https://supermax.github.io/pubsub

SuperMaxim.com

Pub\Sub Messenger for Unity

The Problem:

- **Wiring the Parts with Events** – Tightly Coupled and may cause Memory Leak problems. The Publisher and the Subscriber have to know of each other, and a Subscriber can't be collected by the GC if it's connected with the Publisher with strong event reference.

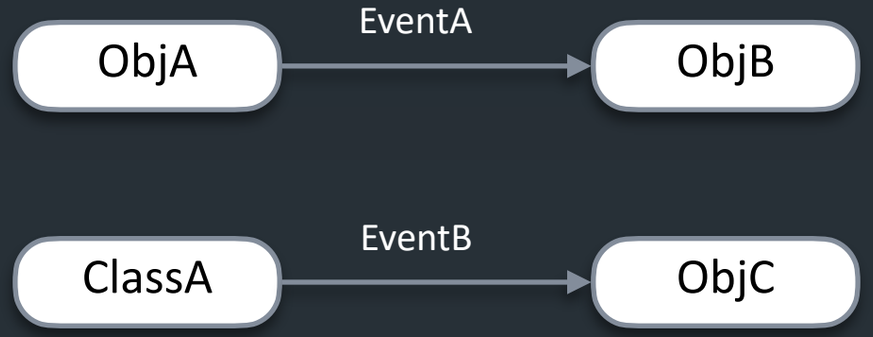
- **Using Unity Event Routing** – Although Unity Event Routing is a very good feature, it is a Unity Specific Solution and we need a generic one. Also, we cannot use it everywhere even if the project is in Unity.

- Bad OPP and Less Encapsulated Objects - In commonly used C# events or delegates classes “familiar” with each other and this prevents good modularity.

The Problem:

Example of common usages of events in C#:

```
public class ClassA
{
    // event that passes instance of ClassA when it is invoked
    public event Action<ClassA> EventA;

    // static event that passes instance of ClassA when it is invoked
    public static event Action<ClassA> EventB;
}


public class ClassB
{
    // event handler - method that is invoked by event and receives objA instance
    public void OnEventA(ClassA objA) { }
}

public class ClassC
{
    private ClassA _objA;
    private ClassB _objB;
    private ClassB _objC;

    public void RegisterEvents()
    {
        // _objB.OnEventA is attached to _objA.EventA and _objB is now referenced by _objA
        // _objB will not be removed from memory until it's handler OnEventA will be detached from EventA
        // or until objA will be removed from memory
        _objA.EventA += _objB.OnEventA;

        // _objC.OnEventA is attached to ClassA.EventB and _objC is not referenced by static pointer in ClassA
        // _objC will not be removed from memory until it's handler OnEventA will be detached from static EventB
        // static references are worse case of memory leaks
        ClassA.EventB += _objC.OnEventA;
    }
}
```



The Solution:

- ✓ **Custom Pub\Sub Messenger -** Container for Events that allows Decoupling of Publishers and Subscribers so they can evolve independently. This Decoupling is useful in Modularized Applications because new modules can be added that respond to events defined by the Shell or, more likely, other modules. All events have a Weak Reference and invocation can be done Async or Sync way.

- ✓ Instead of passing objects or modules, pass small Payloads (Data/ Messages) that are relevant for the specific cases/events.

- ✓ Classes/Modules will not be “familiar” with each other = better encapsulation and less dependencies.

- ✓ In case of subscriber’s destruction, it will be removed automatically from Messenger’s list, since it is via Weak Reference.

- ✓ Pub/Sub can be great pattern in combination with Dependency Injection and with Inversion of Control (all three belong to SOLID).

The Solution:

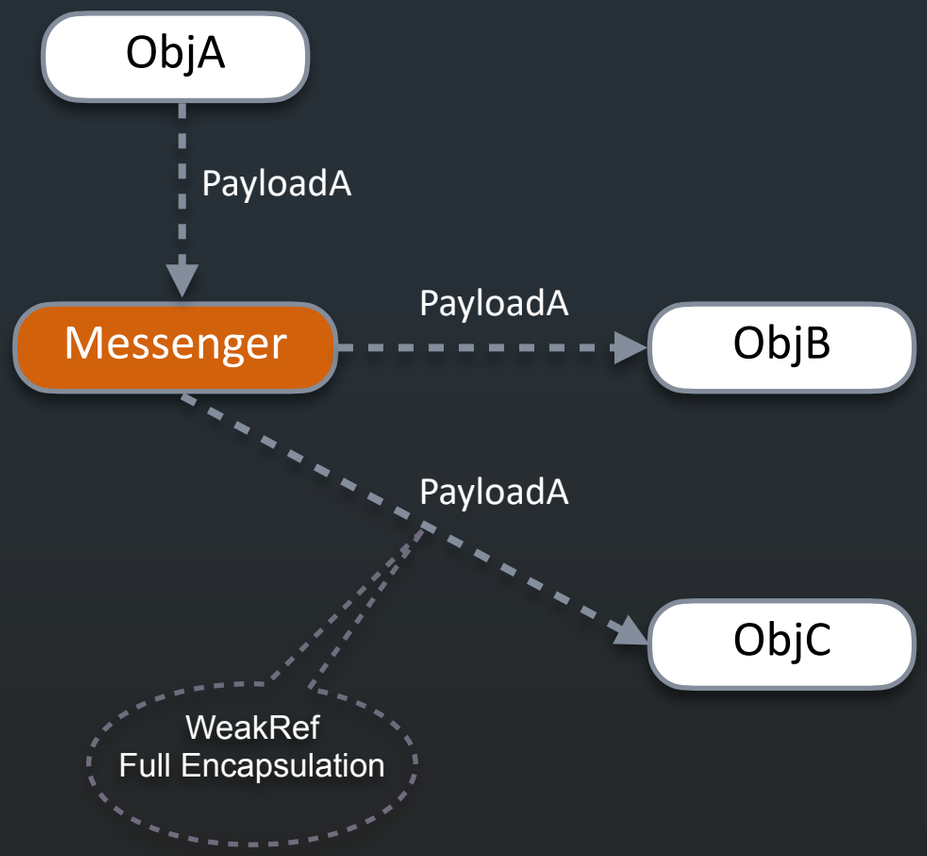
Usage of Messenger as Pub/Sub mechanism:

```
public class ClassA
{
   public void PublishEvent()
    {
        // publish new payload with specific data
        Messenger.Default.Publish(new PayloadA { Id = 123, Name = “ABC” });
    }
}
```

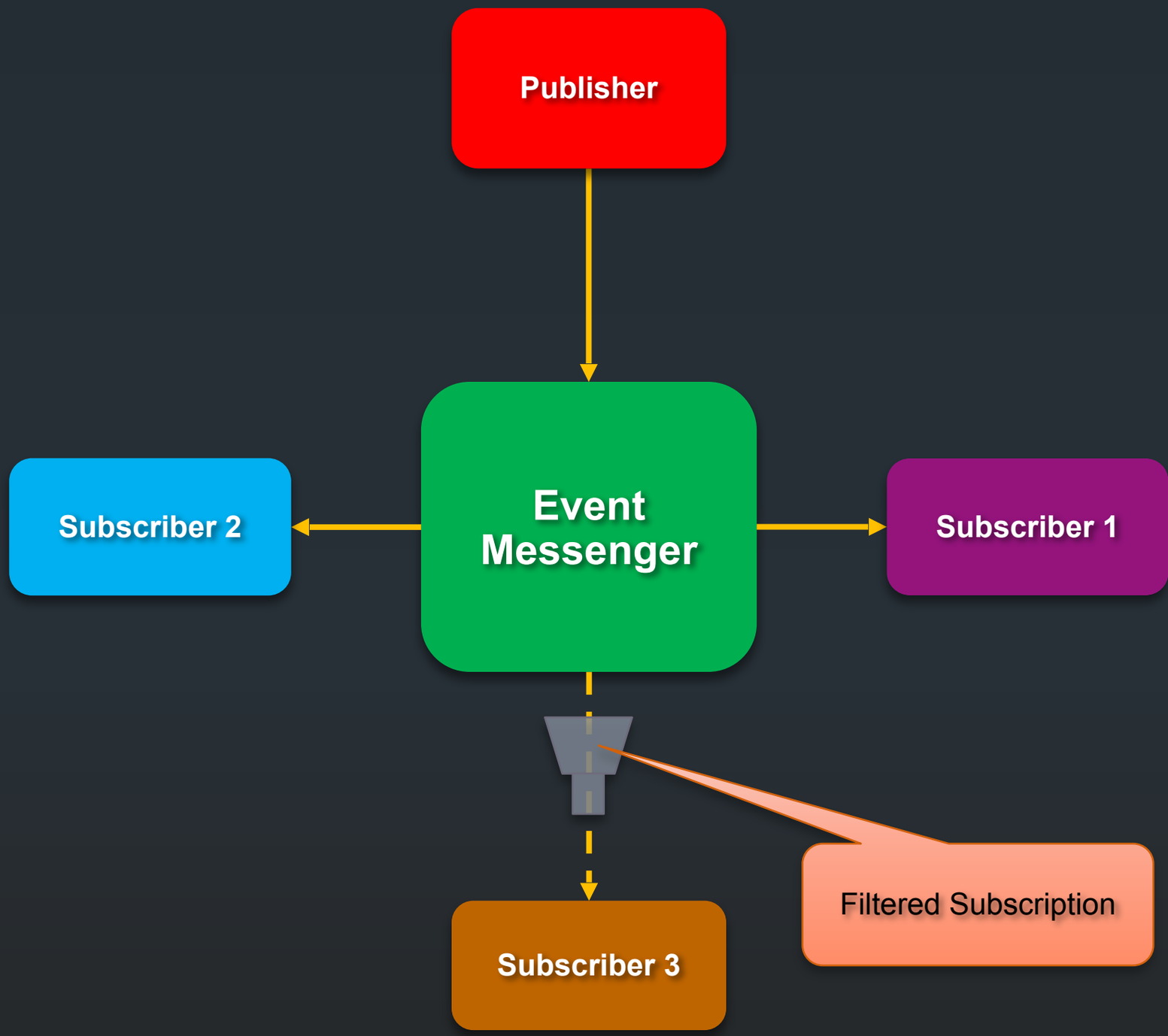
```
public class PayloadA
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

```
public class ClassB
{
    // event handler - method that is invoked by Messenger and receives payload instance
    public void OnEventA(PayloadA payload) { }
}
```

```
public class ClassC
{
    private ClassA _objA;
    private ClassB _objB;
    private ClassB _objC;

    public void RegisterEvents()
    {
        // subscribe handler _objB.OnEventA to PayloadA
        Messenger.Default.Subscribe<PayloadA>(_objB.OnEventA);

        // subscribe handler _objC.OnEventA to PayloadA with filter
        Messenger.Default.Subscribe<PayloadA>(_objC.OnEventA, PayloadAFilter);
    }

    private bool PayloadAFilter(PayloadA payload) { /*TODO*/ }
}
```



Event Routing by Pub\Sub Messenger



Use Cases for Pub\Sub Messenger:

- ✓**Pass Payload** between disconnected/independent parts of code;
- ✓**Thread Safe** Invocation of Events between disconnected/independent parts of code;
- ✓**Asynchronous** Invocation of Events between disconnected/independent parts of code;
- ✓**Filtered** Invocation of Events between disconnected/independent parts of code;
- ✓**Obfuscated** Invocation of Events between disconnected/independent parts of code;

Messenger API:

- ✓ **Messenger**

  Implements this interface:

```
/// <summary>
/// Interface for Pub/Sub Messenger
/// </summary>
public interface IMessenger
{
  /// <summary>
  /// Publish given payload to relevant subscribers
  /// </summary>
  /// <param name="payload">Instance of payload to publish</param>
  /// <typeparam name="T">The type of payload to publish</typeparam>
  void Publish<T>(T payload);

  /// <summary>
  /// Subscribe given callback to receive payload
  /// </summary>
  /// <param name="callback">The callback that will receive the payload</param>
  /// <param name="predicate">The predicate to filter irrelevant payloads (optional)</param>
  /// <typeparam name="T">The type of payload to receive</typeparam>
  void Subscribe<T>(Action<T> callback, Predicate<T> predicate = null);

  /// <summary>
  /// Unsubscribe given callback from receiving payload
  /// </summary>
  /// <param name="callback">The callback that subscribed to receive payload</param>
  /// <typeparam name="T">Type of payload to unsubscribe from</typeparam>
  void Unsubscribe<T>(Action<T> callback);
}
```

Messenger API:

✓ **Messenger**

Access to default Messenger instance via: `Messenger.Default`

✓ **Publish**

```
Messenger.Default.Publish<Payload>(new Payload{ /* payload params */ });
```

```
Payload – the payload that will be published to subscribers of this type
```

Messenger API:

✓ **Subscribe**

```
Messenger.Default.Subscribe<Payload>(Callback);

Payload – the type of Callback parameter
Callback – delegate (Method) that will receive payload

private static void Callback(Payload payload)
{
  // Callback logic
}
```

✓ **Subscribe with Predicate**

```
Messenger.Default.Subscribe<Payload>(Callback, Predicate);

Predicate – delegate (Function) that will receive payload to filter

private static bool Predicate(Payload payload)
{
  // Predicate filter logic
  // if function will return ‘false’ value, the Callback will not be invoked
  return accepted;
}
```

Messenger API:

✓ **Subscribe**

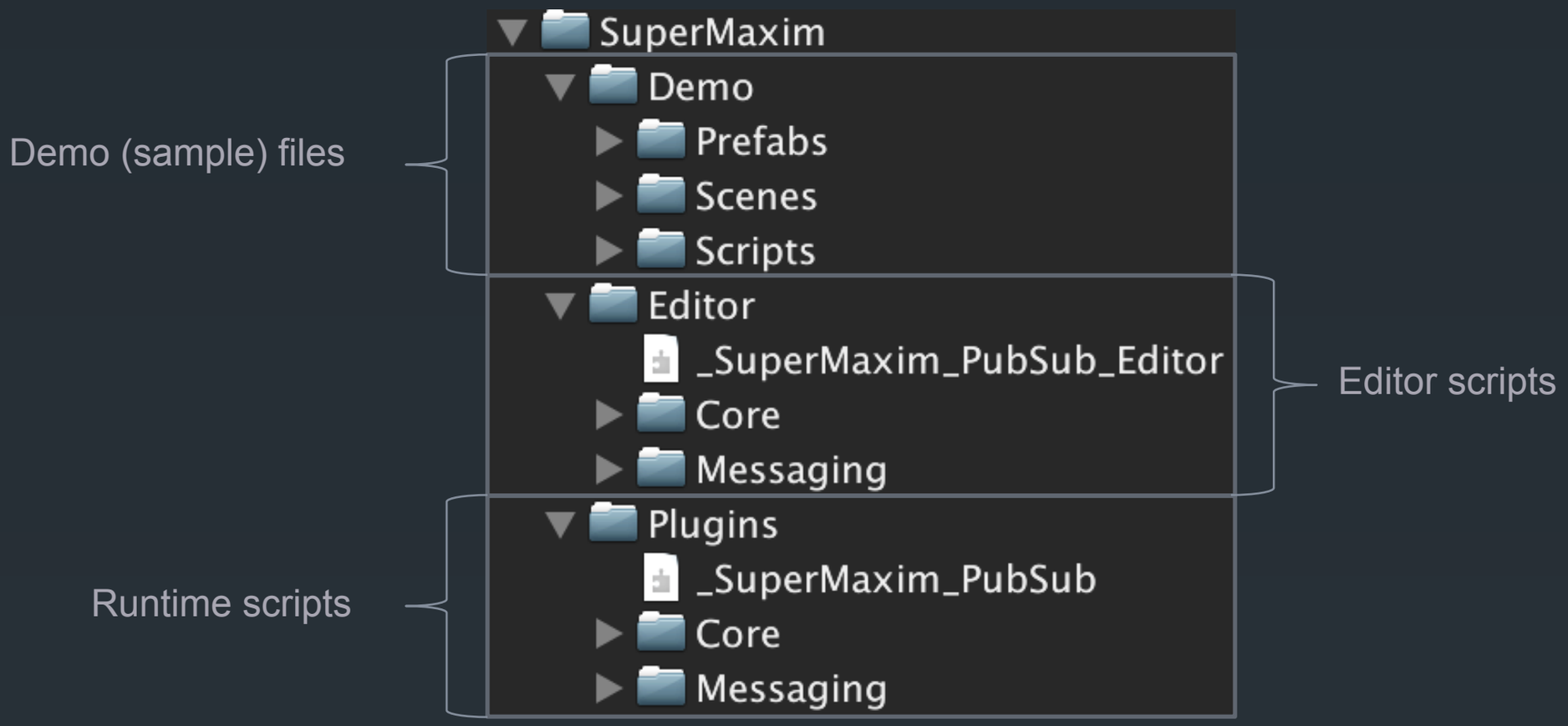
```
Messenger.Default.Unsubscribe<Payload>(Callback);

Payload – the type of Callback parameter
Callback – delegate (Method) that will be removed from subscribers
```

Tips for Correct Usage:

- **DON'T** use Messenger if you have quick access to shared code parts to invoke events/methods in same module/class;
- ✓**ALWAYS** ensure that you unsubscribe when you're done with consuming of shared payloads;
- **DON'T** publish events in endless loop;
- ✓**PREFER** using Filtered subscriptions;

Package Structure:

✓**Folders**

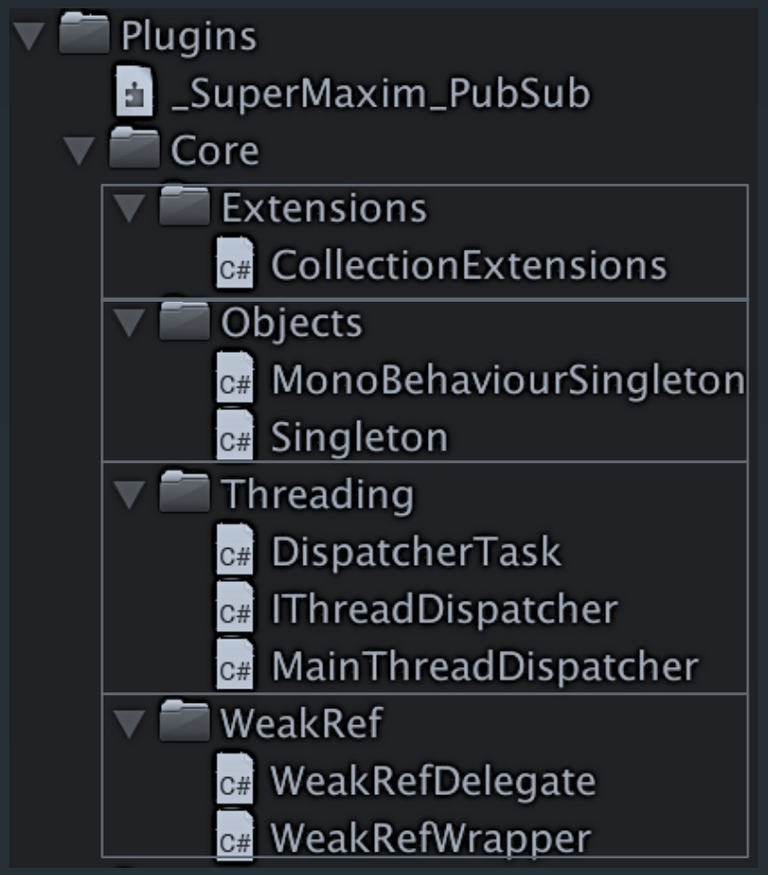


Folders:

Core – base classes and scripts that are not specific for Messenger
Messaging – classes and scripts that are specific for Messenger

Package Structure:

✓Plugins / Core



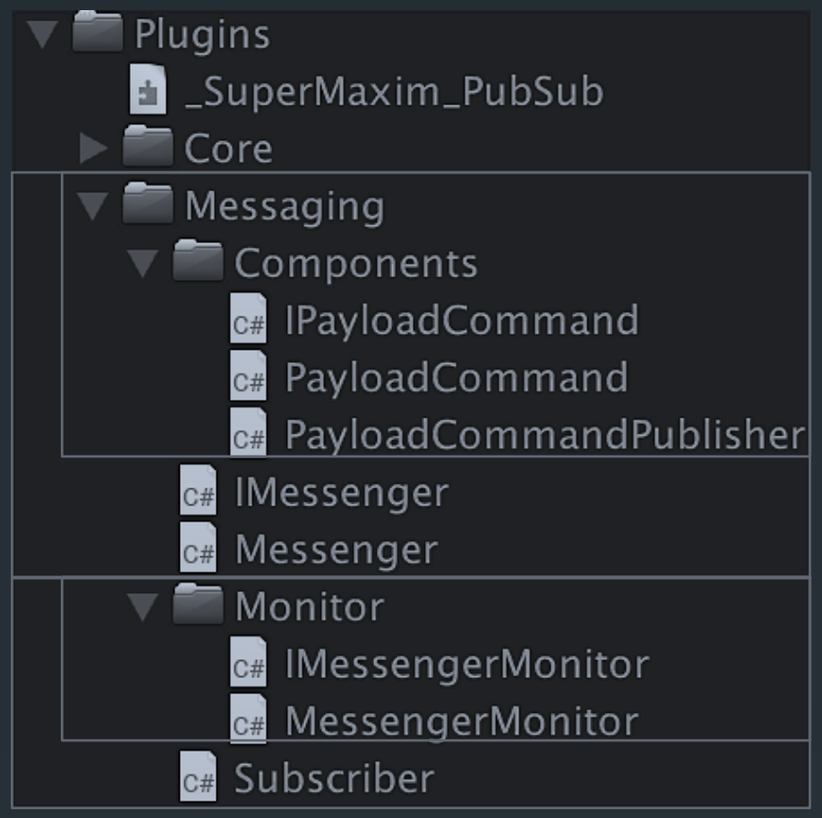
Folder	File/Class/Script	Description
Extensions	CollectionExtensions	Static class with collection extension functions
Objects	MonoBehaviourSingleton	Abstract base class for MB singletons
Objects	Singleton	Abstract base class for classic singletons
Threading	DispatcherTask	Task unit with weak ref. (pointer) to delegate to execute on main thread
Threading	IThreadDispatcher	Interface for Thread Dispatcher API
Threading	MainThreadDispatcher	Singleton class that implements IThreadDispatcher and provides API to sync tasks between main thread and other threads
WeakRef	WeakRefDelegate	Weak reference for delegate (inherits from WeakRefWrapper)
WeakRef	WeakRefWrapper	Weak reference wrapper (disposable)

Folders:

Extensions – extensions classes
Objects – core base classes
Threading – multithreading related classes/scripts
WeakRef – weak reference handling classes

Package Structure:

✓Plugins / Messaging



Folder	File/Class/Script	Description
Messaging / Components	IPayloadCommand	Interface for Payload Command – a generic payload with Id and Dic. of string key value pair Data
Messaging / Components	PayloadCommand	Implementation of generic Payload Command, implementing IPayloadCommand interface
Messaging / Components	PayloadCommandPublisher	MB Script with Publish method that publishes a generic PayloadCommand – can be used with UGUI events or other ways
Messaging	IMessenger	Interface for Messenger API
Messaging	Messenger	Implementation of IMessenger interface with Pub-Sub logic; contains container of subscribers; singleton that is accessible via Messenger.Default
Messaging	Subscriber	Represents single item for subscribers' container. Including weak ref. pointing to callback method and weak ref. pointing to predicate.
Messaging / Monitor	IMessengerMonitor	Interface for MessengerMonitor API
Messaging / Monitor	MessengerMonitor	Debugging and monitoring tool for Messenger

Folders:

Components – useful unity components
Monitor – debugging and monitoring tools

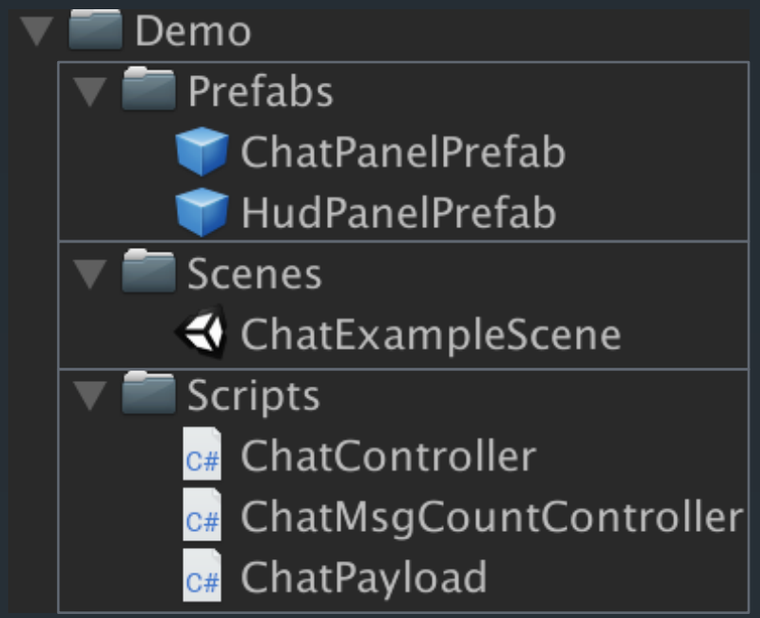
DEMO – Game Chat:

Goals:

- ✓Show basic usage of Messenger;
- ✓Use real world example – Chat between players;
- ✓Include examples of message filtering and multithreading;
- ✓Show “Best Practices” approach in implementation;

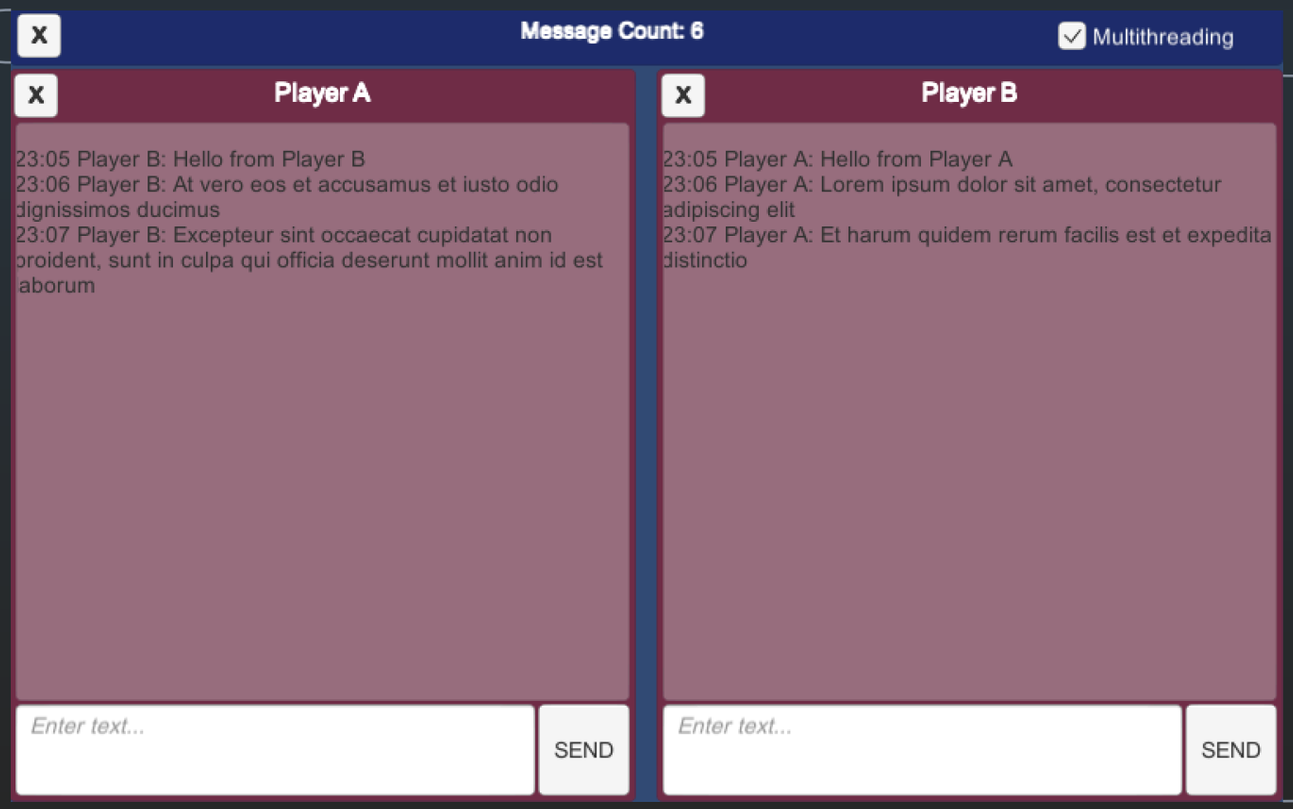
Demo Structure:

✓Demo



Folder	File/Class/Script	Description
Prefabs	ChatPanelPrefab	Prefab that contains all elements for player chat UI (see prefab in editor)
Prefabs	HubPanelPrefab	Prefab that contains top HUD elements (see prefab in editor)
Scenes	ChatExampleScene	Example scene with chat UI (see scene in editor)
Scripts	ChatController	Script that controls ChatPanelPrefab (message input, publish and subscribe)
Scripts	ChatMsgCountController	Scripts that 'listens' to chat messages, counts them and displays count in HUD
Scripts	ChatPayload	Payload class that is used to pass messages between chat panels

HudPanelPrefab



ChatPanelPrefab

Chat Controller:

Implements this basic interface:

```
TBD
```

Unit Tests:

Coverage:

✓…;

✓…;

✓…;

✓…;