

SOFTWARE ARCHITECTURE

Think Big and Keep It Simple

MORE ABOUT UNIT TEST

Déjà vu?

UNIT TEST BEST PRACTICES

What's NOT unit test:

- Add a function, compile and run to see if it works.
- Smoke testing: on new builds 'Does keyboard input work?' 'Can the shop's inventory be populated?' (These tests are usually run by QA team not development team)
- Integration testing: tests on database functions, web services, etc.

Unit tests are nothing more than methods that have attributes to indicate to the test runner that they are unit tests.

A test runner is a program that executes compiled test code and display the results and feedback.

Unit tests can be described as: Setup, do something, assert.

Unit tests can be seen as running experiments on classes and methods, to check if their behaviors are correct.

Doing unit test forces you to understand how the smallest unit of your code should behave, therefore makes the whole code more predictable.

Unit tests almost always use Assert statements.

Unit tests usually use meaningful names which indicates clearly what the tests are.

Unit tests don't have an order, each test should be able to run on its own.

```
using NUnit.Framework;

[TestFixture]
public class AccountTest
{
    Account source;
    Account destination;

    [SetUp]
    public void Init()
    {
        source = new Account();
        source.Deposit(200m);

        destination = new Account();
        destination.Deposit(150m);
    }

    [Test]
    public void TransferFunds()
    {
        source.TransferFunds(destination, 100m);

        Assert.AreEqual(250m, destination.Balance);
        Assert.AreEqual(100m, source.Balance);
    }

    [Test]
    [Ignore("Decide how to implement transaction management")]
    public void TransferWithInsufficientFundsAtomicity()
    {
        try
        {
            source.TransferFunds(destination, 300m);
        }
        catch (InsufficientFundsException expected)
        {
        }
    }
}
```

TEST NEWLY ADDED CLASSES/FUNCTIONS

Examples:

CreateItemWithNegativeAttributeTest (Throw exceptions?)

AllFieldsOfItemAreDeepCopiedTest (All fields of the returned item stay the same after the original item changes?)

Test as many public methods as possible.

REFACTOR EXISTING CODE TO EXTRACT TESTABLE CLASSES.

Let's take a look at two classes taken out of the
a mysterious game starred Unity-Chan:

```
public class UnityChanMeshDrawer{  
    ...  
    public void HardToTestMethodI(){  
        if(IsPointInsideTriangle(InputManager.GetInstance().CurrentPositionVector2, nextTriangle))  
        {  
            //Send out an event to Unity Chan  
        }  
    }  
  
    private bool IsPointInsideTriangle(Vector2 point, Triangle triangle){  
        //boring-interesting math stuff  
    }  
    ...  
}
```

HardToTestMethodI is hard to test because it's coupled with InputManager, IsPointInsideTriangle could be testable, except that it's private. So what should we do?



REFACTOR EXISTING CODE TO EXTRACT TESTABLE CLASSES.

One possible solution:

```
public class PointInsideTriangleChecker{
    public IsPointInsideTriangle(Vector2 point, Triangle triangle){
        ...
    }
}

public class UnityChanMeshDrawer{
    ...
    public void HardToTestMethod(){
        if(IsPointInsideTriangle(InputManager.GetInstance().CurrentPositionVector2))
        {
            //Send out an event to Unity Chan
        }
    }

    private bool IsPointInsideTriangle(Vector2 point, Triangle triangle){
        new PointInsideTriangleChecker().IsPointInsideTriangle(point, triangle)
    }
    ...
}
```

PointInsideTriangleChecker class is testable.

REFACTOR EXISTING CODE TO EXTRACT TESTABLE CLASSES.

```
public class MonsterBehaviour{
    ...
    private Vector2 position2D;

    public void HardToTestMethod2(Triangle triangle){
        if(PositionInsideTriangle(triangle))
        {
            KillSelf();
        }
    }
    private bool PositionInsideTriangle(Triangle triangle){
        //Check if position2D is inside of tirangle
    }
    ...
}
```



HardToTestMethod is hard to test because we can't set up the private field position2D.

We can't move the private field position to a new class to test like the solution above. So what should we do?

REFACTOR EXISTING CODE TO EXTRACT TESTABLE CLASSES.

One possible solution:

```
public class PointInsideTriangleChecker{  
    public IsPointInsideTriangle(Vector2 point, Triangle triangle){  
        ...  
    }  
}
```

//Above class is the same as the last solution

```
public class MonsterBehaviour{  
    ...  
    private Vector2 position2D;  
  
    public void HardToTestMethod2(Triangle triangle){  
        if(new PointInsideTriangleChecker().IsPointInsideTriangle(position2D, triangle))  
        {  
            KillSelf();  
        }  
    }  
    ...  
}
```

REFACTOR EXISTING CODE TO EXTRACT TESTABLE CLASSES.

Look for code that you can move to other classes. In the example above, `isPointInsideTriangle` doesn't have to be in the class `UnityChanMeshDrawer`.

Most of the time you can extract testable classes and methods out of one giant method.

Unit testing helps you to achieve high **cohesion**.

That's a fancy term, let's talk about it!

COHESION.

How closely the elements in a module are related.

High cohesion code is easy to understand, debug and reuse.

When you try to find unit-testable methods, you are mostly looking for low-cohesion methods added to the existing class, therefore by refactoring you increase the cohesion degree.

HIGH OR LOW COHESION.

```
public class Student{  
    public string GetName(){...}  
    public int GetStudentNumber(){...}  
    public string GetNationality(){...}  
    public string GetAddress(){...}  
    public int GetPhone(){...}  
  
    public void ChangeAddress(){...}  
    public void ChangeMajor(){...}  
    public void ChangePhone(){...}  
  
    public int GetCredits(){...}  
    public void CreateNewEducation(){...}  
    public void ModifyExistingEducation(){...}  
}
```

COUPLING

How closely are modules connected to each other.

Tightly coupled modules are hard to work on separately.

Aim for loose coupling, so that the change of one module has minimum impact on other modules.

COUPLING

Sound class of the old GXEngine, is it tightly or loosely coupled to FMod?

```
public Sound( String filename, bool looping = false, bool streaming = false)
{
    if ( _system == 0 ) { // if fmod not initialized, create system and init default
        FMOD.System_Create( out _system );
        FMOD.System_Init( _system, 32, 0, 0 );
    }
    uint loop = FMOD.FMOD_LOOP_OFF; // no loop
    if ( looping ) loop = FMOD.FMOD_LOOP_NORMAL;
    if ( streaming ) {
        FMOD.System_CreateStream( _system, filename, loop, 0, out _id );
        if ( _id == 0 ) {
            throw new Exception ( "Sound file not found: " + filename );
        }
    } else {
        if ( _soundCache.ContainsKey ( filename ) ) {
            _id = _soundCache [ filename ];
        } else {
            FMOD.System_CreateSound ( _system, filename, loop, 0, out _id );
            if ( _id == 0 ) {
                throw new Exception ( "Sound file not found: " + filename );
            }
            _soundCache [ filename ] = _id;
        }
    }
}
```

DECOUPLING PATTERNS

Writing code to get things done is easy.

Writing code that's easy to adapt when requirements change is hard.

Applying decoupling patterns to improve your architecture.

DECOUPLING PATTERNS

Behavioral:

Observer

Event queues

Command

Strategy

Creational:

Facade

OBSERVER

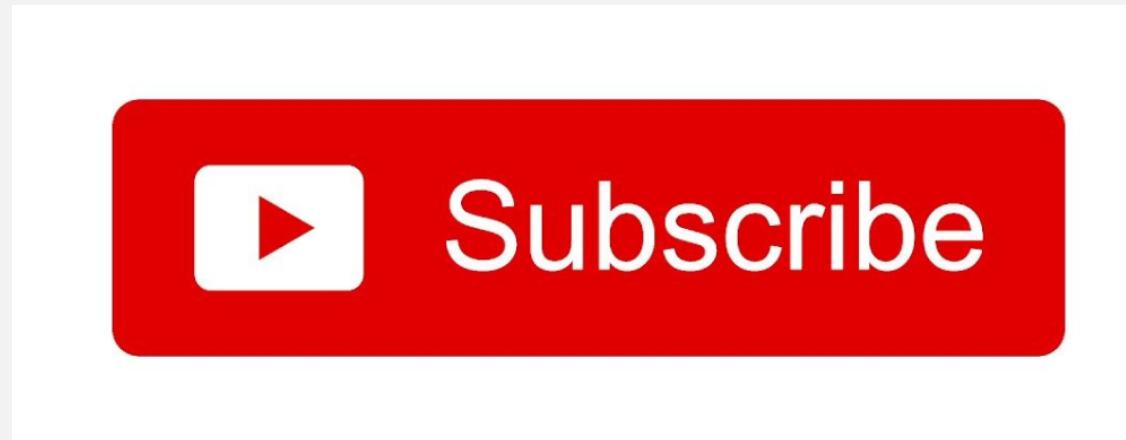
A one-to-many relationship between objects, when one object's state changes, all of its dependents are notified and updated automatically.

The object that holds the state is called a subject.

The dependent objects are called observers/subscribers.

The observers subscribe to the subject, and gets notified when the subject's state changes.

- The word **subscribe** should be enough to give a clear picture of the pattern:



YouTube, Facebook, HumbleBundle, etc.

One subject has multiple subscribers(observers).

The subject **pushes notifications** to you when content is **updates** state changes).

Observers can **subscribe and unsubscribe** from the subject.

Observer Pattern

Subject has a collection of observers.

Subject can also be an abstract class, but it limits the implementation in C#.(Why?)

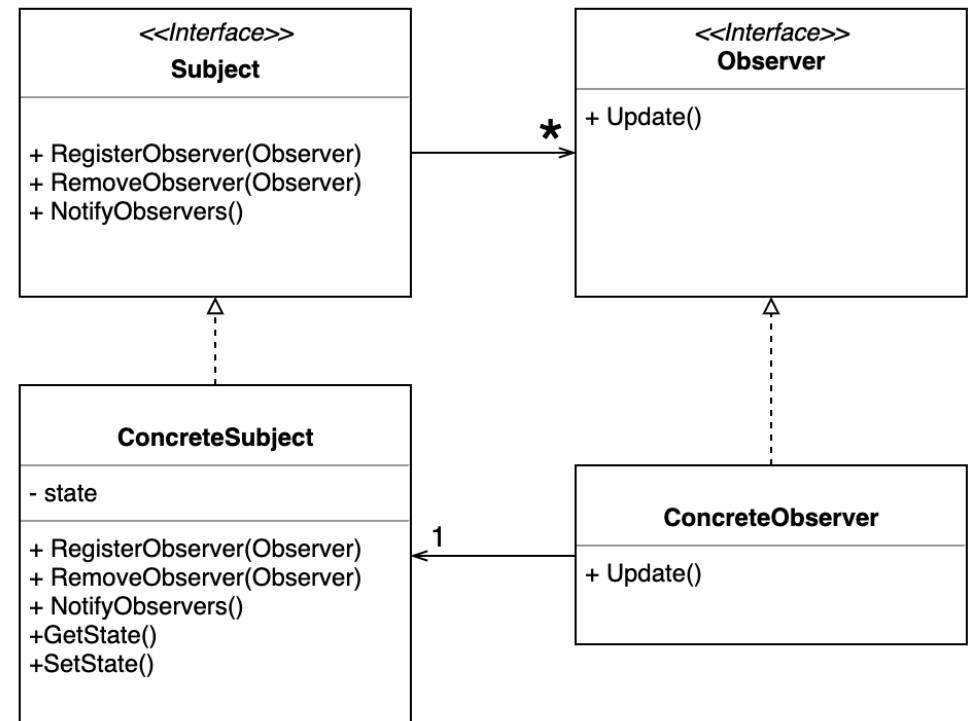
Update can take values or objects pushed by the subject:

```
void Update(int numberFromSubjectUpdate){
    this.number = numberFromSubjectUpdate;
}
void Update(DataFromSubject data){
    this.number = data.number;
    this.myString = data.dataString;
    ...
}
```

Update can also take Subject, and use GetState to pull from ConcreteSubject:

```
void Update(Subject subject){
    if(subject is ConcreteSubject){
        StateData stateData = ((ConcreteSubject)subject).GetState();
        this.number = stateData.number;
        this.myString = stateData.myString;
        ...
    }
}
```

But this implementation may couple observer with the implementation of ConcreteSubject, use it with caution!



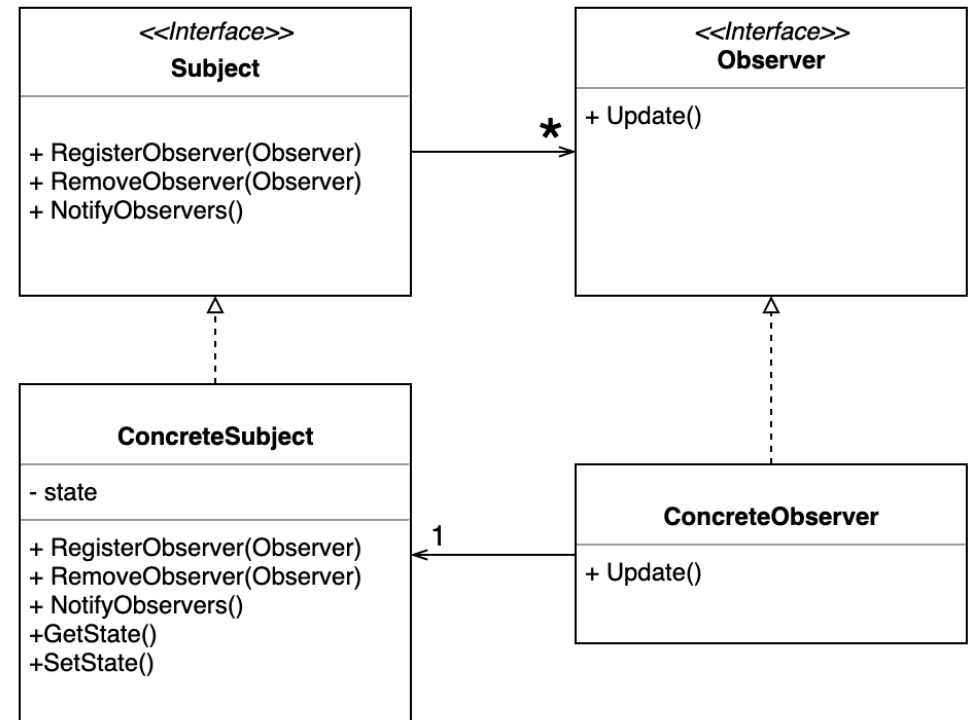
Observer Pattern

Subject and observer are loosely coupled:

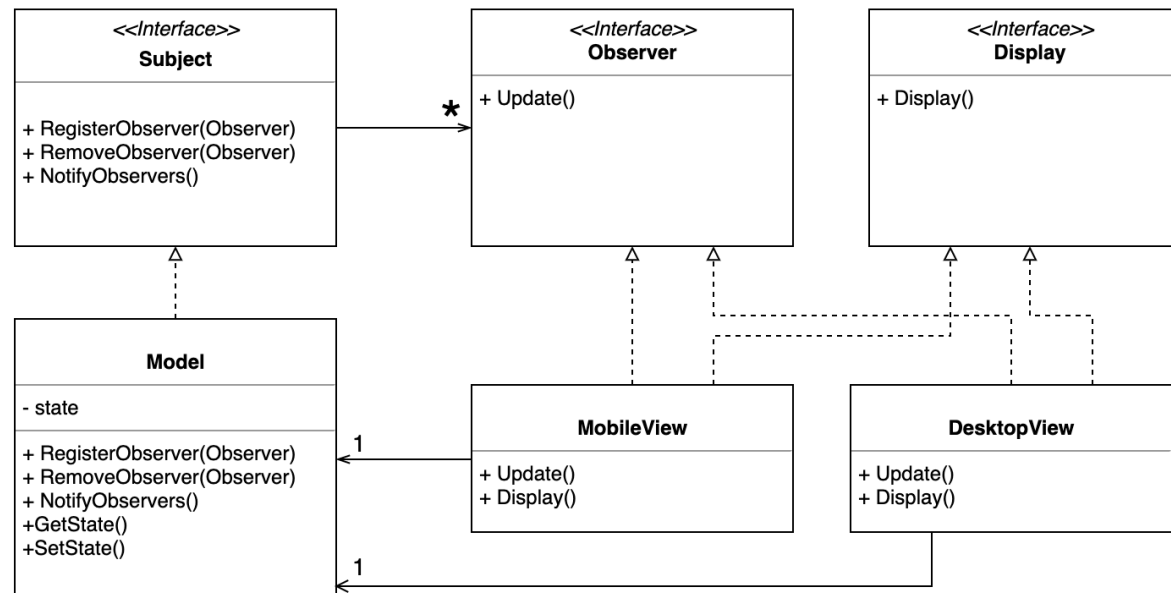
Subject only knows that the observer implements an interface, it interacts with the observer but doesn't know observer's implementation.

Perks of loose coupling:

- We can modify subject and observer classes separately without breaking the code.
- Observers can be registered and removed at runtime, this is because the only dependency of the subject is a collection of observer objects.
- We can add new concrete classes as observers without modifying subject at all, subject works the same as long as the new class implements the Observer interface.
- We can reuse subject and observer classes because they don't depend on each other.



OBSERVER PATTERN IN ACTION



OBSERVER PATTERN IN C#

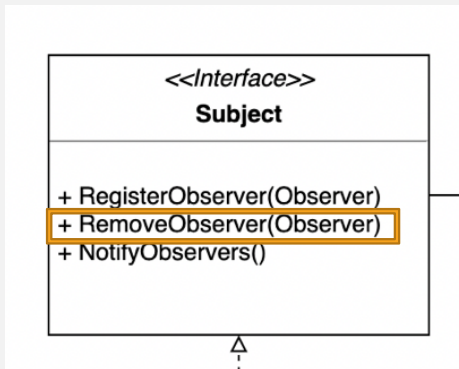
Microsoft's official guide, checking it out is strongly recommended!

<https://docs.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern>

In the .NET Framework, the observer design pattern is applied by implementing the generic `System.IObservable<T>` and `System.IObserver<T>` interfaces. The generic type parameter represents the type that provides notification information.

Let's take a look at the example.

Observer pattern in C#: IObservable (Publisher)



- The subject implements IObservable interface which has one method:
- `IObservable<T>.Subscribe`
- T represents the type that provides notification information, in this case the `BaggageInfo` class.
- The subject has a collection of observers(`IObserver`).
- The concrete subject decide when to notify the observers about state changes. (What is the method that notifies the observers in this example?)
- Something missing?

```
public class BaggageHandler : IObservable<BaggageInfo>
{
    private List<IObserver<BaggageInfo>> observers;
    private List<BaggageInfo> flights;

    public BaggageHandler()
    {
        observers = new List<IObserver<BaggageInfo>>();
        flights = new List<BaggageInfo>();
    }

    public IDisposable Subscribe(IObserver<BaggageInfo> observer)
    {
        // Check whether observer is already registered. If not, add it
        if (!observers.Contains(observer))
        {
            observers.Add(observer);
            // Provide observer with existing data.
            foreach (var item in flights)
                observer.OnNext(item);
        }
        return new Unsubscriber<BaggageInfo>(observers, observer);
    }

    // Called to indicate all baggage is now unloaded.
    public void BaggageStatus(int flightNo)
    {
        BaggageStatus(flightNo, String.Empty, 0);
    }

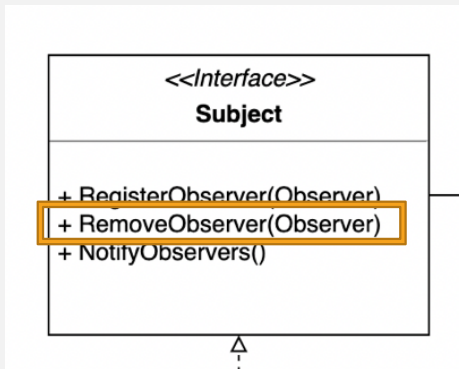
    public void BaggageStatus(int flightNo, string from, int carousel)
    {
        var info = new BaggageInfo(flightNo, from, carousel);

        // Carousel is assigned, so add new info object to list.
        if (carousel > 0 && !flights.Contains(info))
        {
            flights.Add(info);
            foreach (var observer in observers)
                observer.OnNext(info);
        }
        else if (carousel == 0)
        {
            // ...
        }
    }

    public void LastBaggageClaimed()
    {
        foreach (var observer in observers)
            observer.OnCompleted();

        observers.Clear();
    }
}
```

Observer pattern in C#: IDisposable



Removing observers is done this way:

- Make a class that handles unsubscribing, in this example: **Unsubscriber** which implements the **IDisposable** interface.
- **IDisposable** has one method: **Dispose**, it removes the observer from the subject's observer collection.
- When the observer **IObservable<T>.Subscribe**, it gets an instance of **IDisposable** as the return value, the observer keeps the reference of **IDisposable**.
- Observer call **IDisposable.Dispose** to unsubscribe from the subject.

What is good about this implementation?

```
internal class Unsubscriber<TestEventArgs> : IDisposable
{
    private List<IObserver<TestEventArgs>> _observers;
    private IObserver<TestEventArgs> _observer;

    internal Unsubscriber(List<IObserver<TestEventArgs>> observers, IObserver<TestEventArgs> observer)
    {
        this._observers = observers;
        this._observer = observer;
    }

    public void Dispose()
    {
        if (_observers.Contains(_observer))
            _observers.Remove(_observer);
    }
}
```


Observer pattern in C#: IObserver

The observer must implement three methods, all of which are called by the provider:

- IObserver<T>.OnNext, which supplies the observer with new or current information.
- IObserver<T>.OnError, which informs the observer that an error has occurred.
- IObserver<T>.OnCompleted, which indicates that the provider has finished sending notifications.

```
public class ArrivalsMonitor : IObserver<BaggageInfo>
{
    private string name;
    private List<string> flightInfos = new List<string>();
    private IDisposable cancellation;
    private string fmt = "{0,-20} {1,5} {2, 3}";

    public ArrivalsMonitor(string name)
    {
        if (String.IsNullOrEmpty(name))
            throw new ArgumentNullException("The observer must be assigned a name.");

        this.name = name;
    }

    public virtual void Subscribe(BaggageHandler provider)
    {
        cancellation = provider.Subscribe(this);
    }

    public virtual void Unsubscribe()
    {
        cancellation.Dispose();
        flightInfos.Clear();
    }

    public virtual void OnCompleted()
    {
        flightInfos.Clear();
    }

    // No implementation needed: Method is not called by the BaggageHandler class.
    public virtual void OnError(Exception e)
    {
        // No implementation.
    }

    // Update information.
    public virtual void OnNext(BaggageInfo info)
    {
        bool updated = false;

        // Flight has unloaded its baggage; remove from the monitor.
        if (info.Carousel == 0)
        else
        if (updated)
    }
}
```

Using IDisposable allows the observer to unsubscribe itself from the observable.

OBSERVER PATTERN IN C#

Implementing the IObservable and IObservable pattern is a (overkill but) very good practice to get a solid understanding on the observer pattern.

Observer pattern can also be implemented using delegate and events, which provides more flexibility.

Still remember delegate and events?

PREVIOUSLY IN SOFTWARE ARCHITECTURE

DELEGATE AND EVENT

- A delegate specifies a particular method signature, references to one or more methods can be added to a delegate instance.
- Event is a special kind of delegate that can't be called outside of the class.
- Delegate and event are fundamental to MVC pattern.
- Let's just get to the examples to clear things up.

OBSERVER PATTERN IN C#

A delegate is a reference type that can be used to encapsulate a named or an anonymous method.

Delegates are the basis for Events.

The delegate must be instantiated with a method that has a compatible return type and input parameters.

Therefore we can use it to make sure the observers implement the required method to be called by the subject.



```
public delegate void Update(DataPushedBySubject dataPushed);
```

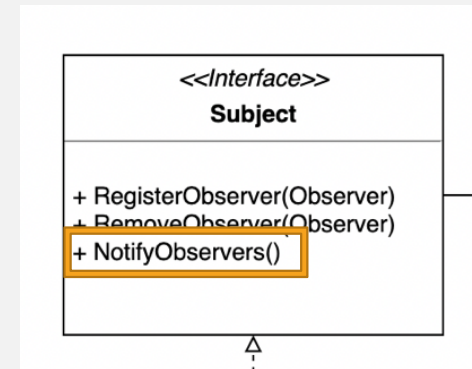
OBSERVER PATTERN IN C#

Event?.Invoke() will call all the methods that subscribe to the event.

```
public class Subject
{
    public event Update OnNotifyObservers;

    public void NotifyObservers()
    {
        OnNotifyObservers?.Invoke(new DataPushedBySubject());
        /*This is the same as
        if(OnNotifyObservers != null)
        {
            OnNotifyObservers(new DataPushedBySubject());
        }*/
    }
}
```

=



OBSERVER PATTERN IN C#

`+=` to subscribe.

`-=` to unsubscribe.

IMPORTANT: Unsubscribe from events before you dispose of a subscriber object. Until you unsubscribe from an event, the multicast delegate that underlies the event in the publishing object has a reference to the delegate that encapsulates the subscriber's event handler. As long as the publishing object holds that reference, garbage collection will not delete your subscriber object.

```
public class Observer
{
    private Subject subject;

    public Observer(Subject iSubject)
    {
        subject = iSubject;
        subject.OnNotifyObservers += Update;
    }

    public void Update(DataPushedBySubject dataPushed)
    {
        //do something with the data
    }

    public void Unsubscribe()
    {
        subject.OnNotifyObservers -= Update;
    }
}
```

Difference between delegate and event:

```
public delegate void Update(DataPushedBySubject dataPushed);

public class Subject
{
    public event Update OnNotifyObservers;
    public Update DelegateOnNotifyObservers;

    public void NotifyObservers()
    {
        OnNotifyObservers?.Invoke(new DataPushedBySubject());
        /*This is the same as
        if(OnNotifyObservers != null)
        {
            OnNotifyObservers(new DataPushedBySubject());
        }*/
    }
}
```

```
public Observer(Subject iSubject)
{
    subject = iSubject;
    subject.OnNotifyObservers += Update;

    subject.DelegateOnNotifyObservers = Update;
    subject.OnNotifyObservers = Update;
}

public void Update(DataPushedBySubject dataPushed)
{
    //do something with dataPushed
}

public void Unsubscribe()
{
    subject.OnNotifyObservers -= Update;
}
```

E DandE.Update Subject.OnNotifyObservers

The event 'Subject.OnNotifyObservers' can only appear on the left hand side of += or -= (except when used from within the type 'Subject')

Event can't be called outside of its class.

Event works with += and -= but not =, which prevents the subscribers to be accidentally removed by assignment =.

```
public class AnotherClass
{
    private Subject subject;

    public AnotherClass(Subject iSubject)
    {
        subject = iSubject;
    }

    public void TryToCallEventAndDelegate()
    {
        subject.DelegateOnNotifyObservers?.Invoke(new DataPushedBySubject());
        subject.OnNotifyObservers?.Invoke(new DataPushedBySubject());
    }
}
```

E Update Subject.OnNotifyObservers

The event 'Subject.OnNotifyObservers' can only appear on the left hand side of += or -= (except when used from within the type 'Subject')

EVENT QUEUE

The observer pattern is very useful, and in most occasions it works very well.

But when you have multiple subjects and multiple subscribers all working together, things could get messy...

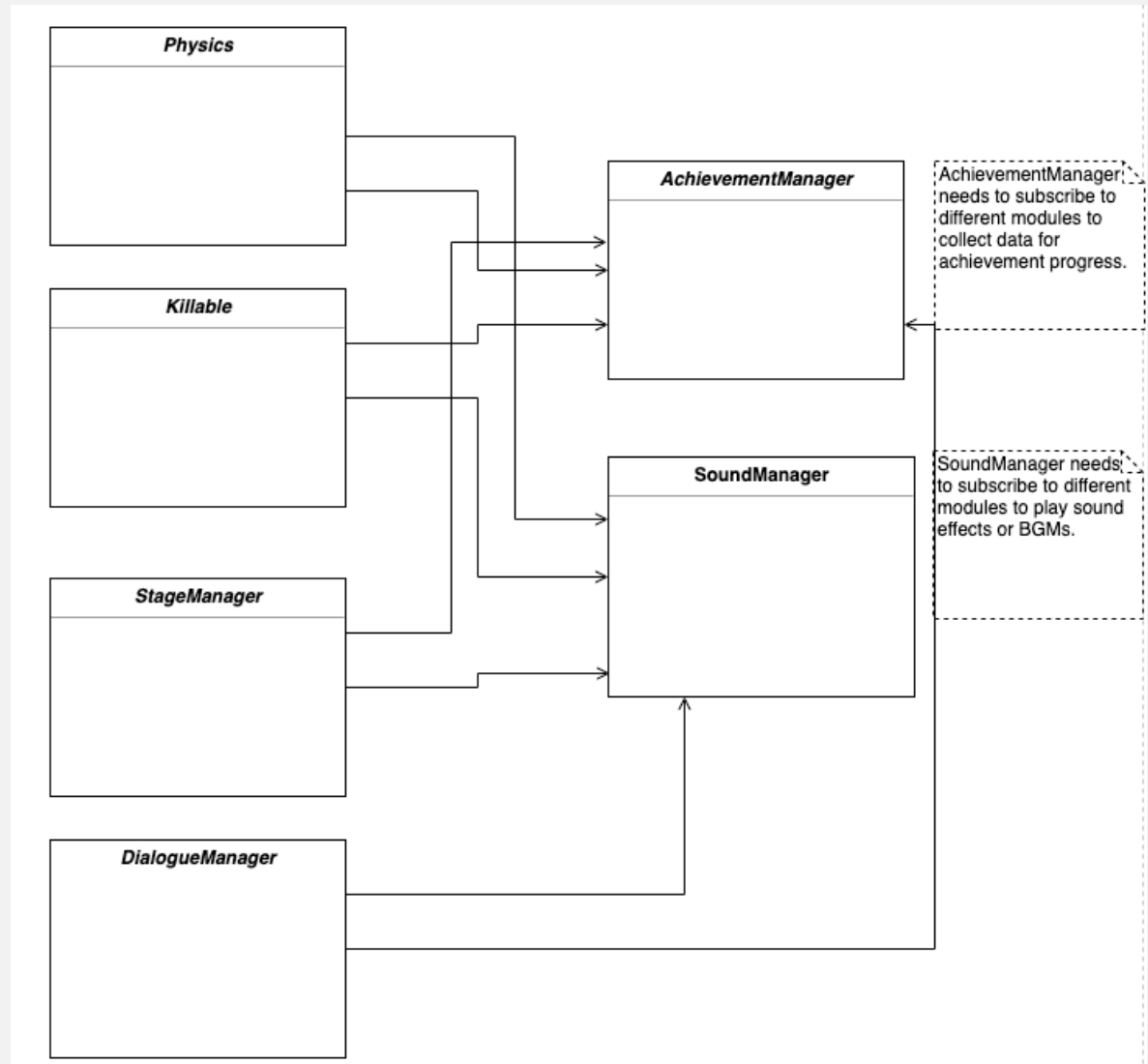
Especially when you have needy modules that need to subscribe to many subjects like an achievement manager.

Problems:

Achievement Manager depends on too many things, it's almost impossible to observe them all (How to get the notification like 'Final boss killed' in the first stage?)

Sound manager needs to load streamed data in run time, observer pattern is synchronous, after the subject notifies the sound manager, it could take too long wait for the sound manager to finish its callback to move on.

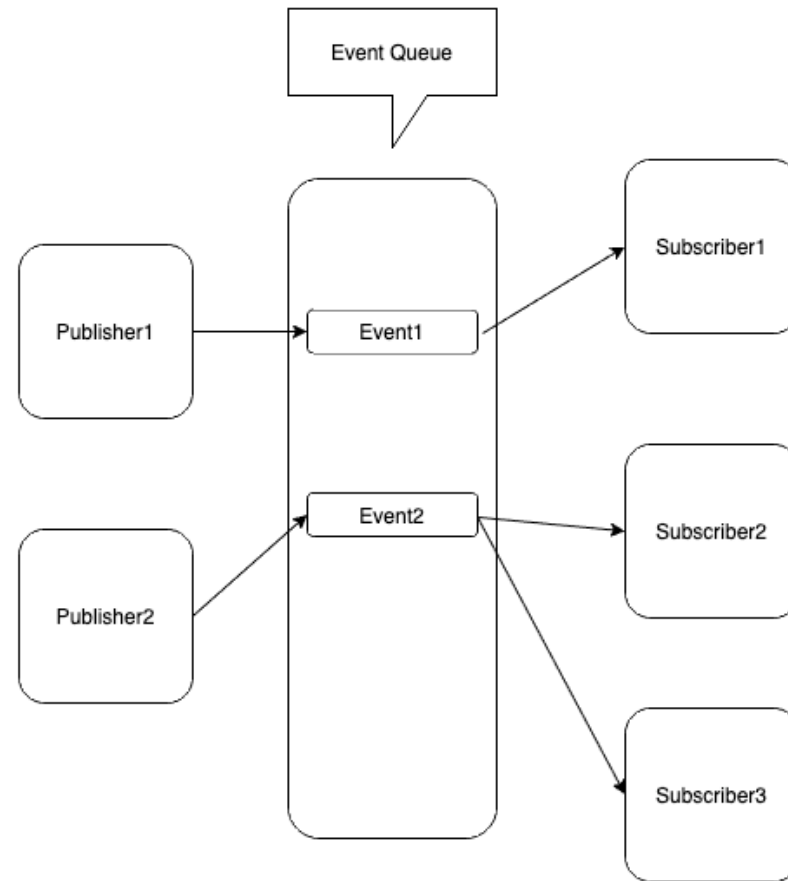
It's too many design puzzles with more modules like these working together.



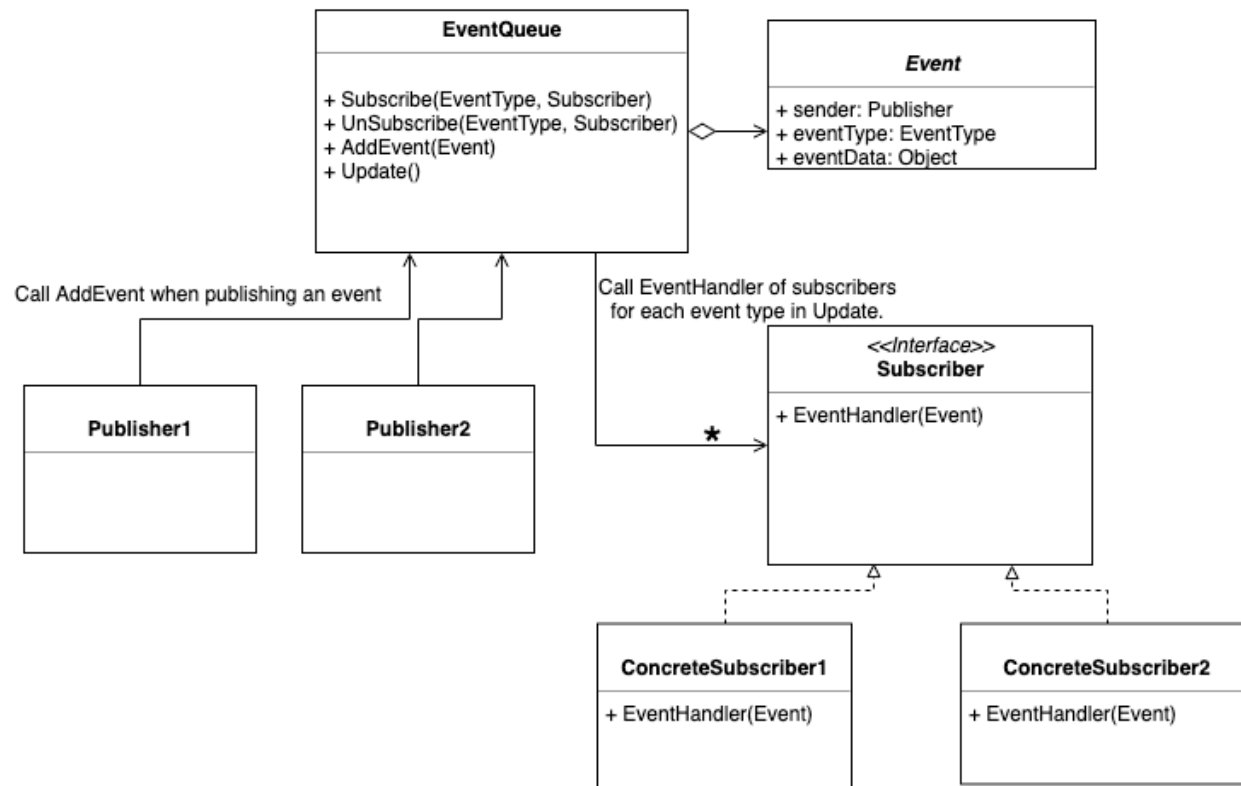
EVENT QUEUE

Event queue, a.k.a Event aggregator, message queue, is a pattern to further decouple subjects(publisher) and observers(subscribers) by adding a queue to store the notifications and direct them to the corresponding subscribers.

Unlike the observer pattern, in this pattern the subjects and observers don't have to know anything about each other, they are decoupled and only communicate via the event queue.



EVENT QUEUE: IMPLEMENTATION



I. Define an enum for EventType and a base EventData class.

Almost real code:

```
public enum EventType
{
    LOADINVENTORY = 0,
    PURCHASEBEGIN,
    PURCHASESUCCESSFUL,
}
```

```
public class EventData
{
    public readonly EventType eventType;

    public EventData(EventType type)
    {
        eventType = type;
    }
}
```

```
public class PurchaseBeginEventData : EventData
{
    public readonly Item item;
    public readonly int price;

    public PurchaseBeginEventData(Item pItem, int pPrice) : base(EventType.PURCHASEBEGIN)
    {
        item = pItem;
        price = pPrice;
    }
}
```

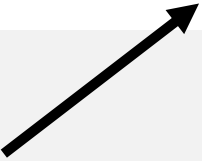
2. Define a delegate for EventHandler.

3. Use a Hashtable or Dictionary to store EventType and EventHandler pairs.

4. Use a list for a collection of events. (A.K.A. event queue, event hub, event aggregator, etc.)

Almost real code:

```
public delegate void EventHandler(EventData eventData);  
  
private Dictionary<EventType, EventHandler> subscriberDict;  
private List<EventData> eventList;
```



This is our humble event queue.

5 Use `subscriberDictionary[eventType] +=` to add new subscribers to a certain type, `subscriberDict[eventType] -=` to unsubscribe.

Almost real code:

```
public void Subscribe(EventType eventType, EventHandler eventHandler)
{
    if (!subscriberDict.ContainsKey(eventType))
    {
        EventHandler handler = null;
        subscriberDict.Add(eventType, handler);
    }

    subscriberDict[eventType] += eventHandler;
}

public void UnSubscribe(EventType eventType, EventHandler eventHandler)
{
    if (subscriberDict.ContainsKey(eventType))
    {
        //If eventHandler is not added in the invocationList, the following code would be just ignored
        subscriberDict[eventType] -= eventHandler;
    }
    else
    {
        Console.WriteLine("Warning: Event type " + eventType.ToString() +
            " doesn't exist in the event manager's subscriber dictionary");
    }
}
```

In subscriber classes:

...

```
eventQueue.Subscribe(EventType.PURCHASEBEGIN, OnPurchaseItem);
```

//The subscriber doesn't know who published the event, because it subscribes through the event queue.

...

```
public void OnPurchaseItem(EventData eventData){...} //Same signature as the delegate EventHandler.
```

6. Make a public function to add event to the event list.

Almost real code:

```
public void AddEvent(EventData eventData)
{
    if (!Enum.IsDefined(typeof(EventType), eventData.eventType))
    {
        throw new ArgumentOutOfRangeException("eventData.eventType", "EventType is invalid.");
    }
    eventList.Add(eventData);
}
```

In publisher classes:

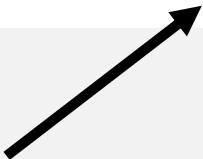
```
...
eventQueue.AddEvent(new PurchaseBeginEventData(selectedItem, selectedItemPrice);
//This will add an event to the event queue, the publisher doesn't have to know who subscribed to
//its event.
...
```


7. Regularly iterate through the event queue, invoke the event, and then remove the event.

Almost real code:

```
public void PublishEvents()
{
    for(int i = eventList.Count - 1; i >=0; i--)
    {
        eventData data = eventList[i];
        if (subscriberDict.ContainsKey(data.eventType))
        {
            subscriberDict[data.eventType]?.Invoke(data);
        }

        else
        {
            Console.WriteLine("Warning: Event type " + data.eventType.ToString() +
                             " doesn't exist in the event manager's subscriber dictionary");
        }
        eventList.Remove(data);
    }
}
```



In Unity, you could call this method in an Update().

EVENT QUEUE: IMPLEMENTATION

1. **Define an enum for EventType and a base EventData class.**
2. **Define a delegate for EventHandler.**
3. **Use a Hashtable or Dictionary to store EventType and EventHandler pairs.**
4. **Use a list for a collection of events. (A.K.A. event queue, event hub, event aggregator, etc.)**
5. **Use subscriberDictionary[eventType]+= to add new subscribers to a certain type, subscriberDict[eventType]-= to unsubscribe.**
6. **Make a public function to add event to the event list.**
7. **Regularly iterate through the event queue, invoke the event, and then remove the event.**

EVENT QUEUE: WITH GREAT POWER COMES GREAT RESPONSIBILITY

Event queue is a powerful decoupling pattern, but keep in mind that powerful != good:

- The pattern has a very wide-reaching effect on your architecture, meaning you have to design very carefully because too many modules are involved.
- The pattern sometimes require a global event queue, anything global is dangerous.
- The pattern can cause a dead loop: A send an event to the event queue, the event was received by B, B react on the event and the event is received by A...

So use it with caution!

For the shop model assignment, it should be easy to keep the event queue under control because there are mostly shop-related events.

For larger projects, design your event system carefully, think about combining observer with event queue, or different event queues for different categories of events.(e.g. battle events, physics events, shop events, UI events...)

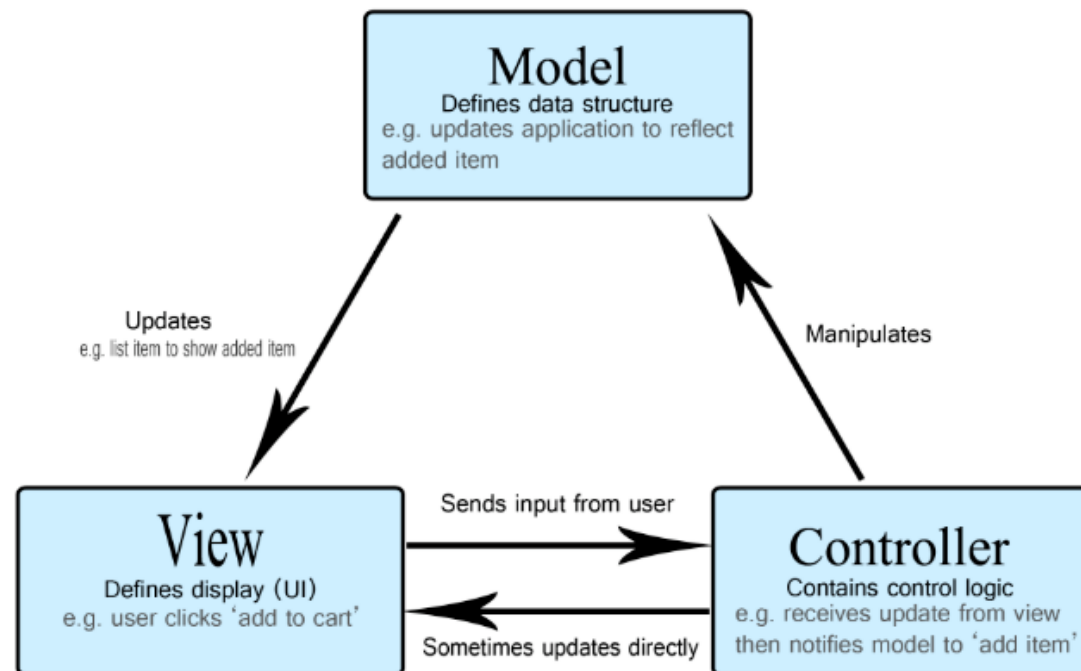
MODEL AND VIEW DECOUPLING

Pub/Sub patterns like observer and event queue are essential to decouple model and view.

Model notifies View to update when model state changes.

In the assignment project, if you don't decouple model and view enough, it's almost impossible to make your shop model works with different views.

MVC



LAB 3 ASSIGNMENTS

- Code review for SA shop version I.
- Build version 2: apply observer/event queue pattern to your shop model, decouple your model and view so that your model works in 2 different views: a grid view and a list view.

ASSIGNMENT TIPS:

- Example code/pseudo code of design patterns can be found at:
- <http://gameprogrammingpatterns.com/contents.html>
- <https://refactoring.guru/design-patterns>
- Or if you prefer videos:
- <https://www.youtube.com/playlist?list=PLrhzvlcii6GNjpARdnO4ueTUAVR9eMBpc>