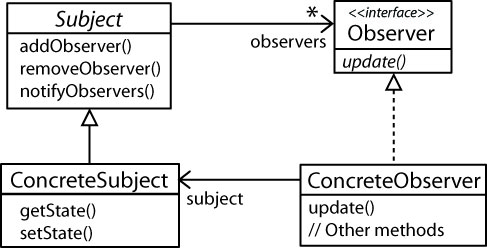
**Chaper 2 Notes – Observer Pattern**

**Definition**

1. Intent: The Observer Pattern provides a mechanism so that an object (Subject, or Observable) can notify dependent objects (Observers) of changes without having to know their concrete classes. The observer pattern is a behavioral pattern as it defines a mechanism for communication.



The Observer Pattern defines a one-to-many dependency between object so that when one object changes state, all of its dependents are notified and updated automatically [HFDP]

1. Design Principles:

How are these exhibited in the example from the text?

1. Identify the aspects of your application that vary and separate them from what stays the same.
2. Program to an interface, not an implementation.
3. Favor composition over inheritance.
4. Strive for loosely coupled designs between objects that interact (new).

Describe how each of these principles is illustrated in the strategy pattern.

1. Example. The Observer pattern transforms:

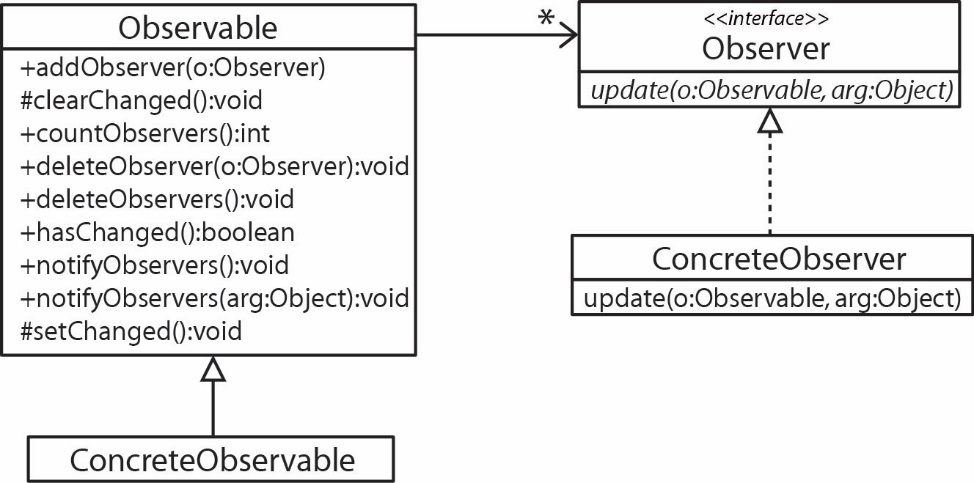
|  |  |
| --- | --- |
| **Dependent Design** | **Loosely-Coupled Design** |
| E:\Data-Classes\CS 4322 - Software Engineering 2\Notes\02-Observer\pics\r7.tif | E:\Data-Classes\CS 4322 - Software Engineering 2\Notes\02-Observer\pics\tv.jpg |

**Observer in Java**

1. The java.util package supplies the (concrete) Observable class and the Observer interface as shown below.

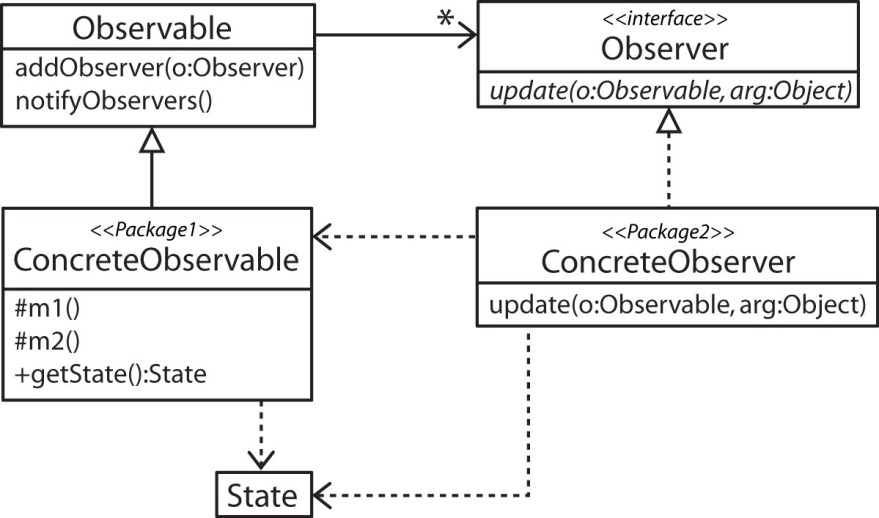
Implementation details:

1. The Observable class has a private flag that is set to *false* initially. When the flag is *false,* a call to *notify* does nothing, the observers are not notified. A protected method, *setChanged* sets the flag to *true*. Then, a subsequent call to *notify* will call *update* on all observers and then sets the flag back to *false*. So, we can view this flag as a protection mechanism in the sense that clients can call *notify* at any time, but *notify* will only work when the subclass has previously called *setChanged*. Thus, a client can call *notify,* but only the concrete observable can allow the notification to take place. The *clearChanged* method sets the flag to *false* so that calls to *hasChanged* return false.
2. The *update* method has two parameters, a reference to the *Observable* and an arbitrary *Object*.
3. The *notify* method is overloaded. If it is called with no argument, then *null* is used as an argument in the *update* method. Otherwise, *notify* can be called with an argument which is then used as an argument to the *update* method. Thus, we can *push* information to the observers if we want.
4. The order of notification of observers is not specified. In other words, if we want a specific order, we would need to write our own Observable.

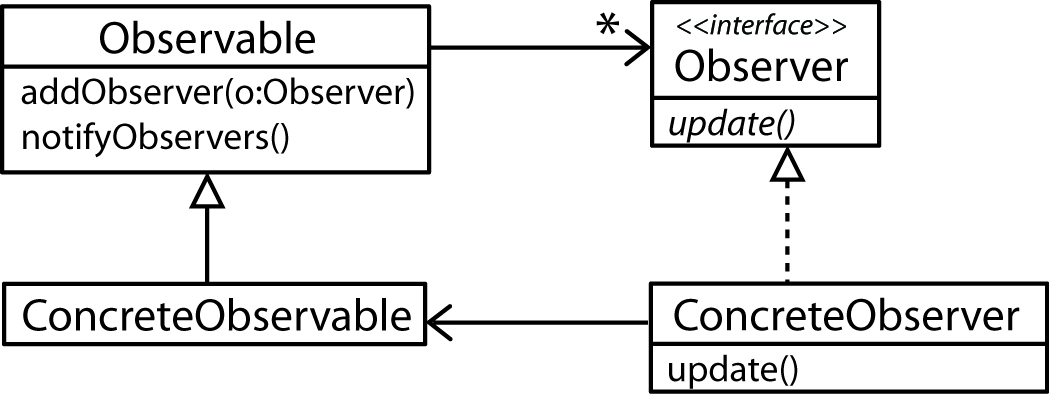


Design questions:

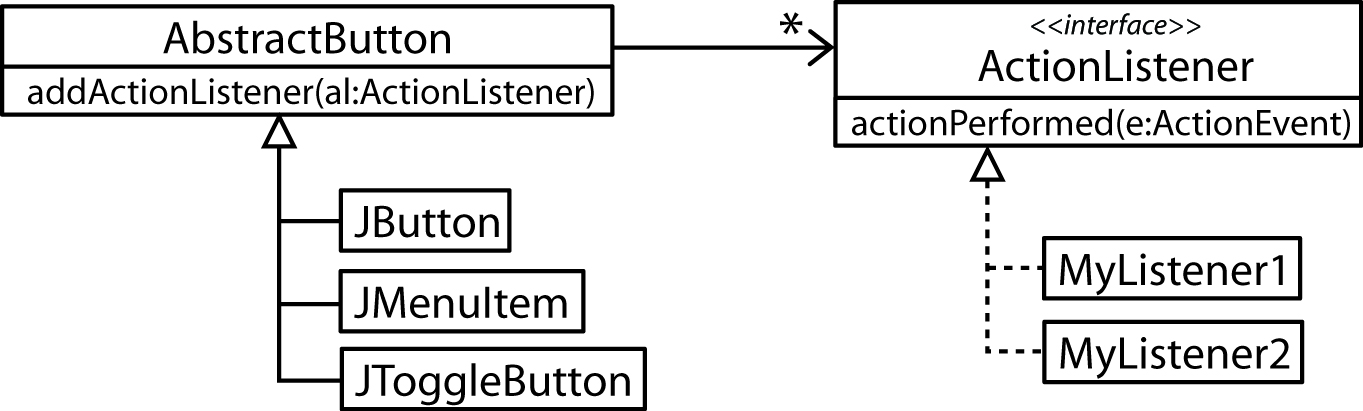
1. Who should call *notify*? The client or the *ConcreteObservable*?
2. Should we *push* data to the observers or should we require that observers *pull* the information they need?
3. How could we prevent clients from being able to call *notify*?
4. You might like to narrow the scope of *notify* in subclass that extends *Observable* but Java doesn’t allow this*.*
5. Packaging will not work because *notify* is public.
6. Use a proxy (e.g. composition and delegation). This works, but when *notify* is called internally, it calls *update* on the *Observers* and passes a reference to itself. So, clients can’t see notify, but *Observers* can.
7. Write a custom observable class.
8. A more protected design might have use introduce a State class to represent the state of the ConcreteObservable (or the part of the state that we want to expose). In other words, we might not want the observers to access everything in the observable. Thus, we could put the ConcreteObservable in a different package than the Observers and expose only a *getState* method for use by the observers.



1. Sometimes the ConcreteObserver(s) may contain a reference to the ConcreteObserverable so that it can disconnect itself at any time or for convenience.

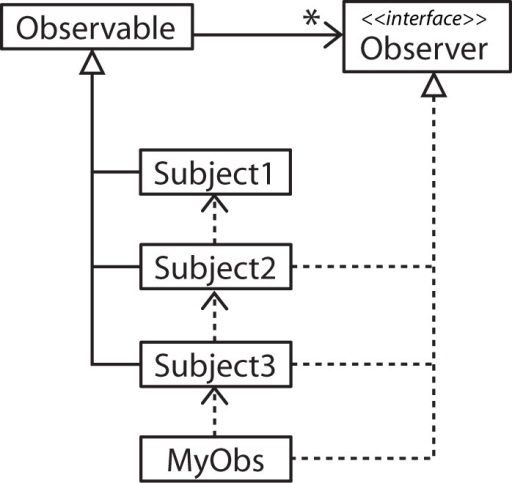


1. GUI Event handling in Java Swing (and AWT and JavaFX) utilizes the Observer Pattern. GUI elements that can be interacted with by a user. For instance, when a user presses a button, an event is fired, which triggers the execution of some code to *handle* the event. In Java, ActionListeners handle events by supplying an *actionPerformed* method and are *registered* with GUI elements.



**Other Examples**

1. Observers that are also observable



1. C# has a more general approach to the Observer pattern through the use of delegates and events.

|  |  |
| --- | --- |
|  |  |
| **Observable**  public class A {  **// Declare the signature of an event handler**  public delegate void MyDelegate( Object arg );  **// Declare the name of the event**  public event MyDelegate MyEvent;  public void MyMethod(); {  ...  **// Fire the event**  MyEvent( myArg );  }  } | **Observer**  public class B {  ...  **// A method that matches the delegate signature**  public void MyEventHandler( Object thing ) {  ...  }  ...  } |

**Driver**

public class C {

...

A a = new A()

B b = new B()

**// Create delegate instance and bind to event handler**

A.MyDelegate del = new A.MyDelegate( b.MyEventHandler );

**// Add delegate to event**

A.MyEvent += del;

...

}

1. How does recalculation work in Excel, or any spreadsheet? What is a circular reference? Draw a class diagram of a circular reference.