



# 互動程式設計III

Interactive Programming Design Integration

# Introduction to Finite State Machine

- In this lesson you'll learn:
  - Explore the concept of FSMs and understand their fundamental structure and behavior.
  - Learn where FSMs are used, such as in game development, AI systems, and workflow modeling.
  - Build a simple FSM from scratch using C# and see how states and transitions are defined and managed.
  - Understand how FSMs align with common design patterns like State and Strategy, to solve specific design challenges.
- Learning Objectives
  - Comprehend the core principles of **states**, **transitions**, and **events** that govern FSMs.
  - Write code to implement a simple FSM and manage state transitions effectively.
  - Use UML state diagrams to visualize FSM design and improve communication within your team.
  - Recognize when to apply FSMs and integrate them with broader software architecture.
- Why This Chapter is Important
  - Improved Software Design with FSM
    - State Management Made Simple: FSMs provide a structured way to handle complex state transitions, ensuring your software behaves predictably.
    - Ideal for Game Development and UI Systems: FSMs are crucial in scenarios where objects or entities transition between distinct states, such as game characters, UI elements, or network protocols.
  - Enhanced Code Maintainability
    - Clear and Manageable Code: By encapsulating state logic within discrete states, FSMs promote separation of concerns and reduce the chance of errors.
    - Easier to Debug and Extend: Modifying behavior becomes more straightforward by adding new states or transitions without impacting the entire system.



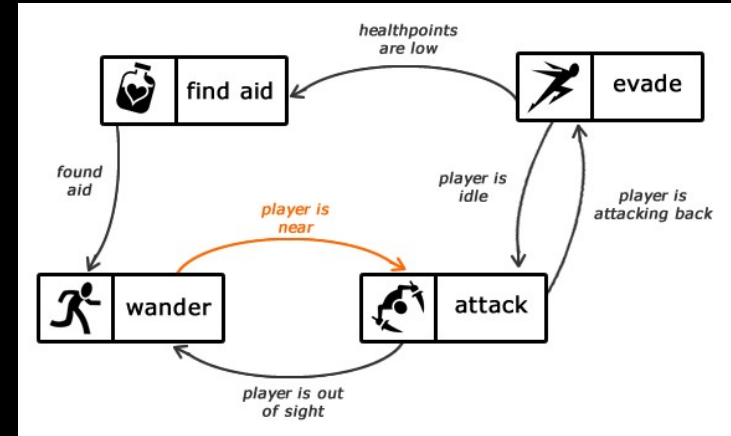
# 互動程式設計III

Interactive Programming Design Integration

- **Finite State Machine**
- **Switch Type**
- **Design Pattern Type**
-

# What is a Finite State Machine (FSM)?

- Definition:
  - A Finite State Machine (FSM) is a computational model used to represent and control the behavior of an object that can exist in a finite number of states.
- Key Components:
  - **States**: Different configurations the system can be in (e.g., On, Off, Broken).
  - **Transitions**: The **internal** rules or conditions for moving from one state to another.
  - **Events**: **External** inputs or triggers that cause state transitions.
- Examples of FSM Applications:
  - Game Development:
    - Managing character states (idle, running, jumping).
  - AI Systems:
    - Defining NPC behavior based on events and triggers.
  - User Interfaces:
    - Handling button interactions, UI element states.
  - Workflow Automation:
    - Modeling steps in business processes.



A example Enemy FSM [\*]

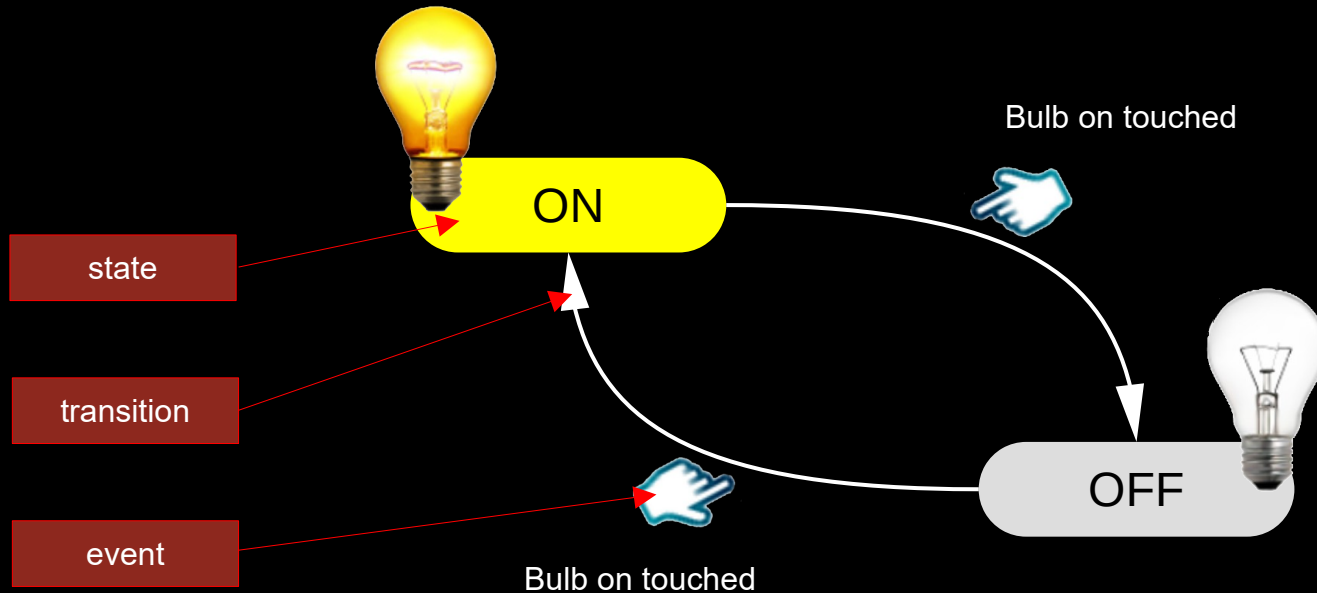
# EX: A Bulb FSM

- In this experiment, we will implement a Finite State Machine (FSM) using Unity3D to simulate a light bulb with three states:
  - On – The bulb is emitting light.
  - Off – The bulb is inactive but functional.
  - Broken – The bulb is damaged and cannot function anymore.
- How It Works
  - Interactive State Changes:
    - The user can tap on the light bulb in the Unity3D scene to switch its state between On and Off.
  - Breaking the Bulb:
    - After three state transitions, the bulb enters the "Broken" state, disabling further interaction.
  - Repairing the Bulb:
    - When the Bulb is broken, the player can press 'r' key to repair all the Bulbs.
  - State Transition Logic:
    - On → Off: Tapping switches the light off.
    - Off → On: Tapping switches the light back on.
    - On/Off → Broken: The bulb reaches this state after three taps, and further interaction is disabled.

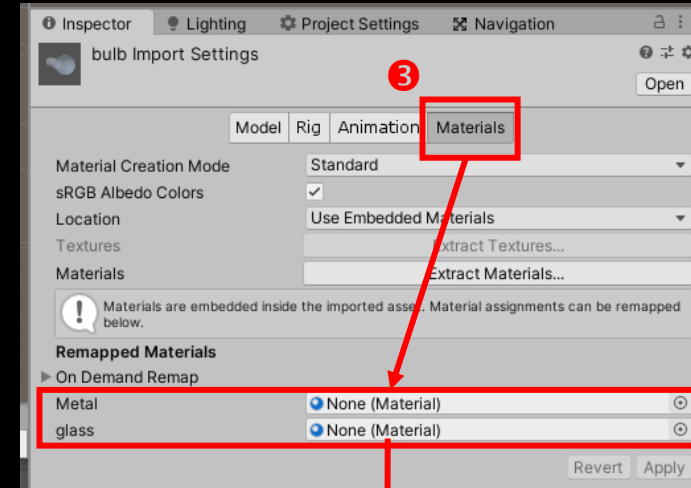
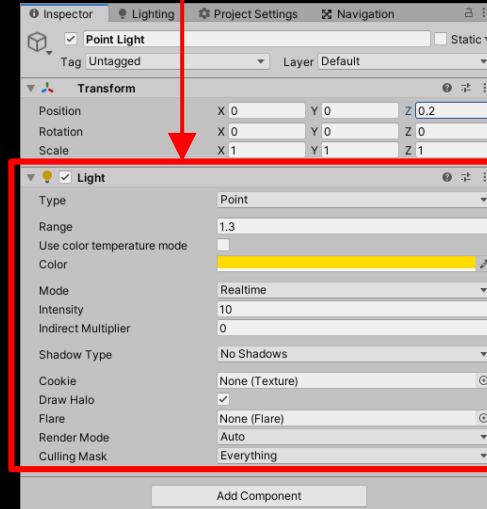
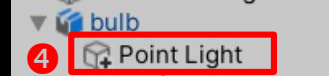
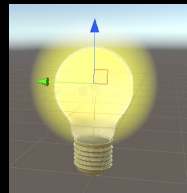
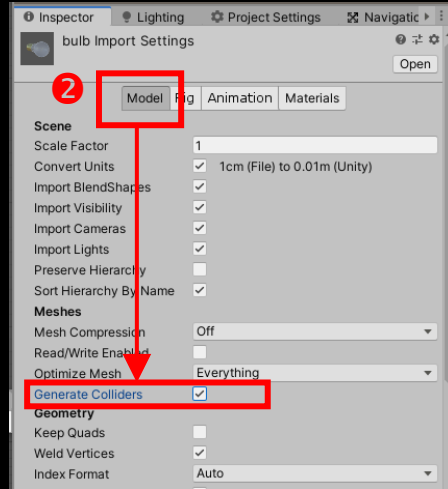
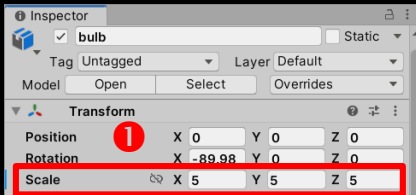
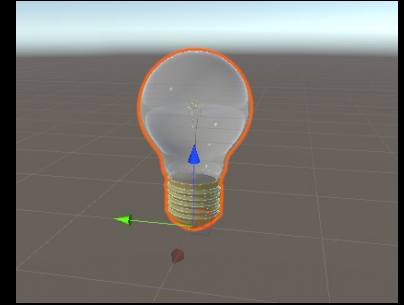


# EX: A Bulb FSM

- Design the FSM **state diagram** for your **device** (the bulb).
  - State: An abstract model which defines and describes the features/attributes of the device.
  - Transition: The process of changing from one state to another. Transitions are typically arised from internal changes of a device.
  - Events: The external stimuli that can enable or trigger specific transition of the device.



- Import <bulb.fbx> to <Asset/Models/bulb>.
  - Set scale to (5, 5, 5). ❶
  - Check Generate Colliders. This will be needed to test user click later. ❷
  - Extract Materials to the same folder. ❸
- Add a bulb gameObject to scene.
  - Add a point light inside the bulb and set as follow. ❹
  - We can turn on and off the light component to dim.



# EX: Simple ON/OFF States

- Put all codes in FSMBulb namespace in order to avoid naming confliction.
- Define a enum of all states. Name this enum 'STATE'.
  - For external class to access this enum, the code should be put like FSMBulb.STATE.ON.
  - Declare a Light component and bind it to the one which is on child gameObject.
- Design all states.
  - Implement visual representation in the state.
- Design all events.
  - Use OnMouseDown to detect left mouse button (LMB) click.
  - Use if/else condition to set transition logic correspondingly.

```
1 using UnityEngine;
2
3 namespace FSMBulb {
4     public enum STATE {
5         ON,
6         OFF
7     }
8
9     public class EX_Bulb_01 : MonoBehaviour {
10         public STATE state;
11         private Light pLight;
12
13         void Start() {
14             pLight = GetComponentInChildren<Light>();
15         }
16
17         void Update() {
18             switch (state) {
19                 case STATE.ON:
20                     pLight.enabled = true;
21                     break;
22                 case STATE.OFF:
23                     pLight.enabled = false;
24                     break;
25                 default:
26                     break;
27             }
28
29             private void OnMouseDown() {
30                 if (state == STATE.ON) {
31                     GoToState(STATE.OFF);
32                 }
33                 else if (state == STATE.OFF) {
34                     GoToState(STATE.ON);
35                 }
36             }
37
38             private void GoToState(STATE targetState) {
39                 state = targetState;
40             }
41         }
42     }
43 }
```

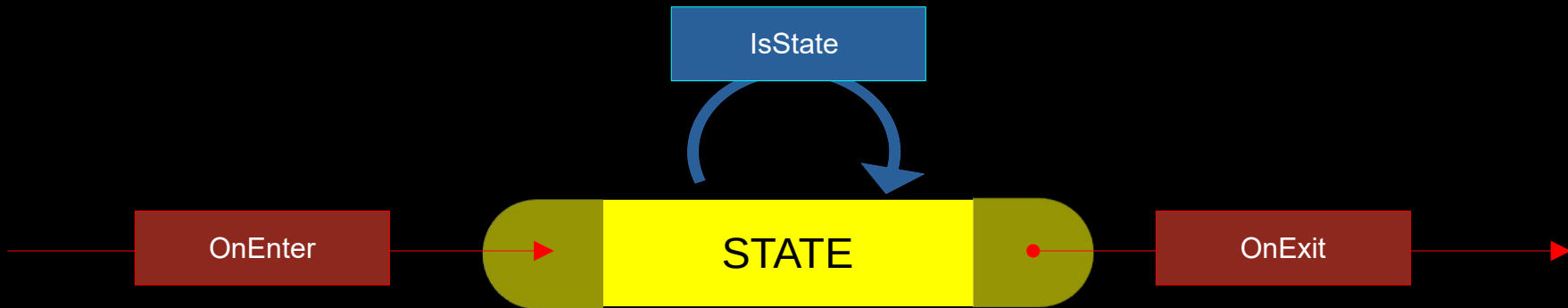


- Test the bulb.



# EX: Fine-grained FSM

- A more fine-grained State design
  - A state can be further splitted into 3 sub-states: OnEnter, IsState, OnExit
  - Run for once: OnEnter & OnExit
    - This type of sub-state only runs for once in one frame. And then transit to the next state.
    - To keep it simple, we just implement OnEnter in the following examples.
  - Run forever: IsState
    - This type of sub-state runs forever if there is no coming external interrupt or matched internal condition.
    - This is what we had implemented in the big switch loop of last example.
- Put the logics that just need to run for once in OnEnter!



- Implementation of OnEnter in switch type FSM:

1. Add a bool triggerEnter.
2. Detect whether triggerEnter is true in the beginning of the codes of state.
3. Reset triggerEnter to false at the end of Update.
4. Set triggerEnter to true when transition starts. In our example this should be put in GoToState().

```
STATE state;  
1 bool triggerEnter;
```

```
switch (state)  
{  
    case STATE.ON:  
        2 if (triggerEnter)  
        {  
            //  
            // Put OnEnter codes here.  
            //  
        }  
        //  
        // Put IsState codes here.  
        //  
        break;
```

⋮

```
    default:  
        break;  
}
```

```
3 triggerEnter= false;
```

```
private void GoToState(STATE targetState) {  
    state = targetState;  
    4 triggerEnter = true;  
}
```

```

1  using UnityEngine;
2
3  namespace FSMBulb {
4      public class EX_Bulb_02 : MonoBehaviour {
5          public STATE state;
6          private Light pLight;
7          private bool triggerEnter;
8
9          void Start() {
10             pLight = GetComponentInChildren<Light>();
11             triggerEnter = false;
12         }
13
14         void Update() {
15             switch (state) {
16                 case STATE.ON:
17                     if (triggerEnter) {
18                         pLight.enabled = true;
19                     }
20                     break;
21                 case STATE.OFF:
22                     if (triggerEnter) {
23                         pLight.enabled = false;
24                     }
25                     break;
26                 default:
27                     break;
28             }
29             triggerEnter = false;
30         }
31     }

```

- Apply the modification to previous example:
  - Add a new script EX\_Bulb\_02.cs
  - Use triggerEnter to detect whether the bulb is just enter a state.
    - Clear triggerEnter at the end of every tick. ❶
    - Set triggerEnter when state transits. ❷

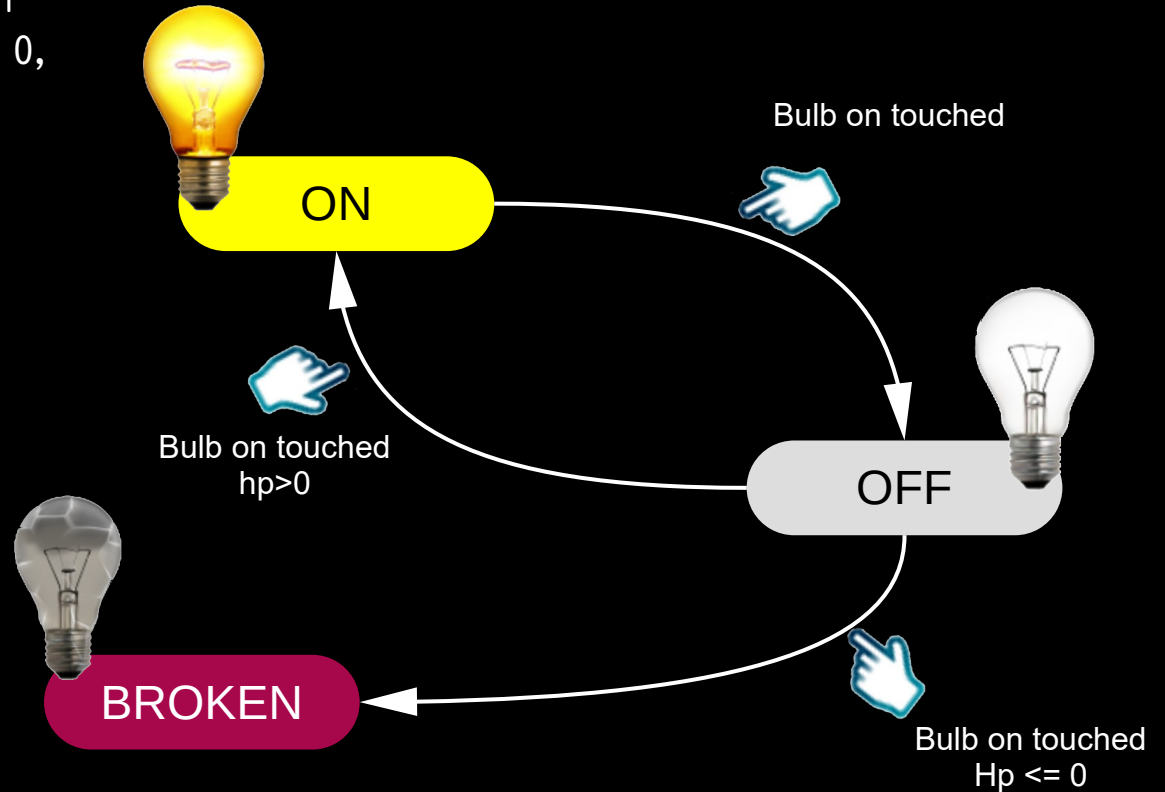
```

31
32     private void OnMouseDown() {
33         if (state == STATE.ON) {
34             GoToState(STATE.OFF);
35         }
36         else if (state == STATE.OFF) {
37             GoToState(STATE.ON);
38         }
39     }
40
41     private void GoToState(STATE targetState) {
42         state = targetState;
43         triggerEnter = true;
44     }
45
46 }

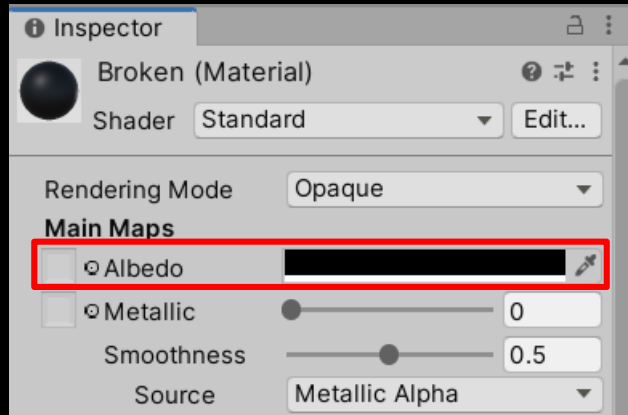
```

# EX: A Broken State

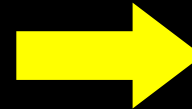
- The rules to break a bulb:
  - Each of the bulb has 3 points of HP at start.
  - When ever a bulb enters ON state, it consumes 1 point of HP.
  - When a bulb is trying to turn on from OFF state but its HP is  $\leq 0$ , it goes to BROKEN state.



- Visual representation of a broken bulb
  - Add a broken material. Set its albedo to RGB(0, 0, 0)
  - Switch the 0<sup>th</sup> material from glass to broken. This will make the bulb look darker.
- To save some processing power, we only update visual representations in the first frame when the bulb just entered a new state.



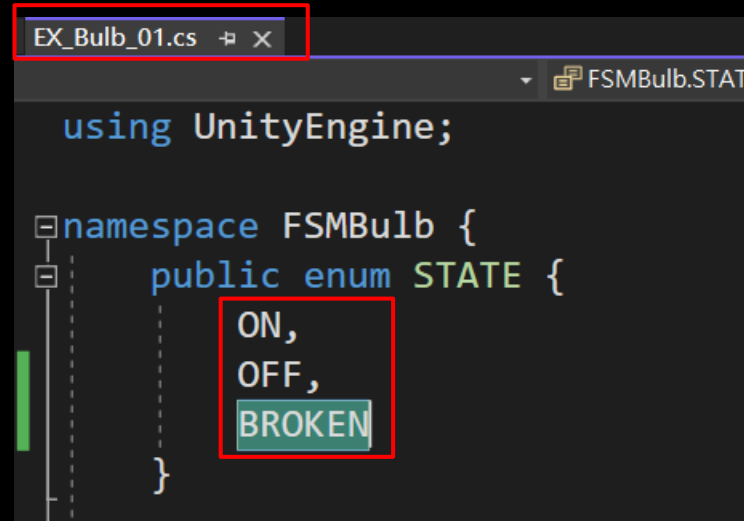
OFF state



BROKEN state



- Append the STATE enum:
  - Go back to EX\_Bulb\_01.cs.
  - Find the definition of STATE and append a BROKEN state at the end.



```
EX_Bulb_01.cs
```

```
using UnityEngine;

namespace FSMBulb {
    public enum STATE {
        ON,
        OFF,
        BROKEN
    }
}
```

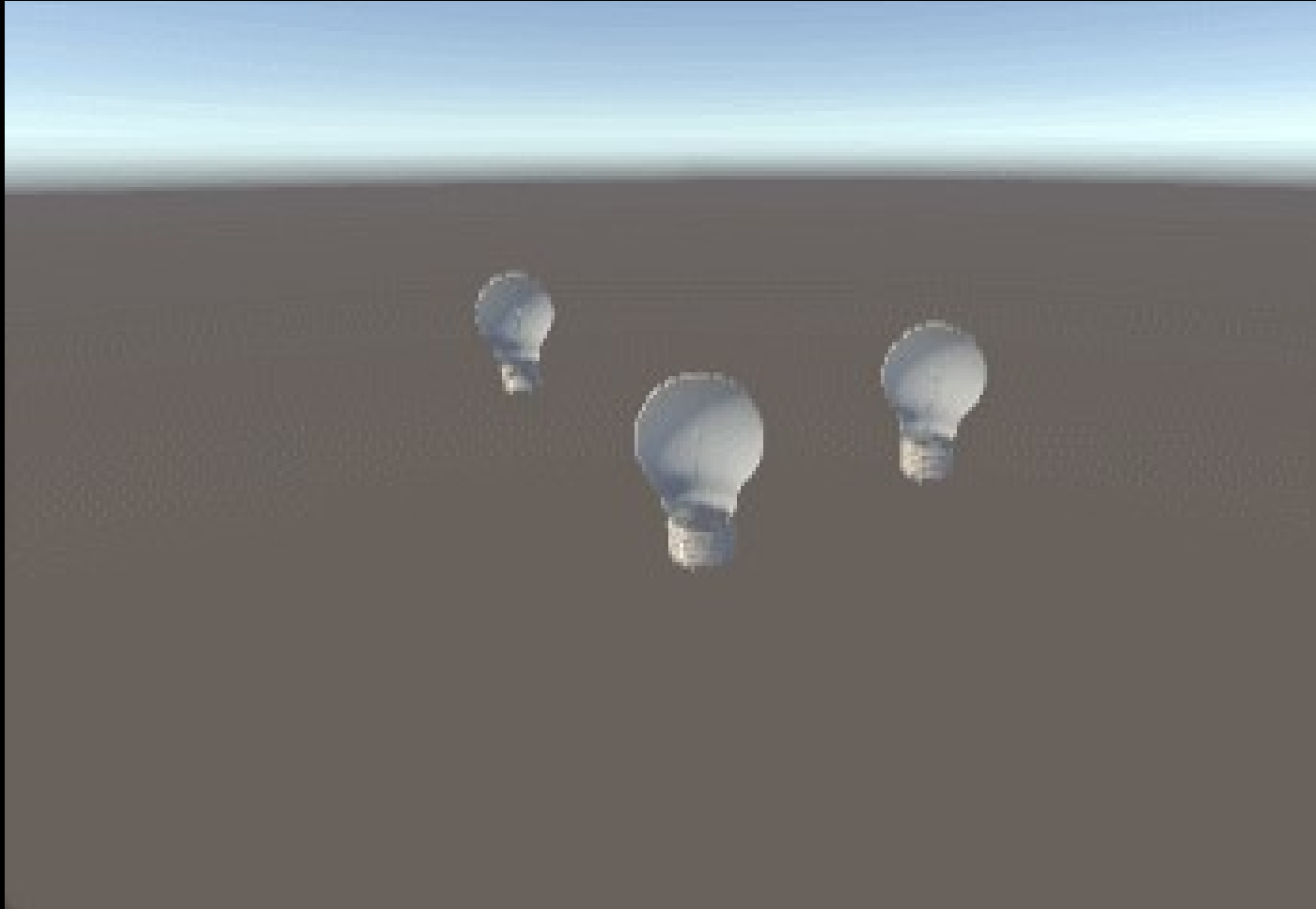
- Add a new script EX\_Bulb\_03.cs

```
1 using UnityEngine;
2
3 namespace FSMBulb
4 {
5     public class EX_Bulb_03 : MonoBehaviour {
6         public STATE state;
7         public Material matNormal;
8         public Material matBroken;
9
10        private Light pLight;
11        private MeshRenderer meshRenderer;
12        [SerializeField]
13        private int hp;
14        private bool triggerEnter;
15
16        void Start() {
17            pLight = GetComponentInChildren<Light>();
18            meshRenderer = GetComponent<MeshRenderer>();
19            triggerEnter = false;
20            hp = 3;
21        }
22    }
```

```
23
24 void Update() {
25     switch (state) {
26         case STATE.ON:
27             if (triggerEnter) {
28                 ChangeMaterial(matNormal);
29                 pLight.enabled = true;
30             }
31             break;
32         case STATE.OFF:
33             if (triggerEnter) {
34                 ChangeMaterial(matNormal);
35                 pLight.enabled = false;
36             }
37             break;
38         case STATE.BROKEN:
39             if (triggerEnter) {
40                 ChangeMaterial(matBroken);
41                 pLight.enabled = false;
42                 print("Broken");
43             }
44             break;
45         default:
46             break;
47     }
48     triggerEnter = false;
49 }
```

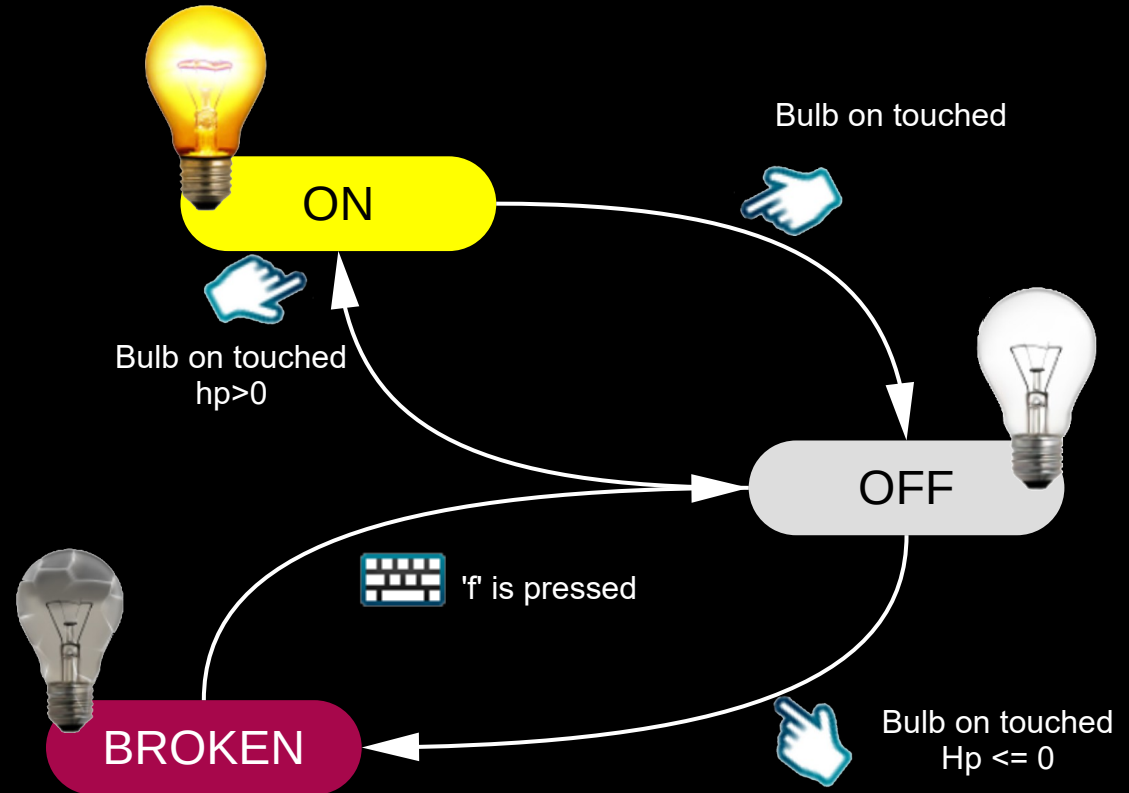
```
50 private void OnMouseDown() {  
51     if (state == STATE.ON) {  
52         GoToState(STATE.OFF);  
53     }  
54     else if (state == STATE.OFF) {  
55         if (hp > 0) {  
56             hp -= 1;  
57             GoToState(STATE.ON);  
58         }  
59         else {  
60             GoToState(STATE.BROKEN);  
61         }  
62     }  
63 }  
64  
65 private void GoToState(STATE targetState) {  
66     state = targetState;  
67     triggerEnter = true;  
68 }  
69  
70 private void ChangeMaterial(Material targetMat)  
71     Material[] mats = meshRenderer.materials;  
72     mats[0] = targetMat;  
73     meshRenderer.materials = mats;  
74 }  
75 }  
76  
77 }
```

- Create multiple bulbs and test them.



# EX: Press 'f' to Fix All Broken Bulbs

- Press 'f' to fix all broken bulbs
  - Create a new transition from BROKEN to OFF. The condition for the transition should be: when 'f' is pressed.
  - Refresh their hp to 3 for all the fixed bulb.



- Add a new script EX\_Bulb\_04.cs and copy the codes from last exercise.
- Go to the bottom of Update and add codes here:
  - Detect whether 'f' is just pressed. ❶
  - If true, change state to OFF and refresh hp.

```
43         case STATE.BROKEN:
44             if (triggerEnter)
45             {
46                 ChangeMaterial(matBroken);
47                 pLight.enabled = false;
48                 print("Broken");
49             }
50             break;
51         default:
52             break;
53     }
54     triggerEnter = false;
55
56     ❶ if (Input.GetKeyDown("f")) {
57         if (state == STATE.BROKEN) {
58             hp = 3;
59             GoToState(STATE.OFF);
60         }
61     }
62 }
63
64 private void OnMouseDown()
65 {
66     if (state == STATE.ON)
67     {
68         GoToState(STATE.OFF);
69     }
```





# 互動程式設計III

Interactive Programming Design Integration

- **Finite State Machine**
- **Switch Type**
- **Design Pattern Type**
-

# What is a State Pattern

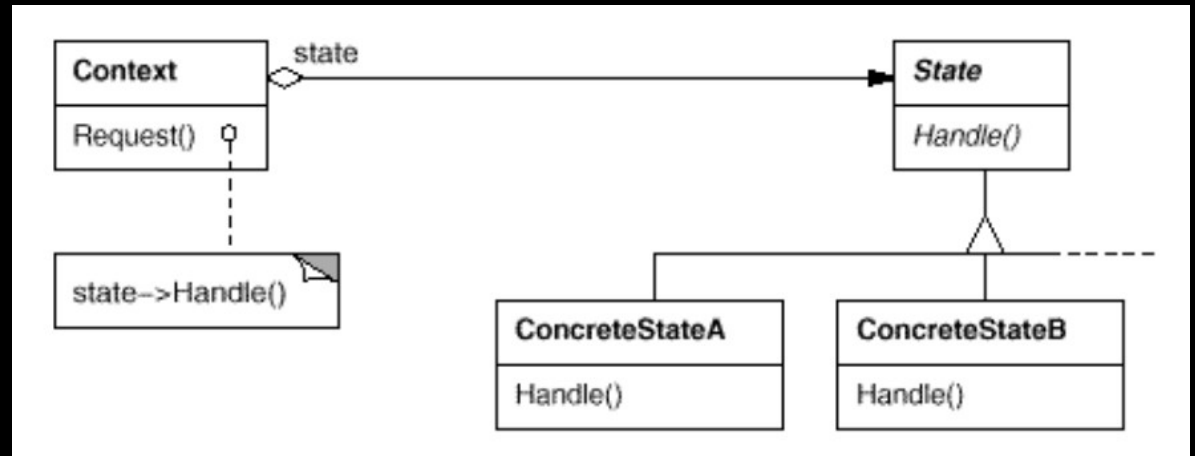
- Introduction to State Pattern
  - Definition: The State Pattern is a behavioral design pattern that allows an object to alter its behavior when its internal state changes. The object will appear to change its class dynamically by switching between state-specific behaviors.
  - Purpose: Encourages the Open/Closed Principle by allowing behavior changes without altering the client code. State transitions are managed within the state objects, improving code structure and flexibility.
  - Use Case: Useful when an object's behavior depends on its state and must change its behavior at runtime based on that state.
- Key Components of State Pattern
  - State Interface: Defines a common interface for all states.
  - Concrete States: Implement specific behavior for each state.
  - Context: Holds a reference to a state object and delegates behavior to the current state.
- Benefits of State Pattern
  - Open/Closed Principle: New states can be added without modifying existing code.
  - Code Maintainability: Each state is isolated in a separate class, improving readability and maintainability.
  - Runtime Behavior Flexibility: The behavior of the object changes dynamically by switching states during runtime.

# From GoF

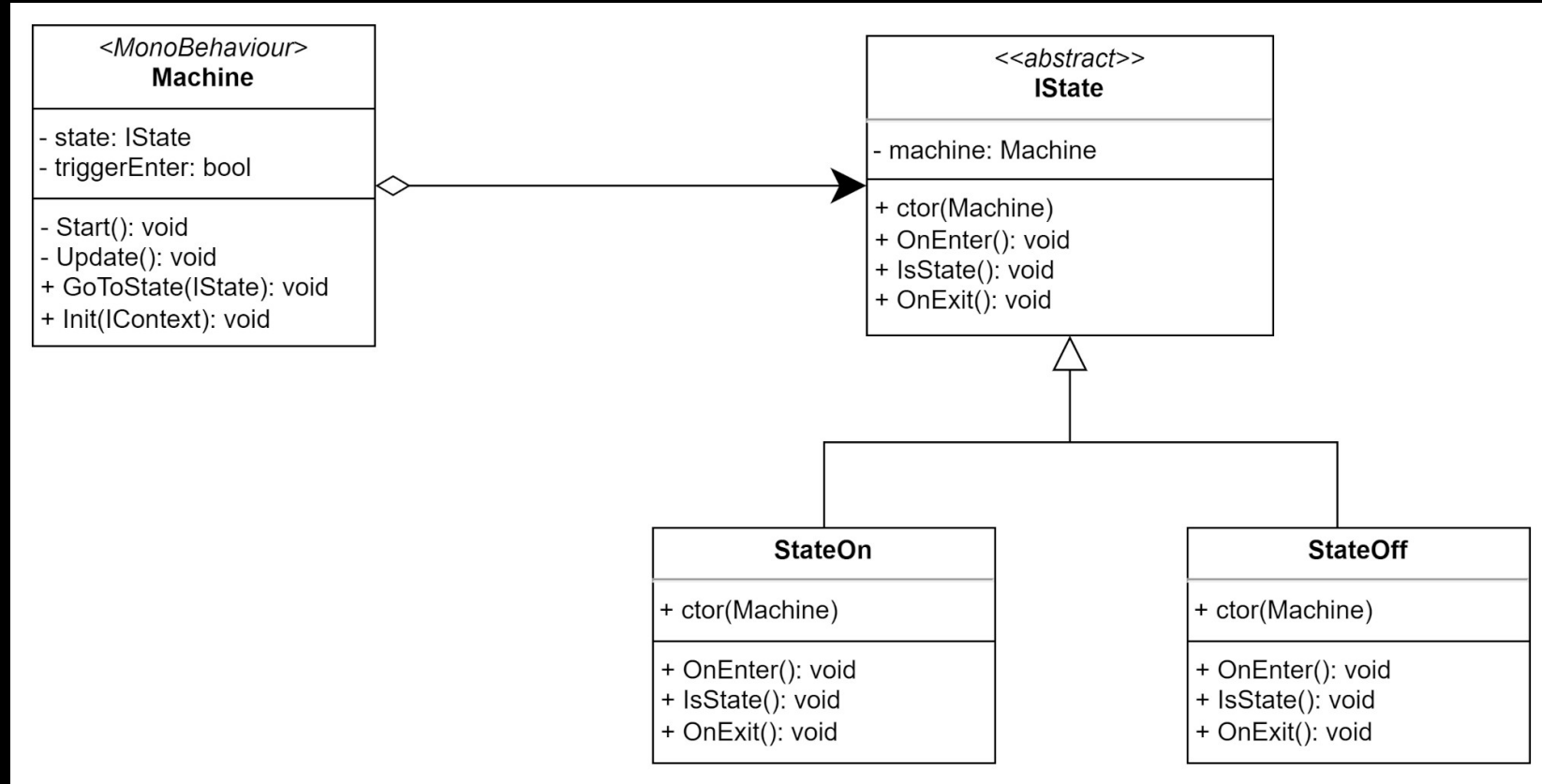
- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Apply the Strategy pattern when
  - An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
  - Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects.

# Class Diagram for State Pattern

- Context
  - defines the interface of interest to clients.
  - maintains an instance of a ConcreteState subclass that defines the current state.
- State
  - defines an interface for encapsulating the behavior associated with a particular state of the context.
- ConcreteState subclasses
  - each subclass implements a behavior associated with a state of the Context.



- Our 1<sup>st</sup> FSM system.



# EX: Basic Framework 01

- In this exercise, we are going to implement one empty state which is StateOn. And then load the state in FSM machine.
- IState

```
1      using UnityEngine;
2
3      namespace FSMBulb
4      {
5          public abstract class IState
6          {
7              protected Machine machine;
8
9              public IState(Machine _machine)
10             {
11                 machine = _machine;
12                 Debug.Log("state created.");
13             }
14
15             public abstract void OnEnter();
16             public abstract void IsState();
17             public abstract void OnExit();
18         }
19     }
```



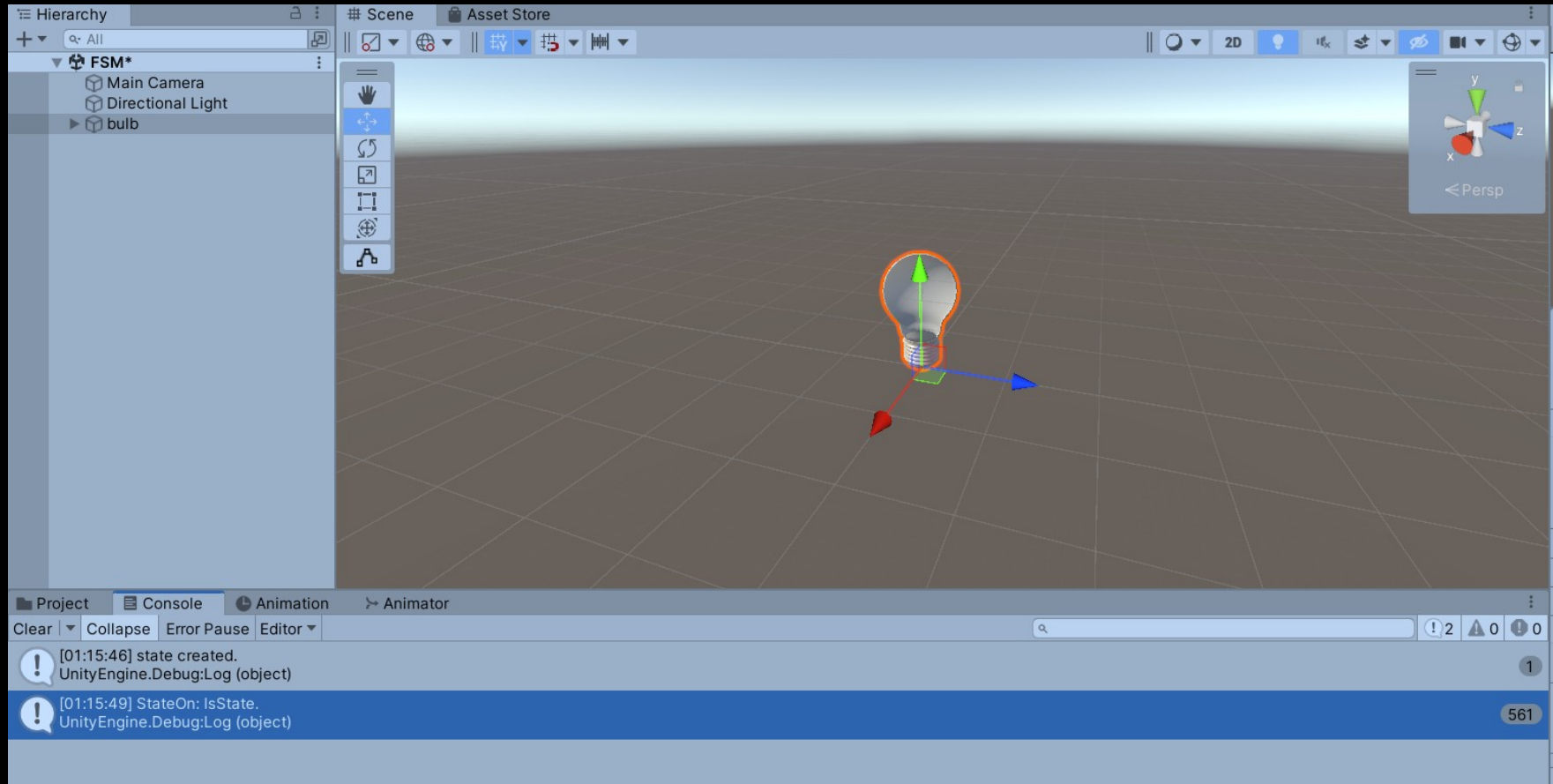
- StateOn

```
1  using UnityEngine;
2
3  namespace FSMBulb
4  {
5      public class StateOn : IState
6      {
7          public StateOn(Machine machine) : base(machine) { }
8
9          public override void OnEnter() {
10             Debug.Log("StateOn: OnEnter.");
11         }
12
13         public override void IsState() {
14             Debug.Log("StateOn: IsState.");
15         }
16
17         public override void OnExit() {
18             Debug.Log("StateOn: OnExit.");
19         }
20     }
21 }
```

- Machine

```
1  using UnityEngine;
2
3  namespace FSMBulb
4  {
5      public class Machine : MonoBehaviour
6      {
7          IState state;
8
9          void Start()
10         {
11             state = new StateOn(this);
12         }
13
14         void Update()
15         {
16             state.IsState();
17         }
18     }
19 }
```

- Add a Machine to the bulb gameObject. Please do remember to remove all other previous custom scripts on the bulb gameObject.
- Run the game. You should be able to see the console logs from StateOn.



## EX: Basic Framework 02

- In this exercise, we are going to add another state which is StateOff. And we are going to toggle the state whenever 't' key is pressed.
- !important: putting a explicit key detection in concrete state classes is not a good idea. Because state should not deal with REAL peripherals directly. Later in this class we are going to write a Input Manager class to translate real key strokes to abstract input events(or you may call it actions).

- StateOn

```
1  using UnityEngine;
2
3  namespace FSMBulb
4  {
5      public class StateOn : IState
6      {
7          public StateOn(Machine machine) : base(machine) { }
8
9          public override void OnEnter() {
10              Debug.Log("StateOn: OnEnter.");
11          }
12
13          public override void IsState() {
14              if (Input.GetKeyDown("t"))
15              {
16                  machine.GoToState(new StateOff(machine));
17              }
18              Debug.Log("StateOn: IsState.");
19          }
20
21          public override void OnExit() {
22              Debug.Log("StateOn: OnExit.");
23          }
24      }
25  }
```

- StateOff

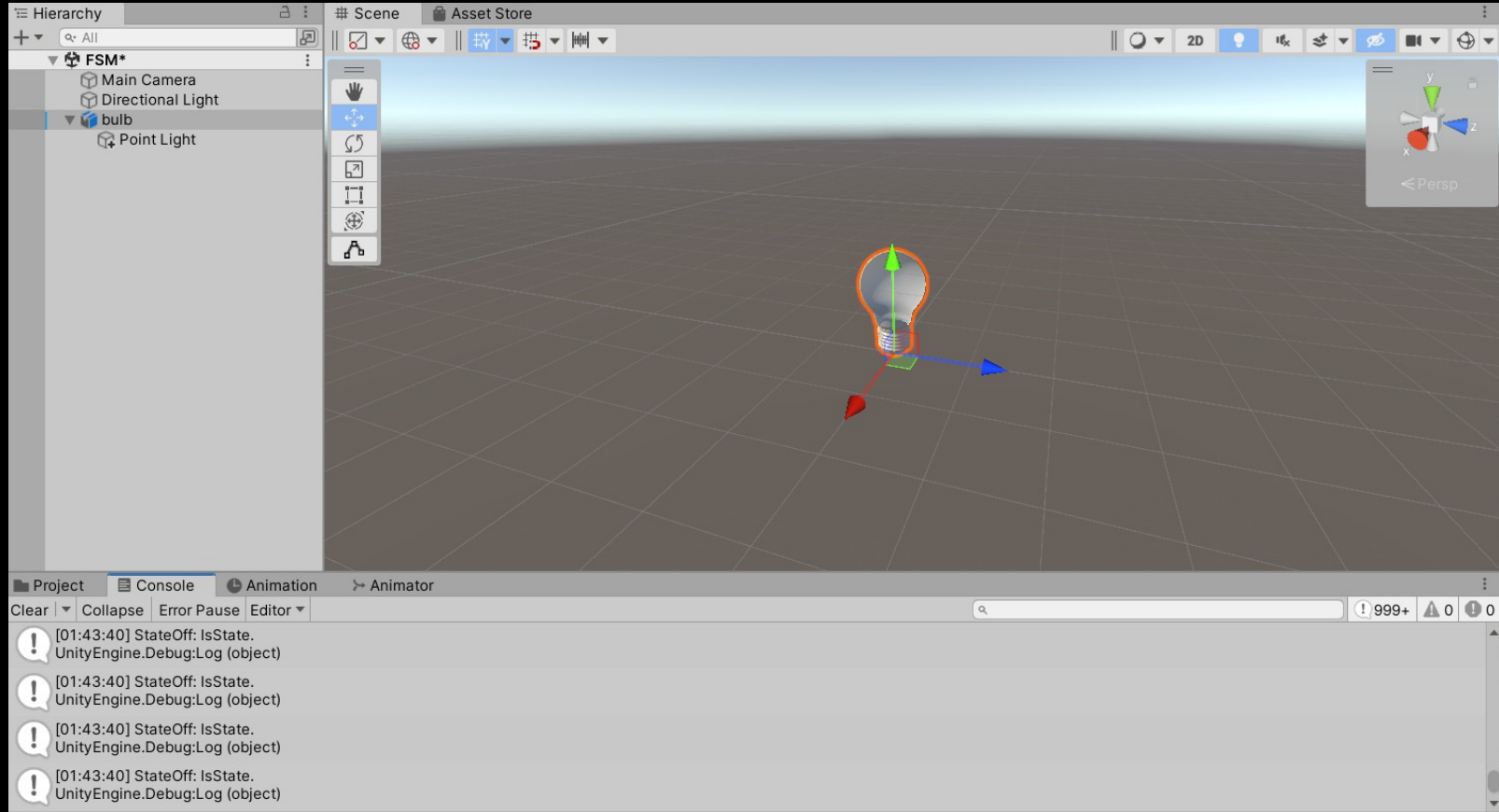
```
1  using UnityEngine;
2
3  namespace FSMBulb
4  {
5      public class StateOff : IState
6      {
7          public StateOff(Machine machine) : base(machine) { }
8
9          public override void OnEnter() {
10              Debug.Log("StateOff: OnEnter.");
11          }
12
13          public override void IsState() {
14              if (Input.GetKeyDown("t"))
15              {
16                  machine.GoToState(new StateOn(machine));
17              }
18              Debug.Log("StateOff: IsState.");
19          }
20
21          public override void OnExit() {
22              Debug.Log("StateOff: OnExit.");
23          }
24      }
25  }
```



- Machine

```
1  using UnityEngine;
2
3  namespace FSMBulb {
4      public class Machine : MonoBehaviour {
5          private IState state;
6          private bool triggerEnter;
7
8          void Start() {
9              //state = new StateOn(this);
10             GoToState(new StateOn(this));
11         }
12
13         void Update() {
14             if (triggerEnter) {
15                 state?.OnEnter();
16                 triggerEnter = false;
17             }
18             state?.IsState();
19         }
20
21         public void GoToState(IState targetState) {
22             state?.OnExit();
23             state = targetState;
24             triggerEnter = true;
25         }
26     }
27 }
```

- Press 't' to toggle state.
- See if IsState log is correctly shown.
- Then comment out the log of IsState. See if OnEnter/OnExit works properly.

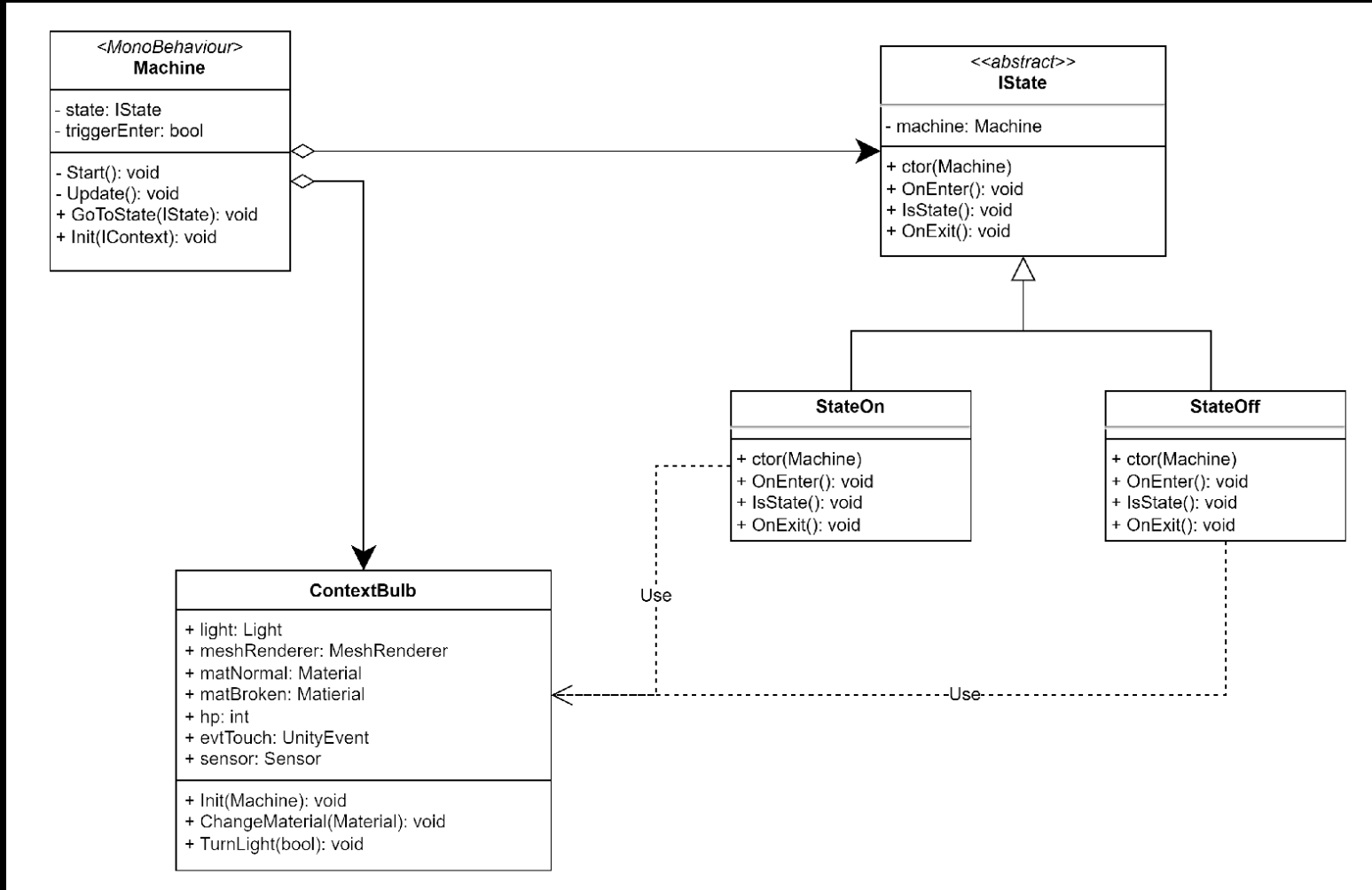


# EX: Basic Framework 03

- Add a Context class to save all dependencies.
- Implement gameObject/Component dependent methods for On/Off states.

```
1  using UnityEngine;
2
3  namespace FSMBulb
4  {
5      [System.Serializable]
6      public class Context
7      {
8          public Light light;
9          public MeshRenderer meshRenderer;
10         public Material matNormal;
11         public Material matBroken;
12
13         public void Init(Machine machine) {
14             light = machine.GetComponentInChildren<Light>();
15             meshRenderer = machine.GetComponent<MeshRenderer>();
16             machine.GoToState(new StateOn(machine));
17         }
18
19         public void ChangeMaterial(Material targetMat) {
20             Material[] mats = meshRenderer.materials;
21             mats[0] = targetMat;
22             meshRenderer.materials = mats;
23         }
24
25         public void TurnLight(bool value) {
26             light.enabled = value;
27         }
28     }
29 }
```

- Our 2<sup>nd</sup> FSM system.



- Our machine should have one context instance and initialize it in Start().
- Notice that we push all the game object/component dependencies to context.

```
1  using UnityEngine;
2
3  namespace FSMBulb {
4      public class Machine : MonoBehaviour {
5          public Context c = new Context();
6
7          private IState state;
8          private bool triggerEnter;
9
10         void Start() {
11             c.Init(this);
12         }
13
14         void Update() {
15             if (triggerEnter) {
16                 state?.OnEnter();
17                 triggerEnter = false;
18             }
19             state?.IsState();
20         }
21     }
22 }
```

- Toggle the light to correct visual representations in the OnEnter of On and Off states.

```
1  using UnityEngine;
2
3  namespace FSMBulb
4  {
5      public class StateOn : IState
6      {
7          public StateOn(Machine machine) : base(machine) { }
8
9          public override void OnEnter() {
10              machine.c.TurnLight(true);
11              //Debug.Log("StateOn: OnEnter.");
12          }
13      }
```

```
1  using UnityEngine;
2
3  namespace FSMBulb
4  {
5      public class StateOff : IState
6      {
7          public StateOff(Machine machine) : base(machine) { }
8
9          public override void OnEnter() {
10              machine.c.TurnLight(false);
11              Debug.Log("StateOff: OnEnter.");
12          }
13      }
```

- Run the game. Type 't' to test.



## EX: Basic Framework 04

- Now we are going to implement StateBroken.
- Add a int member variable to Context. And initialize it to 3 in Init().

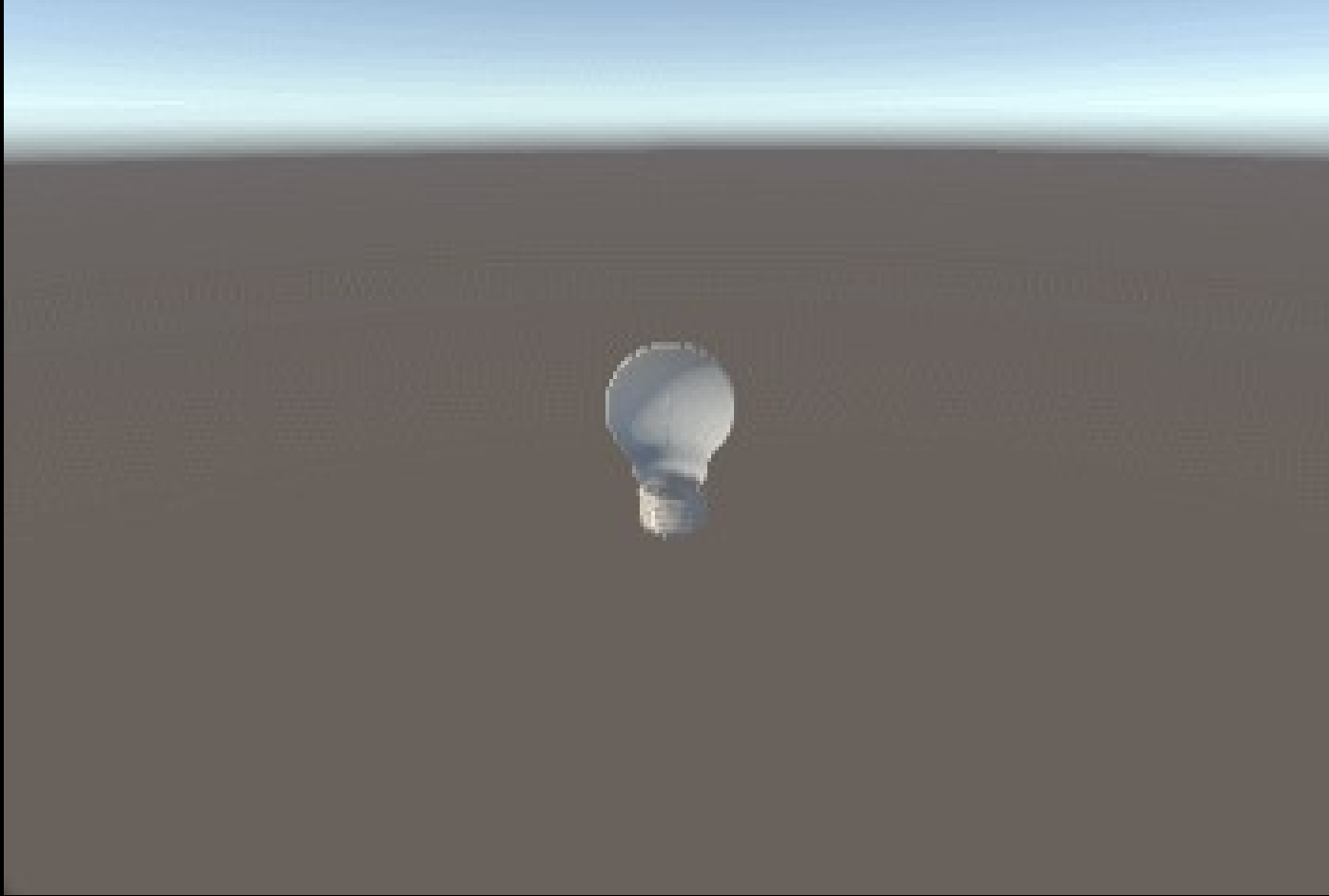
```
1  using UnityEngine;
2
3  namespace FSMBulb
4  {
5      [System.Serializable]
6      public class Context
7      {
8          public Light light;
9          public MeshRenderer meshRenderer;
10         public Material matNormal;
11         public Material matBroken;
12         public int hp;
13
14         public void Init(Machine machine) {
15             light = machine.GetComponent<Light>();
16             meshRenderer = machine.GetComponent<MeshRenderer>();
17             machine.GoToState(new StateOff(machine));
18             hp = 3;
19         }
20     }
21 }
```



- Change the trigger codes in IsState of StateOff:

```
1  using UnityEngine;
2
3  namespace FSMBulb
4  {
5      public class StateOff : IState
6      {
7          public StateOff(Machine machine) : base(machine) { }
8
9          public override void OnEnter() {
10              machine.c.TurnLight(false);
11              Debug.Log("StateOff: OnEnter.");
12          }
13
14          public override void IsState() {
15              if (Input.GetKeyDown("t")) {
16                  if (machine.c.hp > 0) {
17                      machine.c.hp--;
18                      machine.GoToState(new StateOn(machine));
19                  }
20                  else {
21                      machine.GoToState(new StateBroken(machine));
22                  }
23              }
24              Debug.Log("StateOff: IsState.");
25          }
26      }
```

- Run the game. Type 't' to test.

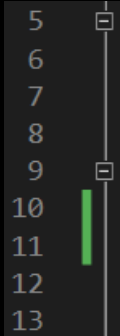


# EX: Basic Framework 05

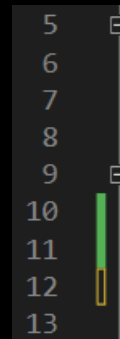
- Now we are going to implement the press 'r' to repair function.
- 

```
1  using UnityEngine;
2
3  namespace FSMBulb
4  {
5      public class StateBroken : IState
6      {
7          public StateBroken(Machine machine) : base(machine) { }
8
9          public override void OnEnter() {
10              machine.c.TurnLight(false);
11              machine.c.ChangeMaterial(machine.c.matBroken);
12              Debug.Log("StateBroken: OnEnter.");
13          }
14
15          public override void IsState() {
16              if (Input.GetKeyDown("r")) {
17                  machine.c.hp = 3;
18                  machine.GoToState(new StateOff(machine));
19              }
20              Debug.Log("StateBroken: IsState.");
21          }
22
23          public override void OnExit() {
24              Debug.Log("StateBroken: OnExit.");
25          }
26      }
27  }
```

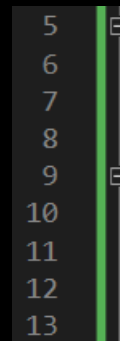
- Make sure all the states update its visual representation in OnEnter() correctly.
  - Light?
  - Material?



```
5 public class StateOn : IState
6 {
7     public StateOn(Machine machine) : base(machine) { }
8
9     public override void OnEnter() {
10         machine.c.TurnLight(true);
11         machine.c.ChangeMaterial(machine.c.matNormal);
12         //Debug.Log("StateOn: OnEnter.");
13     }
```

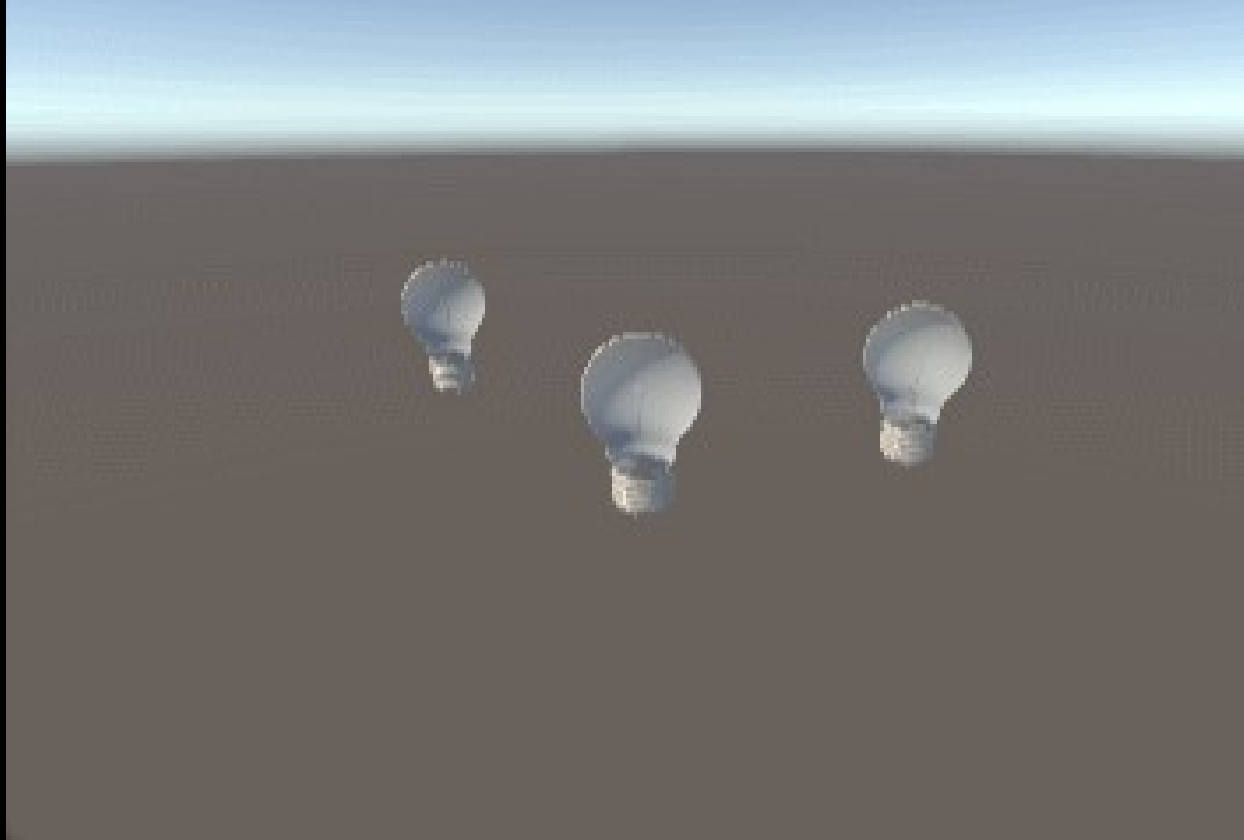


```
5 public class StateOff : IState
6 {
7     public StateOff(Machine machine) : base(machine) { }
8
9     public override void OnEnter() {
10         machine.c.TurnLight(false);
11         machine.c.ChangeMaterial(machine.c.matNormal);
12         //Debug.Log("StateOff: OnEnter.");
13     }
```



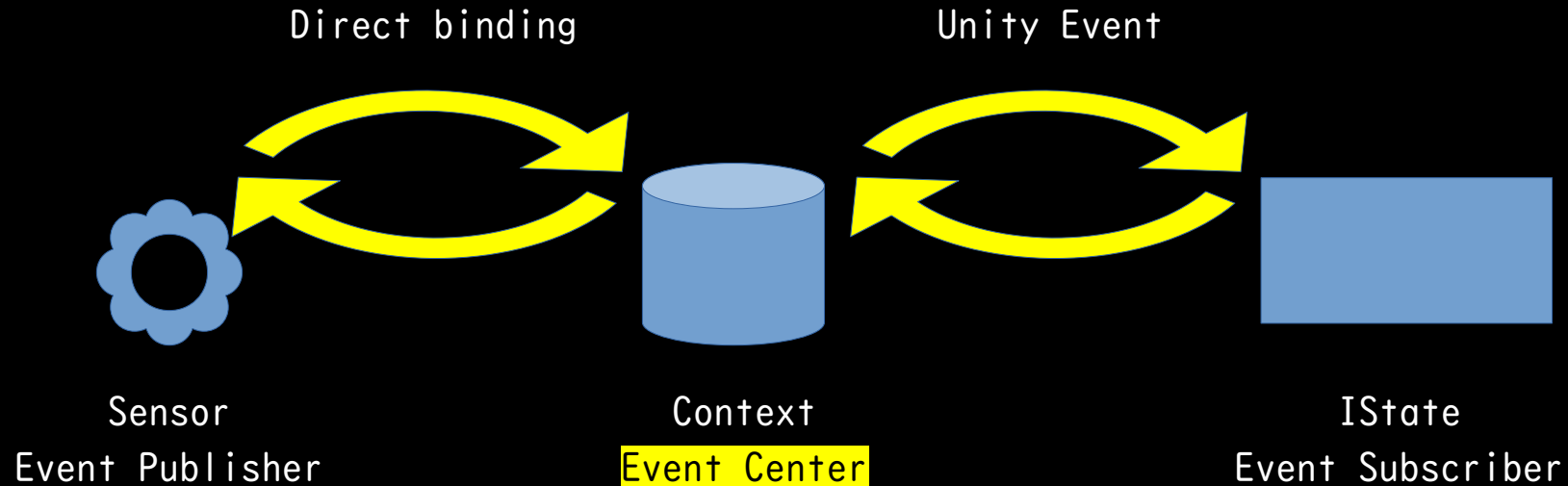
```
5 public class StateBroken : IState
6 {
7     public StateBroken(Machine machine) : base(machine) { }
8
9     public override void OnEnter() {
10         machine.c.TurnLight(false);
11         machine.c.ChangeMaterial(machine.c.matBroken);
12         //Debug.Log("StateBroken: OnEnter.");
13     }
```

- So simple! But...how to enable touch to toggle???



## EX: Basic Framework 06

- The `OnMouseDown()` API belongs to `MonoBehaviour` class. So we plan to create a dedicated `Sensor` which inherit from `MonoBehaviour` and attached to `bulb gameObject`.
- 

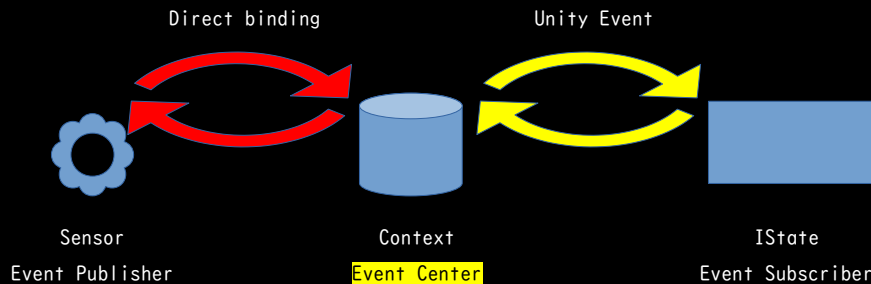


- Sensor

```
1  using UnityEngine;
2
3  namespace FSMBulb
4  {
5      public class Sensor : MonoBehaviour
6      {
7          public Context c;
8
9          public void OnMouseDown()
10         {
11             c?.evtTouch?.Invoke();
12         }
13     }
14 }
```

- Context

- Get sensor through machine.
  - Actually context needs machine to do all the MonoBehaviour related work. But machine needs to be decoupled from context and only focusing on abstract FSM operations.
- It is context to be responsible for doing the direct **Double binding** work.
  - Context knows sensor
  - Sensor knows context



```
1  using UnityEngine;
2  using UnityEngine.Events;
3
4  namespace FSMBulb
5  {
6      [System.Serializable]
7      public class Context
8      {
9          public Light light;
10         public MeshRenderer meshRenderer;
11         public Material matNormal;
12         public Material matBroken;
13         public int hp;
14         public UnityEvent evtTouch;
15         public Sensor sensor;
16
17         public void Init(Machine machine) {
18             light = machine.GetComponentInChildren<Light>();
19             meshRenderer = machine.GetComponent<MeshRenderer>();
20             machine.GoToState(new StateOff(machine));
21             hp = 3;
22             sensor = machine.GetComponent<Sensor>();
23             sensor.c = this;
24         }
25     }
```



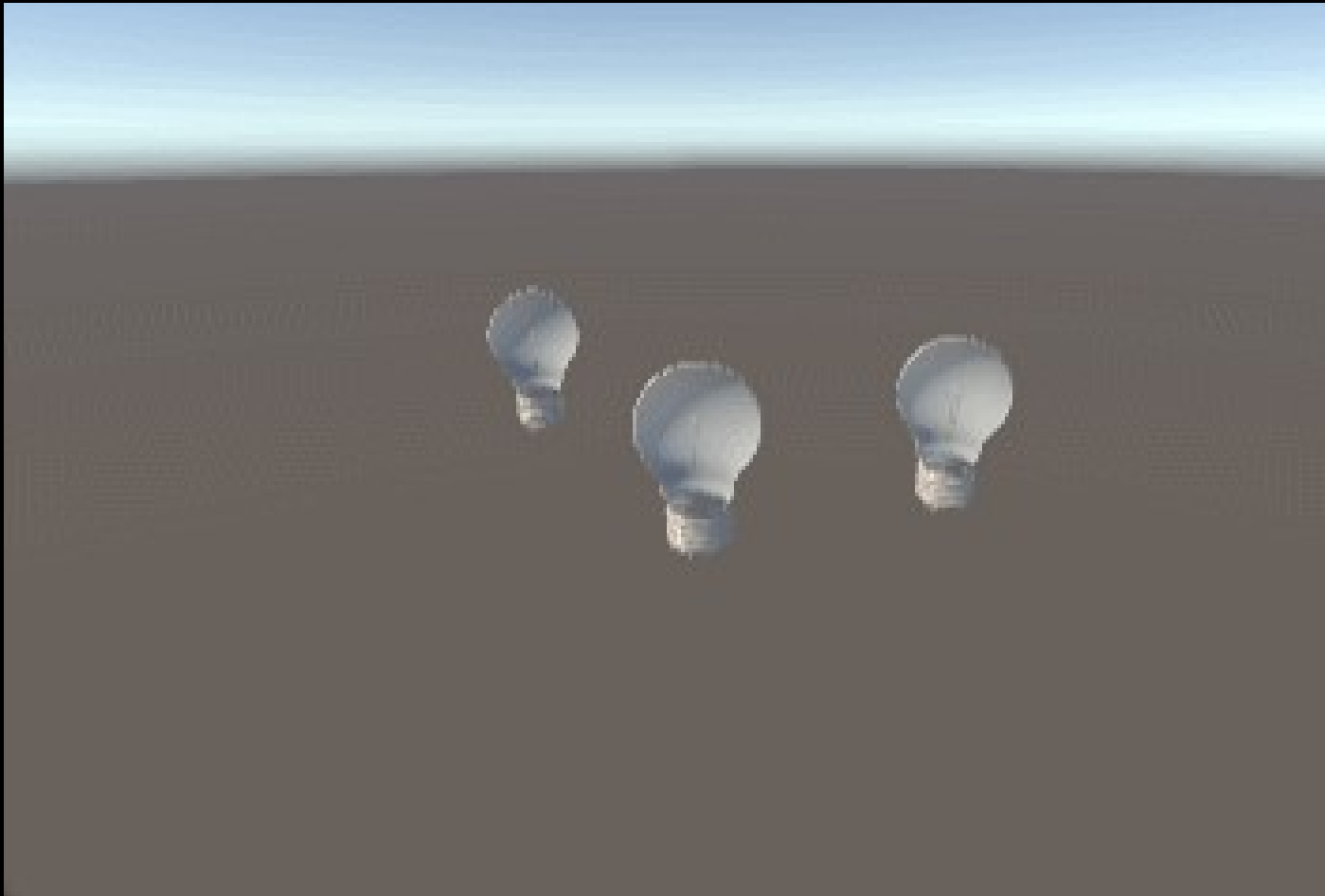
- StateOn
  - Set its own OnMouseDown().
  - Subscribe to Event Center in OnEnter()
  - Unsubscribe in OnExit()!!!

```
1  using UnityEngine;
2
3  namespace FSMBulb
4  {
5      public class StateOn : IState
6      {
7          public StateOn(Machine machine) : base(machine) { }
8
9          public override void OnEnter() {
10              machine.c.TurnLight(true);
11              machine.c.ChangeMaterial(machine.c.matNormal);
12              machine.c.evtTouch.AddListener(OnMouseDown);
13              //Debug.Log("StateOn: OnEnter.");
14          }
15
16          public override void IsState() {
17              //Debug.Log("StateOn: IsState.");
18          }
19
20          public override void OnExit() {
21              machine.c.evtTouch.RemoveListener(OnMouseDown);
22              //Debug.Log("StateOn: OnExit.");
23          }
24
25          public void OnMouseDown() {
26              machine.GoToState(new StateOff(machine));
27          }
28      }
29  }
```

- StateOff
  - Set its own OnMouseDown().
  - Subscribe to Event Center in OnEnter()
  - Unsubscribe in OnExit()!!!

```
1  using UnityEngine;
2
3  namespace FSMBulb
4  {
5      public class StateOff : IState
6      {
7          public StateOff(Machine machine) : base(machine) { }
8
9          public override void OnEnter() {
10              machine.c.TurnLight(false);
11              machine.c.ChangeMaterial(machine.c.matNormal);
12              machine.c.evtTouch.AddListener(OnMouseDown);
13              //Debug.Log("StateOff: OnEnter.");
14          }
15
16          public override void IsState() {
17              //Debug.Log("StateOff: IsState.");
18          }
19
20          public override void OnExit() {
21              machine.c.evtTouch.RemoveListener(OnMouseDown);
22              //Debug.Log("StateOff: OnExit.");
23          }
24
25          public void OnMouseDown() {
26              if (machine.c.hp > 0) {
27                  machine.c.hp--;
28                  machine.GoToState(new StateOn(machine));
29              }
30              else {
31                  machine.GoToState(new StateBroken(machine));
32              }
33          }
34      }
35  }
```

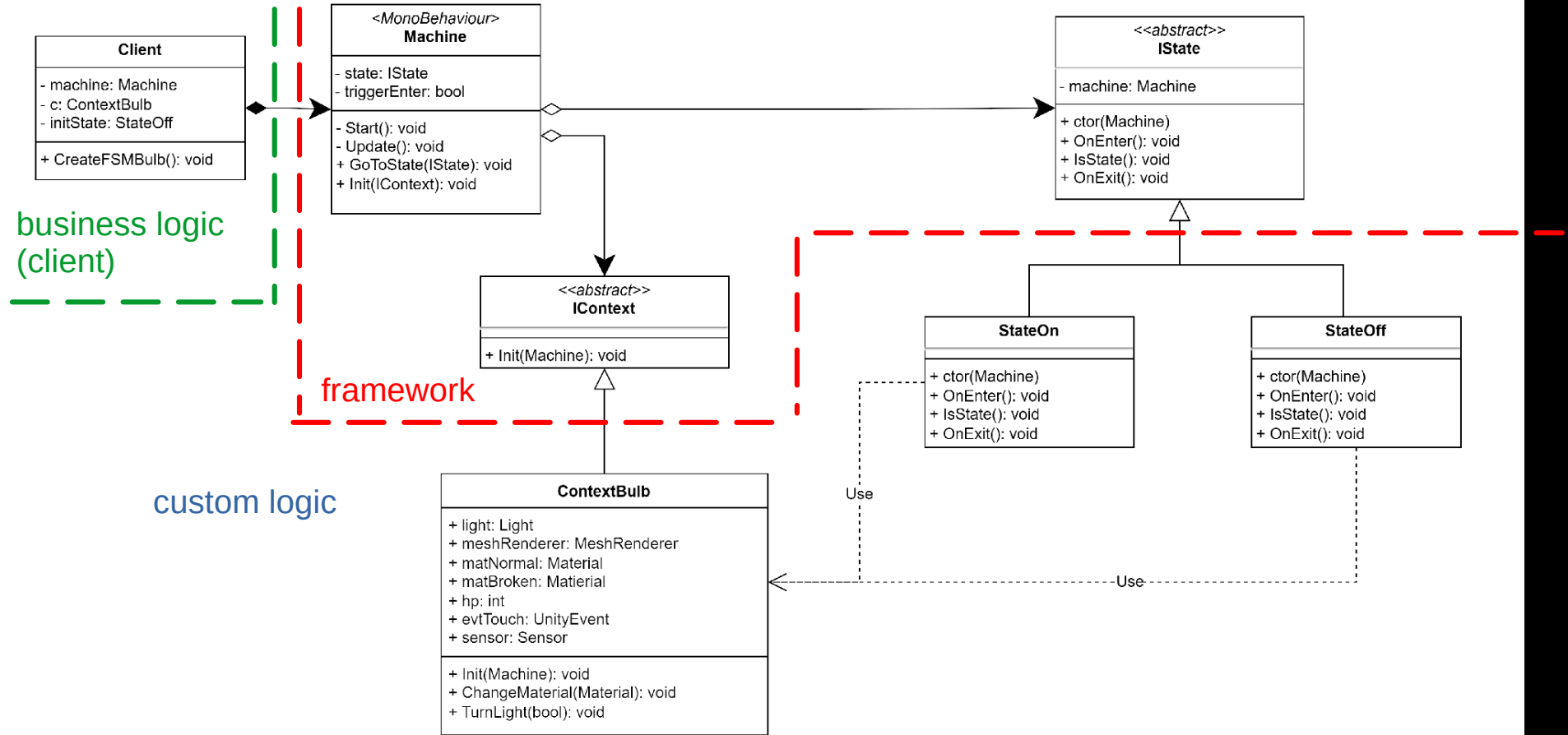
- Test it!



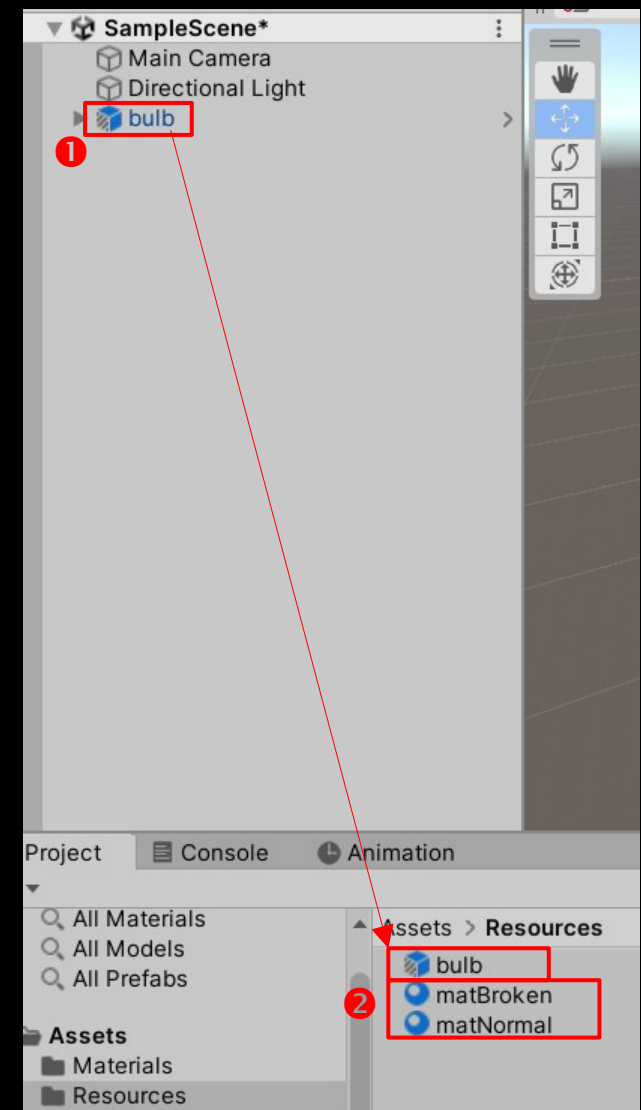
# EX: Advanced Framework

- We want to make the whole FSM design to be reusable.
  - Separate the design into 2 parts: fixed & customizable.

- Our 3<sup>rd</sup> FSM system.



- Drag you bulb in scene to the Resources folder. ❶
  - Files located in Resources folder can be loaded by calling `Resources.Load<T>()`
  - Make sure the light, sensor, machine, context are configured correctly.
- Also create (or move) `matNormal` and `matBroken` in Resources ❷, we are going to load these materials with C#.



- Create a **IContext** class ❶
  - Declare the Init() method.
- **ContextBulb**
  - Refactor the name of the original Context to be ContextBulb.
  - ContextBulb should inherit from IContext. ❷
  - Update Init() to load from external Resources folder. ❸
  - We also need to instantiate UnityEvent for evtTouch(previously editor did this instantiation for us). ❹

```

1 namespace FSMBulb
2 {
3     ❶ public abstract class IContext
4     {
5         public abstract void Init(Machine _machine);
6     }
7 }

```

```

1 using UnityEngine;
2 using UnityEngine.Events;
3
4 namespace FSMBulb
5 {
6     [System.Serializable] ❷
7     public class ContextBulb : IContext
8     {
9         public Light light;
10        public MeshRenderer meshRenderer;
11        public Material matNormal;
12        public Material matBroken;
13        public int hp;
14        public UnityEvent evtTouch;
15        public Sensor sensor;
16
17        public override void Init(Machine machine) {
18            light = machine.GetComponentInChildren<Light>();
19            meshRenderer = machine.GetComponent<MeshRenderer>();
20            machine.GoToState(new StateOff(machine));
21            hp = 3;
22            sensor = machine.GetComponent<Sensor>();
23            sensor.c = this;
24            ❸ matNormal = Resources.Load<Material>("matNormal");
25            matBroken = Resources.Load<Material>("matBroken");
26            ❹ evtTouch = new UnityEvent();
27        }
28
29        public void ChangeMaterial(Material targetMat) {
30            Material[] mats = meshRenderer.materials;
31            mats[0] = targetMat;
32            meshRenderer.materials = mats;
33        }
34
35        public void TurnLight(bool value) {
36            light.enabled = value;
37        }
38    }
39 }
40

```

- **Machine**
  - Change the type of c to IContent. ❶
  - Because we are not allowed to modify the constructor of a class derived from MonoBehaviour, so we create a Init() method to do all initialization tasks. ❷
  - It's the duty of client to inject all dependencies in Init().
    - So the client should organize an instance derived from IContent and pass it to Init().

```
1  using UnityEngine;
2
3  namespace FSMBulb
4  {
5      public class Machine : MonoBehaviour
6      {
7          public IContext c;
8          private bool triggerEnter;
9          private IState state;
10
11      ❷ public void Init(IContext _c)
12      {
13          c = _c;
14          c.Init(this);
15      }
16
17      private void Update()
18      {
19          if (triggerEnter)
20          {
21              state?.OnEnter();
22              triggerEnter = false;
23          }
24          state?.IsState();
25      }
26
27      public void GoToState(IState targetState)
28      {
29          state?.OnExit();
30          state = targetState;
31          triggerEnter = true;
32      }
33  }
34 }
```



- **StateOn** **StateOff** and **StateBroken**
  - Since now machine owns a IContext, the state logic has to convert the IContext to the concrete class Context+Bulb.
  - We keep a class member reference c to convert and save the Context+Bulb.

```
1 namespace FSMBulb
2 {
3     public class StateOn : IState
4     {
5         ContextBulb c;
6
7         public StateOn(Machine machine) : base(machine)
8         {
9             c = (ContextBulb)machine.c;
10        }
11
12        public override void OnEnter()
13        {
14            c.light.enabled = true;
15            c.ChangeMaterial(c.matNormal);
16            c.evtTouch.AddListener(OnMouseDown);
17        }
18
19        public override void IsState() { }
20
21        public override void OnExit()
22        {
23            c.evtTouch.RemoveListener(OnMouseDown);
24        }
25
26        public void OnMouseDown()
27        {
28            machine.GoToState(new StateOff(machine));
29        }
30    }
31 }
```

```

1 namespace FSMBulb
2 {
3     public class StateOff : IState
4     {
5         ContextBulb c;
6
7         public StateOff(Machine machine) : base(machine)
8         {
9             c = (ContextBulb)machine.c;
10        }
11
12        public override void OnEnter()
13        {
14            c.light.enabled = false;
15            c.ChangeMaterial(c.matNormal);
16            c.evtTouch.AddListener(OnMouseDown);
17        }
18
19        public override void IsState()
20        {
21        }
22
23        public override void OnExit()
24        {
25            c.evtTouch.RemoveListener(OnMouseDown);
26        }
27
28        public void OnMouseDown()
29        {
30            if (c.hp > 0)
31            {
32                c.hp--;
33                machine.GoToState(new StateOn(machine));
34            }
35            else
36            {
37                machine.GoToState(new StateBroken(machine));
38            }
39        }
40    }
41 }

```

```

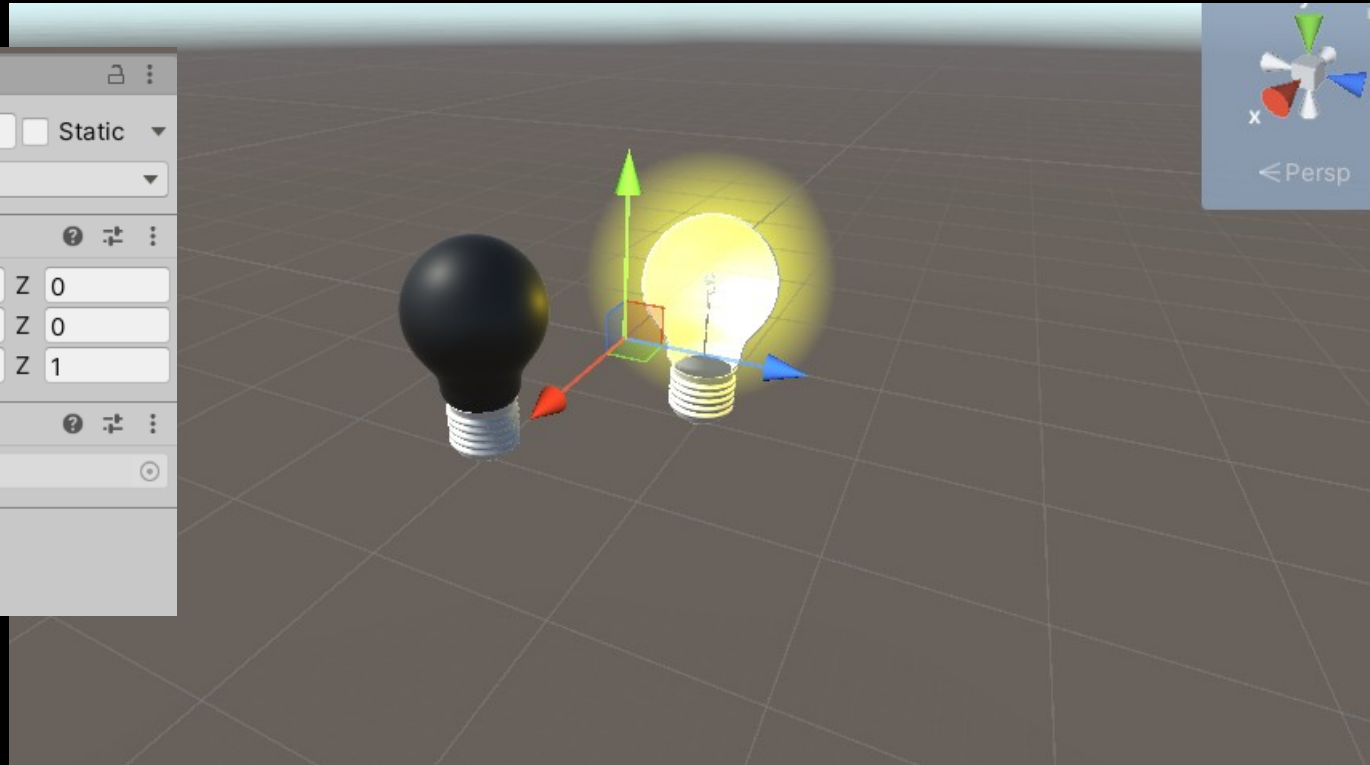
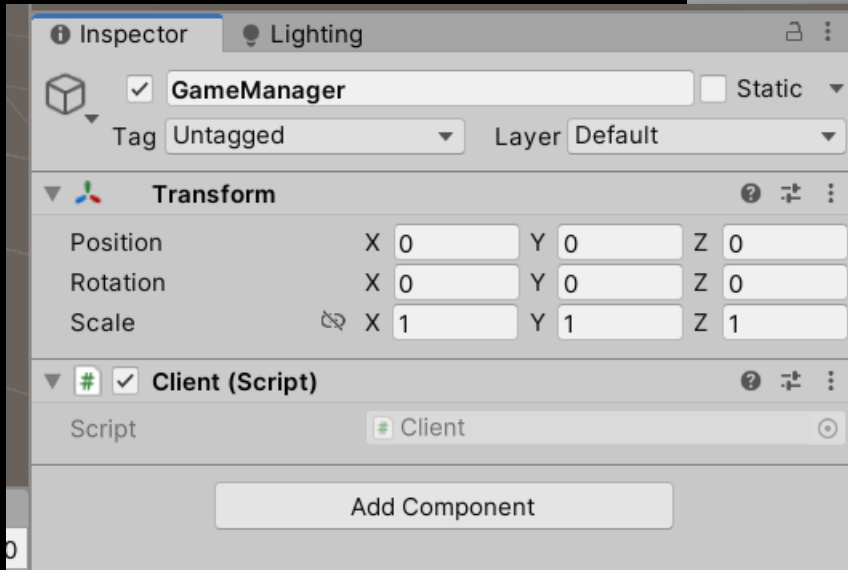
1 using UnityEngine;
2
3 namespace FSMBulb
4 {
5     public class StateBroken : IState
6     {
7         ContextBulb c;
8
9         public StateBroken(Machine machine) : base(machine)
10        {
11            c = (ContextBulb)machine.c;
12        }
13
14        public override void OnEnter()
15        {
16            c.light.enabled = false;
17            c.ChangeMaterial(c.matBroken);
18        }
19
20        public override void IsState()
21        {
22            if (Input.GetKeyDown("r"))
23            {
24                c.hp = 3;
25                machine.GoToState(new StateOff(machine));
26            }
27        }
28
29        public override void OnExit()
30        {
31        }
32    }
33 }

```

- Finally, the **Client** codes:
  - Using the FSMBulb namespace. ❶
  - Put all bulb generating codes into CreateFSMBulb. ❷
    - Requires a position for spawning the bulb. ❸
  - Create a ContextBulb and pass to machine.Init() ❹

```
1 using UnityEngine;
2 using UnityEngine.Events;
3 ❶ using FSMBulb;
4
5 public class Client : MonoBehaviour
6 {
7     void Start()
8     {
9         ❸ CreateFSMBulb(new Vector3(2, 0, 0));
10        CreateFSMBulb(new Vector3(1, 0, 1));
11    }
12
13    ❷ private void CreateFSMBulb(Vector3 pos)
14    {
15        GameObject prefabBulb = Resources.Load<GameObject>("bulb");
16
17        GameObject bulb = Instantiate(prefabBulb);
18        bulb.transform.position = pos;
19        Machine machine = bulb.GetComponent<Machine>();
20
21        ❹ ContextBulb c = new ContextBulb();
22        machine.Init(c);
23    }
24 }
```

- Create a gameObject called GameManager, and add a Client to it. Run the game and verify the functions.





# 互動程式設計III

Interactive Programming Design Integration

Q&A

# What is a Strategy Pattern

- Introduction to Strategy Pattern
  - Definition: The Strategy Pattern is a behavioral design pattern that allows you to define a family of algorithms, encapsulate each one, and make them interchangeable. It enables the algorithm to vary independently from the clients that use it.
  - Purpose: Promotes the Open/Closed Principle by enabling behavior to change without modifying the client code.
  - Use Case: Ideal when you have multiple ways of performing a task, and you need to switch between these algorithms at runtime.
- Key Components of Strategy Pattern
  - Strategy Interface: Defines a common interface for all algorithms.
  - Concrete Strategies: Implement specific algorithms.
  - Context: Holds a reference to a Strategy and uses it to perform operations.
- Benefits of Strategy Pattern
  - Open/Closed Principle: New strategies can be added without modifying existing code.
  - Separation of Concerns: Encapsulates different algorithms in separate classes.
  - Runtime Flexibility: Allows switching between strategies at runtime.

# From GoF

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Apply the Strategy pattern when
  - many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors. (The rules/framework to apply these behaviors are the same)
  - you need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms [H087].
  - an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific datastructures.
  - a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

# Class Diagram for Strategy Pattern

- Strategy
  - declares an interface (or abstract class) common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- ConcreteStrategy
  - implements the algorithm using the Strategy interface.
- Context
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.

