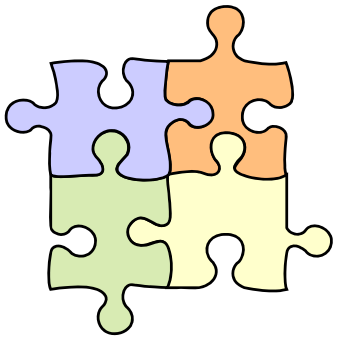
Luca Cabibbo



Architetture Software

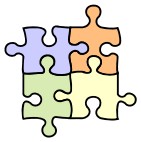
Introduzione ai connettori

Dispensa ASW 810

ottobre 2014

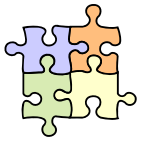
Le relazioni tra elementi
sono ciò che dà valore aggiunto
ai sistemi.

Eberhardt Rechtin



- Fonti

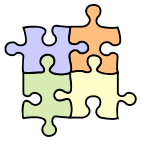
- [Shaw] Mary Shaw, Procedure Calls are the Assembly Language of Software Interconnections: Connectors Deserve First-Class Status, Technical report CMU/SEI-1994-TR-2, 1996
- [TMD] Taylor, Medvidovic, Dashofy, Software Architecture – Foundations, Theory, and Practice, 2010, Chapter 5, Connectors
- [Bernstein] Phil Bernstein, Middleware, Communications of the ACM, 1996



- Obiettivi e argomenti

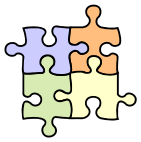
- Obiettivi
  - introdurre e motivare i connettori
  - fornire alcuni cenni sul middleware
- Argomenti
  - introduzione ai connettori
  - un esempio
  - verso il middleware
  - discussione

- Wordle



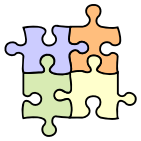
* Introduzione ai connettori

- 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
  - questa distinzione riflette la sostanziale indipendenza tra gli aspetti funzionali e quelli relativi alle interazioni



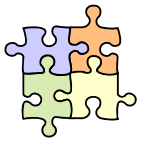
Connettori

- 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 effettivamente essere fatta separatamente da quella dei componenti
    - grazie alla sostanziale indipendenza tra gli interessi di computazione e quelli di interazione
- Inoltre, i connettori – diversamente dai componenti, che forniscono servizi specifici per un'applicazione – sono tipicamente indipendenti dalle applicazioni
  - questo ha portato allo sviluppo di numerosi strumenti di middleware – tecnologie software per l'implementazione di connettori, utili soprattutto nello sviluppo di sistemi distribuiti



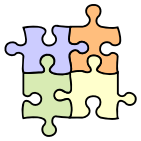
Componenti e connettori – esempi

- Alcune possibili tipologie 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 “dati”
- Alcune possibili tipologie 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 a una base di dati da parte di un processo



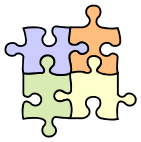
Componenti e connettori

- Secondo [Shaw]
  - 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 – ad es., l’essere client oppure server di un certo servizio



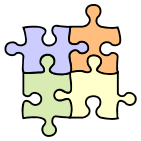
Componenti e connettori

- Inoltre [Shaw]
  - 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 temporale 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



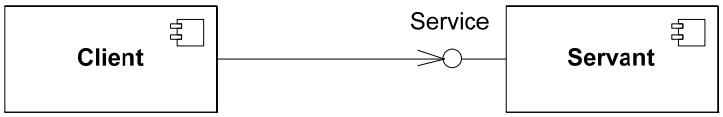
Componenti e connettori

- Alcuni motivi per cui è opportuno trattare i connettori separatamente dai componenti [Shaw]
  - 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



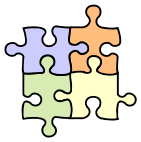
* Un esempio

- Si supponga di aver identificato, nell'ambito di una vista funzionale, un'interfaccia e due "componenti"
  - un'interfaccia che definisce un servizio ***Service***
  - un elemento ***Servant***, in grado di fornire il servizio ***Service***
  - un elemento ***Client***, che richiede l'erogazione del servizio ***Service*** (da parte del ***Servant*** o di chiunque sappia erogarlo)



  - 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?

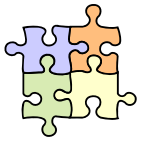


- Un esempio (prima versione)

- 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 Service. */
public interface Service {
    /* Fa qualcosa con arg. */
    public String alpha(String arg);
}
```

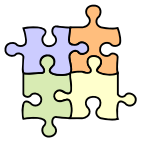
in **blu** indichiamo il codice relativo ad aspetti funzionali



Un esempio (prima versione)

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

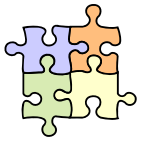
```
package asw.asw810.server;

import asw.asw810.service.Service;

/* Implementazione del servizio Service. */
public class Servant implements Service {

    public String alpha(String arg) { ... fa qualcosa con arg ... }

}
```

Un esempio (prima versione)

- Applicazione Main che crea e avvia il Client

```
package asw.asw810.client;

/* Applicazione client: crea e avvia il client. */
public class Main {

    /* Crea e avvia un oggetto Client. */
    public static void main(String[] args) {
        Client client = new Client();
        client.run();
    }

}
```

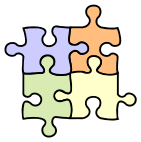


Un esempio (prima versione)

- 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 Service. */
public class Client {

    private Service service;

    public Client() {

    }

    public void run(...) {
        ... service.alpha(...) ...
    }

}
```

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

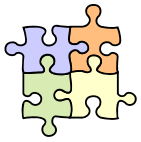


Un esempio (prima versione)

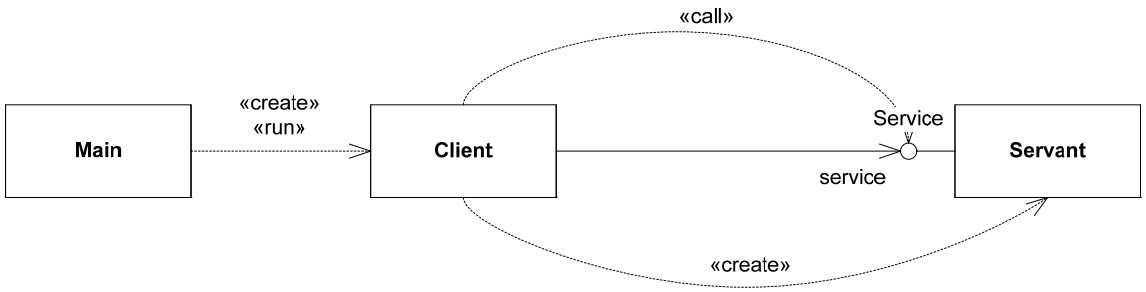
- Il connettore più semplice è la *chiamata di procedura* – o *invocazione di metodo* nei linguaggi OO

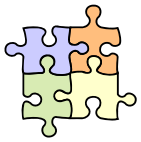
```
package asw.asw810.client;

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

/* Client del servizio Service. */
public class Client {

    private Service service;

    public Client() {
        this.service = new Servant();
    }

    public void run(...) {
        ... service.alpha(...) ...
    }

}
```

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



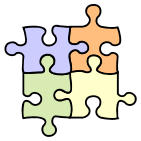
Un esempio (prima versione)





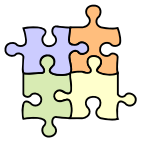
Un esempio (prima versione)

- 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 (seconda versione)

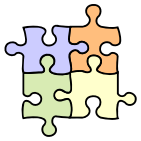
- Concentriamoci, per ora, sull'eliminazione della dipendenza tra il ***Client*** e la particolare implementazione ***Servant*** del ***Service***
  - si può rompere questa dipendenza utilizzando degli oggetti di supporto – in particolare, applicando degli opportuni design pattern
    - ad esempio, usando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



Un (piccolo) passo avanti (seconda versione)

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

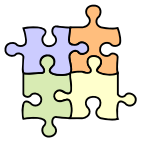
```
package asw.asw810.client.connector;

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

/* Factory per il servizio Service. */
public class ServiceFactory {
    private static ServiceFactory instance = null;    // singleton per la factory
    private Service service = null;    // singleton per il servizio
    private ServiceFactory() { }    // costruttore privato per singleton

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

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



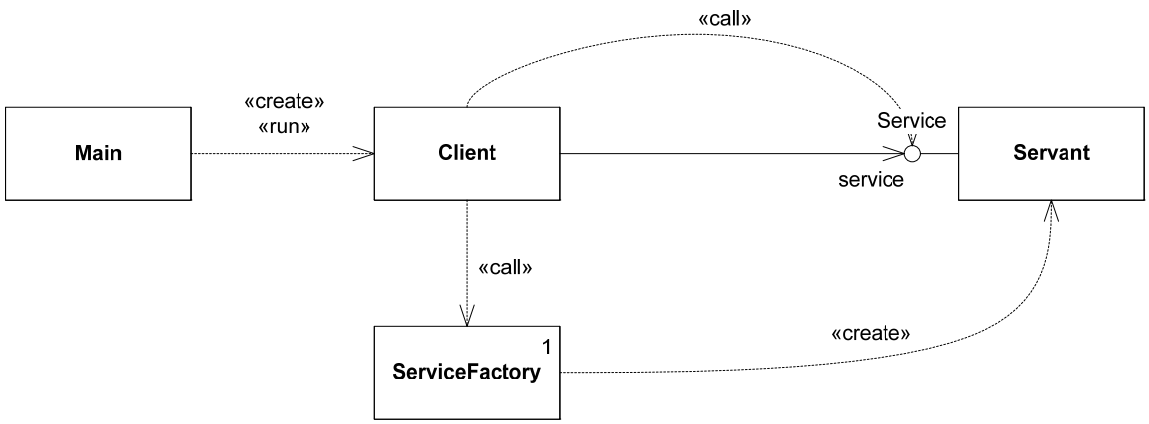
Un (piccolo) passo avanti (seconda versione)

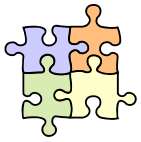
- Ecco la nuova versione per il client – il connettore sarà ancora una chiamata di procedura

```
package asw.asw810.client;

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

/* Client del servizio Service. */
public class Client {

    private Service service;

    public Client() {
        this.service = ServiceFactory.getInstance().getService();
    }

    public void run(...) {
        ... service.alpha(...) ...
    }

}
```

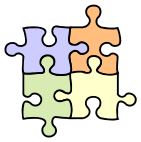
Un (piccolo) passo avanti (seconda versione)





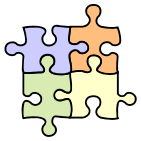
Un (piccolo) passo avanti (seconda versione)

- 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 possiamo considerare 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 servente né la factory
  - c'è ancora il vincolo che ***Client*** e ***Servant*** devono essere scritti nello stesso linguaggio di programmazione



- Un altro passo avanti (terza versione)

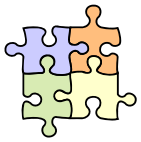
- Ci sono anche altri modi per eliminare la dipendenza tra il ***Client*** e la particolare implementazione ***Servant*** del ***Service***
  - in particolare, l'*iniezione delle dipendenze* è un design pattern che prevede quanto segue
    - ciascun oggetto definisce le sue dipendenze (altri oggetti da cui dipende)
    - le dipendenze di un oggetto non vengono risolte dall'oggetto stesso – ma piuttosto vengono risolte da altri oggetti di supporto, mediante un'iniezione delle dipendenze (di solito al momento della creazione di quest'oggetto)
    - l'iniezione delle dipendenze può essere scritta nel codice, oppure basata su file di configurazione o su annotazioni
  - chiamato anche *inversione del controllo* – poiché un oggetto non deve creare o cercare da solo gli oggetti da cui dipende
  - vediamo una possibilità



Un altro passo avanti (terza versione)

- Ecco la nuova versione per il client – che ora dipende solo dal servizio, e non più dal connettore

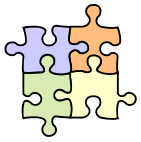
```
package asw.asw810.client;

import asw.asw810.service.Service;

/* Client del servizio Service. */
public class Client {

   private Service service;

   public Client() {
   }

   /* l'iniezione della dipendenza avverrà proprio tramite questo metodo setter */
   public void setService(Service service) {
      this.service = service;
   }

   public void run(...) {
      ... service.alpha(...) ...
   }

}
```

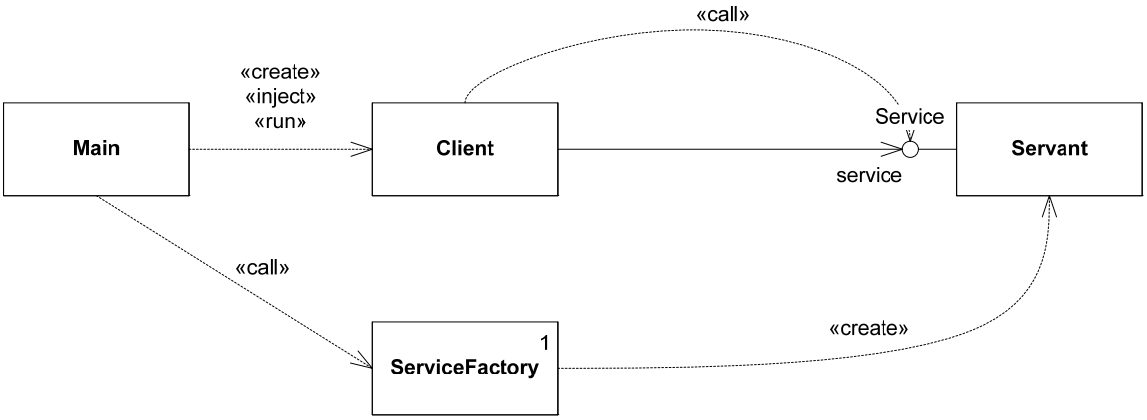
Un altro passo avanti (terza versione)

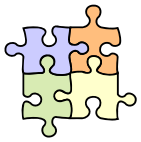
- L'iniezione della dipendenza viene effettuata, in questo caso, dall'applicazione Main

```
package asw.asw810.client;

import asw.asw810.service.Service;
import asw.asw810.client.connector.ServiceFactory;

/* Applicazione client: crea e avvia il client. */
public class Main {

    /* Crea e avvia un oggetto Client. */
    public static void main(String[] args) {
        Service service = ServiceFactory.getInstance().getService();
        Client client = new Client();
        /* iniezione della dipendenza service */
        client.setService(service);
        client.run();
    }

}
```



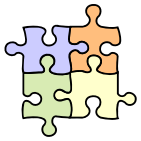
Un altro passo avanti (terza versione)





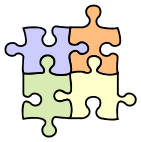
Un altro passo avanti (terza versione)

- Alcune caratteristiche della nuova soluzione
  - rispetto a quanto fatto in precedenza, ora il Client non dipende più dalla factory
  - in pratica, nel codice del “componente” Client non c’è più nessuna traccia di codice “connettore”
  - ciò che è “connettore” è realizzato separatamente da ciò che è “componente”!

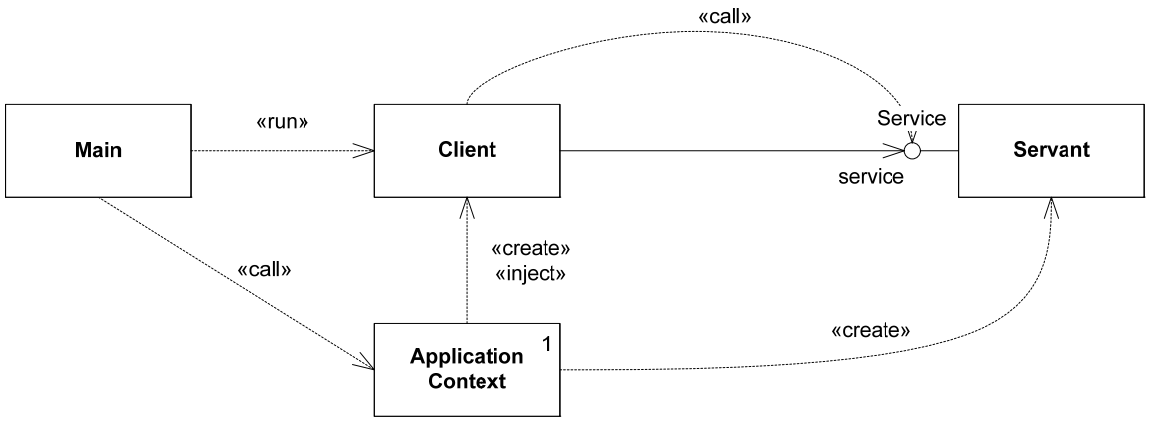


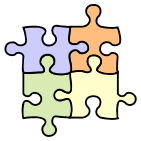
- Un altro passo avanti (quarta versione)

- Può essere utile avere un oggetto che si occupa della creazione di tutti gli oggetti – nel nostro esempio, non solo del servente, ma anche del client – nonché delle iniezioni delle dipendenze necessarie
  - un tale oggetto è chiamato talvolta un *application context*
  - un application context ha le seguenti responsabilità
    - fornire l'accesso a un certo insieme di oggetti
    - creare questi oggetti – se e quando è necessario
    - risolvere le dipendenze tra di essi – mediante iniezione delle dipendenze



Un altro passo avanti (quarta versione)

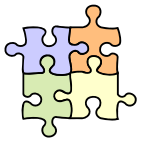




Un altro passo avanti (quarta versione)

- L'application context è simile alla service factory
  - ma ora ha anche un metodo per creare il client

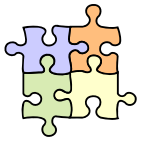
```
package asw.asw810.context;

import asw.asw810. ...

/* Application context. */
public class ApplicationContext {

    ... variabile, costruttore e metodo per singleton, come prima ...

    ... variabile per service e metodo getService, come prima ...

    /* Factory method per il client Client.
     * Ogni volta viene restituito un nuovo client. */
    public Client getClient() {
        Client client = new Client();
        client.setService( this.getService() );
        return client;
    }

}
```

Un altro passo avanti (quarta versione)

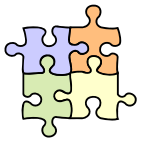
- La classe Main è semplificata

```
package asw.asw810.client;

import asw.asw810.context.ApplicationContext;

/* Applicazione client: ottiene e avvia il client. */
public class Main {

    /* Crea e avvia un oggetto Client. */
    public static void main(String[] args) {
        Client client = ApplicationContext.getInstance().getClient();
        client.run();
    }

}
```

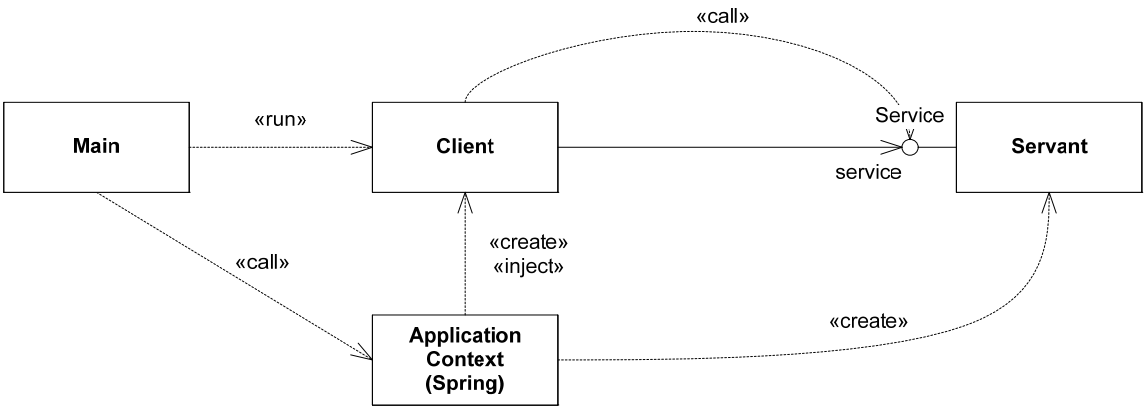


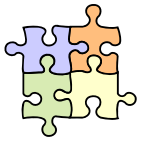
- Una deviazione (quinta versione)

- La creazione di oggetti e l'iniezione delle dipendenze sono compiti comuni nelle applicazioni complesse
  - questo compito può essere svolto da un opportuno framework – anziché scrivere ogni volta del codice apposito
  - ad esempio, è possibile usare Spring Framework – http://spring.io/
  - Spring Framework è una soluzione leggera (lightweight) e modulare per la realizzazione di applicazioni complesse
  - ora viene mostrato l'uso del contenitore IoC (Inversion of Control) di Spring Framework per realizzare l'iniezione delle dipendenze
  - questo avviene mediante l'uso di un "application context" predefinito – insieme ad un apposito file di configurazione



Una deviazione (quinta versione)

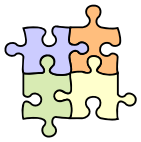




Una deviazione (quinta versione)

- L'application context di Spring è basato su un file di configurazione che descrive gli oggetti (bean) e le dipendenze tra di essi

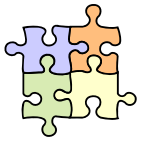
```
<?xml version="1.0" encoding="UTF-8"?>
<beans …>

    <bean id="service" class="asw.asw810.server.Servant"/>

    <bean id="client" class="asw.asw810.client.Client">
        <property name="service" ref="service" />
    </bean>

</beans>
```

Una deviazione (quinta versione)

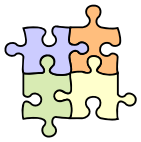
- Cambia solo la classe Main

```
package asw.asw810.client;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/* Applicazione client: ottiene e avvia il client. */
public class Main {

    /* Crea e avvia un oggetto Client. */
    public static void main(String[] args) {
        ApplicationContext context =
                new ClassPathXmlApplicationContext("spring-beans.xml");
        Client client = (Client) context.getBean("client");
        client.run();
    }

}
```



- Un altro passo avanti (sesta versione)

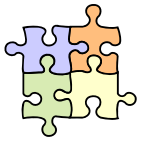
- Mettiamo da parte l'uso di un contenitore IoC come Spring Framework per discutere di un altro aspetto importante
  - nelle soluzioni precedenti, la factory (o l'application context) crea un'istanza del servente e ne restituisce un riferimento che viene memorizzato direttamente dal client
  - è invece spesso comune (e utile) che venga usato un ulteriore intermediario, un *proxy* – un rappresentante del server presso il client
    - il proxy, per ora, gestisce solo un riferimento al vero servente
    - la factory (o l'application context) restituisce (o inietta) non il vero servant, ma piuttosto un suo proxy
    - il client memorizza un riferimento al proxy – pensando che sia un riferimento al servente



Un altro passo avanti (sesta versione)

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

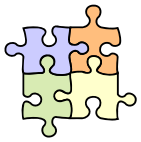
```
package asw.asw810.client.connector;

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 che verrà invocato dal client
     * (anche se il client penserà di parlare direttamente con il servant) */
    public String alpha(String arg) {
        /* chiama il vero servizio */
        return service.alpha(arg);
    }
}
```



Un altro passo avanti (sesta versione)

- L'application context (o la service factory) crea il Servant (se necessario) – ma restituisce un ***ServiceProxy***

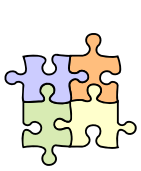
```
package asw.asw810.context;

import asw.asw810. ...

/* Application context. */
public class ApplicationContext {

    ... variabile, costruttore e metodo per singleton, come prima ...

    ... metodo getClient, come prima ...

    private Service service = null;

    /* Factory method per il Servizio.
     * Restituisce un proxy al servizio. */
    public synchronized Service getService() {
        if (service==null) { service = new Servant(); }
        return new ServiceProxy( service );
    }

}
```

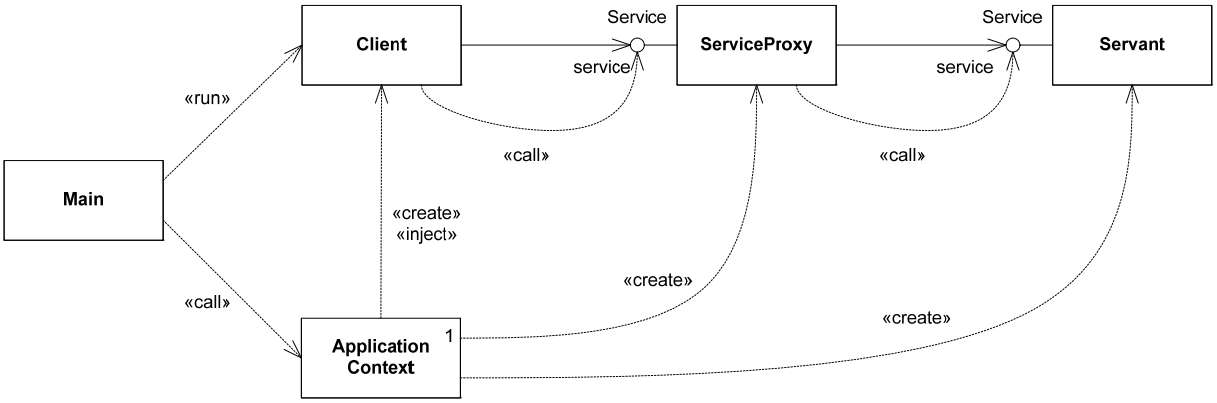
Un altro passo avanti (sesta versione)

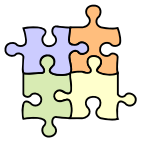
- Non cambia né il client né l'applicazione main
  - il connettore è ancora, a tutti gli effetti, una chiamata di procedura locale

```
package asw.asw810.client;

import asw.asw810.service.Service;

/* Client del servizio Service. */
public class Client {

    private Service service;

    public Client() { }

    public void setService(Service service) {    this.service = service;    }

    public void run(...) {
        ... service.alpha(...) ...
    }

}
```



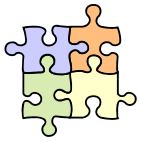
Un altro passo avanti (sesta versione)





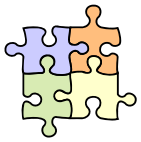
Un altro passo avanti (sesta versione)

- Che cosa è cambiato?
  - in pratica, sembra tutto come prima – ovvero, sembra che abbiamo “cambiato tutto, per non cambiare nulla”
- Ora però abbiamo introdotto 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 proprio nel proxy
    - senza cambiare né client né servente – ovvero, i componenti che si occupano degli 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

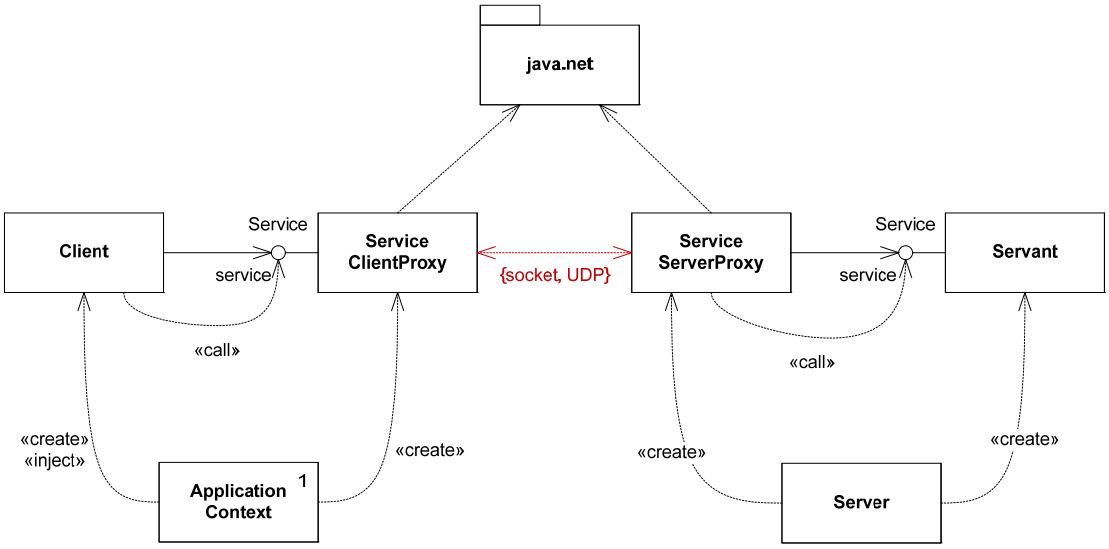


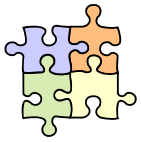
- Ancora un passo avanti (un po' più grande)

- Concentriamoci ora sull'eliminazione della co-localizzazione tra ***Client*** e ***Servant***
  - è possibile gestire una comunicazione remota tra client e server usando un meccanismo di comunicazione interprocesso
    - ad es., le socket – un meccanismo di IPC, 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, che vive nello stesso processo del ***Servant***



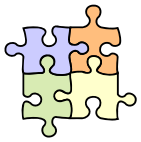
Ancora un passo avanti (un po' più grande)





Ancora un passo avanti (un po' più grande)

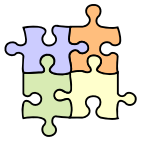
- Utilizzando le socket UDP, client e server possono comunicare scambiandosi messaggi – chiamati datagrammi – come segue
  - il client (qui inteso come processo) effettua una richiesta
  - il client forma un messaggio che codifica la sua richiesta (marshalling della richiesta) – il nome dell'operazione 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 così ricevuto la risposta alla sua richiesta



Ancora un passo avanti (un po' più grande)

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

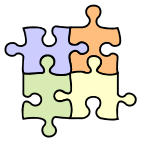
```
package asw.asw810.client.connector;

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 ...
    }
}
```



Ancora un passo avanti (un po' più grande)

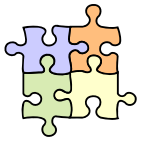
- Il metodo ***alpha*** del "remote proxy" lato client

```
/* 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
        e i relativi parametri ...
    ... invia il datagramma di richiesta ...
    ... ricevi il datagramma di risposta ...
    ... estrai la risposta dal datagramma di risposta ...
    return reply;

}
```

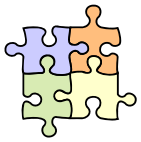
Ancora un passo avanti (un po' più grande)

- L'application context (o la service factory) può essere usato 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
  - tuttavia, l'application context (o la factory) non si può più occupare della creazione del vero ***Servant*** – che in generale vive in un altro processo
- Il ***Client*** e il ***Servant*** (i "componenti") possono rimanere ancora immutati rispetto alla versione precedente



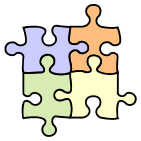
Ancora un passo avanti (un po' più grande)

```
package asw.asw810.context;

import asw.asw810. ...

/* Application context. */
public class ApplicationContext {

    ... variabile, costruttore e metodo per singleton, come prima ...

    ... metodo getClient, come prima ...

    private Service service = null;

    /* Factory method per il Servizio.
     * Restituisce un remote proxy al servizio. */
    public Service getService() {
        InetAddress address = InetAddress.getByName("localhost");
        int port = 6789;
        Service proxy = new ServiceClientProxy(address, port);
        return proxy;
    }

}
```

Ancora un passo avanti (un po' più grande)

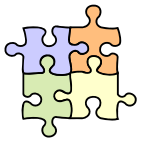
- Il "remote proxy" lato client non comunica direttamente con il ***Servant***
  - piuttosto, è necessario un ulteriore intermediario – un "remote proxy" lato server
  - il "remote proxy" lato server
    - riceve richieste – tramite socket – dal proxy lato client, e le gira al ***Servant***
    - riceve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



Ancora un passo avanti (un po' più grande)

- Struttura del "remote proxy" lato server

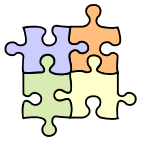
```
package asw.asw810.server.connector;

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 ...
    }
}
```

Ancora un passo avanti (un po' più grande)

- Il metodo ***run*** del "remote proxy" lato server
  - per semplicità, un server "sequenziale" – anche in questo caso si potrebbe fare di meglio

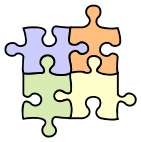
```
    public void run() {

        ... 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 al servente e ottieni la risposta ...
            ... crea il datagramma di risposta ...
            ... invia il datagramma di risposta ...
        }

}
```



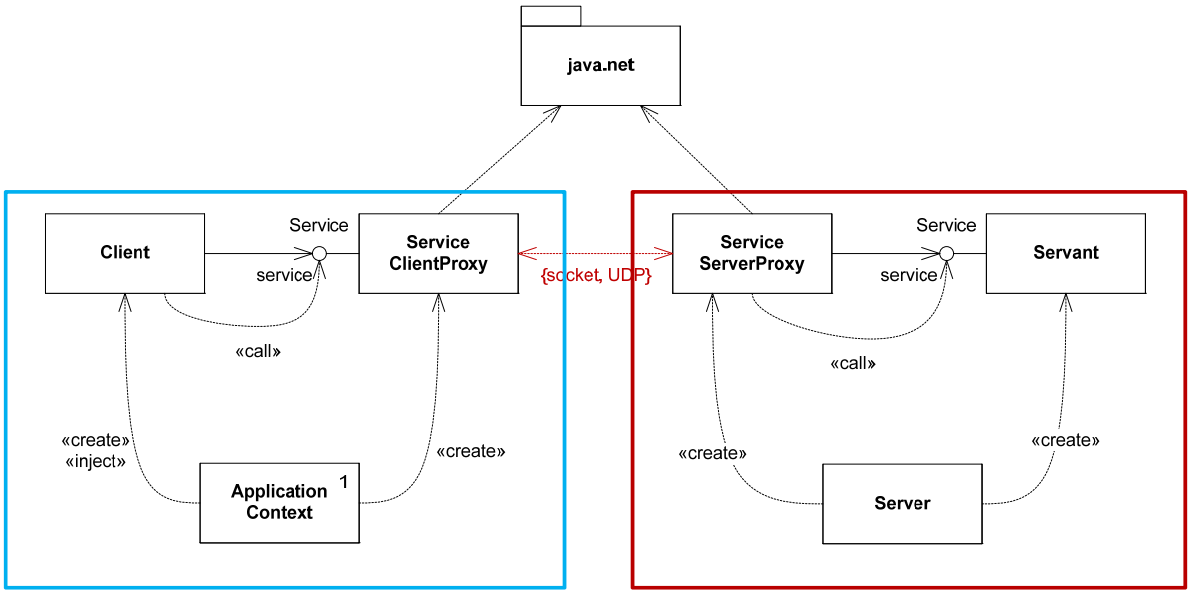
Ancora un passo avanti (un po' più grande)

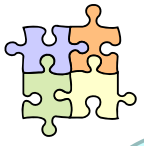
- Lato server, manca ancora un oggetto ***Server*** responsabile
  - della creazione del ***Servant***
  - della creazione e dell'avvio del proxy lato server

```
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();
    }
}
```

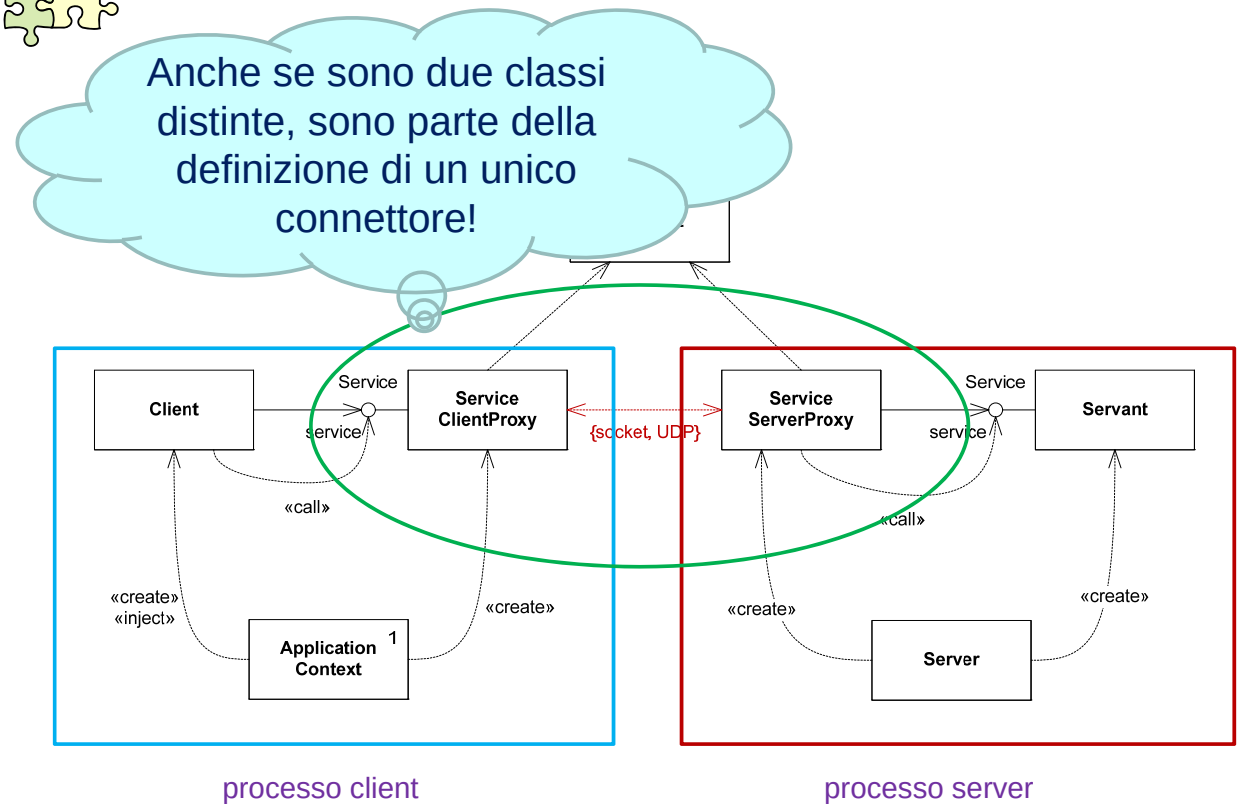



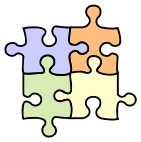
Ancora un passo avanti (un po' più grande)





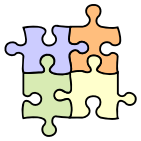
Ancora un passo avanti (un po' più grande)





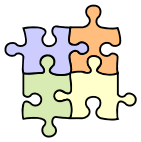
Ancora un passo avanti (un po' più grande)

- 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*** e il ***Servant*** potrebbero essere scritti con linguaggi di programmazione diversi, in esecuzione in ambienti hw/sw differenti
    - grazie al supporto "universale" delle socket



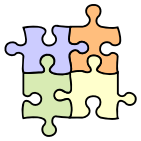
Discussione

- 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 – e 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
  - inoltre, l'esempio ha mostrato l'uso di factory, application context, proxy e dell'iniezione delle dipendenze – che è comune incontrare nell'uso degli strumenti di middleware



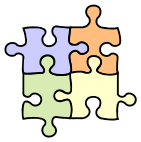
Discussione

- 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 e 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



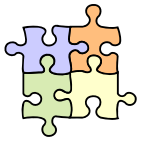
* Verso il middleware

- Meccanismi di base come le socket possono essere considerate l'"assembler" per lo sviluppo dei connettori
  - usando questo "assembler" e 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



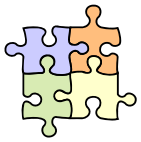
Verso il middleware

- 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 middleware consente di generare automaticamente tutto (o quasi) il codice per i connettori
    - il programmatore "lato server" scrive l'interfaccia del servizio e ne implementa il servente
    - il programmatore "lato client" utilizza il servizio tramite un proxy – creato di solito da una factory o un application context
    - tutto il codice del connettore (compresi proxy e factory) viene generato automaticamente



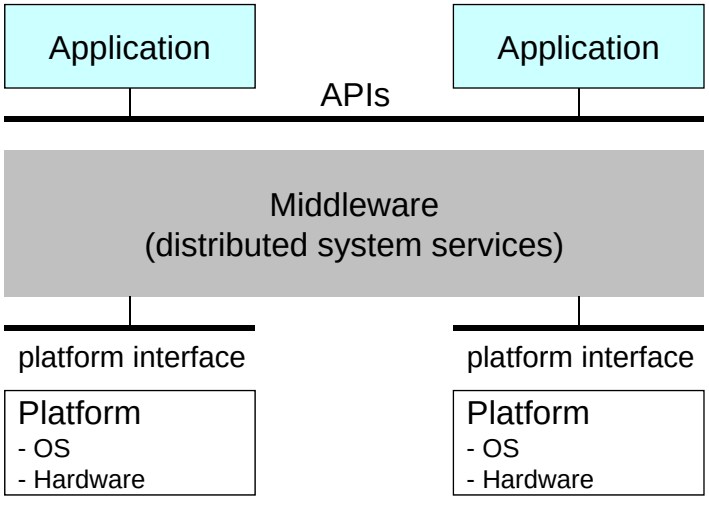
Middleware

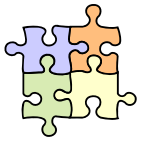
- Il **middleware** è [D.E. Bakken, Middleware, 2003]
  - una classe di tecnologie software sviluppate per aiutare gli sviluppatori nella gestione della complessità e della eterogeneità presenti nei sistemi distribuiti
  - uno strato software “in mezzo”
    - sopra al sistema operativo, ma sotto i programmi 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

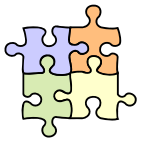
- Un **servizio di middleware** è un servizio general-purpose che si colloca tra piattaforme e applicazioni [Bernstein]





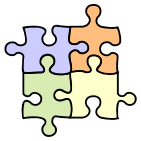
Middleware – esempi

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



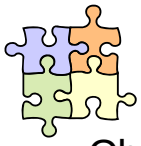
Middleware

- Se utilizzato in modo opportuno, il middleware affronta diverse problematiche significative per i sistemi distribuiti
  - consente di concentrarsi sullo sviluppo della logica applicativa – anziché sui dettagli della comunicazione tra componenti
- Malgrado i molti benefici offerti dal middleware, il middleware non va però considerato una panacea per i sistemi distribuiti
  - il middleware non può risolvere “magicamente” i problemi derivanti da decisioni di progetto povere
  - la decisione del middleware da utilizzare richiede delle considerazioni e delle decisioni esplicite
    - è necessaria una buona comprensione del paradigma di comunicazione implementato da ciascun middleware, della sua struttura e dei suoi principi di funzionamento
    - è inoltre necessario comprendere le relazioni tra middleware e stili architetturali



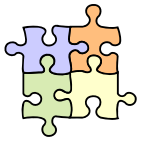
* Discussione

- 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 del 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 (RMI)
    - versione a oggetti della chiamata di procedura remota (RPC)
  - messaging
    - comunicazione asincrona basata sullo scambio di messaggi
  - application server
    - componenti e contenitori
  - web services
    - servizi interoperabili, sincroni e asincroni
  - datastore NoSQL
    - gestione di dati distribuiti sul cloud
  - 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