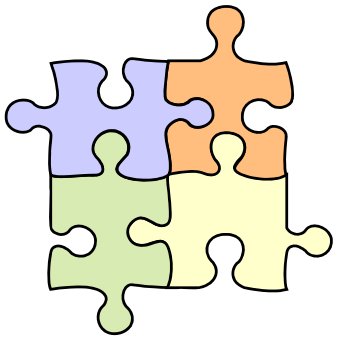
Luca Cabibbo



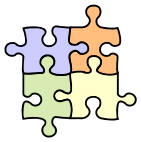
Architetture Software

Pattern software

Dispensa ASW 350
ottobre 2014

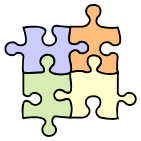
Prima che sia stato provato, è un'opinione.
Dopo che è stato provato, è ovvio.

William C. Burkett



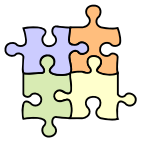
- Fonti

- [SSA] Chapter 11, Using Styles and Patterns
- [SAP] Chapter 13, Architectural Tactics and Patterns
- [POSA1] Pattern-Oriented Software Architecture (vol. 1) – A System of Patterns, 1996
- [POSA4] Pattern-Oriented Software Architecture (vol. 4) – A Pattern Language for Distributed Computing, 2007
  - nota: [POSA] indica genericamente [POSA1] oppure [POSA4] – che peraltro sono parzialmente sovrapposti
- [GoF] Gamma, Helm, Johnson, Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software, 1995
  - oppure: Design Patterns – Elementi per il Riuso di Software a Oggetti, 2002

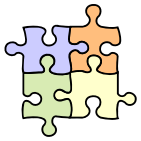


- Obiettivi e argomenti

- Obiettivi
  - comprendere il ruolo dei pattern software – in particolare, nel contesto della definizione dell'architettura
  - esemplificare, in modo sintetico, alcuni pattern
- Argomenti
  - pattern software
  - adapter [GoF]
  - proxy [GoF]
  - facade [GoF]
  - observer [GoF]
  - alcuni stili architetturali comuni
  - discussione

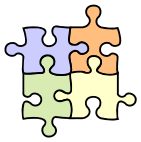


- Wordle



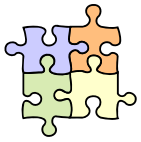
* Pattern software

- Un *pattern software* è
  - una soluzione provata e ampiamente applicabile a un particolare problema di progettazione
  - che è descritta in una forma standard, in modo che possa essere facilmente condivisa e riusata
- Una possibile descrizione strutturata
  - nome
  - contesto – la situazione in cui il pattern può essere applicato
  - problema – e forze in gioco
  - soluzione
  - conseguenze – risultati (positivi e negativi) e compromessi



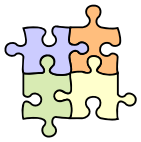
Stili, pattern e idiomi

- Tre principali categorie di pattern software
  - stili architetturali
  - design pattern
  - idiomi
  - granularità diversa – sia del problema che degli elementi nella soluzione
  - non completamente indipendenti tra di loro



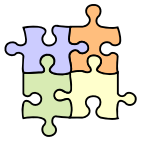
Stili architetturali

- Uno **stile** (o **pattern**) **architetturale** [POSA]
  - descrive un particolare problema di progettazione ricorrente che si manifesta in uno specifico contesto di progettazione
  - e presenta un ben provato schema generico per la sua soluzione
  - lo schema di soluzione è specificato dalla descrizione dei componenti che lo costituiscono, le loro responsabilità e relazioni, e il modo in cui essi collaborano



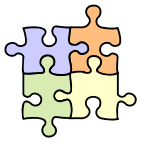
Stili architetturali

- Detto in altro modo
  - uno **stile** (o **pattern**) **architetturale** esprime una schema per l'*organizzazione strutturale fondamentale di sistemi software* [SSA]
  - uno stile architetturale fornisce un insieme di tipi di elementi predefiniti, specifica le loro responsabilità, e comprende regole e linee guida per organizzare le relazioni tra di essi
    - in questo modo, uno stile guida la decomposizione del sistema in elementi di certi tipi e sulla base di certi tipi di relazioni tra questi elementi
  - inoltre, uno stile discute le possibilità di raggiungere (o meno) certi obiettivi di qualità
- Esempi di stili architetturali
  - Layers, Client/Server, ... – ma anche Broker, Container, ...



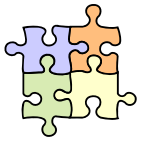
Stili architetturali

- Detto ancora in un altro modo
  - uno **stile** (o **pattern**) **architetturale** [SAP]
    - è un pacchetto (ovvero, una combinazione) di decisioni di progetto – che è stato applicato ripetutamente in pratica
    - che ha delle proprietà note, che ne consente il riuso
    - e descrive una *classe* di architetture
  - per questo, nello studio di un pattern architetturale
    - è necessario comprendere le proprietà del pattern
    - può essere utile comprendere anche le singole decisioni di progetto da cui ha origine



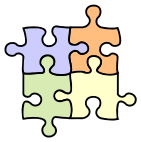
Design pattern

- Un **design pattern** [GoF]
  - è la descrizione di oggetti e classi che comunicano, personalizzati per risolvere un problema generale di progettazione in un contesto particolare



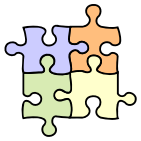
Design pattern

- Detto in altro modo
  - un **design pattern** fornisce uno schema per *raffinare gli elementi di un sistema software o le relazioni tra di essi* [SSA]
  - descrive una struttura ricorrente di elementi di progetto interconnessi, che risolvono un problema di progettazione generale in un contesto particolare
  - in particolare, i design pattern possono essere usati nell'ambito della progettazione di dettaglio di singoli elementi architetturali – nonché delle interazioni tra di essi
- Esempi di design pattern
  - Singleton, Adapter, Proxy, Observer, ...



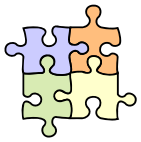
Idiomi

- Un **idioma** è un pattern di basso livello, specifico di un linguaggio di programmazione
  - descrive come implementare *aspetti particolari di elementi o delle relazioni tra essi*, usando una caratteristica di *un certo linguaggio* [SSA]
- Esempi di idiomi
  - come implementare Singleton in Java
    - variabile di classe *instance*, metodo di classe *getInstance*, costruttore privato
    - come si fa in C++?
  - counted pointer
    - per gestire oggetti condivisi in C++ – ogni oggetto sa quanti oggetti lo referenziano – in questo modo è possibile capire quando può essere deallocato
    - utile in Java?

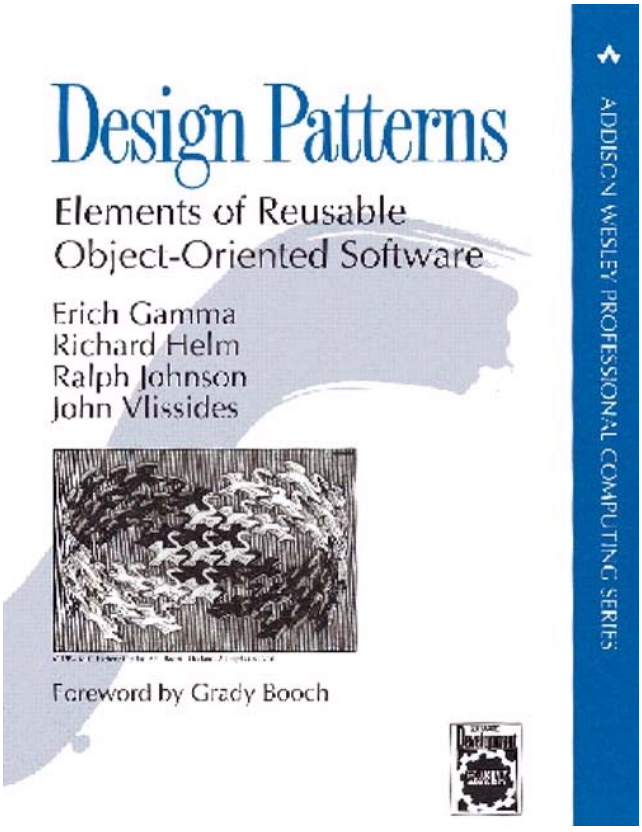


Pattern e linguaggi di pattern

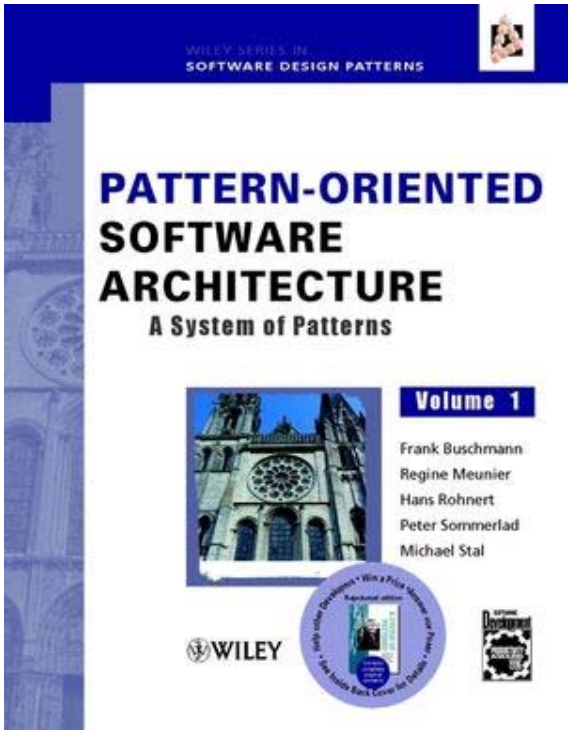
- I pattern non sono tra loro indipendenti
  - alcuni pattern sono alternativi – “ne uso uno oppure l’altro”
  - altri pattern sono sinergici – “se ne uso uno è utile usare anche l’altro”
  - altri possono essere usati in gruppi più complessi
  - dunque è utile ragionare anche sulle relazioni tra pattern
- Un *pattern language* – *linguaggio di pattern*
  - una famiglia di pattern correlati
    - anche con una discussione relativa alle loro correlazioni
  - ne esistono diversi – specifici per la progettazione di certi tipi di sistemi o per certi tipi di requisiti
    - ad es., linguaggio di pattern per la sicurezza
    - ad es., linguaggio di pattern per sistemi distribuiti [POSA4]



Linguaggi di pattern – [GoF]

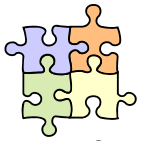


Linguaggi di pattern – [POSA]



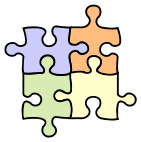
Ruolo dei pattern software

- Alcuni dei ruoli svolti da pattern (e linguaggi di pattern)
  - deposito di conoscenza
  - esempi di buone pratiche
  - un linguaggio per discutere problemi di progettazione
  - un aiuto alla standardizzazione
  - una sorgente di miglioramento continuo
  - incoraggiamento alla generalità
- Il ruolo principale dal punto di vista delle architetture software
  - riduzione del rischio
  - incremento della produttività, della standardizzazione e della qualità



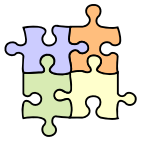
Design pattern e idiomi nell'architettura

- È chiaro il ruolo degli stili architetturali nella definizione di un'architettura – qual è invece il ruolo dei design pattern e degli idiomi nella definizione di un'architettura?
  - i design pattern sono spesso utili nel descrivere le connessioni tra elementi architetturali
    - l'architettura deve comprendere una descrizione delle relazioni tra elementi ed indicare un approccio uniforme nella loro realizzazione
    - ad es., comunicazione tra strati basata su Facade, accesso ai dati persistenti mediante DAO, trasferimento di dati tra livelli mediante DTO, ...
  - in generale, strumento di comunicazione tra architetto e progettista
- Seguono alcuni esempi di design pattern



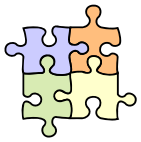
* Adapter [GoF]

- **Adapter** [GoF]
  - adatta l'interfaccia di un elemento di un sistema a una forma richiesta da uno dei suoi client
  - scopo – convertire l'interfaccia di una classe in un'altra interfaccia richiesta dal client – ***Adapter*** consente a classi diverse di operare insieme quando ciò non sarebbe altrimenti possibile a causa di interfacce incompatibili [GoF]



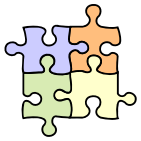
Adapter

- Alcuni esempi di adattatori (non software)



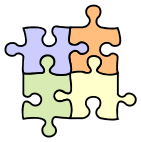
Adapter

- Contesto
  - bisogna connettere diversi elementi eterogenei



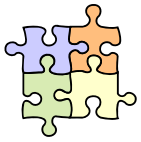
Adapter

- Problema (e forze)
  - un elemento (*client*) potrebbe usare i servizi offerti da un altro elemento (*adaptee*) – ma l'interfaccia dell'adaptee non è adatta al client
    - ad es., il client è .NET mentre l'adaptee è Java
  - il client potrebbe usare direttamente l'adaptee – ma il conseguente accoppiamento stretto è indesiderato
    - ad es., cambiamenti nell'adaptee richiedono cambiamenti nel client – client diversi devono cambiare in modo diverso
  - preferibile l'uso di un intermediario (adattatore)
    - l'adattatore dovrebbe solo fornire un servizio di adattamento e "traduzione" – e non fornire direttamente delle funzionalità
    - l'uso dell'adattatore non dovrebbe avere effetti negativi sulle qualità del servizio sottostante – ad es., sicurezza, prestazioni, affidabilità, ...



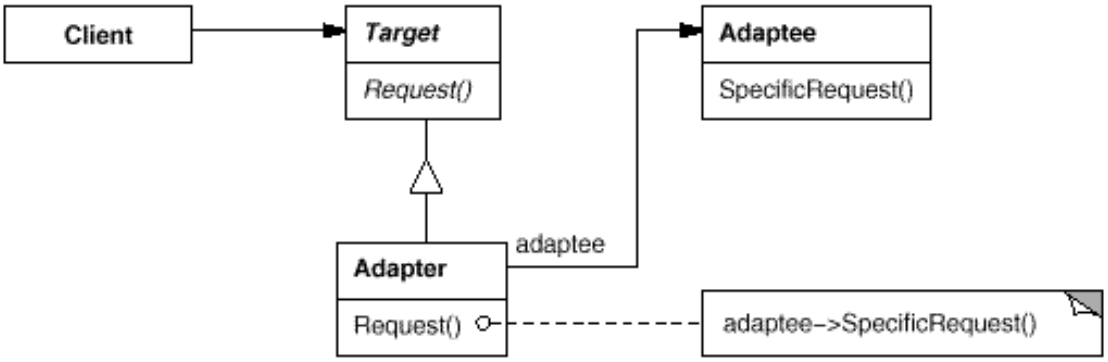
Adapter

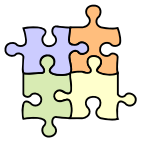
- Soluzione
  - introdurre un terzo elemento separato tra il client e l'adaptee – un elemento adattatore (*adapter*)
    - quando il client vuole comunicare con l'adaptee – il client comunica con l'adattatore e l'adattatore comunica con l'adaptee
    - il ruolo dell'adattatore è semplicemente di interpretare richieste del client, trasformarle in richieste all'adaptee, ottenere risposte dall'adaptee, trasformarle in risposte al client
    - l'adattatore ha in generale un'interfaccia (*target*) diversa da quella dell'adaptee – l'interfaccia dell'adattatore viene scelta in modo tale che sia "gradita" al client



Adapter

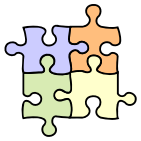
- Struttura della soluzione





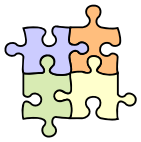
Adapter

- Conseguenze
  - ☺ disaccoppiamento delle implementazioni del client e dell'adaptee – l'implementazione di ciascun elemento può variare senza che questo si ripercuota sull'altro elemento
  - ☺ l'adaptee può essere usato da diversi tipi di client, ciascuno col suo adattatore
  - ☹ l'indirezione addizionale potrebbe ridurre l'efficienza
  - ☹ ci potrebbe essere un aumento nell'overhead per la manutenzione nel caso in cui cambiasse il servizio offerto dall'adaptee – e quindi oltre all'adaptee (ed eventualmente al client) dovrebbe cambiare anche l'adattatore



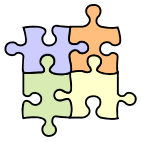
* Proxy [GoF]

- Il design pattern **Proxy** fa comunicare i client di un componente che offre servizi con un rappresentante di quel componente – e non direttamente con il componente stesso
  - l'introduzione di un tale "segnaposto" può sostenere diversi scopi – ad es., aumentare l'efficienza, semplificare l'accesso, consentire la protezione da accessi non autorizzati
  - descritto in [GoF] – ma anche tra i design pattern di [POSA]
- Proxy [GoF]
  - fornisce un surrogato o un segnaposto per un altro oggetto – per controllarne l'accesso



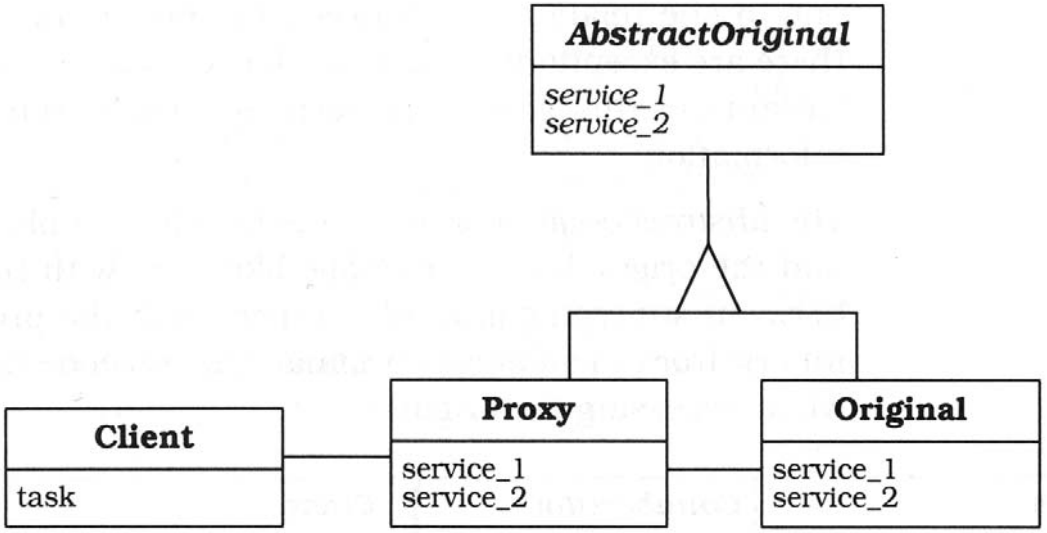
Proxy

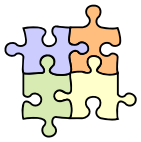
- Contesto
  - l'accesso diretto tra un client e un componente è tecnicamente possibile – ma non è l'approccio migliore
- Problema
  - l'accesso diretto a un componente è spesso inappropriato – per motivi di accoppiamento, efficienza, sicurezza, ...
- Soluzione
  - il client viene fatto comunicare con un rappresentante del componente – il *proxy*
  - il proxy non fa solo da intermediario – ma esegue delle ulteriori pre- e post-elaborazioni
    - ad es., effettua un controllo dell'accesso



Proxy

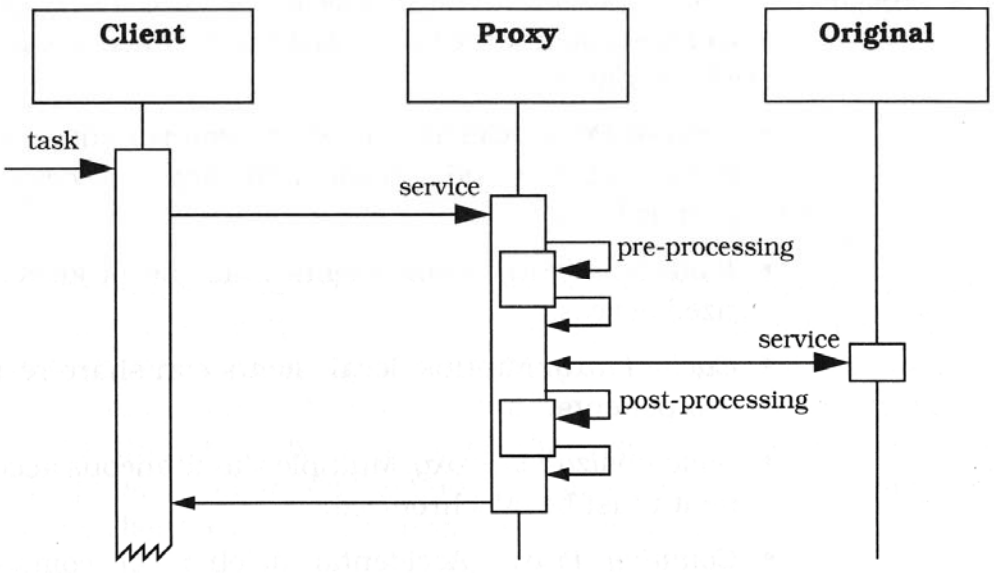
- Struttura della soluzione

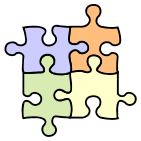




Proxy

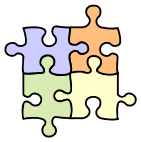
- Il Proxy è un intermediario tra il client e il fornitore del servizio – con la stessa interfaccia
  - ma esegue delle ulteriori pre- e post-elaborazioni





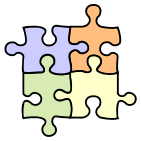
Possibili applicazioni

- Alcuni casi specifici comuni di Proxy
  - *Remote Proxy* – intermediario per l'accesso a un componente remoto – usato, ad esempio, nel pattern architetturale Broker
  - *Protection Proxy* – gestisce il controllo degli accessi a un componente
  - *Cache Proxy* – diversi client locali possono condividere i risultati da componenti remoti
  - *Synchronization Proxy* – sincronizza gli accessi concorrenti a un componente
  - *Virtual Proxy* – sostiene l'accesso pigro a informazioni il cui accesso è costoso
  - *Firewall Proxy* – protegge i client locali nell'accesso a servizi di rete
  - ...



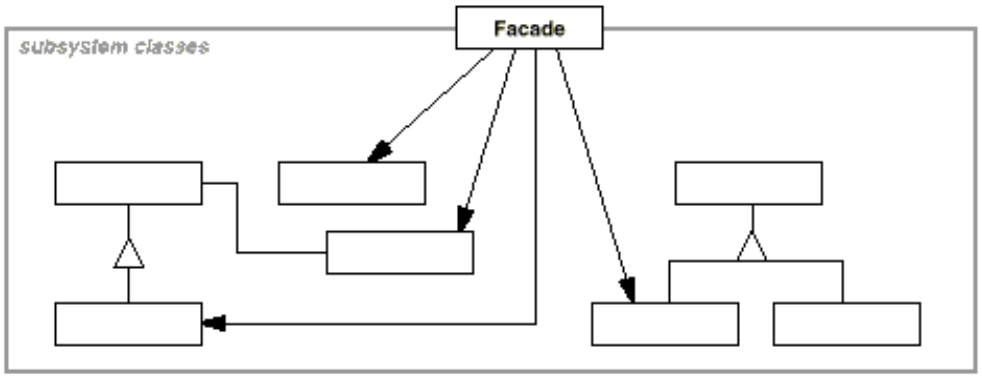
Proxy

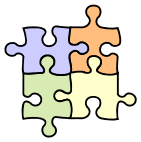
- Conseguenze – con l'osservazione che l'indirezione realizzata da un proxy ha finalità diverse, secondo il tipo di proxy
  - ☺ un proxy remoto può nascondere al client il fatto che il servizio sia remoto
  - ☺ un proxy virtuale può svolgere ottimizzazioni come la creazione di oggetti su richiesta
  - ☺ un proxy di protezione può svolgere operazione aggiuntive di gestione del servizio
  - ☺ …
  - ☹ l'indirezione addizionale potrebbe ridurre l'efficienza



* Facade [GoF]

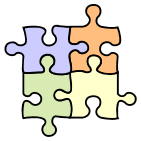
- Il design pattern **Facade** [GoF]
  - fornisce un'interfaccia unificata per un insieme di interfacce presenti in un sottosistema
  - Facade definisce un'interfaccia di livello più alto, che rende il sottosistema più semplice da utilizzare





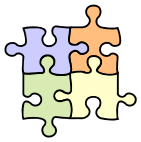
Facade

- Nelle architetture software, Facade può essere applicato per definire un punto d'accesso a un elemento architetturale
- Per capire l'impatto che possono avere le facade in un progetto, è interessante discutere questo pattern in termini delle tattiche architetturali per la modificabilità che consente di applicare
  - una facade sostiene l'applicazione di *Encapsulate*
    - la facade può definire l'interfaccia di un sottosistema
  - una facade può essere un'applicazione di *Use an intermediary*
  - una facade può sostenere l'applicazione di *Abstract common services*
    - la facade può nascondere (ai client di un'astrazione) un'implementazione (complessa) dell'astrazione
  - una facade può essere un'applicazione di *Restrict dependencies*



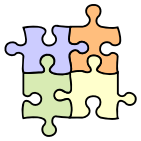
* Observer [GoF]

- **Observer** [GoF] – **Publisher/Subscriber** [POSA]
  - definisce una dipendenza uno a molti fra oggetti, in modo tale che se un oggetto cambia il suo stato, tutti gli oggetti dipendenti da questo siano notificati e aggiornati automaticamente



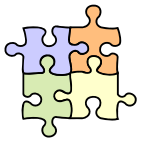
Observer

- Contesto
  - un elemento (*publisher*) crea informazioni che sono di interesse per altri elementi (*subscriber*)
- Problema
  - diversi tipi di oggetti *subscriber* (abbonato) sono interessati ai cambiamenti di stato o agli eventi di un oggetto *publisher* (editore)
  - ciascun subscriber vuole reagire in un modo proprio quando un publisher genera un evento
  - il publisher vuole mantenere un accoppiamento basso verso i suoi subscriber



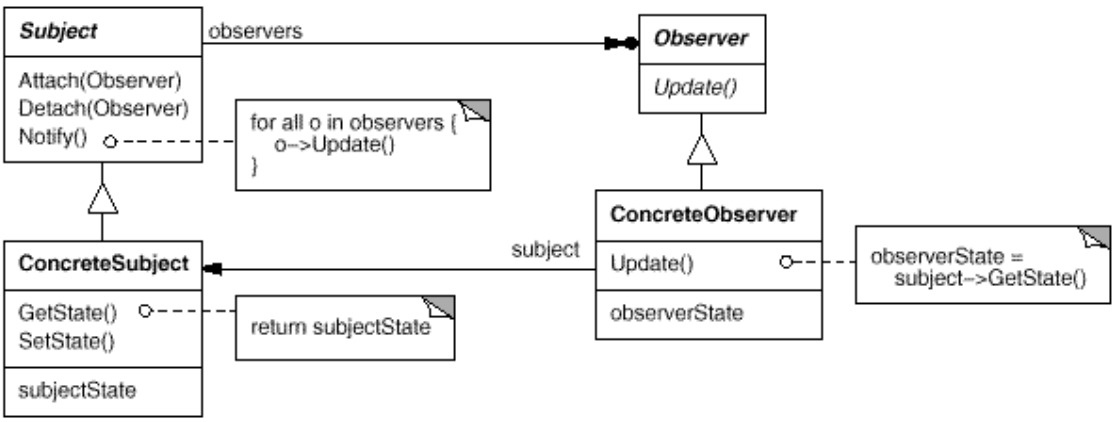
Observer

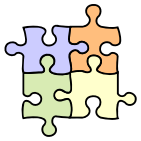
- Soluzione
  - definisci un'interfaccia "subscriber" o "listener" (ascoltatore)
  - i subscriber implementano questa interfaccia
  - il publisher registra dinamicamente i subscriber che sono interessati ai suoi eventi – li avvisa quando si verifica un evento
    - la notifica può contenere tutti i dettagli dell'evento
      - ad es., dicendo "X è cambiato e ora vale 42"
    - oppure, può contenere solo alcuni dettagli dell'evento
      - ad es., dicendo "X è cambiato" – poi il subscriber, se è veramente interessato, interroga il publisher circa l'effettivo cambiamento
  - componenti – publisher e subscriber
  - connettori – un canale affidabile per la trasmissione delle notifiche



Observer

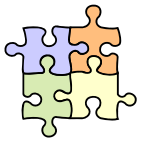
- Struttura della soluzione





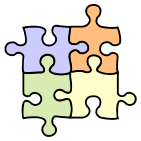
Observer

- Conseguenze
  - ☺ accoppiamento debole (astratto e minimale) tra il publisher e i suoi subscriber
  - ☺ supporto per comunicazione broadcast
  - ☺ possibilità di aggiungere/rimuovere i subscriber dinamicamente
  - ☺ rimuove la necessità del polling da parte dei subscriber
  - ☹ potrebbe essere difficile comprendere le relazioni di dipendenza tra i vari elementi
  - ☹ effetto imprevedibile degli aggiornamenti – una modifica in un publisher può scatenare una catena di aggiornamenti e sincronizzazioni su tutti i subscriber
  - ☹ implementazione complessa – se è richiesta una consegna affidabile dei messaggi



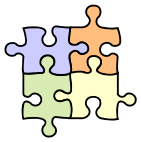
* Alcuni stili architetturali comuni

- Esistono numerosi stili architetturali di uso comune
  - alcuni vengono ora descritti brevemente
    - layers
    - client/server
    - pipes and filters
- La discussione degli stili architetturali sarà ripresa, ampliata e approfondita nel seguito del corso



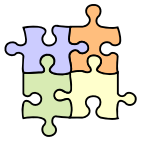
- Layers [POSA]

- **Layers** [POSA] è uno stile architetturale fondamentale
- Un singolo tipo di elemento (sono solitamente moduli) – lo *strato*
- Organizzazione degli elementi
  - elementi organizzati in una pila di strati
  - ogni strato fornisce servizi allo strato superiore e richiede servizi allo strato inferiore
  - strati organizzati per livello di astrazione
- Conseguenze
  - ☺ riuso di strati, buona separazione degli interessi, facilità di manutenzione, portabilità, ...
  - ☹ diminuzione della flessibilità di implementazione, possibile riduzione dell'efficienza, ...



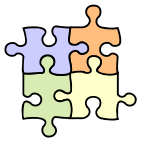
- Client/server

- **Client/server** è uno stile architetturale per sistemi distribuiti
- Due tipi di elementi (componenti) – sono processi
  - un *server* che offre uno o più *servizi* mediante un'interfaccia ben definita
  - uno o più *client* che usano i servizi come parte delle loro operazioni
    - si assume normalmente l'esistenza di una pluralità di istanze di client che agiscono in modo concorrente
- Un altro tipo di elementi (connettori)
  - un formato e un *protocollo* – definito dall'interfaccia del server
- Esempi di uso
  - molti servizi di Internet, accesso alle basi di dati, ...



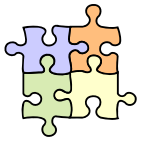
Client/server

- Conseguenze
  - ☺ condivisione di risorse, centralizzazione di elaborazione complessa o sensibile, ...
  - ☹ overhead della comunicazione, ...
- Alcune varianti comuni
  - stateless server – il server non gestisce lo stato delle conversazioni con i suoi client
  - stateful server – il server è responsabile anche della gestione dello stato delle conversazioni con i suoi client
  - client/server a più livelli – può essere considerato un altro stile



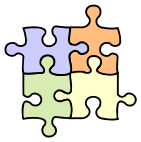
- Pipes and Filters [POSA]

- **Pipes and Filters** [POSA] è un altro stile architetturale fondamentale
- Problema
  - un sistema deve elaborare un flusso di dati
  - l'elaborazione può essere organizzata in una sequenza di trasformazioni
  - la specifica delle singole trasformazioni può cambiare nel tempo
  - l'uso di un singolo processo non è consigliato



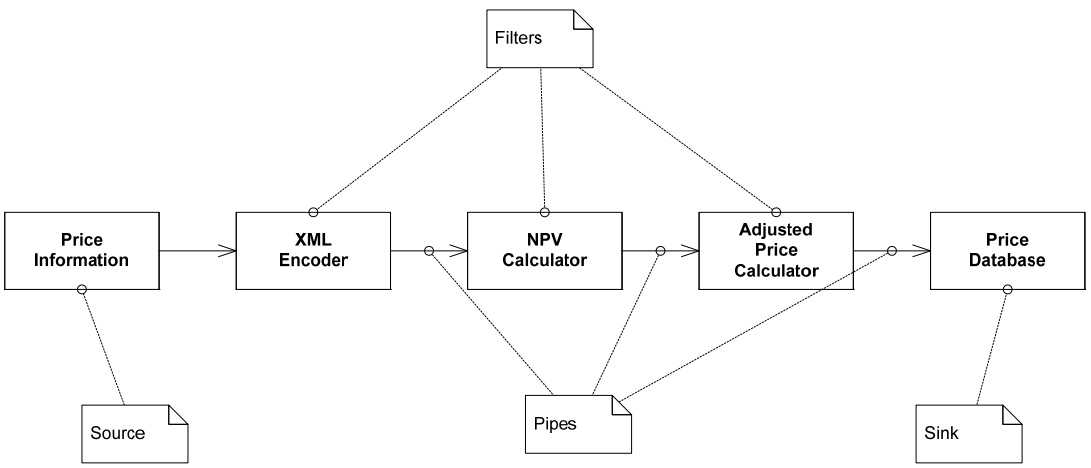
Pipes and Filters

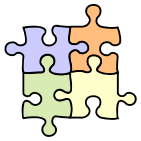
- Soluzione
  - organizza la trasformazione complessiva in una sequenza di passi – collega i passi sulla base del flusso di dati nel sistema
  - l'elaborazione è svolta da componenti *filtro* – che consumano ed elaborano dati in modo incrementale – con un flusso di ingresso ed un flusso di uscita
    - ogni filtro svolge una singola attività di elaborazione
  - i filtri sono connessi mediante connettori *pipe* (tubi) – che collegano flussi di uscita con flussi di ingresso consecutivi
    - le pipe sono l'unico modo consentito per connettere i filtri
    - le pipe definiscono un formato standard per i dati che possono attraversarle
  - la sequenza di filtri combinati da pipe è chiamata una *pipeline* di elaborazione



Esempio

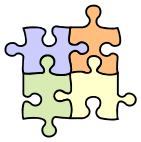
- Un flusso di informazioni su prezzi deve essere memorizzato in una base di dati
  - la rappresentazione nella base di dati è diversa da quella originale
  - le informazioni devono essere arricchite





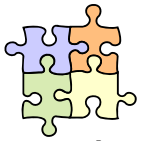
Pipes and Filters

- Conseguenze
  - ☺ la ricombinazione di filtri esistenti può consentire la definizione di nuove pipeline di elaborazione
  - ☺ i filtri possono essere riusati in situazioni diverse
  - ☺ l'implementazione dei filtri può essere cambiata senza modificare gli altri elementi del sistema
  - ☺ possibile l'elaborazione parallela, con filtri eseguiti in modo concorrente
  - ☺ non sono necessari file intermedi – ma possibili
  - ☹ la condivisione di informazioni di stato è difficile
  - ☹ la trasformazione dei dati in un formato inter-filtro comune può richiedere un overhead
  - ☹ la gestione degli errori è difficile



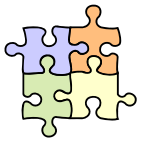
- Esempio

- Si consideri un sistema di trading finanziario
  - permette agli utenti di eseguire transazioni di compravendita di azioni
  - notifica informazioni (ad es., prezzi correnti) agli utenti
- Stili applicabili
  - client/server – per la gestione delle transazioni
    - elaborazione sicura, scalabile, affidabile, buone prestazioni
  - publisher/subscriber – per la notifica delle informazioni
    - diffusione delle informazioni efficace, flessibile, asincrona
  - layers – per l'organizzazione delle singole applicazioni
    - portabilità, modificabilità



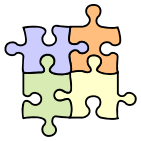
* Discussione

- È comune che ciascuna vista architetturale sia basata su uno stile architetturale "principale" – e talvolta che vengano usati ulteriori stili architetturali "secondari" in porzioni della vista
  - è utile annotare i modelli che formano l'AD con l'indicazione degli stili adottati e delle relative motivazioni
- Benefici nel basare un'architettura su uno stile riconoscibile
  - selezione di una soluzione provata e ben compresa, che definisce i principi organizzativi del sistema
  - più facile comprendere l'architettura e le sue caratteristiche – ovvero il modo in cui sono controllate le varie qualità
- Possibili usi degli stili architetturali
  - soluzione di progetto per il sistema in discussione
  - base per l'adattamento o per un nuovo stile
  - ispirazione per una soluzione correlata



Discussione

- Gli stili architetturali costituiscono un "bagaglio culturale" per l'architetto – che conoscendoli, è in grado i valutare quali possono essere applicati (e quali non vanno assolutamente applicati) nell'ambito di uno specifico sistema software
  - analogia con l'architettura civile
    - anche in questo caso è possibile identificare numerosi stili architettonici – ad es., *Chalet svizzero, Cattedrale gotica* e *Castello medioevale*
    - alcuni stili nascono da alcuni particolari circostanze (di tempo e di luogo) – ad esempio, lo *Chalet svizzero* ha il tetto di una forma particolare per evitare l'accumulazione di neve, e usa legno e pietra come materiali di costruzione, perché disponibili localmente
    - ha senso costruire uno *Chalet svizzero* a Moena? e a Roma? ai Caraibi?



Discussione

- Uno stile architetturale sostiene normalmente una certa combinazione di qualità
  - gli stili architetturali definiscono spesso l'applicazione di un certo numero di tattiche
    - ad es., Layers sostiene modificabilità applicando *Encapsulate* e *Restrict dependencies*
  - utile (all'architetto) comprendere le relazioni tra stili e tattiche architetturali
    - al limite, per "disapplicare" una tattica, per sostenere qualità diverse da quelle previste dal uno stile
    - ad es., posso "disapplicare" *Restrict dependencies* da Layers per migliorare le prestazioni