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

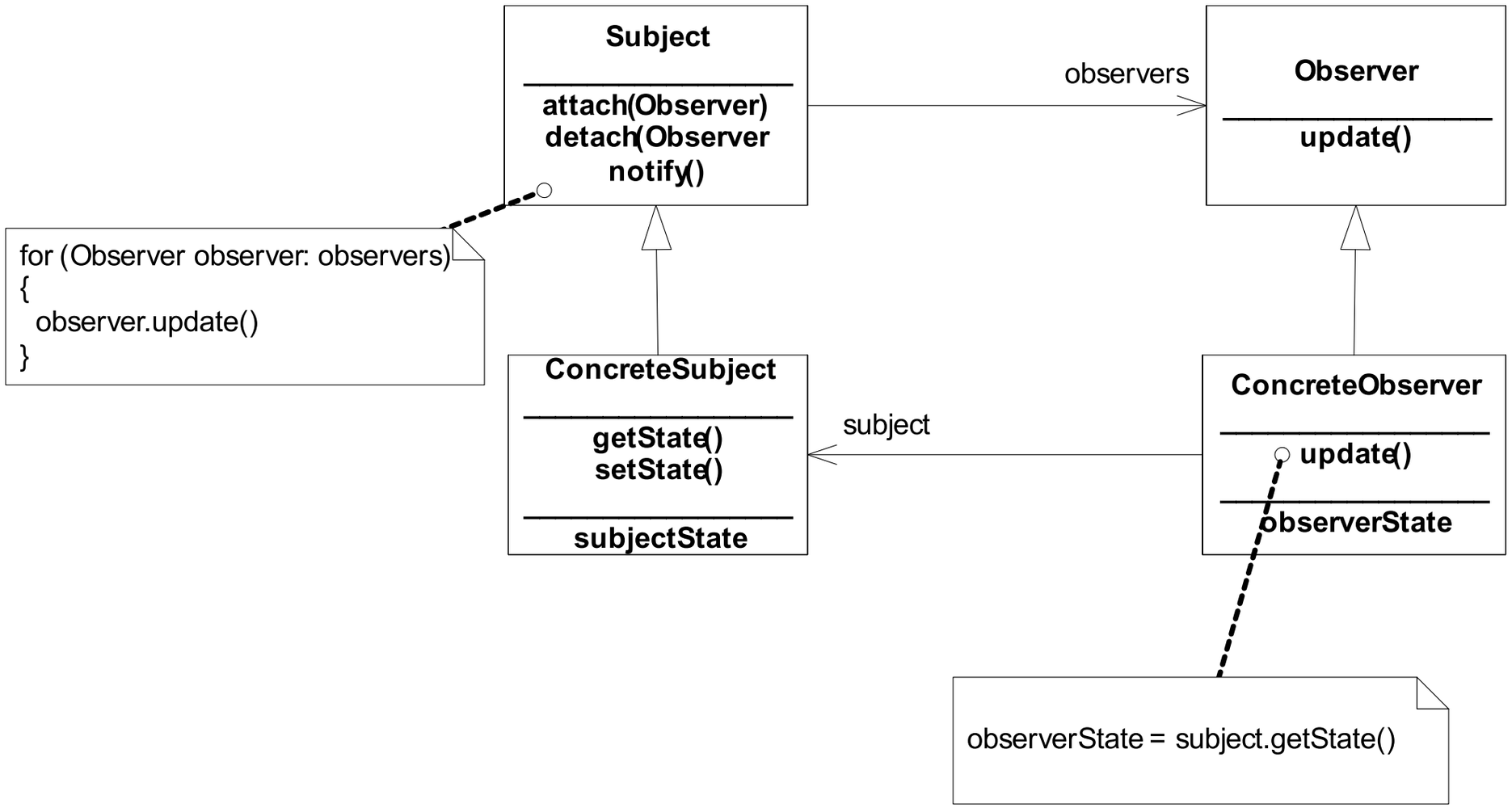
eLearning support unit

Observer

Design Pattern

Summary

- Have objects (Observers) that want to know when an event happens, attach themselves to another object (Subject) that is actually looking for it to occur.
- When the event occurs, the subject tells the observers that it occurred.



Intent

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- Also Known As

  - Dependents, Publish-Subscribe

Motivation(1)

- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.

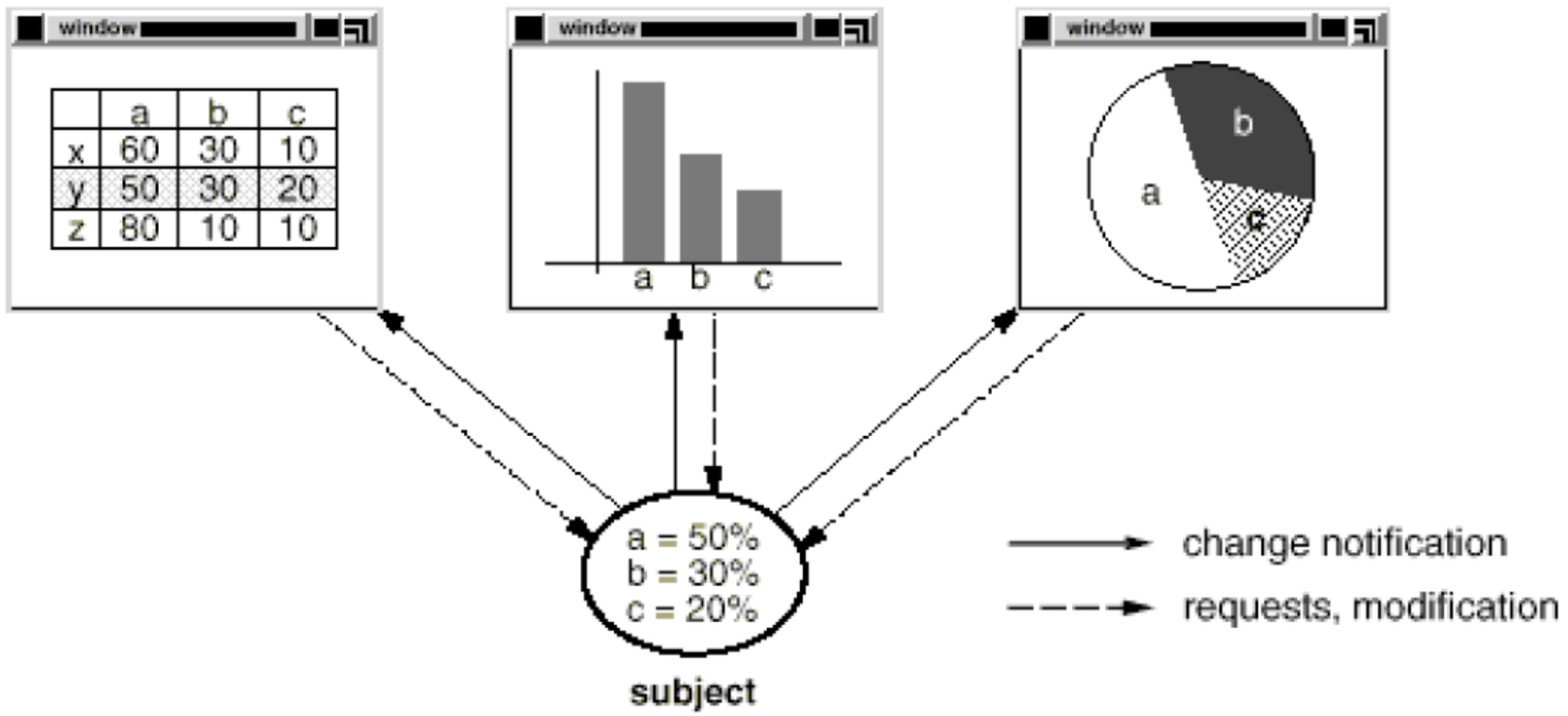
- You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

- For example, many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data

- Classes defining application data and presentations can be reused independently.

Motivation(2)

- E.g. both a spreadsheet object and bar chart object can depict information in the same application data object using different presentations.
- The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need.
- But they behave as though they do. When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa.



Motivation(3)

- The Observer pattern describes how to establish these relationships.

- The key objects in this pattern are subject and observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state.

- In response, each observer will query the subject to synchronize its state with the subject's state.

- This kind of interaction is also known as publish-subscribe.

- The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.

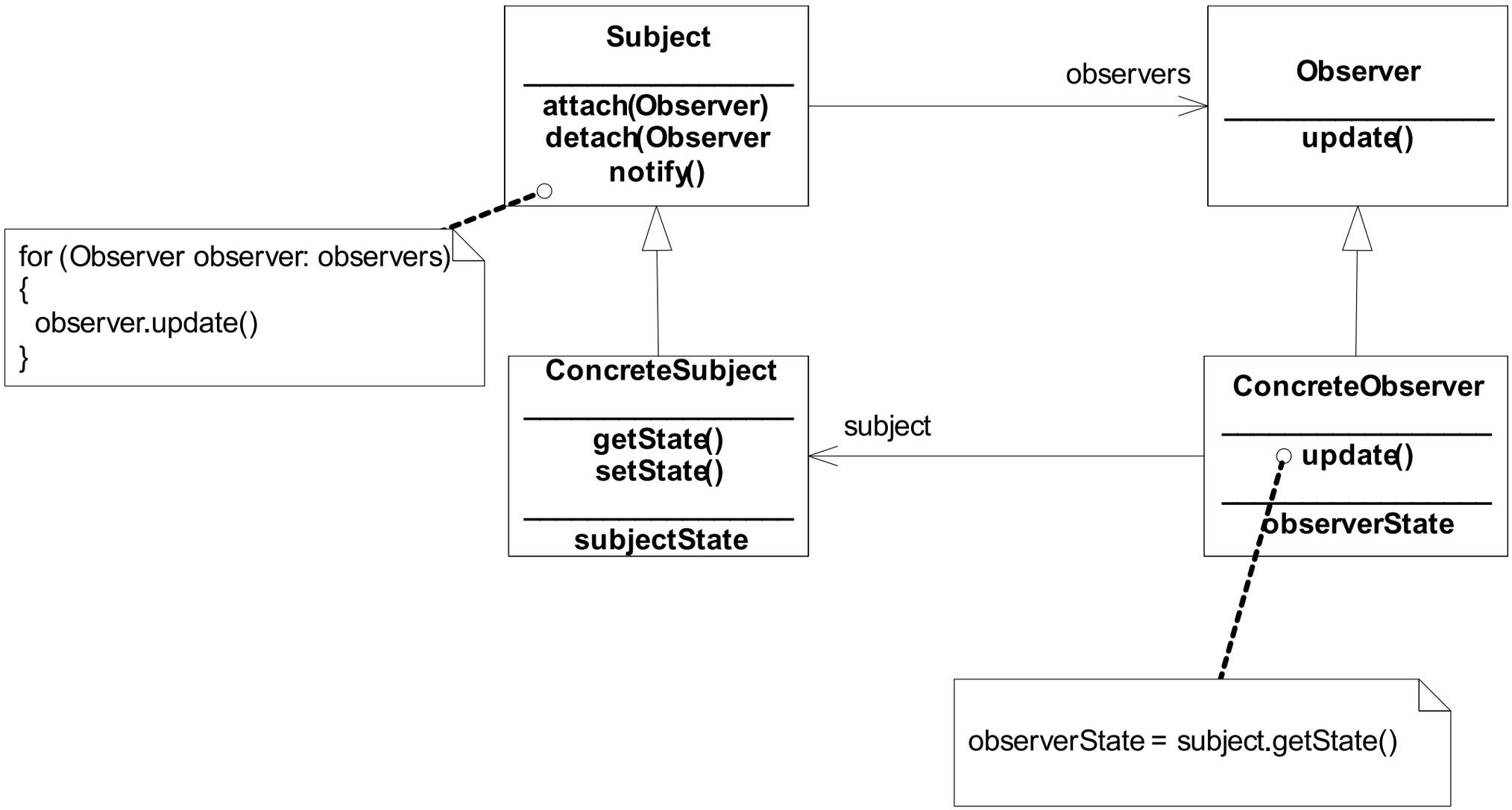
Applicability

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.

- When a change to one object requires changing others, and you don't know how many objects need to be changed.

- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Structure


Subject
attach(Observer)
detach(Observer
notify()
observers
Observer
update()
for (Observer observer: observers)
{
observer.update()
}
ConcreteSubject
getState()
setState()
subjectState
subject
ConcreteObserver
update()
observerState
observerState = subject.getState()

Participants

- Subject
  - knows its observers. Any number of Observer objects may observe a subject.
  - provides an interface for attaching and detaching Observer objects.
- Observer
  - defines an updating interface for objects that should be notified of changes in a subject.
- ConcreteSubject
  - stores state of interest to ConcreteObserver objects.
  - sends a notification to its observers when its state changes.
- ConcreteObserver
  - maintains a reference to a ConcreteSubject object.
  - stores state that should stay consistent with the subject's.
  - implements the Observer updating interface to keep its state consistent with the subject's.

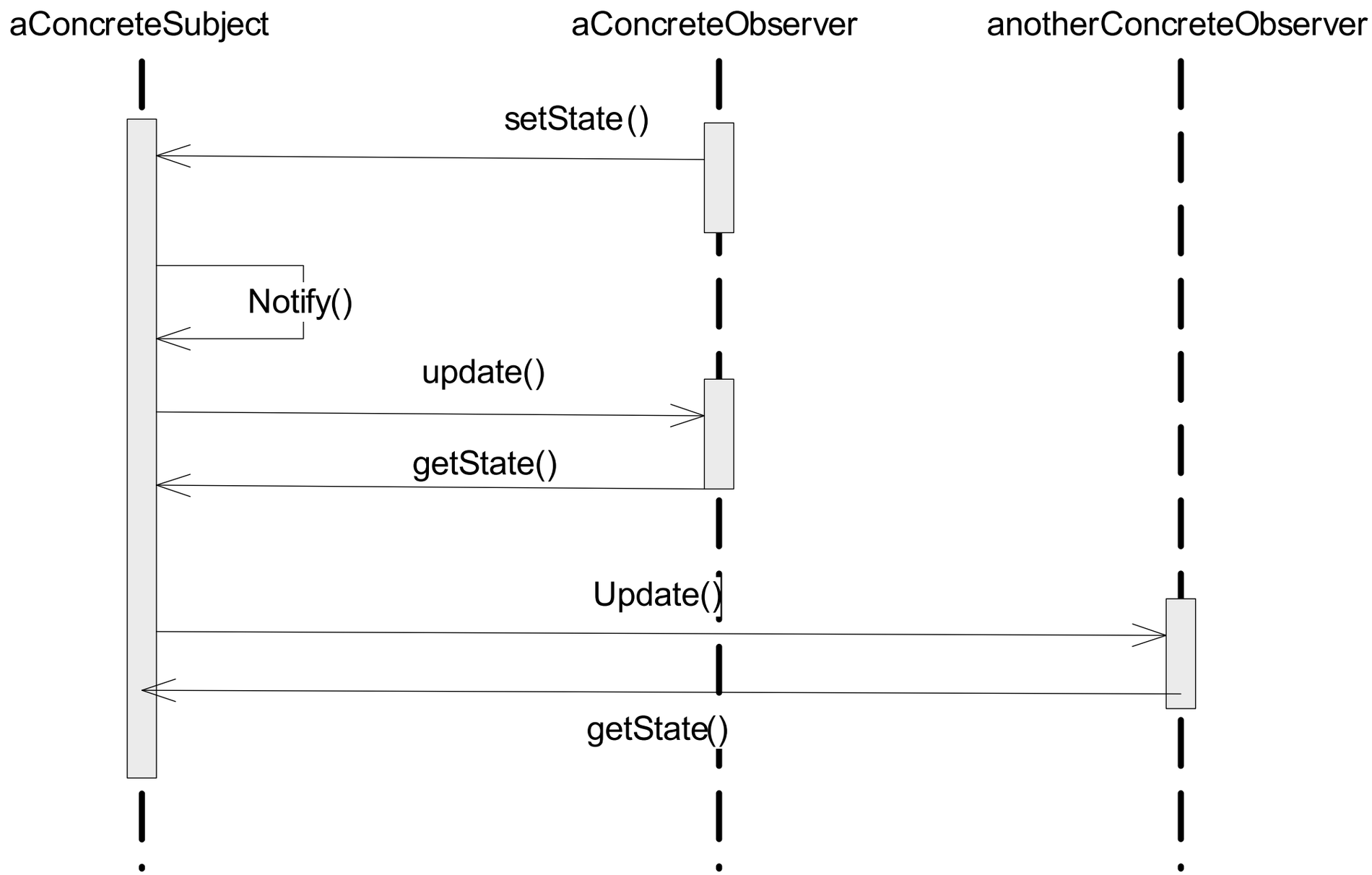
Collaborations(1)

- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.

- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information.

- ConcreteObserver uses this information to reconcile its state with that of the subject.

Collaborations(2)



Consequences(1)

- Abstract coupling between Subject and Observer.
  - All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class.
  - The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal.
- Support for broadcast communication.
  - Unlike an ordinary request, the notification that a subject sends needn't specify its receiver.
  - The notification is broadcast automatically to all interested objects that subscribed to it.
  - The subject doesn't care how many interested objects exist; its only responsibility is to notify its observers. This gives you the freedom to add and remove observers at any time.

Consequences(2)

- Unexpected updates.
  - Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject.
  - A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects.
- Moreover, dependency criteria that aren't well-defined or maintained usually lead to spurious updates, which can be hard to track down.
- This problem is aggravated by the fact that the simple update protocol provides no details on what changed in the subject. Without additional protocol to help observers discover what changed, they may be forced to work hard to deduce the changes.

Example(1)

- Most well know implementation is the event handling mechanism for Widgets in GUI libraries.
- The Widget (button for example) is capable of generating events
- Interested observers “listen” events using a standard protocol.
- The Widgets do not now who the observers are, but merely broadcast state changes in the widget (usually as a result of user interaction) to all interested parties.
- The “listeners” may query the concrete subject (the button) for additional information if necessary.

Example(2)

- Observer & Subject

```
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);
    }
  }
}
```

Example(3)

- ConcreteSubject

```
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;
      setChanged();
      notifyObservers(isOpen);
    }
  }

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

Example(4)

- ConcreteObserver

```
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
    }
  }
```