a i - j u n k i e

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 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 *goldmine*, 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.

(You can download the accompanying project files here (24k))

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`, which 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 book], 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, 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, 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 `GoHomeAnd- SleepTilRested`. Otherwise he will change state to `EnterMine-AndDigForNugget`.

- `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.

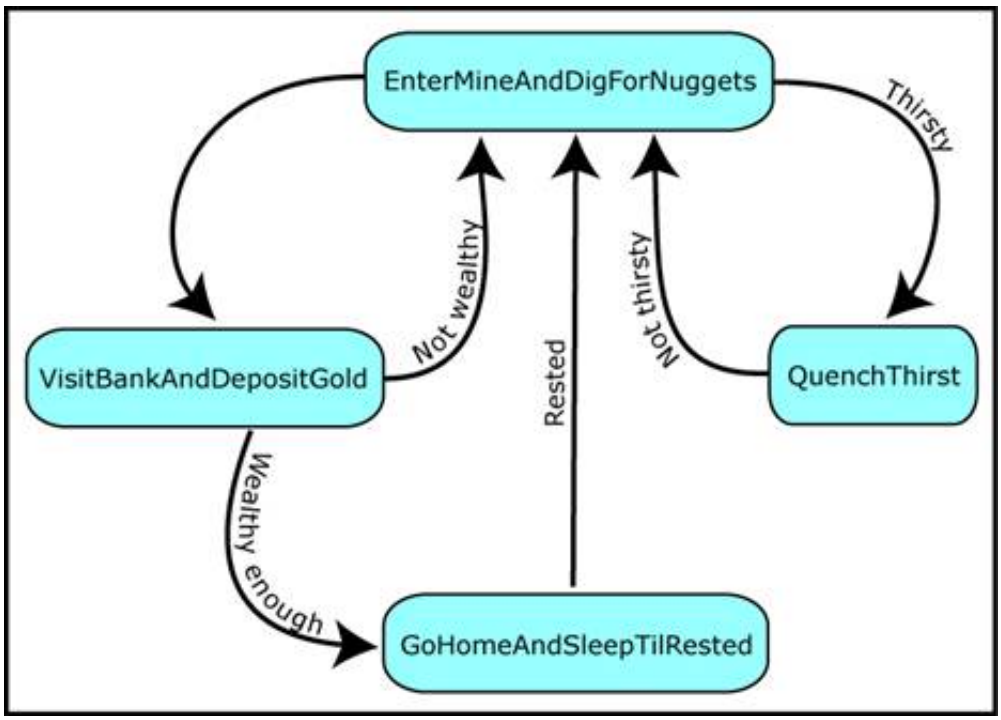

Figure 2.2. Miner Bob's state transition diagram

The State Design Pattern Revisited

You saw a brief description of this design pattern earlier, 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 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. (Click here for an introduction to UML class diagrams)

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 this). 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.

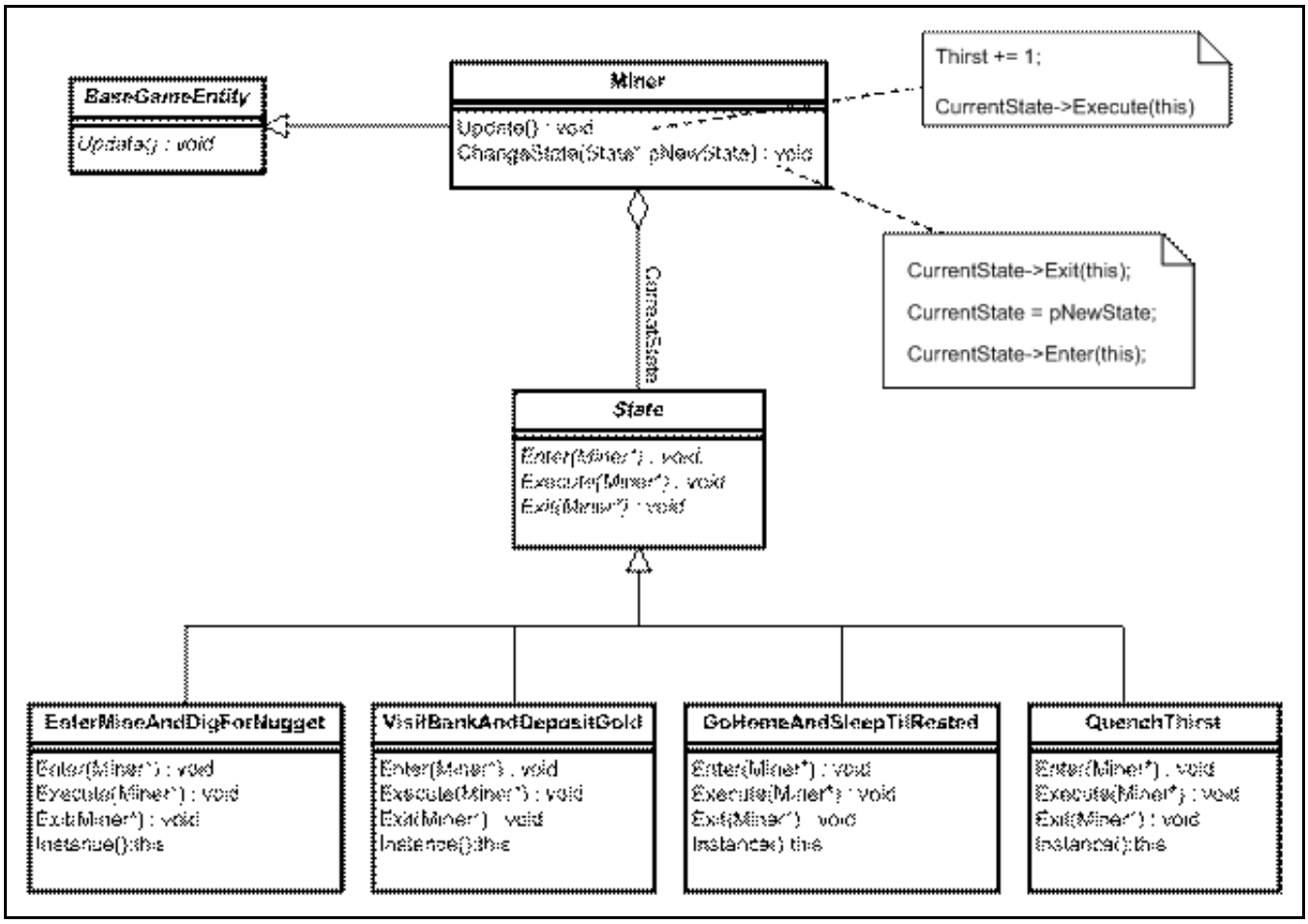

Figure 2.3. UML class diagram for Miner Bob's state machine implementation

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.

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 goldmine, he must
  //change location to the gold mine
  if (pMiner->Location() != goldmine)
  {
    cout << "\n" << GetNameOfEntity(pMiner->ID()) << ": "
         << "Walkin' to the goldmine";

    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 beer.
  pMiner->AddToGoldCarried(1);

  //digging 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 beer
  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 goldmine 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 text 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.
