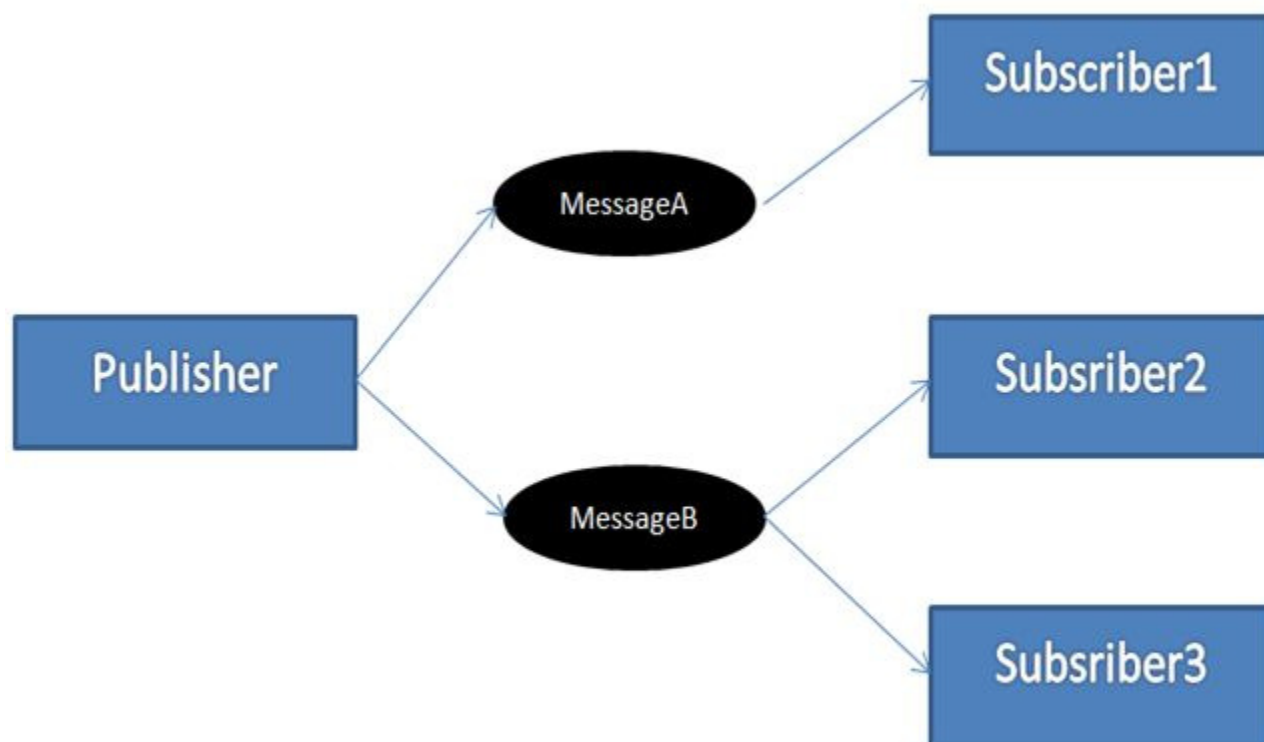
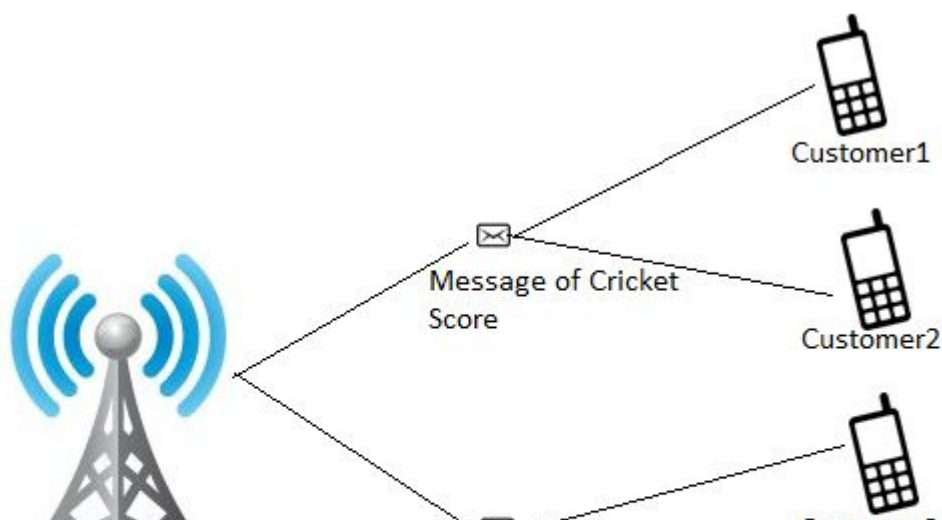


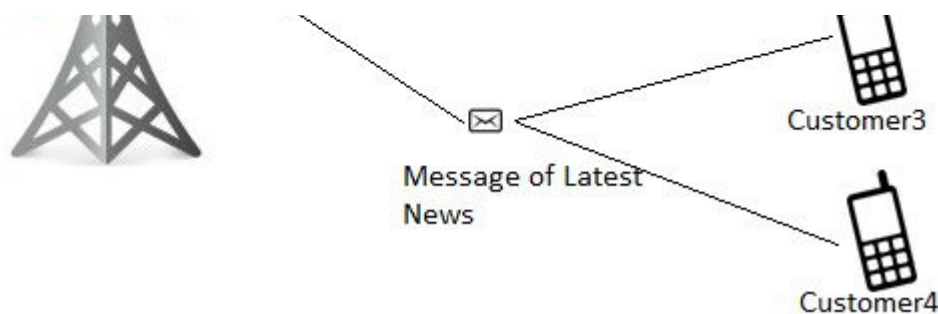
Publisher/Subscriber pattern

The Publisher/Subscriber pattern is one of the variations of the Observer designer pattern introduced by the GOF in software development. In the Publisher/Subscriber pattern a publisher (entity responsible for publishing a message) publishes a message and there are one or more Subscribers (entity subscribing, in other words intested in a message of a specified message type) who capture the published message. The following image describes the senario of the publisher and subscriber pattern where the Publisher publishes two types of messages (MessageA and MessageB) and Subscribers to the messages receive the messages they are subscribed to (Subscriber1 captures MessageA and Subscriber2 and Subscriber3 captures MessageB).



To understand this consider a real-life scenario where Mobile operators are sending messages to their customers.





As in the preceding image a Mobile operator publishes (broadcasts) messages (a message of a Cricket score and a message of latest news) and the messages are captured by the customer cells subscribing to the messages (Customer1 and Customer2 captures the Cricket score messages and Customer3 and Customer4 captures the latest news messages).

Implementation with Event

One way to do the Publisher/Subscriber pattern in an application is to use events and delegates, in other words using a framework. The following is a detailed description of a publisher/subscriber implementation

message. The following is a class that represents a message that is published by a Publisher and is captured by an interested Subscriber.

```

01. public class MessageArgument<T> : EventArgs
02. {
03.     public T Message { get; set; }
04.     public MessageArgument(T message)
05.     {
06.         Message = message;
07.     }
08. }

```

In technical terms a MessageArgument is a generic class so an instance of this class can be of any type that is represented by a T template type.

Publisher: As already described above in the definition, a Publisher is responsible for publishing messages of various types.

```

01. public interface IPublisher<T>
02. {
03.     event EventHandler<MessageArgument<T>> DataPublisher;
04.     void OnDataPublisher(MessageArgument<T> args);
05.     void PublishData(T data);
06. }
07.
08. public class Publisher<T> : IPublisher<T>
09. {
10.     //Defined datapublisher event
11.     public event EventHandler<MessageArgument<T>> DataPublisher;
12.
13.     public void OnDataPublisher(MessageArgument<T> args)
14.     {
15.         var handler = DataPublisher;
16.         if (handler != null)
17.             handler(this, args);
18.     }
19.
20.
21.     public void PublishData(T data)

```

```
18.     }
19.
20.
21.     public void PublishData(T data)
22.     {
23.         MessageArgument<T> message = (MessageArgument<T>)Activator.CreateInstance(typeof(Message
24.         OnDataPublisher(message);
25.     }
26. }
```

Technically Publisher is a generic class that inherits the IPublisher interface. Since Publisher is a generic class an instance of it can be of any type represented by a T template type. A Publisher instance is created of a selected type and publishes that type of message only. To publish a different type of message a different type of Publisher instance must be created. To understand better read the following about how to use a publisher in client class code.

The Publisher class provides the event DataPublisher that a subscriber attaches to listen for messages.

PublishData is a publisher class method that publishes data to Subscribers.

Subscriber: Subscriber captures messages of the type it is interested in.

```
01. public class Subscriber<T>
02. {
03.     public IPublisher<T> Publisher { get; private set; }
04.     public Subscriber(IPublisher<T> publisher)
05.     {
06.         Publisher = publisher;
07.     }
08. }
```

Technically a Subscriber is a generic class that allows creation of multiple instances of a subscriber and each subscriber subscribes to the messages it is interested in using a publisher.

A Subscriber passes an instance of a specific type of publisher to capture messages published by that Publisher.

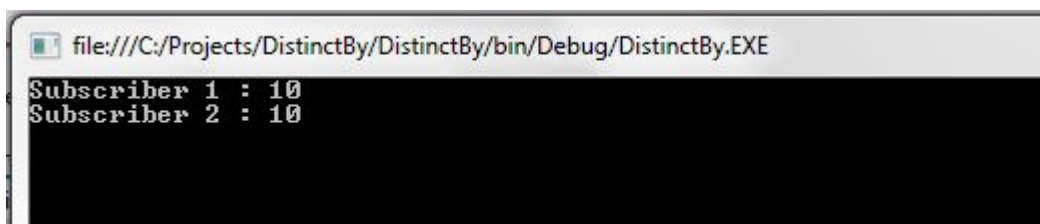
How it works

```
01. public class Client
02. {
03.     private readonly IPublisher<int> IntPublisher;
04.     private readonly Subscriber<int> IntSubscriber1;
05.     private readonly Subscriber<int> IntSubscriber2;
06.
07.     public Client()
08.     {
09.         IntPublisher = new Publisher<int>(); //create publisher of type integer
10.
11.         IntSubscriber1 = new Subscriber<int>(IntPublisher);
12.         //subscriber 1 subscribe to integer publisher
13.         IntSubscriber1.Publisher.DataPublisher += publisher_DataPublisher1;
14.         //event method to listen publish data
15.
16.         IntSubscriber2 = new Subscriber<int>(IntPublisher);
17.         //subscriber 2 subscribe to interger publisher
18.         IntSubscriber2.Publisher.DataPublisher += publisher_DataPublisher2;
19.         //event method to listen publish data
20.
21.         IntPublisher.PublishData(10); // publisher publish message
22.     }
23. }
```

```
16.         IntPublisher.PublishData(10); // publisher publish message
17.     }
18.
19.
20.     void publisher_DataPublisher1(object sender, MessageArgument<int> e)
21.     {
22.         Console.WriteLine("Subscriber 1 : " + e.Message);
23.     }
24.
25.     void publisher_DataPublisher2(object sender, MessageArgument<int> e)
26.     {
27.         Console.WriteLine("Subscriber 2 : " + e.Message);
28.     }
29. }
```

As you can see in the preceding code, Client is a class that creates a publisher and subscriber. The Client class creates a Publisher of integer type (in practice you can create any type of publisher) and creates two Subscriber classes that subscribe to the publisher message by attaching a DataPublisher event that is provided by the Publisher class.

So when you create an instance of the Client class you will receive the following output:



```
file:///C:/Projects/DistinctBy/DistinctBy/bin/Debug/DistinctBy.EXE
Subscriber 1 : 10
Subscriber 2 : 10
```

So as per the output the Publisher of integer type publishes the message "10" and two Subscribers subscribing to the publisher capture and display the message to output.

In a practical scenario, in other words in an actual application, one must create all the publishers during the application start, in other words at the entrypoint of the app and pass the instances of publishers when creating subscribers.

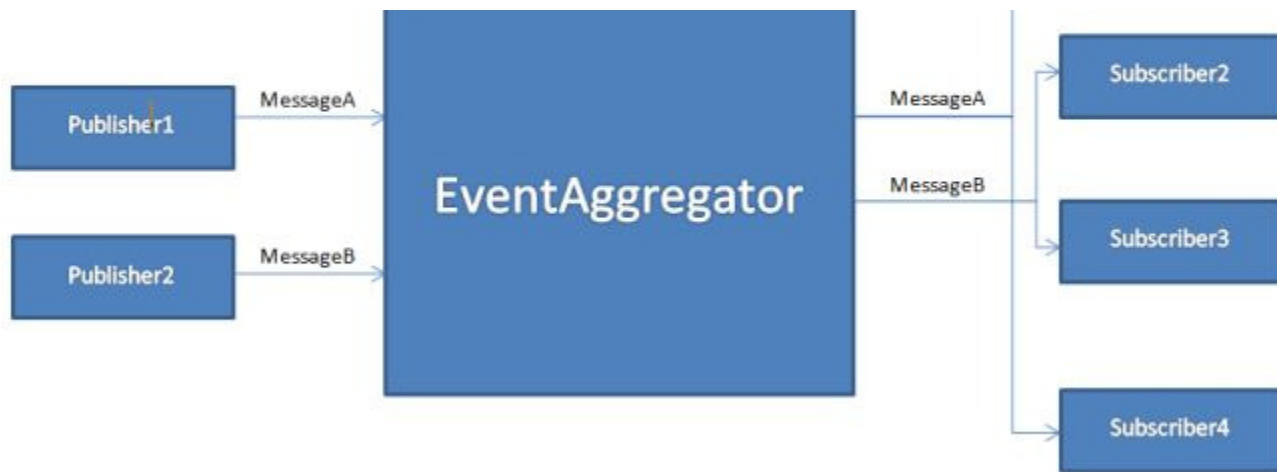
For example in a Windows application create a publisher in the Main() method and in a Web Application create a publisher in the Application_Start method of Global.asax use Dependency Injection to register your publisher and use a container to create it when needed.

Once you have created them you can pass the publisher in the subscriber as done in the preceding client class code.

Implementation with EventAggregator

EventAggregator: by the name one can easily say it aggregates events. An Publisher/Subscriber EventAggregator works as a HUB whose task is to aggregate all the published messages and send the message to the interested subscribers.





As you can see in the preceding image, an EventAggregator comes as a HUB between a publisher and a subscriber. It works like this:

1. Publisher publishes a message.
2. EventAggregator receives a message sent by publishers.
3. EventAggregator gets a list of all subscriber interested messages.
4. EventAggregator sends the messages to the interested subscriber.

EventAggregator Implementation

Subscription: It is a class to create subscription tokens. When a Subscriber subscribes to interested message types via EventAggregator the EventAggregator returns a subscription token that is further used by the subscriber to keep track of its subscriptions.

```

01. //Does used by EventAggregator to reserve subscription
02. public class Subscription<Tmessage> : IDisposable
03. {
04.     public Action<Tmessage> Action { get; private set; }
05.     private readonly EventAggregator EventAggregator;
06.     private bool isDisposed;
07.     public Subscription(Action<Tmessage> action, EventAggregator eventAggregator)
08.     {
09.         Action = action;
10.         EventAggregator = eventAggregator;
11.     }
12.
13.     ~Subscription()
14.     {
15.         if (!isDisposed)
16.             Dispose();
17.     }
18.
19.     public void Dispose()
20.     {
21.         EventAggregator.UnSubscribe(this);
22.         isDisposed = true;
23.     }
24. }
  
```

```
23.     }
24. }
```

EventAggregator

```
01. public class EventAggregator
02. {
03.     private Dictionary<Type, IList> subscriber;
04.
05.     public EventAggregator()
06.     {
07.         subscriber = new Dictionary<Type, IList>();
08.     }
09.
10.     public void Publish<TMessageType>(TMessageType message)
11.     {
12.         Type t = typeof(TMessageType);
13.         IList actionlst;
14.         if (subscriber.ContainsKey(t))
15.         {
16.             actionlst = new List<Subscription<TMessageType>>
17. (subscriber[t].Cast<Subscription<TMessageType>>());
18.
19.             foreach (Subscription<TMessageType> a in actionlst)
20.             {
21.                 a.Action(message);
22.             }
23.         }
24.
25.     public Subscription<TMessageType> Subscribe<TMessageType>(Action<TMessageType> action)
26.     {
27.         Type t = typeof(TMessageType);
28.         IList actionlst;
29.         var actiondetail = new Subscription<TMessageType>(action, this);
30.
31.         if (!subscriber.TryGetValue(t, out actionlst))
32.         {
33.             actionlst = new List<Subscription<TMessageType>>();
34.             actionlst.Add(actiondetail);
35.             subscriber.Add(t, actionlst);
36.         }
37.         else
38.         {
39.             actionlst.Add(actiondetail);
40.         }
41.
42.         return actiondetail;
43.     }
44.
45.     public void UnSbscribe<TMessageType>(Subscription<TMessageType> subscription)
46.     {
47.         Type t = typeof(TMessageType);
48.         if (subscriber.ContainsKey(t))
49.         {
50.             subscriber[t].Remove(subscription);
51.         }
52.     }
53.
54. }
```

In the preceding code

Dictionary<Type, IList> subscriber: is a dictionary in which Type is the type of message and IList is a list of actions. So it holds a list of actions mapped to specific Message Types.

a list of actions. So it holds a list of actions mapped to specific Message Types.

public void Publish<TMessageType>(TMessageType message): is a method to publish messages. As in the code, this method receives messages as input then filters out a list of all subscribers by message type and publishes messages to the subscriber.

public Subscription<TMessageType> Subscribe<TMessageType>(Action<TMessageType> action): is a method for subscribing to interested message types. As in the code this method receives an Action delegate as input. It maps an Action to a specific MessageType, in other words it creates an entry for message type if not present in the dictionary and maps a Subscription object (that wraps an Action) to a message entry.

public void UnSubscribe<TMessageType>(Subscription<TMessageType> subscription): is a method for unsubscribing from a specific message type. It receives a Subscription object as input and removes an object from the dictionary.

How To Use

```
01. static void Main(string[] args)
02. {
03.     EventAggregator eve = new EventAggregator();
04.     Publisher pub = new Publisher(eve);
05.     Subscriber sub = new Subscriber(eve);
06.
07.     pub.PublishMessage();
08.
09.     Console.ReadLine();
10.
11. }
```

The preceding code shows, first create an instance of EventAggregator that passes as an argument to a publisher and subscriber, then the publisher publishes message.

Publisher: Code of Publisher class that shows how a publisher publishes a message using EventAggregator.

```
01. public class Publisher
02. {
03.     EventAggregator EventAggregator;
04.     public Publisher(EventAggregator eventAggregator)
05.     {
06.         EventAggregator = eventAggregator;
07.     }
08.
09.     public void PublishMessage()
10.     {
11.         EventAggregator.Publish(new Mymessage());
12.         EventAggregator.Publish(10);
13.     }
14. }
```

Subscriber: Code of the Subscriber class that shows a subscriber subscribing to messages that it is interested in using EventAggregator.

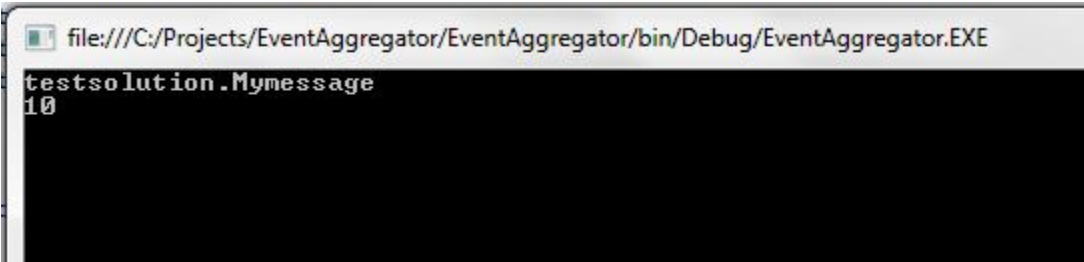
```
01. public class Subscriber
02. {
03.     Subscription<Mymessage> myMessageToken;
04.     Subscription<int> intToken;
```



```

02.  {
03.      Subscription<Mymessage> myMessageToken;
04.      Subscription<int> intToken;
05.      EventAggregator eventAggregator;
06.
07.      public Subscriber(EventAggregator eve)
08.      {
09.          eventAggregator = eve;
10.          eve.Subscribe<Mymessage>(this.Test);
11.          eve.Subscribe<int>(this.IntTest);
12.      }
13.
14.      private void IntTest(int obj)
15.      {
16.          Console.WriteLine(obj);
17.          eventAggregator.UnSbscribe(intToken);
18.      }
19.
20.      private void Test(Mymessage test)
21.      {
22.          Console.WriteLine(test.ToString());
23.          eventAggregator.UnSbscribe(myMessageToken);
24.      }
25.  }
    
```

Output



Note:

In practical scenarios, in other words in an actual application, one must create an EventAggregator at the application start point, in other words at the entrypoint of the app and pass an instance of a publisher and subscriber.

For example in a Windows application create an EventAggregator in the Main() Method as in the preceding example code then in a Web Application create a publisher in the Appication_Start method of Global.asax or use Dependency Injection to register your publisher and use a container to create when needed.

Event/Delegate Vs. EventAggregator

Difference between Event/Delegate and EventAggregator is:

Event/Delegate	EventAggregator
For publishing Different type of message there is need of creating different type of Publisher	
For Example: Publisher - for integer type publisher Publisher - for string type publisher So the if class want to publish different type of messages it require to	As EventAggregator works as HUB there is no need create more than one Instance of Eventaggregator and publisher just need to consume it.

<p>for string type publisher So the if class want to publish different type of messages it require to consume or create different type of publisher.</p> <p>To publish integer abd string type message publisher class will be Publisher (Publisher intPublisher, Publisher strPublisher).</p>	<p>As EventAggregator works as HUB there is no need create more than one Instance of Eventaggregator and publisher just need to consume it. Example : Publisher(EventAggregator event Aggregator)</p>
<p>Tight Coupling bettween Publisher and Subscriber. In this pattern Subscriber require to know publisher as they subscribe to event of publisher.</p>	<p>Loose Coupling between Publisher and Subscriber. In this pattern as EventAggregator is midator Publisher and Subscriber don't know each other they just need to know EventAggregator.</p>

Conclusion

In my point of view, an Event/Delegate is easy to implement and good for small projects or in a project where there are fewer Publishers and Subscribers. An EventAggregator is suitable for large projects or projects with a large number of Publishers and Subscirbers.

But I think it's always good to use EventAggregator because it offers loose coupling.