a i - j u n k i e

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 accept- able, 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.

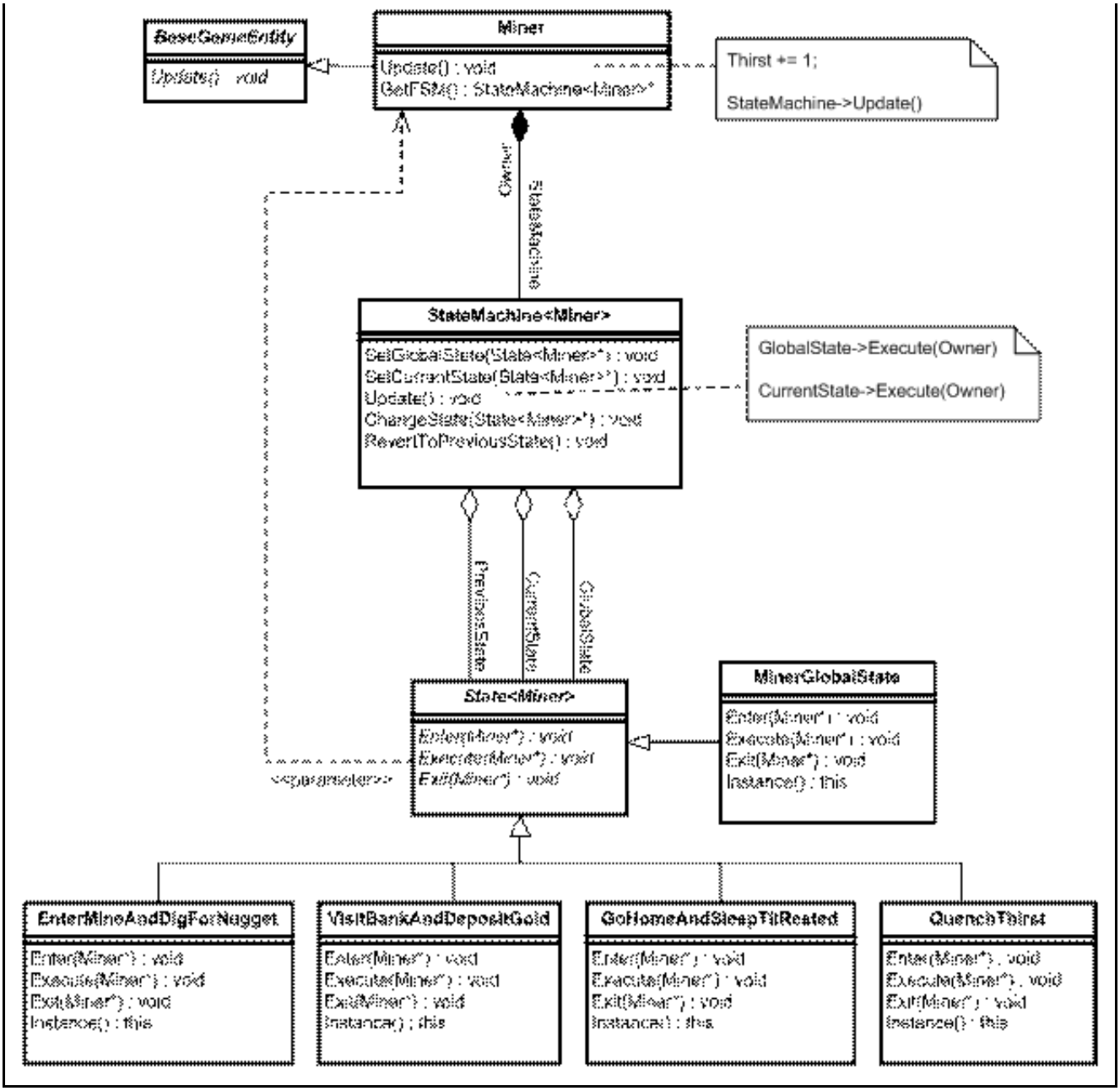

Figure 2.4. The updated design

Introducing Elsa

To demonstrate these improvements, I've created another project: [WestWorldWithWoman](). In this project, West World has gained another inhabitant, Elsa, the gold miner's wife. Elsa doesn't do much; 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.

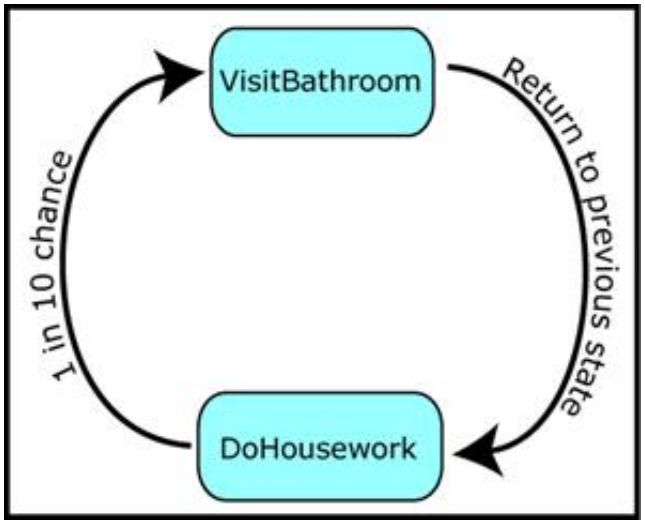

Figure 2.5. Elsa's state transition diagram. The global state is not shown in the figure because its logic is effectively implemented in any state and never changed.

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.

```
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...

Well, that's it folks. 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`.

If you'd like to ask questions or give me some feedback about this article (good or bad, it all helps) then please visit the forum.
