may be made by calling:

(972) 423-0090

Programming Game AI by Example

Mat Buckland

Wordware Publishing, Inc.

Chapter 2

State-Driven Agent Design

Finite state machines, or FSMs as they are usually referred to, have for many years been the AI coder's instrument of choice to imbue a game agent with the illusion of intelligence. You will find FSMs of one kind or another in just about every game to hit the shelves since the early days of video games, and despite the increasing popularity of more esoteric agent architectures, they are going to be around for a long time to come. Here are just some of the reasons why:

They are quick and simple to code. There are many ways of programming a finite state machine and almost all of them are reasonably simple to implement. You'll see several alternatives described in this chapter together with the pros and cons of using them.

They are easy to debug. Because a game agent's behavior is broken down into easily manageable chunks, if an agent starts acting strangely, it can be debugged by adding tracer code to each state. In this way, the AI programmer can easily follow the sequence of events that precedes the buggy behavior and take action accordingly.

They have little computational overhead. Finite state machines use hardly any precious processor time because they essentially follow hard-coded rules. There is no real "thinking" involved beyond the if-*this*-then-*that* sort of thought process.

They are intuitive. It's human nature to think about things as being in one state or another and we often refer to ourselves as being in such and such a state. How many times have you "got yourself into a state" or found yourself in "the right state of mind"? Humans don't really work like finite state machines of course, but sometimes we find it useful to think of our behavior in this way. Similarly, it is fairly easy to break down a game agent's behavior into a number of states and to create the rules required for manipulating them. For the same reason, finite state machines also make it easy for you to discuss the design of your AI with non-programmers (with game producers and level designers for example), providing improved communication and exchange of ideas.

They are flexible. A game agent's finite state machine can easily be adjusted and tweaked by the programmer to provide the behavior required by the game designer. It's also a simple matter to expand the scope of an agent's behavior by adding new states and rules. In addition, as your AI

skills grow you'll find that finite state machines provide a solid backbone with which you can combine other techniques such as fuzzy logic or neural networks.

What Exactly Is a Finite State Machine?

Historically, a finite state machine is a rigidly formalized device used by mathematicians to solve problems. The most famous finite state machine is probably Alan Turing's hypothetical device: the Turing machine, which he wrote about in his 1936 paper, "On Computable Numbers." This was a machine presaging modern-day programmable computers that could perform any logical operation by reading, writing, and erasing symbols on an infinitely long strip of tape. Fortunately, as AI programmers, we can forgo the formal mathematical definition of a finite state machine; a descriptive one will suffice:

> *A finite state machine is a device, or a model of a device, which has a finite number of states it can be in at any given time and can operate on input to either make transitions from one state to another or to cause an output or action to take place. A finite state machine can only be in one state at any moment in time.*

The idea behind a finite state machine, therefore, is to decompose an object's behavior into easily manageable "chunks" or states. The light switch on your wall, for example, is a very simple finite state machine. It has two states: on and off. Transitions between states are made by the input of your finger. By flicking the switch up it makes the transition from off to on, and by flicking the switch down it makes the transition from on to off. There is no output or action associated with the off state (unless you consider the bulb being off as an action), but when it is in the on state electricity is allowed to flow through the switch and light up your room via the filament in a lightbulb. See Figure 2.1.

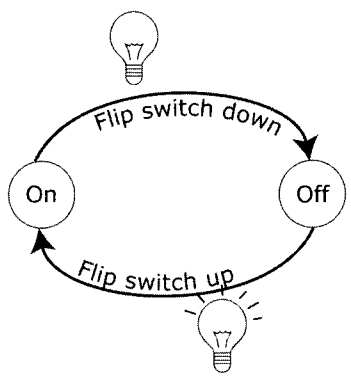

Figure 2.1. A light switch is a finite state machine. (Note that the switches are reversed in Europe and many other parts of the world.)

Of course, the behavior of a game agent is usually much more complex than a lightbulb (thank goodness!). Here are some examples of how finite state machines have been used in games.

- The ghosts' behavior in Pac-Man is implemented as a finite state machine. There is one Evade state, which is the same for all ghosts, and then each ghost has its own Chase state, the actions of which are implemented differently for each ghost. The input of the player eating one of the power pills is the condition for the transition from Chase to Evade. The input of a timer running down is the condition for the transition from Evade to Chase.
- Quake-style bots are implemented as finite state machines. They have states such as FindArmor, FindHealth, SeekCover, and RunAway. Even the weapons in Quake implement their own mini finite state machines. For example, a rocket may implement states such as Move, TouchObject, and Die.
- Players in sports simulations such as the soccer game FIFA2002 are implemented as state machines. They have states such as Strike, Dribble, ChaseBall, and MarkPlayer. In addition, the teams themselves are often implemented as FSMs and can have states such as KickOff, Defend, or WalkOutOnField.
- The NPCs (non-player characters) in RTSs (real-time strategy games) such as Warcraft make use of finite state machines. They have states such as MoveToPosition, Patrol, and FollowPath.

Implementing a Finite State Machine

There are a number of ways of implementing finite state machines. A naive approach is to use a series of if-then statements or the slightly tidier mechanism of a switch statement. Using a switch with an enumerated type to represent the states looks something like this:

```
enum StateType{RunAway, Patrol, Attack};

void Agent::UpdateState(StateType CurrentState)
{
  switch(CurrentState)
  {
  case state_RunAway:

    EvadeEnemy();

    if (Safe())
    {
      ChangeState(state_Patrol);
    }

    break;
```

```
  case state_Patrol:

    FollowPatrolPath();

    if (Threatened())
    {
      if (StrongerThanEnemy())
      {
        ChangeState(state_Attack);
      }
      else
      {
        ChangeState(state_RunAway);
      }
    }

    break;

  case state_Attack:

    if (WeakerThanEnemy())
    {
       ChangeState(state_RunAway);
    }

    else
    {
      BashEnemyOverHead();
    }

    break;

  }//end switch
}
```

Although at first glance this approach seems reasonable, when applied practically to anything more complicated than the simplest of game objects, the switch/if-then solution becomes a monster lurking in the shadows waiting to pounce. As more states and conditions are added, this sort of structure ends up looking like spaghetti very quickly, making the program flow difficult to understand and creating a debugging nightmare. In addition, it's inflexible and difficult to extend beyond the scope of its original design, should that be desirable... and as we all know, it most often is. Unless you are designing a state machine to implement very simple behavior (or you are a genius), you will almost certainly find yourself first tweaking the agent to cope with unplanned-for circumstances before honing the behavior to get the results you thought you were going to get when you first planned out the state machine!

Additionally, as an AI coder, you will often require that a state perform a specific action (or actions) when it's initially entered or when the state is exited. For example, when an agent enters the state RunAway you may

want it to wave its arms in the air and scream "Arghhhhhhh!" When it finally escapes and changes state to Patrol, you may want it to emit a sigh, wipe its forehead, and say "Phew!" These are actions that only occur when the RunAway state is entered or exited and not during the usual update step. Consequently, this additional functionality must ideally be built into your state machine architecture. To do this within the framework of a switch or if-then architecture would be accompanied by lots of teeth grinding and waves of nausea, and produce very ugly code indeed.

State Transition Tables

A better mechanism for organizing states and affecting state transitions is a *state transition table*. This is just what it says it is: a table of conditions and the states those conditions lead to. Table 2.1 shows an example of the mapping for the states and conditions shown in the previous example.

Table 2.1. A simple state transition table

Current State	Condition	State Transition
Runaway	Safe	Patrol
Attack	WeakerThanEnemy	RunAway
Patrol	Threatened AND StrongerThanEnemy	Attack
Patrol	Threatened AND WeakerThanEnemy	RunAway

This table can be queried by an agent at regular intervals, enabling it to make any necessary state transitions based on the stimulus it receives from the game environment. Each state can be modeled as a separate object or function existing external to the agent, providing a clean and flexible architecture. One that is much less prone to spaghettification than the if-then/switch approach discussed in the previous section.

Someone once told me a vivid and silly visualization can help people to understand an abstract concept. Let's see if it works...

Imagine a robot kitten. It's shiny yet cute, and has wire for whiskers and a slot in its stomach where cartridges — analogous to its states — can be plugged in. Each of these cartridges is programmed with logic, enabling the kitten to perform a specific set of actions. Each set of actions encodes a different behavior; for example, "play with string," "eat fish," or "poo on carpet." Without a cartridge stuffed inside its belly the kitten is an inanimate metallic sculpture, only able to sit there and look cute... in a Metal Mickey kind of way.

The kitten is very dexterous and has the ability to autonomously exchange its cartridge for another if instructed to do so. By providing the rules that dictate when a cartridge should be switched, it's possible to string together sequences of cartridge insertions permitting the creation of all

sorts of interesting and complicated behavior. These rules are programmed onto a tiny chip situated inside the kitten's head, which is analogous to the state transition table we discussed earlier. The chip communicates with the kitten's internal functions to retrieve the information necessary to process the rules (such as how hungry Kitty is or how playful it's feeling).

As a result, the state transition chip can be programmed with rules like:

```
IF Kitty_Hungry AND NOT Kitty_Playful
    SWITCH_CARTRIDGE eat_fish
```

All the rules in the table are tested each time step and instructions are sent to Kitty to switch cartridges accordingly.

This type of architecture is very flexible, making it easy to expand the kitten's repertoire by adding new cartridges. Each time a new cartridge is added, the owner is only required to take a screwdriver to the kitten's head in order to remove and reprogram the state transition rule chip. It is not necessary to interfere with any other internal circuitry.

Embedded Rules

An alternative approach is to *embed the rules for the state transitions within the states themselves*. Applying this concept to Robo-Kitty, the state transition chip can be dispensed with and the rules moved directly into the cartridges. For instance, the cartridge for "**play with string**" can monitor the kitty's level of hunger and instruct it to switch cartridges for the "**eat fish**" cartridge when it senses hunger rising. In turn the "**eat fish**" cartridge can monitor the kitten's bowel and instruct it to switch to the "**poo on carpet**" cartridge when it senses poo levels are running dangerously high.

Although each cartridge may be aware of the existence of any of the other cartridges, each is a self-contained unit and not reliant on any external logic to decide whether or not it should allow itself to be swapped for an alternative. As a consequence, it's a straightforward matter to add states or even to swap the whole set of cartridges for a completely new set (maybe ones that make little Kitty behave like a raptor). There's no need to take a screwdriver to the kitten's head, only to a few of the cartridges themselves.

Let's take a look at how this approach is implemented within the context of a video game. Just like Kitty's cartridges, states are encapsulated as objects and contain the logic required to facilitate state transitions. In addition, all state objects share a common interface: a pure virtual class named `State`. Here's a version that provides a simple interface:

```
class State
{
public:
  virtual void Execute (Troll* troll) = 0;
};
```


Now imagine a `Troll` class that has member variables for attributes such as health, anger, stamina, etc., and an interface allowing a client to query and adjust those values. A `Troll` can be given the functionality of a finite state machine by adding a pointer to an instance of a derived object of the `State` class, and a method permitting a client to change the instance the pointer is pointing to.

```
class Troll
{
  /* ATTRIBUTES OMITTED */

  State* m_pCurrentState;

public:

  /* INTERFACE TO ATTRIBUTES OMITTED */

  void Update()
  {
    m_pCurrentState->Execute(this);
  }

  void ChangeState(const State* pNewState)
  {
    delete m_pCurrentState;
    m_pCurrentState = pNewState;
  }
};
```

When the `Update` method of a `Troll` is called, it in turn calls the `Execute` method of the current state type with the `this` pointer. The current state may then use the `Troll` interface to query its owner, to adjust its owner's attributes, or to effect a state transition. In other words, how a `Troll` behaves when updated can be made completely dependent on the logic in its current state. This is best illustrated with an example, so let's create a couple of states to enable a troll to run away from enemies when it feels threatened and to sleep when it feels safe.

```
//-----------------------------------State_RunAway
class State_RunAway : public State
{
public:

  void Execute(Troll* troll)
  {
    if (troll->isSafe())
    {
      troll->ChangeState(new State_Sleep());
    }
```

```
    else
    {
      troll->MoveAwayFromEnemy();
    }
  }
};

//---------------------------------State_Sleep
class State_Sleep : public State
{
public:

  void Execute(Troll* troll)
  {
    if (troll->isThreatened())
    {
      troll->ChangeState(new State_RunAway())
    }

    else
    {
      troll->Snore();
    }
  }
};
```

As you can see, when updated, a troll will behave differently depending on which of the states `m_pCurrentState` points to. Both states are encapsulated as objects and both provide the rules effecting state transition. All very neat and tidy.

This architecture is known as the *state design pattern* and provides an elegant way of implementing state-driven behavior. Although this is a departure from the mathematical formalization of an FSM, it is intuitive, simple to code, and easily extensible. It also makes it extremely easy to add enter and exit actions to each state; all you have to do is create `Enter` and `Exit` methods and adjust the agent's `ChangeState` method accordingly. You'll see the code that does exactly this very shortly.

The West World Project

As a practical example of how to create agents that utilize finite state machines, we are going to look at a game environment where agents inhabit an Old West-style gold mining town named West World. Initially there will only be one inhabitant — a gold miner named Miner Bob — but later in the chapter his wife will also make an appearance. You will have to imagine the tumbleweeds, creakin' mine props, and desert dust blowin' in your eyes because West World is implemented as a simple text-based console application. Any state changes or output from state actions will be sent as text to the console window. I'm using this plaintext-only approach as it

demonstrates clearly the mechanism of a finite state machine without adding the code clutter of a more complex environment.

There are four locations in West World: a *gold mine*, a *bank* where Bob can deposit any nuggets he finds, a *saloon* in which he can quench his thirst, and *home-sweet-home* where he can sleep the fatigue of the day away. Exactly where he goes, and what he does when he gets there, is determined by Bob's current state. He will change states depending on variables like thirst, fatigue, and how much gold he has found hacking away down in the gold mine.

Before we delve into the source code, check out the following sample output from the WestWorld1 executable.

```
Miner Bob: Pickin' up a nugget
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Goin' to the bank. Yes siree
Miner Bob: Depositin' gold. Total savings now: 3
Miner Bob: Leavin' the bank
Miner Bob: Walkin' to the gold mine
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Boy, ah sure is thusty! Walkin' to the saloon
Miner Bob: That's mighty fine sippin liquor
Miner Bob: Leavin' the saloon, feelin' good
Miner Bob: Walkin' to the gold mine
Miner Bob: Pickin' up a nugget
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Goin' to the bank. Yes siree
Miner Bob: Depositin' gold. Total savings now: 4
Miner Bob: Leavin' the bank
Miner Bob: Walkin' to the gold mine
Miner Bob: Pickin' up a nugget
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Boy, ah sure is thusty! Walkin' to the saloon
Miner Bob: That's mighty fine sippin' liquor
Miner Bob: Leavin' the saloon, feelin' good
Miner Bob: Walkin' to the gold mine
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Goin' to the bank. Yes siree
Miner Bob: Depositin' gold. Total savings now: 5
Miner Bob: Woohoo! Rich enough for now. Back home to mah li'l lady
Miner Bob: Leavin' the bank
Miner Bob: Walkin' home
Miner Bob: ZZZZ...
Miner Bob: ZZZZ...
Miner Bob: ZZZZ...
Miner Bob: ZZZZ...
Miner Bob: What a God-darn fantastic nap! Time to find more gold
```

In the output from the program, each time you see Miner Bob change location he is changing state. All the other events are the actions that take place within the states. We'll examine each of Miner Bob's potential states in just a moment, but for now, let me explain a little about the code structure of the demo.

The BaseGameEntity Class

All inhabitants of West World are derived from the base class `BaseGameEntity`. This is a simple class with a private member for storing an ID number. It also specifies a pure virtual member function, `Update`, that must be implemented by all subclasses. `Update` is a function that gets called every update step and will be used by subclasses to update their state machine along with any other data that must be updated each time step.

The `BaseGameEntity` class declaration looks like this:

```
class BaseGameEntity
{
private:

  //every entity has a unique identifying number
  int         m_ID;

  //this is the next valid ID. Each time a BaseGameEntity is instantiated
  //this value is updated
  static int  m_iNextValidID;

  //this is called within the constructor to make sure the ID is set
  //correctly. It verifies that the value passed to the method is greater
  //or equal to the next valid ID, before setting the ID and incrementing
  //the next valid ID
  void SetID(int val);

public:

  BaseGameEntity(int id)
  {
    SetID(id);
  }

  virtual ~BaseGameEntity(){}

  //all entities must implement an update function
  virtual void  Update()=0;

  int           ID()const{return m_ID;}
};
```

For reasons that will become obvious later in the chapter, it's very important for each entity in your game to have a unique identifier. Therefore, on instantiation, the ID passed to the constructor is tested in the `SetID` method to make sure it's unique. If it is not, the program will exit with an assertion failure. In the example given in this chapter, the entities will use an enumerated value as their unique identifier. These can be found in the file EntityNames.h as `ent_Miner_Bob` and `ent_Elsa`.

The Miner Class

The `Miner` class is derived from the `BaseGameEntity` class and contains data members representing the various attributes a `Miner` possesses, such as its health, its level of fatigue, its position, and so forth. Like the troll example shown earlier in the chapter, a `Miner` owns a pointer to an instance of a `State` class in addition to a method for changing what `State` that pointer points to.

```
class Miner : public BaseGameEntity
{
private:

  //a pointer to an instance of a State
  State*               m_pCurrentState;

  // the place where the miner is currently situated
  location_type        m_Location;

  //how many nuggets the miner has in his pockets
  int                  m_iGoldCarried;

  //how much money the miner has deposited in the bank
  int                  m_iMoneyInBank;

  //the higher the value, the thirstier the miner
  int                  m_iThirst;

  //the higher the value, the more tired the miner
  int                  m_iFatigue;

public:

  Miner(int ID);

  //this must be implemented
  void Update();

  //this method changes the current state to the new state
  void ChangeState(State* pNewState);

  /* bulk of interface omitted */
};
```

The `Miner::Update` method is straightforward; it simply increments the `m_iThirst` value before calling the `Execute` method of the current state. It looks like this:

```
void Miner::Update()
{
  m_iThirst += 1;

  if (m_pCurrentState)
  {
    m_pCurrentState->Execute(this);
  }
}
```

Now that you've seen how the Miner class operates, let's take a look at each of the states a miner can find itself in.

The Miner States

The gold miner will be able to enter one of four states. Here are the names of those states followed by a description of the actions and state transitions that occur within those states:

- **EnterMineAndDigForNugget**: If the miner is not located at the gold mine, he changes location. If already at the gold mine, he digs for nuggets of gold. When his pockets are full, Bob changes state to **VisitBankAndDepositGold**, and if while digging he finds himself thirsty, he will stop and change state to **QuenchThirst**.
- **VisitBankAndDepositGold**: In this state the miner will walk to the bank and deposit any nuggets he is carrying. If he then considers himself wealthy enough, he will change state to **GoHomeAndSleepTilRested**. Otherwise he will change state to **EnterMineAndDigForNugget**.
- **GoHomeAndSleepTilRested**: In this state the miner will return to his shack and sleep until his fatigue level drops below an acceptable level. He will then change state to **EnterMineAndDigForNugget**.
- **QuenchThirst**: If at any time the miner feels thirsty (diggin' for gold is thusty work, don't ya know), he changes to this state and visits the saloon in order to buy a whiskey. When his thirst is quenched, he changes state to **EnterMineAndDigForNugget**.

Sometimes it's hard to follow the flow of the state logic from reading a text description like this, so it's often helpful to pick up pen and paper and draw a *state transition diagram* for your game agents. Figure 2.2 shows the state transition diagram for the gold miner. The bubbles represent the individual states and the lines between them the available transitions.

A diagram like this is better on the eyes and can make it much easier to spot any errors in the logic flow.

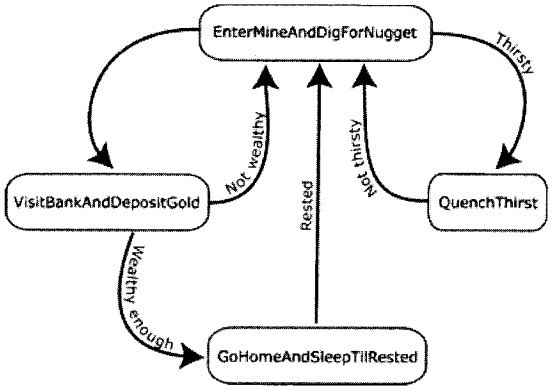

Figure 2.2. Miner Bob's state transition diagram

The State Design Pattern Revisited

You saw a brief description of this design pattern a few pages back, but it won't hurt to recap. Each of a game agent's states is implemented as a unique class and each agent holds a pointer to an instance of its current state. An agent also implements a ChangeState member function that can be called to facilitate the switching of states whenever a state transition is required. The logic for determining any state transitions is contained within each State class. All state classes are derived from an abstract base class, thereby defining a common interface. So far so good. You know this much already.

Earlier in the chapter it was mentioned that it's usually favorable for each state to have associated *enter* and *exit* actions. This permits the programmer to write logic that is only executed once at state entry or exit and increases the flexibility of an FSM a great deal. With these features in mind, let's take a look at an enhanced State base class.

```
class State
{
public:

  virtual ~State(){}

  //this will execute when the state is entered
  virtual void Enter(Miner*)=0;

  //this is called by the miner's update function each update step
  virtual void Execute(Miner*)=0;

  //this will execute when the state is exited
  virtual void Exit(Miner*)=0;
}
```

These additional methods are only called when a Miner changes state. When a state transition occurs, the Miner::ChangeState method first calls the Exit method of the current state, then it assigns the new state to the current state, and finishes by calling the Enter method of the new state (which is now the current state). I think code is clearer than words in this instance, so here's the listing for the ChangeState method:

```
void Miner::ChangeState(State* pNewState)
{
  //make sure both states are valid before attempting to
  //call their methods
  assert (m_pCurrentState && pNewState);

  //call the exit method of the existing state
  m_pCurrentState->Exit(this);

  //change state to the new state
  m_pCurrentState = pNewState;

  //call the entry method of the new state
  m_pCurrentState->Enter(this);
}
```

Notice how a Miner passes the this pointer to each state, enabling the state to use the Miner interface to access any relevant data.

TIP The state design pattern is also useful for structuring the main components of your game flow. For example, you could have a menu state, a save state, a paused state, an options state, a run state, etc.

Each of the four possible states a Miner may access are derived from the State class, giving us these concrete classes: EnterMineAndDigForNugget, VisitBankAndDepositGold, GoHomeAndSleepTilRested, and QuenchThirst. The Miner::m_pCurrentState pointer is able to point to any of these states. When the Update method of Miner is called, it in turn calls the Execute method of the currently active state with the this pointer as a parameter. These class relationships may be easier to understand if you examine the simplified UML class diagram shown in Figure 2.3.

Each concrete state is implemented as a singleton object. This is to ensure that there is only one instance of each state, which agents share (those of you unsure of what a singleton is, please read the sidebar on page 58). Using singletons makes the design more efficient because they remove the need to allocate and deallocate memory every time a state change is made. This is particularly important if you have many agents sharing a complex FSM and/or you are developing for a machine with limited resources.

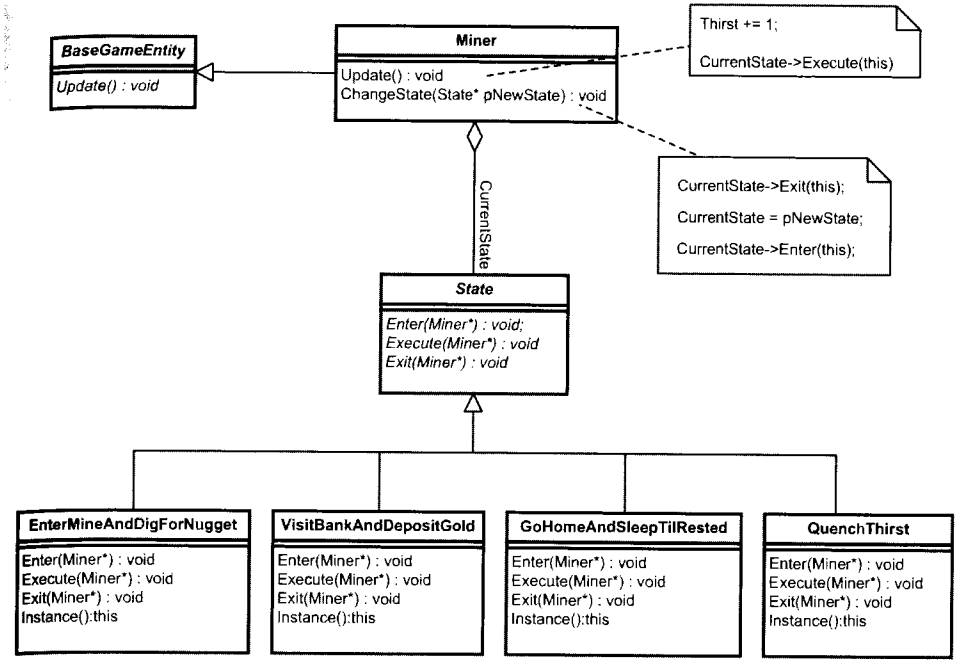

Figure 2.3. UML class diagram for Miner Bob's state machine implementation

NOTE I prefer to use singletons for the states for the reasons I've already given, but there is one drawback. Because they are shared between clients, singleton states are unable to make use of their own local, agent-specific data. For instance, if an agent uses a state that when entered should move it to an arbitrary position, the position cannot be stored in the state itself (because the position may be different for each agent that is using the state). Instead, it would have to be stored somewhere externally and be accessed by the state via the agent's interface. This is not really a problem if your states are accessing only one or two pieces of data, but if you find that the states you have designed are repeatedly accessing lots of external data, it's probably worth considering disposing of the singleton design and writing a few lines of code to manage the allocation and deallocation of state memory.

The Singleton Design Pattern

Often it's useful to guarantee that an object is only instantiated once and/or that it is globally accessible. For example, in game designs that have environments consisting of many different entity types — players, monsters, projectiles, plant pots, etc. — there is usually a "manager" object that handles the creation, deletion, and management of such objects. It is only necessary to have one instance of this object — a singleton — and it is convenient to make it globally accessible because many other objects will require access to it.

The singleton pattern ensures both these qualities. There are many ways of implementing a singleton (do a search at google.com and you'll see what I mean). I prefer to use a static method, `Instance`, that returns a pointer to a static instance of the class. Here's an example:

```
/* ------------------ MyClass.h -------------------- */
#ifndef MY_SINGLETON
#define MY_SINGLETON

class MyClass
{
private:

  // member data
  int m_iNum;

  //constructor is private
  MyClass(){}

  //copy ctor and assignment should be private
  MyClass(const MyClass &);
  MyClass& operator=(const MyClass &);

public:

  //strictly speaking, the destructor of a singleton should be private but some
  //compilers have problems with this so I've left them as public in all the
  //examples in this book
  ~MyClass();

  //methods
  int GetVal()const{return m_iNum;}

  static MyClass* Instance();
};

#endif

/* ------------------- MyClass.cpp ------------------- */

//this must reside in the cpp file; otherwise, an instance will be created
//for every file in which the header is included
MyClass* MyClass::Instance()
{
  static MyClass instance;

  return &instance;
}
```

Member variables and methods can now be accessed via the `Instance` method like so:

```
int num = MyClass::Instance()->GetVal();
```

Because I'm lazy and don't like writing out all that syntax each time I want to access a singleton, I usually `#define` something like this:

```
#define MyCls MyClass::Instance()
```

Using this new syntax I can simply write:

```
int num = MyCls->GetVal();
```

Much easier, don't you think?

> **NOTE** If singletons are a new concept to you, and you decide to search the Internet for further information, you will discover they fuel many a good argument about the design of object-oriented software. Oh yes, programmers love to argue about this stuff, and nothing stokes a dispute better than the discussion of global variables or *objects that masquerade as globals*, such as singletons. My own stance on the matter is to use them wherever I think they provide a convenience and, in my opinion, do not compromise the design. I recommend you read the arguments for and against though, and come to your own conclusions. A good starting place is here:
>
> http://c2.com/cgi/wiki?SingletonPattern

Okay, let's see how everything fits together by examining the complete code for one of the miner states.

The EnterMineAndDigForNugget State

In this state the miner should change location to be at the gold mine. Once at the gold mine he should dig for gold until his pockets are full, when he should change state to ***VisitBankAndDepositNugget****. If the miner gets thirsty while digging he should change state to* ***QuenchThirst****.*

Because concrete states simply implement the interface defined in the virtual base class `State`, their declarations are very straightforward:

```
class EnterMineAndDigForNugget : public State
{
private:

  EnterMineAndDigForNugget(){}

  /* copy ctor and assignment op omitted */

public:

  //this is a singleton
  static EnterMineAndDigForNugget* Instance();

  virtual void Enter(Miner* pMiner);

  virtual void Execute(Miner* pMiner);

  virtual void Exit(Miner* pMiner);
};
```

As you can see, it's just a formality. Let's take a look at each of the methods in turn.

EnterMineAndDigForNugget::Enter

The code for the `Enter` method of `EnterMineAndDigForNugget` is as follows:

```
void EnterMineAndDigForNugget::Enter(Miner* pMiner)
{
  //if the miner is not already located at the gold mine, he must
  //change location to the gold mine
  if (pMiner->Location() != goldmine)
  {
    cout << "\n" << GetNameOfEntity(pMiner->ID()) << ": "
         << "Walkin' to the gold mine";

    pMiner->ChangeLocation(goldmine);
  }
}
```

This method is called when a miner first enters the **EnterMineAndDigForNugget** state. It ensures that the gold miner is located at the gold mine.

An agent stores its location as an enumerated type and the `ChangeLocation` method changes this value to switch locations.

EnterMineAndDigForNugget::Execute

The `Execute` method is a little more complicated and contains logic that can change a miner's state. (Don't forget that `Execute` is the method called each update step from `Miner::Update`.)

```
void EnterMineAndDigForNugget::Execute(Miner* pMiner)
{
  //the miner digs for gold until he is carrying in excess of MaxNuggets.
  //If he gets thirsty during his digging he stops work and
  //changes state to go to the saloon for a whiskey.
  pMiner->AddToGoldCarried(1);

  //diggin' is hard work
  pMiner->IncreaseFatigue();

  cout << "\n" << GetNameOfEntity(pMiner->ID()) << ": "
       << "Pickin' up a nugget";

  //if enough gold mined, go and put it in the bank
  if (pMiner->PocketsFull())
  {
    pMiner->ChangeState(VisitBankAndDepositGold::Instance());
  }

  //if thirsty go and get a whiskey
  if (pMiner->Thirsty())
  {
    pMiner->ChangeState(QuenchThirst::Instance());
  }
}
```

Note here how the `Miner::ChangeState` method is called using `QuenchThirst`'s or `VisitBankAndDepositGold`'s `Instance` member, which provides a pointer to the unique instance of that class.

EnterMineAndDigForNugget::Exit

The `Exit` method of `EnterMineAndDigForNugget` outputs a message telling us that the gold miner is leaving the mine.

```
void EnterMineAndDigForNugget::Exit(Miner* pMiner)
{
  cout << "\n" << GetNameOfEntity(pMiner->ID()) << ": "
       << "Ah'm leavin' the gold mine with mah pockets full o' sweet gold";
}
```

I hope an examination of the preceding three methods helps clear up any confusion you may have been experiencing and that you can now see how each state is able to modify the behavior of an agent or effect a transition into another state. You may find it useful at this stage to load up the WestWorld1 project into your IDE and scan the code. In particular, check

out all the states in MinerOwnedStates.cpp and examine the Miner class to familiarize yourself with its member variables. Above all else, make sure you understand how the state design pattern works before you read any further. If you are a little unsure, please take the time to go over the previous few pages until you feel comfortable with the concept.

You have seen how the use of the state design pattern provides a very flexible mechanism for state-driven agents. It's extremely easy to add additional states as and when required. Indeed, should you so wish, you can switch an agent's entire state architecture for an alternative one. This can be useful if you have a very complicated design that would be better organized as a collection of several separate smaller state machines. For example, the state machine for a first-person shooter (FPS) like Unreal 2 tends to be large and complex. When designing the AI for a game of this sort you may find it preferable to think in terms of several smaller state machines representing functionality like "defend the flag" or "explore map," which can be switched in and out when appropriate. The state design pattern makes this easy to do.

Making the State Base Class Reusable

As the design stands, it's necessary to create a separate State base class for each character type to derive its states from. Instead, let's make it reusable by turning it into a class template.

```
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(){}
};
```

The declaration for a concrete state — using the **EnterMineAndDigForNugget** miner state as an example — now looks like this:

```
class EnterMineAndDigForNugget : public State<Miner>
{

public:

  /* OMITTED */
};
```

This, as you will see shortly, makes life easier in the long run.

Global States and State Blips

More often than not, when designing finite state machines you will end up with code that is duplicated in every state. For example, in the popular game The Sims by Maxis, a Sim may feel the urge of nature come upon it and have to visit the bathroom to relieve itself. This urge may occur in any state the Sim may be in and at any time. Given the current design, to bestow the gold miner with this type of behavior, duplicate conditional logic would have to be added to every one of his states, or alternatively, placed into the Miner::Update function. While the latter solution is acceptable, it's better to create a *global state* that is called every time the FSM is updated. That way, all the logic for the FSM is contained within the states and not in the agent class that owns the FSM.

To implement a global state, an additional member variable is required:

```
//notice how now that State is a class template we have to declare the entity type
State<Miner>* m_pGlobalState;
```

In addition to global behavior, occasionally it will be convenient for an agent to enter a state with the condition that when the state is exited, the agent returns to its previous state. I call this behavior a *state blip*. For example, just as in The Sims, you may insist that your agent can visit the bathroom at any time, yet make sure it always returns to its prior state. To give an FSM this type of functionality it must keep a record of the previous state so the state blip can revert to it. This is easy to do as all that is required is another member variable and some additional logic in the Miner::ChangeState method.

By now though, to implement these additions, the Miner class has acquired two extra member variables and one additional method. It has ended up looking something like this (extraneous detail omitted):

```
class Miner : public BaseGameEntity
{
private:

  State<Miner>*   m_pCurrentState;
  State<Miner>*   m_pPreviousState;
  State<Miner>*   m_pGlobalState;
  ...

public:

  void ChangeState(State<Miner>* pNewState);
  void RevertToPreviousState();
  ...
};
```

Hmm, looks like it's time to tidy up a little.

Creating a State Machine Class

The design can be made a lot cleaner by encapsulating all the state related data and methods into a state machine class. This way an agent can own an instance of a state machine and delegate the management of current states, global states, and previous states to it.

With this in mind take a look at the following StateMachine class template.

```
template <class entity_type>
class StateMachine
{
private:

  //a pointer to the agent that owns this instance
  entity_type*          m_pOwner;

  State<entity_type>*   m_pCurrentState;

  //a record of the last state the agent was in
  State<entity_type>*   m_pPreviousState;

  //this state logic is called every time the FSM is updated
  State<entity_type>*   m_pGlobalState;

public:

  StateMachine(entity_type* owner):m_pOwner(owner),
                                   m_pCurrentState(NULL),
                                   m_pPreviousState(NULL),
                                   m_pGlobalState(NULL)
  {}

  //use these methods to initialize the FSM
  void SetCurrentState(State<entity_type>* s){m_pCurrentState = s;}
  void SetGlobalState(State<entity_type>* s) {m_pGlobalState = s;}
  void SetPreviousState(State<entity_type>* s){m_pPreviousState = s;}

  //call this to update the FSM
  void  Update()const
  {
    //if a global state exists, call its execute method
    if (m_pGlobalState)   m_pGlobalState->Execute(m_pOwner);

    //same for the current state
    if (m_pCurrentState) m_pCurrentState->Execute(m_pOwner);
  }

  //change to a new state
  void  ChangeState(State<entity_type>* pNewState)
  {
    assert(pNewState &&
           "<StateMachine::ChangeState>: trying to change to a null state");
```



```
    //keep a record of the previous state
    m_pPreviousState = m_pCurrentState;

    //call the exit method of the existing state
    m_pCurrentState->Exit(m_pOwner);

    //change state to the new state
    m_pCurrentState = pNewState;

    //call the entry method of the new state
    m_pCurrentState->Enter(m_pOwner);
  }

  //change state back to the previous state
  void  RevertToPreviousState()
  {
    ChangeState(m_pPreviousState);
  }

  //accessors
  State<entity_type>*  CurrentState()  const{return m_pCurrentState;}
  State<entity_type>*  GlobalState()   const{return m_pGlobalState;}
  State<entity_type>*  PreviousState() const{return m_pPreviousState;}

  //returns true if the current state's type is equal to the type of the
  //class passed as a parameter.
  bool  isInState(const State<entity_type>& st)const;
};
```

Now all an agent has to do is to own an instance of a StateMachine and implement a method to update the state machine to get full FSM functionality.

The improved Miner class now looks like this:

```
class Miner : public BaseGameEntity
{
private:

  //an instance of the state machine class
  StateMachine<Miner>*  m_pStateMachine;

  /* EXTRANEOUS DETAIL OMITTED */

public:

  Miner(int id):m_Location(shack),
                m_iGoldCarried(0),
                m_iMoneyInBank(0),
                m_iThirst(0),
                m_iFatigue(0),
                BaseGameEntity(id)

  {
    //set up state machine
    m_pStateMachine = new StateMachine<Miner>(this);

    m_pStateMachine->SetCurrentState(GoHomeAndSleepTilRested::Instance());
    m_pStateMachine->SetGlobalState(MinerGlobalState::Instance());
  }

  ~Miner(){delete m_pStateMachine;}

  void Update()
  {
    ++m_iThirst;
    m_pStateMachine->Update();
  }

  StateMachine<Miner>* GetFSM()const{return m_pStateMachine;}

  /* EXTRANEOUS DETAIL OMITTED */
};
```

Notice how the current and global states must be set explicitly when a StateMachine is instantiated.

The class hierarchy is now like that shown in Figure 2.4.

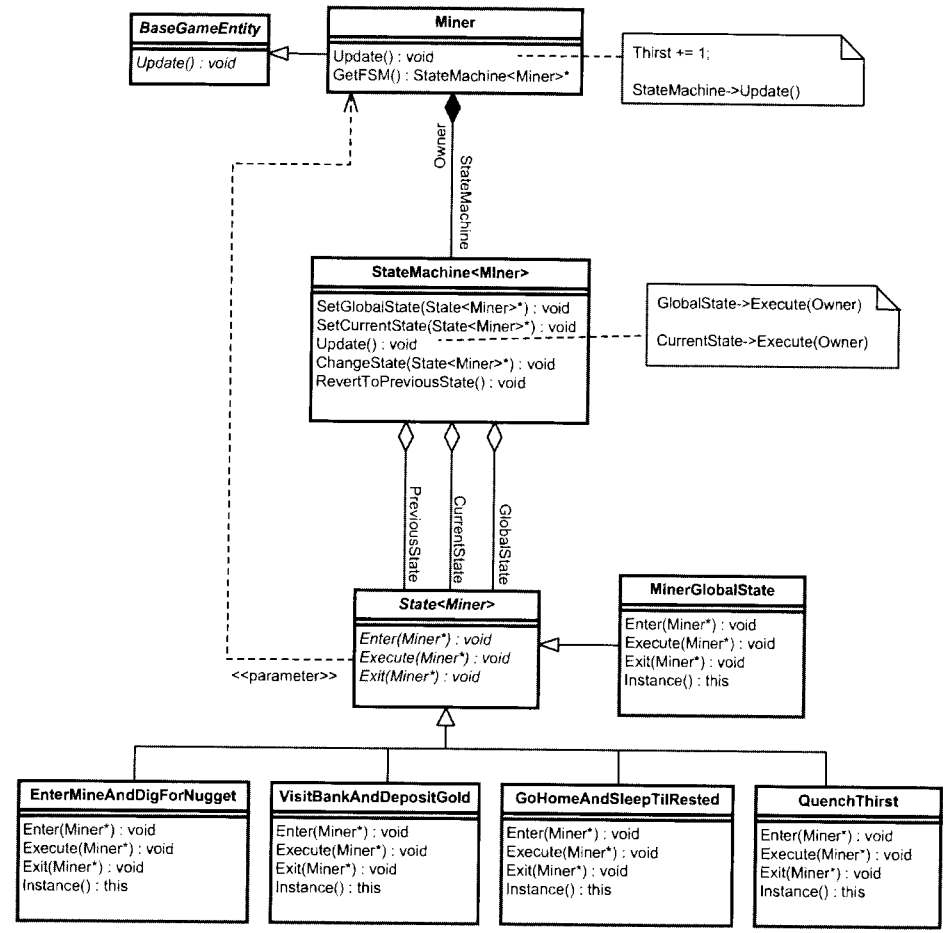

Figure 2.4. The updated design

Introducing Elsa

To demonstrate these improvements, I've created the second project for this chapter: WestWorldWithWoman. In this project, West World has gained another inhabitant, Elsa, the gold miner's wife. Elsa doesn't do much just yet; she's mainly preoccupied with cleaning the shack and emptying her bladder (she drinks way too much cawfee). The state transition diagram for Elsa is shown in Figure 2.5.

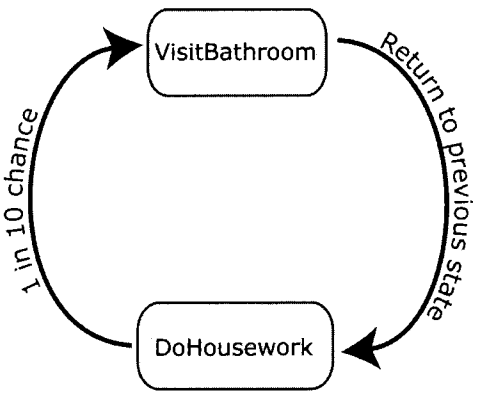

Figure 2.5. Elsa's state transition diagram. The global state is not shown in the figure because its logic is effectively implemented in any state and never changed.

When you boot up the project into your IDE, notice how the `VisitBathroom` state is implemented as a blip state (i.e., it always reverts back to the previous state). Also note that a global state has been defined, `WifesGlobalState`, which contains the logic required for Elsa's bathroom visits. This logic is contained in a global state because Elsa may feel the call of nature during any state and at any time.

Here is a sample of the output from WestWorldWithWoman. Elsa's actions are shown italicized.

```
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Goin' to the bank. Yes siree
Elsa: Walkin' to the can. Need to powda mah pretty li'l nose
Elsa: Ahhhhhh! Sweet relief!
Elsa: Leavin' the john
Miner Bob: Depositin' gold. Total savings now: 4
Miner Bob: Leavin' the bank
Miner Bob: Walkin' to the gold mine
Elsa: Walkin' to the can. Need to powda mah pretty li'l nose
Elsa: Ahhhhhh! Sweet relief!
Elsa: Leavin' the john
Miner Bob: Pickin' up a nugget
Elsa: Moppin' the floor
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Boy, ah sure is thusty! Walkin' to the saloon
Elsa: Moppin' the floor
Miner Bob: That's mighty fine sippin' liquor
Miner Bob: Leavin' the saloon, feelin' good
Miner Bob: Walkin' to the gold mine
Elsa: Makin' the bed
Miner Bob: Pickin' up a nugget
Miner Bob: Ah'm leavin' the gold mine with mah pockets full o' sweet gold
Miner Bob: Goin' to the bank. Yes siree
```



```
Elsa: Walkin' to the can. Need to powda mah pretty li'l nose
Elsa: Ahhhhhh! Sweet relief!
Elsa: Leavin' the john
Miner Bob: Depositin' gold. Total savings now: 5
Miner Bob: Woohoo! Rich enough for now. Back home to mah li'l lady
Miner Bob: Leavin' the bank
Miner Bob: Walkin' home
Elsa: Walkin' to the can. Need to powda mah pretty li'l nose
Elsa: Ahhhhhh! Sweet relief!
Elsa: Leavin' the john
Miner Bob: ZZZZ...
```

Adding Messaging Capabilities to Your FSM

Well-designed games tend to be event driven. That is to say, when an event occurs — a weapon is fired, a lever is pulled, an alarm tripped, etc. — the event is broadcast to the relevant objects in the game so that they may respond appropriately. These events are typically sent in the form of a packet of data that contains information about the event such as what sent it, what objects should respond to it, what the actual event is, a time stamp, and so forth.

The reason event-driven architectures are generally preferred is because they are efficient. Without event handling, objects have to continuously poll the game world to see if a particular action has occurred. With event handling, objects can simply get on with their business until an event message is broadcast to them. Then, if that message is pertinent, they can act upon it.

Intelligent game agents can use the same idea to communicate with each other. When endowed with the power to send, handle, and respond to events, it's easy to design behavior like the following:

- **A wizard throws a fireball at an orc**. The wizard sends a message to the orc informing it of its impending doom so it may respond accordingly, i.e., die horribly and in magnificent style.
- **A football player makes a pass to a teammate**. The passer can send a message to the receiver, letting it know where it should move to intercept the ball and at what time it should be at that position.
- **A grunt is injured**. It dispatches a message to each of its comrades requesting help. When one arrives with aid, another message is broadcast to let the others know they can resume their activities.
- **A character strikes a match to help light its way along a gloomy corridor**. A delayed message is dispatched to warn that the match will burn down to his fingers in thirty seconds. If he is still holding the match when he receives the message, he reacts by dropping the match and shouting out in pain.

Good, eh? The remainder of this chapter will demonstrate how agents can be given the ability to handle messages like this. But before we can figure out how to transmit them and handle them, the first thing to do is to define exactly what a message is.

The Telegram Structure

A message is simply an enumerated type. This could be just about anything. You could have agents sending messages like `Msg_ReturnToBase`, `Msg_MoveToPosition`, or `Msg_HelpNeeded`. Additional information also needs to be packaged along with the message. For example, we should record information about who sent it, who the recipient is, what the actual message is, a time stamp, and so forth. To do this, all the relevant information is kept together in a structure called `Telegram`. The code is shown below. Examine each member variable and get a feel for what sort of information the game agents will be passing around.

```
struct Telegram
{
  //the entity that sent this telegram
  int          Sender;

  //the entity that is to receive this telegram
  int          Receiver;

  //the message itself. These are all enumerated in the file
  //"MessageTypes.h"
  int          Msg;

  //messages can be dispatched immediately or delayed for a specified amount
  //of time. If a delay is necessary, this field is stamped with the time
  //the message should be dispatched.
  double       DispatchTime;

  //any additional information that may accompany the message
  void*        ExtraInfo;

  /* CONSTRUCTORS OMITTED */
};
```

The `Telegram` structure should be reusable, but because it's impossible to know in advance what sort of additional information future game designs will need to pass in a message, a void pointer `ExtraInfo` is provided. This can be used to pass any amount of additional information between characters. For example, if a platoon leader sends the message `Msg_MoveToPosition` to all his men, `ExtraInfo` can be used to store the coordinates of that position.

Miner Bob and Elsa Communicate

For the purposes of this chapter, I've kept the communication between Miner Bob and Elsa simple. They only have two messages they can use, and they are enumerated as:

```
enum message_type
{
  Msg_HiHoneyImHome,
  Msg_StewReady
};
```

The gold miner will send `Msg_HiHoneyImHome` to his wife to let her know he's back at the shack. `Msg_StewReady` is utilized by the wife to let herself know when to take dinner out of the oven and for her to communicate to Miner Bob that food is on the table.

The new state transition diagram for Elsa is shown in Figure 2.6.

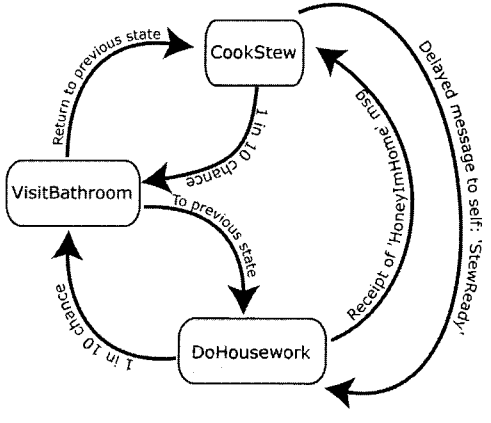

Figure 2.6. Elsa's new state transition diagram

Before I show you how telegram events are handled by an agent, let me demonstrate how they are created, managed, and dispatched.

Message Dispatch and Management

The creation, dispatch, and management of telegrams is handled by a class named `MessageDispatcher`. Whenever an agent needs to send a message, it calls `MessageDispatcher::DispatchMessage` with all the necessary information, such as the message type, the time the message is to be dispatched, the ID of the recipient, and so on. The `MessageDispatcher` uses this information to create a `Telegram`, which it either dispatches immediately or stores in a queue ready to be dispatched at the correct time.

Before it can dispatch a message, the MessageDispatcher must obtain a pointer to the entity specified by the sender. Therefore, there must be some sort of database of instantiated entities provided for the MessageDispatcher to refer to — a sort of telephone book where pointers to agents are cross-referenced by their ID. The database used for the demo is a singleton class called EntityManager. Its declaration looks like this:

```
class EntityManager
{
private:

  //to save the ol' fingers
  typedef std::map<int, BaseGameEntity*> EntityMap;

private:

  //to facilitate quick lookup the entities are stored in a std::map, in
  //which pointers to entities are cross-referenced by their identifying
  //number
  EntityMap          m_EntityMap;

  EntityManager(){}

  //copy ctor and assignment should be private
  EntityManager(const EntityManager&);
  EntityManager& operator=(const EntityManager&);

public:

  static EntityManager* Instance();

  //this method stores a pointer to the entity in the std::vector
  //m_Entities at the index position indicated by the entity's ID
  //(makes for faster access)
  void           RegisterEntity(BaseGameEntity* NewEntity);

  //returns a pointer to the entity with the ID given as a parameter
  BaseGameEntity* GetEntityFromID(int id)const;

  //this method removes the entity from the list
  void           RemoveEntity(BaseGameEntity* pEntity);
};

//provide easy access to the instance of the EntityManager
#define EntityMgr EntityManager::Instance()
```

When an entity is created it is registered with the entity manager like so:

```
Miner* Bob = new Miner(ent_Miner_Bob); //enumerated ID
EntityMgr->RegisterEntity(Bob);
```

A client can now request a pointer to a specific entity by passing its ID to the method EntityManager::GetEntityFromID in this way:

```
Entity* pBob = EntityMgr->GetEntityFromID(ent_Miner_Bob);
```


The client can then use this pointer to call the message handler for that particular entity. More on this in a moment, but first let's look at the way messages are created and routed between entities.

The MessageDispatcher Class

The class that manages the dispatch of messages is a singleton named MessageDispatcher. Take a look at the declaration of this class:

```
class MessageDispatcher
{
private:

  //a std::set is used as the container for the delayed messages
  //because of the benefit of automatic sorting and avoidance
  //of duplicates. Messages are sorted by their dispatch time.
  std::set<Telegram> PriorityQ;

  //this method is utilized by DispatchMessage or DispatchDelayedMessages.
  //This method calls the message handling member function of the receiving
  //entity, pReceiver, with the newly created telegram
  void Discharge(Entity* pReceiver, const Telegram& msg);

  MessageDispatcher(){}

public:

  //this class is a singleton
  static MessageDispatcher* Instance();

  //send a message to another agent.
  void DispatchMessage(double      delay,
                       int         sender,
                       int         receiver,
                       int         msg,
                       void*       ExtraInfo);

  //send out any delayed messages. This method is called each time through
  // the main game loop.
  void DispatchDelayedMessages();
};

//to make life easier...
#define Dispatch MessageDispatcher::Instance()
```

The MessageDispatcher class handles messages to be dispatched immediately and time stamped messages, which are messages to be delivered at a specified time in the future. Both these types of messages are created and managed by the same method: DispatchMessage. Let's go through the source. (In the companion file this method has some additional lines of code for outputting some informative text to the console. I've omitted them here for clarity.)

```
void MessageDispatcher::DispatchMessage(double       delay,
                                        int          sender,
                                        int          receiver,
                                        int          msg,
                                        void*        ExtraInfo)
{
```

This method is called when an entity sends a message to another entity. The message sender must provide as parameters the details required to create a `Telegram` structure. In addition to the sender's ID, the receiver's ID, and the message itself, this function must be given a time delay and a pointer to any additional info, if any. If the message is to be sent immediately, the method should be called with a zero or negative delay.

```
  //get a pointer to the receiver of the message
  Entity* pReceiver = EntityMgr->GetEntityFromID(receiver);

  //create the telegram
  Telegram telegram(delay, sender, receiver, msg, ExtraInfo);

  //if there is no delay, route the telegram immediately
  if (delay <= 0.0)
  {
    //send the telegram to the recipient
    Discharge(pReceiver, telegram);
  }
```

After a pointer to the recipient is obtained via the entity manager and a `Telegram` is created using the appropriate information, the message is ready to be dispatched. If the message is for immediate dispatch, the `Discharge` method is called straight away. The `Discharge` method passes the newly created `Telegram` to the message handling method of the receiving entity (more on this shortly). Most of the messages your agents will be sending will be created and immediately dispatched in this way. For example, if a troll hits a human over the head with a club, it could send an instant message to the human telling it that it had been hit. The human would then respond using the appropriate action, sound, and animation.

```
  //else calculate the time when the telegram should be dispatched
  else
  {
    double CurrentTime = Clock->GetCurrentTime();

    telegram.DispatchTime = CurrentTime + delay;

    //and put it in the queue
    PriorityQ.insert(telegram);
  }
}
```

If the message is to be dispatched at some time in the future, then these few lines of code calculate the time it should be delivered before inserting the new telegram into a priority queue — a data structure that keeps its

elements sorted in order of precedence. I have utilized a `std::set` as the priority queue in this example because it automatically discards duplicate telegrams.

Telegrams are sorted with respect to their time stamp and to this effect, if you take a look at Telegram.h, you will find that the < and == operators have been overloaded. Also note how telegrams with time stamps less than a quarter of a second apart are to be considered identical. This prevents many similar telegrams bunching up in the queue and being delivered en masse, thereby flooding an agent with identical messages. Of course, this delay will vary according to your game. Games with lots of action producing a high frequency of messages will probably require a smaller gap.

The queued telegrams are examined each update step by the method `DispatchDelayedMessages`. This function checks the front of the priority queue to see if any telegrams have expired time stamps. If so, they are dispatched to their recipient and removed from the queue. The code for this method looks like this:

```
void MessageDispatcher::DispatchDelayedMessages()
{
  //first get current time
  double CurrentTime = Clock->GetCurrentTime();

  //now peek at the queue to see if any telegrams need dispatching.
  //remove all telegrams from the front of the queue that have gone
  //past their sell-by date
  while( (PriorityQ.begin()->DispatchTime < CurrentTime) &&
         (PriorityQ.begin()->DispatchTime > 0) )
  {
    //read the telegram from the front of the queue
    Telegram telegram = *PriorityQ.begin();

    //find the recipient
    Entity* pReceiver = EntityMgr->GetEntityFromID(telegram.Receiver);

    //send the telegram to the recipient
    Discharge(pReceiver, telegram);

    //and remove it from the queue
    PriorityQ.erase(PriorityQ.begin());
  }
}
```

A call to this method must be placed in the game's main update loop to facilitate the correct and timely dispatch of any delayed messages.

Message Handling

Once a system for creating and dispatching messages is in place, the handling of them is relatively easy. The `BaseGameEntity` class must be modified so any subclass can receive messages. This is achieved by declaring another pure virtual function, `HandleMessage`, which all derived classes

must implement. The revised BaseGameEntity base class now looks like this:

```
class BaseGameEntity
{
private:

  int         m_ID;

  /* EXTRANEOUS DETAIL REMOVED FOR CLARITY*/

public:

  //all subclasses can communicate using messages.
  virtual bool  HandleMessage(const Telegram& msg)=0;

 /* EXTRANEOUS DETAIL REMOVED FOR CLARITY*/
};
```

In addition, the State base class must also be modified so that a BaseGameEntity's states can choose to accept and handle messages. The revised State class includes an additional OnMessage method as follows:

```
template <class entity_type>
class State
{
public:

  //this executes if the agent receives a message from the
  //message dispatcher
  virtual bool OnMessage(entity_type*, const Telegram&)=0;

  /* EXTRANEOUS DETAIL REMOVED FOR CLARITY*/
};
```

Finally, the StateMachine class is modified to contain a HandleMessage method. When a telegram is received by an entity, it is first routed to the entity's current state. If the current state does not have code in place to deal with the message, it's routed to the entity's global state's message handler. You probably noticed that OnMessage returns a bool. This is to indicate whether or not the message has been handled successfully and enables the code to route the message accordingly.

Here is the listing of the StateMachine::HandleMessage method:

```
bool  StateMachine::HandleMessage(const Telegram& msg)const
{
  //first see if the current state is valid and that it can handle
  //the message
  if (m_pCurrentState && m_pCurrentState->OnMessage(m_pOwner, msg))
  {
    return true;
  }

  //if not, and if a global state has been implemented, send
  //the message to the global state
  if (m_pGlobalState && m_pGlobalState->OnMessage(m_pOwner, msg))
  {
    return true;
  }

  return false;
}
```


And here's how the Miner class routes messages sent to it:

```
bool Miner::HandleMessage(const Telegram& msg)
{
  return m_pStateMachine->HandleMessage(msg);
}
```

Figure 2.7 shows the new class architecture.

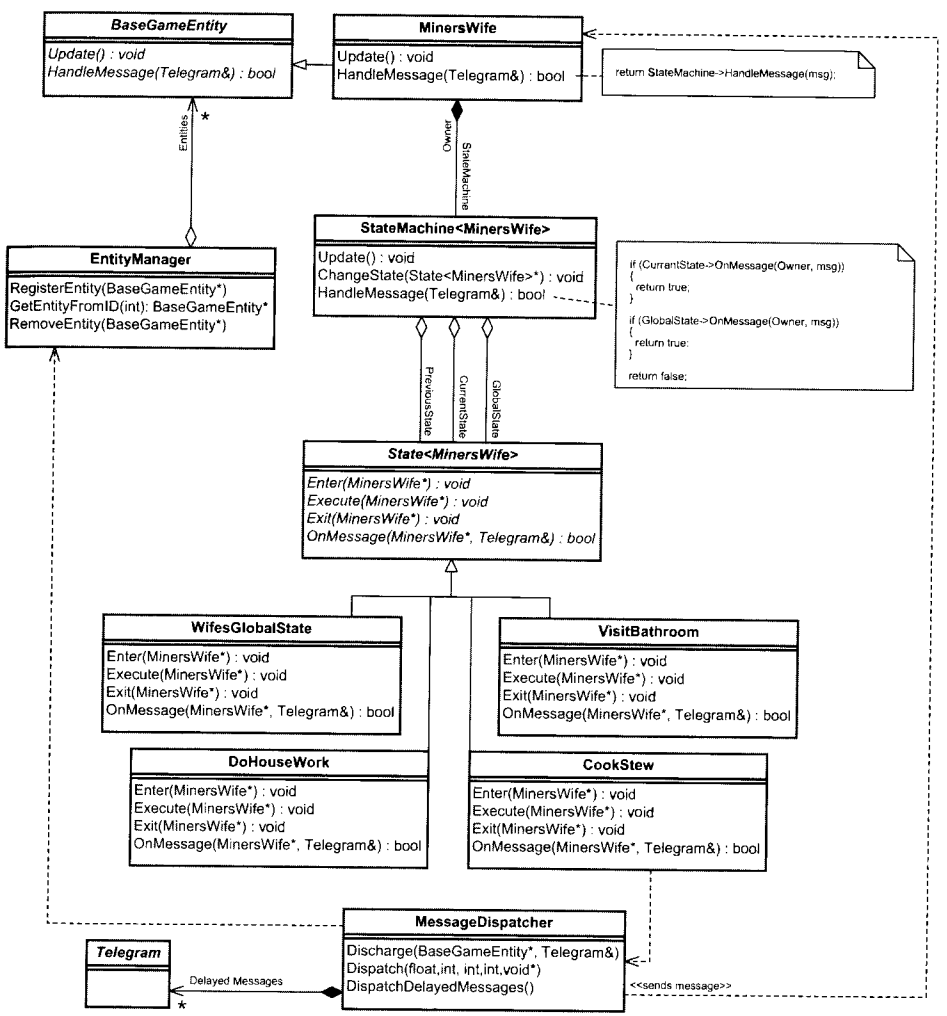

Figure 2.7. The updated design incorporating messaging

Elsa Cooks Dinner

At this point it's probably a good idea to take a look at a concrete example of how messaging works, so let's examine how it can be integrated into the West World project. In the final version of this demo, WestWorldWithMessaging, there is a message sequence that proceeds like this:

1. Miner Bob enters the shack and sends a `Msg_HiHoneyImHome` message to Elsa to let her know he's arrived home.
2. Elsa receives the `Msg_HiHoneyImHome` message, stops what she's currently doing, and changes state to **CookStew**.
3. When Elsa enters the **CookStew** state, she puts the stew in the oven and sends a delayed `Msg_StewReady` message to herself to remind herself that the stew needs to be taken out of the oven at a specific time in the future. (Normally a good stew takes at least an hour to cook, but in cyberspace Elsa can rustle one up in just a fraction of a second!)
4. Elsa receives the `Msg_StewReady` message. She responds to this message by taking the stew out of the oven and dispatching a message to Miner Bob to inform him that dinner is on the table. Miner Bob will only respond to this message if he is in the **GoHomeAndSleepTilRested** state (because in this state he is always located at the shack). If he is anywhere else, such as at the gold mine or the saloon, this message would be dispatched and dismissed.
5. Miner Bob receives the `Msg_StewReady` message and changes state to **EatStew**.

Let me run through the code that executes each of these steps.

Step One

Miner Bob enters the shack and sends a `Msg_HiHoneyImHome` *message to Elsa to let her know he's arrived home.*

Additional code has been added to the `Enter` method of the **GoHomeAndSleepTilRested** state to facilitate sending a message to Elsa. Here is the listing:

```
void GoHomeAndSleepTilRested::Enter(Miner* pMiner)
{
  if (pMiner->Location() != shack)
  {
    cout << "\n" << GetNameOfEntity(pMiner->ID()) << ": "
         << "Walkin' home";

    pMiner->ChangeLocation(shack);

    //let the wife know I'm home
    Dispatch->DispatchMessage(SEND_MSG_IMMEDIATELY,  //time delay
                              pMiner->ID(),          //ID of sender
                              ent_Elsa,              //ID/name of recipient
```



```
                              Msg_HiHoneyImHome,     //the message
                              NO_ADDITIONAL_INFO);   //no extra info attached
  }
}
```

As you can see, when Miner Bob changes to this state the first thing he does is change location. He then dispatches `Msg_HiHoneyImHome` to Elsa by calling the `DispatchMessage` method of the `MessageDispatcher` singleton class. Because the message is to be dispatched immediately, the first parameter of `DispatchMessage` is set to zero. No additional information is attached to the telegram. (The constants `SEND_MSG_IMMEDIATELY` and `NO_ADDITIONAL_INFO` are defined with the value 0 in the file MessageDispatcher.h to aid legibility.)

⌘ TIP You don't have to restrict the messaging system to game characters such as orcs, archers, and wizards. Provided an object is derived from a class that enforces a unique identifier (like `BaseGameEntity`) it's possible to send messages to it. Objects such as treasure chests, traps, magical doors, or even trees are all items that may benefit from the ability to receive and process messages.

For example, you could derive an `OakTree` class from the `BaseGameEntity` class and implement a message handling function to react to messages such as HitWithAxe or StormyWeather. The oak tree can then react to these messages by toppling over or by rustling its leaves and creaking. The possibilities you can construct with this sort of messaging system are almost endless.

Step Two

Elsa receives the `Msg_HiHoneyImHome` *message, stops what she's currently doing, and changes state to* **CookStew**.

Because she never leaves the shack, Elsa should respond to `Msg_HiHoneyImHome` when in any state. The easiest way to implement this is to let her global state take care of this message. (Remember, the global state is executed each update along with the current state.)

```
bool WifesGlobalState::OnMessage(MinersWife* wife, const Telegram& msg)
{
  switch(msg.Msg)
  {
  case Msg_HiHoneyImHome:
   {
     cout << "\nMessage handled by " << GetNameOfEntity(wife->ID())
          << " at time: " << Clock->GetCurrentTime();

     cout << "\n" << GetNameOfEntity(wife->ID()) <<
          ": Hi honey. Let me make you some of mah fine country stew";

     wife->GetFSM()->ChangeState(CookStew::Instance());
   }

   return true;

  }//end switch
```

```
  return false;
}
```

Step Three

When Elsa enters the **CookStew** *state, she puts the stew in the oven and sends a delayed* `Msg_StewReady` *message to herself as a reminder to take the stew out before it burns and upsets Bob.*

This is a demonstration of how delayed messages can be used. In this example, Elsa puts the stew in the oven and then sends a delayed message to herself as a reminder to take the stew out. As we discussed earlier, this message will be stamped with the correct time for dispatch and stored in a priority queue. Each time through the game loop there is a call to `MessageDispatcher::DispatchDelayedMessages`. This method checks to see if any telegrams have exceeded their time stamp and dispatches them to their appropriate recipients where necessary.

```
void CookStew::Enter(MinersWife* wife)
{
  //if not already cooking put the stew in the oven
  if (!wife->Cooking())
  {
    cout << "\n" << GetNameOfEntity(wife->ID())
         << ": Puttin' the stew in the oven";

    //send a delayed message to myself so that I know when to take the stew
    //out of the oven
    Dispatch->DispatchMessage(1.5,                  //time delay
                              wife->ID(),           //sender ID
                              wife->ID(),           //receiver ID
                              Msg_StewReady,        //the message
                              NO_ADDITIONAL_INFO);  //no extra info attached

    wife->SetCooking(true);
  }
}
```

Step Four

Elsa receives the `Msg_StewReady` *message. She responds by taking the stew out of the oven and dispatching a message to Miner Bob to inform him that dinner is on the table. Miner Bob will only respond to this message if he is in the* **GoHomeAndSleepTilRested** *state (to ensure he is located at the shack).*

Because Miner Bob does not have bionic ears, he will only be able to hear Elsa calling him for dinner if he is at home. Therefore, Bob will only respond to this message if he is in the **GoHomeAndSleepTilRested** state.

```
bool CookStew::OnMessage(MinersWife* wife, const Telegram& msg)
{
```



```
  switch(msg.Msg)
  {
    case Msg_StewReady:
    {
      cout << "\nMessage received by " << GetNameOfEntity(wife->ID()) <<
           " at time: " << Clock->GetCurrentTime();
      cout << "\n" << GetNameOfEntity(wife->ID())
           << ": Stew ready! Let's eat";

      //let hubby know the stew is ready
      Dispatch->DispatchMessage(SEND_MSG_IMMEDIATELY,
                                wife->ID(),
                                ent_Miner_Bob,
                                Msg_StewReady,
                                NO_ADDITIONAL_INFO);

      wife->SetCooking(false);

      wife->GetFSM()->ChangeState(DoHouseWork::Instance());
    }

    return true;

  }//end switch

  return false;
}
```

Step Five

Miner Bob receives the `Msg_StewReady` *message and changes state to* **EatStew**.

When Miner Bob receives `Msg_StewReady` he stops whatever he's doing, changes state to **EatStew**, and settles down at the table ready to eat a mighty fine and fillin' bowl of stew.

```
bool GoHomeAndSleepTilRested::OnMessage(Miner* pMiner, const Telegram& msg)
{
  switch(msg.Msg)
  {
  case Msg_StewReady:

    cout << "\nMessage handled by " << GetNameOfEntity(pMiner->ID())
    << " at time: " << Clock->GetCurrentTime();

    cout << "\n" << GetNameOfEntity(pMiner->ID())
         << ": Okay hun, ahm a-comin'!";

    pMiner->GetFSM()->ChangeState(EatStew::Instance());

    return true;

  }//end switch
```

```
    return false; //send message to global message handler
}
```

Here is some example output from the WestWorldWithMessaging program. You can see clearly where the preceding message sequence occurs.

```
Miner Bob: Goin' to the bank. Yes siree
Elsa: Moppin' the floor
Miner Bob: Depositin' gold. Total savings now: 5
Miner Bob: Woohoo! Rich enough for now. Back home to mah li'l lady
Miner Bob: Leavin' the bank
Miner Bob: Walkin' home
Instant telegram dispatched at time: 4.20062 by Miner Bob for Elsa. Msg is
HiHoneyImHome
Message received by Elsa at time: 4.20062
Elsa: Hi honey. Let me make you some of mah fine country stew
Elsa: Puttin' the stew in the oven
Delayed telegram from Elsa recorded at time 4.20062 for Elsa. Msg is StewReady
Elsa: Fussin' over food
Miner Bob: ZZZZ...
Elsa: Fussin' over food
Miner Bob: ZZZZ...
Elsa: Fussin' over food
Miner Bob: ZZZZ...
Elsa: Fussin' over food
Queued telegram ready for dispatch: Sent to Elsa. Msg is StewReady
Message received by Elsa at time: 5.10162
Elsa: Stew ready! Let's eat
Instant telegram dispatched at time: 5.10162 by Elsa for Miner Bob. Msg is
StewReady
Message received by Miner Bob at time: 5.10162
Miner Bob: Okay hun, ahm a-comin'!
Miner Bob: Smells reaaal goood, Elsa!
Elsa: Puttin' the stew on the table
Elsa: Time to do some more housework!
Miner Bob: Tastes real good too!
Miner Bob: Thank ya li'l lady. Ah better get back to whatever ah wuz doin'
Elsa: Washin' the dishes
Miner Bob: ZZZZ...
Elsa: Makin' the bed
Miner Bob: All mah fatigue has drained away. Time to find more gold!
Miner Bob: Walkin' to the gold mine
```

Summing Up

This chapter has shown you the skills required to create very flexible and extensible finite state machines for your own games. As you have seen, the addition of messaging has enhanced the illusion of intelligence a great deal — the output from the WestWorldWithMessaging program is starting to look like the actions and interactions of two real people. What's more, this is only a very simple example. The complexity of the behavior you can create with finite state machines is only limited by your imagination. You don't have to restrict your game agents to just one finite state machine

either. Sometimes it may be a good idea to use two FSMs working in parallel: one to control a character's movement and one to control the weapon selection, aiming, and firing, for example. It's even possible to have a state itself contain a state machine. This is known as a hierarchical state machine. For instance, your game agent may have the states **Explore**, **Combat**, and **Patrol**. In turn, the **Combat** state may own a state machine that manages the states required for combat such as **Dodge**, **ChaseEnemy**, and **Shoot**.

Practice Makes Perfect

Before you dash away and start coding your own finite state machines, you may find it good practice to expand the WestWorldWithMessaging project to include an additional character. For example, you could add a Bar Fly who insults Miner Bob in the saloon and they get into a fight. Before you write the code, grab a pencil and a sheet of paper and sketch out the state transition diagrams for each new character. Have fun!