

4/5/2009



MOBILECONTRIB

## MOBILE CLIENT SOFTWARE FACTORY: EVENT AGGREGATOR GETTING STARTED

Event Aggregator Getting Started | Andy Wigley

## Contents

Introduction .....	3
Event Publish – Subscribe Between Modules.....	4
Defining Event Classes .....	6
Publishing and Subscribing Events.....	6
Performance, Thread Usage and Filtering .....	7
Summary .....	9

## Introduction

In a well-designed software system, a module has a clearly defined purpose and only interacts with other modules when they offer services (such as logging or security) that the module wishes to use. Many modules however consume data or messages, perform some processing and then need to publish an alert out to the system to report something of interest. For example, a module monitoring an incoming data stream on a serial port from some external equipment may detect a peak value of some data and needs to alert the rest of the system. This is done by the module publishing an event.

Other modules may be interested in receiving notifications of an event so that they can perform some processing themselves to react to that event. These modules subscribe to the event and thus are notified when the event fires. Note that the publisher of the event should not have any knowledge – or care – about who is subscribing to the event. It just does ‘fire and forget’, and usually should not contain any logic that has a dependency on some action some other module performs.

The .NET Framework has always included first class support for the observer pattern (sometimes known as publish/subscribe) through the use of events and delegates. The standard way for an object to provide automatic notifications to its dependents (or observers), is for it to simply expose a public event, and any subscriber classes can get a reference to an instance of the publisher class and subscribe to that event and add an event handler for that event. For example:

```
public class ClassA
{
    public event EventHandler ClassAEvent;

    public void DoSomeWork()
    {
        // Do something...
    }
}

public class ClassB
{
    public void Initialise()
    {
        // Create an instance of ClassA
        ClassA classA = new ClassA();

        // Subscribe to the event
        classA.ClassAEvent += new
            EventHandler(classA_ClassAEvent);
    }

    void classA_ClassAEvent(object sender, EventArgs e)
    {
        // Do something when ClassA's event fires
    }
}
```

```

    // ...
}

```

While this is effective, the result is that ClassA and ClassB are tightly coupled, and could not be developed or tested independently.

In a loosely coupled architecture such as one built using the ContainerModel Dependency Injection container, modules need to be able to communicate using an eventing mechanism, but must not need the other module to be present during development or testing. This is achieved by using a different application block, the **EventAggregator**.

## Getting Started with the Event Aggregator Mobile Application Block

To use the Event Aggregator, you need to include the following project from the MobileContrib.codeplex.com project in your application:

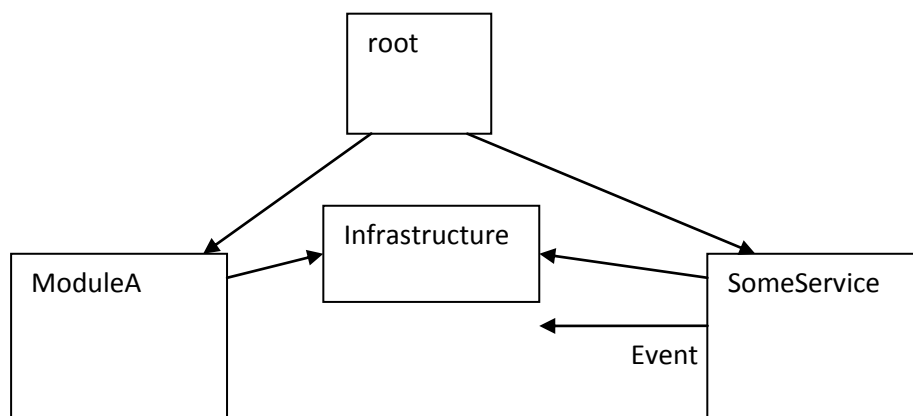
- Microsoft.Practices.Composite – this contains the Event Aggregator

Although not required to make use the of the Event Aggregator, most applications that require its services will be built using an architecture that promotes loose coupling between modules. The EventAggregation Quickstart sample project uses the ContainerModel dependency injection container:

- ContainerModel – included in the mobile application blocks from mobile.codeplex.com

## Event Publish – Subscribe Between Modules

The EventAggregation Quickstart example application consists of a root module, a module called ModuleA, and another module called SomeService that does not have a UI but performs some long-running processing. SomeService publishes events that ModuleA subscribes to. The following diagram shows the basic building blocks of this application, and also includes an Infrastructure project that is used to define business objects that are used cross-module.



The EventAggregator app block includes the EventAggregator class which implements the IEventAggregator interface; you need an instance of a class implementing IEventAggregator in your application to provide event publish – subscribe services to your modules. The sample registers this class in the root D.I. container in the ApplicationRoot class which is called during application startup from Program.cs:

```
/// <summary>
/// Initializes the root container.
/// </summary>
public static void Initialize()
{
    // Create the root container
    var container = new Container();
    ...

    // Register the Event Aggregator
    container
        .Register<IEventAggregator>(c => new EventAggregator());
    ...
}
```

ModuleA registers its own classes with the D.I. container in a similar way in the ModuleAConfigurator class, which again executes during application startup. This includes the following to register the Presenter class for the module, the constructor of which takes an IEventAggregator instance. Therefore, when the Presenter class is constructed by the D.I. container, it will be ‘injected with’ a reference to the EventAggregator instance that was registered in the ApplicationRoot.

```
private void RegisterTypes(Container container)
{
    ...

    // Register the factory function for the Presenter.
    container.Register(c => new ModuleAPresenter(
        c.Resolve<IModuleAView>(),
        c.Resolve<IRegionManagerService>(),
        c.Resolve<IEventAggregator>()));
}
```

The SomeService module configurator class similarly registers the SomeService class with the D.I. container and also gets a reference to the IEventAggregator.

## Defining Event Classes

Next you must define your event classes, and these need to be defined in the infrastructure class library that you use to define cross-module objects such as common business objects and your event objects.

Event classes inherit from `CompositePresentationEvent<T>` (in the `Microsoft.Practices.Composite.Presentation.Events` namespace) where 'T' is the type of the `EventArgs` object. For example, to define a simple event that just passes an `EventArgs` object, use:

```
public class SimpleEvent : CompositePresentationEvent<EventArgs>
{
}
```

If you want to pass some data in the event arguments, define a custom class to hold the data, and then define the event referencing that type. For example:

```
public class DataForSomeServiceEvent
{
    public int SomeData { get; set; }
}

public class SomeServiceEvent :
    CompositePresentationEvent<DataForSomeServiceEvent>
{
}
```

## Publishing and Subscribing Events

Once the event types are defined, using the services of the `EventAggregator` is simple. Any class that wants to publish or subscribe to an event first calls the `T`

`IEventAggregator.GetEvent<T>()` method to get a reference to the event, where 'T' is the type of the event. Then to publish the event, simply call the **Publish** method of the event, passing the appropriate event args object for example:

```
// Publish the SimpleEvent
SimpleEvent customEvent =
    eventAggregator.GetEvent<SimpleEvent>();
customEvent.Publish(new EventArgs());
```

To publish an event and pass some data, use:

```
// Publish an event
SomeServiceEvent customEvent =
    this.eventAggregator.GetEvent<SomeServiceEvent>();
```

```

customEvent.Publish(
    new DataForSomeServiceEvent()
    {
        SomeData =
            Convert.ToInt32(DateTime.Now.TimeOfDay.TotalSeconds)
    }
);

```

To subscribe to an event, get a reference to the event from the `IEventAggregator`, and then call the `Subscribe` method, specifying the callback method to run when the event notification is received:

```

{
    ...

    // Subscribe to the SomeServiceEvent
    SomeServiceEvent customEvent =
        eventAggregator.GetEvent<SomeServiceEvent>();
    customEvent.Subscribe(SomeEventEventHandler, false);
}

// Event Handler for the event published somewhere else in the
// system
private void SomeEventEventHandler(DataForSomeServiceEvent data)
{
    // Do something with data.SomeData...
}

```

## Performance, Thread Usage and Filtering

The boolean parameter to **Subscribe** is the **keepSubscriberReferenceAlive** parameter which determines whether a strong or weak reference is maintained to the event:

- **false** – a weak delegate reference is maintained, which means that the subscriber class can still be garbage collected if it goes out of scope but still has a reference to the event
- **true** – a strong delegate reference is maintained. In this case, you must call the `Unsubscribe(Action<T> action)` method of the event to manually unsubscribe from the event before the subscriber class can be garbage collected.

```

customEvent.Subscribe(SomeEventEventHandler, false);

```

If you are raising multiple events in a short period of time and have noticed performance concerns with them, you may need to subscribe with strong delegate references—and therefore manually unsubscribe from the event when disposing the subscriber—instead of the default weak delegate references maintained by `CompositePresentationEvent`.

One feature of this system is that the EventAggregator registers the event in its internal catalog the first time it is referenced, so subscribers can subscribe to a particular event before a publisher module has requested a reference to that event. There can also be multiple publishers of the same event at the same time.

Use the **ThreadOption** parameter to specify on which thread to receive the event; the choices are as follows:

- Background – execute the event handler on a background thread
- Publisher – execute the event handler on whichever thread the publisher is executing (the default)
- Subscriber – execute on the same thread used to subscribe. This can be useful if you subscribe to an event on the UI thread and want to ensure that the event handler also runs on the UI thread.

```
customEvent.Subscribe(  
    SomeEventHandler,  
    ThreadOption.Subscriber,  
    false);
```

You can also specify a filter function to use if your subscriber class receives a large number of notifications for an event, but is only interested in acting on a number of them that fulfil some criteria. The filter function simply tests the incoming data and returns true if the event handler should be actioned. The filter function is often expressed using a lambda function, as shown in the following example:

```
FundAddedEvent fundAddedEvent =  
    eventAggregator.GetEvent<FundAddedEvent>();  
fundAddedEvent.Subscribe(  
    FundAddedEventHandler,  
    ThreadOption.Background,  
    false,  
    fundOrder => fundOrder.CustomerId == _customerId  
);
```



## Summary

The Event Aggregator mobile application block provides an easy to use mechanism for publishing and subscribing to events. In use, you include in your application an object that implements the IEventAggregator interface that provides event publish and subscribe services to modules in your application; the EventAggregator class supplied in the blocks will suffice for most mobile applications.

The advantages of using the Event Aggregator rather than publishing a 'traditional' .NET event, is that the publisher and subscriber modules remain loosely coupled and do not need to have a reference to each other, only to the common module that implements the event classes and event args classes. This loose coupling promotes effective reuse and unit testing.