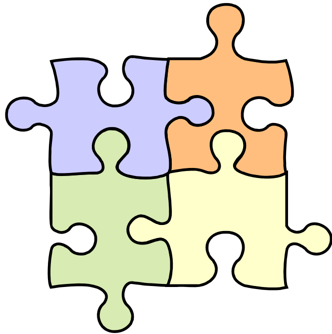
Luca Cabibbo



Architetture Software

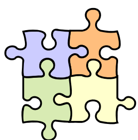
Messaging (stile architetturale) e integrazione di applicazioni

Dispensa ASW 430

ottobre 2014

Una specifica d'interfaccia di buona qualità deve essere semplice, non ambigua, completa, concisa e focalizzata sulla sostanza.

Harry Hillaker



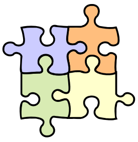
- Fonti

- [POSA4] Pattern-Oriented Software Architecture – A Pattern Language for Distributed Computing, 2007
- [EIP] Hohpe, Woolf, Enterprise Integration Patterns, 2004
  - *http://www.enterpriseintegrationpatterns.com/*
  - *http://eaipatterns.com/*

- Obiettivi e argomenti

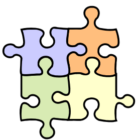
- Obiettivi
  - presentare il messaging – qui inteso come stile architetturale
  - presentare e discutere uno studio di caso sull'applicazione del messaging all'integrazione di applicazioni
- Argomenti
  - introduzione
  - messaging [POSA4]
  - ulteriori pattern per il messaging
  - messaging e integrazione di applicazioni

- Wordle



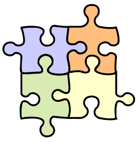
* Introduzione

- Le tecnologie di middleware, utilizzate per lo sviluppo dei sistemi distribuiti, sostengono di solito pochi stili fondamentali di comunicazione distribuita
  - ad esempio, le architetture client/server e le architetture a oggetti distribuiti, studiate in precedenza, sono basate su un paradigma di interazione di tipo richiesta-risposta – ovvero, su un meccanismo distribuito di chiamata di procedure remote/ invocazione di metodi remoti
    - comunicazione molti-a-uno – diretta o indiretta
    - basato su interfacce procedurali, con una tipizzazione statica (forte)
    - accoppiamento forte dei client nei confronti dei server



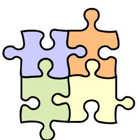
Broker

- [POSA4] associa questa modalità di interazione allo stile architetturale *Broker* – che costituisce un'infrastruttura di comunicazione fondamentale per i sistemi distribuiti
  - *Broker* organizza un sistema distribuito come un insieme di componenti che interagiscono tramite l'invocazione di metodi remoti
    - gli aspetti della comunicazione tra questi componenti sono gestiti da un broker (o da una federazione di broker) – che rende (rendono) trasparente la locazione dei servizi in rete
  - in questo stile, molti client possono effettuare invocazioni remote ai componenti server
    - i client comunicano con i server in una modalità molti-a-uno
    - la comunicazione avviene sulla base di interfacce procedurali, con una tipizzazione forte

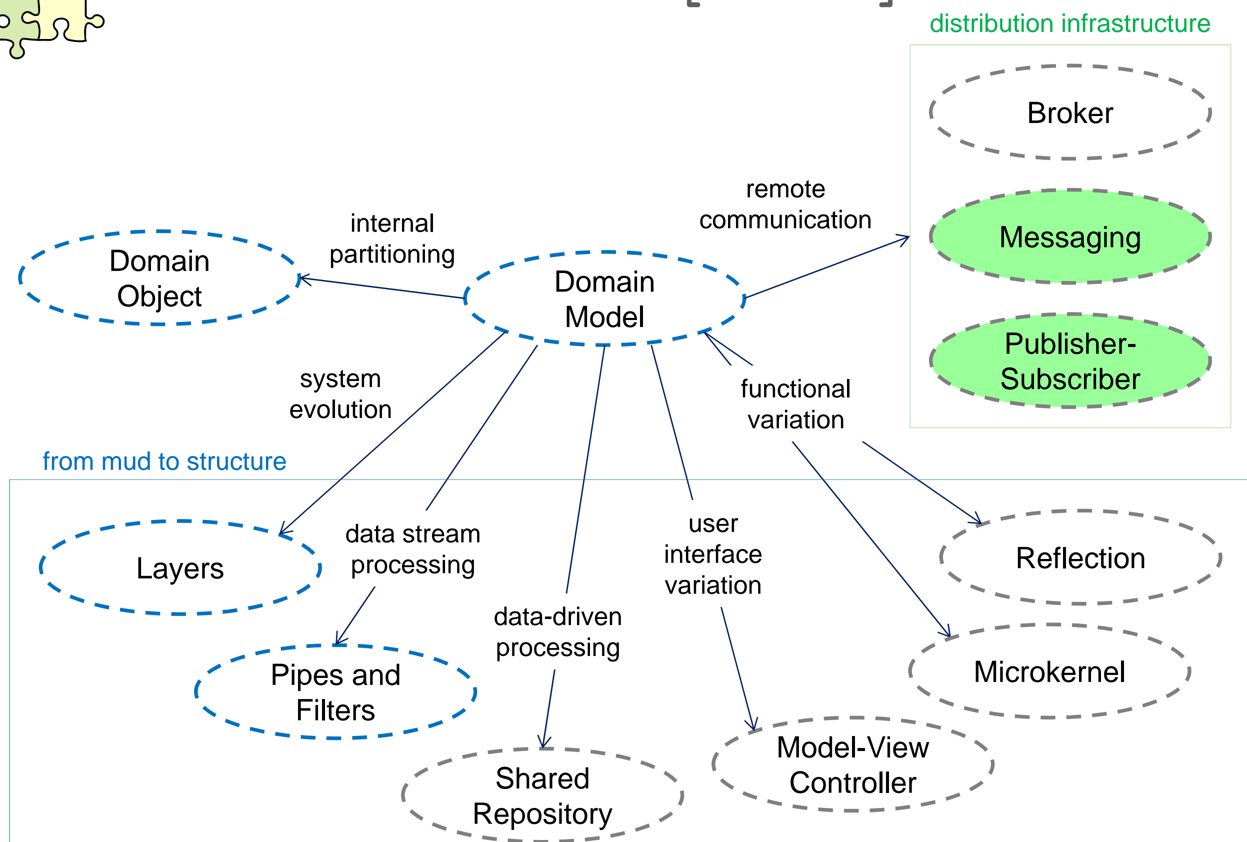


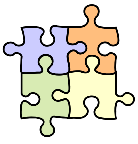
Messaging e Publisher/Subscriber

- Oltre a *Broker*, [POSA4] propone altri pattern stili architetturali, che descrivono altre due modalità principali di interazione – *Messaging* e *Publisher/Subscriber* – questi pattern costituiscono degli ulteriori stili fondamentali di comunicazione nei sistemi distribuiti
  - questi due stili di comunicazione
    - sono basati sullo scambio di messaggi/documenti oppure di notifiche di eventi – per quanto possibile auto-descrittivi – insieme a meccanismi di invocazione implicita
    - comunicazione molti-a-uno o uno-a-molti
    - accoppiamento debole tra i componenti



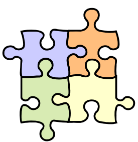
Stili architetturali di [POSA4]





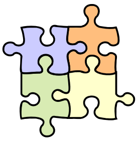
Messaging

- *Messaging* organizza un sistema distribuito come un insieme di componenti che interagiscono tramite lo scambio di messaggi
  - lo scambio di messaggi viene gestito da un insieme di canali di comunicazione – insieme ad ulteriori elementi
  - questo stile rilassa accoppiamento e tipizzazione
    - i componenti inviano messaggi tipati ad altri componenti – sulla base di un'interfaccia "a messaggi/documenti" – ma non richiedono l'esecuzione di metodi specifici
    - comunicazione molti-a-uno, ma senza dipendenze statiche tra le interfacce procedurali dei componenti



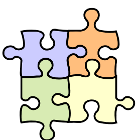
Publisher/Subscriber

- *Publisher/Subscriber* organizza un sistema distribuito in termini di componenti che interagiscono tramite lo scambio asincrono di eventi in una configurazione uno-a-molti
  - publisher e subscriber di eventi non si conoscono tra loro
    - i subscriber sono interessati a consumare eventi, ma non sono interessati a chi li produce
    - analogamente, i publisher forniscono eventi, ma non sono interessati a chi li consumerà
  - questo stile disaccoppia i componenti ancor di più
    - i componenti si scambiano eventi senza conoscersi
    - i subscriber reagiscono agli eventi eseguendo qualche azione – ma i publisher non avviano direttamente l'esecuzione di operazioni specifiche sui subscriber



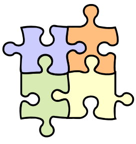
Messaging

- In generale, questi due stili architetturali sono sostenuti dal middleware per il *messaging* – attenzione a non confondere stile e tecnologia
  - i componenti o le applicazione comunicano, in una relazione tra pari, scambiandosi messaggi in modo asincrono
    - un *componente* (nel ruolo di *produttore*) può inviare *messaggi* a un altro *componente* (nel ruolo di *consumatore*)
    - ciascun messaggio può codificare dei dati oppure un documento oppure la notifica di un evento
  - il messaging è un paradigma di interazione significativamente diverso da quello richiesta/risposta su cui sono basati RPC e RMI

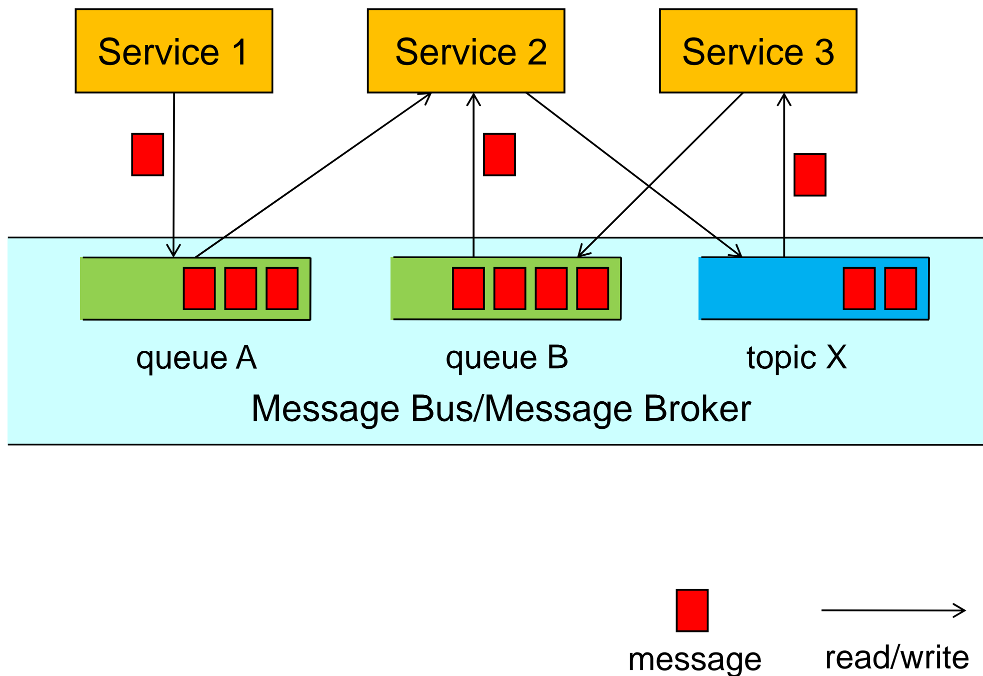


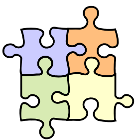
Messaging

- Paradigma di interazione del messaging
  - la comunicazione viene iniziata da un componente (*produttore*) che produce un messaggio
  - il produttore non conosce 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 non è la norma
  - invio e ricezione dei messaggi avvengono in modo asincrono



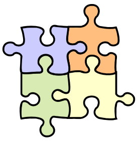
Messaging



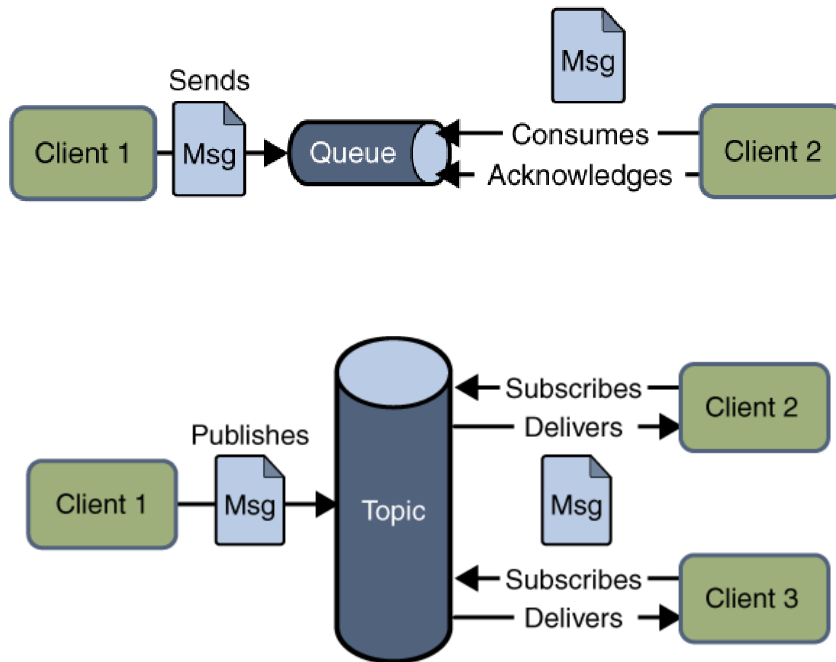


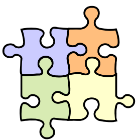
Messaging e Publisher-Subscriber

- 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 *molti-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



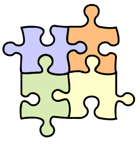
Code e argomenti





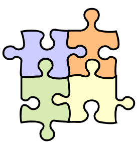
Middleware message-oriented

- 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



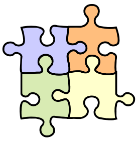
Messaging e strumenti di middleware

- 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



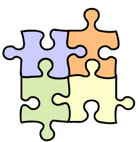
Messaging

- Benefici specifici dell'uso delle tecnologie di messaging
  - abilita la comunicazione remota
  - comunicazione asincrona
    - temporizzazione variabile – ogni componente può partecipare all'interazione secondo la propria velocità
    - eliminazione di colli di bottiglia – nella comunicazione sincrona, un numero elevato di richieste concorrenti può sovraccaricare il server
    - abilita operazioni disconnesse
  - affidabilità della comunicazione
  - può favorire l'interoperabilità tra linguaggi e piattaforme
  - consente una migliore gestione dei thread



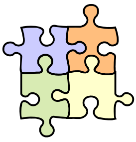
Messaging

- Sfide associate all'uso del messaging
  - maggior complessità del paradigma di programmazione basato sullo scambio di messaggi asincroni (vedi lucido successivo)
  - l'overhead della comunicazione può avere un impatto negativo sulle prestazioni
  - problemi indotti dall'ordine di ricezione dei messaggi
  - è comunque di solito necessario gestire anche scenari che sono inerentemente sincroni
  - molti strumenti di messaging sono basati su protocolli e implementazioni proprietarie – può limitare l'interoperabilità – lock-in dalla piattaforma



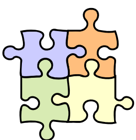
Messaging

- Alcune implicazioni della comunicazione asincrona
  - non c'è più un singolo thread di esecuzione
    - la concorrenza favorisce prestazioni e scalabilità – ma rende testing e debugging più difficili
  - se l'esecuzione di un'operazione prevede dei risultati, anche questi arrivano di solito in modo asincrono, mediante dei callback
    - i componenti devono prevedere l'interruzione delle loro attività per gestire notifiche di questo tipo
  - l'esecuzione di sottoprocessi che comunicano in modo asincrono può procedere secondo un ordine qualunque
    - i sottoprocessi devono poter procedere indipendentemente dagli altri – la loro sincronizzazione è più difficile



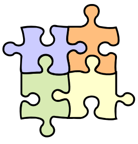
* Messaging [POSA4]

- Lo stile architetturale **Messaging** – proposto inizialmente in [EIP], poi ripreso da [POSA4]
  - 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 dello stile Messaging richiede normalmente anche l'uso di ulteriori pattern architetturali (più specifici, e di portata più limitata), alcuni dei quali saranno descritti nel seguito



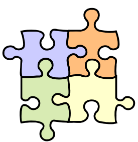
Messaging e Publisher-Subscriber

- [POSA4] presenta due stili architetturali fondamentali distinti per la comunicazione asincrona
  - messaging
    - comunicazione basata sullo scambio di messaggi
    - accoppiamento debole tra produttori e consumatori di messaggi
    - comunicazione molti-a-uno – "code"
  - publisher-subscriber
    - comunicazione basata sulla notifica di eventi
    - accoppiamento ancora più debole
    - comunicazione uno-a-molti – "topic"
  - in questa trattazione, ci concentriamo inizialmente sul messaging – e consideriamo publisher-subscriber una sua variante, ovvero uno stile "sinergico" al messaging



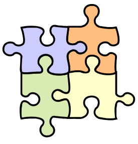
Messaging

- Contesto
  - integrazione di un insieme di componenti (o servizi) sviluppati indipendentemente
- Problema
  - bisogna realizzare un sistema composto a partire da un insieme di componenti (o servizi) sviluppati indipendentemente
  - questi componenti devono essere integrati – per formare un sistema coerente – *Enterprise Application Integration* (*EAI*)
  - l'accoppiamento complessivo tra i componenti deve rimanere basso
    - i componenti sono inizialmente indipendenti – ovvero, non sono a conoscenza l'uno dell'altro – dopo l'integrazione, i componenti devono rimanere ancora indipendenti
  - questi componenti devono interagire in modo affidabile



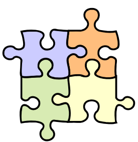
Messaging

- 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)



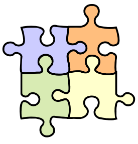
Messaging

- 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 l'introduzione 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 può avvenire applicando opportuni pattern di supporto allo stile architetturale del messaging, descritti nel seguito



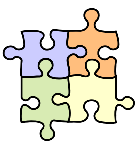
Messaging e accoppiamento

- Il messaging è una tecnologia che abilita l'integrazione di applicazione – poiché sostiene un *accoppiamento debole* tra componenti software
  - 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"



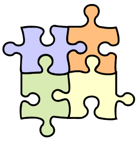
Messaging

- 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



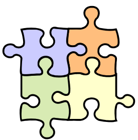
Messaging

- Conseguenze
  - ☹ prestazioni – overhead dovuto alla gestione delle destinazioni (code ed argomenti) e degli eventi
  - ☹ prestazioni – overhead dovuto alla necessità di codificare/decodificare messaggi
  - ☹ verifica e affidabilità – la mancanza di interfacce tipate staticamente (o della loro conoscenza) rende difficile verificare/validare il comportamento del sistema
  - ☹ verifica e affidabilità – 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



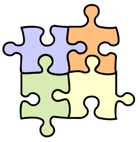
* Ulteriori pattern per il Messaging

- L'applicazione del Messaging per l'EAI richiede anche l'uso di ulteriori pattern – relativi ad aspetti più specifici, tra cui
  - la comunicazione avviene mediante lo scambio di *messaggi*
  - la comunicazione avviene tramite *canali per messaggi*
  - i componenti vengono collegati ai canali mediante componenti *endpoint* – alcuni endpoint sono realizzati nella forma di *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 tuttavia necessità di comunicare



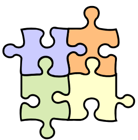
- Message [EIP]

- Problema
  - come è possibile connettere due componenti o servizi per consentire lo scambio di pezzi di informazioni – ma anche l'invocazione di operazioni/servizi?
- Soluzione
  - incapsula i dati da scambiare (oppure un documento oppure la notifica di un evento oppure la richiesta di invocazione di un servizio) in un messaggio (*message*)
  - il messaggio è formato da
    - un body (o payload) – contiene i dati effettivi
    - un header – specifica metadati circa i dati trasmessi – ad es., origine, destinazione, dimensione, scadenza, ...



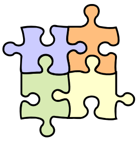
Message

- Conseguenze
  - consente il trasferimento di strutture di dati, documenti ed eventi tra componenti
    - i dati posseduti da un componente possono essere codificati in un messaggio, trasmessi ad altri 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 – ad es., usando un formato di interscambio come XML o JSON
  - overhead nella codifica/decodifica dei messaggi



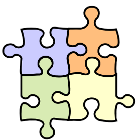
- Message Channel [POSA4]

- 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 conoscere 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 componenti che devono interagire – piuttosto, collegali tramite un canale per messaggi (*message channel*) che gli consente di scambiare messaggi
  - quando un client deve comunicare un messaggio, lo inoltra in un canale per messaggi
  - i servizi interessati al messaggio (oppure disponibili ad elaborarlo) lo possono poi prelevare dal canale ed elaborare



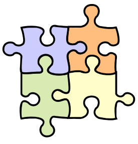
Message Channel

- Discussione
  - possibile (anzi, comune) l'uso di una molteplicità di canali per 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



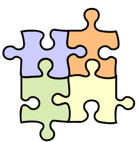
- Message Endpoint [POSA4]

- 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, si vuole sostenere un accoppiamento basso tra questi componenti e la soluzione tecnologica
    - certamente, non si vogliono accoppiare questi componenti agli elementi della soluzione di messaging – la tecnologia, i messaggi, le destinazioni, ...
    - se possibile, non si vogliono nemmeno accoppiare questi componenti tra loro – spesso, infatti, non si possono (oppure non si vogliono) modificare questi componenti
  - è tuttavia necessario abilitarli all'invio e/o alla ricezione di messaggi



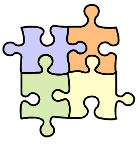
Message Endpoint

- 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, li passerà all'endpoint a lui associato
    - l'endpoint converte i dati in un messaggio, e poi inoltra il messaggio in un message channel
    - in alcuni casi, il client non comunica direttamente con l'endpoint – piuttosto, l'endpoint intercetta i dati del client
  - il messaggio non viene ricevuto direttamente da un componente o servizio – piuttosto, viene ricevuto da un altro endpoint
    - questo endpoint estrae i dati dal messaggio, e poi li passa, in un formato appropriato, al servizio che li può utilizzare



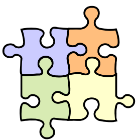
Message Endpoint

- Discussione
  - un message endpoint incapsula certamente tutto il codice per l'accesso alle API del middleware di messaging – in questo modo client e servizi non sono accoppiati allo specifico sistema di messaging utilizzato
    - un message endpoint che ha solo questa finalità è anche chiamato un *messaging gateway* [EIP]
    - usando un messaging gateway, i componenti possono essere a conoscenza dell'esistenza di un'infrastruttura di messaging – ma il messaging gateway protegge questi componenti dai dettagli della tecnologia utilizzata, che rimangono dunque trasparenti ai componenti



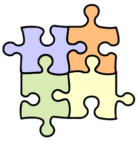
Message Endpoint

- Discussione
  - un altro caso particolare (e importante) di message endpoint è un *channel adapter* [EIP]
    - 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 conosce nemmeno che viene utilizzato nell'ambito di una soluzione di messaging
  - molti elementi che fungono da "collante" in un sistema di integrazione, per "collegare" degli elementi preesistenti, sono proprio dei channel adapter



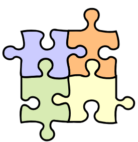
Message Endpoint

- Discussione – alcuni esempi di channel adapter
  - in un'applicazione per basi di dati
    - un channel adapter potrebbe essere basato su un trigger per catturare un particolare cambiamento nella base di dati (ad es., "è stato memorizzato un nuovo ordine") – l'adapter si attiva per generare e trasmettere degli opportuni messaggi
    - non è necessario modificare il componente che inserisce gli ordini nella base di dati



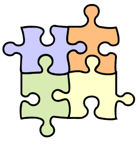
Message Endpoint

- Discussione – alcuni esempi di channel adapter
  - in un'applicazione client-server
    - un channel adapter si potrebbe interporre tra i client e un server – intercetta le richieste inviate al server, e le utilizza per generare ed inviare messaggi
    - non è necessario modificare né i componenti client né il componente che implementa il servizio
  - in un'applicazione client-server
    - un channel adapter potrebbe ricevere un messaggio relativo a un ordine, per 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
    - non è necessario modificare il componente che implementa il servizio richiesto



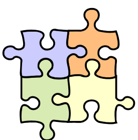
- Message Translator [POSA4]

- 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 allora spesso necessario trasformare i messaggi dal formato utilizzato dai client al formato compreso dai servizi



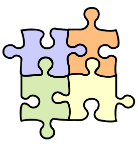
Message Translator

- 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 anche possibile definire dei message translator che realizzano una traduzione bidirezionale tra formati di messaggi



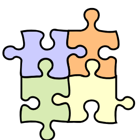
Message Translator

- 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 per XML



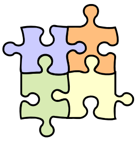
- Message Router [POSA4]

- 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 altri canali, 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 destinatario più opportuno



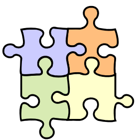
- Publisher-Subscriber [POSA4]

- 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 di eventi – ad esempio, per propagare informazioni ad alcuni oppure 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 consente a dei *publisher* di diffondere eventi che potrebbero interessare altri – e a dei *subscriber* di essere notificati di questi eventi quando tali informazioni sono pubblicate
  - usa degli opportuni *canali publish-subscribe*



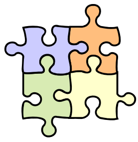
* Messaging e integrazione di applicazioni

- 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
  - basi di dati condivise
  - 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



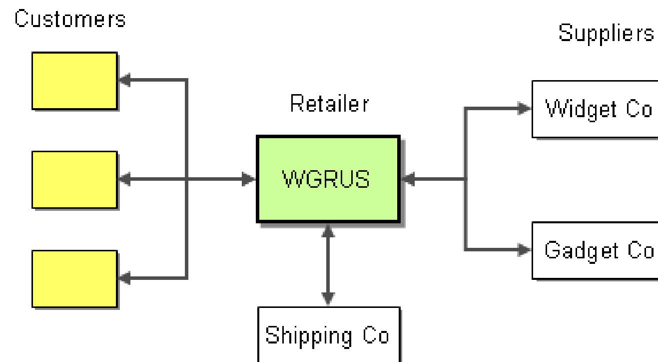
Messaging e integrazione di applicazioni

- Il messaging è oggi una tecnologia fondamentale nell’integrazione di applicazioni
  - l’integrazione avviene realizzando un’infrastruttura di comunicazione tra le applicazioni preesistenti, basata appunto sul messaging
  - il messaging viene spesso preferito ad altre tecnologie perché richiede un accoppiamento basso tra i componenti ed 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 [EIP]
  - http://www.enterpriseintegrationpatterns.com/Chapter1.html

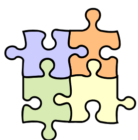


- Studio di caso [EIP]

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

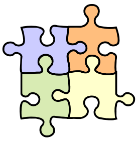


  - il sistema *WGRUS* deve integrare alcuni componenti (preesistenti) dei sistemi informatici (preesistenti) di Widget Co e Gadget Co



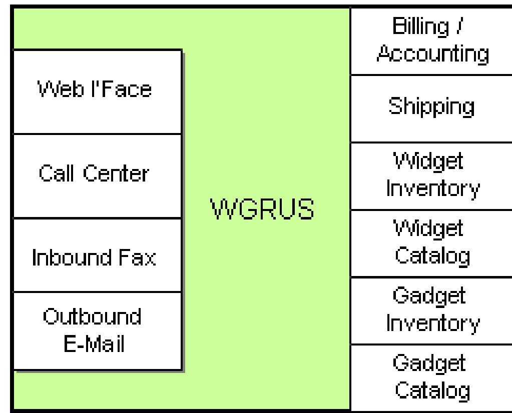
WGRUS

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

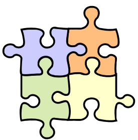


WGRUS come problema di integrazione

- 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

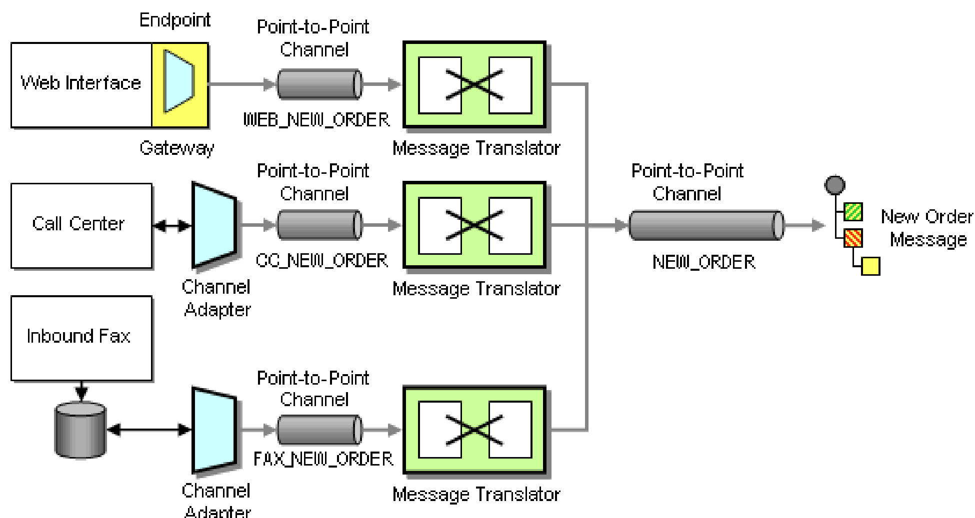


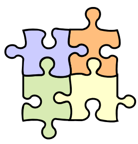
  - a sinistra, sono mostrati i canali di interazione con i clienti
  - a destra, i componenti applicativi preesistenti da riusare



- Ricezione di ordini

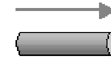
- Gli ordini possono essere ricevuti/immessi da vari client
  - un client web, un client per un addetto al telefono, ordini ricevuti via fax – ciascuno genera ordini con un formato diverso
  - si vuole invece avere un flusso di messaggi (unico e omogeneo) per tutti gli ordini



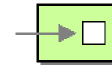


Pattern per l'EAI (1)

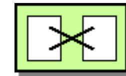
- Alcuni pattern per l'Enterprise Application Integration
  - *Message*
    - un messaggio – ovvero, un tipo/flusso di messaggi



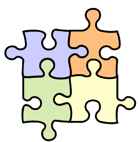
  - *Message* (*Point-to-Point*) *Channel*
    - un canale (una coda) per lo scambio di messaggi



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

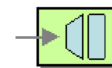


  - *Message Translator*
    - una trasformazione che cambia il formato di un messaggio

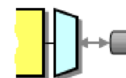


Pattern per l'EAI (2)

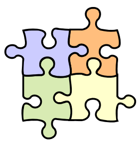
- Esistono vari tipi di *Message Endpoint* – tra cui



  - *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

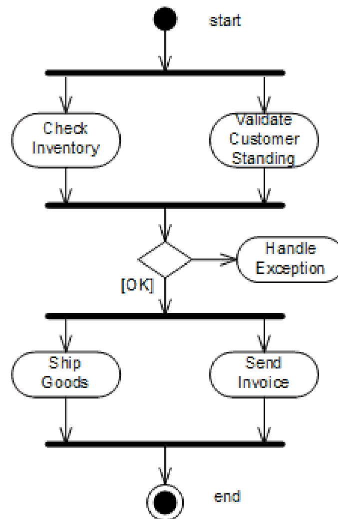


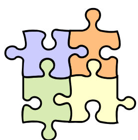
  - *Channel Adapter*
    - endpoint che realizza una connessione tra un'applicazione (di solito preesistente) e il sistema di messaging



- Elaborazione di ordini

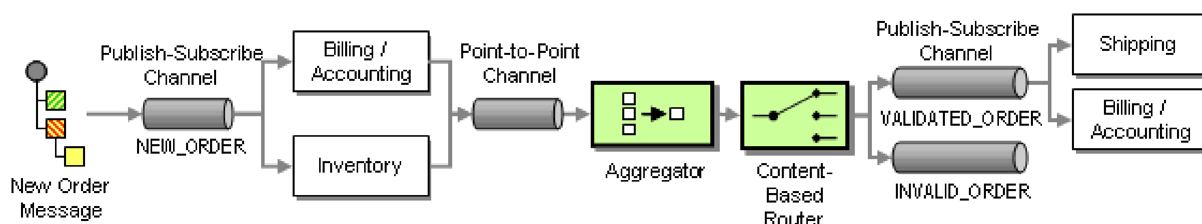
- Ora abbiamo un flusso consistente di ordini – l'elaborazione di un ordine richiede
  - verifica dello stato del cliente – nessun debito in sospeso
  - verifica dell'inventario – disponibilità degli articoli ordinati
  - se tutto ok, si può procedere

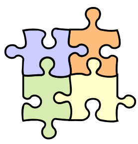




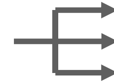
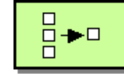
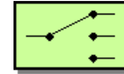
Elaborazione di ordini

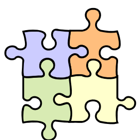
- Come elaborare gli ordini?
  - ordini inviati separatamente e in parallelo a contabilità e inventario per le verifiche
  - le due risposte devono poi essere aggregate
  - gli ordini confermati vanno poi inviati ai sistemi di spedizione e di fatturazione





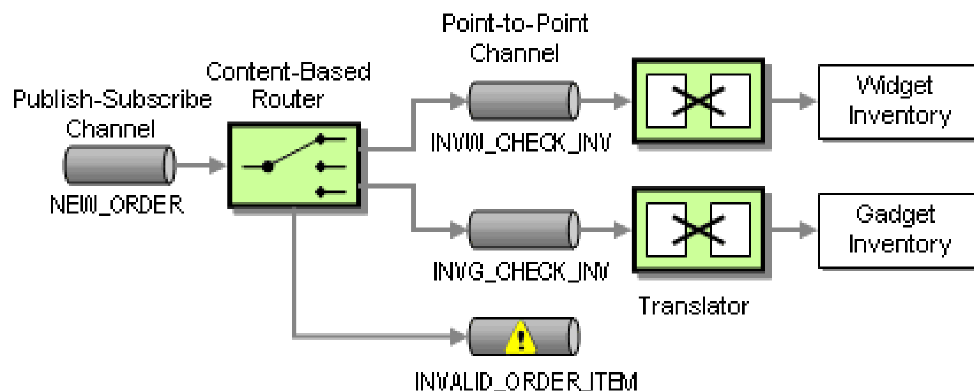
Pattern per l'EAI (3)

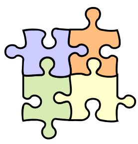
- *Publish-Subscribe Channel*
  
  - un canale (un topic/argomento) per lo scambio di messaggi
- *Aggregator*
  
  - combina il contenuto di messaggi diversi ma correlati
- *Content-Based Router*
  
  - gira un messaggio a un'opportuna destinazione, sulla base del contenuto del messaggio



- Controllo dell'inventario

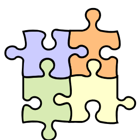
- In realtà, ci sono due sistemi/funzionalità per il controllo dell'inventario
  - una per i Widget e una per i Gadget
  - ciascuna richiesta va instradata al sistema giusto
    - ipotesi (temporanea): una sola riga d'ordine per ordine
    - ipotesi (semplificativa): il primo carattere del codice del prodotto è G o W





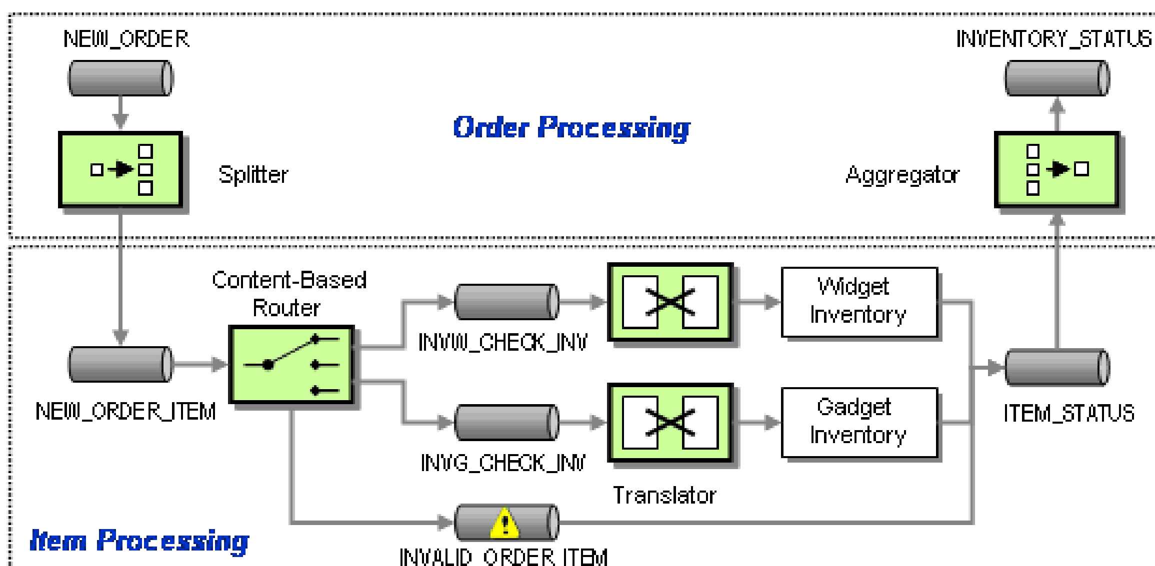
Pattern per l'EAI (4)

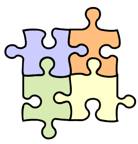
- *Invalid Message Channel*
  - destinazione di messaggi non validi



- Ordini con più righe d'ordine

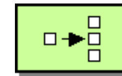
- In realtà, un ordine contiene normalmente più righe d'ordine
  - alcune saranno relative a widget, altre a gadget
  - la disponibilità delle merci va verificata riga d'ordine per riga d'ordine



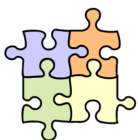


Pattern per l'EAI (5)

- *Splitter*

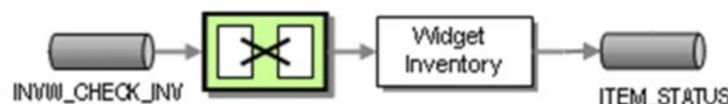


  - decompone un messaggio in un insieme di messaggi, ciascuno dei quali può richiedere (successivamente) una diversa elaborazione

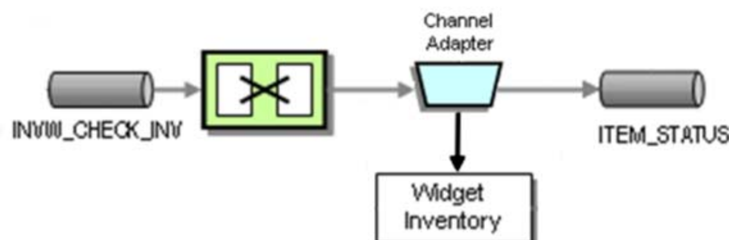


Un'osservazione

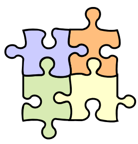
- Si consideri questa porzione del diagramma



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

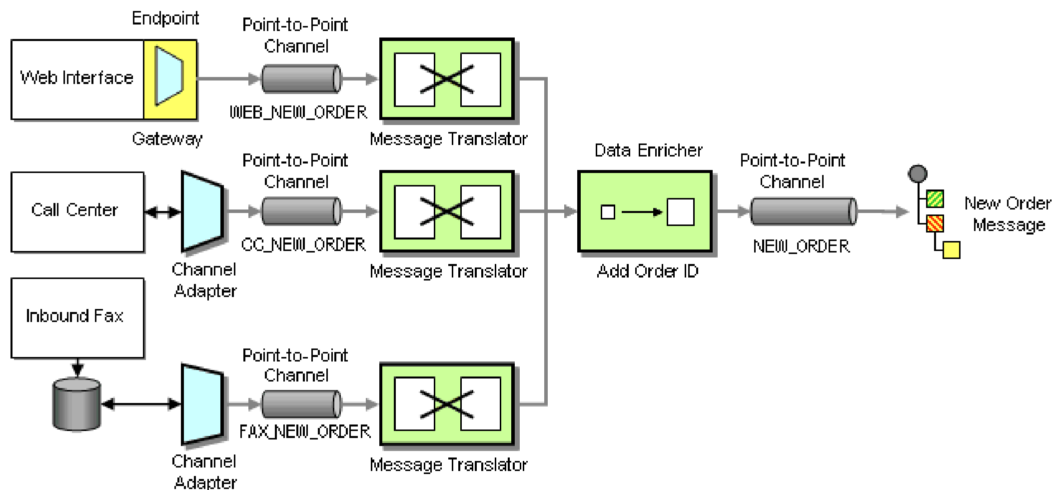


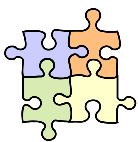
  - considerazioni analoghe si possono fare per accedere alle funzionalità di altri componenti preesistenti



- Identificatore d'ordine

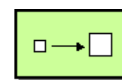
- Messaggi elaborati separatamente possono essere ricombinati mediante un Aggregator sulla base di opportune informazioni di correlazione
  - ad es., un identificatore d'ordine
  - ma è necessario aggiungere un identificatore a ciascun ordine

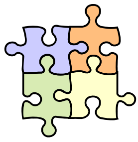




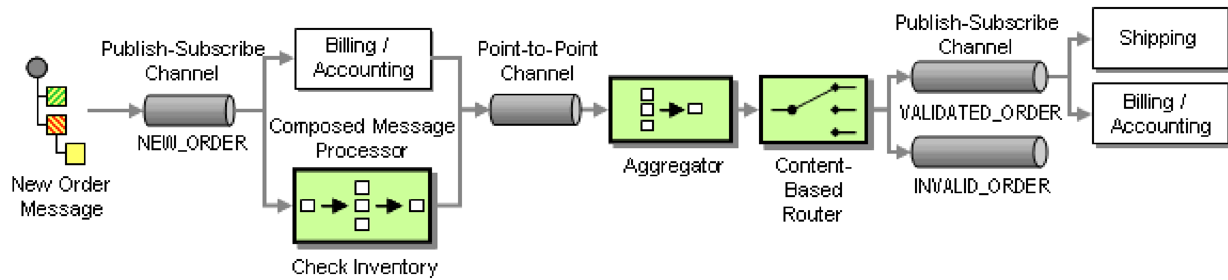
Pattern per l'EAI (6)

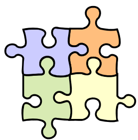
- *Content Enricher*
  - aggiunge informazioni a un messaggio



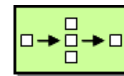


- Gestione degli ordini - rivista

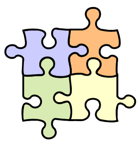




Pattern per l'EAI (7)

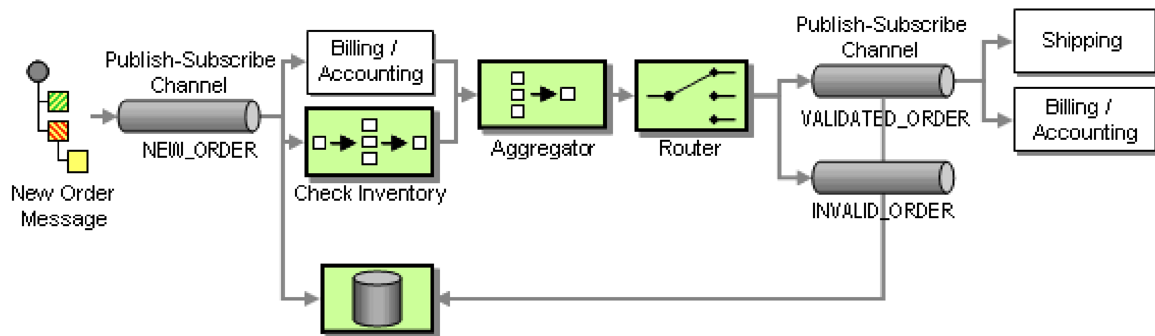


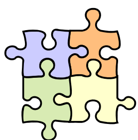
- *Composed Message Processor*
  - mantiene il flusso di messaggi complessivo, anche se i diversi messaggi richiedono elaborazioni diverse



- Verificare lo stato di un ordine

- L'elaborazione di un ordine richiede lo svolgimento di varie attività
  - come è possibile verificare lo stato di avanzamento di un ordine? è stata effettuata la spedizione? è in attesa di prodotti? è 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

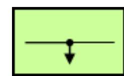




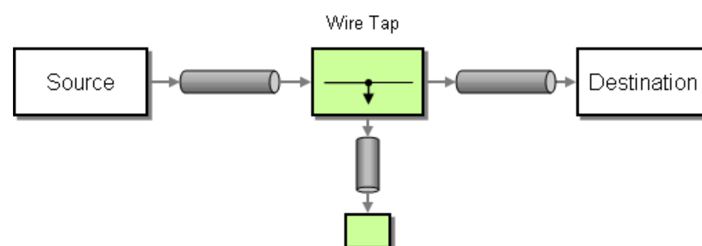
Pattern per l'EAI (8)

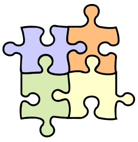
- *Message Store*

  - 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*



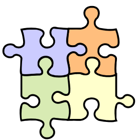
  - per duplicare su più canali i messaggi inviati su un certo canale





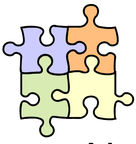
- Discussione

- 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 dei nuovi componenti (message endpoint/ channel adapter) – questi componenti incapsulano l'accesso al sistema di messaging, e fungono da "collante" tra i componenti preesistenti
- Il messaging può essere utilizzato anche per lo sviluppo di nuove applicazioni



Discussione

- Oltre a quelli visti, [EIP] 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*



Discussione

- Nell'esempio relativo al sistema WGRUS – ma questo è vero in generale nei sistemi basati sul messaging – è possibile vedere l'applicazione di due stili architetturali fondamentali
  - *Domain Model* – il modello di dominio utilizzato in questo caso è un modello delle attività
    - il processo di gestione degli ordini richiede di svolgere un certo numero di attività, in un certo ordine
    - si può notare come l'architettura del sistema integrato per la gestione degli ordini abbia proprio la forma del diagramma delle attività che è stato mostrato
  - *Pipes and Filters* – i sistemi di messaging sono chiaramente basati su questo stile
    - il processo di gestione degli ordini deve trasformare un flusso di messaggi in ingresso (ordini) in flussi di messaggi in uscita (spedizioni, fatture)
    - la trasformazione è stata decomposta in una sequenza di passi successivi – guidati dal diagramma delle attività