

Design Patterns

Produced
by

Eamonn de Leastar
edeleastar@wit.ie

Department of Computing, Maths & Physics
Waterford Institute of Technology

<http://www.wit.ie>

<http://elearning.wit.ie>



Waterford Institute of Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE



Door/Light Example

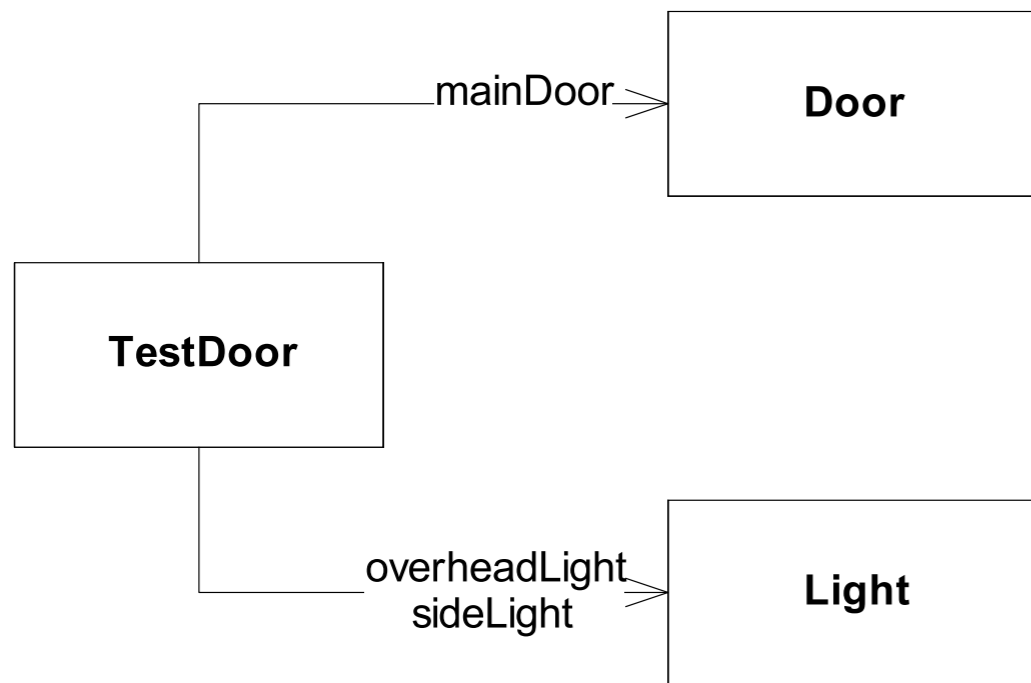
Motivational Example for Observer Pattern

Motivational Example

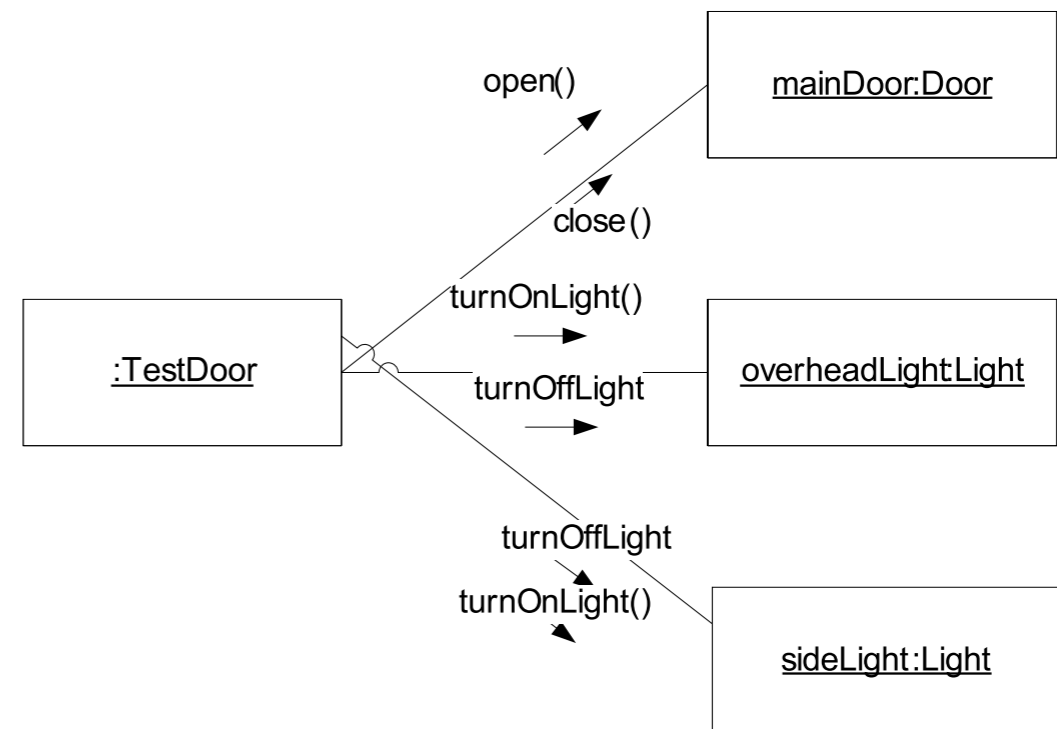
- In an embedded system a Door opening should trigger a Light to turn on
- The Door closing should turn the light off
- Explore the structure of various designs by building a series of test cases
- Six versions:
 - Version 0: No coupling between Door & Light
 - Version 1: Couple Door & Light; Enable door to activate multiple lights
 - Version 2: Introduce Cameras, to be also coupled to Door
 - Version 3: Decouple Light & Camera from Door by refactoring to use Observer Pattern
 - Version 4: Use `java.util` observer implementation
 - Version 5: Anonymous inner class idiom

Version 0

- Direct coupling between Door & Light



Class Diagram



Communication Diagram

0:Test

```
public class DoorTest
{
    private Door mainDoor;
    private Light overheadLight;
    private Light sideLight;

    @Before
    public void setUp() throws Exception
    {
        mainDoor = new Door("Main Door");
        overheadLight = new Light("Overhead Light");
        sideLight = new Light("Side Light");
    }

    @After
    public void tearDown() throws Exception
    {
        mainDoor = null;
        overheadLight = null;
        sideLight = null;
    }
}
```

```
@Test
public void testOpenClose()
{
    mainDoor.open();
    overheadLight.turnOnLight();
    sideLight.turnOnLight();

    assertTrue(overheadLight.getLightState());
    assertTrue(sideLight.getLightState());

    mainDoor.close();
    overheadLight.turnOffLight();
    sideLight.turnOffLight();

    assertFalse(overheadLight.getLightState());
    assertFalse(sideLight.getLightState());
}
}
```

0:Door

- Door is not aware of Light class

```
public class Door
{
    private String name;
    private boolean isOpen;

    public Door(String name)
    {
        this.name = name;
        isOpen = false;
    }

    public void open()
    {
        if (isOpen == false)
        {
            System.out.println("Opening " + name);
            isOpen = true;
        }
    }

    public void close()
    {
        if (isOpen == true)
        {
            System.out.println("Closing " + name);
            isOpen = false;
        }
    }
}
```

0:Light

- Light has no dependencies

```
public class Light
{
    private String name;
    private boolean lightOn;

    public Light(String nm)
    {
        name = nm;
        lightOn = false;
    }

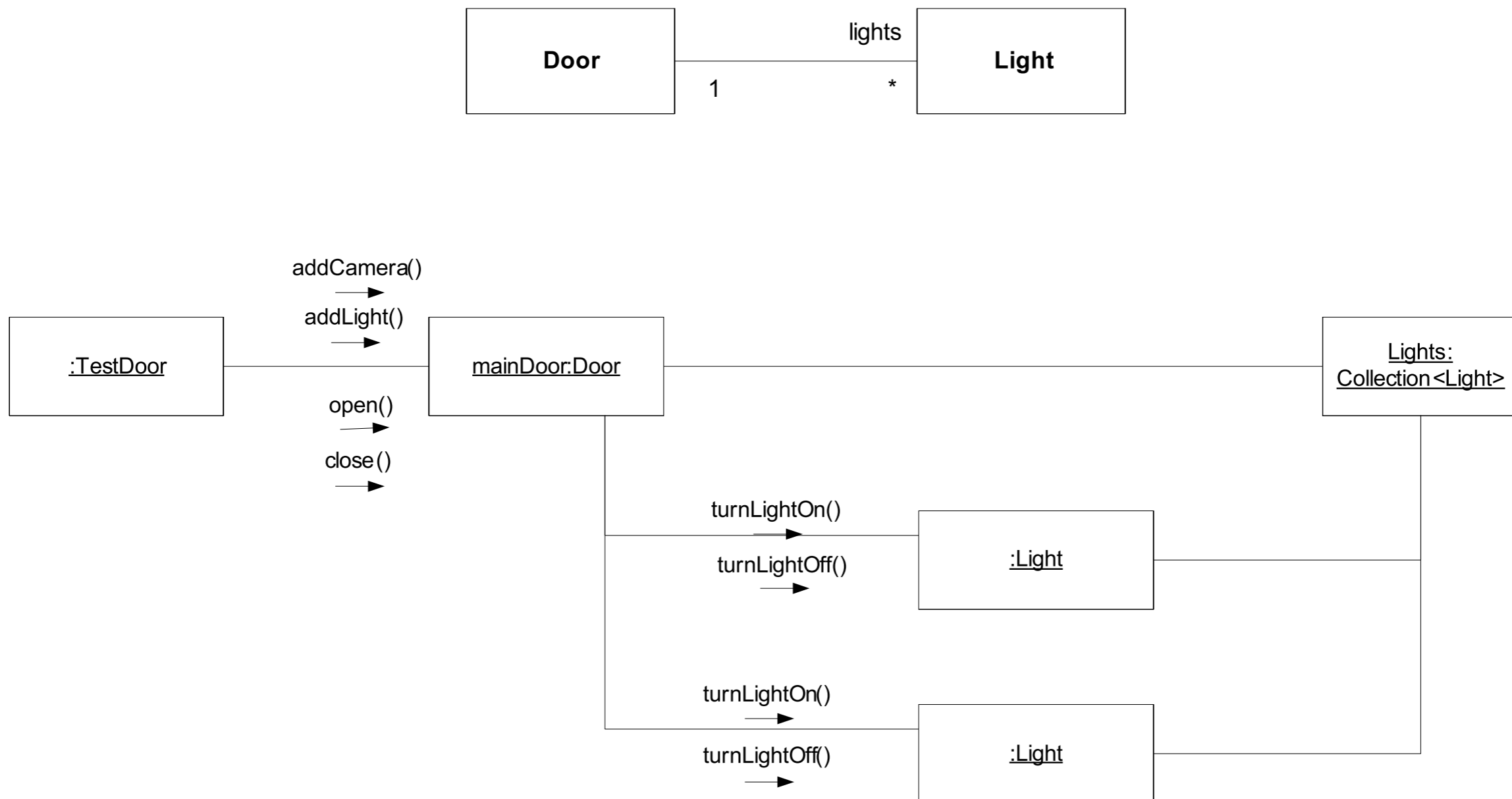
    public void turnOnLight()
    {
        if (!lightOn)
        {
            System.out.println("Turning on light" + name);
            lightOn = true;
            // activate light sensor
        }
    }

    public void turnOffLight()
    {
        if (lightOn)
        {
            System.out.println("Turning Off light" + name);
            lightOn = false;
            // deactivate light sensor
        }
    }

    public boolean getLightState()
    {
        return lightOn;
    }
}
```

Version 1

- Couple Door & Light; Enable door to control multiple lights



1:TestDoor

```
public class DoorTest
{
    private Door mainDoor;
    private Light overheadLight;
    private Light sideLight;

    @Before
    public void setUp() throws Exception
    {
        mainDoor = new Door("Main Door");
        overheadLight = new Light("Overhead Light");
        sideLight = new Light("Side Light");
    }

    @After
    public void tearDown() throws Exception
    {
        mainDoor = null;
        overheadLight = null;
        sideLight = null;
    }
}
```

```
@Test
public void testOpenClose()
{
    mainDoor.addLight(overheadLight);
    mainDoor.addLight(sideLight);

    mainDoor.open();
    assertTrue(overheadLight.getLightState());
    assertTrue(sideLight.getLightState());

    mainDoor.close();
    assertFalse(overheadLight.getLightState());
    assertFalse(sideLight.getLightState());

    mainDoor.open();
    assertTrue(overheadLight.getLightState());
    assertTrue(sideLight.getLightState());
}
```

1:TestDoor - using JMock

- Use the JMock libraries to verify that open/closing the door triggers an on/off call to the light object.

```
public class DoorTest
{
    private Door mainDoor;
    private Mockery context;

    @Before
    public void setUp() throws Exception
    {
        mainDoor = new Door("Main Door");
        context = new Mockery() {{
            setImposteriser(ClassImposteriser.INSTANCE);
        }};
    }

    @Test
    public void testOpenCloseMock()
    {
        final Light light = context.mock(Light.class);

        mainDoor.addLight(light);
        context.checking(new Expectations() {{
            one(light).turnOnLight();
        }});

        mainDoor.open();
        context.assertIsSatisfied();
        context.checking(new Expectations() {{
            one(light).turnOffLight();
        }});

        mainDoor.close();
        context.assertIsSatisfied();
    }
}
```

```

public class Door
{
    private String name;
    private boolean isOpen;
    private Collection<Light> lights;

    public Door(String name)
    {
        this.name = name;
        isOpen = false;
        lights = new HashSet<Light>();
    }

    public void open()
    {
        if (isOpen == false)
        {
            System.out.println("Opening " + name);
            isOpen = true;
            turnOnLights();
        }
    }

    public void close()
    {
        if (isOpen == true)
        {
            System.out.println("Closing " + name);
            isOpen = false;
            turnOffLights();
        }
    }
}

```

1:Door

```

    public void addLight(Light light)
    {
        lights.add(light);
    }

    private void turnOnLights()
    {
        for (Light light : lights)
        {
            light.turnOnLight();
        }
    }

    private void turnOffLights()
    {
        for (Light light : lights)
        {
            light.turnOffLight();
        }
    }
}

```

1:Light

- No change from previous version

```
public class Light
{
    private String name;
    private boolean lightOn;

    public Light(String nm)
    {
        name = nm;
        lightOn = false;
    }

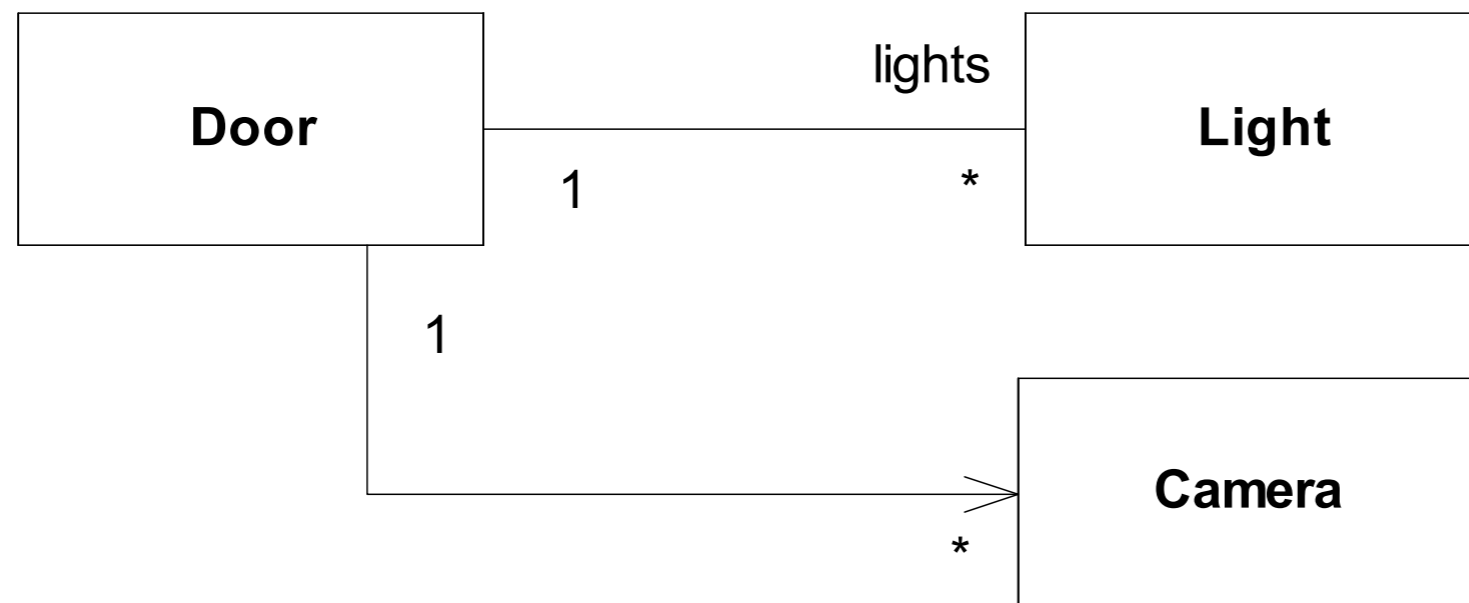
    public void turnOnLight()
    {
        if (!lightOn)
        {
            System.out.println("Turning on light" + name);
            lightOn = true;
            // activate light sensor
        }
    }

    public void turnOffLight()
    {
        if (lightOn)
        {
            System.out.println("Turning Off light" + name);
            lightOn = false;
            // deactivate light sensor
        }
    }

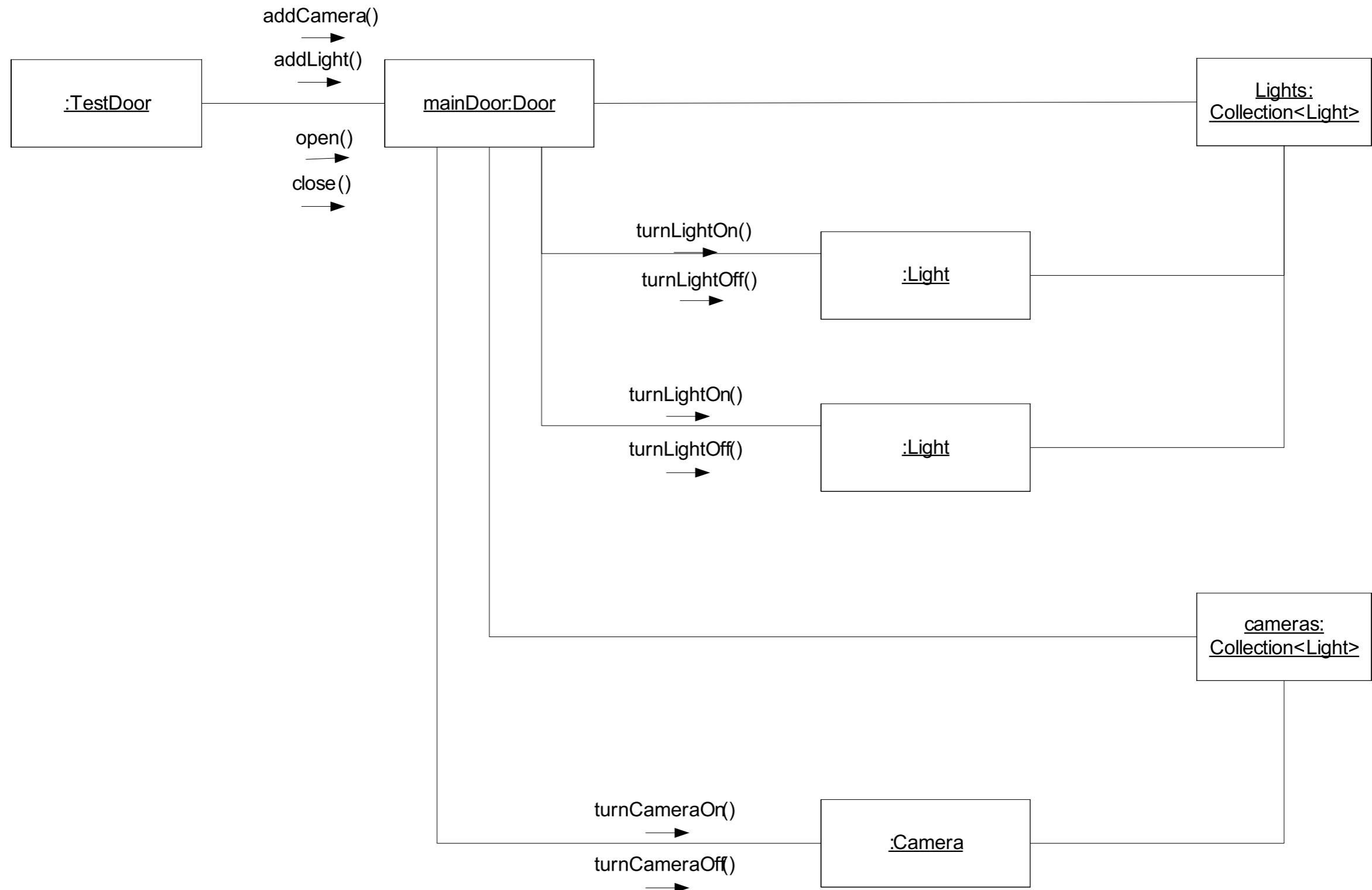
    public boolean getLightState()
    {
        return lightOn;
    }
}
```

Version 2: Introduce Camera

- Introduce Cameras, to be also coupled to Door



2: Communication Diagram



2:TestDoor

```
public class DoorTest
{
    private Door mainDoor;
    private Light light;
    private Camera camera;
    private Mockery context;

    @Before
    public void setUp() throws Exception
    {
        context = new Mockery() {{
            setImposteriser(ClassImposteriser.INSTANCE);
        }};
        light = context.mock(Light.class);
        camera = context.mock(Camera.class);

        mainDoor = new Door("Main Door");
        mainDoor.addLight(light);
        mainDoor.addCamera(camera);
    }

    @After
    public void tearDown() throws Exception
    {
        mainDoor = null;
    }
}
```

```
@Test
public void testOpenCloseMock()
{
    context.checking(new Expectations() {{
        one(light).turnOnLight();
        one(camera).turnOnCamera();
    }});

    mainDoor.open();
    context.assertIsSatisfied();

    context.checking(new Expectations() {{
        one(light).turnOffLight();
        one(camera).turnOffCamera();
    }});

    mainDoor.close();
    context.assertIsSatisfied();
}
}
```

```

public class Door
{
    private String name;
    private boolean isOpen;
    private Collection<Light> lights;
    Collection<Camera> cameras;

    public Door(String name)
    {
        this.name = name;
        isOpen = false;
        lights = new HashSet<Light>();
        cameras = new HashSet<Camera>();
    }
    ...
    public void addCamera(Camera camera)
    {
        cameras.add(camera);
    }

    public void open()
    {
        if (isOpen == false)
        {
            System.out.println("Opening " + name);
            isOpen = true;
            turnOnLights();
            turnOnCameras();
        }
    }
    ...

```

2:Door

```

...
    private void turnOnCameras()
    {
        for (Camera camera : cameras)
        {
            camera.turnOnCamera();
        }
    }

    private void turnOffCameras()
    {
        for (Camera camera : cameras)
        {
            camera.turnOffCamera();
        }
    }
}

```


3:Light

- No change from previous versions

```
public class Light
{
    private String name;
    private boolean lightOn;

    public Light(String nm)
    {
        name = nm;
        lightOn = false;
    }

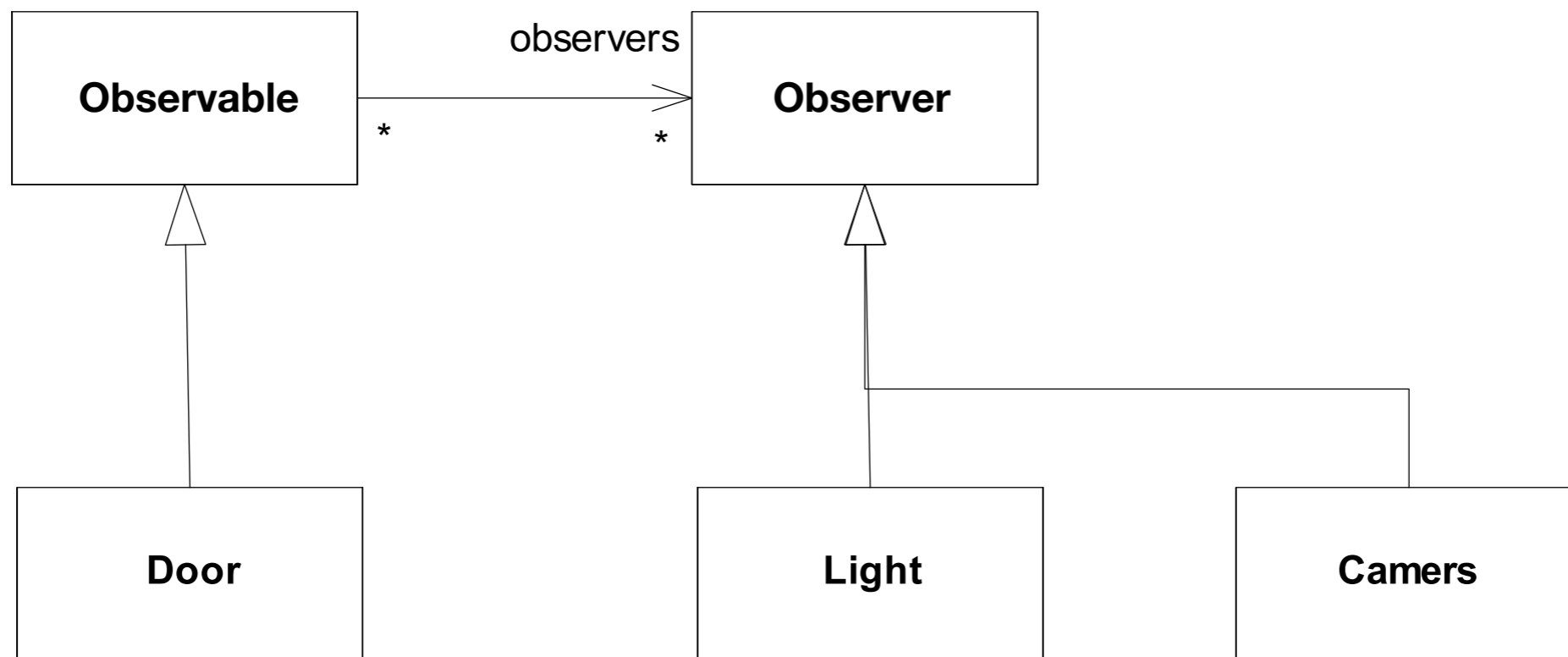
    public void turnOnLight()
    {
        if (!lightOn)
        {
            System.out.println("Turning on light" + name);
            lightOn = true;
            // activate light sensor
        }
    }

    public void turnOffLight()
    {
        if (lightOn)
        {
            System.out.println("Turning Off light" + name);
            lightOn = false;
            // deactivate light sensor
        }
    }

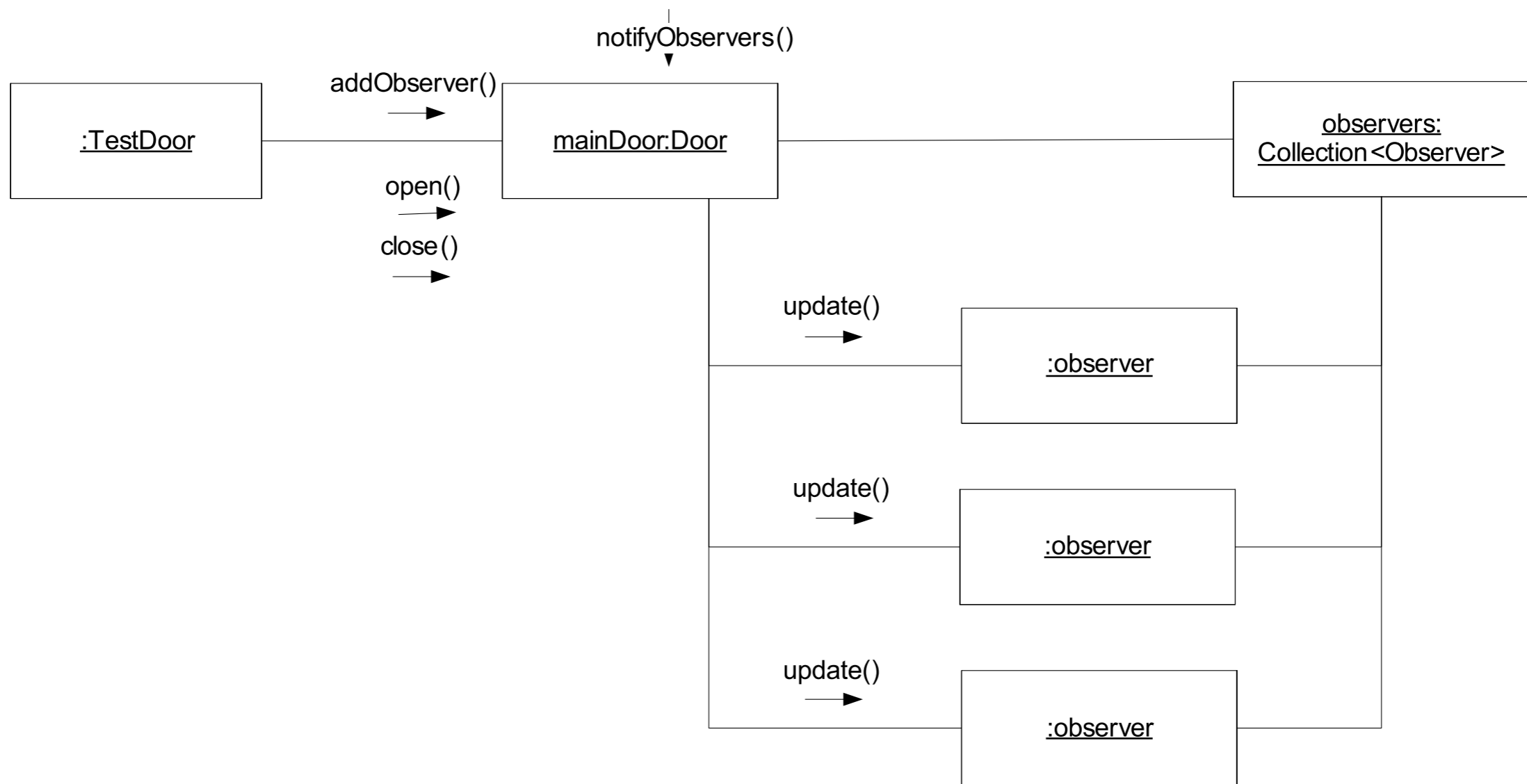
    public boolean getLightState()
    {
        return lightOn;
    }
}
```

Version 3: Observer

- Decouple Light & Camera from Door by refactoring to use Observer Pattern



3: Communication Diagram



3:TestDoor

```
public class DoorTest
{
    private Door mainDoor;
    private Light light;
    private Camera camera;
    ...

    @Before
    public void setUp() throws Exception
    {
        ...

        mainDoor = new Door("Main Door");
        mainDoor.addObserver(light);
        mainDoor.addObserver(camera);
    }
}
```

```
@Test
public void testOpenCloseMock()
{
    context.checking(new Expectations() {{
        one(light).update(mainDoor, true);
        one(camera).update(mainDoor, true);
    }});

    mainDoor.open();
    context.assertIsSatisfied();
    context.checking(new Expectations() {{
        one(light).update(mainDoor, false);
        one(camera).update(mainDoor, false);
    }});

    mainDoor.close();
    context.assertIsSatisfied();
}
}
```

3:Observer & Observable

```
public interface Observer
{
    public void update(Observable ob, Object o);
}
```

```
public class Observable
{
    protected Collection<Observer> observers;

    public Observable()
    {
        observers = new HashSet<Observer>();
    }

    public void addObserver(Observer light)
    {
        observers.add(light);
    }

    public void notifyObservers(Object o)
    {
        for (Observer observer : observers)
        {
            observer.update(this, o);
        }
    }
}
```

3:Door

- Door is significantly simplified.
- All dependency management is handled in the Observable base class.

```
public class Door extends Observable
{
    private String name;
    private boolean isOpen;

    public Door(String name)
    {
        this.name = name;
        isOpen = false;
    }

    public void open()
    {
        if (isOpen == false)
        {
            System.out.println("Opening " + name);
            isOpen = true;
            notifyObservers(isOpen);
        }
    }

    public void close()
    {
        if (isOpen == true)
        {
            System.out.println("Closing " + name);
            isOpen = false;
            notifyObservers(isOpen);
        }
    }
}
```

3:Camera (Light similar)

```
public class Camera implements Observer
{
    private String name;
    private boolean cameraOn;

    public Camera(String nm)
    {
        name = nm;
        cameraOn = false;
    }

    public void update(Observable ob, Object o)
    {
        Boolean doorStatus = (Boolean) o;

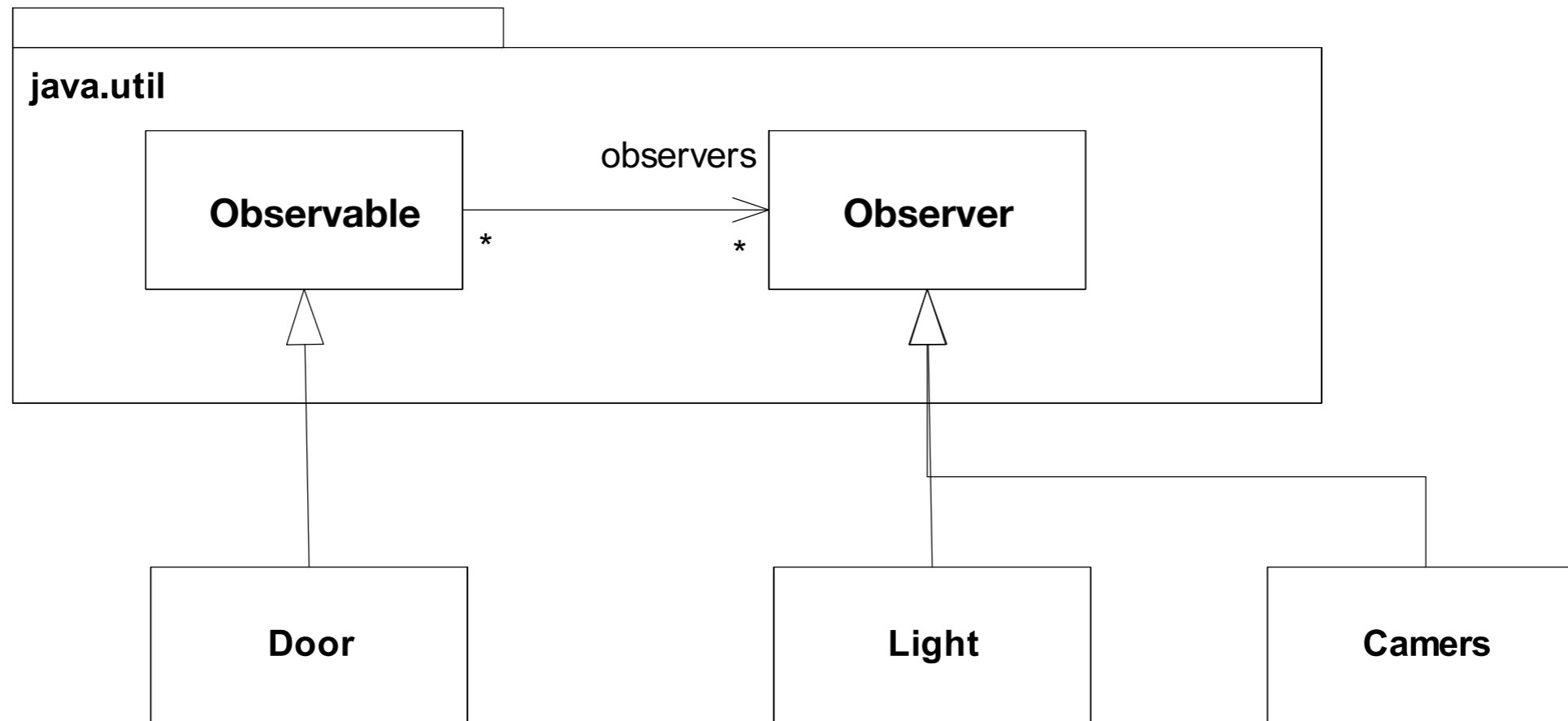
        if (doorStatus == true)
        {
            turnOnCamera();
        }
        else
        {
            turnOffCamera();
        }
    }

    public boolean getCameraState()
    {
        return cameraOn;
    }
}
```

```
public void turnOnCamera()
{
    if (!cameraOn)
    {
        System.out.println("Turning on camera " + name);
        cameraOn = true;
        // activate light sensor
    }
}

public void turnOffCamera()
{
    if (cameraOn)
    {
        System.out.println("Turning Off camera " + name);
        cameraOn = false;
        // deactivate light sensor
    }
}
```

4:Java.util Observer Implementation



4:Door, Light & Camera

```
public class Door extends Observable  
{  
    ...  
}
```

```
public class Light implements Observer  
{  
    ...  
}
```

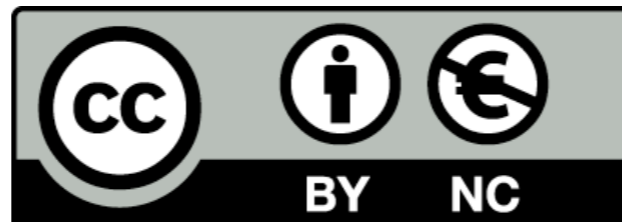
```
public class Camera implements Observer  
{  
    ...  
}
```

5: Anonymous Inner Class Idiom

- With Observe implementations in Java, it is common to implement the observer inline.
- This is called an “Anonymous Inner Class”.
- In the example here, a new Observer implementation is being created, and the single required method provided, all within the parameter list to the addObserver() method.
- Can be difficult to read - but extremely common, particularly in GUI code.

```
@Test
public void demonstrateAnonymousInnerClass()
{
    mainDoor.addObserver(new Observer()
        {
            public void update(Observable obs, Object o)
            {
                Boolean open = (Boolean) o;

                if (open == true)
                {
                    System.out.println("main Door opening");
                }
                else
                {
                    System.out.println("main Door closing");
                }
            }
        }
    );
}
```



Except where otherwise noted, this content is licensed under a Creative Commons Attribution-NonCommercial 3.0 License.

For more information, please see <http://creativecommons.org/licenses/by-nc/3.0/>



Waterford Institute of Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE



eLearning
support unit