C++

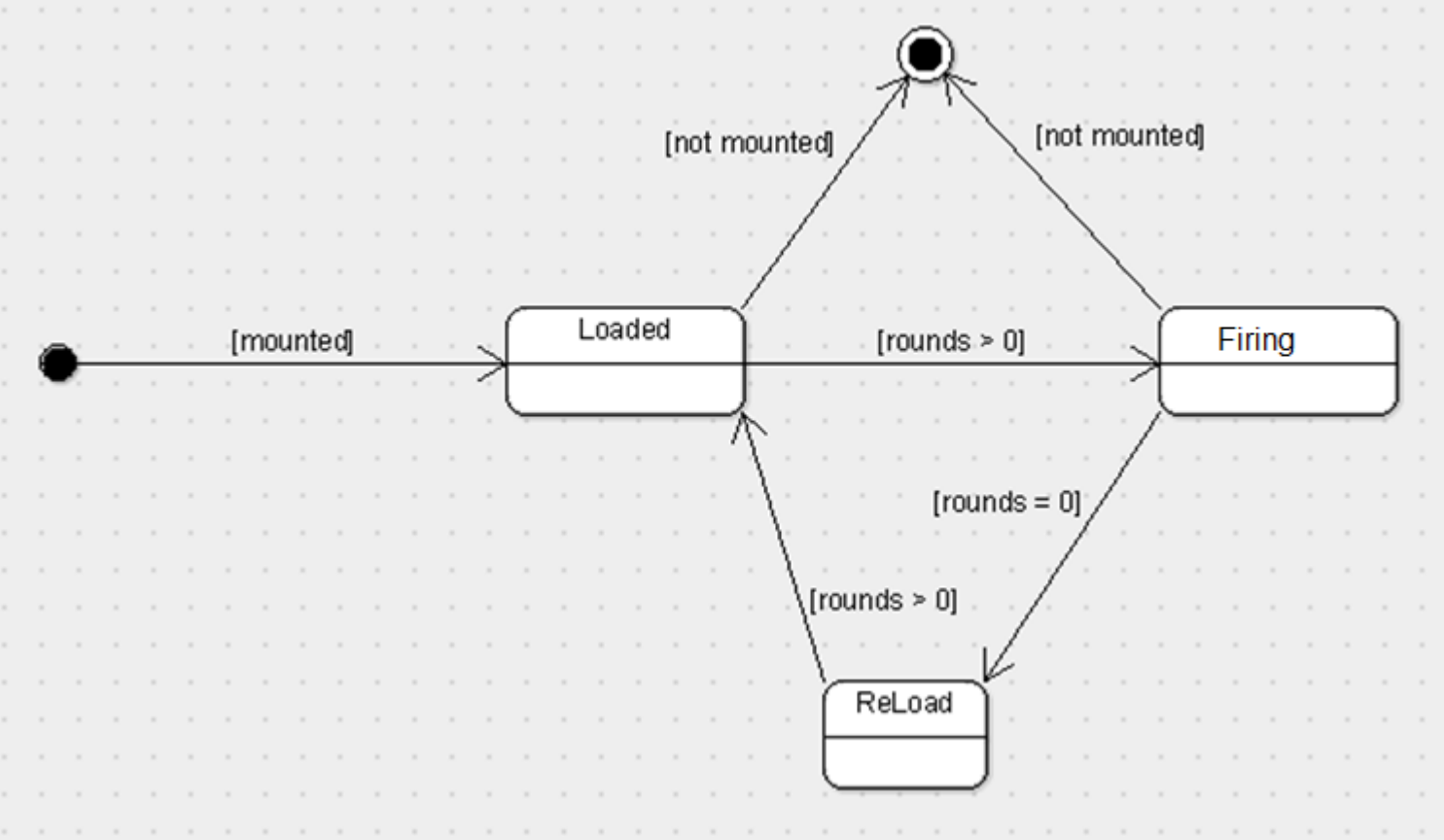
State Machines

CO658 Data Structures & Algorithms

Topics

- State Diagrams
- Implementation with if statements & enum types
- Call-backs
- Object implementation

Weapon State Diagram



Implementation

- Use enum type to record current state.

```
enum TStates {InActive,Loaded,Firing,Reload};
TStates currentState = InActive;
```

- If statements to check current state and implement transitions

```
if (currentState == InActive && mounted)
        currentState = Loaded;
```

```
enum TStates {InActive,Loaded,Firing,Reload};
int rounds;
bool mounted;

TStates CheckState(TStates currentState){
        if (currentState == InActive && mounted)
                currentState = Loaded;
        else if (currentState == Loaded && rounds > 0)
                currentState = Firing;
        else if (currentState == Loaded && !mounted)
                currentState == InActive;
        else if (currentState == Firing && rounds == 0)
                currentState = Reload;
        else if (currentState == Firing && !mounted)
                currentState = InActive;
        else if (currentState == Reload && rounds > 0)
                currentState = Loaded;
        return currentState;
}

int main(){
        TStates weaponState = InActive;
        rounds = 10;
        mounted = true;
        weaponState = CheckState(weaponState);
        return 0;
}
```

Call-backs

- Sometimes it is necessary to execute code when a transition occurs.
- The statements to be executed could be placed within a call-back function.

```
void OnFire(){
      rounds--;
}
```

```
if (currentState == Loaded && rounds > 0){
                currentState = Firing;
                OnFire();
}
```

Disadvantages

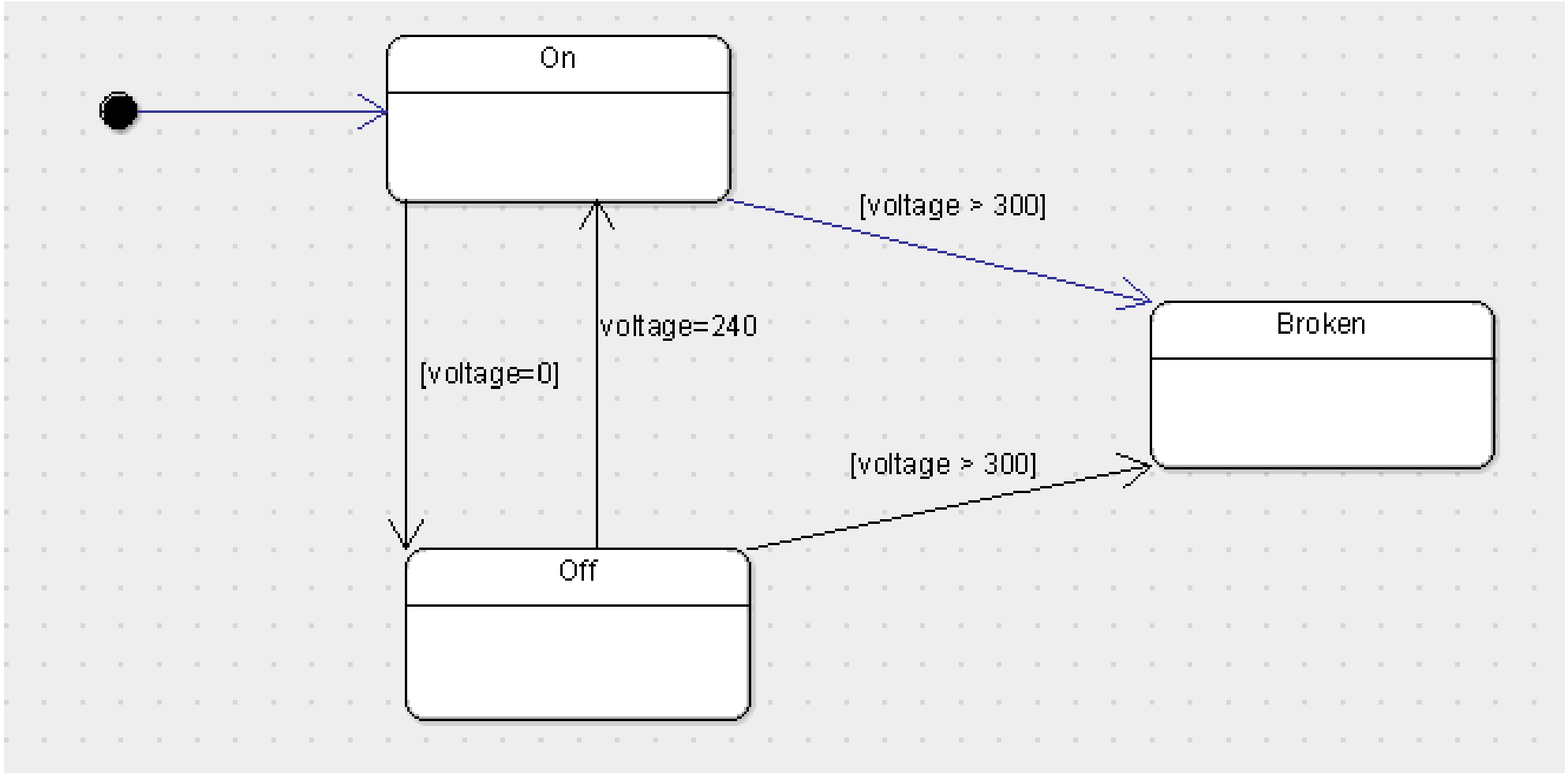
- Its easy to introduce invalid transitions.
- Code may not be centralised and therefore difficult to manage.
- Poor readability and extensibility

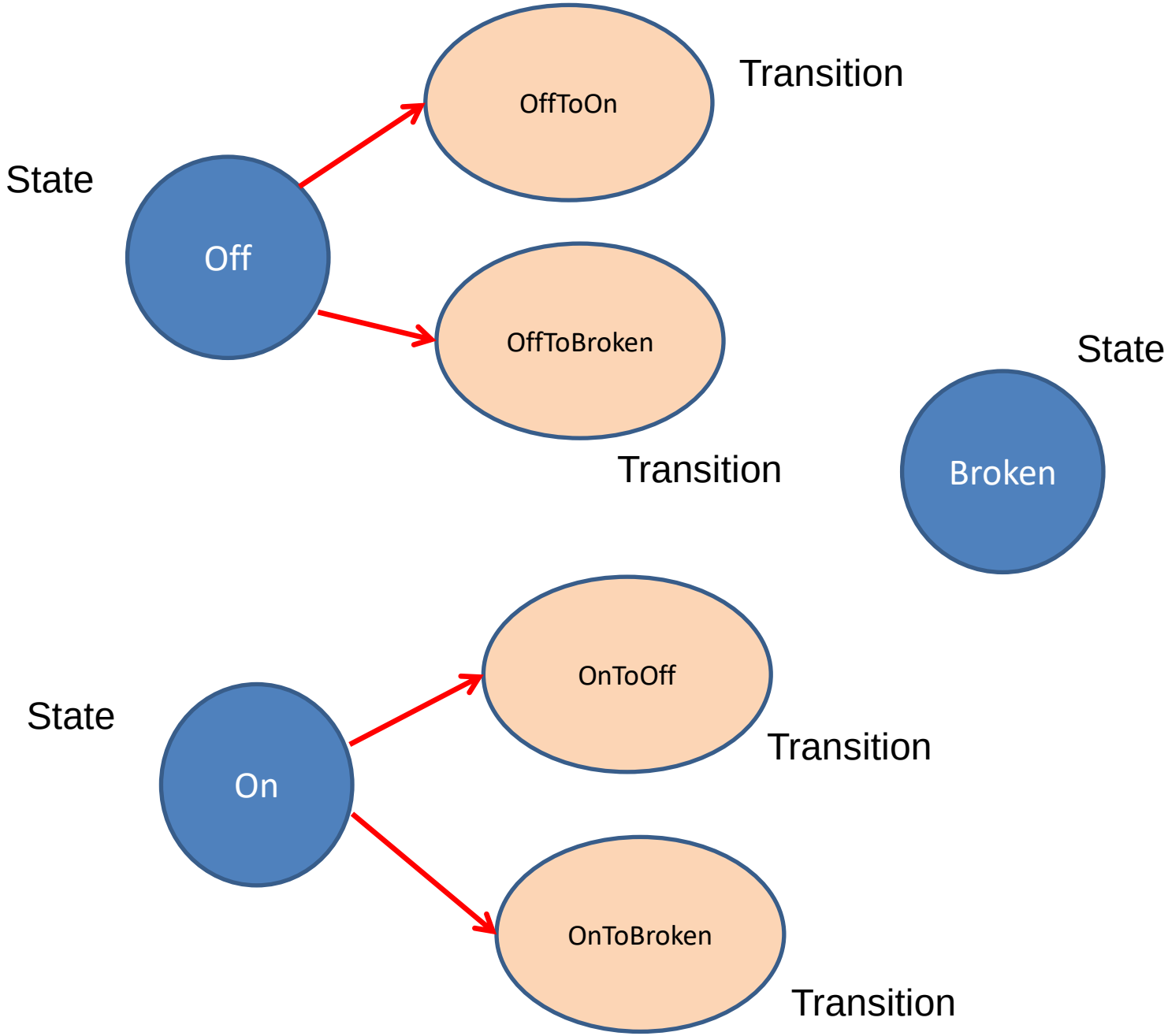
Object Approach

An alternative approach would be to

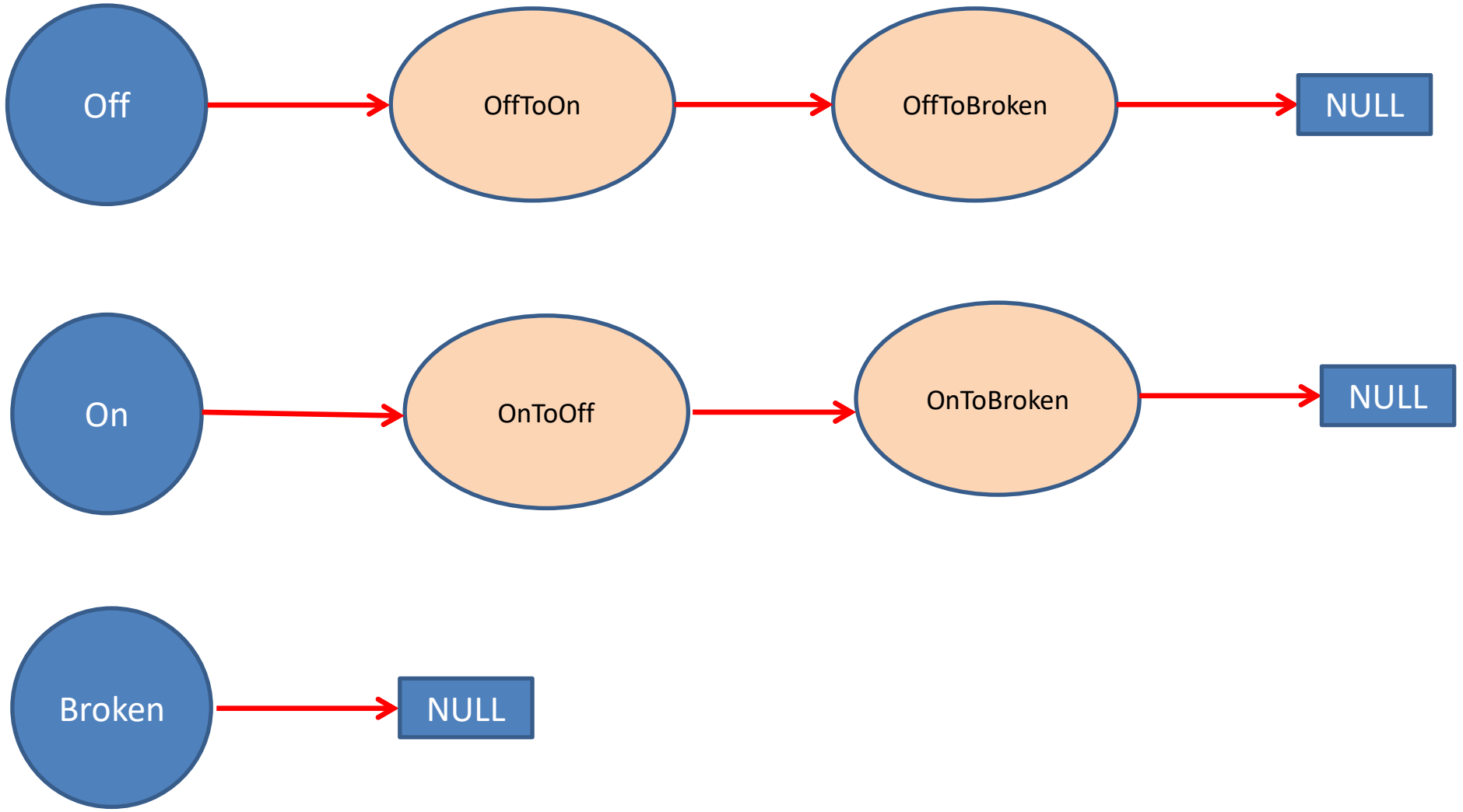
- Represent States as Objects.
- Represent Transitions as Objects.
- Guards as Functions.
- The guard function return a bool value to determine if a transition should take place.
- Each state object is assigned an enum value
- Each state object would have a list of one or more transitions.

Example : Light Bulb




State
Off
OffToOn
Transition
OffToBroken
Transition
State
Broken
State
On
OnToOff
Transition
OnToBroken
Transition

States have a Linked list of Transitions



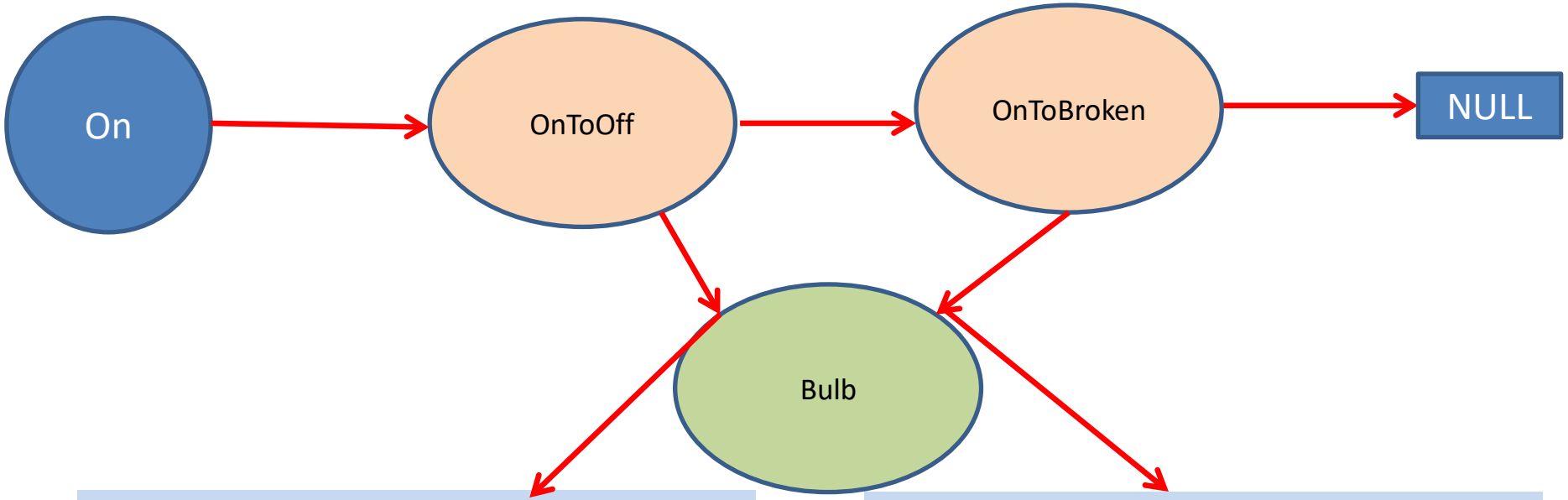
Guard Functions

- Each transition has a pointer to a guard function.
- The Guard determines if the transition can take place.
- The guard functions are centrally located in the Bulb object.


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

Check Guards

- If the current state is On.
- Iterate through the Linked List checking the transition's guards.

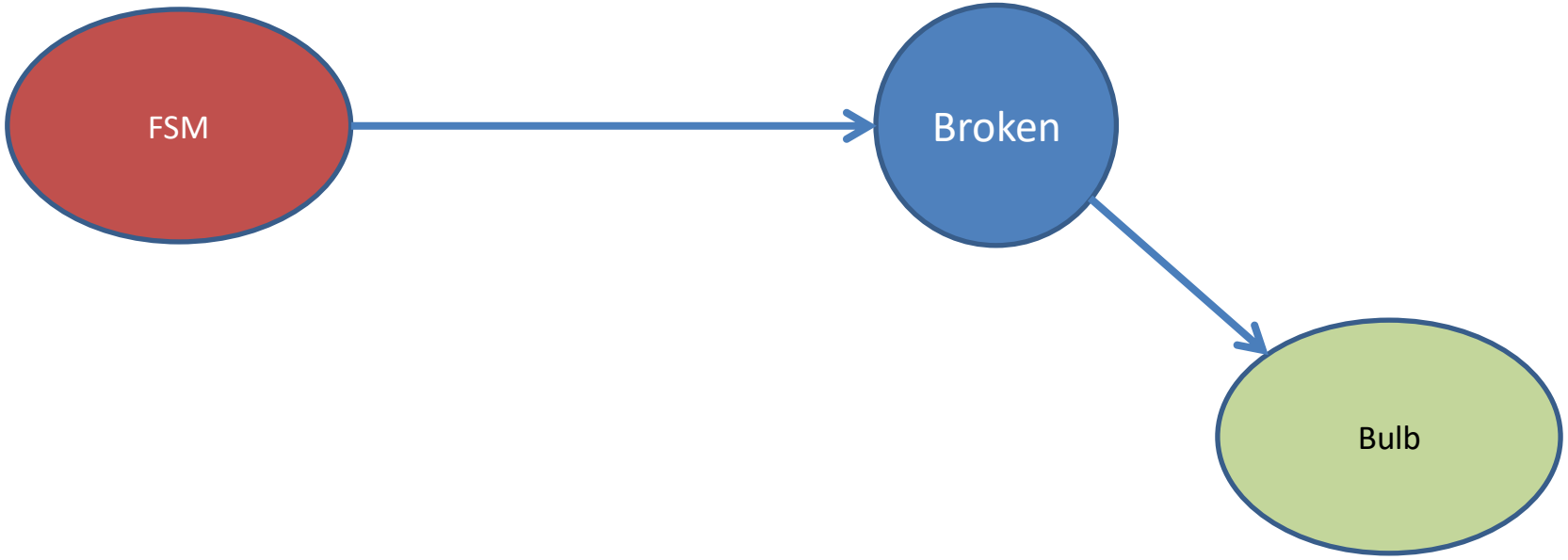



```
bool Bulb::GuardOnToOff(){
return voltage == 0;
}
```

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

Call-back

- If a guard returns true and the state changes then the FSM object changes the current state and invokes the call-back of the new state.




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

Classes

Conclusion

- Whilst the object approach is more complicated, the frame work is reusable.
- The FSM is partly configurable and less reliant on coding.
- Guards and call-backs are centralised.