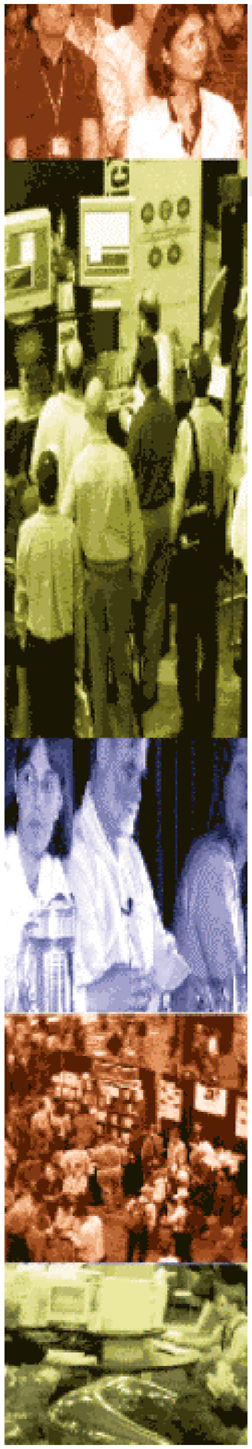
Enterprise Integration Patterns

Gregor Hohpe

ThoughtWorks, Inc.

ThoughtWorks® The art of heavy lifting.™

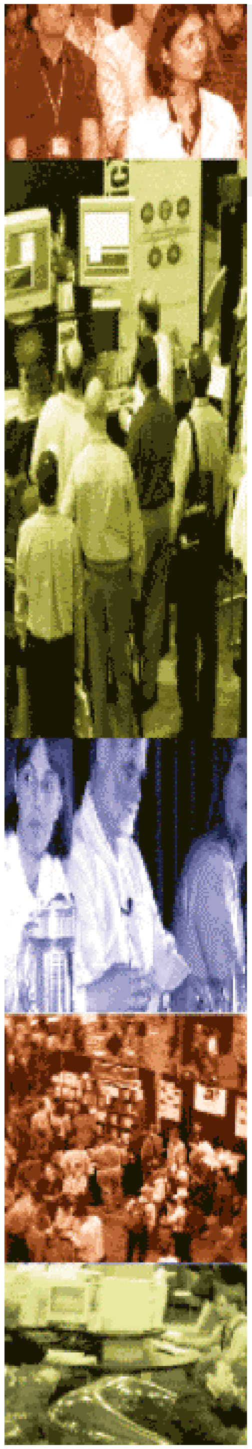


What Are We Talking About?

Enterprise
- Complex applications
- Support vital business functions

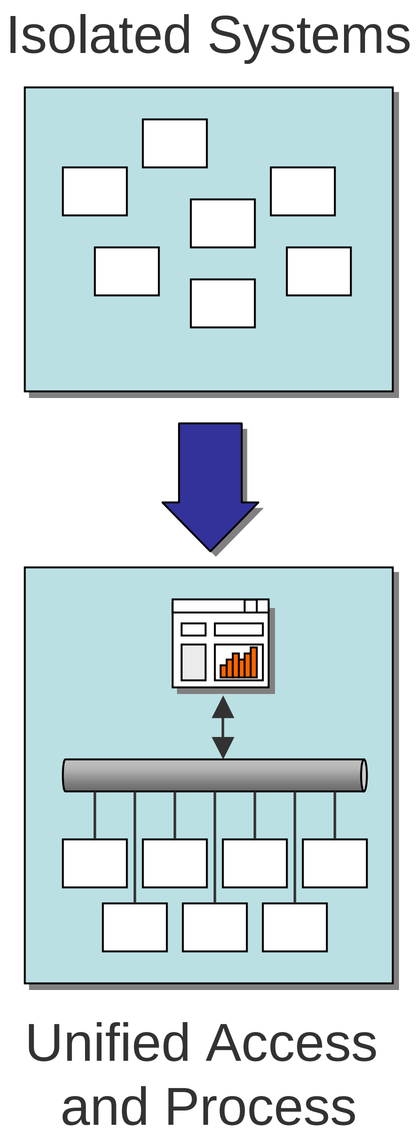
Integration
Communication between two or more applications, users, or business partners

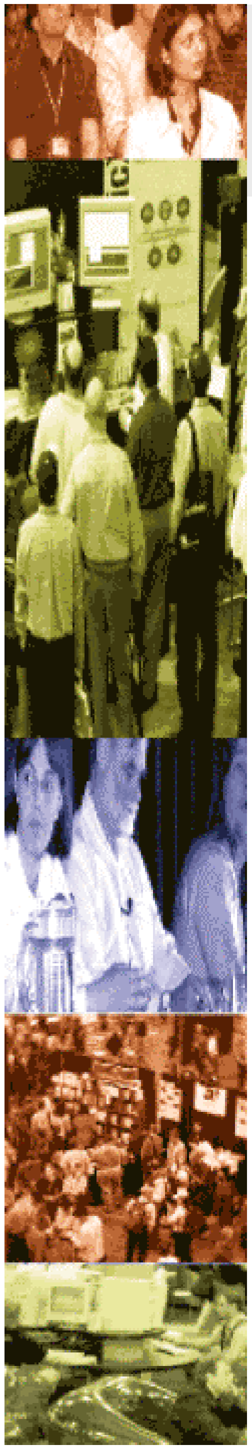
Patterns
- Solution to recurring problem
- Capture knowledge and pass it on
- Establish Vocabulary / Language



Why Do We Need Integration?

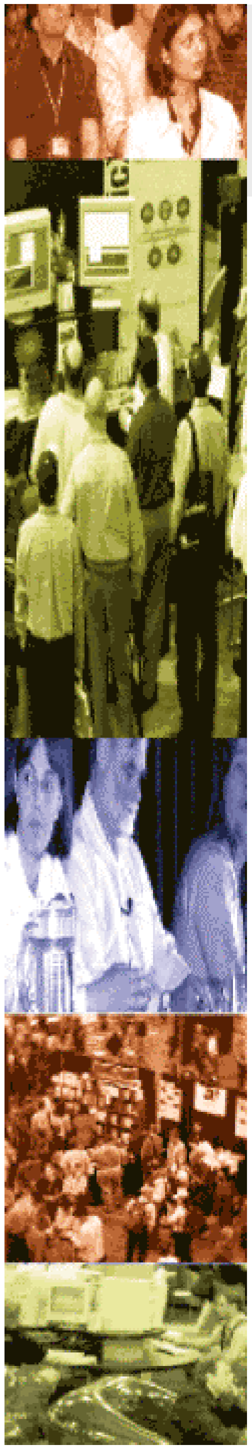
- More than one application (often hundreds or thousands)
  - Single application too hard and inflexible
  - Vendor specialization
  - Corporate politics / organization
  - Historical reasons, e.g. mergers
- Customers see enterprise as a whole, want to execute business functions that span multiple applications
- Need to share information





Why Is Integration Difficult?

- Inherently large-scale and complex
- Underlying paradigm different from object-oriented app. development
- Limited control over entities / applications
- Spans many levels of abstraction
- Far-reaching implications, business critical
- Intertwined with corporate politics
- Few standards exist, still evolving

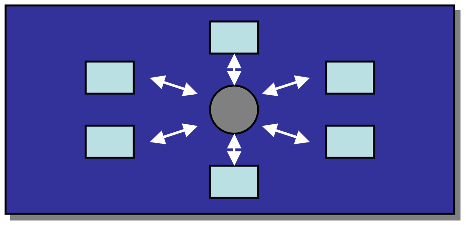


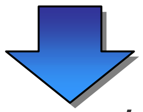
Why Hasn't It Gotten Any Easier?

- Most existing literature either vendor-specific or very high level
- Lots of talk about new standards and specs, but little about best practices for actual use (sort of like Java in early 2000)
- Good integration architects hard to find – even in this job market

Integration Patterns help us to:
- Reduce the gap between high-level vision and reality
- Capture architects' knowledge and experience so it can be reused

High-Level Vision



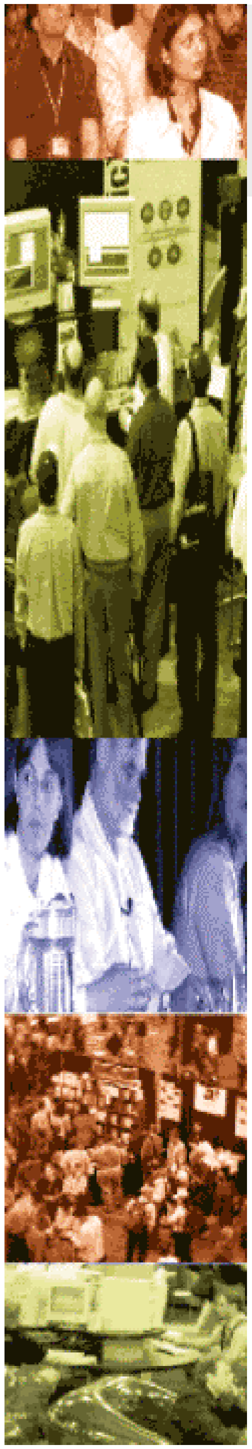


Architecture / Design

Integration Patterns

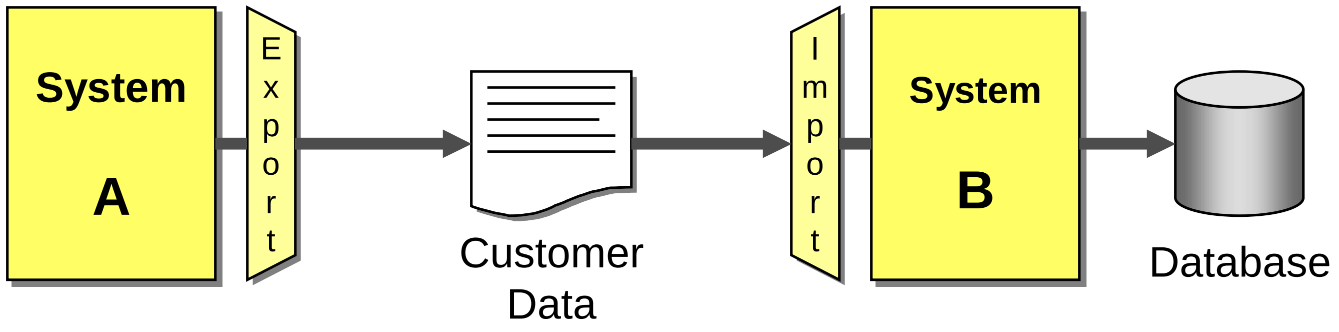
```
MapMessage msg = session.c
msg.setStringProperty(PROP
msg.setString(ITEMID, bid.
```

Implementation



A Brief History in Integration: 70s: Batch Data Exchange

- Export information into a common file format, read into the target system
- Example: COBOL Flat files

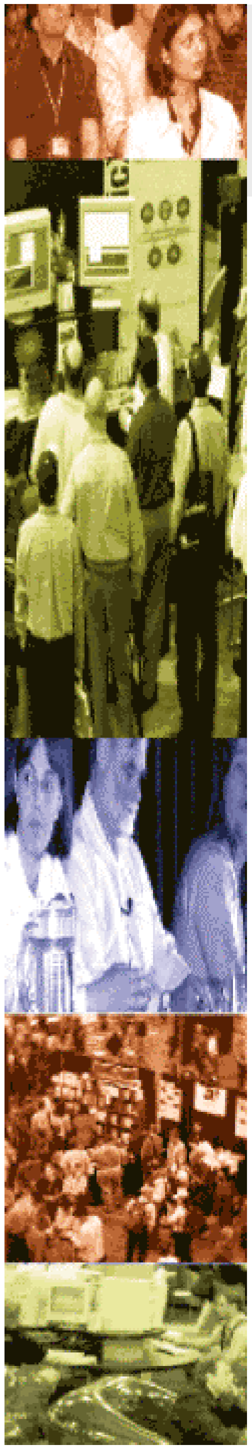


Pros:

- Good physical decoupling
- Language and system independent

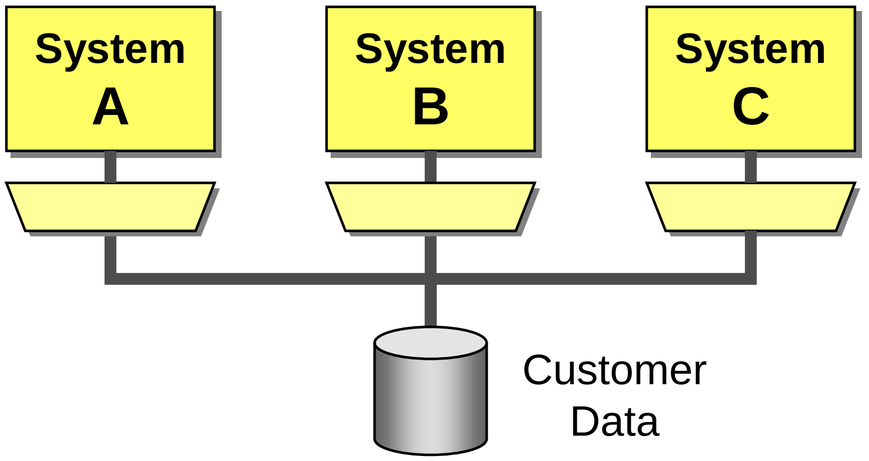
Cons:

- Data transfer not immediate
- Systems may be out of sync
- Large amounts of data



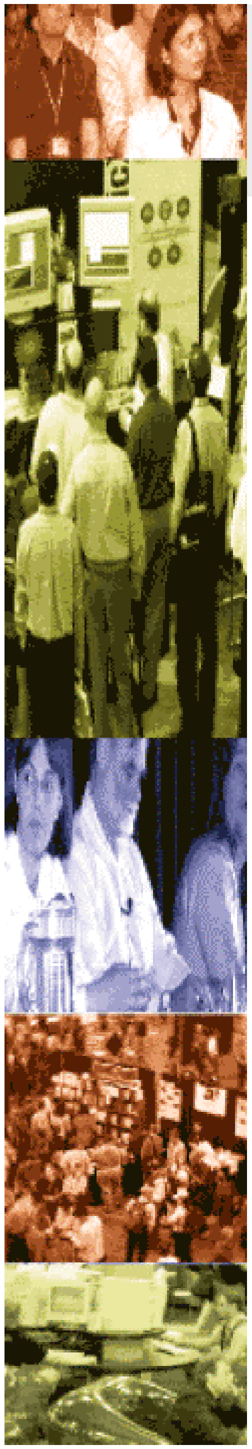
80s: Central Database

- Make all applications access a common database



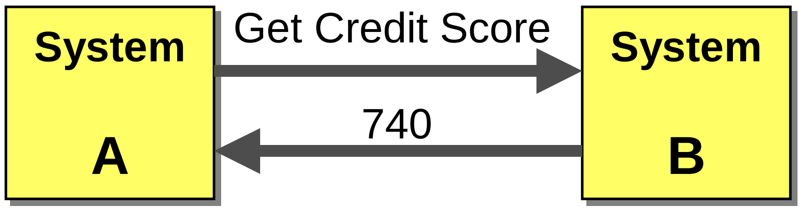
Pros:
- Consistent Data
- Reporting

Cons:
- Integration of data, not business functions
- Difficult to find common representation



90s: Remote Procedure Calls

- One application calls another directly to perform a function.
- Data necessary for the call is passed along. Results are returned to calling application.

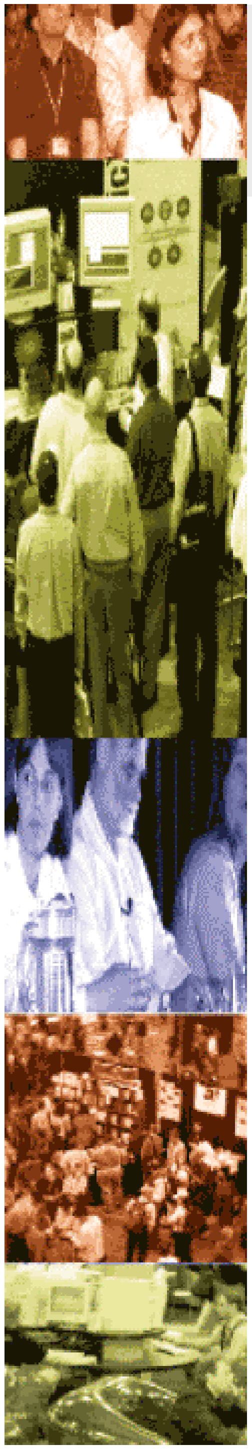


Pros:

- Data exchanged only as needed
- Integration of business function, not just data

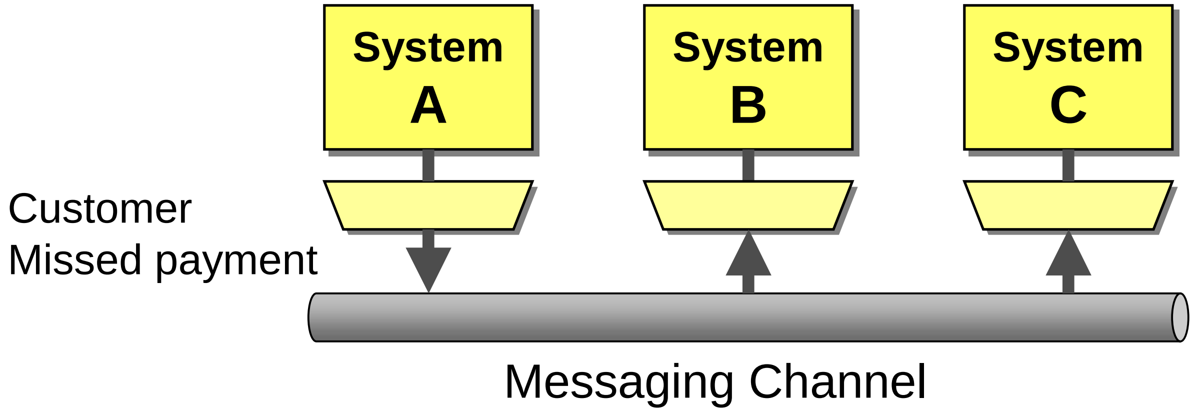
Cons:

- Works well only with small number of systems
- Fragile (tight coupling)
- Performance



Now: Messaging

- Publish events to a bus or queue
- Allow multiple subscribers to a message

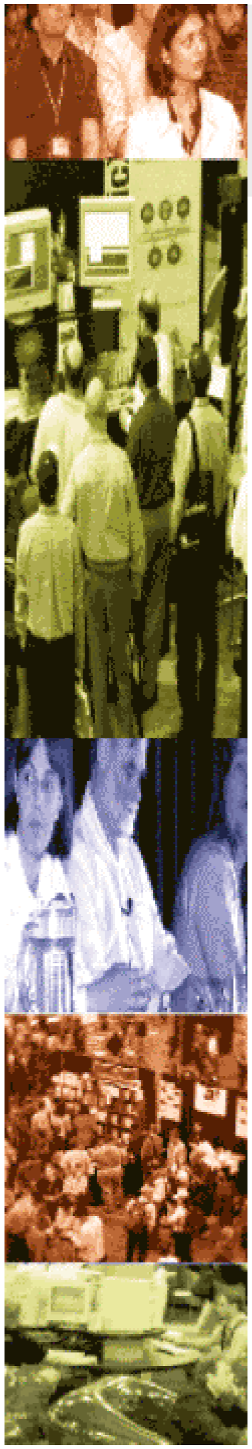


Pros:

- Data exchanged only as needed
- Integration of business function, not just data
- Loose coupling, asynchron.

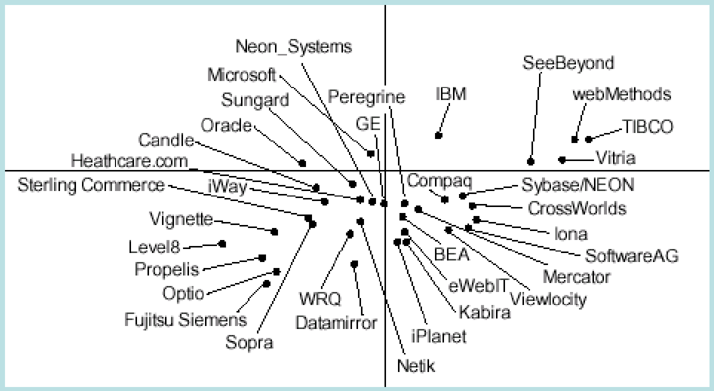
Cons:

- Not familiar
- Difficult to test / debug
- Sometimes you need a synchronous response



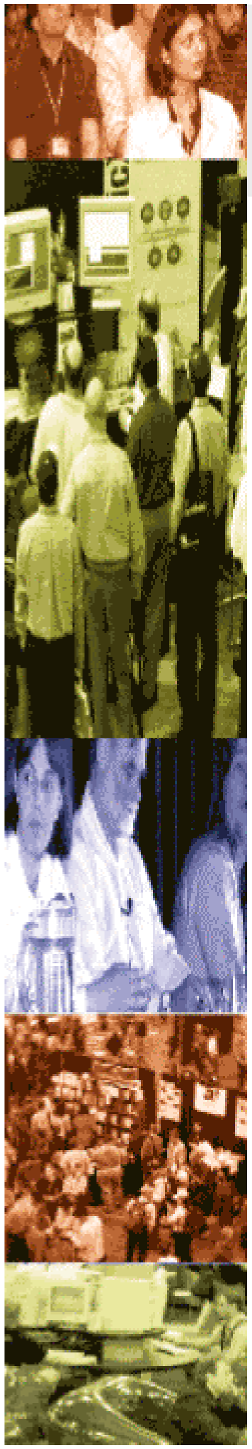
Many Vendors Provide Messaging Tools

- “EAI Vendors”
  - IBM WebSphere MQ
  - TIBCO
  - WebMethods
  - SeeBeyond
  - CrossWorlds etc.
- Java Messaging (JMS)
- Microsoft .NET System.Messaging
- Asynchronous Web Services



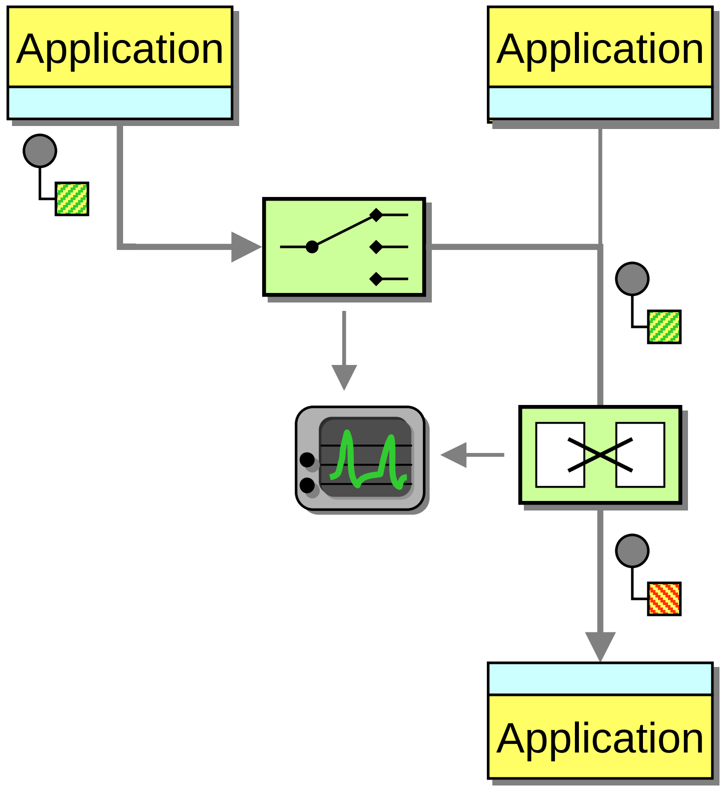
Gartner “Magic Quadrant” for Integration and Middleware (08/2001)

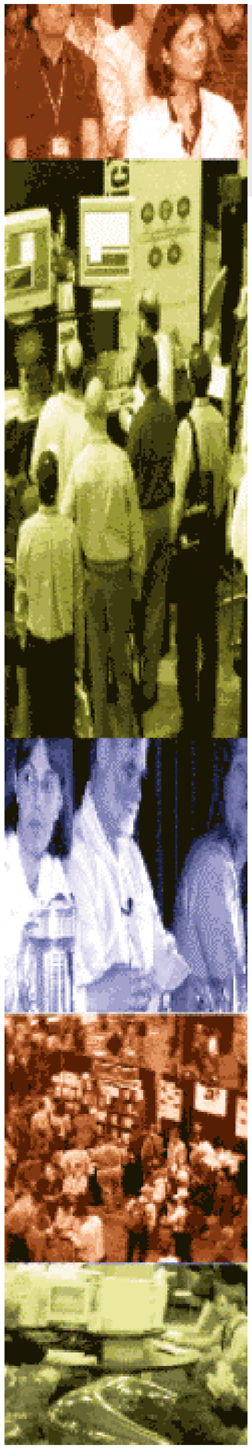
We are looking for vendor-neutral, practical design guidelines and best practices



What Do We Need to Make Messaging Work?

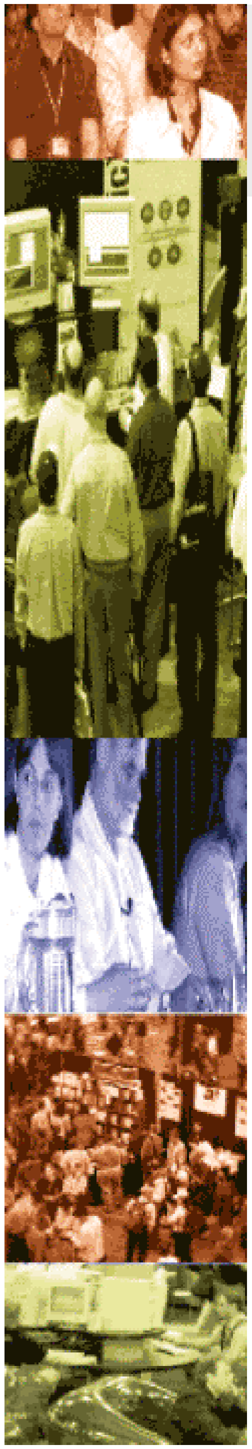
1. Transport messages
2. Design messages
3. Route the message to the proper destination
4. Transform the message to the required format
5. Produce and consume messages
6. Manage and Test the System





What Do We Need to Make Messaging Work?

1. Transport messages	⇨	*Channel Patterns*
2. Design messages	⇨	*Message Patterns*
3. Route the message to the proper destination	⇨	*Routing Patterns*
4. Transform the message to the required format	⇨	*Transformation Patterns*
5. Produce and consume messages	⇨	*Endpoint Patterns*
6. Manage and Test the System	⇨	*Management Patterns*



The Integration Pattern Language (subset)

Channel Patterns
- Message Channel
- Point-to-Point Channel
- Publish Subscribe Channel

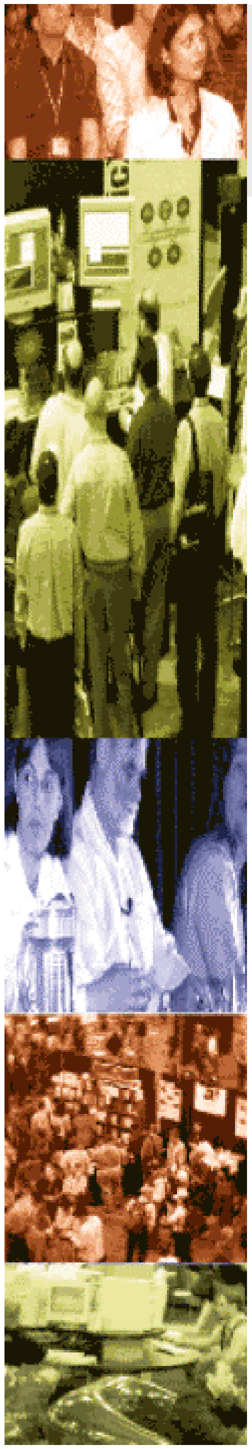
Message Patterns
- Return Address
- Correlation Identifier

Routing Patterns
- Message Router
- Splitter
- Aggregator
- Resequencer
- Auction

Transformation Patterns
- Data Enricher
- Content Filter
- Check Baggage

Endpoint Patterns
- Polling Consumer
- Event-Driven Consumer
- Messaging Mapper

Management Patterns
- Message Store
- Test Message



So What Does One of These Patterns Look Like?

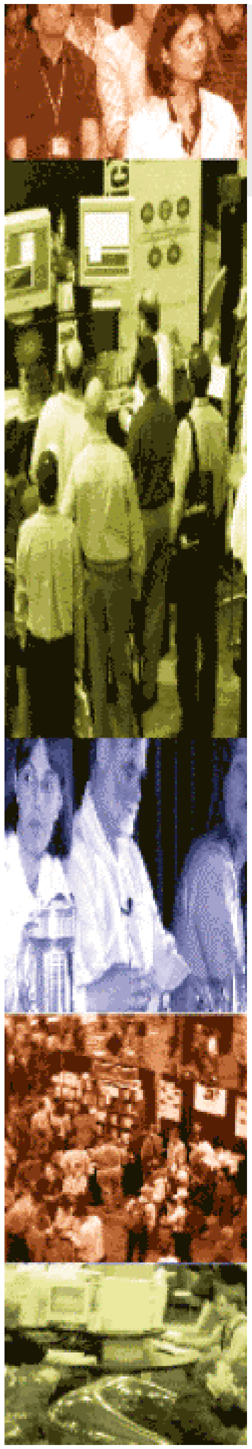
- Context
- Problem
- Forces
- Solution
- Picture
- Resulting Context
- Known Uses
- Related Patterns
- Example
- Icon (optional)

Warning:

If you have worked with messaging, you are likely to have used some of these solutions.

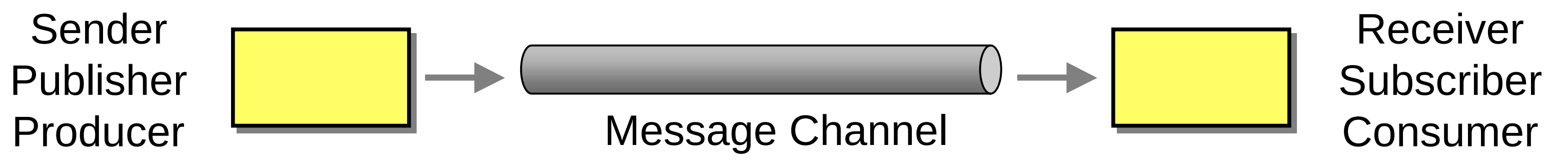
Patterns are harvested from actual use, not “invented”!

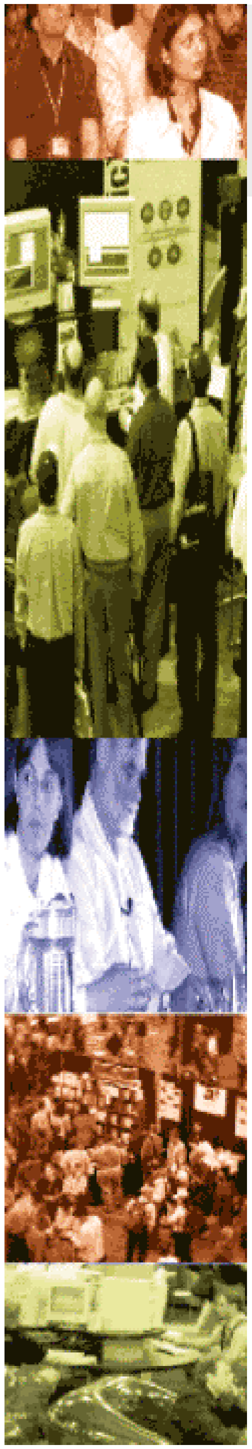
We start simple, go into more depth later.



Channel Pattern: Message Channel

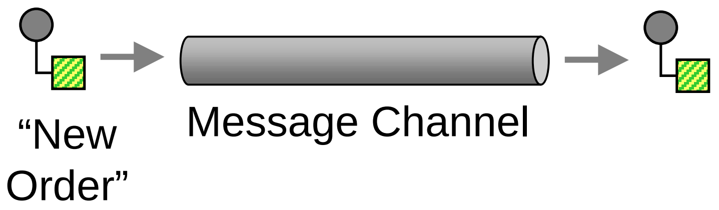
- Asynchronous, reliable communication
- Sender can send message even if receiver is not available
- Sender considers message delivered as soon as message is placed in channel
- The channel stores the message until the receiver is available
- Sender and receiver agree on a channel

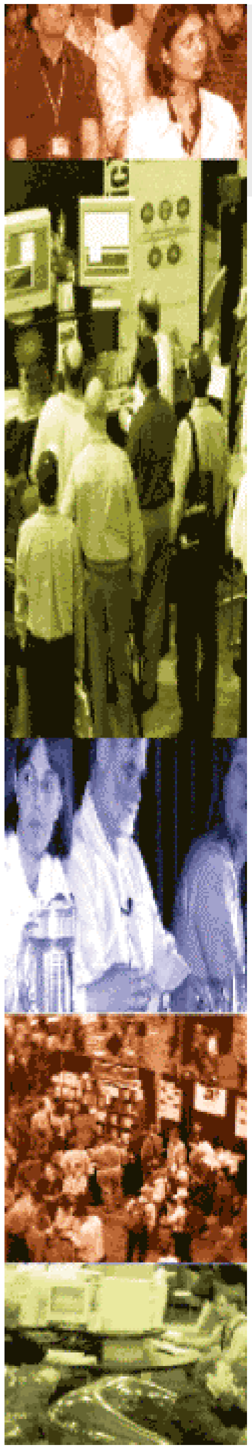




Channel Pattern: Point-to-Point Channel

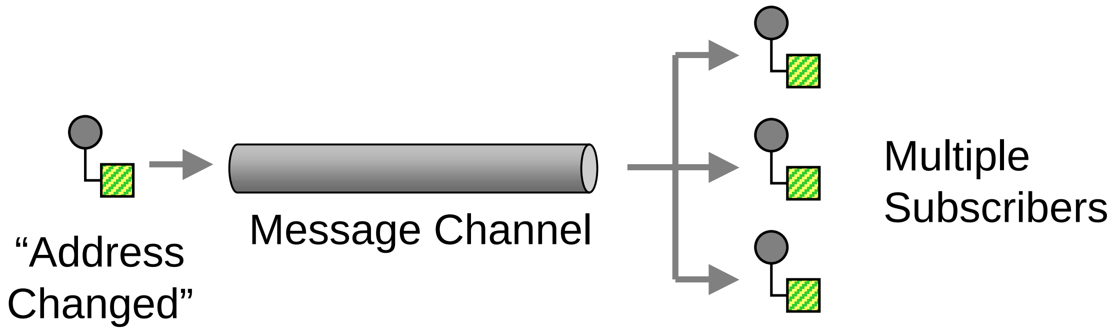
- How can the caller be sure that only one receiver will receive the document or perform the call ?
  - A single recipient for a each message
  - In case of multiple possible consumers, exactly one will receive the message (“competing consumers”)
  - Message Queue, Document / Command model
  - E.g., MSMQ, IBM WebSphere MQ, JMS Queue

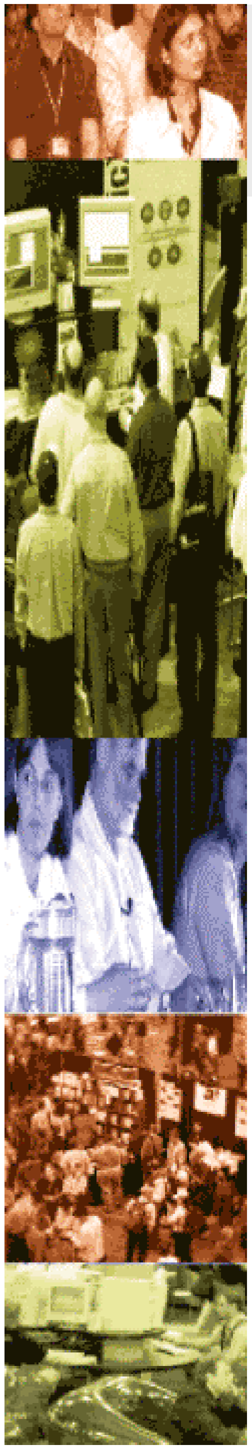




Channel Pattern: Publish-Subscribe Channel

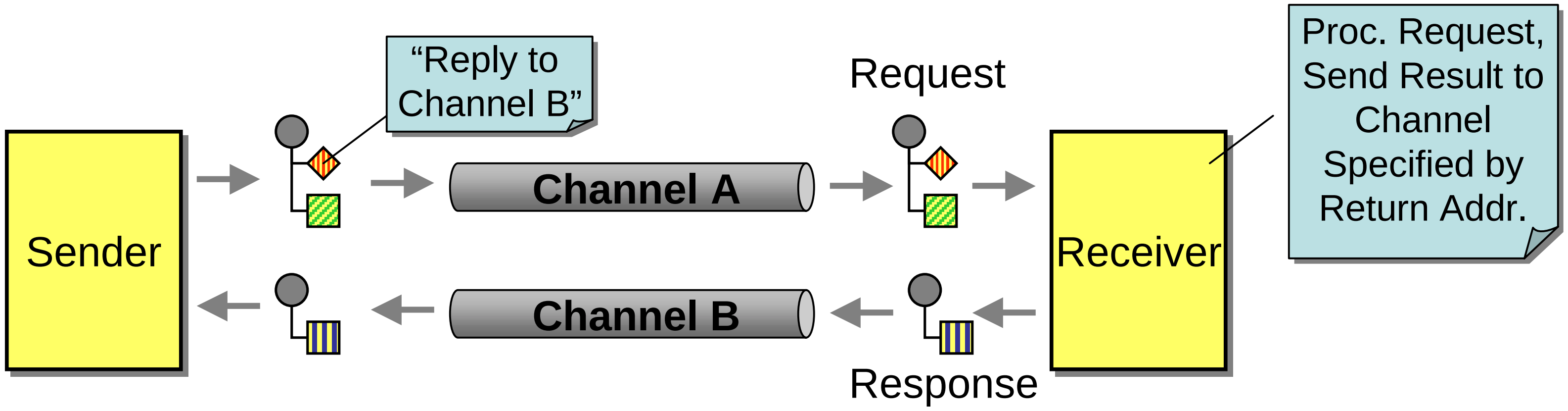
- How can the sender broadcast an event to all interested receivers?
  - Multiple recipients for a single message
  - Sender has no knowledge of recipients
  - Message is stored in the channel until each recipient consumed it
  - Broadcast / Multicast, Event model
  - E.g. TIBCO RendezVous, JMS Topic



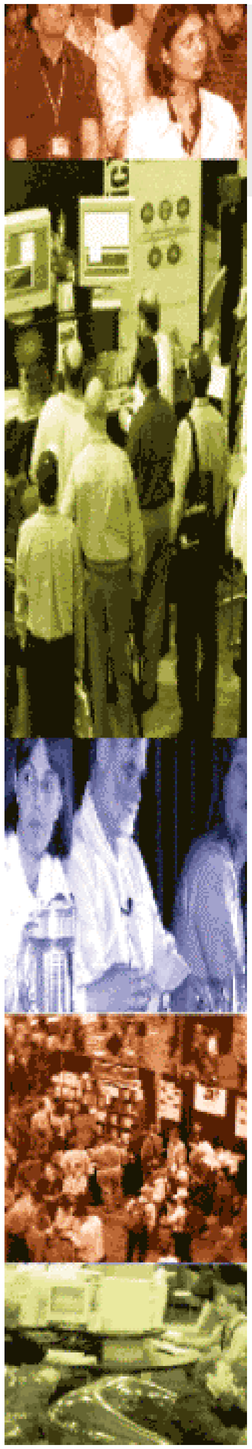


Message Pattern: Return Address

- How does the receiver of a message know where to send the reply message?
  – Loose coupling means the receiver may not know who the sender is.
  – Reply channel is usually different from request channel.

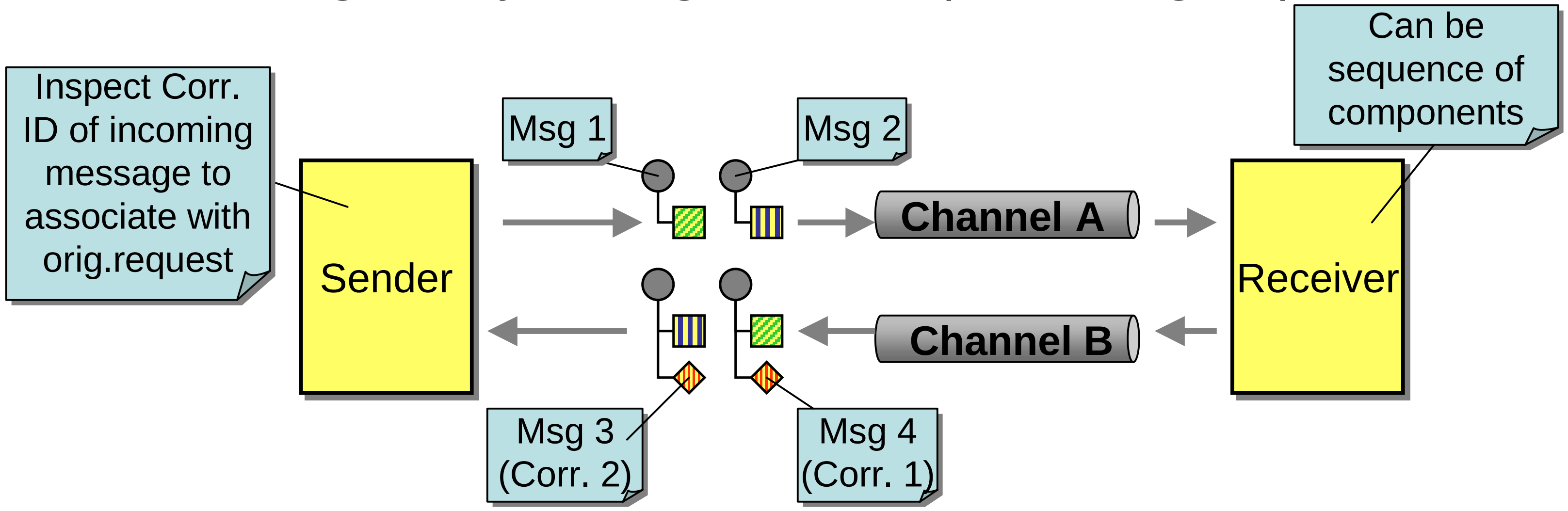


- Sender includes a *Return Address* in the request message. Receiver sends response message to channel specified by Return Address.



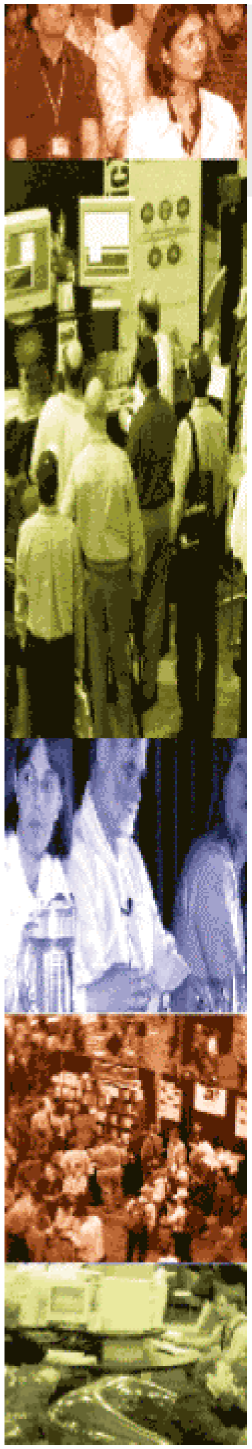
Message Pattern: Correlation Identifier

- How does a sender that receives a reply message know which request the reply belongs to?
  - Asynchronous messages may arrive out of order
  - Messages may undergo different processing steps



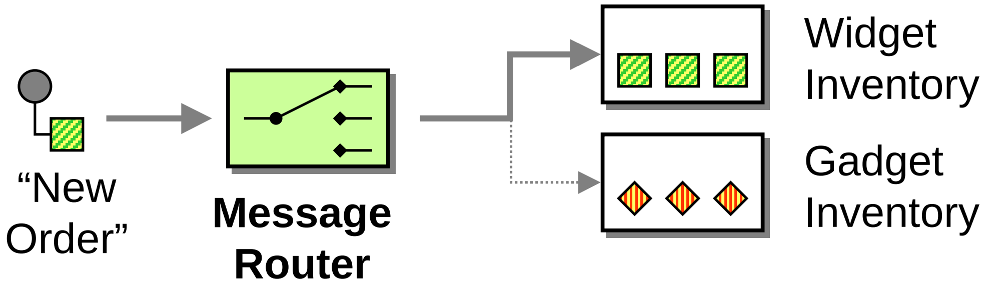
Inspect Corr. ID of incoming message to associate with orig.request
Msg 1
Msg 2
Can be sequence of components
Sender
Channel A
Receiver
Channel B
Msg 3 (Corr. 2)
Msg 4 (Corr. 1)

- Each reply message should contain a *Correlation Identifier*, a unique identifier that indicates which request message this reply is for.

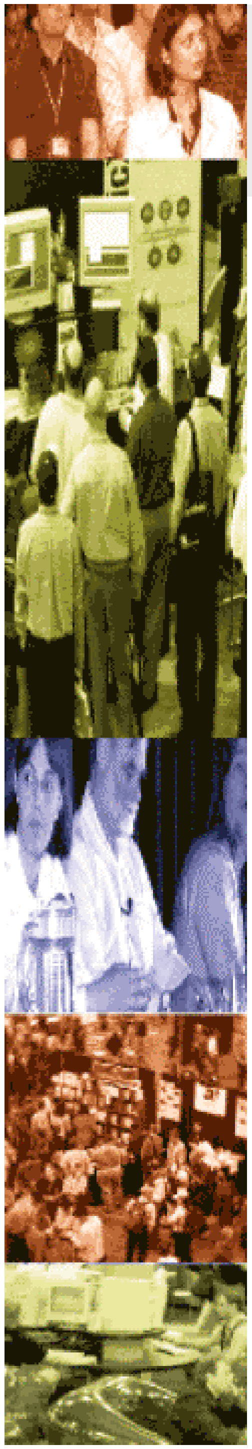


Routing Pattern: Message Router

- How can we decouple individual processing steps so that messages can be passed to different components depending on some conditions?
  - Different channels depending on message content, run-time environment (e.g. test vs. production), …
  - Do not want to burden sender with decision (decoupling)

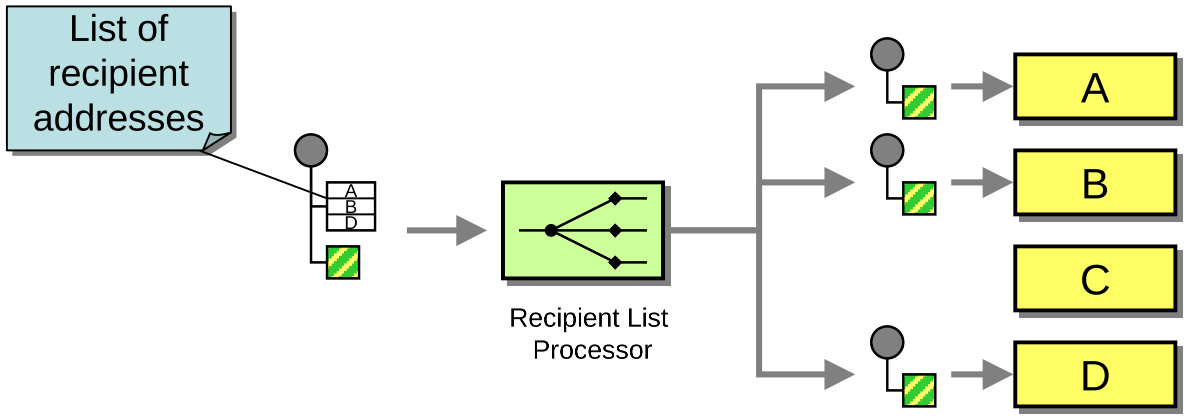


- Use a special component, a *Message Router*, to route messages from one channel to a different channel.

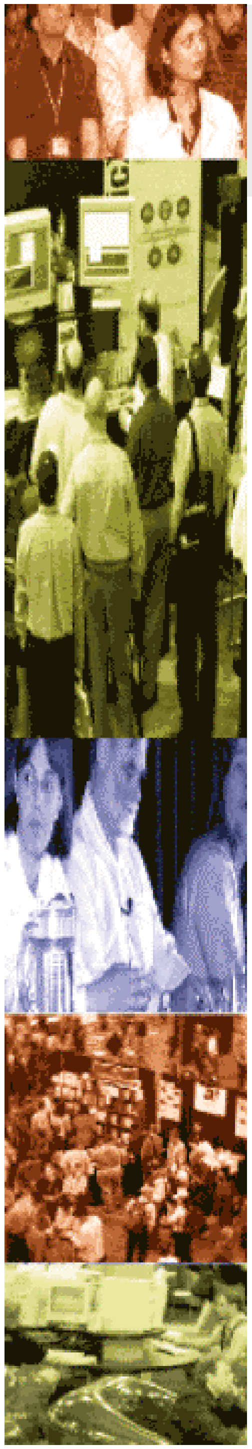


Routing Pattern: Recipient List

- How do we route a message to a dynamically specified list of recipients?
  – Want more control than Pub-Sub channel
  – Want to determine recipients by message

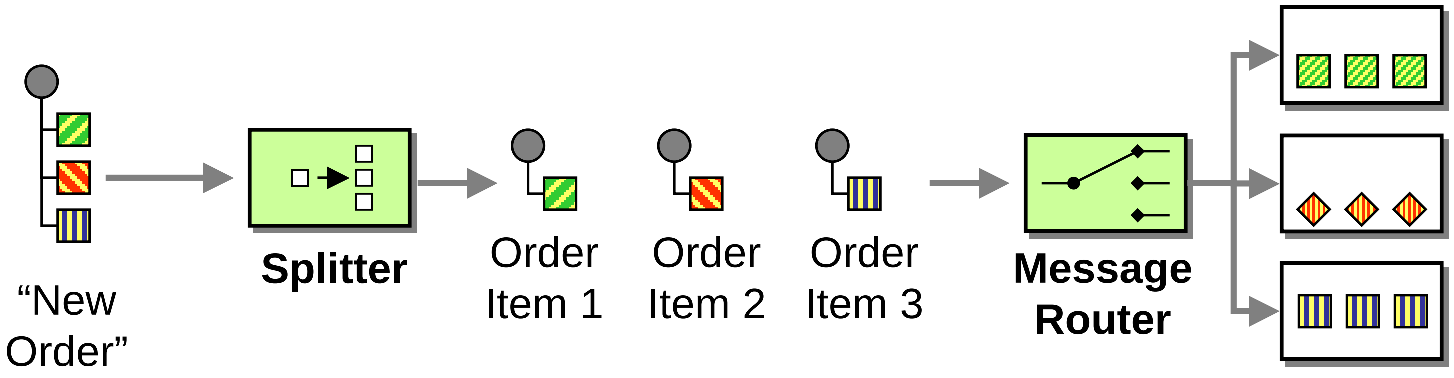


- Use a *Recipient List* to first compile a list of intended recipients and then deliver the message to every recipient in the list.
  – More control, (possibly) tighter coupling



Routing Pattern: Splitter

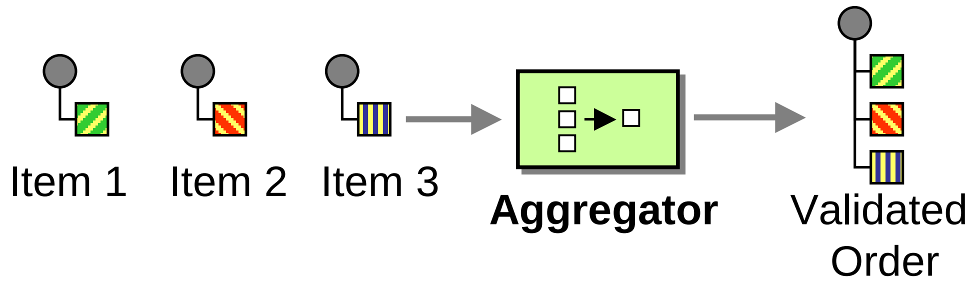
- How can we process a message if it contains multiple elements, each of which may have to be processed in a different way?
  - Treat each element independently
  - Need to avoid missing or duplicate elements
  - Make efficient use of network resources



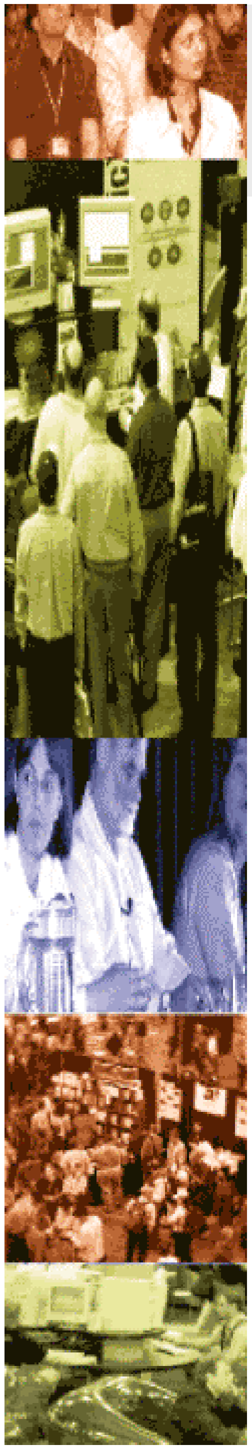
- Use a *Splitter* to break out the composite message into a series of individual messages, each containing data related to one item.

Routing Pattern: Aggregator

- How do we combine the results of individual, but related messages back into a single message?
  - Responses may be out of sequence
  - Responses may be delayed

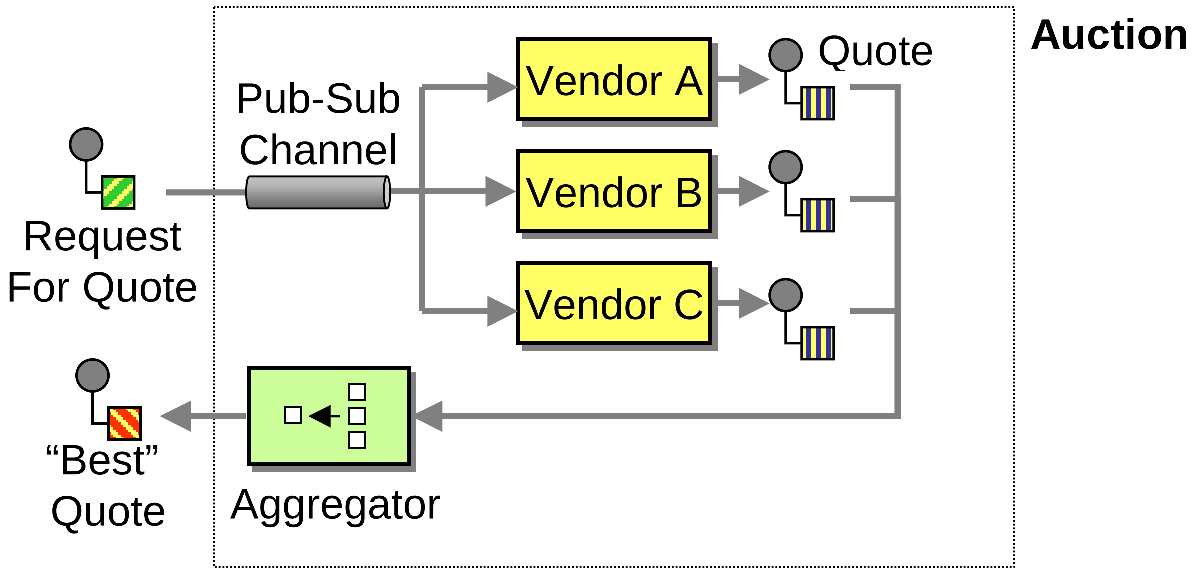


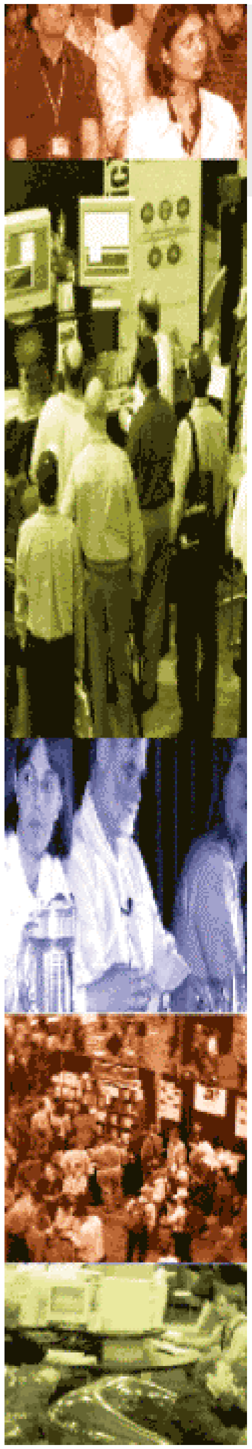
- An Aggregator manages the reconciliation of multiple, related messages into a single message
  - Stateful component
  - When do we send the aggregate message?
    - Wait for all responses - Take first best answer
    - Wait for amount of time - Wait until criteria met



Routing Pattern: Auction

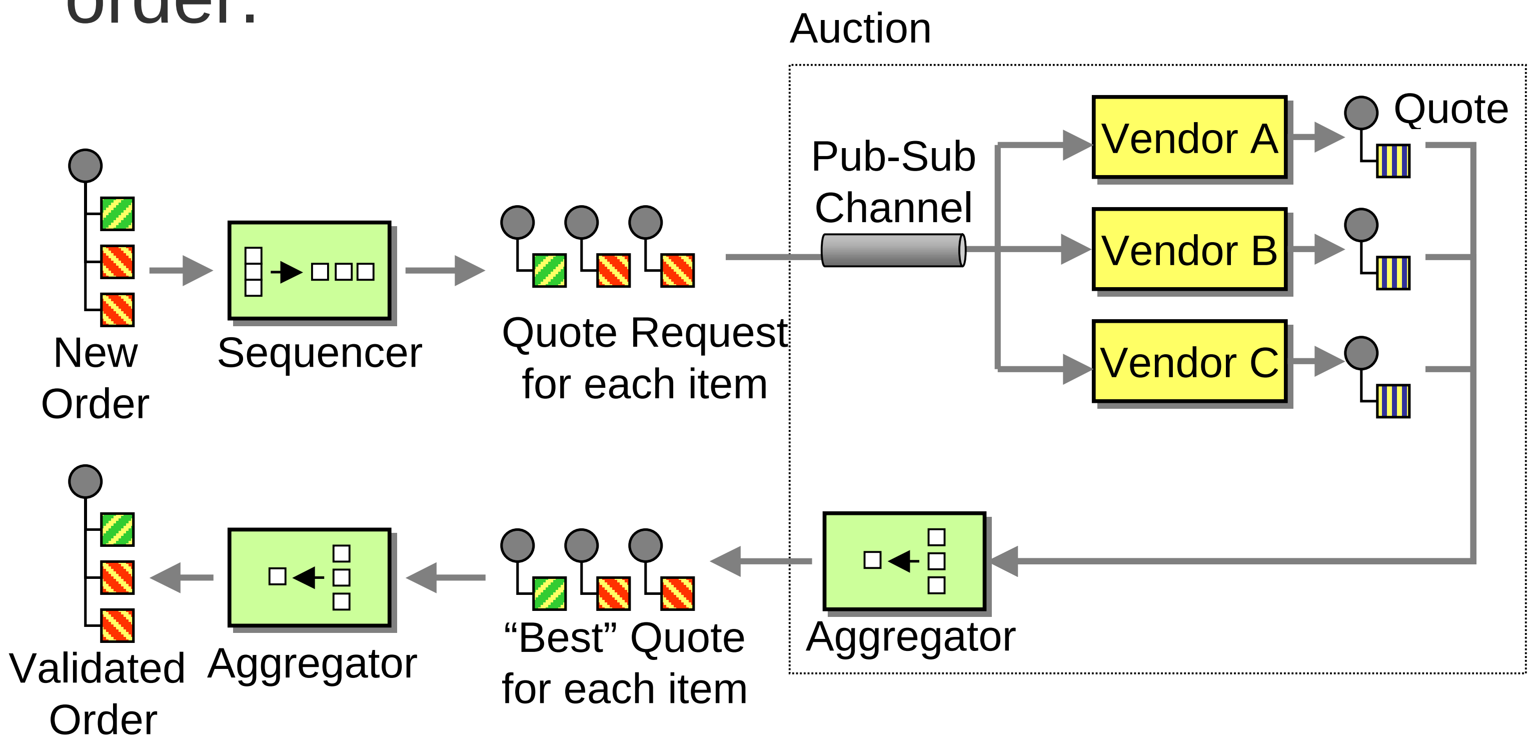
- We need to send a message to a dynamic set of recipients, and return a single message that incorporates the responses.

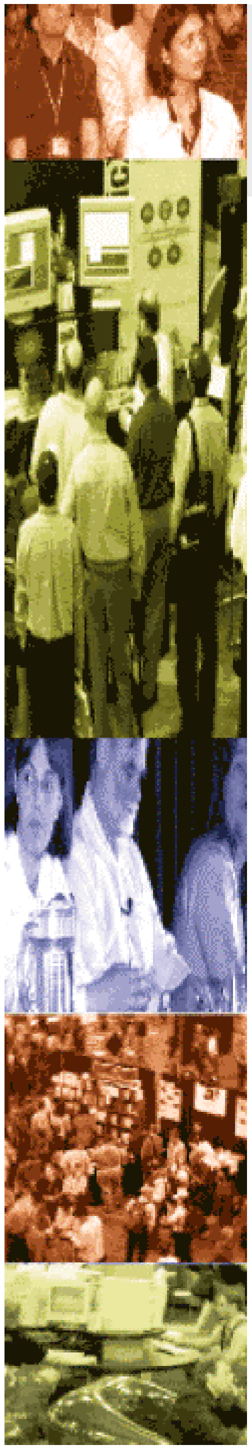




Example: Combining Routing Patterns

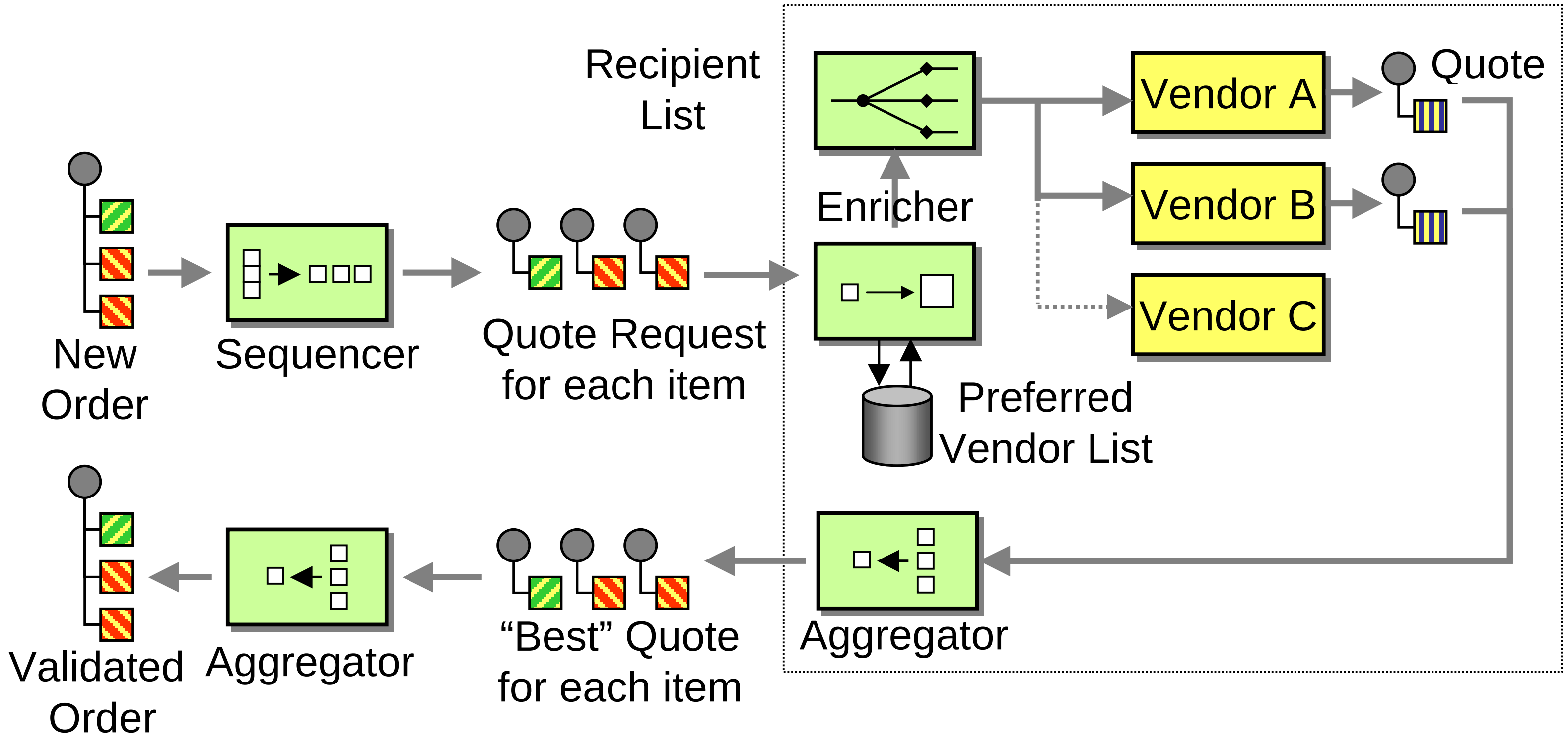
- Receive an order, get best offer for each item from vendors, combine into validated order.





Example Continued: Replace Auction with Recipient List

- Only vendors on the preferred vendor list get to bid on an item.



Transformation Patterns

Data Enricher

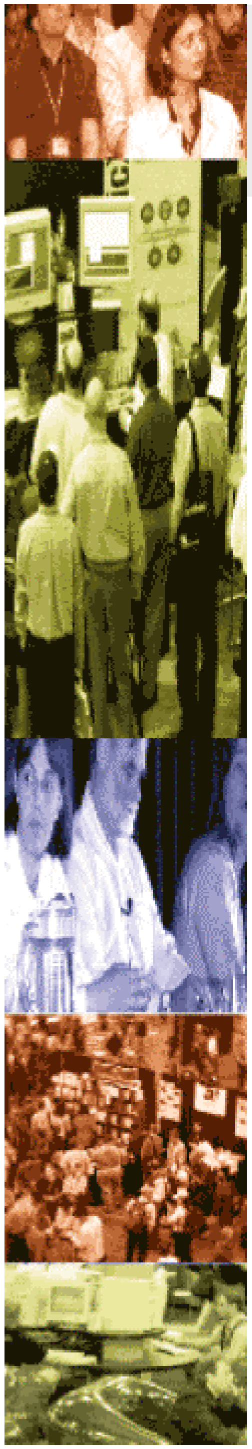
- How do we communicate with another system if the message originator does not have all the required data items available?

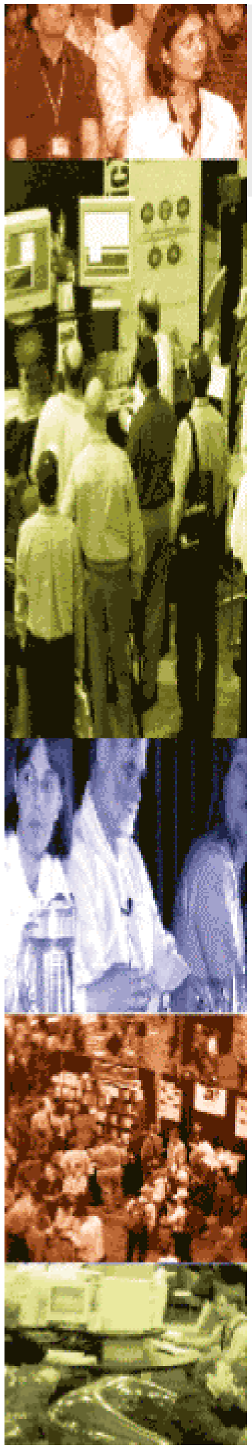
Content Filter

- How do we deal with a large message when we are interested only in a few data items?

Check Baggage

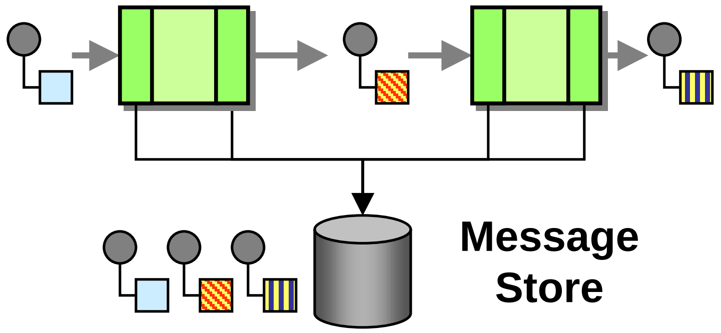
- How can we reduce the data volume of a message sent across the network without sacrificing information content?



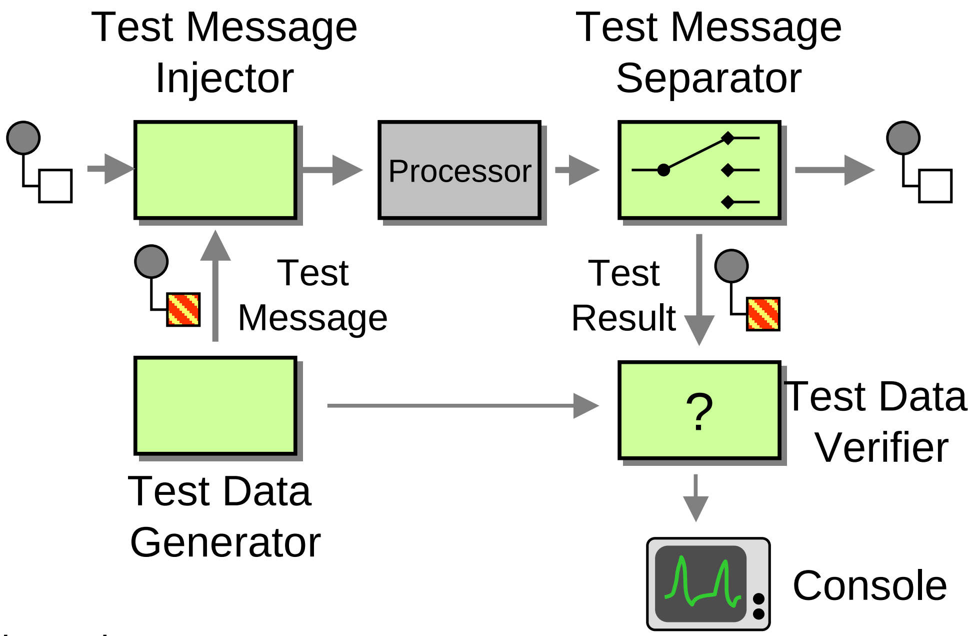


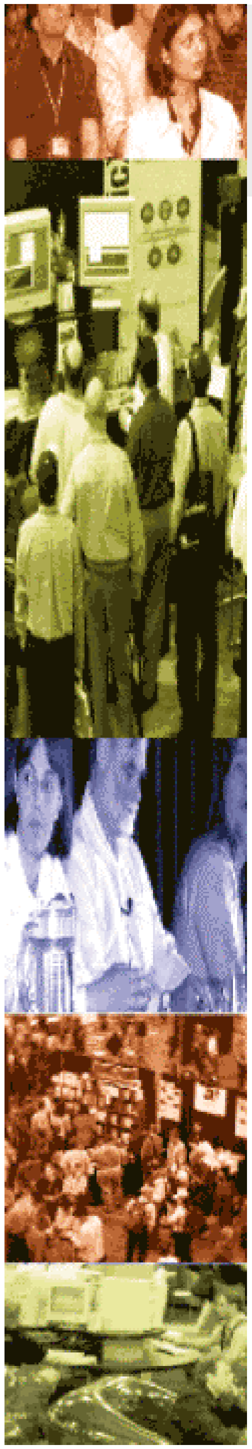
Management Patterns

- Message Store
  - How can we report against message information without disturbing the loosely coupled and transient nature of a messaging system?



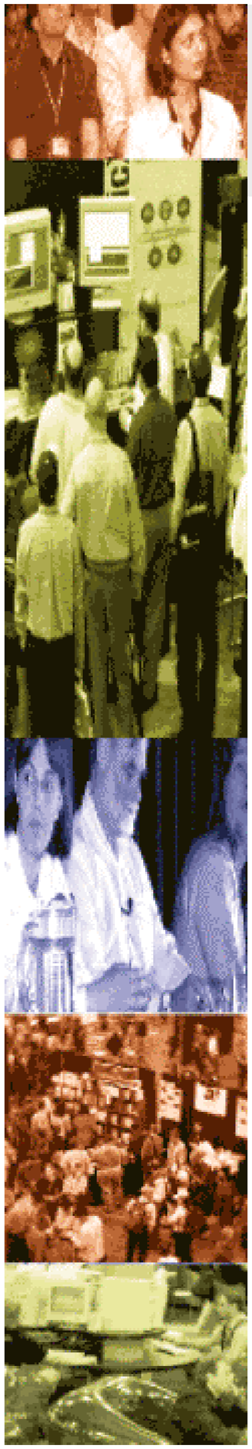
- Test Message
  - How can we detect a component that is actively processing messages, but garbles outgoing messages due to an internal fault?





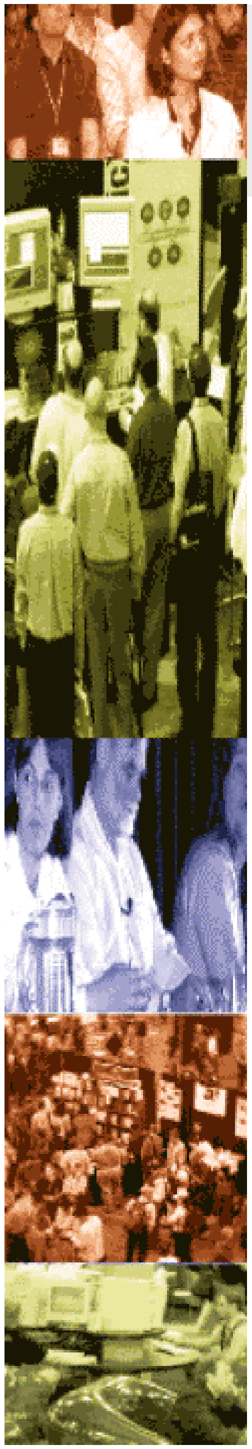
In Summary...

- We established a visual and verbal language to describe integration solutions
- Individual patterns can be combined to describe larger solutions
- We need no fancy tools besides PowerPoint (or paper and pencil)
- We stayed away from vendor jargon
- Each pattern describes trade-offs and considerations not included in this presentation



What Are These Patterns NOT?

- NOT a precise specification language
  – (e.g., see UML Profile for EAI)
- NOT a visual programming environment
- NOT a new “methodology”
- NOT complete
- NOT fool-proof
- NOT a silver bullet



If This Was Interesting…

- Gregor Hohpe, ghohpe@thoughtworks.com
- Visit www.eaipatterns.com:
  - Lots of detail
  - Bibliography, related papers
  - Sign up for the mailing list
- info@enterpriseintegrationpatterns.com
- Expect a book to hit the shelves sometime later this year: “Patterns of Enterprise Integration”
- More on patterns: www.hillside.net
- ThoughtWorks: www.thoughtworks.com

Thank You!