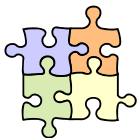


# Architetture Software

## Messaging (stile architetturale) e integrazione di applicazioni

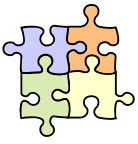
Dispensa ASW 430

ottobre 2011



### - Fonti

- [POSA4] Pattern-Oriented Software Architecture – A Pattern Language for Distributed Computing
- [Hohpe&Woolf 2004] Enterprise Integration Patterns
  - <http://www.enterpriseintegrationpatterns.com/>
  - <http://eaipatterns.com/>



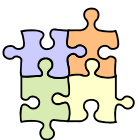
## - Obiettivi e argomenti

### □ Obiettivi

- presentare il messaging – qui inteso come stile architetturale
- discutere (ed esemplificare) l'applicazione del messaging all'integrazione di applicazioni

### □ Argomenti

- introduzione
- messaging [POSA4]
- altri pattern per il messaging [POSA4]
- messaging per l'integrazione di applicazioni



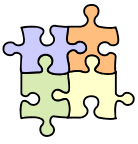
## \* Introduzione

### □ Le architetture client/server e le architetture a oggetti distribuiti sono basate su un paradigma di interazione richiesta-risposta

- estensione a un contesto distribuito della chiamata di procedure/invocazione di metodi – meccanismo fondamentale della programmazione imperativa
- comunicazione uno-a-uno
- basato su interfacce procedurali e una tipizzazione forte
- accoppiamento forte dei client nei confronti dei server

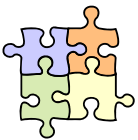
### □ In alcuni casi è preferibile un paradigma di interazione diverso

- basato su scambio di messaggi/documenti (per quanto possibile auto-descrittivi) e invocazione implicita
- comunicazione multi-a-uno o uno-a-molti
- accoppiamento debole tra le parti coinvolte



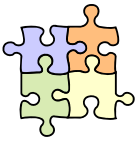
## Middleware message-oriented

- Middleware message-oriented (MOM)
  - famiglia di middleware basata sullo scambio asincrono di messaggi – e non su protocolli sincroni di richiesta/risposta
  - sostengono un accoppiamento debole tra componenti
  - possono offrire elevata flessibilità e affidabilità
  - numerose implementazioni, sia “centralizzate” (ad es., JMS in Java EE) che “distribuite” (ad es., Tibco)
  - si veda la dispensa sul Messaging (middleware)
  - questa modalità di interazione
    - è disponibile anche nelle tecnologie a componenti
    - può essere utilizzata nell’integrazione di applicazioni
    - è un ingrediente essenziale anche nelle architetture orientate ai servizi



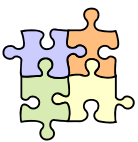
## Messaging e Publisher-Subscriber

- [POSA4] distingue due stili architetturali fondamentali per la comunicazione asincrona
  - messaging
    - comunicazione basata sullo scambio di messaggi
    - accoppiamento debole tra produttori e consumatori di messaggi
    - comunicazione multi-a-uno – “code”
  - publisher-subscriber
    - comunicazione basata sulla notifica di eventi
    - accoppiamento ancora più debole
    - comunicazione uno-a-molti – “topic”
  - ci concentriamo soprattutto sul messaging – è possibile considerare publisher-subscriber una sua variante (o uno stile “sinergico”)



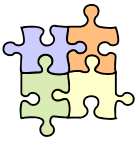
## Messaging e strumenti di middleware

- Strumenti di middleware e modalità di comunicazione
  - gli strumenti MOM offrono normalmente entrambe le modalità di comunicazione asincrona (messaging e publisher-subscriber)
  - le tecnologie a componenti offrono sia meccanismi di comunicazione sincroni (basati su interfacce procedurali) che asincroni (basati sullo scambio di messaggi/documenti/eventi)
  - anche la tecnologia dei Web Services consente sia la modalità di comunicazione sincrona che quella asincrona



## \* Messaging [POSA4]

- Il pattern architetturale **Messaging**
  - definisce un'infrastruttura di comunicazione – per sostenere l'integrazione di componenti sviluppati indipendentemente in un sistema coerente
  - la comunicazione è basata sullo scambio asincrono di messaggi (o documenti) tra i vari componenti
    - non sulla base di invocazioni remote “esplicite”
    - la ricezione di un messaggio da parte di un componente scatena l'esecuzione di un'operazione per gestire il messaggio ricevuto – il pattern Messaging è anche chiamato *Implicit Invocation*
- L'applicazione del pattern Messaging richiede normalmente anche l'uso di ulteriori pattern (più specifici), alcuni dei quali saranno descritti nel seguito



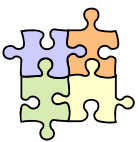
## Messaging

### □ Contesto

- integrazione di componenti (o servizi) sviluppati indipendentemente

### □ Problema

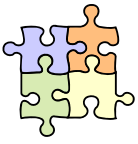
- un sistema composto da componenti (o servizi) sviluppati indipendentemente
- questi componenti devono essere integrati – per formare un sistema coerente
  - *Enterprise Application Integration – EAI*
- questi componenti devono interagire in modo affidabile
- l'accoppiamento complessivo tra i componenti deve rimanere basso
  - per essere indipendenti, i componenti non sono (e devono continuare a non essere) a conoscenza l'uno dell'altro



## Messaging

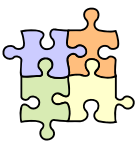
### □ Soluzione

- collega i componenti (o servizi) mediante un *bus per messaggi* – che consente ai componenti di scambiarsi *messaggi* in modo *asincrono*
- codifica i messaggi in modo che mittente e destinatario possano comunicare in modo affidabile – e senza dover conoscere staticamente tutte le informazioni sui tipi di dati
  - i messaggi
    - incapsulano richieste e strutture di dati
    - spesso auto-descrittivi – contengono dati (valori) e meta-dati (che descrivono organizzazione e significato dei dati)



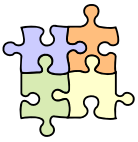
## Messaging

- Un'osservazione sulla soluzione proposta da Messaging
  - un aspetto centrale della soluzione è il “collegamento” tra i componenti indipendenti – e spesso preesistenti – che devono essere integrati
  - in effetti, l'integrazione viene proprio realizzata sulla base di questo “collegamento” – che avviene mediante la realizzazione di ulteriori elementi software, che fungono da “collante” tra gli elementi preesistenti
    - non saranno i componenti preesistenti a scambiarsi messaggi “direttamente”
    - piuttosto, saranno i nuovi elementi “collante” a scambiarsi messaggi
    - questo è descritto da opportuni pattern “di supporto” allo stile architetturale del messaging



## Messaging

- Modalità d'interazione dello stile del messaging
  - la comunicazione viene di solito iniziata dal componente (*produttore*) che produce il messaggio
  - in genere, il produttore non specifica l'identità del componente (*consumatore*) che consumerà il messaggio
    - piuttosto, il produttore invierà il suo messaggio a un *canale* (o *destinazione*) intermedio
  - un consumatore leggerà messaggi da questi canali intermedi
    - la lettura del messaggio scatenerà l'esecuzione di un'operazione opportuna da parte del consumatore, per gestire il messaggio
    - è possibile che la gestione del messaggio preveda un messaggio di risposta al produttore – ma questa non è la norma
  - invio e ricezione dei messaggi avvengono in modo asincrono



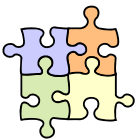
## Messaging e Publisher-Subscriber

- Due modalità principali di comunicazione asincrona basata sullo scambio di messaggi
  - sulla base di due tipi principali di canali/destinazioni
  - **Messaging**, basato su **code**
    - ciascun messaggio (documento) viene consumato da uno e un solo consumatore – è un canale di comunicazione *multi-a-uno*
  - **Publisher-Subscriber**, basato su **topic** (argomenti)
    - ciascun messaggio (evento) può essere consumato da più consumatori, registrati presso il canale – è un canale publisher-suscriber, *uno-a-molti* – i canali sono generalmente “tematici”, ovvero ciascuno è legato a un argomento – possibile un’organizzazione gerarchica degli argomenti

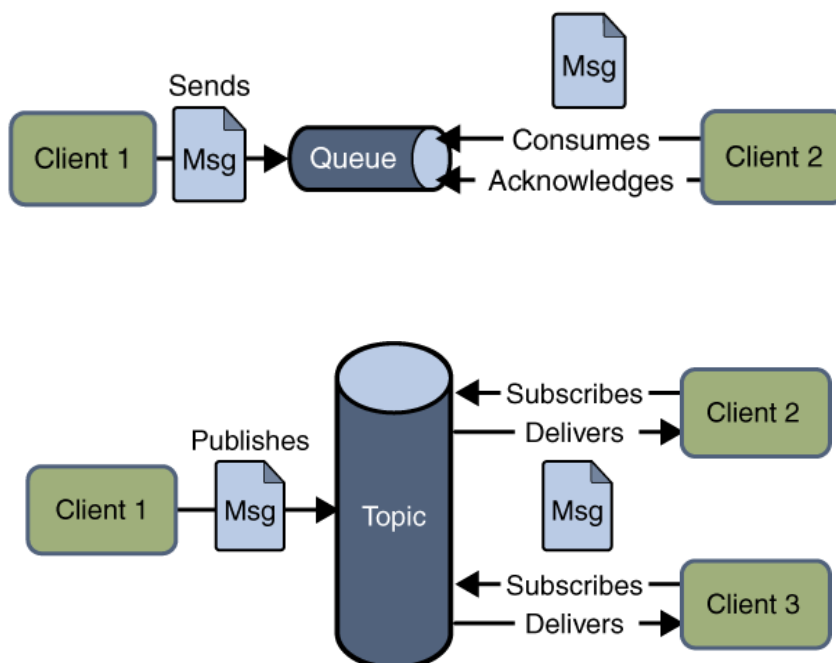
13

Messaging (stile architetturale) e integrazione di applicazioni

Luca Cabibbo – ASw



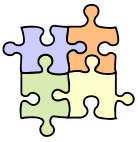
## Code e argomenti



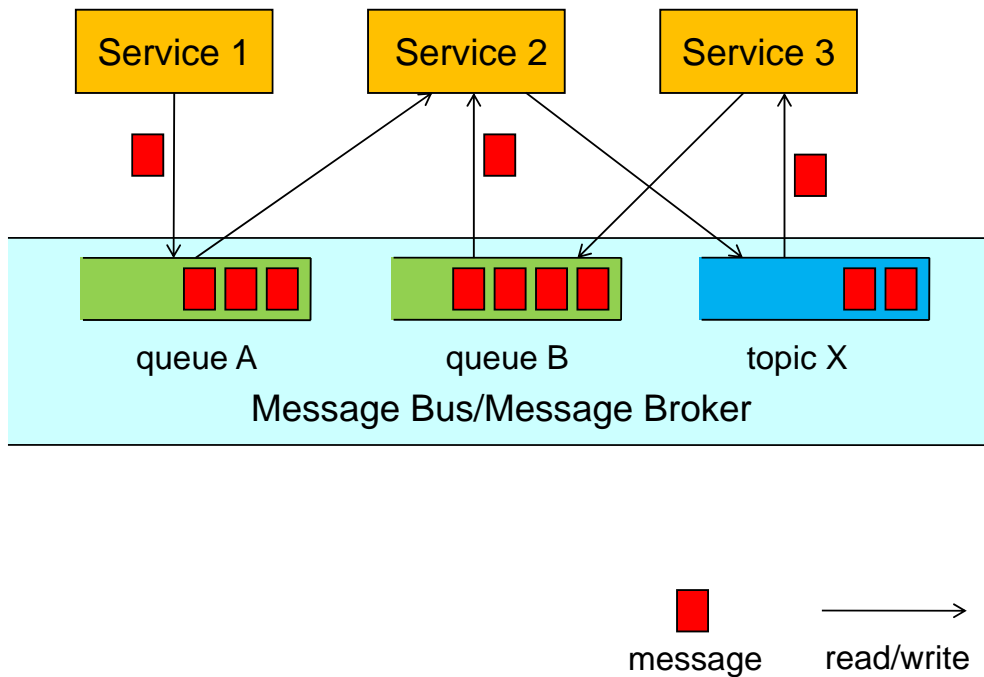
14

Messaging (stile architetturale) e integrazione di applicazioni

Luca Cabibbo – ASw



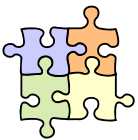
## Messaging



15

Messaging (stile architetturale) e integrazione di applicazioni

Luca Cabibbo – ASw



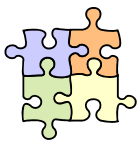
## Messaging e accoppiamento

- Il messaging sostiene un **accoppiamento debole** tra componenti
  - produttori e consumatori, per comunicare, devono essere d'accordo
    - sul formato dei messaggi scambiati
    - sul canale usato per lo scambio dei messaggi
  - produttori e consumatori non devono conoscersi ulteriormente
    - non devono conoscere l'uno l'identità dell'altro
    - non devono conoscere l'uno l'interfaccia (in senso procedurale) dell'altro
    - non devono essere attivi in modo sincrono
  - l'accoppiamento tra componenti può essere "astratto e minimale"

16

Messaging (stile architetturale) e integrazione di applicazioni

Luca Cabibbo – ASw



## Messaging

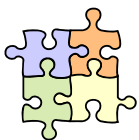
### □ Conseguenze

- ☺ manutenibilità – è possibile l'aggiunta/rimozione/sostituzione di componenti
- ☺ prestazioni – possibile replicare i consumatori di messaggi
- ☺ affidabilità – possibile la consegna affidabile (persistente o transazionale) di messaggi
- ☺ affidabilità – possibilità di effettuare il broadcast di guasti
- ☹ prestazioni – overhead dovuto alla gestione delle destinazioni (code ed argomenti) e degli eventi
- ☹ prestazioni – overhead dovuto alla necessità di codificare/decodificare messaggi
- ☹ nella comunicazione publisher-subscriber, un componente che genera eventi non sa se i suoi eventi verranno effettivamente gestiti – oppure ci potrebbero essere conflitti se un evento viene gestito da più componenti

17

Messaging (stile architetturale) e integrazione di applicazioni

Luca Cabibbo – ASw



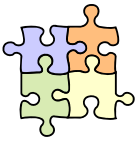
## \* Altri pattern per il Messaging [POSA4]

- L'applicazione di Messaging richiede normalmente anche l'uso di ulteriori pattern – relativi ad aspetti più specifici, e in particolare
  - la comunicazione avviene mediante lo scambio di *messaggi*
  - la comunicazione avviene tramite *canali per messaggi*
  - i componenti vengono collegati ai canali mediante componenti *endpoint* – componenti preesistenti vengono collegati mediante *adattatori*
  - possibile avere ulteriori componenti aggiuntivi
    - ad es., componenti che si occupano della *trasformazione* di messaggi (filtri) – oppure del *routing* di messaggi
  - inoltre, *publisher-subscriber* è una variante di messaging
  - tutti questi pattern hanno, tra l'altro, lo scopo di ridurre l'accoppiamento necessario tra i componenti (solitamente indipendenti) che hanno necessità di comunicare

18

Messaging (stile architetturale) e integrazione di applicazioni

Luca Cabibbo – ASw



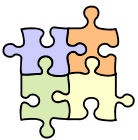
## - Message [POSA4]

### □ Problema

- come è possibile connettere due applicazioni/componenti per consentire lo scambio di pezzi di informazioni – ma anche l'invocazione di servizi?

### □ Soluzione

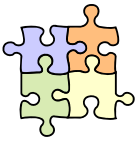
- incapsula i dati da scambiare (oppure la richiesta di invocazione) in un messaggio (*message*)
- il messaggio contiene
  - un header – specifica metadati circa le informazioni trasmesse – ad es., origine, destinazione, dimensione, scadenza, ...
  - un body (o payload) – contiene le informazioni effettive



## Message

### □ Conseguenze

- consente il trasferimento di strutture di dati e documenti tra componenti
  - i dati posseduti da un componente possono essere codificati in un messaggio, trasmessi a un altro componenti, e da questi ricostruiti
- sostiene un accoppiamento debole
  - conoscere il formato dei messaggi accettati da un destinatario è una forma di accoppiamento più debole che non conoscere l'interfaccia procedurale del destinatario
- flessibilità
  - se i messaggi sono autodescrittivi
- overhead nella codifica/decodifica dei messaggi



## - Message Channel [POSA4]

### □ Problema

- i messaggi contengono solo i dati che devono essere scambiati tra client e servizi
- per sostenere un accoppiamento debole, client e servizi non dovrebbero sapere chi è interessato a quali messaggi
- è necessario un meccanismo per connettere client e servizi, consentendo lo scambio di messaggi

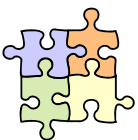
### □ Soluzione

- non connettere direttamente i client/servizi che devono collaborare – piuttosto, collegali tramite un canale per messaggi (*message channel*) che gli consente di scambiare messaggi
- quando un client deve comunicare un messaggio, lo scrive nel canale dei messaggi
- i servizi interessati al messaggio lo possono prelevare dal canale e poi elaborare

21

Messaging (stile architetturale) e integrazione di applicazioni

Luca Cabibbo – ASw



## Message Channel

### □ Discussione

- possibile (anzi, comune) l'uso di una molteplicità di canali di messaggi, ciascuno specializzato in un certo tipo di messaggi

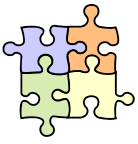
### □ Conseguenze

- sostiene un accoppiamento debole
  - conoscere una destinazione intermedia condivisa con un destinatario è una forma di accoppiamento più debole che non conoscere l'identità del destinatario
- possibile di assegnare al canale la responsabilità per alcuni attributi di qualità
  - ad es., il livello di affidabilità per la consegna dei messaggi – best effort, persistente o transazionale
- la gestione di un message channel richiede memoria, risorse di rete e eventualmente anche memorizzazione persistente

22

Messaging (stile architetturale) e integrazione di applicazioni

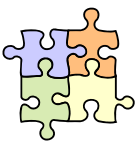
Luca Cabibbo – ASw



## - Message Endpoint [POSA4]

### □ Problema

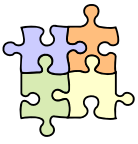
- si vogliono far comunicare, mediante lo scambio di messaggi, applicazioni o componenti autonomi (in particolare, anche preesistenti) – ma in questi componenti la comunicazione basata sullo scambio di messaggi non era prevista
- per quanto possibile, non si vogliono modificare tali applicazioni o componenti
  - non si vogliono accoppiare tali applicazioni tra loro, ma nemmeno agli elementi della soluzione di messaging – la tecnologia, i messaggi, le destinazioni, ...
- è tuttavia necessario abilitarli all'invio e/o alla ricezione di messaggi



## Message Endpoint

### □ Soluzione

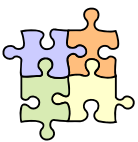
- connetti i client e i servizi che devono interagire all'infrastruttura di messaging mediante dei *message endpoint* specializzati che gli consentono di scambiare messaggi
- quando un client deve comunicare dei dati, questi dati vengono passati al (o intercettati dal) message endpoint che gli è associato – che converte i dati in un messaggio e scrive il messaggio in un message channel
- il messaggio viene ricevuto da un altro message endpoint – che estrae i dati dal messaggio e li passa al servizio che li può utilizzare, in un formato appropriato



## Message Endpoint

### □ Discussione

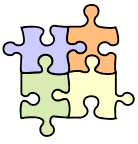
- un message endpoint incapsula certamente il codice per l'accesso al middleware di messaging – in questo modo client e servizi non sono accoppiati al sistema di messaging utilizzato (ovvero, alle sue API)
  - un message endpoint che ha solo questa finalità è anche chiamato un *messaging gateway*
  - usando un messaging gateway, un client o servizio potrebbe sapere che c'è un'infrastruttura di messaging – ma i dettagli della tecnologia utilizzata gli sono completamente trasparenti



## Message Endpoint

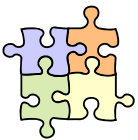
### □ Discussione

- un altro caso particolare (e importante) di message endpoint è un *channel adapter*
  - ha lo scopo di nascondere completamente l'infrastruttura di messaging a un componente o applicazione
  - ovvero, con l'utilizzo di un channel adapter, un client o servizio non sa nemmeno che c'è un'infrastruttura di messaging
- molti elementi che fungono da “collante” in un sistema di integrazione, per “collegare” degli elementi preesistenti, sono proprio dei channel adapter



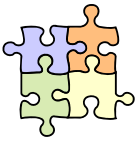
## Message Endpoint

- Discussione – alcuni esempi di channel adapter
  - in un'applicazione per basi di dati
    - un channel adapter utilizza un trigger che cattura un particolare cambiamento nella base di dati (ad es., “è stato memorizzato un nuovo ordine”) e si attiva per generare e trasmettere degli opportuni messaggi
  - in un'applicazione client-server
    - un channel adapter riceve un messaggio relativo a un ordine, e deve verificare la disponibilità del prodotto ordinato – si comporta da client di un servizio di inventario, e comunica l'esito della verifica tramite un messaggio inviato a un altro componente



## - Message Translator [POSA4]

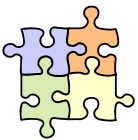
- Problema
  - per sostenere un accoppiamento debole, non è sempre possibile ipotizzare che i servizi che ricevono i messaggi comprendano il formato dei messaggi utilizzato dai client che generano tali messaggi
  - è spesso necessario trasformare i messaggi dal formato utilizzato dai client al formato compreso dai servizi



# Message Translator

## □ Soluzione

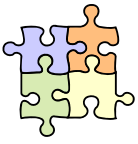
- introduci dei traduttori di messaggi (*message translator*) tra client e servizi, in grado di convertire messaggi da un formato all'altro
- un client invia un messaggio nel formato che preferisce
- il message translator garantisce che il servizio riceva il messaggio nel formato a lui preferito
- è possibile definire message translator che realizzano una traduzione bidirezionale tra formati di messaggi



# Message Translator

## □ Discussione

- sostiene un accoppiamento debole
- ci sono strumenti dedicati alla traduzione di messaggi tra formati diversi – tipicamente basati sull'uso di XML e linguaggi di interrogazione/trasformazione ad esso associati



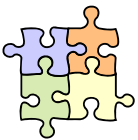
## - Message Router [POSA4]

### □ Problema

- i messaggi scambiati tra client e servizi devono essere instradati nell'infrastruttura di messaging
- client e servizi (e anche canali e traduttori) non dovrebbero avere conoscenza del cammino di instradamento da utilizzare
- è tuttavia necessario scegliere un percorso per la propagazione dei messaggi

### □ Soluzione

- introduci dei *message router* che consumano messaggi da un canale e li re-inseriscono in un altro canale, sulla base di alcune condizioni (ad esempio, sull'header o sul contenuto del messaggio)
- un message router connette un insieme di canali per messaggi in una rete di canali per messaggi – muovendo ciascun messaggio verso il ricevitore più opportuno



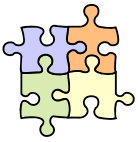
## - Publisher-Subscriber [POSA4]

### □ Problema

- i componenti di un gruppo di applicazioni distribuite sono debolmente accoppiati tra di loro, e operano in modo largamente indipendente
- è necessario un meccanismo di notifica – ad esempio, per propagare informazioni ad alcuni o a tutti i componenti
- queste notifiche sono relative ad eventi che potrebbero avere effetto sull'elaborazione svolta dai singoli componenti

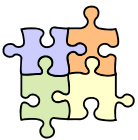
### □ Soluzione

- definisci un'infrastruttura per propagare notifiche che consenta a *publisher* di diffondere eventi che potrebbero interessare altri, e a *subscriber* di essere notificati di questi eventi quando tali informazioni sono pubblicate
- usa degli opportuni *canali publish-subscribe*



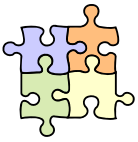
## \* Messaging per l'integrazione di applicazioni

- Le applicazioni “interessanti” vivono raramente in isolamento
  - devono spesso comunicare tra loro, per scambiarsi dati o servizi
  - questo solleva il problema dell'integrazione di applicazioni – *Enterprise Application Integration* o *EAI*
- Nel corso del tempo, sono stati introdotti e utilizzati in pratica diversi approcci per l'integrazione di applicazioni
  - trasferimento di file
  - base di dati condivisa
  - invocazione di procedure remote
    - tuttavia questi approcci si sono mostrati spesso insoddisfacenti, per diversi motivi
  - il messaging è un ulteriore approccio per l'EAI – che supera numerosi limiti degli approcci precedenti



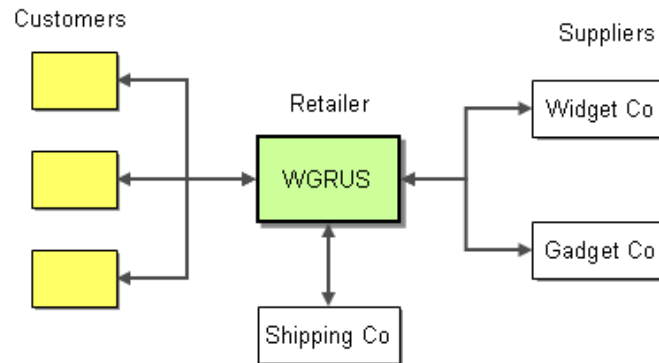
## Messaging per l'integrazione di applicazioni

- Il messaging è considerata una tecnologia fondamentale nell'integrazione di applicazioni preesistenti
  - l'integrazione avviene realizzando un'infrastruttura di comunicazione tra le applicazioni preesistenti, basata appunto sul messaging
  - il messaging viene (talvolta) preferito ad altre tecnologie perché richiede un accoppiamento basso tra i componenti e offre una maggior flessibilità
    - alcuni vantaggi – accoppiamento debole, asincronia, consegna “immediata” (appena possibile), affidabilità, formati personalizzati, ...
- Il messaging per l'integrazione di applicazioni viene esemplificato con riferimento allo studio di caso Widgets & Gadgets 'R Us
  - <http://www.enterpriseintegrationpatterns.com/Chapter1.html>

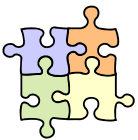


## - Studio di caso: Widgets & Gadgets 'R Us

- Widgets & Gadgets 'R Us è un rivenditore che acquista e rivende “widgets” e “gadgets”
  - nato dalla fusione di due aziende

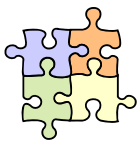


- il sistema **WGRUS** deve integrare alcuni componenti (preesistenti) dei sistemi informatici (preesistenti) di Widget Co e Gadget Co



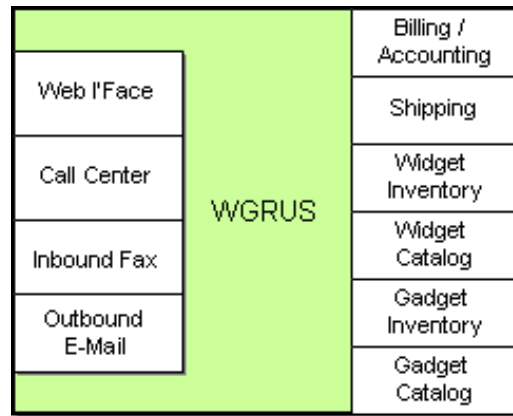
## WGRUS

- Funzionalità di WGRUS
  - inserimento ordini
  - elaborazione ordini
  - verifica stato dell'ordine
  - gestione clienti
  - gestione catalogo prodotti
  - ....
- Consideriamo (parzialmente) solo la gestione degli ordini (inserimento, elaborazione, verifica stato)



## WGRUS come problema di integrazione

- Come detto, il sistema WGRUS deve realizzare le varie funzionalità integrando alcuni componenti preesistenti
  - tra cui i sistemi preesistenti di Widget Co e Gadget Co – a loro volta composti da vari elementi

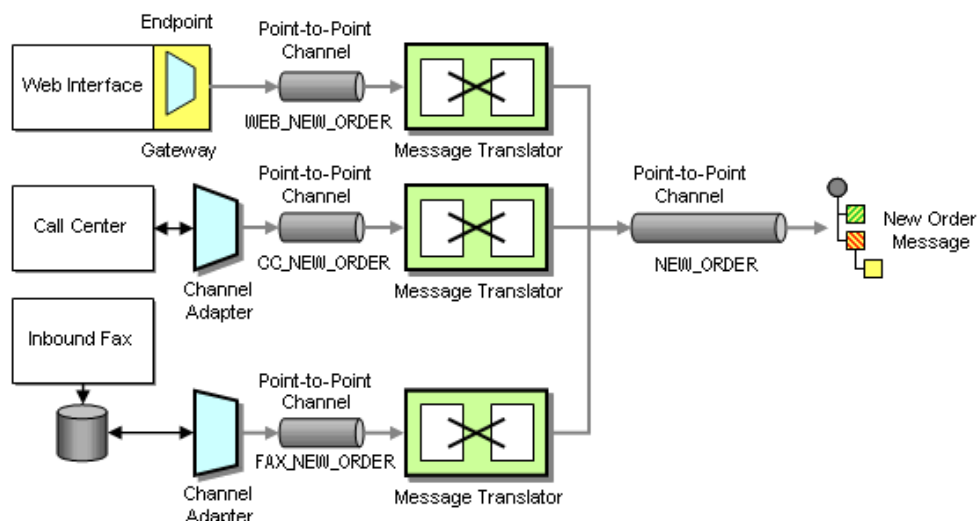


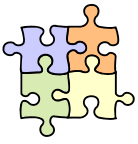
- a sinistra, sono mostrati i canali di interazione con i clienti
- a destra, i componenti applicativi preesistenti da riusare



## - Ricezione di ordini

- Gli ordini possono essere ricevuti/immessi da vari client
  - un client web, un client per un addetto al telefono, ordini ricevuti via fax – ciascuno genera ordini con un formato diverso
  - si vuole invece avere un flusso di messaggi (unico e omogeneo) per tutti gli ordini





## Pattern per l'EAI (1)

### □ Alcuni pattern per l'Enterprise Application Integration

#### ▪ *Message*

- un messaggio – ovvero, un tipo/flusso di messaggi



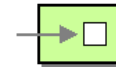
#### ▪ *Message (Point-to-Point) Channel*

- un canale (una coda) per lo scambio di messaggi



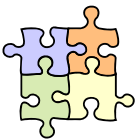
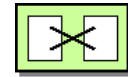
#### ▪ *Message Endpoint*

- collega un componente al sistema di messaging, per trasmettere/ricevere messaggi



#### ▪ *Message Translator*

- una trasformazione che cambia il formato di un messaggio

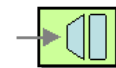


## Pattern per l'EAI (2)

### □ Esistono vari tipi di *Message Endpoint* – tra cui

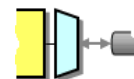
#### ▪ *Messaging Gateway*

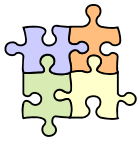
- endpoint che incapsula l'accesso al sistema di messaging, fornendo un'interfaccia con i metodi specifici del dominio applicativo – ma indipendenti dal sistema di messaging usato



#### ▪ *Channel Adapter*

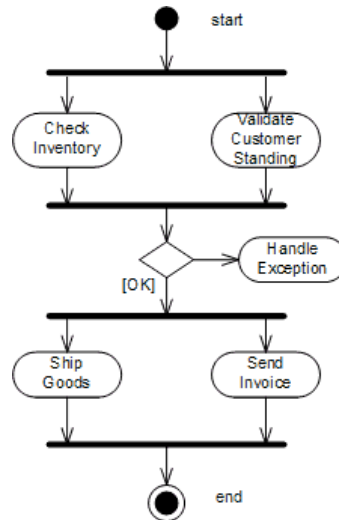
- endpoint che realizza una connessione tra un'applicazione (di solito preesistente) e il sistema di messaging





## - Elaborazione di ordini

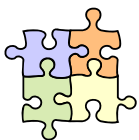
- Ora abbiamo un flusso consistente di ordini – l'elaborazione di un ordine richiede
  - verifica dello stato del cliente – nessun debito in sospeso
  - verifica dell'inventario – disponibilità degli articoli ordinati
  - se tutto ok, si può procedere



41

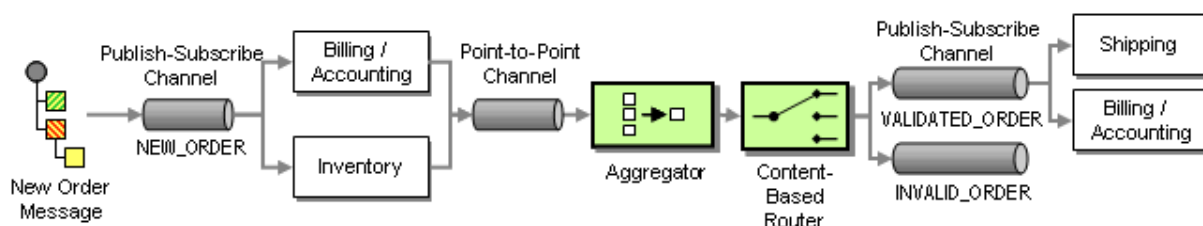
Messaging (stile architetturale) e integrazione di applicazioni

Luca Cabibbo – ASw



## Elaborazione di ordini

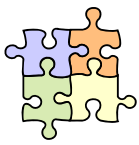
- Come elaborare gli ordini?
  - ordini inviati separatamente e in parallelo a contabilità e inventario per le verifiche
  - le due risposte devono poi essere aggregate
  - gli ordini confermati vengono inviati ai sistemi di spedizione e di fatturazione



42

Messaging (stile architetturale) e integrazione di applicazioni

Luca Cabibbo – ASw



## Pattern per l'EAI (3)

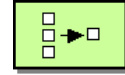
- **Publish-Subscribe Channel**

- un canale (un topic/argomento) per lo scambio di messaggi



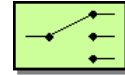
- **Aggregator**

- combina il contenuto di messaggi diversi ma correlati



- **Content-Based Router**

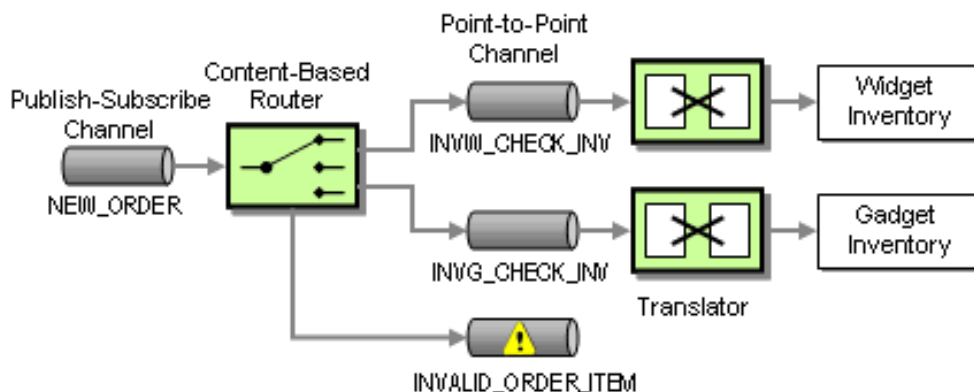
- gira un messaggio a un'opportuna destinazione, sulla base del contenuto del messaggio

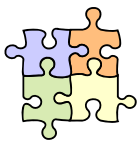


## - Controllo dell'inventario

- In realtà, ci sono due sistemi/funzionalità per il controllo dell'inventario

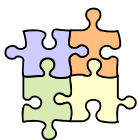
- una per i Widget e una per i Gadget
- ciascuna richiesta va instradata al sistema giusto
  - ipotesi (temporanea): una sola riga d'ordine per ordine
  - ipotesi (semplificativa): il primo carattere del codice del prodotto è G o W





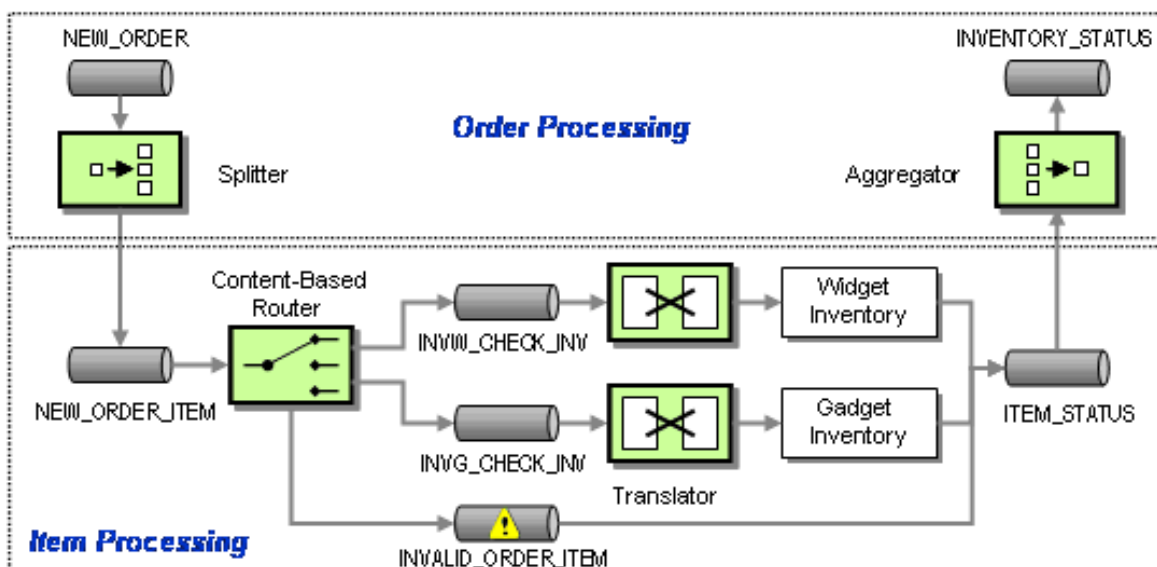
## Pattern per l'EAI (4)

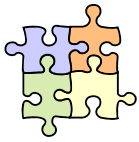
- **Invalid Message Channel**
  - destinazione di messaggi non validi



## - Ordini con più righe d'ordine

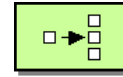
- In realtà, un ordine contiene normalmente più righe d'ordine
  - alcune saranno relative a widget, altre a gadget
  - la disponibilità delle merci va verificata riga d'ordine per riga d'ordine





## Pattern per l'EAI (5)

### ▪ *Splitter*



- decompone un messaggio in un insieme di messaggi, ciascuno dei quali può richiedere (successivamente) una diversa elaborazione

47

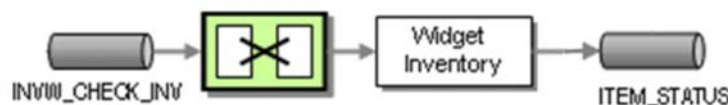
Messaging (stile architetturale) e integrazione di applicazioni

Luca Cabibbo – ASw

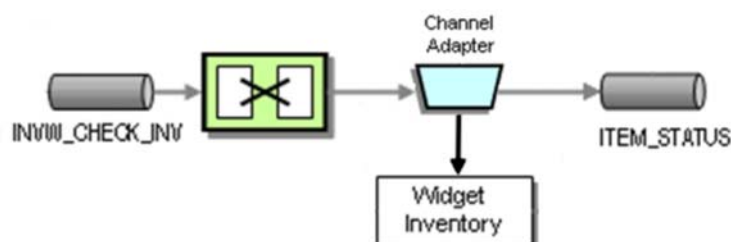


## Un'osservazione

- ▣ Si consideri questa porzione del diagramma



- sembra che il componente preesistente Widget Inventory partecipi in prima persona alla soluzione di integrazione – ma è proprio lui che consuma e produce messaggi direttamente?
- no, tale componente verrà probabilmente acceduto mediante un opportuno message endpoint (ad es., un channel adapter)

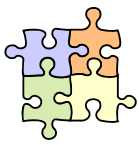


- considerazioni analoghe per gli altri componenti preesistenti

48

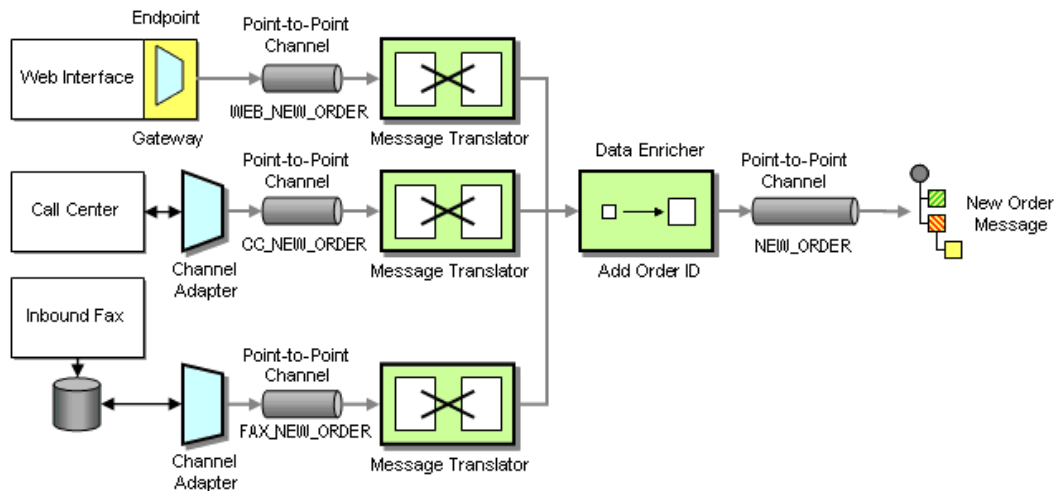
Messaging (stile architetturale) e integrazione di applicazioni

Luca Cabibbo – ASw



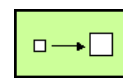
## - Identificatore d'ordine

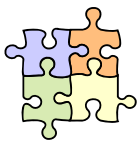
- Messaggi elaborati separatamente possono essere ricombinati mediante un Aggregator sulla base di opportune informazioni di correlazione
  - ad es., un identificatore d'ordine
  - ma è necessario aggiungere un identificatore a ciascun ordine



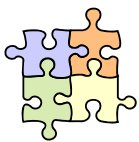
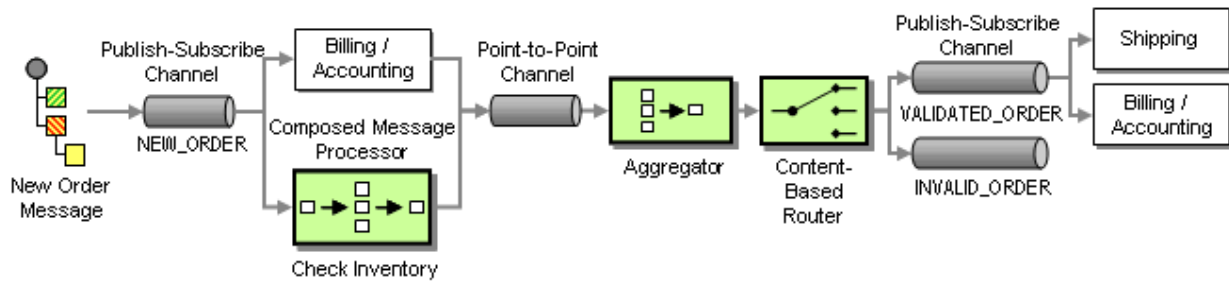
## Pattern per l'EAI (6)

- **Content Enricher**
  - aggiunge informazioni a un messaggio



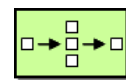


## - Gestione degli ordini - rivista

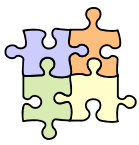


## Pattern per l'EAI (7)

### ▪ *Composed Message Processor*

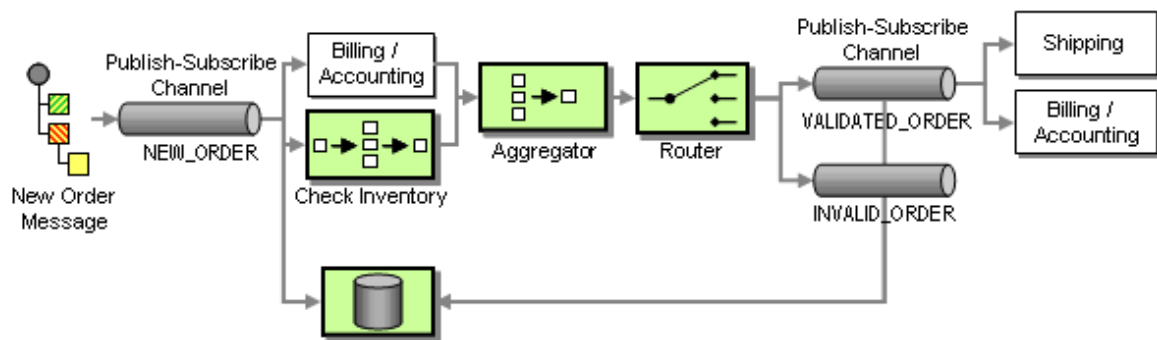


- mantiene il flusso di messaggi complessivo, anche se i diversi messaggi richiedono elaborazioni diverse



## - Verificare lo stato di un ordine

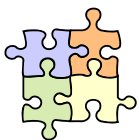
- L'elaborazione di un ordine richiede lo svolgimento di varie attività
  - come consentire a un utente di verificare lo stato di un suo ordine? è stata effettuata la spedizione? è in attesa di prodotti? bloccato perché il cliente ha debiti in sospeso?
  - è possibile rispondere conoscendo l'“ultimo” messaggio scambiato nel sistema circa l'ordine – questo può essere fatto memorizzando i messaggi rilevanti in un repository di messaggi



53

Messaging (stile architetturale) e integrazione di applicazioni

Luca Cabibbo – ASw



## Pattern per l'EAI (8)

### ▪ *Message Store*

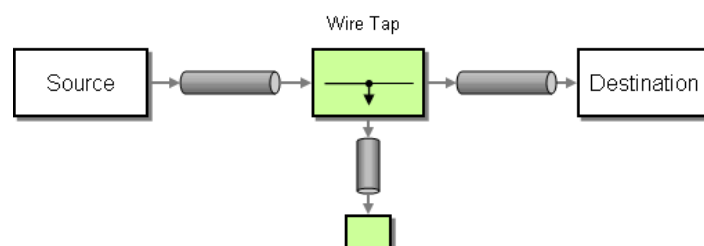


- quando viene inviato un messaggio nel sistema, viene inviato anche un messaggio duplicato e memorizzato in un repository di messaggi
  - semplice se il canale di cui bisogna memorizzare i messaggi è di tipo *Publish-Subscribe*

### ▪ *Wire Tap*



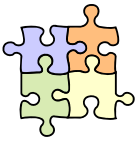
- per duplicare su più canali i messaggi inviati su un certo canale



54

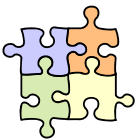
Messaging (stile architetturale) e integrazione di applicazioni

Luca Cabibbo – ASw



## - Discussione

- L'esempio WGRUS mostra l'applicazione di alcuni pattern per il messaging per l'integrazione di applicazioni
  - i componenti preesistenti non vengono collegati tra di loro direttamente
  - piuttosto, l'integrazione è basata su costruzione, consumo, trasformazione, splitting, aggregazione e routing di messaggi, anche con riferimento a un certo numero di canali di comunicazione
  - i componenti preesistenti sono collegati al sistema di messaging mediante adattatori (message endpoint) – nuovi componenti software “collante”, che comunque incapsulano l'accesso al sistema di messaging
  
- Il messaging può essere utilizzato anche per lo sviluppo di nuove applicazioni



## Discussione

- Oltre a quelli visti, [Hohpe&Woolf 2004] presenta diversi altri pattern per il messaging – alcuni dei quali ripresi da [POSA4] – per rappresentare, tra l'altro
  - elementi dei sistemi di messaging – ad es., *Message* o *Message Channel*
  - canali di messaging – ad es., *Point-to-point Channel* o *Guaranteed Delivery*
  - tipi di messaggi – ad es., *Document Message* o *Request-Reply*
  - routing di messaggi – ad es., *Splitter* o *Aggregator*
  - trasformazioni di messaggi – ad es., *Content Enricher* o *Content Filter*
  - estremità per lo scambio di messaggi – ad es., *Messaging Gateway*
  - gestione e monitoraggio del sistema – ad es., *Control Bus* o *Process Manager*