The Observer Pattern

"Don't call us, we'll call you!"

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Updating Interested Parties

- Many applications require notifications when things change
  - Files change on disk
  - User has authenticated
  - Data has arrived on a socket

Problem

- How do you notify an object that data has changed?
  - User tries to authenticate
  - User logouts

Players

- Subject
  - Detects somehow that the data has changed
  - e.g. Determines that user wants to authenticate
  - e.g. Determines that user logs out
- Observers
  - Want to know when the data has changed
  - e.g. Authenticated user
  - e.g. Logged out user

Implement an update method in listener

- Simplest solution
  - Have subject notify observer when data changes
  - May have multiple observers

```
public class User
{
    protected void UserHasAuthenticated() {
        service1.Authenticated();
        service2.Authenticated();
        service3.Authenticated();
    }
}
```

What are the problems with this approach?

- Many problems
  - Very tight coupling
  - Need to change code to add a new observer
  - Observers cannot be added or removed dynamically

Decoupling the Listener

- Define an interface with an update method on
  - Have listener implement the interface

Interface definition and implementation

- All observers implement the interface

```
public interface IAuthenticatedService
{
    void Authenticated();
    void LoggedOut();
}
```

```
public class LoginPage: IAuthenticatedService
{
    public void Authenticated() {}
    p
}
```

```
public class WebSite: IAuthenticatedService
{
    public void Authenticated() {}
    p
}
```

```
public class Bank: IAuthenticatedService
{
    public void Authenticated() {}
    public void LoggedOut() {}
}
```

Supporting Dynamic Listeners

- Subject maintains collection of listener interface
  - Needs method to add listeners
  - Needs method to remove listeners
  - Iterates over collection to call listeners
  - What about synchronization?

Subject Interface

- Subject should also be an interface
  - Better to code to interface rather than implementation

```
public abstract class User
{
    public virtual void AddAuthenticatedService
                                        (IAuthenticatedService idc);
    public virtual void RemoveAuthenticatedService
                                        (IAuthenticatedService idc);
    protected virtual void NotifyAuthenticationStatus();
}
```

Subject Add and Remove

- Use collections to manage observers

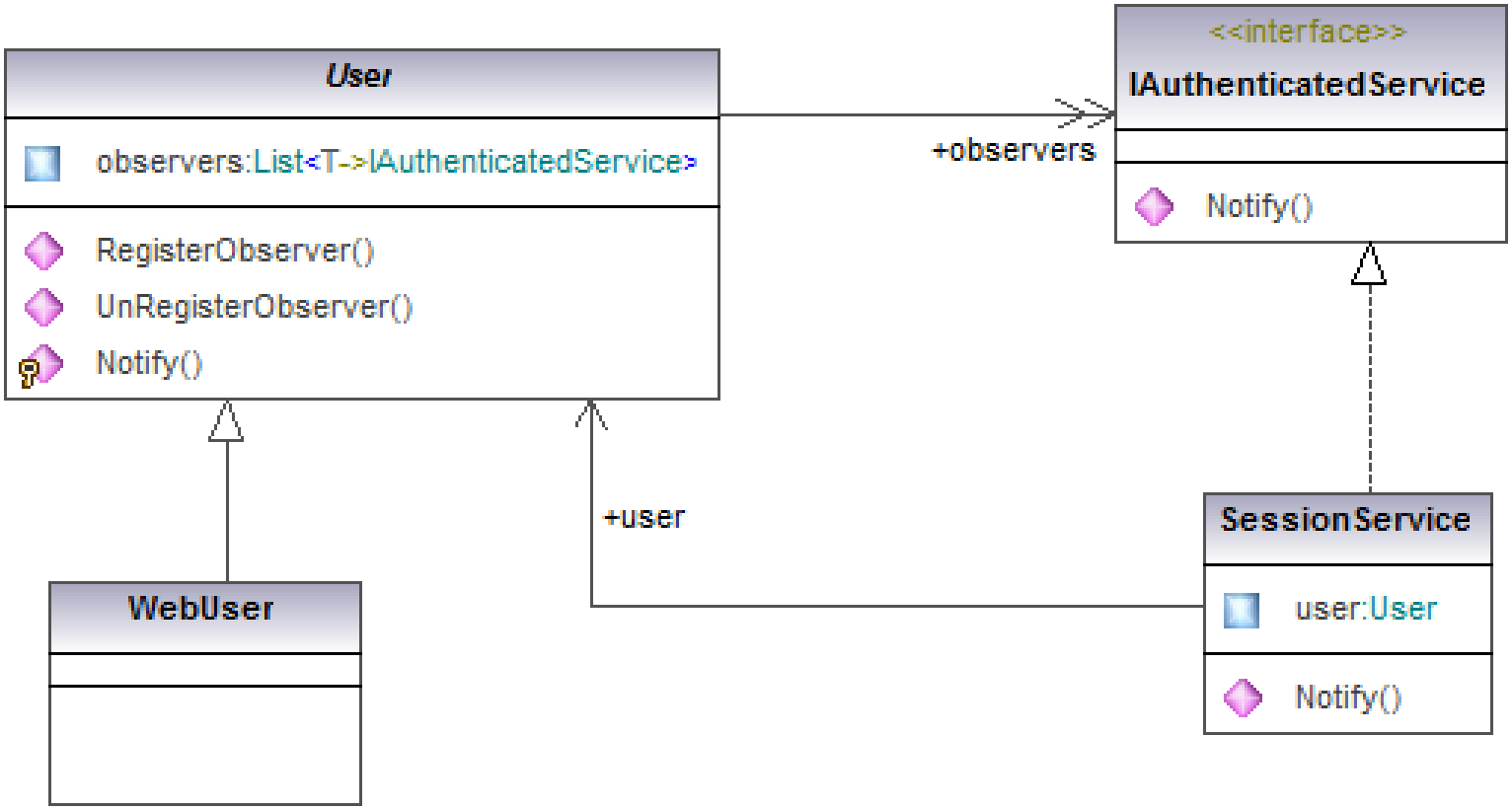
```
public abstract class User
{
    List< IAuthenticatedService > services =
                                new List< IAuthenticatedService >();

    public virtual void AddAuthenticatedService(
                                        IAuthenticatatedService idc)
    {
        services.Add(idc);
    }

    public virtual void RemoveAuthenticatedService(
                                        IAuthenticatatedService idc)
    {
        services.Remove(idc);
    }
}
```

Subject Update

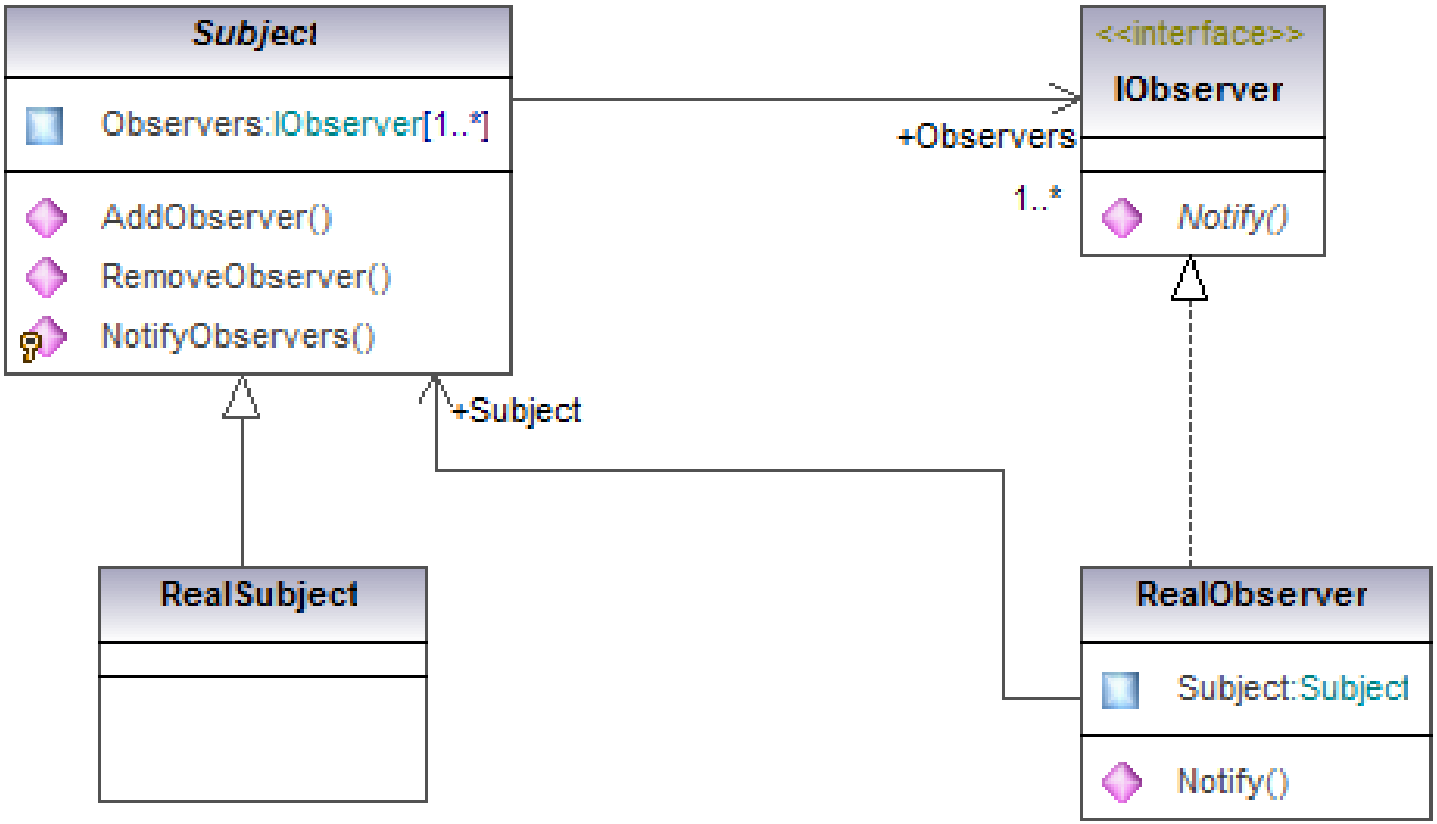
- Iterate over all observers

```
public abstract class User
{
    protected virtual void NotifyAuthenticationStatus()
    {
        foreach (IAuthenticatatedService observer in services)
        {
            observer.Authenticate();
        }
    }
}
```

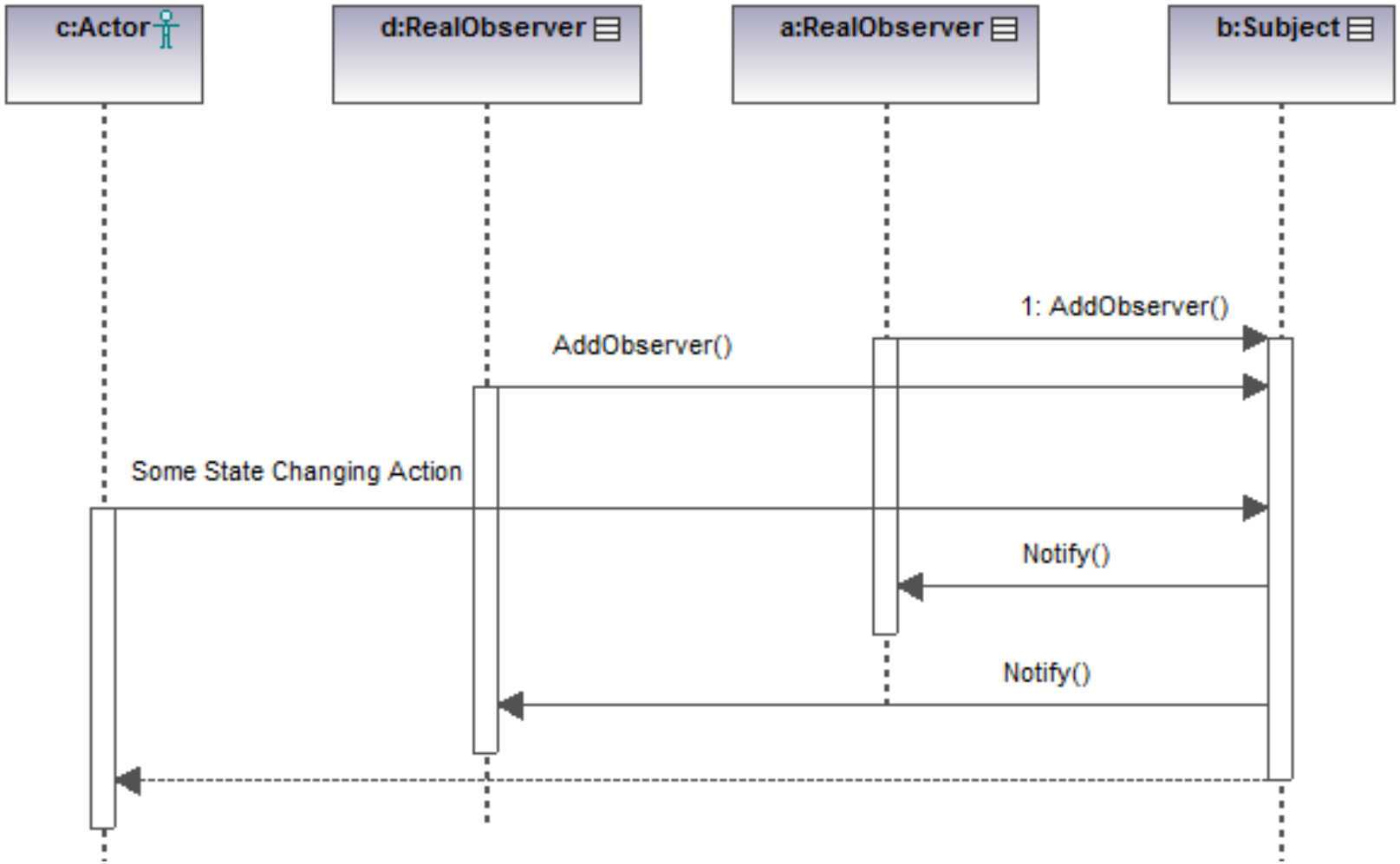
Authentication Class Diagram



More Generally



Observer in action



Problem with Previous Solution?

- Observer defines more than one method
  - All listeners need to implement all the methods on the interface
  - May only care about one method
- Boilerplate code in the subject
  - Adding and removing listeners
  - Gets repeated for every subject
  - Easy to get wrong (think synchronization)

.Net Solution - Delegates

- Delegates allow for
  - Very loose coupling
  - Easy management
- Declare delegate in the Subject
- Register the delegate in the Observer

Delegate Definitions

- Define the delegates in the Subject

```
public delegate bool Authenticated();
public delegate void LoggedOut();

Public abstract class User {

    public Authenticated authenticated;
    public LoggedOut loggedout;

    protected void RequestAuthentication() {
        if (authenticated != null)
            authenticated();
    }

    protected void RequestLogout() {
        if (loggedout != null)
            loggedout();
    }
}
```

Implement Delegate in Observers

- Only implement the delegates you care about
  - Authenticate and not Logout
- Don't forget to de-register

```
public sealed class LoginPage : IDisposable {
    User subject;
    public LoginPage(User subject) {
        this.subject = subject;
        this.subject.authenticated += Authenticated;
    }

    public void Authenticated() {
        // work here
    }

    public void Dispose() {
        this.subject.authenticate -= Authenticate;
    }
}
```

Only cares about authentication

Delegates Defined this way are Public

- Delegate declarations are public
  - This is how the observers subscribe

```
public Authenticated authenticated;
public Loggedout loggedout;
this.subject.authenticated += Authenticated;
```

- Leads to problems
  - Anybody can fire delegate

```
loginPage.Authenticated();
```

  - Can overwrite delegate

```
this.subject.authenticated = Hack.Bypass;
```

Enter 'Events'

- Use `event` keyword
  - Syntactic sugar
- Changes nature of delegate definition
  - Changes declaration of delegate instance to private
  - Adds **`add_`**/**`remove_`** methods to class
  - Only allows calls to **`+=`** and **`-=`**

```
public abstract class User {
    public event Authenticated authenticated;
    public event Loggedout loggedout;
}
```

Threading Issues

- There can still be problems in thread hot environments
  - Adding and removing delegates is synchronized (+=/-=)
  - May have thread Safety issues when firing an event
  - Delegates are immutable, so no locking needed, however...
  - ...check for null is necessary

```
public delegate void LoggedOut();

public abstract class User {
    public event LoggedOut loggedout;

    protected void RequestLogout() {
        LoggedOut localHandler;
        localHandler = loggedOut;
        if (localHandler != null)
            localHandler();
    }
}
```

Getting rid of the null check

- Use the 'null object' pattern
  - In this case simply a do-nothing method
  - Simplifies code in many cases

```
public class User {
    public event LoggedOut loggedout;

    private void NullLogoutCallback(){}

    public User() {
        loggedOut += NullLogoutCallback;
    }

    private void RequestLogout() {
        loggedOut();
    }
}
```

Common Uses of Observer

- Event Handlers
  - Windows Forms
  - ASP.NET
- File system
  - FileSystemWatcher
- HTML DOM
  - Javascript events

Automatic unregister

- Observer pattern requires applications too register and unregister
  - Failure to do so, could cause memory leaks
- Since we know programmers often forget this step we can defensively program around it using Weak References
  - Recipe
    - Subject holds references to observers as Weak References
    - When calling observers, subject turns each Weak Reference to strong reference, if null observer is no more
- By holding Weak References to observers, the observer is not prevented from being GC'd due to the subject reference.

Summary

- Observer is a very common pattern
  - Typically used without being aware of it
  - Mostly used in UI type applications
  - Delegates do all the heavy lifting in .NET
  - Remember to de-register the delegate to avoid memory leaks
    - Consider Weak Reference implementation