Messaging Channels

System Integration
PBA Softwareudvikling/BSc Software Development
Tine Marbjerg
Fall 2018

Today’s Agenda

- Exercise Coffee Shop (MsgKit.zip)
  - What did we learn?
- Messaging Channels patterns (EIP Chapter 4)
- Message Construction patterns (EIP Chapter 5)
  - Only briefly covered today via exercises
- Programming exercises with MSMQ and AMQP protocols

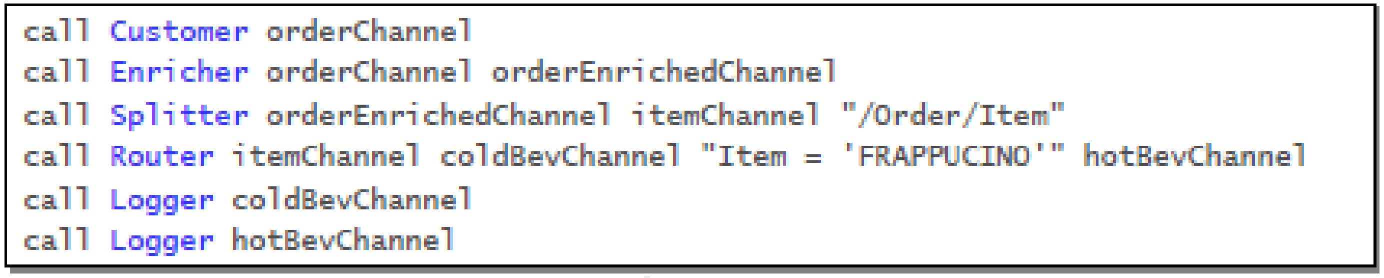
Coffee Shop Exercise

Follow up

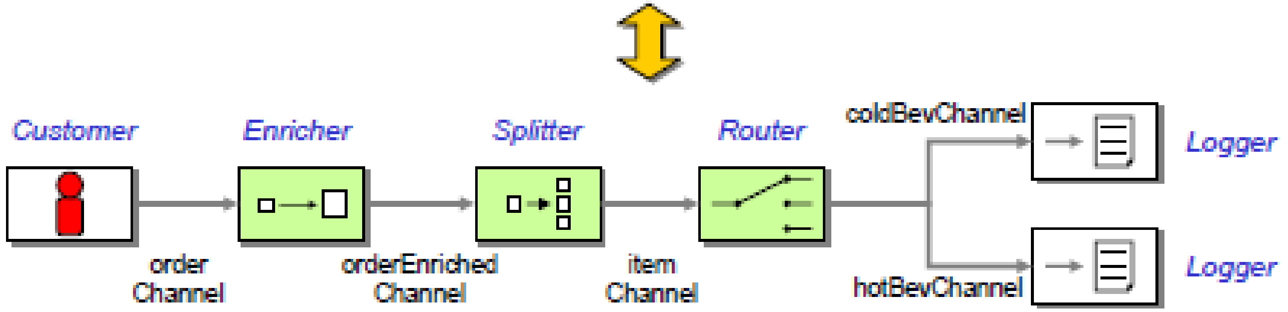
What did you learn?

How to run Coffee Shop exercises

- Composition of solution from predefined components (.bat files)
- Components interact in *Pipes & Filter* Architecture. What does that mean?



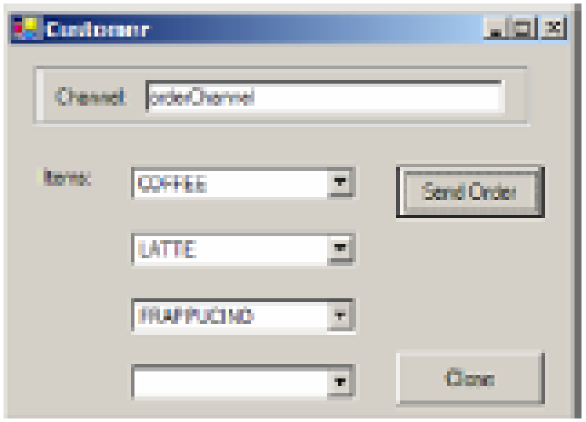
```
call Customer orderChannel
call Enricher orderChannel orderEnrichedChannel
call Splitter orderEnrichedChannel itemChannel "/Order/Item"
call Router itemChannel coldBevChannel "Item = 'FRAPPUCINO'" hotBevChannel
call Logger coldBevChannel
call Logger hotBevChannel
```



NB! Exercises use Messaging Domain Specific Language listed in Tutorial Reference Chart

Convenience and Test Components

Customer

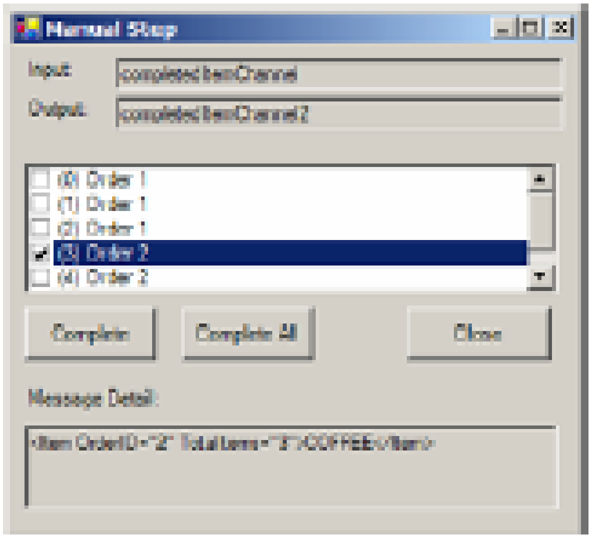


Channel: orderChannel

Items: COFFEE, LATTE, FRAPPUCINO

Sends order messages to specified channel

Manual Step



Input: completedItemChannel

Output: completedItemChannel2

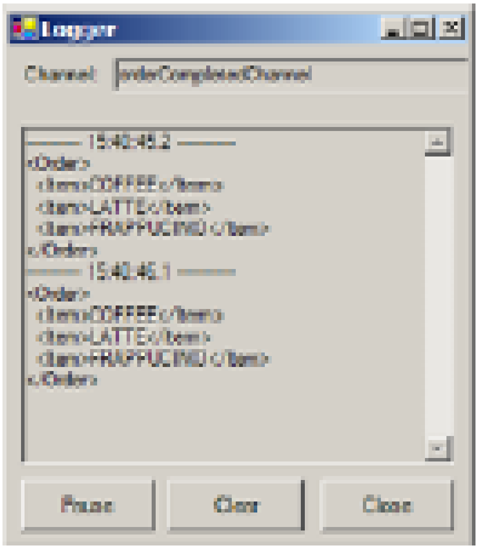
- [ ] (0) Order 1
- [ ] (1) Order 1
- [ ] (2) Order 1
- [x] (3) Order 2
- [ ] (4) Order 2

Message Detail:

```
<Item OrderID="2" TotalItems="3">COFFEE</Item>
```

Allows inspection of messages and out-of-sequence completion

Logger



Channel: orderCompletedChannel

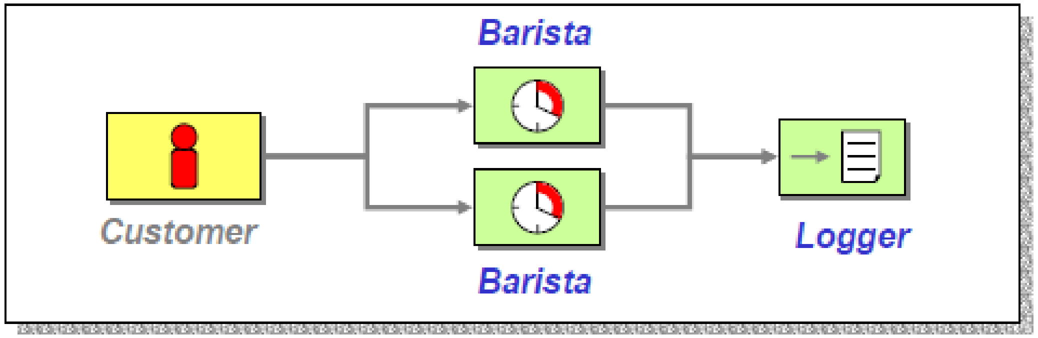
```
——— 15:40:45.2 ———
<Order>
  <Item>COFFEE</Item>
  <Item>LATTE</Item>
  <Item>FRAPPUCINO</Item>
</Order>
——— 15:40:46.1 ———
<Order>
  <Item>COFFEE</Item>
  <Item>LATTE</Item>
  <Item>FRAPPUCINO</Item>
</Order>
```

Display messages and time stamps

Coffee Shop follow-up – exercise 1

Exercise 1

- Higher throughput with 2 baristas
  - 1 barista: 1 coffee per second
  - 2 baristas: 2 coffee per second



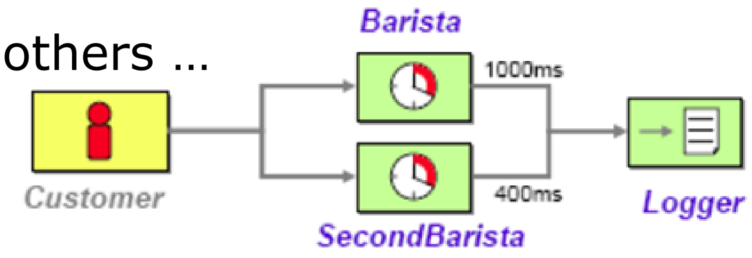
Observation

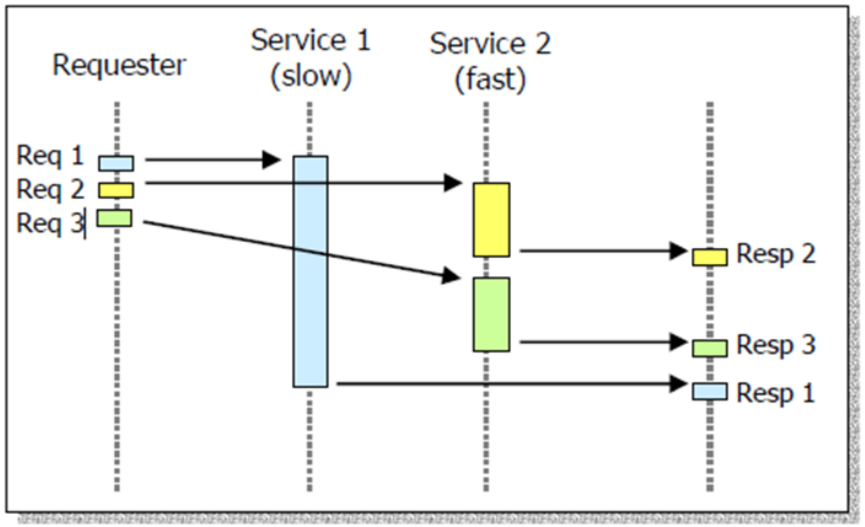
- Messaging architectures scale through *Competing Consumers*
- Scalability: Adding more baristas did not require changes to the architecture or existing components

Coffee Shop follow-up – exercise 2

Exercise 2

Some components might be faster than others …



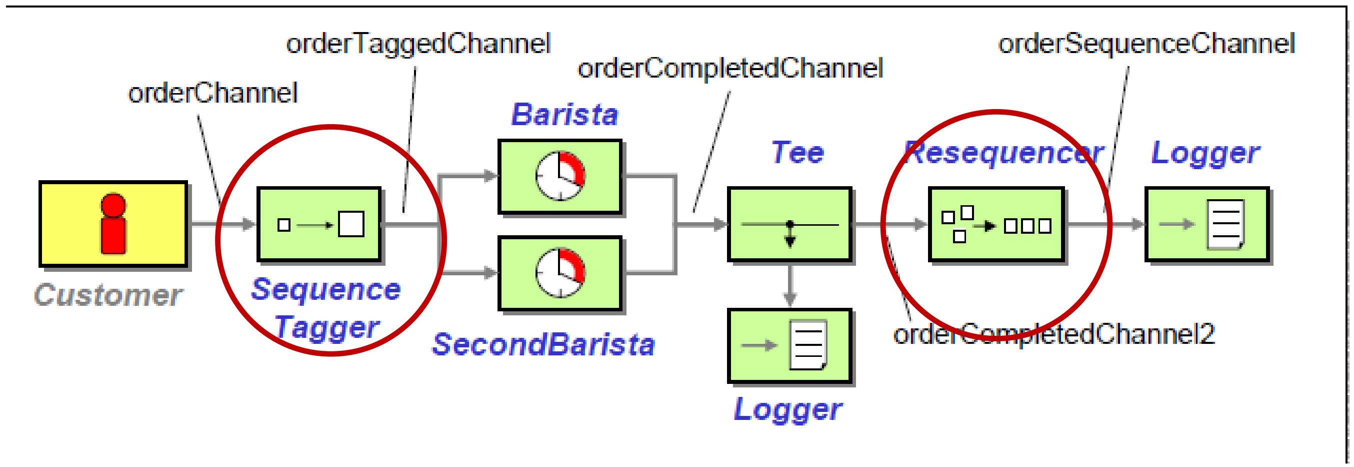


Observation

- Parallel processing may cause messages to get out of order
- We need a stateful filter to collect and re-order messages so that they can be published to an output channel in a specified order

Coffee Shop follow-up – exercise 2

Possible solution to sequencing problem:



- SequenceTagger (i.e. *Content Enricher*) adds consecutive numbers to messages
- *Resequencer* brings messages back in order
  - stateful component which needs to persist messages to be robust
  - Resequencing increases latency because it holds messages
  - One missing message can stall everything

Coffee Shop follow-up – exercise 3

Exercise 3

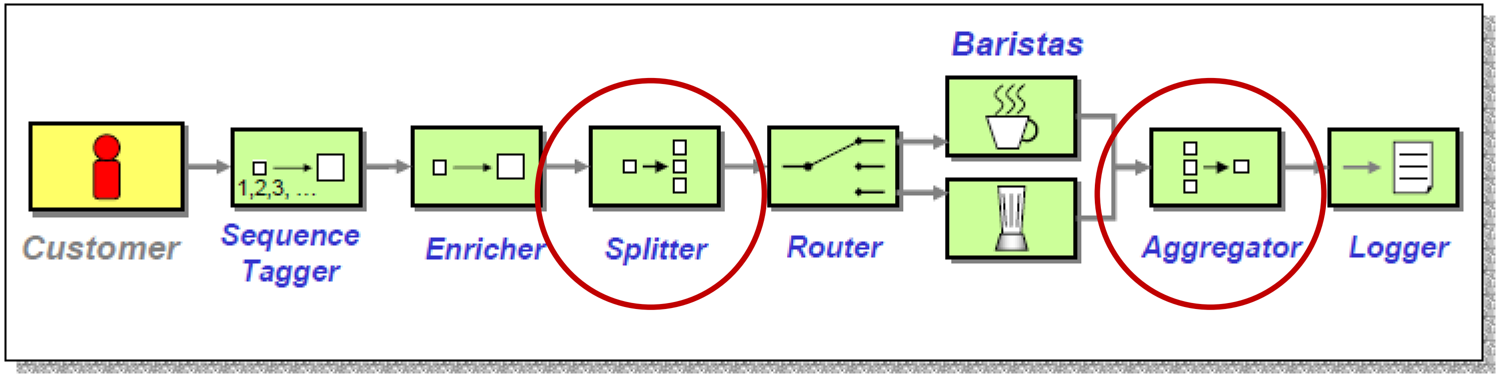
- Processing a whole order at one time limits our scaling options
- Creating a specialized Barista each for iced beverages and for hot beverages allows us to fine-tune baristas

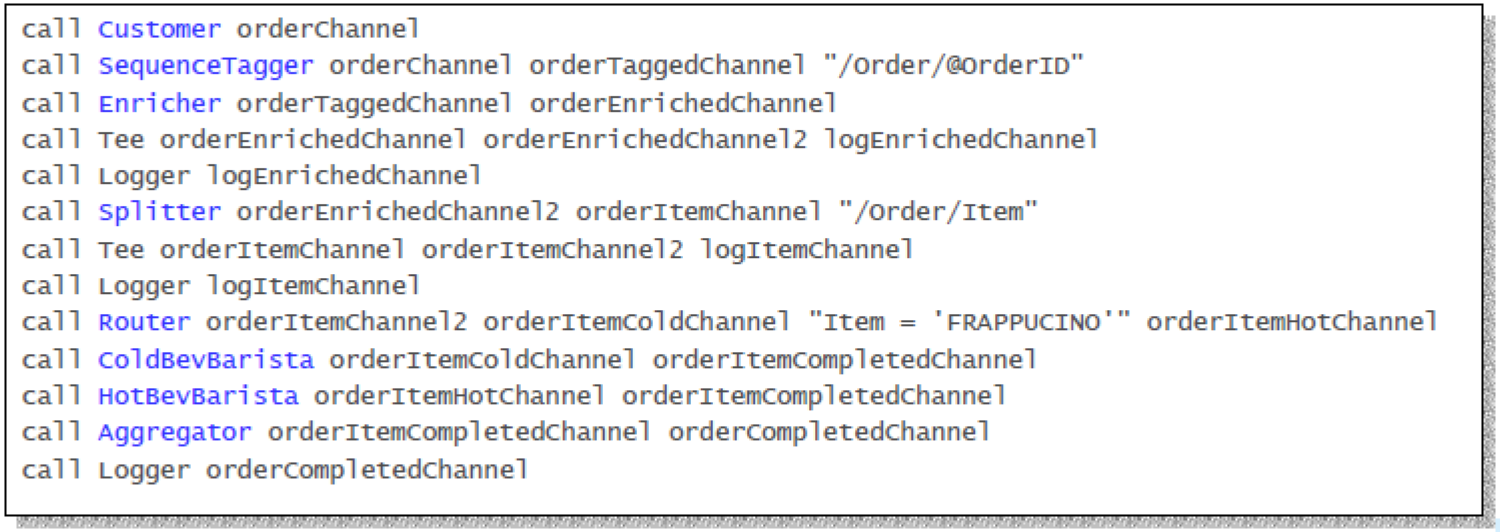
Observations

- Splitting allows different message types to be processed individually.
- Separating tasks into smaller pieces can improve throughput for the application and support greater scalability.
- Messages will get out of order and need to be re-aggregated.

Coffee Shop follow-up – exercise 3

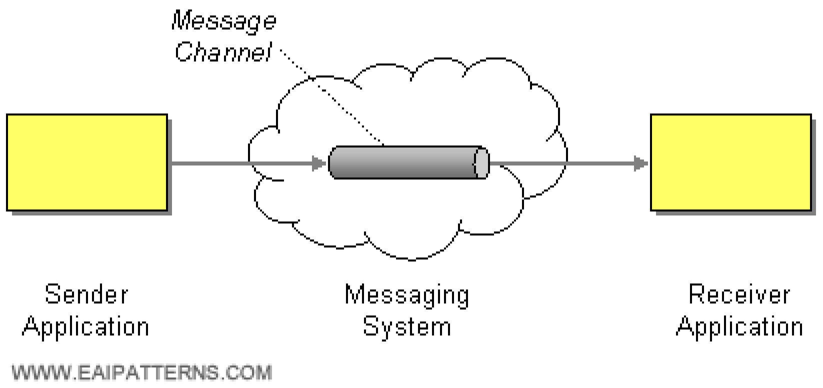
- Possible solution exercise 3:





```
call Customer orderChannel
call SequenceTagger orderChannel orderTaggedChannel "/Order/@OrderID"
call Enricher orderTaggedChannel orderEnrichedChannel
call Tee orderEnrichedChannel orderEnrichedChannel2 logEnrichedChannel
call Logger logEnrichedChannel
call Splitter orderEnrichedChannel2 orderItemChannel "/Order/Item"
call Tee orderItemChannel orderItemChannel2 logItemChannel
call Logger logItemChannel
call Router orderItemChannel2 orderItemColdChannel "Item = 'FRAPPUCINO'" orderItemHotChannel
call ColdBevBarista orderItemColdChannel orderItemCompletedChannel
call HotBevBarista orderItemHotChannel orderItemCompletedChannel
call Aggregator orderItemCompletedChannel orderCompletedChannel
call Logger orderCompletedChannel
```

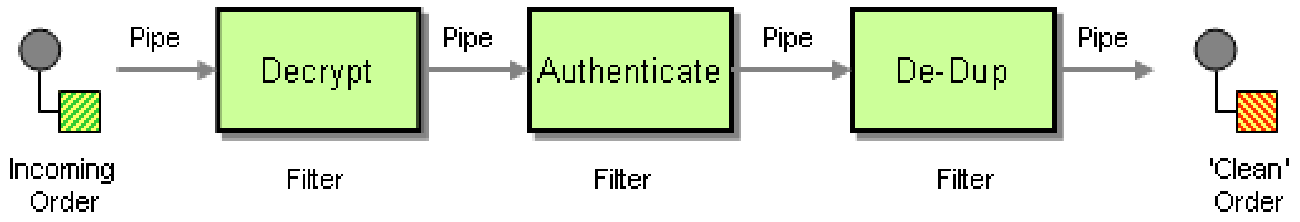
Overall considerations about messaging channels



Message Channel Characteristics I

Fixed set of channels

- Number of channels tends to be static - agreed upon at design time
  - Possible exception: reply channel in *Request-Reply*

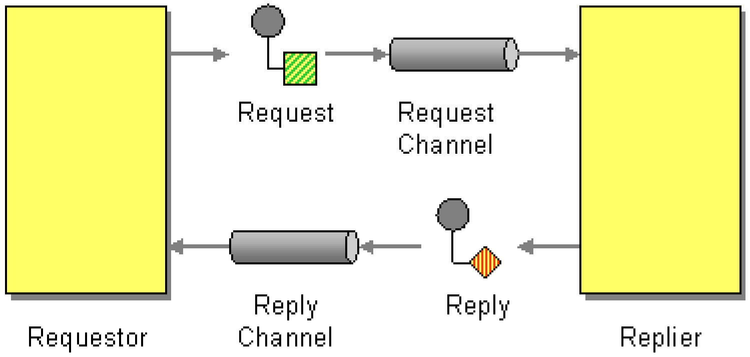


Message Channel Characteristics II

Unidirectional channels

- Channels are like buckets that applications add and take data from, but message gives direction

- For practical reasons, two-way communication need two channels (i.e. makes channels unidirectional)



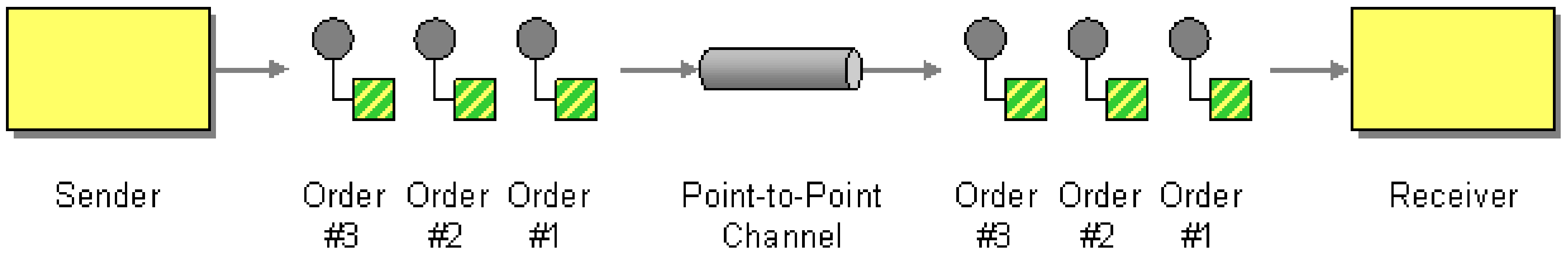
Message Channel Decisions (1)

One-to-one or one-to-many channel?

- Message will be received by only one application (*Point-to-Point*)
- Message copied for each of the receivers (*Publish-Subscribe*)

Point-to-Point Channel (103)

- How can the caller be sure that exactly one receiver will receive the document or perform the call?



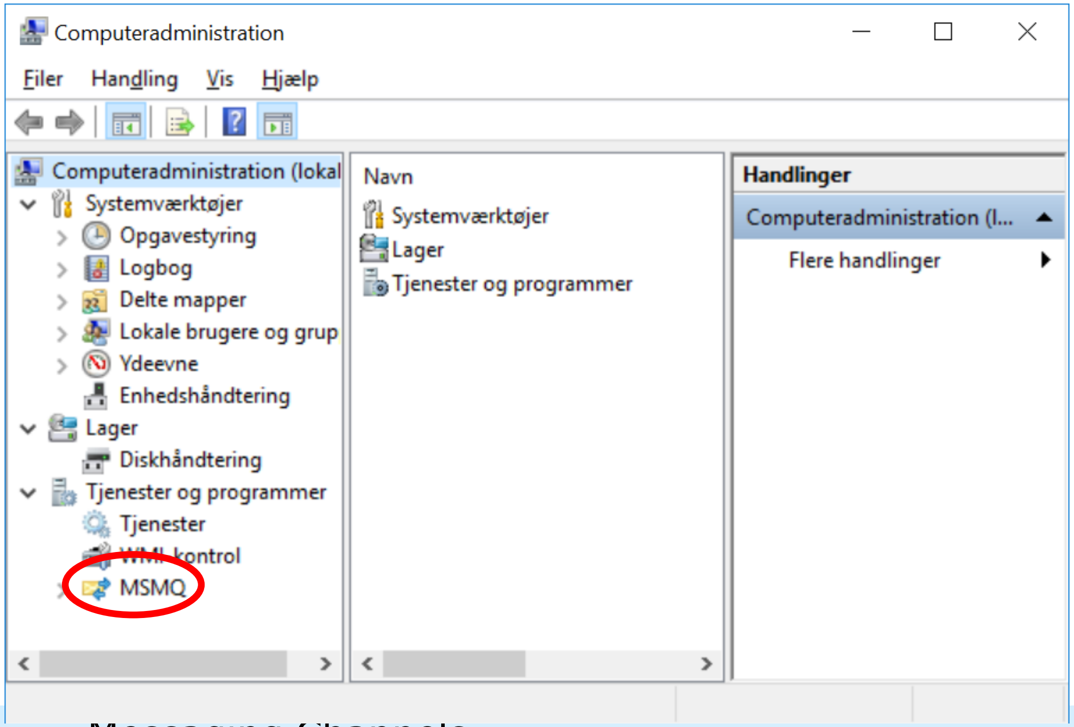
- Send the message on a *Point-to-Point Channel*, which ensures that only one receiver will receive a particular message.

OBS!

- If the channel has multiple receivers, only <u>one</u> of them can successfully consume a particular message.
- The <u>channel</u> <u>ensures</u> that only one of them succeeds, i.e. the receivers do not have to coordinate with each other.

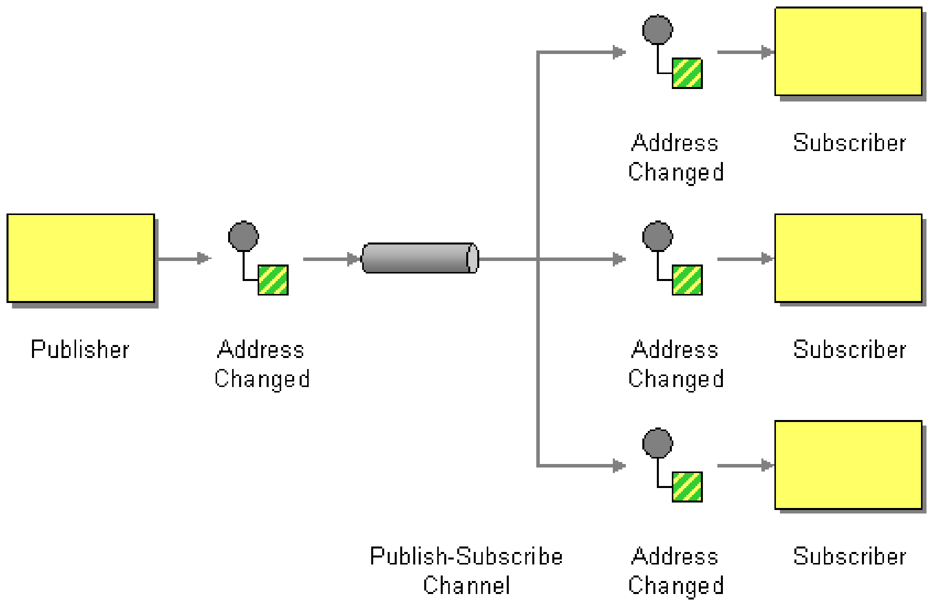
MSMQ Demo

- Let’s see some C# code working on local MSMQ queue
- You can see what happens in Computer Administration window:



Publish-Subscribe Channel (106)

- How can the sender broadcast an event to all interested receivers?



- Send the event on a *Publish-Subscribe Channel*, which delivers a copy of a particular event to each receiver.
  - One input channel splits into multiple output channels
  - Each output channel has only one subscriber
  - MSMQ doesn't support natively

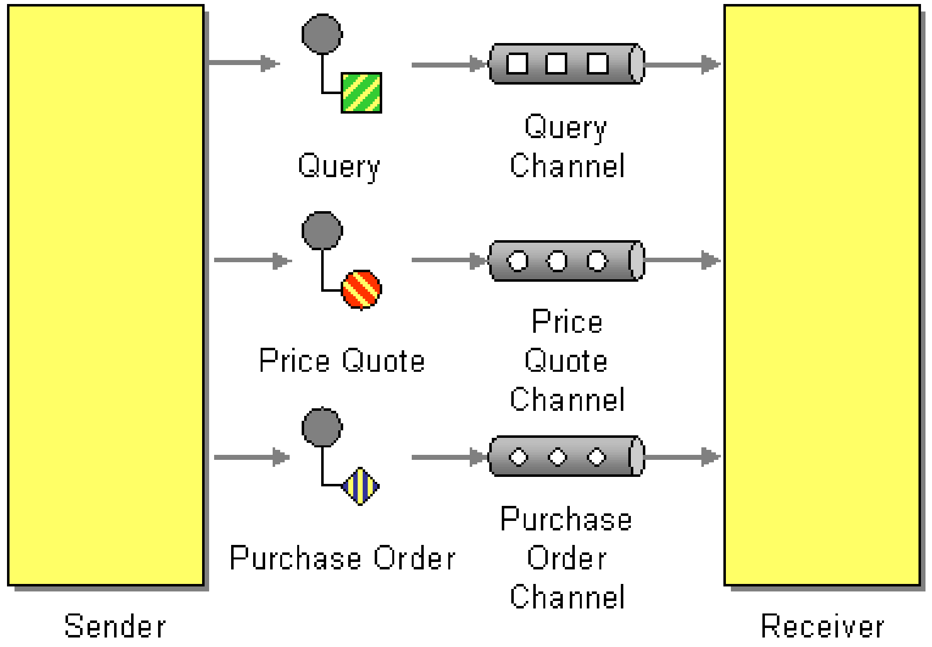
Message Channel Decisions (2)

What type of data on channel?

- All data on a channel should be of the same type, i.e. same structure, format etc. (*Datatype Channel*)
- Main reason that messaging systems needs lots of channels

Datatype Channel (111)

- How can the application send a data item such that the receiver will know how to process it?



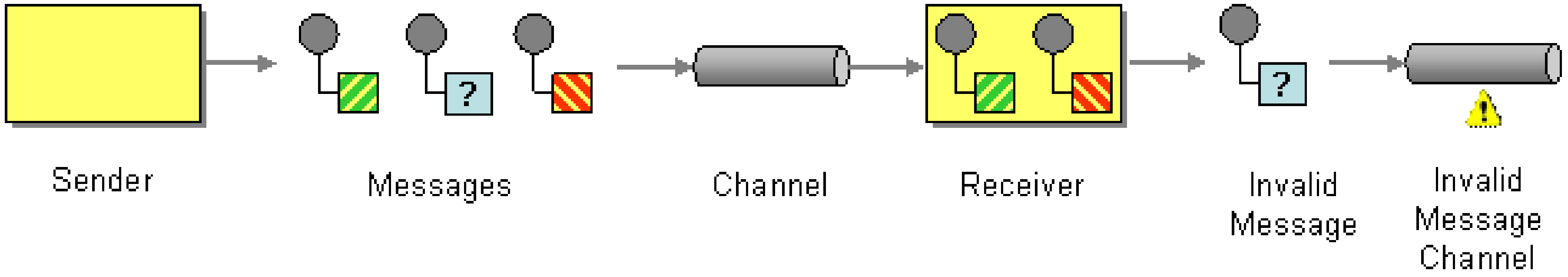
- Use a separate *Datatype Channel* for each data type, so that all data on a particular channel is of the same type.

Message Channel Decisions (3)

- **What happens to invalid and undeliverable messages?**
  - If delivered properly, there is no guarantee the receiver knows what to do
    - Receiver puts the 'strange' message on *Invalid Message Channel*
  - Delivery problem
    - Messaging system puts message on *Dead Letter Channel*

Invalid Message Channel (115)

- How can a messaging receiver gracefully handle a message that makes no sense?



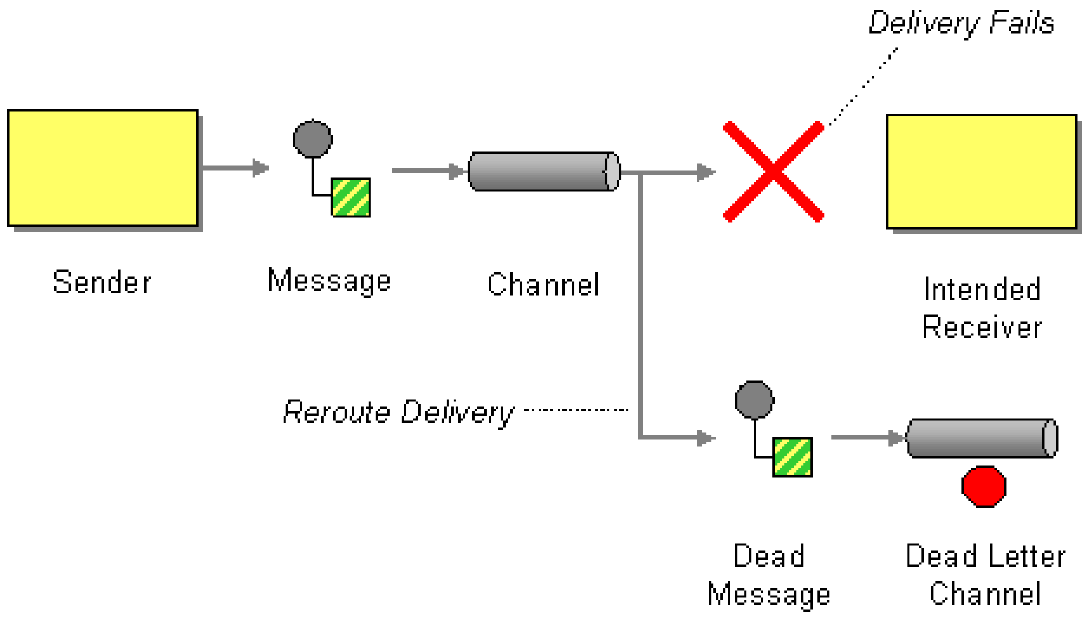
- The receiver should move the improper message to an *Invalid Message Channel*, a special channel for messages that could not be processed by their receivers.

Invalid Message Example

```
//Receiver
...
try {
   // read message
}
catch ( Exception )
{
   //Invalid message detected
   invalidQueue.Send(requestMessage);
}
```

Dead Letter Channel (119)

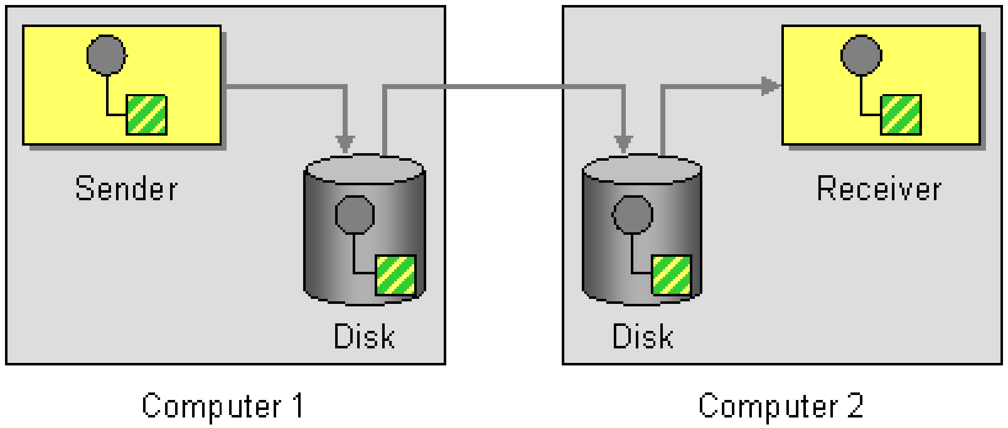
- What will the messaging system do with a message it cannot deliver?



- When a messaging system determines that it cannot deliver a message, it can move the message to a *Dead Letter Channel*.

Guaranteed Delivery (122)

- How can the sender make sure that a message will be delivered, even if the messaging system fails?



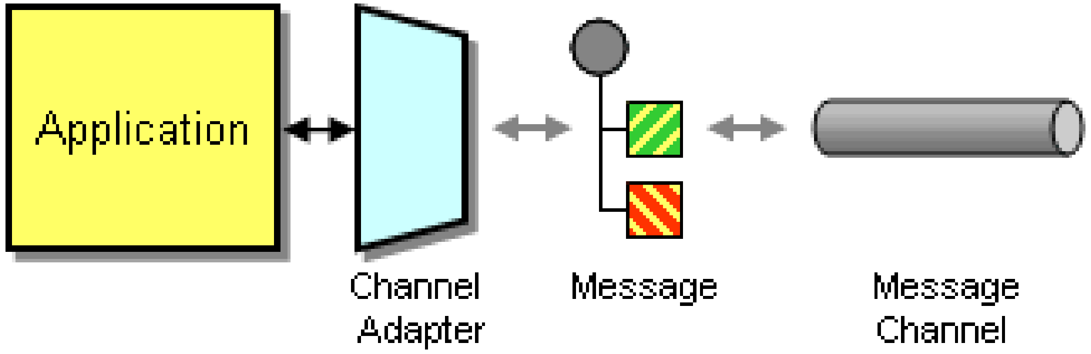
- Use *Guaranteed Delivery* to make messages persistent so that they are not lost even if the messaging system crashes.
  - The msg. system uses a built-in data store to persist messages.
  - Hurts performance, but more reliable

Message Channel Decisions (5)

- **What to do with clients not built for messaging?**
  - *Channel Adapter* makes applications (clients) that cannot connect to a messaging system able to connect to a channel without modifying application

Channel Adapter (127)

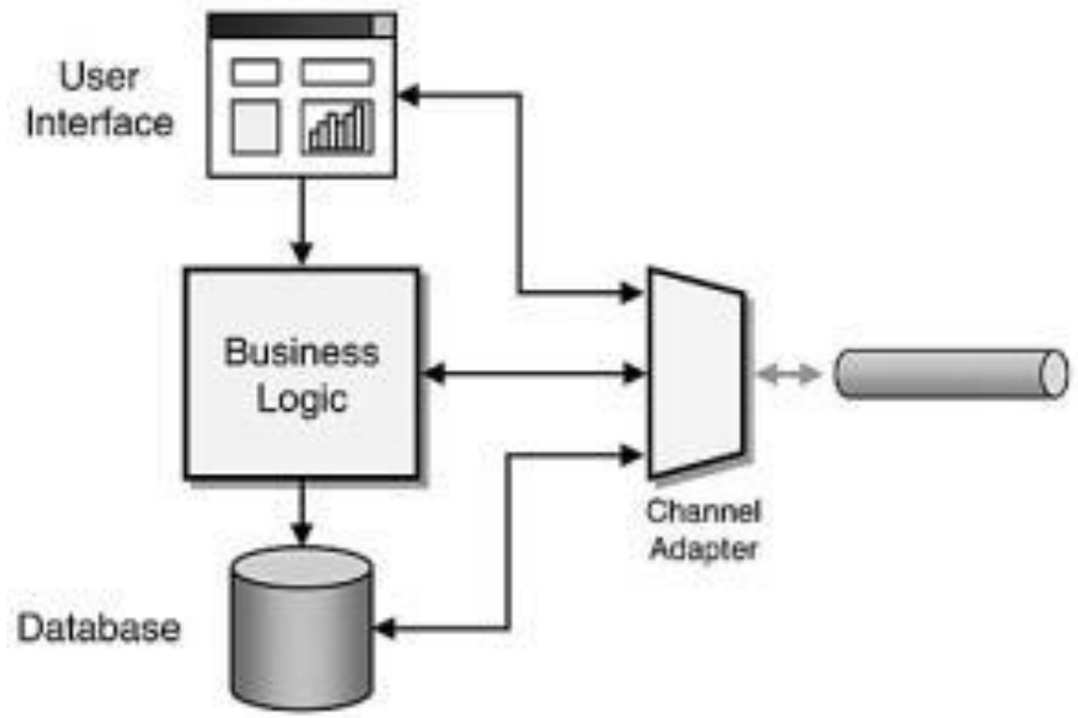
- How can you connect an application to the messaging system so that it can send and receive messages?



- Use a *Channel Adapter* that can access the application's API or data to publish messages on a channel based on this data, and that likewise can receive messages and invoke functionality inside the application.

Adapter - Connect to different layers

- Depending on application architecture, the Channel Adapter can connect to different layers in application:



Connect to different layers – pros & cons

Which layer is best for adaption ?

- User Interface Adapter
  - HTML based UI → make HTTP request and parse result
  - Screen scraping (e.g. from 3270 terminal)
  - ✘ UI typically brittle. Also slow
- Business Logic Adapter
  - Access core functions exposed as API
  - ☑ If well-defined API, often the best solution: More efficient and more stable (API made specifically for access by other applications)
- Database Adapter
  - Data can be extracted from database without application noticing
  - Adapter can add trigger to relevant tables and send messages when changes happen
  - Non intrusive to application, but deep into internals of data structure (brittle if database design changes)

Examples of Data Extraction

- Camel - A routing engine with domain specific languages

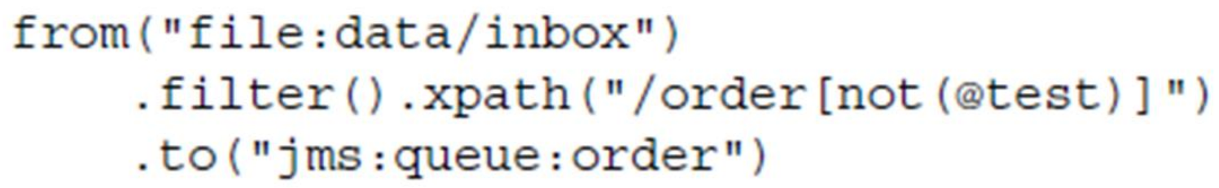
❑ **Java example 1**

Define route that consumes files from a file endpoint to JMS channel:

```
from("file:data/inbox").to("jms:queue:order");
```

❑ **Java example 2**

Messages are routed to a filter, which uses XPath to check whether the message is a test order or not. If message passes the check, it routes to JMS endpoint.



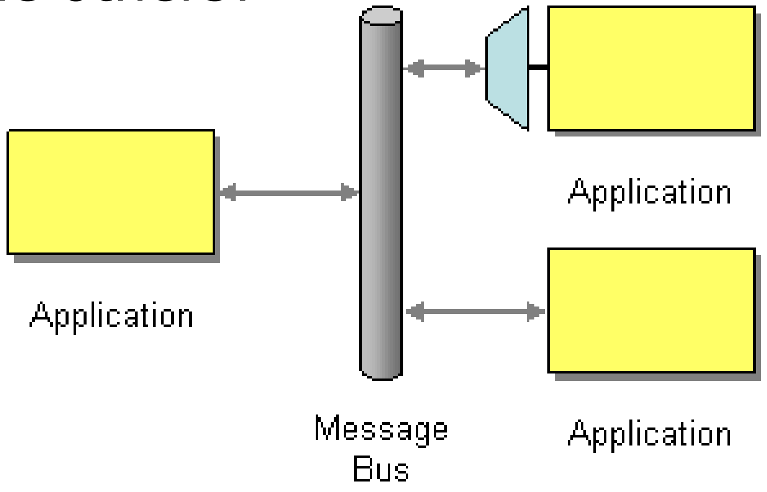
```
from("file:data/inbox")
     .filter().xpath("/order[not(@test)]")
     .to("jms:queue:order")
```

Message Channel Decisions (6)

- **Channels as communication backbone**
  - Messaging system can become a centralized point for shared functionality in the enterprise
  - *Message Bus* architecture*:* a backbone of channels that gives unified access to an enterprise's applications and makes them share functionality

Message Bus (137)

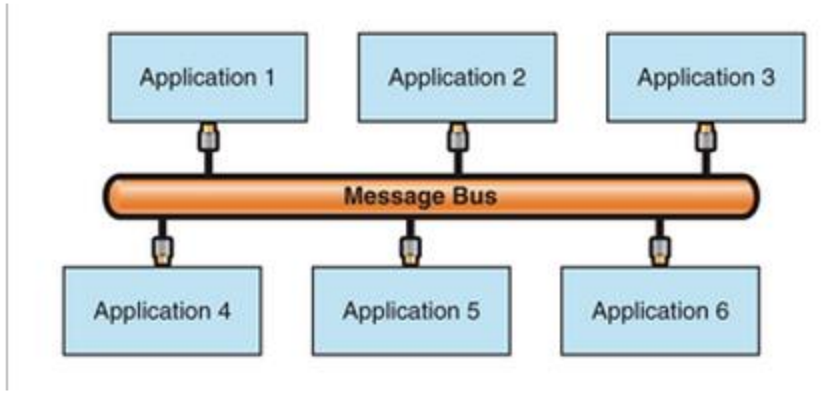
- What architecture enables separate applications to work together, but in a decoupled fashion such that applications can be easily added or removed without affecting the others?

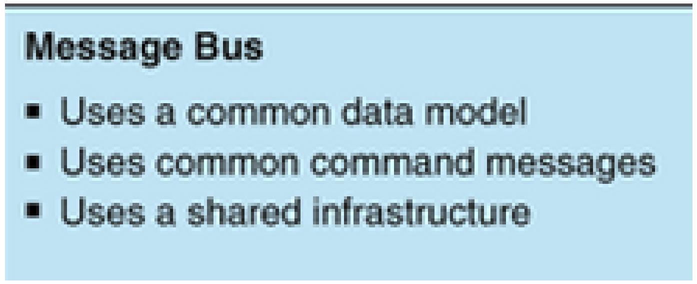


- Structure the connecting middleware between these applications as a *Message Bus* that enables them to work together using messaging.

Applications communicating through bus

- Application that sends messages must prepare the messages so that they comply with the type of messages the bus expects.
- Application that receives messages must be able to understand (syntactically) the message types.
- If all applications in the integration solution implement the bus interface, adding applications or removing applications from the bus incurs no changes.




Message Bus

- Uses a common data model
- Uses common command messages
- Uses a shared infrastructure

C# MSMQ example

```
class Demo  {
        private MessageQueue mq;
        public string myText = "Not initialized";

        private void GetChannel(){
            if (MessageQueue.Exists(@".\Private$\MyQueue1"))
                mq = new System.Messaging.MessageQueue(@".\Private$\MyQueue1");
            else
                mq = MessageQueue.Create(@".\Private$\MyQueue1");
            Console.WriteLine(" Queue Created ");
        }
        private void Populate(){
            Message msg = new System.Messaging.Message();
            myText = "Body text";
            msg.Body = myText;
            msg.Label = "Tine Marbjerg";
            mq.Send(msg);
            Console.WriteLine(" Posted in MyQueue1");
        }
        private string GetResult(){
            Message msg;
            string str = "";
            string label = "";
            try  {
                msg = mq.Receive(new TimeSpan(0, 0, 50));
                msg.Formatter = new XmlMessageFormatter(new String[] { "System.String,mscorlib" });
                str = msg.Body.ToString();
                label = msg.Label;
            }
            catch { str = " Error in GetResult()"; }
            Console.WriteLine(" Received from " + label);
            return str;
        }
        static void Main(string[] args) {
            Demo d = new Demo();
            d.GetChannel();
            d.Populate();
            string result = d.GetResult();
            Console.WriteLine("  send: {0} ", d.myText);
            Console.WriteLine("  receive: {0} ", result);
            Console.ReadLine();
        }
    }
```