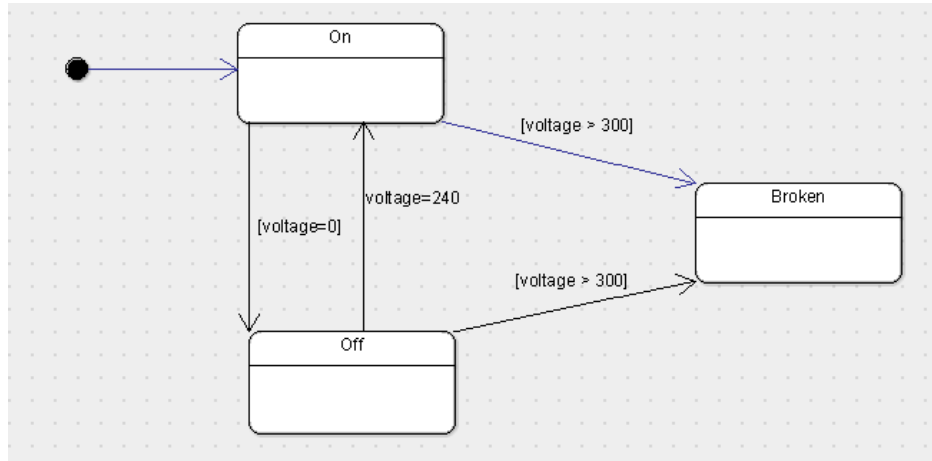


C++ Week 8 State Machines

Introduction

This week we will develop a state machine to model the behaviour of a light bulb. The diagram below describes the states and valid transitions for our bulb.



The solution will contain 5 classes

State : Each state within the state machine is represented as an object instantiated from the state class.

Transition : Each transition from one state to another will be represented by a Transition object.

FSM : Is the controlling class that will be used to manage the state machine.

LinkedList : Will record each state & transitions.

Bulb : This class is responsible for building the State Machine and handling any call-backs.

Getting Started

Create a new C++ project, name it SM and copy the LinkedList header file from BlackBoard to your project folder (..SM/SM). Now add the file to the project. Within the solutions explorer right click Header Files->Add->Existing Item and select the LinkedList.h file.

Before we start defining the classes we must create an enumerate type to represent the valid bulb states. This enumerate type will be used within the State class to identify the object's state and in the Transition class to represent the states from which the transition comes from and goes to. As the enumerate type is shared amongst a number of classes we will define it within the stdafx.h file (at the bottom) as this file will be included within all our source files.

```
enum TStates {On, Off, Broken};
```

C++ Week 8 State Machines

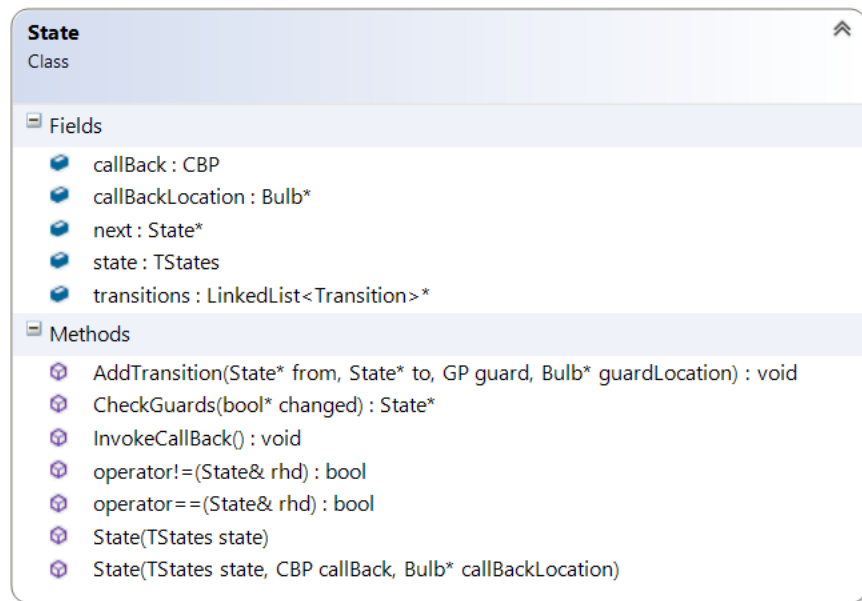
Bulb Class

Now create a couple of function pointers that will be used in both the State, Transition and FSM classes. As they are shared too, we shall place the pointers at the bottom of the stdafx.h file. Ensure you include the declaration of the Bulb class too. This prevents a compiler error being generated as the pointer types refer to the Bulb class that stdafx.h doesn't know about.

```
class Bulb;  
  
typedef void (Bulb::*CBP)();  
typedef bool (Bulb::*GP)();
```

The pointer types represent the types of member functions located within the Bulb class. CBP being the type of the call-back invoked by the FSM when a transition has occurred to a new State and GP is the type of member function, used to check if a transition's guard is true and the state can change.

Now define the State class based on the diagram below.



C++ Week 8 State Machines

Data Members

callback : The pointer to the function within Bulb that will be invoked when the state represented by an instance of State is entered.

callbackLocation : A pointer to the Bulb object

next : State is a link within the FSM's LinkedList so next points to the next link (see last week's notes on LinkedLists).

state : The enumerate state type. E.g. if the State object represented Broken then the state data member would be assigned the enum type Broken.

transitions: A linked List containing pointers to transition objects that represent the transitions from this state.

Member Functions

These include two overloaded constructors.

AddTransition : adds a new transition object to the Linked List.

InvokeCallback : Invokes the call-back function on the Bulb object when the state is entered.

CheckGuards : Checks each transition in the Linked List to see if the state should change.

The two overloaded operators "==" and "!=" are required within the FSM class when search the FSM's Linked List of States.

Now add the definitions of the constructors to the cpp file

```
State::State(TStates state){
    this->state = state;
}

State::State(TStates state,CBP callback, Bulb * callbackLocation ){
    this->state = state;
    this->callback = callback;
    this->callbackLocation = callbackLocation;
    transitions = new LinkedList<Transition>();
    next = 0;
}
```

The Overloaded operators can also be included within the cpp file. Note how the variable of the enumerate type TState is checked to see if the objects are the same.

```
bool State::operator != (State& rhd){
    return (this->state != rhd.state);
}

bool State::operator == (State& rhd){
    return (this->state == rhd.state);
}
```

C++ Week 8 State Machines

State Class

Now add the function responsible for inserting a new transition into the Linked list.

```
void State::AddTransition(State * from, State * to, GP guard, Bulb * guardLocation){
    transitions->Insert(new Transition(from,to,guard,guardLocation));
}
```

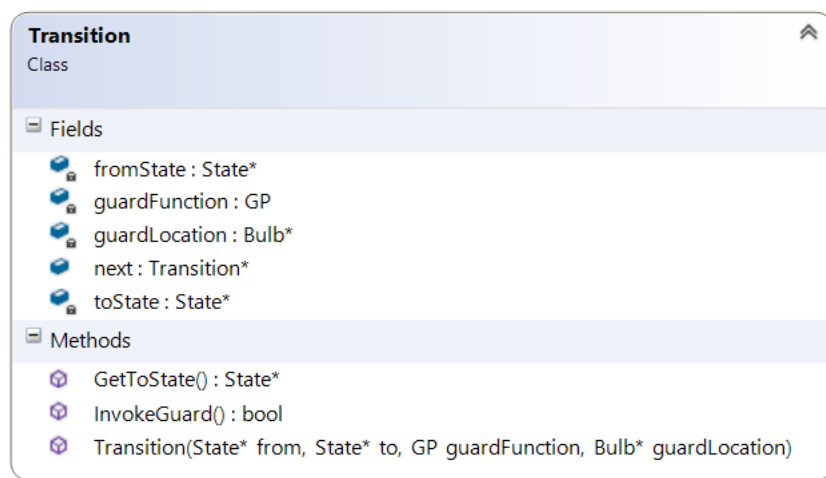
The member function that invokes the call-back

```
void State::InvokeCallBack(){
    (callBackLocation->*callBack)();
}
```

Finally add the function that checks the guards to see if a transition should take place.

```
State * State::CheckGuards(bool * changed){
    for (int n = 0; n < transitions->Count(); n++){
        // Can not overload pointer operators so must dereference linked list to ensure [] works.
        if((*transitions)[n]->InvokeGuard()){
            *changed = true;
            return (*transitions)[n]->GetToState();
        }
    }
    return 0;
}
```

The Transition class can now be added to your project.



C++ Week 8 State Machines

Transition Class

Above the class within Transitions.h include the declarations of the State classes (as below).

```
class State;
```

As State refers to Transition and Transition to State the application will not compile as they are both dependant on the other class being compiled first. By declaring State in Transition we tell the compiler that the State class exists so that Transition can be compiled.

Data Members

fromState : Pointer to state object from which the transition originates. e.g. the transition from On to Off originates from the On state.

toState : Pointer to state object to which the transition flows to.

next : Transition is a link within the State's Linked List, so it must have a next member pointing to the next Transition in the linked list.

guardFunction : pointer to the function within Bulb that will check the guard to see if a transition can take place.

guardLocation : pointer to the bulb object.

Member Functions

A constructor that initialises the data members.

InvokeGuard : Invokes the function whose address is assigned to guardFunction.

GetToState : returns a pointer to the toState data member which is private;

The definitions of these functions should now be placed within the cpp file.

```
Transition::Transition(State * from, State * to, GP guardFunction, Bulb *guardLocation){
    this->fromState = from;
    this->toState = to;
    this->guardFunction = guardFunction;
    this->guardLocation = guardLocation;
}

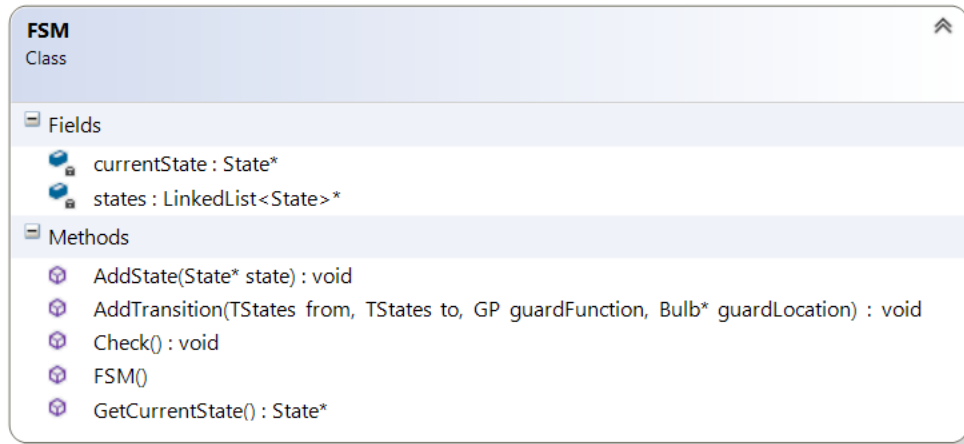
bool Transition::InvokeGuard(){
    if (guardFunction != 0 && guardLocation != 0){
        return (guardLocation->*guardFunction)();
    }
    return false;
}

State * Transition::GetToState(){
    return toState;
}
```

C++ Week 8 State Machines

Finite State Machine Class

Now we can add the FSM class to the project.



Data Members

currentState : A pointer to the current State object.

states : A Linked List of all the states in the State Diagram.

Member Functions

AddState : Adds a state to the states Linked List

AddTransition : Adds a transition to the State's class transitions Linked List

Check : Checks the guards of the transitions for the current state to see if a transition should take place.

GetCurrentState : returns the currentState, which is a private data member.

Let's now define these FSM member functions within the cpp file.

First the constructor and GetCurrentState.

```
FSM::FSM(void){
    states = new LinkedList<State>();
    currentState = 0;
}

State * FSM::GetCurrentState(){
    return currentState;
}
```

C++ Week 8 State Machines

FSM Class continued

Now the member functions that Add States and Transitions.

```
void FSM::AddState(State *state){
    // The first state to be added becomes the initial/current state
    if (states->IsEmpty()){
        currentState = state;
        currentState->InvokeCallBack();
        states->Insert(state);
    }
    else {
        // Check to ensure the state has not already been added
        if (!states->Find(state)){
            states->Insert(state);
        }
        else{
            cout << "The state has already been added" << endl;
        }
    }
}

void FSM::AddTransition(TStates from, TStates to, GP guardFunction, Bulb * guardLocation){
    // Ensure states have been added
    State * foundState = states->Find( new State(from));
    State * toState = states->Find( new State(to));
    if (foundState != 0 && toState != 0){
        foundState->AddTransition(foundState,toState,guardFunction,guardLocation);
    }
    else{
        cout << "Error: Invalid Transition : State/s missing" << endl;
    }
}
```

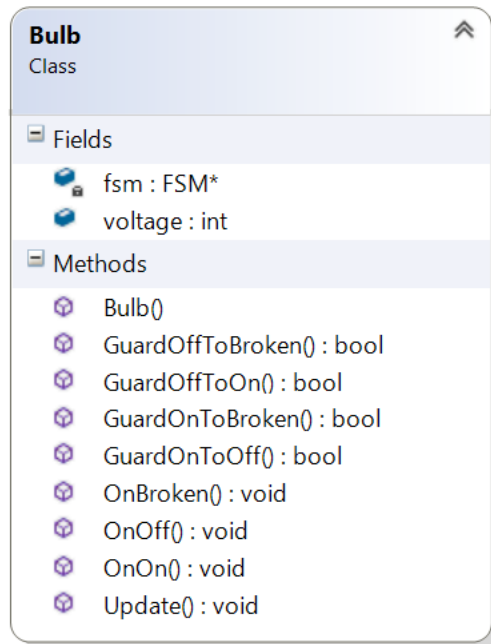
```
void FSM::Check(){
    bool changed = false;
    // Check all guards on the current state to see if a transition should take place.
    // Return the new state if a guard has been meet or 0 if no guards meet.
    State * toState = currentState->CheckGuards(&changed);
    // If a guard has been meet
    if (changed){
        // Update the current state
        currentState = toState;
        currentState->InvokeCallBack();
    }
}
```

C++ Week 8 State Machines

Bulb Class

The three classes created so far define the state machine framework. We will now use this framework to create a state machine for the light bulb. To do this we create a Bulb class that will be responsible for populating the framework with the states and transitions.

Create the skeleton of the Bulb class below.



Data members

fsm : a pointer to the FSM

voltage: Used within the guards to determine if the transition should take place.

Member Functions

A default constructor that instantiates the FSM and adds the states and transitions to it. Three `On**` call-backs that are invoked when the State changes. e.g. if the state changes to the Off state the `OnOff()` call-back will be invoked. These call-backs simply display a message to the console displaying the state name.

Four Guard functions that represent the guards on each of the transitions. These functions compare the voltage (see state diagram) and return true if the guard is met.

`Update` : Invokes the FSM's `Check` member function to see if a transitions should take place.

C++ Week 8 State Machines

Bulb Constructor

First let's define the constructor in the cpp file.

```
Bulb::Bulb(){
    voltage = 240;
    fsm = new FSM();
    fsm->AddState(new State(On,&Bulb::OnOn,this));
    fsm->AddState(new State(Off,&Bulb::OnOff,this));
    fsm->AddState(new State(Broken,&Bulb::OnBroken,this));

    fsm->AddTransition(On,Off,&Bulb::GuardOnToOff,this);
    fsm->AddTransition(On,Broken,&Bulb::GuardOnToBroken,this);
    fsm->AddTransition(Off,Broken,&Bulb::GuardOffToBroken, this);
    fsm->AddTransition(Off,On,&Bulb::GuardOffToOn, this);
}
```

Now the Call-backs

```
void Bulb::OnOn(){
    cout << "OnOn" << endl;
}

void Bulb::OnOff(){
    cout << "OnOff" << endl;
}

void Bulb::OnBroken(){
    cout << "OnBroken" << endl;
}
```

Now the Guards

```
bool Bulb::GuardOnToOff(){
    return voltage == 0;
}

bool Bulb::GuardOnToBroken(){
    return voltage > 300;
}

bool Bulb::GuardOffToBroken(){
    return voltage > 300;
}

bool Bulb::GuardOffToOn(){
    return voltage == 240;
}
```

```
void Bulb::Update(){
    fsm->Check();
}
```

C++ Week 8 State Machines

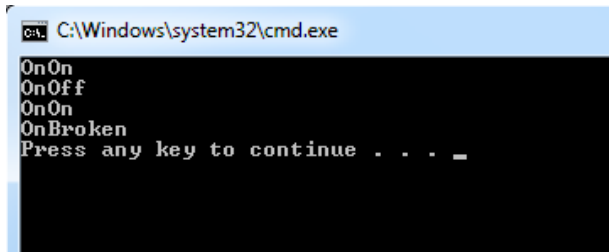
Main function of the bulb example

The last step is to create an instance of Bulb within main and test it by setting a new voltage and then invoking Update to force the FSM to check the guards to see if a transition has taken place and if so change the state.

```
Bulb *bulb = new Bulb();  
bulb->Update();  
bulb->voltage = 0;  
bulb->Update();  
bulb->voltage = 240;  
bulb->Update();  
bulb->voltage = 500;  
bulb->Update();  
system("pause");
```

Be sure to include the Bulb header file.

Now run the application and your output should resemble this

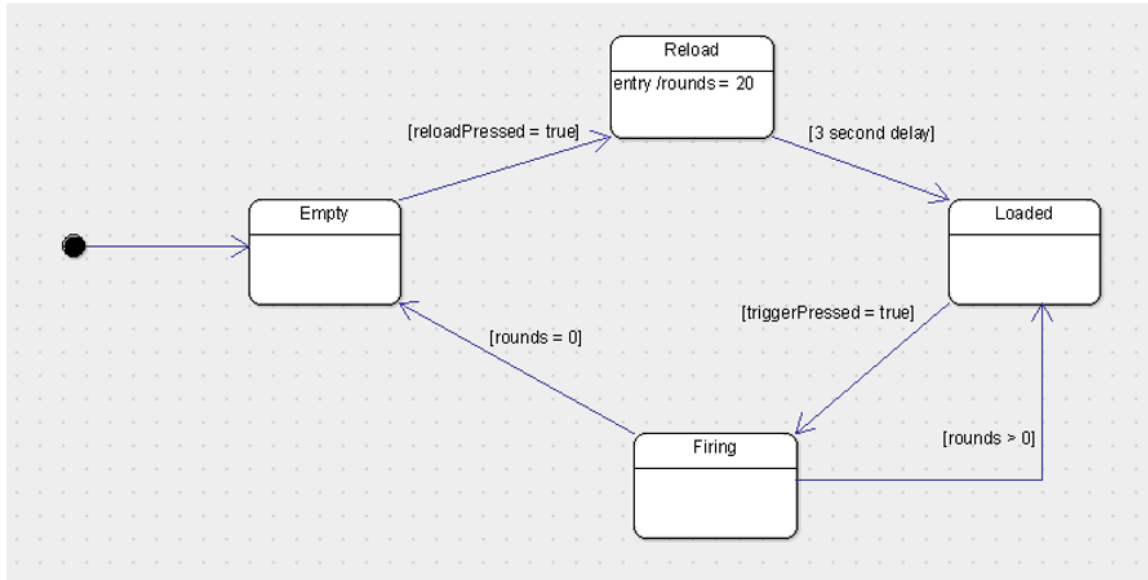


```
C:\Windows\system32\cmd.exe  
OnOn  
OnOff  
OnOn  
OnBroken  
Press any key to continue . . . _
```

C++ Week 8 State Machines

Exercise (Weapon)

The state diagram below models a simple weapons system. Create a new Project within Visual Studio that will implement the state behaviour described below.



Hints:

1. Add the .h LinkedList files to the project.
2. Assuming the class name you give the class that contains the call-back and guard functions is named `Weapon`, add a declaration of `weapon` to the `stdafx.h` file along with the call-back and guard function types.
3. Create an enum type named `TStates` that contains all the weapon's states.
4. Recreate the State, Transition & FSM classes within this project. Make sure you replace any references to `Bulb` with `Weapon`.
5. Create the `Weapon` class (similar to `Bulb`) but ensure there are call-backs for each state and guard. Within the constructor create the states and transitions.
6. Within `main` create an instance of `weapon` and test it.