Temaran messenger

A better messaging mechanism for Unity3D.

Created by Fredrik Lindh (Temaran)
Questions to: temaran (at) gmail (dot) com

About Messengers / event aggregators in general:

From http://joshearl.wordpress.com/2011/03/31/introduction-to-messaging-with-caliburn-micros-eventaggregator/

… A message is an object that is broadcast throughout your application when something important happens. When using messaging, an object "publishes" a message, which can be just about anything you want it to be. This object bubbles through your application, moving up the object hierarchy. Other objects that are subscribed to the message will receive it and can take action upon receiving it.

One of the major advantages to messaging is the way it can help you decouple classes. Because the message bubbles through the system, your subscribing code doesn't need to know about the sender—it just listens for the message.

Traditional event handling, on the other hand, tightly couples classes together…

In the case of unity3d, the standard framework supports a monobehaviour method called BroadcastMessage, which can be thought of as a simple messenger mechanism. The problems with this method are many however; type safety (it has none as it uses strings), performance (it uses immediate reflection, which is excruciatingly slow) and direct coupling (if you want to do this on other objects than yourself, you need to get a direct reference) just to name a few. There are many more.

Now, there are a couple of other messengers available, both on the asset store and on the wiki (http://www.unifycommunity.com/wiki/index.php?title=CSharpMessenger) but these implementations share many of the problems of BroadcastMessage (type safety for one) and the ones on the asset store even cost you money!

By this point I think that most of you reading this agree with me that this is something that is ***CORE*** when making a game. From small scale indie projects, to major AAA titles, having an object communication system that doesn't need constant attention when refactoring, allows you to produce code quickly and is still high-performant is a system that needs to be as close to perfect as possible, which is why I decided to make an alternative that solves as many problems as possible.

Introduction:

One of the biggest problems I've had with Unity3D so far is the annoying lack of a decent messenger / event aggregator.

In addition, all that are available for sale at the moment do not support type safe programming. They are also unnecessarily clunky to set up for each class and hard to extend. This usually leads to code being hard to develop and refactor.

My solution attempts to solve all of these problems, while still being blazing fast. Hopefully this will be useful for other people too, as it has been for me.

You can send global messages like this, everyone who registered for the message in the current scene will get it.
There are many reasons why you would want to use this over the other options you have available:

1. It decouples your sending code from your receiving code and encapsulation of the client code itself. For example; A weapon script does not have to know that it needs to look for a health script, and much more, if you add more types of receivers (for example, different types of health scripts) you do not have to remember to modify the weapon script to look for these new types. Everything is done client side (the new health script implements a handle for the damage-message, and you're done). Refactoring communication code has never been this easy. It also promotes re-usability, since your components only need to know their input / output messages, and hold very few direct references. This means it is very easy to move a complete component between projects.
2. Easy to test since you can mock simply by sending dummy messages.
3. It is MUCH faster, since almost everything here is cached, and it is only ever bothering objects that are interested.
4. Thread safety (although you're free to remove this if it doesn't suit you)

It also offers these additional advantages (which it seems that other messengers on the asset store does not offer, at least not from what I have seen):

1. It is type safe, when you refactor code with unity's standard interface, and you miss to change one method name string, you might not notice it for some time. Since we're using real types here, the compiler will throw complaints right away.
2. It's easier, only one line of code to publish, and you don't have to explicitly remember to subscribe and unsubscribe for clients
3. It's faster than all implementations I've tested. I've only included the tests for free messengers.
4. It is more useful, since you can also use it for much more than just sending plain messages (see for example the tagging example or the request example)
5. It's FREE.

How to use:

To illustrate the easy of use, below is all code needed to get a sample running :)

Publisher:

```
public class GlobalExamplePublisher : MonoBehaviourEx
{
    public void Update()
    {
        Messenger.Publish(new ExampleMessage("OMG THIS JUST HAPPENED", renderer));
    }
}
```

Subscriber:

```
public class ExampleSubscriber : MonoBehaviourEx, IHandle<ExampleMessage>
{
    public void Handle(ExampleMessage message)
    {
        //Do something here
    }
}
```

For more advanced examples, please refer to the unity project.

The idea behind the code is very simple. Firstly, to make it easy to use, any MonoBehaviours in the client project should derive from **MonoBehaviourEx** instead, or move the code in that class to the base-class that you happen to be using. This is to provide easy access to the Messenger. You could of course locate it in a static class somewhere instead, and do subscription setup in your start method, but that get's tedious very fast in my opinion. It's up to you however :)

Next, you should design your message. A good message is a single specific instruction that is very clear on what it is supposed to do. For example, ”DoStuff” is not a good message, while ”InventoryUpdated” could be a good notification message. It is good to realize there are many different types of use cases, for example, notification messages like the ”InventoryUpdatedNotification” example tell other objects that something happened in your object. Command messages such as ”TakeDamageCommand” is appropriate for telling other objects to do something. Request messages such as ”GetPlayerRequest” can be used to find certain groups of objects. The messenger in itself does not enforce any types like this, and does not handle them differently as a consequence, but I encourage everyone to impose this to their projects as it makes it significantly easier to identify what a message is supposed to do when coming back to code written long ago.

When you want to send a message, simply do Messenger.Publish(new MyMessage(MyParam1, MyParam2, ...)); and the instance of the message you created will be sent to all subscribers. Subscribers themselves listen to messages by simply implementing the IHandle<T> interface (provided they derive from MonoBehaviourEx, if they don't you have to subscribe to a messenger instance in your awake method as well). An example could be that if you have an example message ”MyCoolMessage” and you want a script to accept it, your class definition should look something like this:

```
public class MySubscriberScript : MonoBehaviourEx, IHandle<MyCoolMessage>
```

To react to the message, simply put your processing code in the implementation of the interface method. For completeness, this method would look like this in our example class:

```
public void Handle(MyCoolMessage message)
{
    //Do something here
}
```

About the project:

I took a lot of inspiration from WPF frameworks such as Caliburn micro and MVVM light when designing this messenger for Unity3d. Especially the IHandle<T> pattern from Caliburn is lifted almost as-is. I also have to give thanks to Jon Skeet as I took a lot of inspiration from his blog when designing the delegate caching mechanism of the messenger. Most of the actual work was spent speeding it up and making sure it would be easy to use in Unity since it provides some additional dimensions (transform hierarchies etc). All in all, there are some things I would still like to do, such as making request messengers more pretty and maybe adding a “high-speed” mode where I give support for caching message channels in subscribers (this would remove the dictionary lookup and its overhead, providing a speed boost for those scenarios.

The project is hosted at https://github.com/Temaran/TemaranUnityMessenger
Please check it out, and feel free to work on it if you would like :)