# My Messaging Pattern

The goal of this pattern is to work as a bridge between the subscriber and the publisher side of the events patterns.  
By using this bridge then there would be no need to the publisher to know its subscriber and each subscriber to know its publisher creating then a loosely coupled design in the application.

## General Functioning:

This pattern has two sides:

* Globally Accessible Message Container.
* Global “Accessor” Factory: A Message Publisher.

The Factory should be a controller class with access to every other class in the application. Because every single class in the application should have access to the container, then the container cannot have references to those classes.

The Factory would be in charge of registering each of the messages and its allowed publishers, add all that logic setup to the container and then, optionally, forbidding further registering of new messages.

The goal of this pattern is to decouple the event-based communication by acting as a bridge between the subscriber and the publishers. By using this bridge then there would be no need to the publisher to know its subscriber and each subscriber to know its publisher creating then a loosely coupled design in the application.

The Message Container’s users would be able to subscribe to any message by providing, as usual, the message (event) handler.

Any failed attempt of any kind will result in the correspondent method returning the false Boolean value with the objective of minimize the “noise” in its utilization. The only exception to the above would be the attempt to register new messages after the Factory has explicitly forbid it.

## Design and Implementation:

### Public API:

The information regarding to a Message would be wrapped inside a MessageBase class. This should include the following fields that would be used by the MessageContainer ONLY:

*   
  Set of types that are allowed to publish this message.
*   
  Set of listeners subscribed to this message’s publishing.

In order to code defensively, a generic abstract class would inherit from the above MessageBase class. Every instance of a message should inherit from this class. The goal of this class is to offer a strong type context regarding to the EventArgs class that the Message would work with. So far:

*   
  This approach would allow to the different MessageContainer implementation to ensure that the message’s to be subscribed are strong typed avoiding the casting’s complexity on the user side and promoting a type safe behavior. Every message that would be register to the container should inherit from this class.

Almost every MessageContainer’s API would require to be provided with the MessageType and its MessageEventArgs. This is making use of the fact that all the allowed messages should inherit from MessageGenericBase and args from EventArgs.

This is defensive code. It will make the method calls a little harder because the need to, almost always, specify the message type and its matching EventArgs. But, at the long run, what would be better, a little larger statements or hours debugging trying to notice what is wrong?

The way the MessageContainer’s API is designed would force the MessageContainer’s user to always provide the right types without losing any grade of extensibility.

#### Registering Messages:

The MessageContainer should provide methods for registration of messages. These same methods should allow the Factory to indicate the types that are allowed to publish this message: Even when “anyone” is allowed to subscribe to a message not everyone should be allowed to publish them.

* 
* 

Even when the Register Method receives the message instance to register, so far, the messages would be unique for each type.

Finally, the container provide a method oriented to block all future registering of instances:

*   
  A call to this method should guarantee that no future calls to instance registration’s methods would be allowed.

The only case in which an exception would be generated (for the entire API) is when calling any RegisterMessage method after EndRegistering has been called.

#### Publishing Methods:

The MessageContainer should provide one method to allow the publishers to publish specific messages:

*   
  The method expect to receive, along, of course, the MessageEventArgs, an object which represents the publisher’s instance. The type of this object is tested against the allowed publishers for the message it’s trying to publish. If the type does not match it will mean that the given publisher is not allowed to publish that kind of message.  
  Of course, any “evil” developer can fake this by specifying an allowed publisher although the type which is really calling the method isn’t allowed. So far, every approach I’ve checked to overcome this vulnerability constitute a next problem by itself. After all, one can’t never be enough cautious.  
  Also, a “stupid” developer, can think this parameter is shit and pass it as null. It will, of course, generate an exception.

#### Subscriber’s Methods:

Finally, the MessageContainer should provide methods to allow other types to subscribe themselves to any kind of message as well as remove their previous subscriptions.

*   
  Following the same defensive approach, the subscribers should explicit specify not only the type of the message they are attempting to subscribe to but also the type of the EventArgs it expect to receive.  
  This method, along, of course, a messageHandler for the MessageType would expect to receive the instance of the type that is subscribing to the method. This instance would be used as the identifier of the subscriber, allowing him to remove the subscription in the future.
*   
  This method breaks the defensive approach so far because a very simple assumption: Because the only allowed MessageType are the ones who inherits form MessageGeneric, then it would be impossible to remove the subscription to a MessageType who violate this rule.  
  The method expect to receive the instance of the object that is requesting remove the subscription. It will simply locate the given key on the message subscribers.

All these methods are exception-safe, meaning that they will never throw an exception by themselves. If the expected operations couldn’t be completed it only returns false.

### Proposed Internal Structure:

The Sample Implementation I will provide is designed to reflect (and test) all the features exposed in this document.

This implementation is proposed to implement the Classic Thread-Safe Singleton Pattern. So far we don’t conceive a situation in which a non-singleton MessageContainer would be needed.

#### Fields:

The MessageContainer would store its registered message in simple dictionaries.

*   
  The key (Type) should be the Message type while the Value should be any message who inherit from MessageBase.  
  Due to the impossibility of constraint the Value to a MessageGeneric (that would be silly thing) we are engages to constraint this Value on insertion (Registering) time.
*   
  Flag value initialized in true, that will worth as the sign that the EndRegistering() has been called.

#### Methods:

So far all the methods the MessageContainer implements are the implementations of its API, but for one helper method:

*   
  Will attempt to get the given message by its type. If the message doesn’t exists, it will return null.

## Working Example:

  
