MESSAGE CHANNEL DECISIONS

*Message Channel*s are consider the decisions involved in using them:

One-to-one or one-to-many — When your application shares a piece of data, do you want to share it with just one other application or with any other application that is interested? To send the data to a single application, use a *Point-to-Point Channel*. This does not guarantee that every piece of data sent on that channel will necessarily go to the same receiver, because the channel might have multiple receivers; but it does ensure that any one piece of data will only be received by one of the applications. If you want all of the receiver applications to be able to receive the data, use a *Publish-Subscribe Channel*. When you send a piece of data this way, the channel effectively copies the data for each of the receivers.

What type of data — Any data in any computer memory has to conform to some sort of *type*: a known format or expected structure with an agreed upon meaning. Otherwise, all data would just be a bunch of bytes and there would be no way to make any sense of it. Messaging systems work much the same way; the message contents must conform to some type so that the receiver understands the data's structure. *Datatype Channel* is the principle that all of the data on a channel has to be of the same type.

This is the main reason why messaging systems need lots of channels; if the data could be of any type, the messaging system would only need one channel (in each direction) between any two applications.

Invalid and dead messages — The message system can ensure that a message is delivered properly, but it cannot guarantee that the receiver will know what to do with it. The receiver has expectations about the data's type and meaning; when it receives a message that doesn't meet these expectations, there's not much it can do. What it can do, though, is put the strange message on a specially designated *Invalid Message Channel*, in hopes that some utility monitoring the channel will pick up the message and figure out what to do with it. Many messaging systems have a similar built-in feature, a *Dead Letter Channel* for messages which are successfully sent but ultimately cannot be successfully delivered. Again, hopefully some utility monitoring the channel will know what to do with the messages that could not be delivered.

Crash proof — If the messaging system crashes or is shut down for maintence, what happens to its messages? When it is back up and running, will its messages still be in its channels? By default, no; channels store their messages in memory. However, *Guaranteed Delivery* makes channels persistent so that their messages are stored on disk.

This hurts performance but makes messaging more reliable, even when the messaging system isn’t.

Non-messaging clients — What if an application cannot connect to a messaging system but still wants to participate in messaging? Normally it would be out of luck; but if the messaging system can connect to the application somehow—through its user interface, its business services API, its database, or through a network connection such as TCP/IP or HTTP—then a *Channel Adapter* on the messaging system can be used to connect a channel (or set of channels) to the application without having to modify the application and perhaps without having to have a messaging client running on the same machine as the application. Sometimes the "non-messaging client" really is a messaging client, just for a different messaging system. In that case, an application that is a client on both messaging systems can build a *Messaging Bridge* between the two, effectively connecting them into one composite messaging system.

Communications backbone — As more and more of an enterprise’s applications connect to the messaging system and make their functionality available through messaging, the messaging system becomes a centeralized point of one-stop-shopping for functionality in the enterprise.

A new application simply needs to know which channels to use to request functionality and which others to listen on for the results. The messaging system itself essentially becomes a *Message Bus*, a backbone providing access to all of the enterprise's various and ever-changing applications and functionality. You can achieve this integration nirvana more quickly and easily by specifically designing for it from the beginning.

So as you can see, getting applications set up for *Messaging* involves more than just connecting them to the messaging system so that they can send messages. The messages must have *Message Channel*s to transmit on. Slapping in some channels doesn't get the job done either. They have to be designed with a purpose, based on the data type being shared, the sort of application making the data available, and the sort of application receiving the data.

Source:

http://www.enterpriseintegrationpatterns.com/patterns/messaging/MessagingChannelsIntro.html