Enterprise Integration Patterns

Gregor Hohpe | www.eaipatterns.com
Topics for Today

1. Me
2. The Book
3. Enterprise Integration
4. Messaging
5. Messaging Patterns
6. Patterns and Pattern Languages Revisited
7. Messaging Patterns in Action
8. Conversations
9. Conversation Patterns
10. Conclusion
Me
Bounced around a lot

Diplom  Computer Science

Masters  Comp Science

Masters  Engineering Management

Startup

Consulting

Software

Corporate IT
Around the world in 20 years
The Book
PLoP 2002 Proceedings (Draft)

Note to authors: Please check the link to the paper and make sure that it contains your final revision. Any corrections should be sent to Weerasak Withawaskul at plop2002chair@yahoo.com.

Copyright 2002 by paper authors. Permission is granted only to copy for the PLoP 2002 conference.

Update: 9 Sep 2002 Mock Workshop Paper - Distributed Cache Pattern

Section 1 Accepted Papers

<table>
<thead>
<tr>
<th>#</th>
<th>Authors</th>
<th>Title</th>
<th>Shepherd</th>
<th>Program Committee</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>L. Araujo, M. Weiss</td>
<td>Linking Patterns and Non-Functional Requirements (was 'Using the NFR Framework for Representing Patterns')</td>
<td>Brian Marks</td>
<td>Eric Evans</td>
</tr>
<tr>
<td><strong>Group 2</strong></td>
<td>Leader: Martin Fowler and Ali Arsaniani</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>A. Arsaniani</td>
<td>Patterns for Implementing Grammar-Oriented Object Design</td>
<td>Masao Tomono</td>
<td>John Vlissides</td>
</tr>
<tr>
<td>2</td>
<td>A. Arsaniani</td>
<td>Towards a Pattern Language for Web Services Architecture (was 'Patterns for Web Services Architectures')</td>
<td>Gustavo Reis</td>
<td>John Vlissides</td>
</tr>
<tr>
<td>14</td>
<td>G. Hohpe</td>
<td>Enterprise Integration Patterns</td>
<td>Philip Eskenz</td>
<td>John Vlissides</td>
</tr>
<tr>
<td>4</td>
<td>A. Corea, D. C. Schmidt, R. Kiefstät, C. O’Ryan</td>
<td>Virtual Component A Design Pattern for Memory-Constrained Embedded Applications</td>
<td>Michael Kircher</td>
<td>Doug Schmidt</td>
</tr>
</tbody>
</table>

Section 2 Large Pattern Language Group Papers

<table>
<thead>
<tr>
<th>Group</th>
<th>Pattern Language</th>
<th>Leaders</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Patterns of System Integration with Enterprise Messaging</td>
<td>Bobby Woolf, Kyle Brown</td>
</tr>
<tr>
<td>2</td>
<td>Strategic Design (excerpt from Domain Driven Design) - Entire manuscript can be downloaded from here.</td>
<td>Eric Evans</td>
</tr>
<tr>
<td>3</td>
<td>Some Algorithm Structure and Support Patterns for Parallel Application Programs (abstract)</td>
<td>Berna Massingill, Timothy G. Mattson, Beverly A. Sanders</td>
</tr>
</tbody>
</table>

“Enterprise Integration Patterns”
G. Hohpe

“Patterns of System Integration with Enterprise Messaging”
B. Woolf, K. Brown
What are Enterprise Integration Patterns?

Very few business applications can live in isolation. More often than not, applications have to be integrated with other applications inside and outside the enterprise. This integration is usually achieved through the use of some form of "middleware". Middleware provides the "plumbing" such as data transport, data transformation, routing etc. Popular implementations of these concepts are found in EAI suites such as IBM MQ, TIBCO, SeeBeyond etc., as well as messaging specifications such as JMS or Web service standards like SOAP.

Architecting integration solutions is a complex task. There are many conflicting drivers and even more possible 'right' solutions. Whether the architecture was in fact a good choice usually is not known until many months or even years later, when inevitable changes and additions put the original architecture to test. There is no cookbook for enterprise integration solutions. Most integration vendors provide methodologies and best practices, but these instructions tend to be very much geared towards the vendor-provided tool set and often lack treatment of the bigger picture, including underlying guidelines and principles.

Therefore, we started to collect enterprise integration patterns, similar to the architecture and design patterns who have helped many application architects design robust applications over the past years. The patterns on this site have been harvested from multiple years of hands-on enterprise integration work with a variety of organizations. Still, the effort has just begun and is quite incomplete.

Who can use Enterprise Integration Patterns?

The patterns presented on this site help integration architects and developers design and implement integration solutions more rapidly and reliably. Most of the patterns assume a basic familiarity with publish-subscribe messaging architectures. However, the patterns are not tied to a specific implementation. Most patterns apply to EAI suites as well as Web Services or JMS-based applications. In some cases, a pattern may already be embedded in the middleware package. This is a sign that the vendor recognized the recurring problem and incorporated the solution into the package. We still present these patterns for two reasons. First, not all packages implement the same patterns, so a user workign with another package will still find the pattern useful. Second, despite the default implementation of the pattern in the middleware package, a description of the forces and alternatives is insightful for any architect or developer who is interested in EAI concepts beyond the specific package implementation.
OOPSLA 2003

- 185,000 Words
- 730 pages
- 65,000 copies sold

Languages

- English
- Russian
- Chinese Traditional
- Korean

www.eaipatterns.com

- Sketches, summaries under Creative Commons
- Visio, Omnigraffle stencils
Software Patterns

- Buschman, *Pattern-Oriented Software Architecture*
- Dyson, *Architecting Enterprise Solutions*
- Fowler, *Patterns of Enterprise Application Architecture*
- Gamma et al, *Design Patterns*
- Hohpe et al, *Enterprise Integration Patterns*
- Kircher, *Pattern-Oriented Software Architecture*
- Schmidt, *Pattern-Oriented Software Architecture*
- New/Garland Software Architecture
- Cambridge et al, *Integration Patterns*
Enterprise Integration Patterns in Java using a DSL via Apache Camel

For those of you who missed me rambling about this at JavaOne I thought I'd introduce Camel to you.

Apache Camel is a powerful rule based routing and mediation engine which provides a POJO based implementation of the Enterprise Integration Patterns using an extremely powerful fluent API (or declarative Java Domain Specific Language) to configure routing and mediation rules.

The Domain Specific Language means that Apache Camel can support type-safe smart completion of routing and mediation rules in your IDE using regular Java code without huge amounts of XML configuration files; though Xml Configuration inside of Spring 2 is also supported.

A good way to get started is to take a look at the Enterprise Integration Patterns catalog and see what the Java code of an example looks like. For example, try the message filter, content based router or splitter.
Enterprise Integration
Isolated Systems

Unified Access
Why This Is Still Interesting

• Large-scale and complex
• Far-reaching implications, business critical
• Distributed, heterogeneous environment
• Applications not designed to be connected
• Semantic Dissonance
• Not object-oriented
• Variety of skills and technologies
• Corporate politics

Plus
• Distributed applications are the norm
• Increased customer expectations
• REST services, simpler protocols
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

All applications access a common database

Pros:
- Consistent Data
- Reporting
- Transactional guarantees

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
Messaging
Asynchronous Messaging Style

Systems send messages across Channels
Channels have logical (location-indep.) addresses
Placing a message into the Channel is quick (“fire-and-forget”)
The Channel queues messages until the receiving application is ready

An "honest" architectural style that does not try to deny the limitations of the underlying medium.
Why Asynchronous Messaging?

Asynchrony
- Sender does not have to wait for receiver to process message
- Temporal decoupling

Throttling
- Receiver can consume messages at its own pace
- Processing units can be tuned independently

Can be Reliable Over Unreliable Networks
- Messages can transparently be re-sent until delivered
- Think cell phones – intermittent and unreliable

Insertion of intermediaries (Pipes-and-Filters)
- Composability
- Transformation, routing etc.

Throughput over latency
- “Wider bridges not faster cars”
A New “Tower of Babel”

Gartner “Magic Quadrant” for Integration and Middleware 2001
Messaging Patterns
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
Messaging Pattern Language

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
Visual Language

- Content-Based Router
- Message Filter
- Recipient List
- Splitter
- Aggregator
- Resequencer
- Routing Slip (Itinerary)
- Process Manager
Composing Patterns

Receive an order
Get best offer for each item from vendors
Combine into validated order.

Scatter-Gather

Splitter

New Order

Quote Request for each item

Vendor A
Vendor B
Vendor C

Pub-Sub Channel

Quote

Aggregator

Validated Order

“Best” Quote for each item
Patterns & Pattern Languages
Patterns Revisited

• Shows a good solution to a common problem within a specific context
• “Mind sized” chunks of information (Ward Cunningham)
• Expresses intent (the “why” vs. the “how”)
• Observed from actual experience

NOT:
• A firm rule – always a time when not to use
• Copy-paste code snippet – just example
• Isolated – Part of a Pattern Language
Patterns and Architecture Styles

Patterns exist at different levels

- Idioms (usually language specific)
- Design (usually system specific)
- Architecture

Patterns “belong” to an architectural style

- OO Patterns ≠ Messaging Patterns
- Architectural style provides vocabulary to express patterns
- Different vocabulary, composition rules, semantic interpretation

Integration uses a variety of architectural styles

- Messaging (pipes-and-filters), Data transformation (functional), endpoints (object-oriented), conversations (state machine)
BED ALCOVE

Design problem
Bedrooms make no sense.

Forces
First, the bed in a bedroom creates awkward spaces around it: dressing, working, watching television, sitting, are all rather foreign to the side spaces left over around a bed. (...
Second, the bed itself seems more comfortable in a space that is adjusted to it.

Solution
Don't put single beds in empty rooms called bedrooms, but instead put individual bed alcoves off rooms with other nonsleeping functions, so the bed itself becomes a tiny private haven.

Related Patterns
Communal Sleeping, Marriage Bed Ceiling Height Variety, Half-open Room, Thick Walls
Pattern Structure

Name

Icon

Context

Problem

Forces

Sketch

Solution

Results

Next

Examples

Aggregator

A **Splitter** is useful to break out a single message into a sequence of sub-messages that can be processed individually. Likewise, a **Recipient List** or a **Publish-Subscribe Channel** is useful to forward a request message to multiple recipients in parallel in order to get multiple responses to choose from. In most of these scenarios, the further processing depends on successful processing of the sub-messages. For example, we want to select the best bid from a number of vendor responses or we want to bill the client for an order after all items have been pulled from the warehouse.

**How do we combine the results of individual, but related messages so that they can be processed as a whole?**

Use a stateful filter, an **Aggregator**, to collect and store individual messages until a complete set of related messages has been received. Then, the **Aggregator** publishes a single message distilled from the individual messages.

The **Aggregator** is a special **Filter** that receives a stream of messages and identifies messages that are correlated. Once a complete set of messages has been received (more on how to decide when a set is 'complete' below), the **Aggregator** collects information from each correlated message and publishes a single, aggregated message to the output channel for further processing.
Pattern Language

Patterns don’t live in isolation
• Pattern Compounds
• Pattern Sequences
• Pattern Collections
• Pattern Languages

Patterns are “harvested”
• Story behind the scenes for GoF
• How patterns are refined and applied
Pattern Language: Alternatives

Simple

- Process one msg at a time (stateless)
  - Single msg out
    - Exactly One
    - Zero or One
  - Mult. msgs out
    - Parallel
    - Sequential

Composed

- Process multiple msgs at a time (stateful)
  - Less msgs out
  - Same number of msgs out
  - Split Message
    - Parallel
    - Broadcast Message
    - Predetermined, Linear
  - Any Path
  - Parallel
  - Sequential
  - Any Path

Content-Based Router
Message Filter
Recipient List
Splitter
Aggregator
Resequencer
Compos. Msg. Processor
Scatter-Gather
Routing Slip
Process Manager

© 2015 Gregor Hohpe
Pattern “Sketches”: The Icons / Gregorgrams

• Biggest step was having a “box in the middle”

  ![Diagram 1]

• Pipes-and-filters = Simplest form of Composability

  ![Diagram 2]

• Some icons missing

  ![Diagram 3]

• Endpoint patterns compose differently

  ![Diagram 4]
Pattern “Sketches”: Enriching the Vocabulary

Synchronous
- Polling Consumer

Asynchronous
- Event-Driven Consumer

Endpoints

Connecting Elements
- Puller
- Pusher
- Pool / Buffer
- Driver

Source ➔ Buffer ➔ Sink

http://www.EnterpriseIntegrationPatterns/ramblings/80_syncorswim.html
Fun with Pattern Icons
Richer Pattern Relationships

Source: Logica
Richer Pattern Relationships

MESSAGE ENDPOINTS

Source: Logica
Patterns Hands-on
Messaging Patterns in Action
Pattern: Request-Reply

Service Provider and Consumer (similar to RPC)
Channels are unidirectional
Two asynchronous \textit{Point-To-Point Channels}
Separate request and reply messages
Multiple Consumers

Each consumer has its own reply queue

How does the provider know where to send the reply?

- Could send to all consumers → very inefficient
- Hard code → violates principle of context-free service
Pattern: *Return Address*

Consumer specifies *Return Address* (reply channel) in the request message

Service provider sends reply message to specified channel
Multiple Service Providers

Request message can be consumed by more than one service provider

*Point-to-Point Channel* supports *Competing Consumers*, where only one service receives each request message

Channel queues up pending requests
Multiple Service Providers

Reply messages get out of sequence

How to match request and reply messages?

- Only send one request at a time → very inefficient
- Rely on natural order → bad assumption
Pattern: *Correlation Identifier*

**Equip each message with a unique *Correlation Identifier***

- Message ID (simple, but has limitations)
- GUID (Globally Unique ID)
- Business key (e.g. Order ID)

Provider copies the ID to the reply message

Consumer can match request and response

Insert a *SmartProxy* if provider does not support this
Pattern: *Pipes-And-Filters*

Connect individual processing steps (filters) with message channels (pipes)

- Pipes decouple sender and receiver
- Participants are unaware of intermediaries
- Compose patterns into larger solutions
Multiple Specialized Providers

Each provider can only handle specific type of message
Route request to the “appropriate” provider based on the content of the request message

- Do not want to burden sender with decision (decoupling)
- Letting each consumer “pick out” desired messages requires distributed coordination
Pattern: *Content-Based Router*

Insert a *Content-Based Router*

Message routers forward incoming messages to different output channels without changing message content. Mostly stateless, but can be stateful (e.g. de-duper)
Composite Message

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
Pattern: *Splitter*

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

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

Then use a **Content-Based Router** to route the individual messages to the proper destination.
Producing a Single Response

How to combine the results of individual, but related messages so that they can be processed as a whole?

- Messages out of order
- Message delayed
- Which messages are related?
- Avoid separate channel for each system
Use a stateful filter, an *Aggregator*, to collect and store individual messages until a complete set of related messages has been received.

- Aggregator publishes a single message distilled from the individual messages.
**Aggregator Design Decisions**

**Correlation:** Which incoming messages belong together?

**Completeness Condition:** When to publish the result message?
- Wait for all
- Time out (absolute, incremental)
- First best

**Aggregation Algorithm:** How to combine the received messages?
- Single best answer
- Condense data (e.g., average)
- Time box with override
- External event
- Concatenate data for later analysis
Pattern: *Scatter-Gather*

Send a message to a dynamic set of recipients, and return a single message that incorporates the responses.
Composing Patterns

Receive an order, get best offer for each item from vendors, combine into validated order.
Pattern: Control Bus

Application Message Flow

Configuration
Heartbeat
Test messages
Exceptions / logging
Statistics / Quality-of-Service (QoS)
Live console
Pattern: Test Message

Inject application specific test messages
Extract result from regular message flow
Compare result against predefine (computed) result
Messaging Patterns Today
Google Cloud Pub-Sub

Publisher B
Message 2
Topic B
Subscription B
Subscriber B1
Subscriber B2

Publisher C
Message 3
Topic C
Subscription YC
Subscriber Y
Subscription ZC
Subscriber Z

Cloud Pub/Sub
Publish-Subscribe Channel
Polling Consumer

Competing Consumers
Message 2
Message Expiration

Message 3
Message 3

Durable Subscriber

Point-to-Point
Transactional Client

Publish-Subscribe
13. Architectural Patterns
13.1 Asynchronous Messaging
13.2 Big Ball of Mud
13.3 Command and Query Responsibility Segregation (CQRS)
13.4 Event-Driven Architecture
13.5 Orchestrated Workflow
13.6 Pipes and Filters

14. Microservice Roles
14.1 Message Originator
14.2 Content Enricher
14.3 Event Mediator
14.4 Event Processor
14.5 Coexistent Versions
14.6 Fanout
14.7 Async Waterfall (with optional Fanout)
14.8 Need Solution
14.9 Transformer
14.10 Worker

15. Integration Styles
15.1 File Transfer
15.2 Shared Database
15.3 Remote Procedure Invocation
15.4 Messaging

16. Messaging Systems
16.1 Message Channel
16.2 Message
16.3 Message Router
16.4 Message Translator
16.5 Message Endpoint
Extending Messaging Patterns
Expanding the Integration Patterns

Pattern Family

Deepen

Pattern

Broaden

Project

Platform Tools

Other Patterns

?
Patterns as Domain Language

- Messaging toolkit
- Compose solutions from the command line
- Raised level of abstraction

```
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
```
Patterns

• Human communication
• Fuzzy
• Design tool
• Platform independent

Components

• System Communication
• Precise
• Executable
• Platform dependent

- Simple composability: Pipes and Filters

- Easy formalization: Input ports, Output ports

- Other domain languages: XSLT, XPath
## Improving Projection – Variability Points

### Aggregator

<table>
<thead>
<tr>
<th>Element ID</th>
<th>Input Channel</th>
<th>Output Channel</th>
<th>Correlation Function</th>
<th>Completeness Condition</th>
<th>Aggregation Algorithm</th>
</tr>
</thead>
</table>
Conversations
Request-Reply

Simplest conversation
Single Conversation state: waiting for reply, complete
Gets more complicated once error conditions considered
Request-Reply with Retry

Sender can repeat request $n$ times
Provider has to be idempotent
Receiver also has to be idempotent
Example: RosettaNet Implementation Framework (RNIF)
Enterprise Integration or Messaging Patterns?

Enterprise Integration

Messaging
Enterprise Integration or Messaging Patterns?

Enterprise Integration

- Messaging
- Conversations
- Processes
- Events

© 2015 Gregor Hohpe
www.EnterpriseIntegrationPatterns.com
Messaging

Flow of messages through processing nodes

- Stateless -> scaleable, decoupled
- Error handling?
- Complex interactions (no guarantees)
Conversations

- Each conversation corresponds to one process instance
- Each participant has a (potentially different) process definition
Example: Subscriptions

Publish-Subscribe Channel

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

- Follows the message
- Multiple receivers
- One-way
- Deals with transport issues

Subscribe-Notify

How can one participant receive information from another participant if that information cannot easily be packaged into a single message?

- Follows time
- Single receiver
- Two-way
- Deals with state / resources
Conversation Patterns
Challenges: Describing Conversations

• Sequence Diagrams (UML 1.x) only show one instance, not the rules of interaction
• Sequence Diagrams (UML 2.0) more powerful, but non-intuitive notation
• WS-CDL pretty much died.
• WS-BPEL too verbose and technical, looking from participant perspective
• Temporal Logic expressive, but not good for sketch
• BPMN probably best choice, but tough to see the essence.

Dynamic views are much tougher for the brain to process as it requires a translation from a static image to a dynamic process.
Conversation Sketches

- Prefer a sketch with loose semantics that highlights the essence
- Use BPMN as implementation example
De-Junking the Notation

Focus on Actions
Sequence Numbers

Focus on messages
Named participants
Top-down timeline
Simpler graphics
Conversation Pattern Language

**Setting Up**

- Discovery

**Participants**

- Basic Conversations

**Application-level**

- Resource Management

**Initiation**

- Intermediaries

**Ensuring Consistency**
Conversation Pattern Language

**Discovery**
- Dynamic Discovery
- Advertise Availability
- Consult Directory
- Referral
- Leader Election

**Basic Conversations**
- Fire-and-Forget
- Asynchronous Req-Resp
- Req-Resp with Retry
- Polling
- Subscribe-Notify
- Quick Acknowledgment

**Resource Management**
- Incremental State
- Lease
- Renewal reminder

**Initiation**
- Three-way Handshake
- Acquire Token First
- Rotate Tokens
- Verify Identity
- User Grants Access

**Intermediaries**
- Proxy
- Relay
- Load Balancer
- Scatter Gather

**Ensuring Consistency**
- Ignore Error
- Compensating Action
- Tentative Operation
- Coordinated Agreement
How can a conversation initiator find a partner when it has no knowledge whatsoever about available partners?

Point-to-point communication requires knowledge of the conversation partner (or channel).
Late binding between a participants lowers the location coupling.
Discovery may be on the critical path to establishing a conversation.
Even in the presence of a central lookup service, a new participant has to first establish a connection to the lookup service.
Dynamic Discovery

1. Broadcast *Lookup* request
2. Interested providers send *Available* responses
3. Requestor initiates interaction with chosen provider

Examples: DHCP, TIBCO Repository discovery
How can a participant let others know that it is available?

Central services for discovery are bound to get out of sync with reality. Centralized administration may result in a single point of failure. Dynamic Discovery can flood the network with requests. The number of available providers is often small compared to the number of initiated conversations.
Advertise Availability

Directory may store additional metadata about the service

"Match making based on"

Unique Identifiers

Interface Definition / Type

Attributes

Keyword match
How can a conversation initiator find a partner across a large network without flooding the network with requests?

Late binding between participants lowers the location coupling. Many networks do not route broadcast packets beyond the local network. Often centralized administration is involved in setting up a new service.
Directory may store additional metadata about the service
"Match making based on"

*Unique Identifiers, Interface Definition / Type, Attributes*

Example: UDDI Directory, DNS
The choice of conversation partner may depend on the context of a conversation or may change over time. How can an initiator discover the right conversation partner?

A participant may be required to interact with the same partner that another participant is already interacting with. Directories are generally context free, i.e. they do not keep track of existing conversations and when assigning an initiator to a partner. Some participants may not want to be "discovered". However, "friends of friends" are allowed to interact with them.
Consult Directory is a specialized case of Referral
Requires *addressability*, i.e. to embed addresses in messages

Example: HTTP 302
Multi-Party Conversations: Intermediaries

Peer-to-peer

Intermediaries

Coordinators

Connectors
Proxy

How can a participant communicate with a partner that is not visible or not reachable?

Initiator can hide identity using a *Proxy*

Proxy can monitor conversations

Proxy may need to be stateful for two-way conversations

Proxy can become a bottleneck
Relay

How can participants engage in a two-way communication when each participant is limited to outbound requests?

High overhead when using *Polling*

All other conversations can be layer on top of *Relay*

Needs to be stateful

Example: Amazon SQS
Scatter-Gather (Aggregator)

How can a participant solicit responses from a number of participants without connecting to all of them

Widespread business model, e.g. “Aggregators”
Resource Management

Automatic Expiration

“Lease” model
Heartbeat / keep-alive
Subscriber has to renew actively
Example: Jini

Renewal Request

“Magazine Model”
Subscriber can be simple
Provider has to manage state for each subscriber
REST Conversations

- Simpler transport protocols are more likely to hold conversations
- Loose coupling generates conversations: discovery, negotiation
- HTTP has built-in conversation patterns, e.g. 302

201 Created
Location: http://starbucks.example.org/order/1234
Content-Type: application/xml

<order xmlns="http://starbucks.example.org">
  <drink>latte</drink>
  <cost>3.0</cost>
  <next xmlns="http://example.org/state-machine"
       rel="http://starbucks.example.org/payment"
       uri="http://starbucks.example.org/payment/order/1234"
       type="application/xml"/>
</order>

Pautasso et al:
Modeling RESTful Conversations with Extended BPMN Choreography Diagrams
Conclusion
Conclusions

• Enterprise Integrations is more than messaging
• Enterprise Integration needs multiple pattern languages
• Good patterns languages are timeless, but difficult to make
• A good notation is a critical element of a pattern language
• Follow evolution of conversation patterns

@ghohpe, #eaipatterns

eaipatterns.com
eaipatterns.com/patterns/conversation