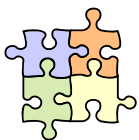


# Introduzione ai connettori e al middleware

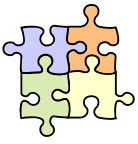
Dispensa ASW 810

ottobre 2011



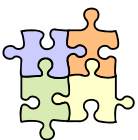
## - Fonti

- [TMD] Software Architecture – Foundations, Theory, and Practice, Chapter 5, Connectors
- [Shaw, 1994] Procedure Calls are the Assembly Language of Software Interconnections: Connectors Deserve First-Class Status, Technical report CMU/SEI-1994-TR-2
- [Bernstein] Middleware, Communications of the ACM, 1996



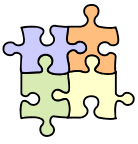
## - Obiettivi e argomenti

- Obiettivi
  - introdurre e motivare i connettori
  - introdurre il middleware
  
- Argomenti
  - introduzione ai connettori
  - middleware
  - discussione



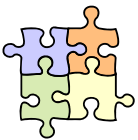
## \* Introduzione ai connettori

- In un'architettura software è possibile distinguere due tipi principali di elementi software
  - *componenti*
    - elementi responsabili dell'implementazione di *funzionalità* e della gestione di *dati/informazioni*
  - *connettori*
    - elementi responsabili delle *interazioni* tra componenti – i connettori caratterizzano assemblaggio e integrazione di componenti
  
- Si tratta di un'applicazione del principio di separazione degli interessi
  - riflette la sostanziale indipendenza tra gli aspetti funzionali e quelli relativi alle interazioni



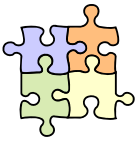
## Connettori

- L'esperienza ha mostrato che
  - la scelta e progettazione dei connettori (ovvero, delle interazioni) è importante tanto quanto quella dei componenti
    - ad es., è fondamentale nella realizzazione di sistemi basati sull'integrazione di componenti software pre-esistenti
  - la progettazione dei connettori può essere fatta separatamente da quella dei componenti
    - sostenendo così una separazione tra gli interessi di computazione e quelli di interazione
- Inoltre, i connettori sono tipicamente indipendenti dalle applicazioni – diversamente dai componenti, che forniscono servizi specifici per un'applicazione
  - questo ha portato allo sviluppo di numerosi strumenti di middleware – tecnologie software per l'implementazione di connettori, utili soprattutto nello sviluppo di sistemi distribuiti



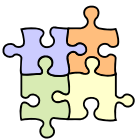
## Componenti e connettori - esempi

- Alcuni esempi di componenti
  - un modulo – ovvero, un pezzo di codice
    - un elemento software di natura statica
  - un processo – ovvero, un modulo in esecuzione
    - un elemento software di natura dinamica
  - una base di dati – caratterizzata dal suo schema
    - un elemento “data”
- Alcuni esempi di connettori
  - una chiamata di procedura tra moduli
  - una chiamata di procedura remota, una pipe, oppure un protocollo che regola lo scambio di messaggi tra due processi
  - l'accesso ad una base di dati da parte di un processo



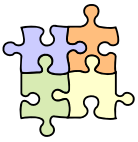
## Componenti e connettori

- Secondo [Shaw, 1994]
  - i **componenti** sono il luogo della computazione e dello stato
    - ogni componente ha una **specifica di interfaccia** che definisce le sue proprietà (sia funzionalità che proprietà di qualità, ad esempio circa le prestazioni)
    - ogni componente è di un qualche tipo – ad es., filtro, server, memoria, ...
    - l'interfaccia di un componente comprende la specifica dei “ruoli” (chiamati **player**) che un componente può rivestire nell'interazione con altri componenti



## Componenti e connettori

- Inoltre, secondo [Shaw, 1994]
  - i **connettori** sono il luogo delle relazioni tra componenti
    - i connettori sono mediatori di interazioni – sono “ganci” tra componenti
    - ogni connettore ha una **specifica di protocollo** che definisce le sue proprietà – queste proprietà comprendono regole sul tipo di interfacce che è in grado di mediare, nonché impegni sulle proprietà dell'interazione, come ad es. affidabilità, prestazioni e ordine in cui le cose avvengono
    - ogni connettore è di un qualche tipo – ad es., chiamata di procedura remota, pipe, evento, broadcast, ...
    - il protocollo di un connettore comprende la specifica dei **ruoli** che devono essere soddisfatti – ad es., client e server
  - la composizione dei componenti avviene mettendo in relazione player di componenti con ruoli di connettori



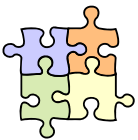
## Componenti e connettori

- Alcuni motivi, secondo [Shaw, 1994], per cui è opportuno trattare i connettori separatamente dai componenti
  - i connettori possono essere piuttosto sofisticati – con definizioni elaborate e specifiche complesse
  - la definizione di un connettore dovrebbe essere localizzata
  - alcune informazioni (scelte architetturali) del sistema non hanno una collocazione naturale in nessuno dei suoi componenti
  - i connettori sono potenzialmente astratti – e riutilizzabili in più contesti
  - i connettori possono richiedere un supporto distribuito
  - i componenti dovrebbero essere indipendenti
  - i connettori dovrebbero essere indipendenti
  - le relazioni tra componenti non sono fissate
  - sistemi diversi riusano spesso degli stessi pattern di composizione

9

Introduzione ai connettori e al middleware

Luca Cabibbo – ASw



## - Un esempio

- Si supponga di aver identificato, nell'ambito di una vista funzionale, due elementi "componenti"
  - un elemento **Servant** in grado di fornire un certo servizio **Service**
  - un elemento **Client**, che richiede l'erogazione di **Service** da parte del **Servant**

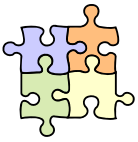


- si dice che **Service** è un'*interfaccia fornita* da **Servant** – e che **Service** è un'*interfaccia richiesta* da **Client**
- A che cosa corrisponde/può corrispondere ciò nel codice?

10

Introduzione ai connettori e al middleware

Luca Cabibbo – ASw

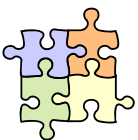


## Un esempio

- Supponiamo, per semplicità, di avere la seguente definizione del **Service**

- in effetti, limitata ai soli aspetti “funzionali” del servizio

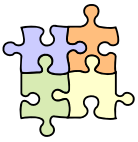
```
package asw.asw810.service;  
  
/* interfaccia del servizio */  
public interface Service {  
    public String alpha(String arg);  
}
```



## Un esempio

- Inoltre, supponiamo di avere la seguente definizione del **Servant**

```
package asw.asw810.server;  
  
import asw.asw810.service.Service;  
  
/* implementazione del servizio */  
public class Servant implements Service {  
    public String alpha(String arg) { ... fa qualcosa ... }  
}
```



## Un esempio

- Inoltre, supponiamo che il contesto – sempre limitato ai soli aspetti funzionali – dell’uso del **Service** da parte di **Client** sia il seguente

```
package asw.asw810.client;  
  
import asw.asw810.service.Service;  
  
/* client del servizio */  
public class Client {  
    ...  
    private Service service;  
  
    public void run(...) {  
        ...  
        ... service.alpha(...) ...  
        ...  
    }  
    ...  
}
```

in **rosso**  
indichiamo  
l’invocazione del  
servizio – è  
ancora un aspetto  
funzionale

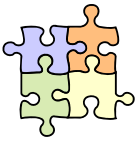


## Un esempio

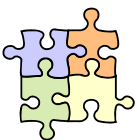
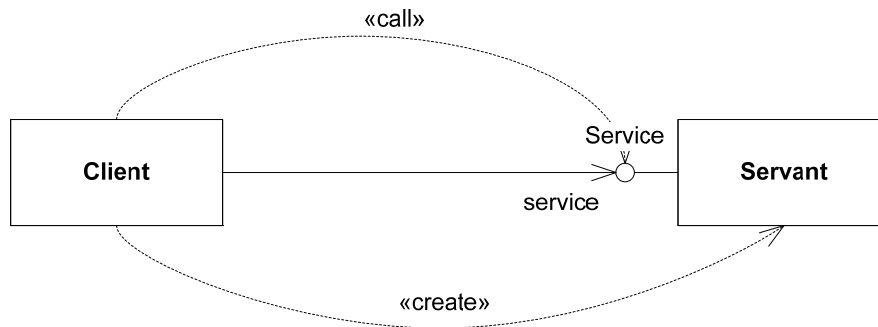
- Il connettore più semplice è la **chiamata di procedura** – o **invocazione di metodo** nei linguaggi OO

```
package asw.asw810.client;  
  
import asw.asw810.service.Service;  
import asw.asw810.server.Servant;  
  
/* client del servizio */  
public class Client {  
    ...  
    private Service service = new Servant();  
  
    public void run(...) {  
        ...  
        ... service.alpha(...) ...  
        ...  
    }  
    ...  
}
```

in **nero**  
indichiamo il  
codice relativo al  
connettore –  
ovvero,  
all’interazione  
tra componenti

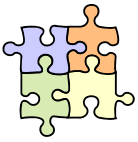


## Un esempio



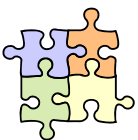
## Un esempio

- Caratteristiche dell'uso che è stato appena fatto della chiamata di procedura
  - una *chiamata di procedura locale*
    - **Client** e **Servant** vivono nello stesso processo – tuttavia, le scelte funzionali dovrebbero essere indipendenti da scelte relative, ad es., a concorrenza e deployment
  - una chiamata sincrona
    - durante l'esecuzione del **Service**, il **Client** rimane in attesa del **Servant** – non viene sfruttata un'eventuale concorrenza
  - il **Client** è accoppiato alla particolare implementazione del **Service** offerta dal **Servant**
    - non c'è indipendenza dall'implementazione del servizio
  - **Client** e **Servant** devono essere scritti nello stesso linguaggio di programmazione
    - anche questo potrebbe essere un vincolo indesiderato



## - Un (piccolo) passo avanti

- Concentriamoci, per ora, sull'eliminazione della dipendenza tra il **Client** e la particolare implementazione **Servant** del **Service**
  - si può rompere questa dipendenza utilizzando degli oggetti di supporto – in particolare, applicando degli opportuni design pattern
    - ad esempio, usando una factory – ovvero, un singleton che (per ora) si occupa (solo) della creazione del **Servant**



## Un (piccolo) passo avanti

- La **ServiceFactory** incapsula la creazione del **Servant**

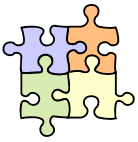
```
package asw.asw810.client.connector;

import asw.asw810.service.Service;
import asw.asw810.server.Servant;

/* factory per il servizio */
public class ServiceFactory {
    private static ServiceFactory instance = null; // singleton per la factory
    private Service service = null; // singleton per il servizio

    public static synchronized ServiceFactory getInstance() {
        if (instance==null) instance = new ServiceFactory();
        return instance;
    }

    public synchronized Service getService() {
        if (service==null) service = new Servant();
        return service;
    }
}
```



## Un (piccolo) passo avanti

- Ecco la nuova versione per il client – il connettore sarà ancora una chiamata di procedura

```
package asw.asw810.client;
```

```
import asw.asw810.service.Service;  
import asw.asw810.client.connector.*;
```

```
/* client del servizio */
```

```
public class Client {
```

```
...
```

```
private Service service = ServiceFactory.getInstance().getService();
```

```
public void run(...) {
```

```
...
```

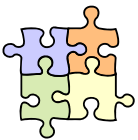
```
... service.alpha(...) ...
```

```
...
```

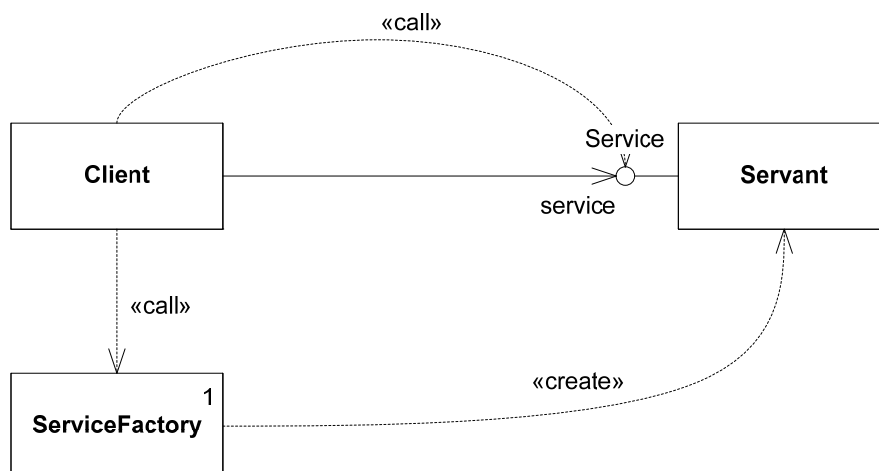
```
}
```

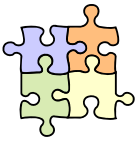
```
...
```

```
}
```



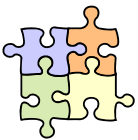
## Un (piccolo) passo avanti





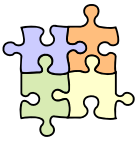
## Un (piccolo) passo avanti

- Alcune caratteristiche della nuova soluzione
  - in pratica, si tratta ancora di una chiamata di procedura locale e di una chiamata sincrona
  - tuttavia, non c'è più l'accoppiamento diretto tra il **Client** e la particolare implementazione del **Service** offerta dal **Servant**
    - o meglio, l'accoppiamento è localizzato nella factory – che consideriamo parte del connettore
    - l'accoppiamento può essere ulteriormente ridotto usando per la factory un progetto data-driven
      - ad es., memorizzando in un file di configurazione il nome della classe che implementa il servizio che si vuole utilizzare – è possibile cambiare questa scelta senza modificare né il client né il connettore
  - c'è ancora il vincolo che **Client** e **Servant** devono essere scritti nello stesso linguaggio di programmazione



## - Un altro (piccolo) passo avanti

- Nella soluzione precedente, la factory crea un'istanza del servente e ne restituisce un riferimento al client
  - come vedremo più avanti, è invece spesso comune (e utile) che la factory restituisca un altro intermediario, un **proxy** – usato come rappresentante del server presso il client
  - il proxy, per ora, gestisce solo un riferimento al vero servente



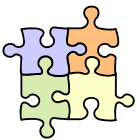
## Un altro (piccolo) passo avanti

- Il **ServiceProxy** è un'indirezione verso il vero servizio, ed implementa la stessa interfaccia del servizio

```
package asw.asw810.client.connector;

import asw.asw810.service.Service;

public class ServiceProxy implements Service {
    /* il vero servizio */
    private Service service;
    public ServiceProxy(Service service) {
        this.service = service;
    }
    /* questo è proprio il metodo alpha invocato dal client
    * (anche se il client pensa di parlare direttamente con il servant) */
    public String alpha(String arg) {
        /* chiama il vero servizio */
        return service.alpha(arg);
    }
}
```



## Un altro (piccolo) passo avanti

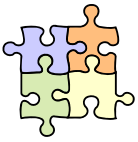
- La **ServiceFactory** crea il **Servant** (se necessario) – ma restituisce un **ServiceProxy**

```
package asw.asw810.client.connector;

import asw.asw810.service.Service;
import asw.asw810.server.Servant;

/* factory per il servizio */
public class ServiceFactory {
    ... metodi e variabile per singleton ...

    /* factory per il servizio */
    public synchronized Service getService() {
        if (service==null) service = new Servant();
        return new ServiceProxy( service );
    }
}
```



## Un altro (piccolo) passo avanti

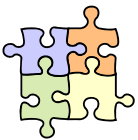
- Il client non cambia – ed il connettore è ancora, a tutti gli effetti, una chiamata di procedura locale

```
package asw.asw810.client;
```

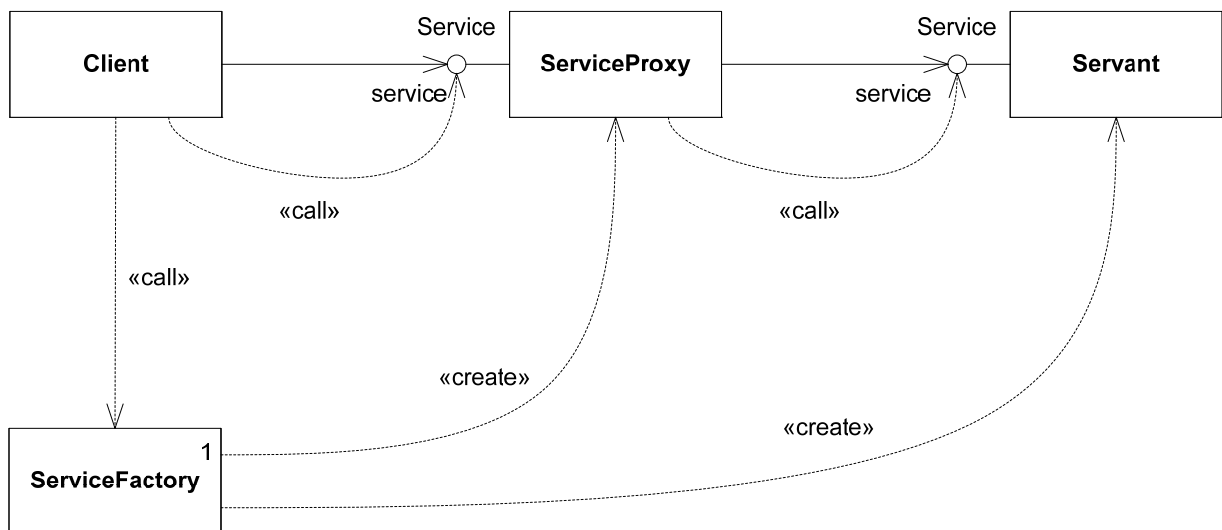
```
import asw.asw810.service.Service;  
import asw.asw810.client.connector.*;
```

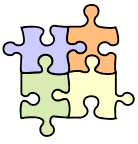
```
/* client del servizio */  
public class Client {
```

```
    ...  
    private Service service = ServiceFactory.getInstance().getService();  
  
    public void run(...) {  
        ...  
        ... service.alpha(...) ...  
        ...  
    }  
    ...  
}
```



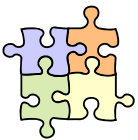
## Un altro (piccolo) passo avanti





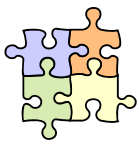
## Un altro (piccolo) passo avanti

- Che cosa è cambiato?
  - in pratica, sembra tutto come prima – ovvero, sembra che abbiamo “cambiato tutto, per non cambiare nulla”
- Ora però abbiamo un elemento (il proxy) a cui possiamo dare delle responsabilità accessorie “da connettore”
  - ad esempio, se si vuole effettuare il logging degli accessi al servizio, le istruzioni per la gestione del logging possono essere localizzate nel proxy
    - senza cambiare né client né servente – ovvero, gli aspetti funzionali
  - oppure, se la computazione eseguita dal servente è onerosa, possiamo fare caching delle richieste e delle risposte
    - aggiungendo codice nel proxy
    - senza cambiare né client né servente



## - Un altro passo avanti (un po' più grande)

- Concentriamoci ora sull'eliminazione della co-localizzazione tra il **Client** ed il **Servant**
  - è possibile gestire una comunicazione remota tra client e server usando un meccanismo di comunicazione interprocesso
    - ad es., le socket – un meccanismo di comunicazione interprocesso, fornito dal sistema operativo, basato sullo scambio di messaggi – datagrammi, con UDP – oppure su un canale di comunicazione bidirezionale – con TCP
  - la comunicazione può essere separata dagli aspetti funzionali tramite una coppia di remote proxy
    - un remote proxy lato client – un intermediario che si occupa dell'interazione remota tra **Client** e **Servant**, che vive nello stesso processo del **Client**
    - un remote proxy lato server – un altro intermediario che si occupa dell'interazione remota, nel processo del **Servant**



## Un altro passo avanti (un po' più grande)

- Adottando le socket UDP, client e server possono comunicare scambiandosi messaggi – chiamati datagrammi – come segue
  - il client effettua una richiesta
  - il client forma un messaggio che codifica la sua richiesta (marshalling della richiesta) – il nome dell'operazione e l'elenco dei parametri – e invia questo messaggio al server
  - il server riceve il messaggio di richiesta e lo decodifica (unmarshalling della richiesta) – identificando il nome dell'operazione e l'elenco dei parametri
  - il server esegue l'operazione richiesta, generando una risposta
  - il server forma un messaggio che codifica la risposta (marshalling della risposta) – e lo invia al client
  - il client riceve il messaggio di risposta e lo decodifica (unmarshalling della risposta)
  - il client ha ricevuto la risposta alla sua richiesta



## Un altro passo avanti (un po' più grande)

- Il **ServiceClientProxy** è un “remote proxy” lato client, con la seguente struttura

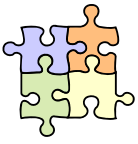
```
package asw.asw810.client.connector;

import asw.asw810.service.Service;
import java.net.*; // per le socket

/* remote proxy lato client per il servizio */
public class ServiceClientProxy implements Service {
    private InetAddress address; // indirizzo del server
    private int port;           // porta per il servizio

    public ServiceClientProxy(InetAddress address, int port) {
        this.address = address;  this.port = port;
    }

    public String alpha(String arg) {
        ... segue ...
    }
}
```



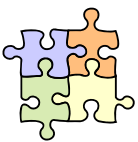
## Un altro passo avanti (un po' più grande)

- Il metodo **alpha** del “remote proxy” lato client

```
/* questo è proprio il metodo alpha invocato dal client
 * (anche se il client pensa di parlare direttamente con il servant) */
public String alpha(String arg) {

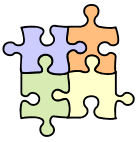
    ... crea un datagramma che codifica la richiesta di servizio
    ed i relativi parametri ...
    ... invia il datagramma di richiesta ...
    ... ricevi il datagramma di risposta ...
    ... estrai la risposta dal datagramma di risposta ...
    return reply;

}
```



## Un altro passo avanti (un po' più grande)

- La **ServiceFactory** può essere usata per incapsulare la creazione del proxy lato client
  - si può usare un approccio data-driven per quanto riguarda, ad es., l'indirizzo di rete e la porta del server
    - ma anche per specificare il tipo del proxy da creare – se sono disponibili più proxy relativi a modalità di connessione diverse
  - questa factory non si può più occupare della creazione del vero **Servant** – che in generale vive in un altro processo
- Il **Client** ed il **Servant** possono rimanere ancora immutati rispetto alla versione precedente



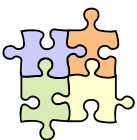
## Un altro passo avanti (un po' più grande)

```
package asw.asw810.client.connector;

import asw.asw810.service.Service;
import java.net.*;

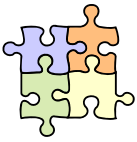
/* factory per il servizio */
public class ServiceFactory {
    ... variabile e metodo per singleton, come prima ...

    /* factory per il servizio */
    public Service getService() {
        InetAddress address = InetAddress.getByName("localhost");
        int port = 6789;
        Service service = new ServiceClientProxy(address, port);
        return service;
    }
}
```



## Un altro passo avanti (un po' più grande)

- Il “remote proxy” lato client non comunica direttamente con il **Servant**
  - piuttosto, è necessario un ulteriore intermediario – un “remote proxy” lato server
  - il “remote proxy” lato server
    - riceve richieste – tramite socket – dal proxy lato client, e le gira al **Servant**
    - riceve risposte dal **Servant** – e le gira al proxy lato client



## Un altro passo avanti (un po' più grande)

### □ Struttura del “remote proxy” lato server

```
package asw.asw810.server.connector;

import asw.asw810.service.Service;
import asw.asw810.server.Servant;
import java.net.*;

/* remote proxy lato server per il servizio */
public class ServiceServerProxy {
    private Service service;          // il vero servizio
    private int port;                 // porta per il servizio

    public ServiceServerProxy(Service service, int port) {
        this.service = service;    this.port = port;
    }

    public void run() {
        ... segue ...
    }
}
```

35

Introduzione ai connettori e al middleware

Luca Cabibbo – ASw



## Un altro passo avanti (un po' più grande)

### □ Il metodo *run* del “remote proxy” lato server

- per semplicità, un server “sequenziale” – anche in questo caso di potrebbe fare di meglio

```
public void run() {

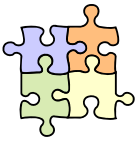
    ... crea la socket su cui ricevere le richieste ...
    while (true) {
        ... aspetta un datagramma di richiesta ...
        ... estrai la richiesta dal datagramma di richiesta ...
        ... chiedi l'erogazione del servizio ed ottieni la risposta ...
        ... crea il datagramma di risposta ...
        ... invia il datagramma di risposta ...
    }

}
```

36

Introduzione ai connettori e al middleware

Luca Cabibbo – ASw



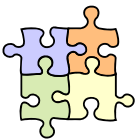
## Un altro passo avanti (un po' più grande)

- Lato server, manca ancora un oggetto **Server** responsabile
  - della creazione del **Servant**
  - della creazione e dell'avvio del proxy lato server

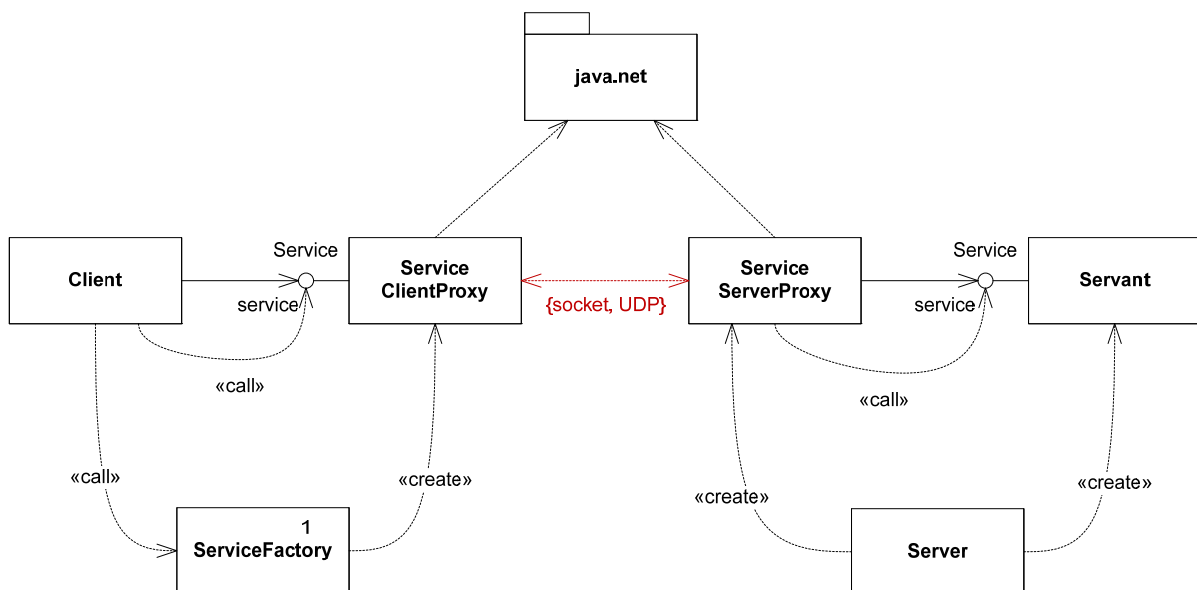
```
package asw.asw810.server.connector;
```

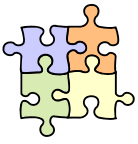
```
import asw.asw810.service.Service;  
import asw.asw810.server.Servant;
```

```
/* server per il servizio */  
public class Server {  
    public static void main(String[] args) {  
        Service service = new Servant();  
        int port = 6789;  
        ServiceServerProxy server = new ServiceServerProxy(service, port);  
        server.run();  
    }  
}
```



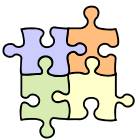
## Un altro passo avanti (un po' più grande)





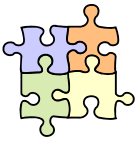
## Un altro passo avanti (un po' più grande)

- Alcune caratteristiche della nuova soluzione
  - si tratta ancora di una chiamata di procedura
    - è però una *chiamata di procedura remota*
  - si tratta ancora di una chiamata sincrona
  - l'accoppiamento tra il **Client** e il **Servant** è localizzato nel connettore – che comprende diverse classi
    - attenzione, a livello di codice, non c'è più nessun accoppiamento diretto, se non tramite l'interfaccia **Service**
    - l'accoppiamento – localizzato nel connettore – è finito nel protocollo di comunicazione adottato per la comunicazione remota
  - in linea di principio, il **Client** ed il **Servant** potrebbero essere scritti con linguaggi di programmazione diversi, in esecuzione in ambienti hw/sw differenti
    - grazie al supporto “universale” delle socket



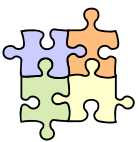
## Discussione

- L'esempio ha mostrato la comunicazione – locale e remota – tra due componenti
  - scopo dell'esempio era soprattutto mostrare che gli aspetti funzionali (codice “blu” e “rosso”) possono – ed in genere devono – essere considerati separatamente da quelli relativi all'interazione/comunicazione (codice “nero”)
  - nello specifico, l'esempio ha mostrato (anche se in modo parziale) l'uso delle socket come libreria per realizzare connettori nel contesto dei sistemi distribuiti
    - ma, come vedremo, spesso si utilizzano soluzioni basate su librerie più ricche per l'implementazione dei connettori – il cosiddetto middleware



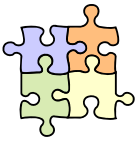
## Discussione

- Alcune domande relative all'esempio visto
  - come realizzare il logging degli accessi al servizio?
    - questi aspetti possono essere gestiti a livello del connettore
  - come rendere confidenziale lo scambio di messaggi in rete – in particolare, sulla base di meccanismi di cifratura?
    - questi aspetti possono essere gestiti a livello del connettore
  - come realizzare un meccanismo di autenticazione ed autorizzazioni basato su id e password?
    - questi aspetti possono essere gestiti a livello del connettore
  - che cosa fare, ad es., se si verifica un guasto nella rete o si perde un datagramma UDP? usare socket TCP è una soluzione o no?
    - questi aspetti possono essere gestiti a livello del connettore



## Verso il middleware

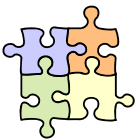
- Meccanismi di base come le socket possono essere considerate l'“assembler” per lo sviluppo dei connettori
  - usando questo “assembler” ed altre librerie di base è possibile realizzare connettori molto complessi – che gestiscono qualità complesse del software – separatamente dai componenti, che si possono così occupare dei soli aspetti funzionali
- In pratica, diversi connettori di uso comune sono stati generalizzati – definendo la classe degli strumenti di middleware
  - un insieme di tecnologie che offre una molteplicità di paradigmi di interazione tra componenti – sostenendo e semplificando la realizzazione dei connettori



## Verso il middleware

### □ Verso il middleware

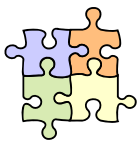
- i connettori realizzati “a mano” sono complessi da sviluppare e da mantenere
- inoltre, diversi connettori di uso comune sono stati generalizzati – e realizzati sulla base di opportuni “generatori di codice”
- in pratica
  - il programmatore “lato server” scrive l’interfaccia del servizio e ne implementa il servente
  - il programmatore “lato client” utilizza il servizio tramite un proxy (di solito generato) da una factory
  - tutto il codice del connettore (compresi proxy e factory) viene generato automaticamente



## \* Middleware

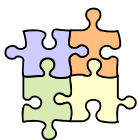
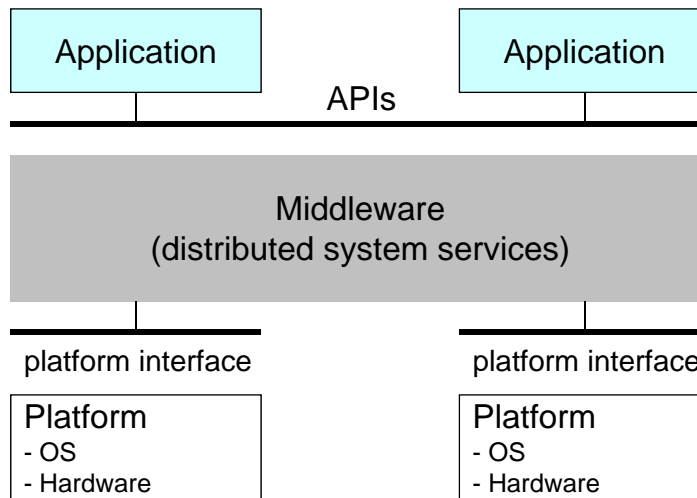
### □ Il **middleware** è [Bakken]

- una classe di tecnologie software sviluppate per aiutare gli sviluppatori nella gestione della complessità e della eterogeneità presenti nei sistemi distribuiti
- uno strato software “in mezzo”
  - sopra al sistema operativo, ma sotto i programmi applicativi
  - fornisce un’astrazione di programmazione distribuita – un modello computazionale uniforme
  - per mascherare le eterogeneità di elementi sottostanti – reti, hardware, sistemi operativi, linguaggi di programmazione, ...
- Il middleware ha lo scopo di sostenere lo sviluppo di elementi utili – i connettori – per costruire componenti software che possano lavorare insieme ad altri in un sistema distribuito



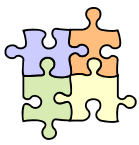
# Middleware

- Un **servizio di middleware** è un servizio general-purpose che si colloca tra piattaforme e applicazioni [Bernstein]



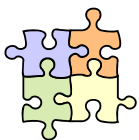
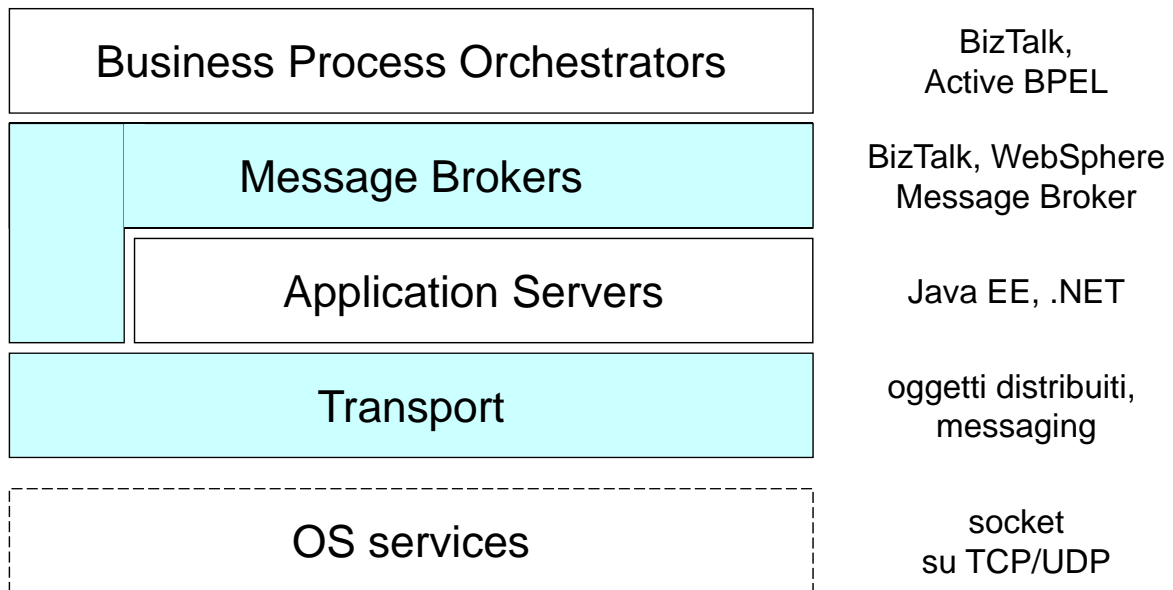
# Middleware - esempi

- Alcuni esempi di (famiglie di) prodotti di middleware
  - RPC
  - Java RMI
  - Messaging
  - SQL Stored Procedures
  - ORB
  - TP monitor
  - EJB e servlet
  - Web Services
  - ...



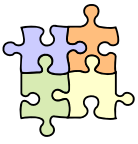
## Middleware - esempi

- Una possibile classificazione delle tecnologie di middleware [Gorton]



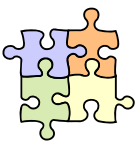
## Middleware

- Il middleware è il software che sostiene il collegamento (plumbing/piping/wiring) tra componenti software – e rende facilmente programmabili i sistemi distribuiti
  - ciascuno strumento di middleware offre una specifica modalità di interazione
    - ad es., RPC offre un paradigma di interazione basato sulla chiamata (sincrona) di procedure remote
    - il messaging, invece, offre un paradigma di programmazione basata sullo scambio (asincrono) di messaggi
  - sulla base di meccanismi di programmazione e API relativamente semplici



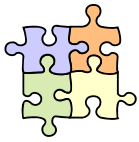
## Middleware e stili architetturali

- Relazioni tra strumenti di middleware e stili architetturali
  - l'applicazione di alcuni stili architetturali è sostenuta, dal punto di vista tecnologico, da opportuni strumenti di middleware
    - ad es., lo stile C/S può essere basato su RPC, lo stile C/S a più livelli sugli application server (AS) e middleware a componenti, le SOA sui WS...
  - altri stili architetturali, invece, descrivono l'architettura di alcuni strumenti di middleware
    - ad es., RMI è basato su Broker, gli AS su Container
- E' chiaramente utile conoscere e comprendere queste relazioni
  - per capire come quale middleware utilizzare per realizzare un certo stile architetturale – e per capire come utilizzare al meglio un certo middleware
  - per comprendere il funzionamento del middleware – e per realizzare (se serve) nuovi tipi di connettori



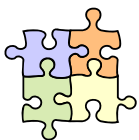
## Middleware e trasparenza

- Ciascuno strumento di middleware ha lo scopo di mascherare qualche tipo di eterogeneità comunemente presente in un sistema distribuito
  - il middleware maschera sempre l'eterogeneità delle reti e dell'hardware
  - alcuni strumenti di middleware mascherano eterogeneità nel sistema operativo e/o nei linguaggi di programmazione
  - alcuni strumenti di middleware mascherano eterogeneità nelle diverse implementazioni di uno stesso standard di middleware
    - ad es., alcune implementazione di Corba o Java EE
  - le astrazioni di programmazione offerte dal middleware possono fornire trasparenza rispetto ai seguenti aspetti
    - posizione, concorrenza, replicazione, fallimenti, mobilità
  - in alternativa, il programmatore dovrebbe farsi esplicitamente carico di queste eterogeneità e di questi aspetti



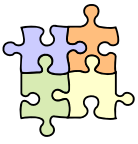
## - Discussione

- Malgrado i molti benefici offerti dal middleware, il middleware non è una panacea per i sistemi distribuiti
  - il middleware non può risolvere magicamente i problemi derivanti da decisioni di progetto povere
    - che possono avere conseguenze negative su stabilità, prestazioni e scalabilità
    - ad esempio, le applicazioni distribuite devono essere comunemente preparate a gestire i fallimenti della rete ed i guasti nei server
  - inoltre il middleware è focalizzato solo sulla comunicazione tra componenti
    - le responsabilità di natura applicativa sono completamente fuori dalla sua portata



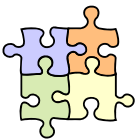
## Discussione

- Se utilizzato in modo opportuno, il middleware consente di affrontare diverse problematiche significative per i sistemi distribuiti
  - consente di generare automaticamente tutto (o quasi) il codice per i connettori
  - in questo modo, consente di concentrarsi sullo sviluppo della logica applicativa – anziché sui dettagli della comunicazione e della piattaforma hardware/software utilizzata
- Tuttavia, per aumentare effettivamente la produttività, il middleware deve essere scelto e utilizzato in modo corretto
  - la decisione del middleware da utilizzare richiede delle considerazioni e delle decisioni esplicite
  - è inoltre necessaria una buona comprensione del paradigma di comunicazione implementato dal middleware, nonché della sua struttura e dei suoi principi di funzionamento



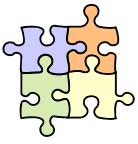
## \* Discussione

- Che cosa vedremo in questo corso
  - le dispense 8xx introducono alcune tecnologie di middleware
  - delle varie tecnologie, vedremo solo alcuni esempi di uso, peraltro piuttosto semplici
    - lo scopo è introdurre alcune tecnologie rappresentative, alcune delle loro finalità, alcuni dei problemi affrontati e risolti, alcuni dei problemi non risolti o che devono essere presi in considerazione dal programmatore
  - cercheremo di capire alcune funzionalità offerte esternamente da alcuni strumenti di middleware
    - talvolta cercheremo anche di comprendere la modalità di funzionamento e/o la struttura interna di questi strumenti
  - gli aspetti più metodologici – circa l'uso dei vari paradigmi di comunicazione e tipi di connettori – sono invece affrontati soprattutto dalle dispense 4xx



## Discussione

- Che cosa vedremo in questo corso
  - socket
    - meccanismo di base della IPC
    - non è uno strumento di middleware
    - per intuire alcune problematiche affrontate dal middleware
  - oggetti distribuiti
    - versione a oggetti della chiamata di procedura remota
  - messaging
    - comunicazione asincrona basata sullo scambio di messaggi
  - application server
    - componenti e contenitori
  - web services
    - servizi interoperabili, sincroni e asincroni
  - cloud computing



## Discussione

- Che cosa non vedremo in questo corso
  - non vedremo tutte le possibili tipologie di middleware
  - non vedremo molti strumenti
  - non vedremo strumenti commerciali
    - di solito molto più efficaci di quelli open source
  - non vedremo esempi complessi
  - non vedremo aspetti metodologici o pattern specifici per le varie tecnologie
  - non vedremo soluzioni tecnologiche complete circa l'applicazione delle varie metodologie generali