Sound Code

a place for me to talk about programming and .NET ... because my friends and family aren't interested

THURSDAY, 19 JUNE 2014

Why Use an Event Aggregator?

All programs need to react to events. When X happens, do Y. Even the most trivial “Hello, World” application prints its output in response to the “program was started” event. And as our programs gain features, the number of events they need to respond to (such as button clicks, or incoming network messages) grows, meaning the strategy we use for handling events will have a big impact on the overall maintainability of our codebase.

In this post I want to compare four different ways you can respond to a simple event. For our example, the event is a “Create User” button being clicked in a Windows Forms application. And in response to that event our application needs to do the following things:

1. Ensure that the entered user data is valid
2. Save the new user into the database
3. Send the user a welcome email
4. Update the GUI to add the new user to a ListBox of users

Approach 1 - Imperative Code

This first approach is in one sense the simplest. When the event happens, just do whatever is needed. So in this example, right in the button click handler, we’d construct whatever objects we needed to perform those four tasks:

```
private void buttonCreateUser_Click(object sender, EventArgs e)
{
    // get user
    var user = new User()
               {
                   Name = textBoxUserName.Text,
                   Password = textBoxPassword.Text,
                   Email = textBoxEmail.Text
               };
    // validate user
    if (string.IsNullOrEmpty(user.Name) ||
        string.IsNullOrEmpty(user.Password) ||
        string.IsNullOrEmpty(user.Email))
    {
        MessageBox.Show("Invalid User");
        return;
    }

    // save user to database
    using (var db = new SqlConnection(@"Server=(localdb)\v11.0;
    {
        db.Open();
        using (var cmd = db.CreateCommand())
```

Keep me updated

email address

Subscribe

ABOUT ME

Mark Heath
I'm a software developer, based in Southampton, UK specialising in .NET, trying to keep up with best practices and latest technologies. In my spare time, I maintain a variety of open source .NET applications, and this blog is for sharing things I discover along the way. Read more....

MY PLURALSIGHT COURSES

- Creating Modern WPF Applications with MahApps.Metro
- Windows Forms Best Practices
- Understanding and Eliminating Technical Debt
- Digital Audio Fundamentals
- Audio Programming with NAudio
- Understanding Distributed Version Control Systems
- ClickOnce Deployment Fundamentals

MY SOFTWARE

- NAudio .NET Audio Library
- Skype Voice Changer
- .NET Voice Recorder
- TypeScript Tetris
- Silverlight Audio Player
- NLayer Managed MP3 Decoder
- SilverNibbles Silverlight Snake Game
- MIDI File Mapper
- MIDI File Splitter
- Asterisk.NET retro game

```
        {
            cmd.CommandText = "INSERT INTO Users (UserName, Pas
            cmd.Parameters.Add("UserName", SqlDbType.VarChar).V
            cmd.Parameters.Add("Password", SqlDbType.VarChar).V
            cmd.Parameters.Add("Email", SqlDbType.VarChar).Valu
            cmd.ExecuteNonQuery();
        }

        // get the identity of the new user
        using (var cmd = db.CreateCommand())
        {
            cmd.CommandText = "SELECT @@IDENTITY";
            var identity =  cmd.ExecuteScalar();
            user.Id = Convert.ToInt32(identity);
        }
    }

    // send welcome email
    try
    {
        var fromAddress = new MailAddress(AppSettings.EmailSend
        var toAddress = new MailAddress(user.Email, user.Name);
        const string subject = "Welcome";
        const string body = "Congratulations, your account is a

        var smtp = new SmtpClient
        {
            Host = AppSettings.SmtpHost,
            Port = AppSettings.SmtpPort,
            EnableSsl = true,
            DeliveryMethod = SmtpDeliveryMethod.Network,
            UseDefaultCredentials = false,
            Credentials = new NetworkCredential(fromAddress.Add
        };
        using (var message = new MailMessage(fromAddress, toAdd
        {
            Subject = subject,
            Body = body
        })
        {
            smtp.Send(message);
        }
    }
    catch (Exception emailException)
    {
        MessageBox.Show(String.Format("Failed to send email {0}
    }

    // update gui
    listBoxUsers.Items.Add(user.Name);
}
```

What’s wrong with this code? Well multiple things. Most notably it violates the Single Responsibility Principle. Here inside our GUI we have code to talk to the database, code to send emails, and code that knows about our business rules. This means it’s going to be almost **impossible to unit test** in its current form. It also means that we’ll likely end up with **cut and paste code** if later we discover that another part of our system needs to create new users, because that code will need to perform the same sequence of actions.

The reason we’re in this mess is that we have **tightly coupled** the code that publishes the event (in this case the GUI), to the code that handles that event.

- Learning Games

SUBSCRIBE TO

Posts

Comments

SEARCH THIS BLOG

Search

BLOG ARCHIVE

- ► 2016 (2)
- ► 2015 (15)
- ▼ 2014 (50)
    - ► December (2)
    - ► November (3)
    - ► October (4)
    - ► September (2)
    - ► August (1)
    - ► July (4)
    - ▼ June (8)
        - How to create zip backups of previous git commits
        - How to Create a Breaking Non-Space in HTML
        - How to Zip a Folder with ASP.NET Web Pages
        - Why Use an Event Aggregator?
        - New Blog-markheath.net
        - Running Windows Forms on Linux with Mono
        - Is Windows Forms Dead Yet?
        - My Headphone Recommendations
    - ► May (6)
    - ► April (6)
    - ► March (5)
    - ► February (4)
    - ► January (5)
- ► 2013 (19)
- ► 2012 (30)
- ► 2011 (35)
- ► 2010 (29)
- ► 2009 (26)
- ► 2008 (64)
- ► 2007 (27)

TAGS

NAudio (63) XAML (37)
WPF (35) Silverlight (33)
audio (32) Mercurial (14)

So what can we do to fix this? Well, if you know the “SOLID” principles, you know its always a good idea to introduce some “Dependency Injection”. So let’s do that next…

Approach 2 – Dependency Injection

What we could do here is create a few classes each with a single responsibility. A UserValidator validates the entered data, a UserRepository saves users to the database, and an EmailService sends the welcome email. And we give each one an interface, allowing us to mock them for our unit tests.

Suddenly, our create user button click event handler has got a whole lot more simple:

```
public NewUserForm(IUserValidator userValidator, IUserRepositor
{
    this.userValidator = userValidator;
    this.userRepository = userRepository;
    this.emailService = emailService;
    InitializeComponent();
}

private void buttonCreateUser_Click(object sender, EventArgs e)
{
    // get user
    var user = new User()
               {
                   Name = textBoxUserName.Text,
                   Password = textBoxPassword.Text,
                   Email = textBoxEmail.Text
               };
    // validate user
    if (!userValidator.Validate(user)) return;

    // save user to database
    userRepository.AddUser(user);

    // send welcome email
    const string subject = "Welcome";
    const string body = "Congratulations, your account is all s
    emailService.Email(user, subject, body);

    // update gui
    listBoxUsers.Items.Add(user.Name);
}
```

So we can see we’ve improved things a lot, but there are still some issues with this style of code. First of all, we’ve probably not taken DI far enough. Our GUI still has quite a lot of knowledge about the workflow of handling this event – we need to validate, then save, then send email. And so probably we’d find ourselves wanting to create another class just to orchestrate these three steps.

Another related problem is the construction of objects. Inversion of Control containers can help you out quite a bit here, but it’s not uncommon to find yourself having classes with dependencies on dozens of interfaces, just because you need to pass them on to child objects that get created later. Even in this simple example we’ve got three dependencies in our constructor.

But does the GUI really need to know anything whatsoever about how this event should be handled? What if it simply publishes a CreateUserRequest event and lets someone else handle it?

IronPython (13) C# (11) DVCS (11) HOWTO (11) ASP.NET (10) MVVM (10) unit testing (10) Python (9) Software Development (9) TDD (9) LINQ (8) git (8) Refactoring (7) Single Responsibility Principle (7) TFS (7) Technical Debt (7) Windows 8 (7) Windows Forms (7) IoC (6) Pluralsight (6) Developer Principles (5) MSBuild (5) NUnit (5) Visual Studio (5) ASP.NET MVC (4) Data Binding (4) Dependency Inversion Principle (4) MP3 (4) NuGet (4) Open Closed Principle (4) Unity (4) clean code (4) screencast (4) Code Reviews (3) F# (3) HTML (3) JavaScript (3) Live Mesh (3) Media Foundation (3) Threading (3) VistaDB (3) Windows Vista (3) architecture (3) Azure (2) ClickOnce (2) Liskov Substitution Principle (2) Model View Presenter (2) PowerShell (2) Reactive Extensions (2) SQL (2) Version Control (2) event aggregator (2) interop (2) AIFF (1) Excel (1) Interface Segregation Principle (1) JSON (1) LINQPad (1) MEF (1) Observer Pattern (1) Razor (1) SOLID (1) Skype Voice Changer (1) SubSonic (1) Subversion (1) WASAPI (1) WCF (1) WebMatrix (1) book review (1) branching (1) code signing (1) coding dojo (1) complexity (1) css (1) debugging (1) headphones (1) kata (1) knockout (1) merging (1) microservices (1) mono (1) nupack (1) templating engine (1) typescript (1)

Approach 3 – Raising Events

So the third approach is to take advantage of .NET’s built-in events, and simply pass on the message that the CreateUser button has been clicked:

```
public event EventHandler<UserEventArgs> NewUserRequested;

protected virtual void OnNewUserRequested(UserEventArgs e)
{
    var handler = NewUserRequested;
    if (handler != null) handler(this, e);
}

private void buttonCreateUser_Click(object sender, EventArgs e)
{
    var user = new User()
               {
                   Name = textBoxUserName.Text,
                   Password = textBoxPassword.Text,
                   Email = textBoxEmail.Text
               };
    // send an event
    OnNewUserRequested(new UserEventArgs(user));
}

public void OnNewUserCreated(User newUser)
{
    // now a user has been created, we can update the GUI
    listBoxUsers.Items.Add(newUser.Name);
}
```

What’s going on here is that the button click handler now does nothing except gather up what was entered on the screen and raise an event. It’s now completely up to whoever subscribes to that event to deal with it (performing our tasks of Validate, Save to Database and Send Email), and then they need to call us back on our “OnNewUserCreated” method so we can update the GUI.

This approach is very nice in terms of simplifying the GUI code. But one issue that you can run into is similar to the one faced with the dependency injection approach. You can easily find yourself handling an event only to pass it on to the thing that really needs to handle it. I’ve seen applications where an event is passed up through 7 or 8 nested GUI controls before it reaches the class that knows how to handle it. Can we avoid this? Enter the event aggregator…

Approach 4 – The Event Aggregator

The event aggregator **completely decouples the code that raises the event from the code that handles it**. The event publisher doesn’t know or care who is interested, or how many subscribers there are. And the event subscriber doesn’t need to know who is responsible for publishing it. All that is needed is that both the publisher and subscriber can talk to the event aggregator. And it may be acceptable to you to use a Singleton in this case, although you can inject it if you prefer.

So in our next code sample, we see that the GUI component now just publishes one event to the event aggregator (a NewUserRequested event) when the button is clicked, and subscribes to the NewUserCreated event in order to perform its GUI update. It needs no knowledge of who is listening to NewUserRequested or who is publishing NewUserCreated.

```
public CreateUserForm()
{
    InitializeComponent();
    EventPublisher.Instance.Subscribe<NewUserCreated>
```

```
        (n => listBoxUsers.Items.Add(n.User.Name));
}

private void buttonCreateUser_Click(object sender, EventArgs e)
{
    // get user
    var user = new User()
               {
                   Name = textBoxUserName.Text,
                   Password = textBoxPassword.Text,
                   Email = textBoxEmail.Text
               };
    EventPublisher.Instance.Publish(new NewUserRequested(user))
}
```

As you can see, this approach leaves us with trivially simple code in our GUI class. The subscribers too are simplified since they don’t need to be wired up directly to the class publishing the event.

Benefits of the Event Aggregator Approach

There are many benefits to this approach beyond the **decoupling of publishers from subscribers**. It is **conceptually very simple** and easy for developers to get up to speed on. You can introduce new types of messages easily without making changes to public interfaces. It’s very **unit testing friendly**. It also discourages chatty interactions with dependencies and **encourages a more asynchronous way of working** - send a message with enough information for the handlers to deal with it, and then wait for the response, which is simply another message on the event bus.

There are several upgrades to a vanilla event aggregator that you can create to make it even more powerful. For example, you can give subscribers the capability to specify what thread they want to handle a message on (e.g. GUI thread or background thread). Or you can use WeakReferences to reduce memory leaks when subscribers forget to unsubscribe. Or you can put global exception handling around each callback to a subscriber so you can guarantee that when you publish a message every subscriber will get a chance to handle it, and the publisher will be able to continue.

There are many situations in which the event aggregator really shines. For example, imagine you need an audit trail tracking many different events in your application. You can create a single Auditor class that simply needs to subscribe to all the messages of interest that come through the event aggregator. This helps keep cross-cutting concerns in one place.

Another great example is a single event that can be fired from multiple different places in the code, such as when the user requests help within a GUI application. We simply need to publish a HelpRequested message to the aggregator with contextual information of what screen they were on, and a single subscriber can ensure that the correct help page is launched.

Where Can I Get An Event Aggregator?

Curiously, despite the extreme usefulness of this pattern, there doesn’t seem to be an event aggregator implementation that has emerged as a “winner” in the .NET community. Perhaps this is because it is so easy to write your own. And perhaps also because it really depends what you are using it for as to what extensions and upgrades you want to apply. Here’s a few to look at to get you started:

- Udi Dahan’s Domain Event pattern
    - Uses an IDomainEvent marker interface on all messages
    - Integrates with the container to find all handlers of a given event
    - Has a slightly odd approach to threading (pubs and subs always on the same thread)

- José Romaniello’s Event Aggregator using Reactive Extensions
  - A very elegant and succinct implementation using the power of Rx
  - Subscriptions are Disposable
  - Use the power of Rx to filter out just events you are interested in, and handle on whatever thread you want
- Laurent Bugnion’s Messenger (part of MVVM Light)
  - Uses weak references to prevent memory leaks
  - Can pass “tokens” for context
  - Includes passing in object as the subscriber, allowing you to unsubscribe from many events at once
  - Allows you to subscribe to a base type and get messages of derived types
- PRISM Event Aggregator (from Microsoft patterns and practices)
  - A slightly different approach to the interface - events inherit from EventBase, which has Publish, Subscribe and Unsubscribe methods.
  - Supports event filtering, subscribing on UI thread, and optional strong references (weak by default)

I’ve made a few myself which I may share at some point too. Let me know in the comments what event aggregator you’re using.

Posted by Mark Heath at 16:24

Labels: event aggregator

G+

1 comment

Add a comment as Shibu Thomas Varughese

Top comments

Alex Salnikov 2 years ago - Shared publicly

Hello Mark, i really liked your Windows Forms Best Practices course,
i love your simple Event Aggregator and also Laurent Bugnion’s Messenger. I understand how EA make better decoupled code but my concern is what under performance compared to .Net Event this EA have.

Read more

+1 1 · Reply

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)