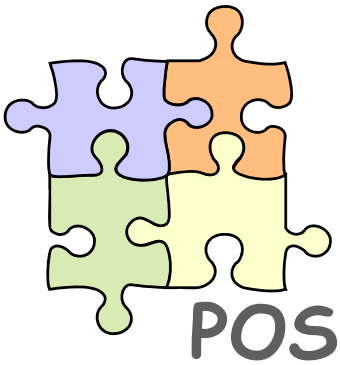
Luca Cabibbo



Architetture Software

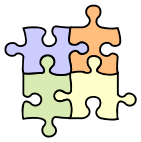
POSA: Un catalogo di pattern architetturali *(prima parte)*

Dispensa ASW 360

ottobre 2014

La struttura di un sistema dovrebbe assomigliare alla struttura funzionale.

Eberhardt Rechtin



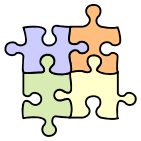
- Fonti

- [POSA1] Pattern-Oriented Software Architecture – A System of Patterns, 1996
- [POSA4] Pattern-Oriented Software Architecture – A Pattern Language for Distributed Computing, 2007
  - nota: [POSA] indica genericamente [POSA1] oppure [POSA4] – che sono parzialmente sovrapposti
- [DDD] Eric Evans, Domain-Driven Design, Tackling Complexity in the Heart of Software, 2004
- [Bachmann, Bass, Nord] Modifiability Tactics, Technical report CMU/SEI-2007-TR-002, 2007

- Obiettivi e argomenti

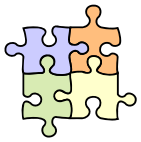
- Obiettivi
  - conoscere alcuni pattern architetturali [POSA] diffusi
  - esemplificare le relazioni tra pattern architetturali e tattiche
- Argomenti
  - introduzione
  - Domain Model [POSA4]
  - Domain Object [POSA4]
  - Layers [POSA]
  - Pipes and Filters [POSA]
  - Model-View-Controller [POSA]
  - Shared Repository [POSA]
  - Database Access Layer [POSA4]
  - Microkernel [POSA]
  - Reflection [POSA]
  - discussione

- Wordle



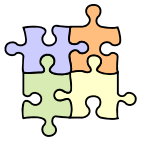
* Introduzione

- Un *pattern* (*software*)
  - la descrizione strutturata di una soluzione esemplare a un problema (software) ricorrente
- Uno *pattern architetturale* – o *stile architetturale*
  - un pattern che guida l'organizzazione di un'architettura software
  - gli elementi descrivono sotto-sistemi o comunque macro-componenti
- Un *pattern language* – *linguaggio di pattern*
  - una famiglia di pattern correlati
    - comprende anche una discussione correlazioni tra di essi
  - di solito, ogni pattern language è specifico per la progettazione di un certo tipo di sistema o per un certo tipo di qualità
    - ad es., per sistemi distribuiti o per la sicurezza



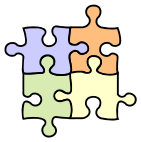
POSA: Un catalogo di pattern architetturali

- [POSA1], pubblicato nel 1996, identifica e descrive quattro categorie di pattern architetturali
  - dal fango alla struttura – ad es., layers, pipes-and-filters, ...
    - per sostenere una decomposizione controllata del sistema complessivo
  - sistemi distribuiti – ad es., broker, ...
    - per fornire un'infrastruttura per applicazioni distribuite
  - sistemi interattivi – ad es., model-view-controller, ...
    - per strutturare sistemi software che prevedono un'interazione uomo-macchina
  - sistemi adattabili – ad es., reflection, microkernel, ...
    - per sostenere l'adattamento del sistema – a fronte dell'evoluzione della tecnologia e/o dei requisiti funzionali



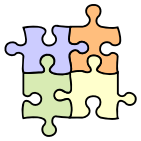
Una premessa

- **Big Ball of Mud** (è un anti-pattern) [POSA1]
  - a Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle
  - these systems show unmistakable signs of unregulated growth, and repeated, expedient repair
  - information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated
  - the overall structure of the system may never have been well defined – if it was, it may have eroded beyond recognition
  - programmers with a shred of architectural sensibility shun these quagmires – only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems



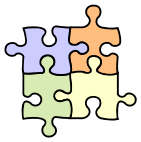
Altri cataloghi di pattern architetturali

- [POSA4], pubblicato nel 2007
  - definisce un linguaggio di pattern per sistemi distribuiti
  - rivisita e correla numerosi pattern architetturali definiti in precedenza
    - dai volumi precedenti della serie POSA
    - da Patterns of Enterprise Application Architecture [Fowler]
    - da Enterprise Integration Patterns [Hohpe&Woolf]
    - ...
- Esistono anche altri pattern e altri cataloghi...



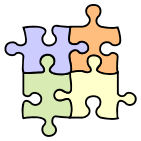
Benefici nell'uso degli stili architetturali

- 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
  - ispirazione per una soluzione correlata
  - motivazioni per un nuovo stile
- È possibile che un'architettura sia basata su più stili
  - ma in genere uno è dominante



Stili architetturali e tattiche

- Ciascuno stile architetturale è un "pacchetto di decisioni di progetto" [SAP] – per l'applicazione di una certa combinazione di tattiche – che possono riguardare attributi di qualità differenti
  - è pertanto importante comprendere/analizzare le tattiche che sono normalmente applicate da uno stile
    - quali sono e quali sono le loro conseguenze – questo fornisce una maggiore comprensione dello stile
    - quali altre tattiche possono essere applicate nel contesto di uno stile – e con quale effetto in tale contesto
    - quali tattiche – magari indesiderate – possono essere "disapplicate"
  - in alcuni casi faremo delle analisi in questo senso
  - più in generale, è importante comprendere come realizzare delle opportune *strategie architetturali* – basate sull'applicazione congiunta di stili e tattiche architetturali

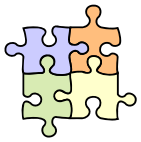


- A che punto siamo?

- In quale punto della definizione di un'architettura software è opportuno prendere in considerazione gli stili architetturali?
  - abbiamo già identificato gli scenari architetturali
  - tra di questi, abbiamo scelto un insieme di scenari più rilevanti
  - che cosa possiamo fare affinché l'architettura sostenga questi scenari più rilevanti?

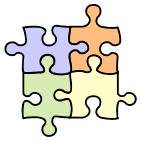
    questo è il punto in cui entrano in gioco gli stili architetturali

  - dobbiamo scegliere uno o più stili architetturali (uno per ciascuna vista) in grado di sostenere questi scenari più rilevanti
  - dobbiamo poi applicare questi stili architetturali alle varie viste – per identificare, in ciascuna vista, un insieme di elementi architetturali, e un insieme di relazioni e interazioni tra di essi
  - questa decomposizione dovrebbe sostenere gli scenari architetturali più rilevanti



A che punto siamo?

- In quale punto della definizione di un'architettura software è opportuno prendere in considerazione gli stili architetturali?
  - l'applicazione di uno stile architetturale (che richiede un insieme di scelte di progetto) ha impatto
    - sulla scelta di un insieme di elementi architetturali
    - nonché sulla scelta di un insieme di relazioni tra questi elementi
    - nell'ambito di una o più viste architetturali

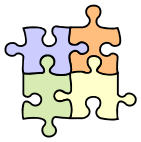


A che punto siamo?

- In quale punto della definizione di un'architettura software è opportuno prendere in considerazione gli stili architetturali?
  - in alternativa, abbiamo già effettuato una prima decomposizione architetturale – sulla base di uno o più stili architetturali
  - ma abbiamo capito che uno o più elementi architetturali sono troppo complessi – e devono essere ulteriormente decomposti
  - che cosa possiamo fare affinché uno di questi elementi architetturali complessi possieda gli attributi di qualità desiderati?

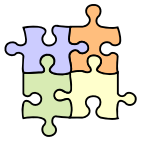
    anche qui possono entrare in gioco gli stili architetturali

  - uno stile architetturale può infatti essere usato anche per guidare la decomposizione di un elemento dell'architettura che altrimenti sarebbe troppo complesso



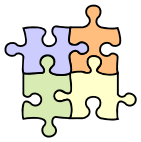
- Osservazione sulla notazione

- I pattern architetturali propongono criteri di decomposizione di un sistema in elementi architetturali (macro-elementi)
  - questi elementi sono spesso mostrati usando un linguaggio di modellazione ad oggetti – ad es., OMT o UML
  - gli elementi architetturali non sono mai dei semplici "oggetti"
    - sono piuttosto dei "macro-oggetti"
  - tuttavia, è comune che ciascun elemento abbia
    - un nome/riferimento
    - un'interfaccia pubblica – descrive i servizi che offre
    - un'implementazione privata
  - ed è anche comune che le interazioni tra elementi siano mostrati da scambi di messaggi (sincroni oppure asincroni)
- Dunque, una notazione ad oggetti è adeguata
  - ma i rettangoli indicano elementi architetturali, non oggetti



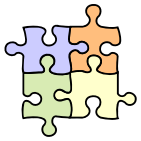
* Domain Model [POSA4]

- Il pattern architetturale **Domain Model**
  - il pattern architetturale più astratto, “radice” della gerarchia dei pattern architetturali POSA
  - essenzialmente, una chiave di lettura comune per diversi pattern architetturali



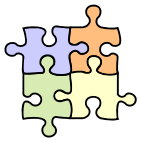
Domain Model

- Contesto
  - all’inizio della progettazione o costruzione di un sistema software
  - è necessaria una struttura iniziale per il software da sviluppare
- Problema
  - i requisiti descrivono funzionalità e qualità del sistema da sviluppare – ma non forniscono nessuna struttura concreta che possa guidare lo sviluppo
  - in mancanza di un’“intuizione” precisa e ragionata del dominio applicativo e della portata del sistema, la sua realizzazione rischia di essere “una grossa palla di fango” (“a big ball of mud”) – che è difficile da comprendere, da comunicare, da valutare e da usare come base per la costruzione del sistema



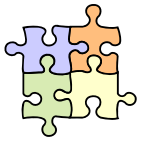
Domain Model

- Soluzione
  - crea, usando un metodo appropriato, un modello (*domain model*) che definisce e limita le responsabilità di business del sistema e le loro varianti
    - gli elementi nel modello sono astrazioni significative nel dominio applicativo – i loro ruoli e le loro interazioni riflettono il flusso di lavoro nel dominio
  - usa il modello di dominio come fondamenta per l'architettura software del sistema – l'architettura diventa un'espressione del modello, e i due possono evolvere insieme in modo coerente



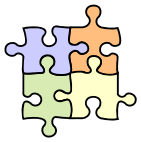
Domain Model

- In questo contesto, per *modello di dominio* si intende una qualche rappresentazione del dominio del problema di interesse per il sistema
  - si tratta di una nozione di modello di dominio certamente più generale che non secondo l'accezione usata da Larman – in cui un modello di dominio è un "modello a oggetti delle informazioni che il sistema deve gestire"
  - [POSA4] suggerisce di creare il modello di dominio usando "un metodo appropriato"
    - ad esempio, il metodo di [DDD] – di cui si parla più avanti
  - il tipo di "modello di dominio" da creare e il "metodo appropriato" dipendono fortemente dalle caratteristiche del sistema in discussione
  - si veda anche la discussione sul pattern Domain Object



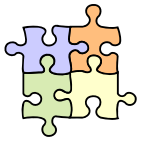
Domain Model

- Alcune possibile interpretazioni per “modello di dominio”
  - un modello dei casi d’uso
  - una decomposizione rappresentazionale delle informazioni del dominio
    - ad es., ad oggetti alla Larman, oppure secondo un diagramma ER
  - una decomposizione comportamentale
    - ad es., una decomposizione delle attività di un’organizzazione in servizi e processi, oppure basata sull’analisi dei flussi di dati
  - una decomposizione guidata da interessi tecnici
    - ad es., presentazione, logica applicativa, accesso a dati, ...
  - ...



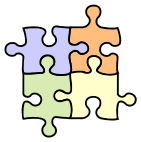
Domain Model e DDD

- Il pattern Domain Model accetta molte delle idee proposte nel libro Domain-Driven Design [DDD] di Eric Evans – ad esempio
  - un modello di dominio non è solo un diagramma – è l’idea stessa che il diagramma vuole comunicare
  - il modello di dominio è la struttura portante del linguaggio usato nella comunicazione tra le parti interessate
  - il modello di dominio e il progetto danno forma l’uno all’altro
  - bisogna abbandonare la dicotomia tra “modello di analisi” e “modello di progetto” – per cercare un modello unico che serva entrambi gli scopi
    - tuttavia, è certamente lecito usare un “modello di analisi” come punto di partenza per identificare un “modello di dominio”



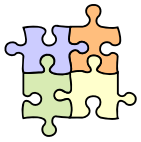
Domain Model e DDD

- In particolare, il pattern Domain Model deriva dal pattern *Model-Driven Design* di [DDD]
  - se il progetto, o qualche sua parte importante, non è in corrispondenza con il modello di dominio, allora il valore del modello è scarso, oppure la correttezza del progetto è sospetta – se invece c'è una corrispondenza ma è complessa, allora la realizzazione e/o manutenzione del sistema sarà problematica
  - pertanto, progetta una porzione del sistema software in modo che rifletta (in modo letterale) il modello di dominio, con una corrispondenza ovvia
  - rivisita continuamente il modello e il progetto, in modo che entrambi riflettano una profonda comprensione del dominio

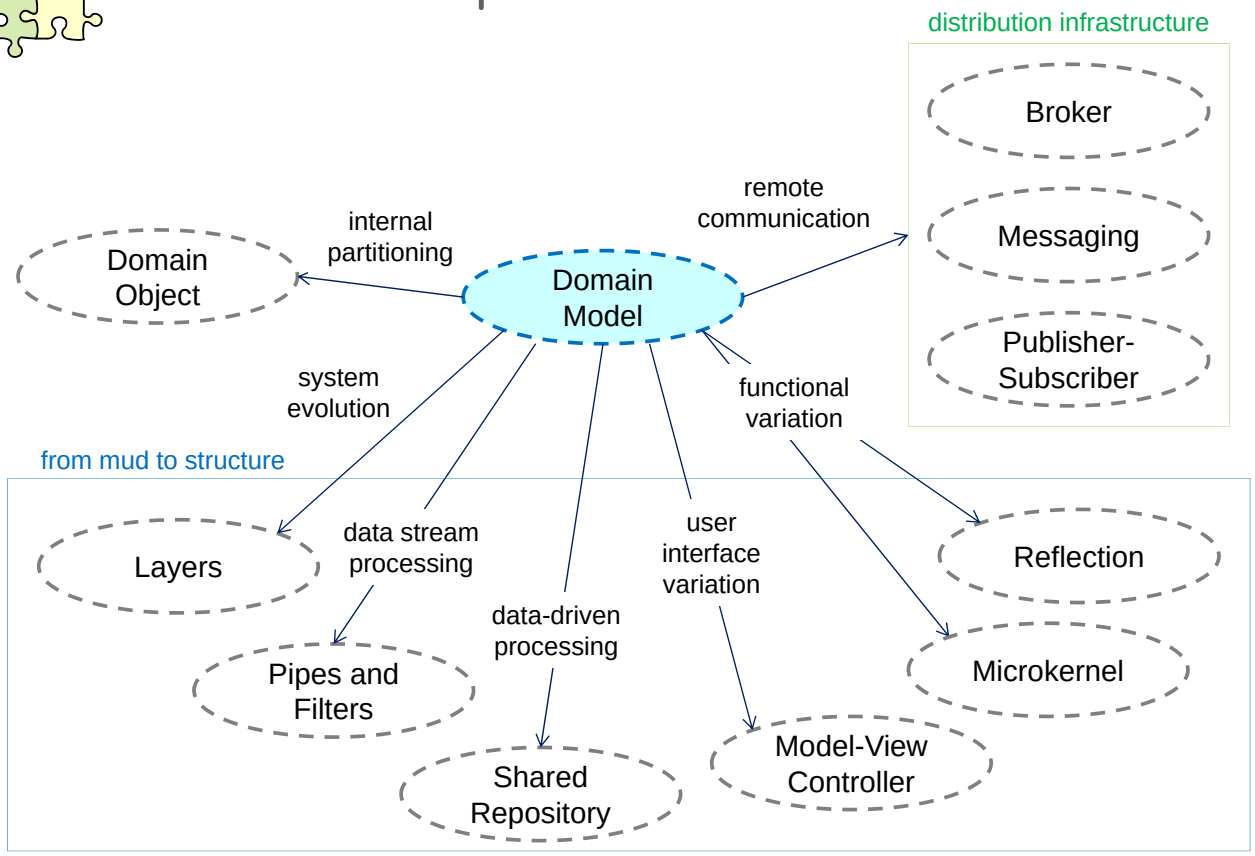


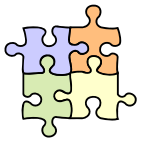
Discussione

- Un modello di dominio sostiene il passo iniziale nella definizione di un'architettura software
  - aiuta a identificare gli elementi software, e a definire le corrispondenze tra requisiti ed entità software
  - sostiene la comunicazione tra progettisti software, esperti di dominio e clienti – poiché gli elementi del modello di dominio fanno riferimento alla terminologia del dominio applicativo
- Ciascuna entità del dominio, auto-contenuta e con responsabilità coese, può essere rappresentata da un *Domain Object* separato
- Altri stili architetturali aiutano
  - ad organizzare e collegare gli elementi del modello di dominio, per sostenere uno specifico stile di computazione
  - a raggruppare e a separare elementi del modello di dominio – per sostenere gli interessi di modificabilità del sistema



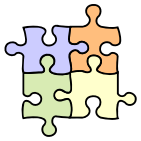
Relazioni tra pattern





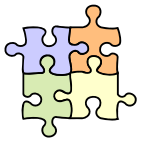
* Domain Object [POSA4]

- Il pattern architetturale **Domain Object**
  - guida la decomposizione di un intero sistema nella realizzazione di un Domain Model
  - può anche guidare la decomposizione di elementi architetturali grandi
    - ad es., uno strato di un'architettura secondo Layers
  - suggerisce i criteri di base per operare una tale decomposizione
    - il principio di separazione degli interessi e la modularità



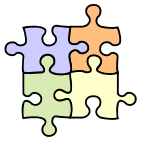
Domain Object

- Contesto
  - decomposizione di un intero sistema, oppure
  - decomposizione di un elemento architetturale
- Problema
  - le parti che formano un sistema software sono spesso caratterizzate da collaborazioni e relazioni di contenimento molteplici e variegate – realizzare queste funzionalità correlate senza cura può portare a un progetto con un'elevata complessità strutturale
  - la separazione degli interessi è una qualità fondamentale del software – ed è pertanto opportuno decomporre il sistema software in modo opportuno
  - questa decomposizione deve sostenere la realizzazione e l'evoluzione del sistema – e deve sostenere anche le sue qualità operazionali, come prestazioni e sicurezza



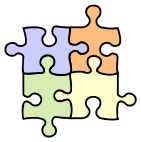
Domain Object

- Soluzione
  - incapsula ciascuna responsabilità *funzionale* distinta di un'applicazione in un elemento auto-contenuto – un *oggetto di dominio*
    - ciascun elemento deve avere un'interfaccia separata dalla sua implementazione
    - le collaborazioni tra gli elementi devono avvenire solo sulla base della loro interfaccia
    - ciascun elemento deve essere internamente coeso e debolmente accoppiato agli altri elementi



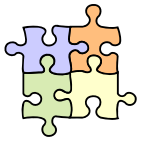
Domain Object

- Discussione
  - la decomposizione, coerentemente con *Domain Model*, può essere guidata da un opportuno modello del dominio
    - ciascun “oggetto di dominio” sarà poi usato per rappresentare un elemento del “modello di dominio”



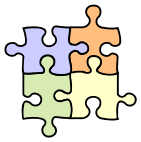
Domain Object

- La decomposizione va guidata da un qualche modello del dominio
  - la decomposizione può essere guidata dai casi d’uso
    - ad es., nello strato di presentazione, un elemento potrebbe rappresentare un gruppo di casi d’uso correlati, per uno stesso attore primario
  - il criterio può essere una decomposizione rappresentazionale delle informazioni del dominio
    - ad es., per lo strato di dominio – vedi Larman e UML Components
  - si può anche usare una decomposizione comportamentale
    - ad es., un’organizzazione a servizi e processi, basata su un modello delle attività, dei processi e/o di flussi di dati
  - la decomposizione può anche essere guidata da interessi tecnici
    - ad es., nello strato dei servizi tecnici



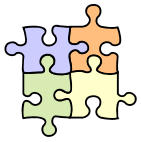
Domain Object

- Discussione
  - il pattern Domain Object è basato sull'applicazione del principio di separazione degli interessi e del principio di modularità
    - la separazione degli interessi è relativa al fatto che ciascun oggetto di dominio rappresenta un elemento distinto del modello di dominio
    - la modularità è relativa a incapsulamento, coesione e accoppiamento
      - è opportuno tenere alta la coesione interna di ciascuna parte e tenere basso l'accoppiamento tra le diverse parti
      - questo ne favorisce, ad esempio, sviluppo ed evoluzione indipendenti da quello di altre parti del sistema

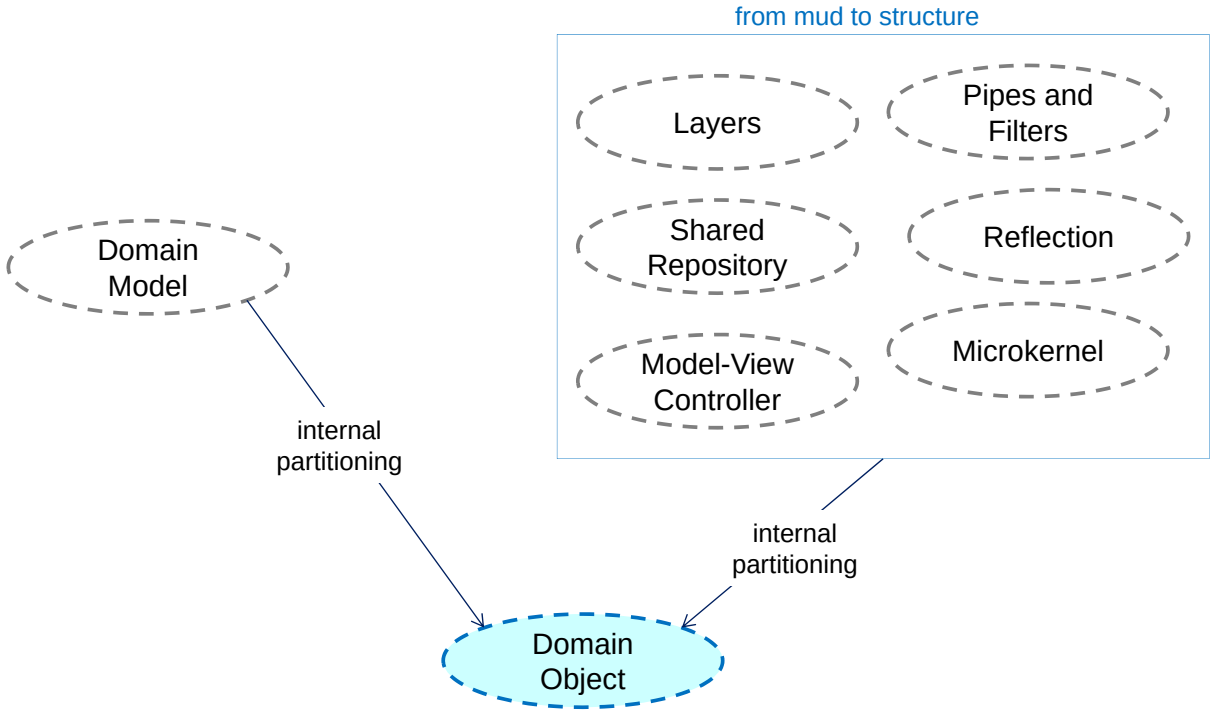


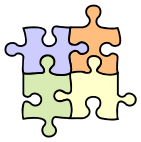
Domain Object

- Discussione
  - la soluzione suggerisce anche che la decomposizione sia relativa a responsabilità *funzionali*
    - dunque, non suggerisce di utilizzare elementi come "gestore della disponibilità" o "gestore della modificabilità"
    - tuttavia, la decomposizione non può essere indipendente dalle qualità (*non funzionali*) che il sistema deve possedere – ad es., prestazioni, affidabilità, ...



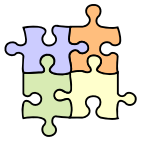
Relazioni tra pattern





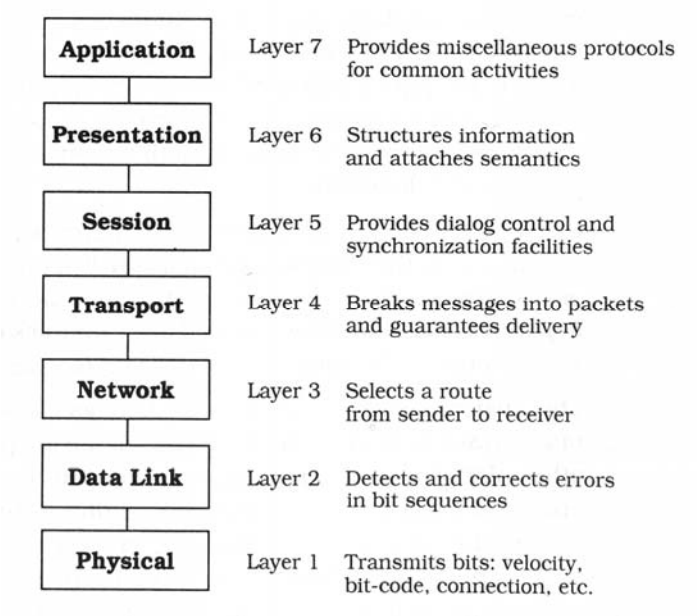
* Layers [POSA]

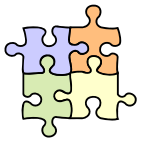
- Il pattern architetturale **Layers**
  - aiuta a strutturare applicazioni che possono essere decomposte in gruppi di compiti
  - in cui ciascun gruppo di compiti è a un particolare livello di astrazione



Esempio

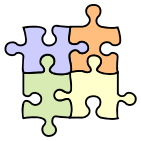
- Un esempio molto noto di architettura a strati – protocolli di rete
  - ogni strato si occupa di uno specifico aspetto della comunicazione – di compiti ad uno specifico livello di astrazione





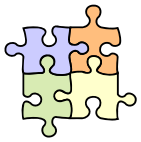
Layers

- Contesto
  - un sistema grande – richiede di essere decomposto
  - le diverse parti del sistema devono poter essere sviluppate e devono poter evolvere in modo indipendente tra loro
- Problema
  - il sistema si deve occupare della gestione di diversi aspetti – ad esempio, a differenti livelli di astrazione o granularità
    - ad es., un sistema per la tariffazione di telefonate – che da una parte deve gestire dati che vengono da sensori, e dall'altra politiche di alto livello come le tariffe telefoniche
  - le diverse parti del sistema
    - devono poter essere sviluppate indipendentemente
    - devono anche poter evolvere in modo indipendente
  - è richiesta modificabilità, portabilità e riusabilità



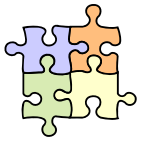
Layers

- Soluzione
  - partiziona il sistema in una gerarchia verticale di elementi – ciascuno dei quali è chiamato uno *strato*
  - ciascuno strato ha una responsabilità distinta e ben specifica
    - tutti i componenti nell'ambito di ciascuno strato lavorano per soddisfare quella responsabilità
  - costruisci le funzionalità di uno strato in modo che dipendano effettivamente solo dallo stesso strato o da strati inferiori
    - ciascuno strato può comunicare solo con lo strato sottostante (talvolta anche con più strati sottostanti)
    - ovvero, gli strati sono ordinati rispetto alla relazione di "uso"
  - fornisci, in ciascuno strato, un'interfaccia che è separata dalla sua implementazione
    - fa sì che la comunicazione tra gli strati avvenga solo tramite queste interfacce



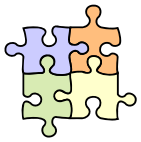
Layers

- Soluzione – ecco le caratteristiche del generico strato J
  - gestisce un certo tipo di responsabilità
  - fornisce (tramite un'interfaccia) servizi allo strato J+1 – ovvero, al suo strato "superiore"
  - richiede servizi/delega compiti allo strato J-1 (tramite la sua interfaccia) – ovvero, al suo strato "inferiore"
  - collabora con lo strato J-1 – ovvero, con il suo strato "inferiore"



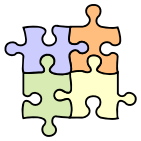
Sulla natura degli strati

- Finora, niente è stato detto sulla natura degli elementi “strati”
  - l’interpretazione più comune (ad es., [SAP]) è che questi elementi sono dei moduli
  - ma sono possibili anche interpretazioni in cui gli strati sono elementi funzionali, componenti runtime, processi o componenti di deployment, …
- Ad esempio
  - se gli strati sono moduli
    - la comunicazione tra strati (i connettori) potrebbe essere realizzata come chiamata di procedure (locali)
  - se gli strati sono processi
    - la comunicazione tra strati (i connettori) deve essere realizzata mediante un meccanismo di comunicazione interprocesso – ad es., RPC o RMI
  - ...



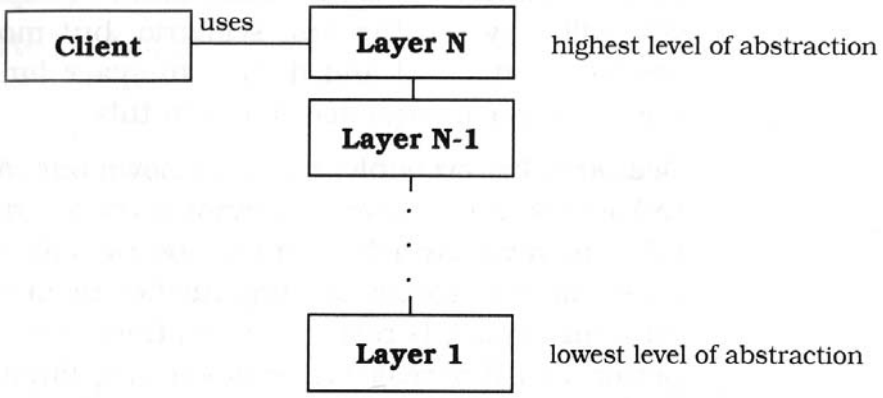
Pattern, soluzione e scenari

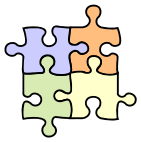
- La *soluzione* di un pattern descrive il *principio* o l’*idea risolutiva* fondamentale del pattern
  - la soluzione comprende una descrizione della *struttura statica* e una descrizione del *comportamento dinamico* del pattern
- In particolare, la dinamica di un pattern può essere descritta tramite un insieme di scenari
  - ciascuno *scenario* descrive un possibile comportamento dinamico archetipale della soluzione
    - un possibile modo di comportarsi degli elementi di un pattern
    - per fornire un certo comportamento (funzionale)
    - oppure per descrivere come è possibile controllare una certa qualità (non funzionale)
  - gli scenari possono anche descrivere modi diversi (varianti) di applicare uno stesso pattern (inteso come idea risolutiva)



Scenario 1 – comunicazione top-down

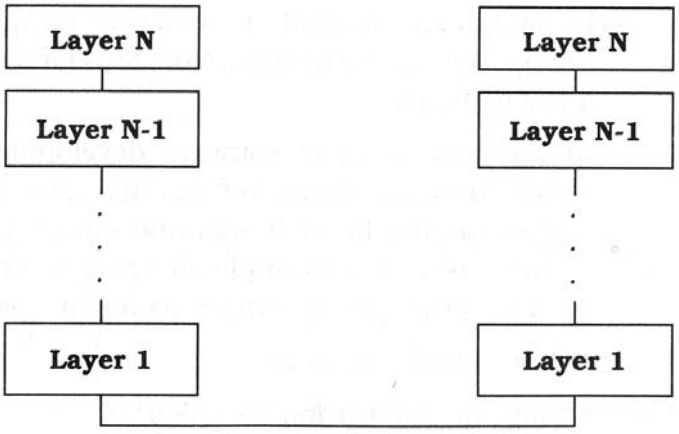
- Lo scenario più comune per Layers – la *comunicazione top-down*
  - un client effettua una richiesta al livello N
  - la richiesta viene via via decomposta in sotto-richieste di livello N-1, N-2 – scendendo tra gli strati
  - le eventuali risposte alle sotto-richieste vengono via via combinate – risalendo tra gli strati

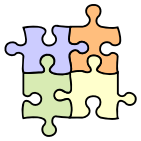




Scenario 2 – comunicazione bottom-up

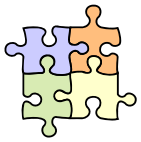
- Un altro scenario comune – la *comunicazione bottom-up*
  - al livello più basso arrivano delle notifiche – ciascuna descrive il verificarsi di un evento, insieme ai relativi dati
  - queste notifiche (e i relativi dati) risalgono, venendo interpretate opportunamente
- Ad esempio, una parte della comunicazione con la pila ISO-OSI





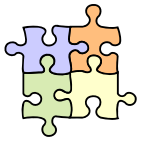
Altri scenari

- Altri scenari comuni sono relativi ai casi in cui le richieste/notifiche attraversano solo alcuni livelli – senza attraversare l'intera pila di strati
  - una richiesta al livello N viene servita ad un livello intermedio, ad es., al livello N-1 o N-2
    - ad es., c'è una cache
  - una notifica dal livello 1 viene gestita ad un livello intermedio, ad es., al livello 2 o 3
    - ad es., ricezione di un messaggio ripetuto ma già ricevuto – il messaggio viene scartato senza arrivare al livello N
  - una richiesta viene gestita da un sottoinsieme degli strati delle due pile
    - ad es., se viene trovato un errore in un pacchetto, viene effettuata una richiesta di ritrasmissione nell'ambito di uno strato intermedio



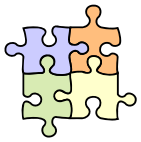
Applicazione di Layers

- Un pattern architetturale
  - definisce un certo numero di tipologie di elementi architetturali e di relazioni tra elementi architetturali
  - fornisce anche delle linee guida per la sua applicazione – relative soprattutto alla scelta degli specifici elementi architetturali e delle relazioni tra di essi
- Un aspetto cruciale nell'applicazione di Layers è la scelta degli strati e delle responsabilità assegnate a ciascuno strato
  - ovvero, la scelta del criterio di partizionamento delle funzionalità tra i diversi strati
  - in particolare, si vuole che ciascuno strato
    - incapsuli un ben preciso gruppo di funzionalità
    - possa essere sviluppato ed evolvere indipendentemente dagli altri strati – ovvero, dagli altri gruppi di funzionalità



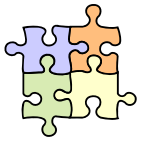
Layers – Applicazione (1/3)

- Definisci il criterio di partizionamento delle funzionalità
  - nella formulazione originale di [POSA1], il criterio di riferimento per il partizionamento delle funzionalità è il “livello di astrazione”
    - ad es., possibili livelli di astrazione in un’applicazione per il gioco degli scacchi: pezzi (es., alfiere), mosse (es., arrocco), tattiche (es., difesa siciliana), strategia di gioco complessiva
  - per sostenere modificabilità, il criterio di partizionamento deve essere legato ai cambiamenti attesi – di solito può essere motivato da *Increase semantic coherence*
  - alcuni criteri di partizionamento comuni
    - uso di livelli di astrazione come presentazione, logica applicativa e servizi tecnici
    - granularità, ad es., con uno strato per i processi di business e un altro strato per i servizi di business
    - distanza dall’hardware, dipendenza dall’applicazione, tasso di cambiamento atteso, ...



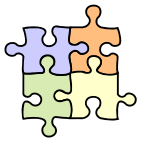
Layers – Applicazione (2/3)

- Determina gli strati – uno strato per ciascun livello di astrazione
  - ovvero, determina il numero degli strati, ma soprattutto i compiti (le responsabilità) di ciascuno strato
- Specifica i servizi offerti da ciascuno strato
- Raffina la definizione degli strati – in modo iterativo
- Definisci l’interfaccia di ciascuno strato
  - quali i servizi offerti da ciascuno strato?
  - quali le notifiche accettate?
- Struttura individualmente gli strati – descritto dopo



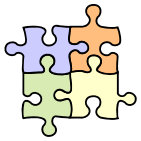
Layers – Applicazione (3/3)

- Specifica la comunicazione tra strati
  - comunicazione top-down – le richieste scendono
  - comunicazione bottom-up – le notifiche salgono
- Disaccoppia gli strati – usa opportuni design pattern
  - ad es., uso di Facade, Observer oppure di callback
- Progetta una strategia per la gestione degli errori
  - le eccezioni possono essere talvolta gestite nello strato in cui si verificano – altre volte negli strati superiori



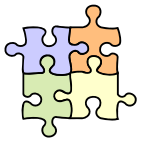
Conseguenze

- Modificabilità
  - ☺ la modificabilità può essere alta – ma solo se le dipendenze sono opportunamente localizzate
  - ☺ in particolare, è possibile sostenere la portabilità, usando strati dedicati alla piattaforma di esecuzione
  - 😐 in realtà, la modificabilità dipende da come i cambiamenti si ripercuotono sul sistema – la modificabilità è alta se i cambiamenti sono localizzati in singoli componenti/strati
  - ☹ tuttavia, se i cambiamenti che si verificano sono diversi da quelli che sono stati presi in considerazione nell'organizzazione delle funzionalità del sistema, allora questi cambiamenti potranno coinvolgere molti strati – con un effetto domino



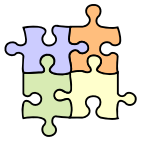
Conseguenze

- Prestazioni
  - ☹ le funzionalità sono spesso implementate attraverso più strati – possono essere richiesti diversi cambiamenti di contesto, penalizzando le prestazioni
  - 😐 in genere, allocare ciascuno strato ad un diverso processo (concorrenza) non migliora le prestazioni – talvolta le può peggiorare
  - ☺ viceversa, in alcuni casi è possibile migliorare le prestazioni associando un diverso thread di esecuzione a ciascun evento che deve essere elaborato dal sistema



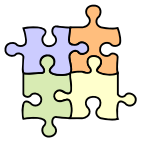
Conseguenze

- Affidabilità
  - ☹ spesso i dati devono essere elaborati da molti strati – questo diminuisce l'affidabilità
  - ☺ tuttavia, è possibile usare l'organizzazione a strati per far sì che uno strato più alto contenga funzionalità per gestire guasti che si verificano negli strati inferiori
  - ☺ è possibile introdurre degli strati intermedi per effettuare il monitoraggio del sistema – ad es., per controllare che i dati scambiati tra gli strati siano "ragionevoli" (*Sanity checking*)



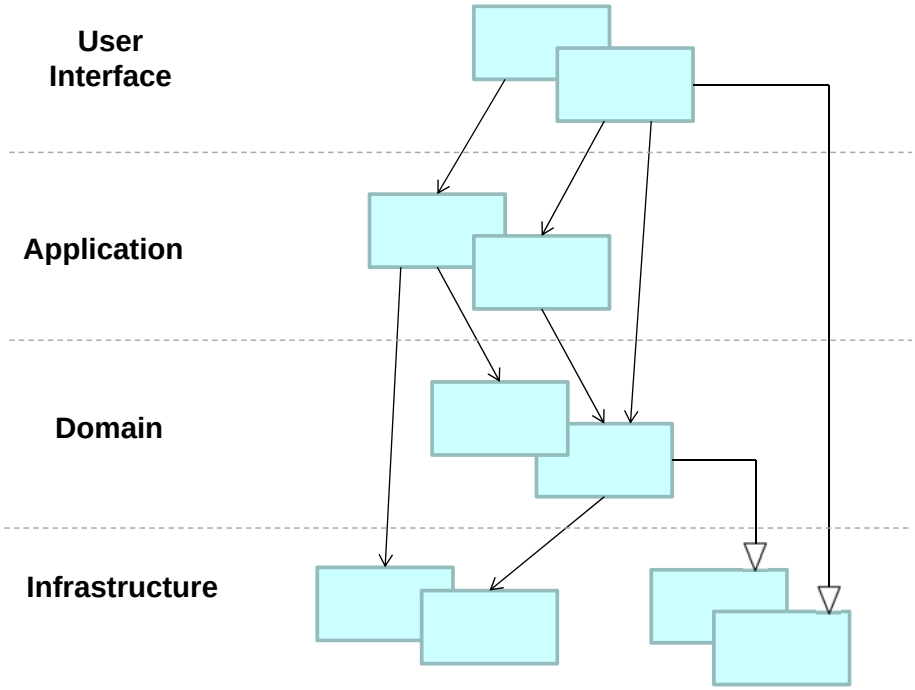
Conseguenze

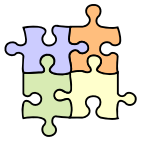
- Sicurezza
  - ☺ è possibile inserire strati che introducono opportuni meccanismi di sicurezza – ad es., autenticazione, autorizzazioni, crittografia, ...
- Altre conseguenze
  - ☺ possibilità di riusare strati
  - 😐 l'uso degli strati può aumentare gli sforzi iniziali necessari e la complessità del sistema – ma questi sforzi poi sono di solito ripagati
  - ☹ può essere difficile stabilire la granularità/il numero/il livello di astrazione degli strati



- Layered Architecture [DDD]

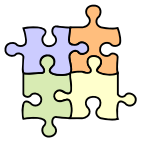
- Una scelta comune per gli strati di un'applicazione è, ad esempio, quella suggerita da [DDD] – *Layered Architecture*





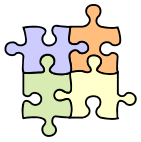
Layered Architecture [DDD]

- Una scelta comune per gli strati di un'applicazione è, ad esempio, quella suggerita da [DDD] – *Layered Architecture*
  - *presentation layer* – UI o presentazione
    - mostra informazioni all'utente e interpreta le sue richieste
  - *application layer* – applicazione
    - definisce un'API unificata per sostenere la realizzazione di più UI o client
  - *domain layer* – dominio
    - rappresenta il modello di dominio
  - *infrastructure layer* – infrastruttura
    - fornisce capacità tecniche per supportare gli strati superiori – ad esempio, l'accesso ai dati



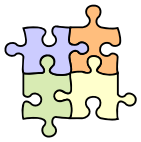
Layered Architecture [DDD]

- Secondo [DDD], in una *Layered Architecture*
  - il *domain layer* contiene il codice che rappresenta il modello di dominio
    - è isolato dal codice degli strati di presentazione, applicazione e infrastruttura
    - gli oggetti di dominio – liberi dalla responsabilità di visualizzarsi, di memorizzarsi e di gestire compiti dell'applicazione – si possono focalizzare sulla rappresentazione del modello di dominio
    - è responsabile di gestire lo stato complessivo dell'applicazione (*attenzione all'ambiguità con il nome dello strato applicazione!*)
      - questo stato può essere gestito anche con l'ausilio di una base di dati



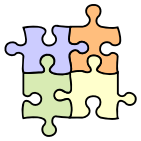
Layered Architecture [DDD]

- Secondo [DDD], in una *Layered Architecture*
  - l'*application layer* è lo strato che definisce i compiti che il sistema è chiamato a fare – ovvero, le "operazioni di sistema"
    - implementazione: due varianti principali
      - implementazione mediante un insieme di facade sottili, che delegano lo svolgimento delle operazioni di sistema a oggetti opportuni dello strato del dominio – ovvero, un insieme di controller (nell'accezione ***GRASP***)
      - implementazione mediante un insieme di classi più spesse che definiscono degli "operation script" – ovvero, che (1) implementano direttamente la logica applicativa ma (2) delegano la logica di dominio allo strato del dominio
    - di solito, è responsabile anche di gestire lo stato delle conversazioni (sessioni) con i suoi client



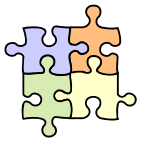
- Discussione

- [POSA4] colloca Layers nella categoria "system evolution"
  - la qualità che viene sostenuta principalmente da Layers è infatti la modificabilità – Layers va applicato appunto quando la modificabilità è una qualità desiderata importante
    - se la modificabilità è la qualità desiderata più importante, allora deve essere alla base del criterio di decomposizione utilizzato



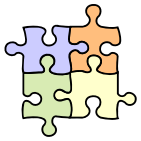
Discussione

- Layers è basato sul principio di *modularità*
  - un progetto è *modulare* se è caratterizzato da accoppiamento basso e coesione alta
    - le responsabilità di ciascuno elemento devono essere coese (ovvero, logicamente correlate) tra loro e non accoppiate a (ovvero, mischiate con) responsabilità di altri elementi
  - we propose that one begins with a list of difficult design decisions or design decisions which are likely to change – each module is then designed to hide such a decision from others [Parnas, 1972]
    - il suggerimento è incapsulare ciascuna dimensione di cambiamento atteso in un elemento diverso – in modo che ciascun cambiamento atteso abbia impatto (se possibile) su un singolo elemento (ovvero, possa essere gestito modificando un solo elemento) – ma non gli altri



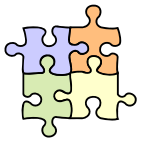
Discussione

- In effetti, [POSA4] suggerisce che l'applicazione di Layers sia basata, *in generale*, su
  - un partizionamento delle funzionalità del sistema o sottosistema – in modo che ciascun gruppo di funzionalità sia chiaramente incapsulato in uno strato e possa evolvere indipendentemente (dagli altri gruppi di funzionalità)
- Tuttavia, il criterio di partizionamento *specifico* può essere definito secondo varie dimensioni – ad esempio
  - astrazione – presentazione, logica applicativa e dati
  - dipendenza dall'applicazione – specifico per l'applicazione, specifico per il dominio, indipendente dal dominio
  - granularità – processi, servizi, entità
  - distanza dall'hardware – ad es., con strati per l'astrazione dal sistema operativo e dal canale di comunicazione
  - tasso di cambiamento atteso



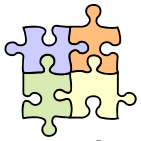
Discussione

- Qual è la motivazione per la scelta degli strati, ad esempio, nella *Layered Architecture* di [DDD]?
- Ci sono almeno due possibili spiegazioni per questa scelta
  - livello di astrazione
    - la decomposizione, basata sul principio di *separazione degli interessi*, tende a separare interessi diversi (livelli di astrazione diversi) in elementi differenti
    - in genere, interessi diversi sono caratterizzati da richieste di cambiamento tra di loro indipendenti
  - tasso di cambiamento atteso – c'è un'evidenza empirica che
    - le interfacce utente tendono a cambiare più rapidamente della logica applicativa – che cambia più rapidamente della logica di dominio – che a sua volta cambia più rapidamente della struttura dei dati persistenti
    - meglio mantenere gli elementi più stabili negli strati più bassi



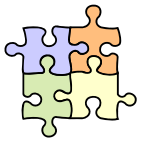
- Usi conosciuti

- Il pattern Layers è applicato in modo pervasivo
  - ad esempio, nei sistemi informativi e nei sistemi basati su macchine virtuali
  - l'architettura client/server corrisponde ad un'applicazione di Layers
    - le architetture applicative di Java EE e .NET sono architetture client/server a più livelli
  - l'architettura ANSI a tre livelli dei DBMS (esterno, logico, interno) è un'architettura a strati
    - che sostiene l'indipendenza dei livelli
  - l'architettura orientata ai servizi è un'architettura a strati
    - sostiene agilità di business
  - l'architettura del cloud computing viene descritta a strati
  - ...



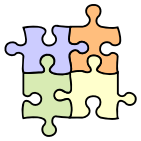
- Layers e tattiche per la modificabilità

- È possibile comprendere come Layers sostiene modificabilità con riferimento al modo in cui Layers implementa/può essere applicato per implementare alcune delle relative tattiche
  - questa discussione raffina quella fatta informalmente in precedenza, e mostra la correlazione tra pattern architetturali e tattiche architetturali



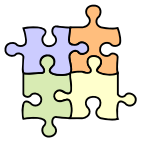
Layers e tattiche per la modificabilità

- *Increase semantic coherence – increase cohesion*
  - la tattica di mantenere la coerenza semantica ha l'obiettivo di garantire che le responsabilità di un elemento lavorino tutte insieme – senza far eccessivo affidamento ad altri elementi
    - l'effetto desiderato è legare insieme responsabilità che possono essere interessate da un cambiamento
  - questo obiettivo può essere raggiunto assegnando responsabilità agli strati in modo che abbiano una qualche coerenza semantica
  - ad es., separando le responsabilità di gestione dell'hardware da quelle di natura applicativa
    - infatti, questi due tipi di responsabilità non hanno di solito coerenza semantica



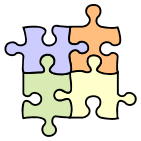
Layers e tattiche per la modificabilità

- *Abstract common services – reduce coupling*
  - uno strato può essere utilizzato per fornire servizi comuni ed astratti agli strati superiori
    - ad es., uno strato può raggruppare responsabilità per la gestione della memoria – e fornire il servizio corrispondente
  - nell'architettura a strati, i servizi possono esser resi astratti e collocati in uno strato (immediatamente) inferiore a quello del/dei consumatore/i del servizio
  - inoltre, gli strati possono essere utilizzati per definire una "scala" di servizi astratti – poiché ciascuno strato può alzare il livello di astrazione dei servizi offerti dallo strato inferiore
    - i moduli/servizi negli strati più bassi possono avere una rappresentazione più generale e astratta negli strati superiori
    - ad es., un driver concreto per un particolare dispositivo ed un'interfaccia per il driver di un dispositivo generico



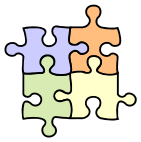
Layers e tattiche per la modificabilità

- *Encapsulate – reduce coupling*
  - l'incapsulamento è basato su una separazione tra interfaccia pubblica ed implementazione privata di un elemento
  - nell'architettura a strati ciascuno strato espone la propria interfaccia
    - modifiche relative all'implementazione di uno strato ma non alla sua interfaccia non si ripercuotono sugli strati superiori
    - viceversa, modifiche relative all'interfaccia di uno strato possono richiedere adattamenti negli strati superiori



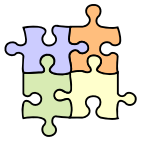
Layers e tattiche per la modificabilità

- *Use an intermediary – reduce coupling*
  - un intermediario ha lo scopo di ridurre/rompere le dipendenze tra due elementi – delegando la gestione della dipendenza all'intermediario
  - nell'architettura a strati è possibile che uno strato abbia lo scopo di definire un'interfaccia per uno strato inferiore
    - ad es., una facade
  - cambiamenti nello strato inferiore possono essere nascosti agli strati superiori sulla base di un adattamento dello strato che funge da intermediario



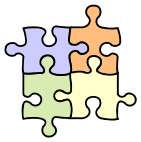
Layers e tattiche per la modificabilità

- *Restrict dependencies – reduce coupling*
  - la dipendenza relativa ad una necessità di comunicazione viene rimossa incanalando la comunicazione in un intermediario
  - nell'architettura a strati, ciascuno strato costituisce un intermediario tra gli strati più in alto e gli strati più in basso
    - questa limitazione sulle dipendenze tra strati ha l'effetto, in particolare, di limitare le conseguenze della sostituzione di uno strato – importante, ad es., ai fini della portabilità



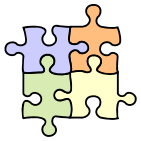
Layers e tattiche per la modificabilità

- Un commento sull'architettura a strati rilassata – in cui uno strato J può dipendere da tutti gli strati a esso sottostanti e non solo dallo strato J-1
  - è basata sulla rimozione di intermediari e di *Restrict dependencies*
    - aumenta le dipendenze e riduce la modificabilità
  - può essere motivata da *Reduce overhead*
    - di solito la rimozione di intermediari ha lo scopo di migliorare le prestazioni



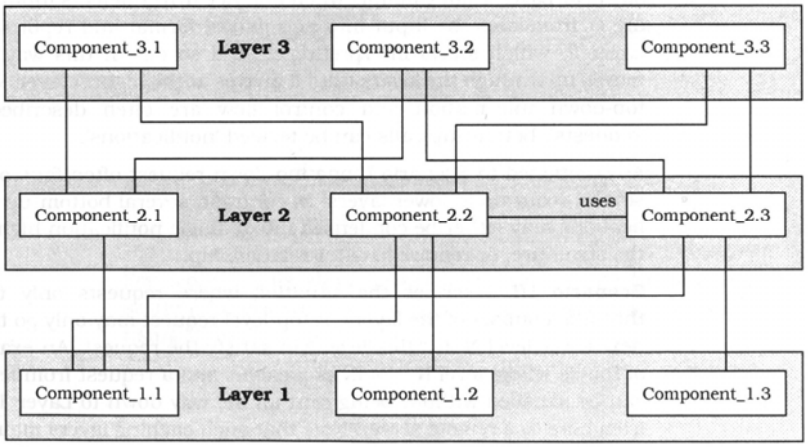
- Sull'applicazione di Layers

- Il pattern Layers può essere applicato
  - nella decomposizione/strutturazione di un intero sistema
  - nella decomposizione/strutturazione di un sotto-sistema del sistema
    - ad es., un singolo componente potrebbe essere internamente organizzato a strati
- Ciascuno strato può/deve essere strutturato internamente – sulla base di altri pattern architetturali

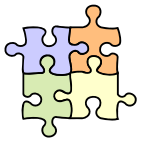


Sull'applicazione di Layers

- Ciascuno strato può/deve essere strutturato internamente – sulla base di altri pattern architetturali
  - ciascuno strato può essere internamente composto da più componenti

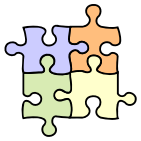


  - questa ulteriore decomposizione può essere basata su Domain Object



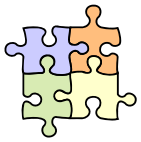
* Pipes and Filters [POSA]

- Il pattern architetturale **Pipes and Filters**
  - fornisce una struttura per sistemi che devono elaborare flussi di dati
    - appartiene alla categoria "data stream processing" di [POSA4]
  - l'elaborazione complessiva è decomposta in passi di elaborazione
  - ciascun passo di elaborazione è incapsulato in un *filtro*
  - i dati sono trasferiti tra filtri adiacenti mediante *pipe* (tubi)
  - è possibile costruire famiglie di sistemi correlati mediante un'opportuna combinazione di filtri e pipe – *pipeline*



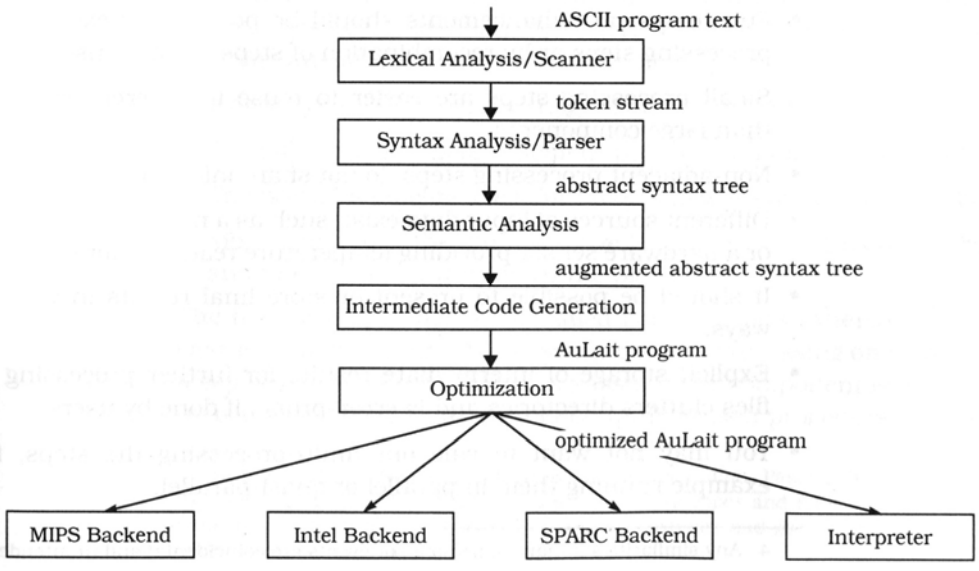
Elaborazione di flussi di dati

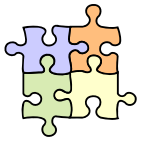
- Il problema affrontato da Pipes and Filters è “elaborare flussi di dati”
  - ovvero, trasformare un flusso di dati in ingresso in un flusso di dati in uscita
  - elaborazioni di questo tipo sono molto più comuni e rilevanti di quanto si possa pensare – ad esempio
    - trasformare un flusso di ordini in un flusso di spedizioni
    - gestire flussi di pratiche assicurative – per generare rimborsi
    - gestire flussi di pratiche studenti
    - ma anche elaborare grandi quantità di dati – data analytics
  - in corrispondenza, si parla di applicazioni guidate da flussi di dati (data-flow-driven application)
    - può essere utilizzata una modellazione di dominio basata sulla descrizione di questi flussi di dati



Esempio

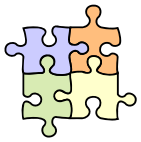
- Si vuole definire un nuovo linguaggio di programmazione (Mocha) – e costruire un compilatore portabile – basato tra l’altro su un linguaggio intermedio (AuLait)





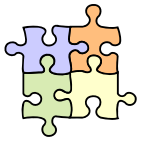
Pipes and Filters

- Contesto
  - un sistema (o componente) deve elaborare flussi di dati
- Problema
  - l'applicazione (o componente) deve elaborare flussi di dati
    - flussi di ingresso vanno trasformati in flussi di uscita
  - una decomposizione in elementi che comunicano mediante meccanismi di tipo richiesta/risposta è considerata inadeguata
  - è possibile descrivere la trasformazione del flusso di dati in termini di un modello "data-flow"
    - l'elaborazione complessiva può essere organizzata come un gruppo di passi di elaborazione successivi
    - i dettagli relativi a ciascun passo potrebbero cambiare indipendentemente dagli altri
    - i passi potrebbero essere svolti in parallelo



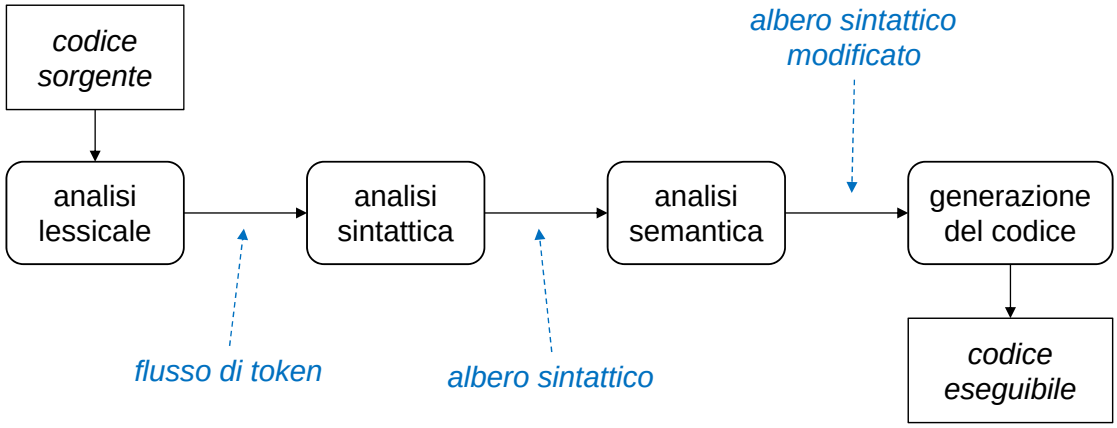
Pipes and Filters

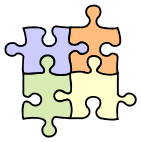
- Soluzione
  - suddividi il compito complessivo di elaborazione in una sequenza di passi di elaborazione dei dati auto-contenuti
  - implementa ciascun passo di elaborazione mediante un componente *filtro* – un filtro consuma dati da un flusso di input e genera dati in un flusso di output, in modo incrementale
  - collega questi passi (filtri) in una *pipeline* di elaborazione dei dati – la pipeline definisce il compito complessivo di elaborazione, modellando il flusso di dati complessivo dell'applicazione
  - collega filtri successivi nella pipeline tramite connettori *pipe* – ciascuna pipe è un buffer di dati intermedio usato per collegare una coppia di filtri successivi, mantenendone però basso l'accoppiamento



Esempio

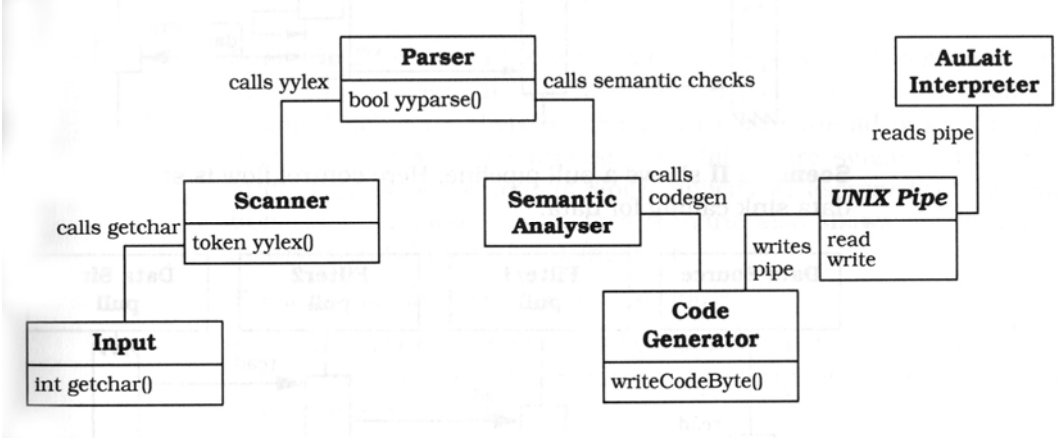
- La compilazione di un programma
  - ha lo scopo di trasformare un flusso di caratteri in un flusso di istruzioni binarie
  - può essere svolta come una sequenza di passi, con riferimento ad alcune rappresentazioni intermedie

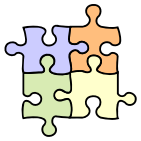




Esempio

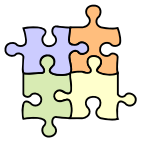
- Il compilatore può usare
  - alcuni filtri standard – lex per l'analisi lessicale e yacc per l'analisi sintattica
  - ulteriori filtri specifici – ad es., per la generazione del codice
  - le pipe di Unix – come pipe





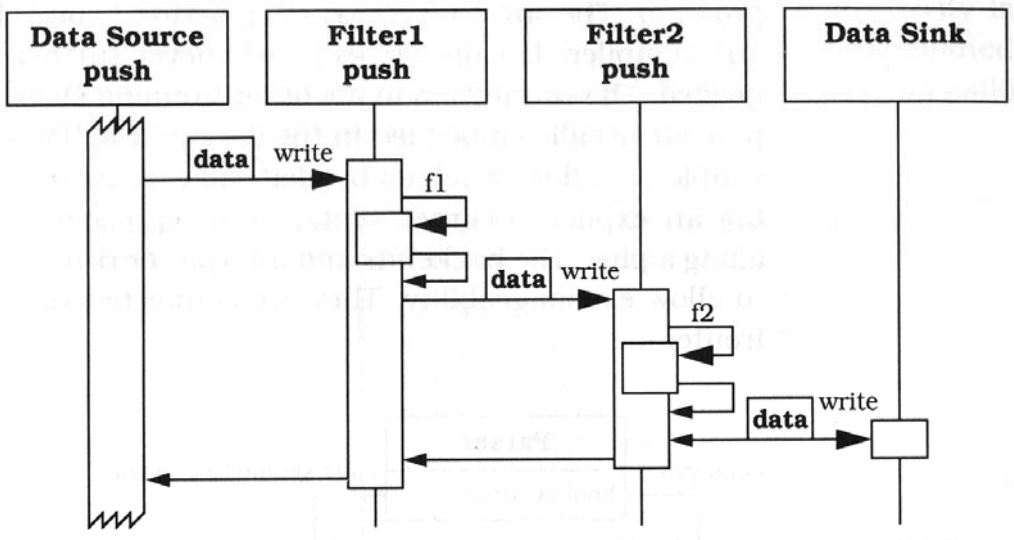
Scenari

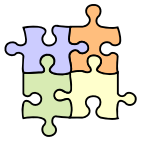
- Sono possibili diverse realizzazioni (scenari) di Pipes and Filters
  - nei diversi scenari, il flusso dei dati va sempre nella stessa direzione, dalla sorgente verso la destinazione
    - tuttavia, la gestione del controllo è diversa da scenario a scenario
  - in alcuni scenari (1,2,3), c'è un solo elemento attivo
    - i filtri potrebbero essere moduli – filtri, sorgente e destinazione potrebbero vivere in un solo processo
    - le pipe potrebbero essere realizzate mediante chiamate dirette e sincrone
  - in altri scenari, più comuni e più significativi (4), ci sono più filtri/componenti attivi – che vivono in processi o thread diversi
    - le pipe possono essere asincrone, e definire dei buffer di dati intermedi – ad es., pipe di Unix, destinazioni in un sistema di messaging



Scenario 1

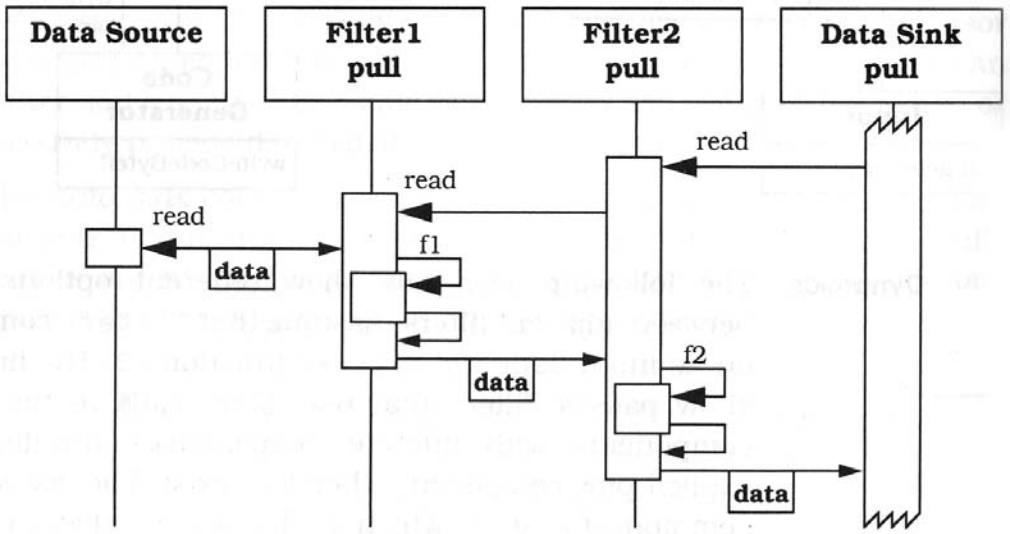
- *Pipeline di tipo push*
  - l'elaborazione inizia nella sorgente dei dati
  - filtri passivi – potrebbero essere chiamate dirette

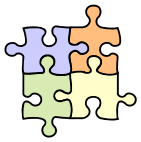




Scenario 2

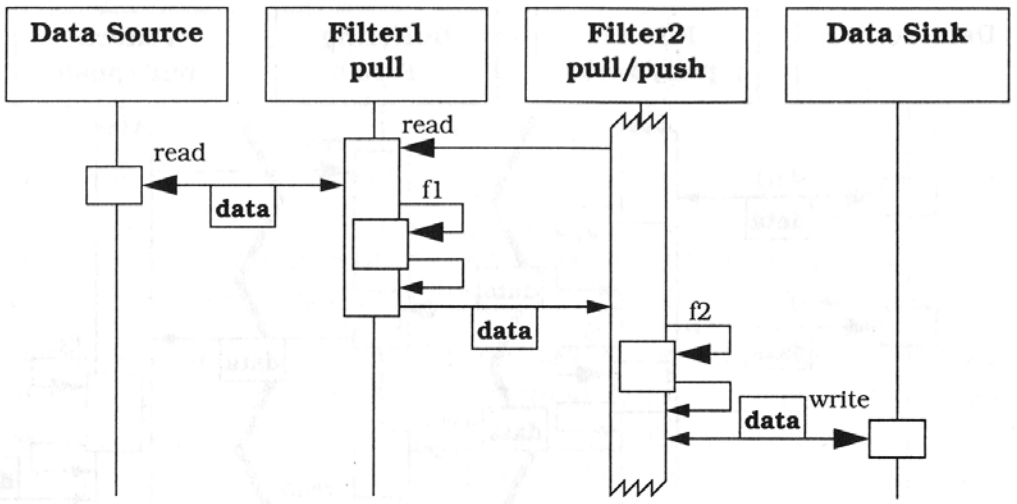
- *Pipeline di tipo pull*
  - l'elaborazione inizia dalla destinazione dei dati
  - filtri passivi – potrebbero essere chiamate dirette





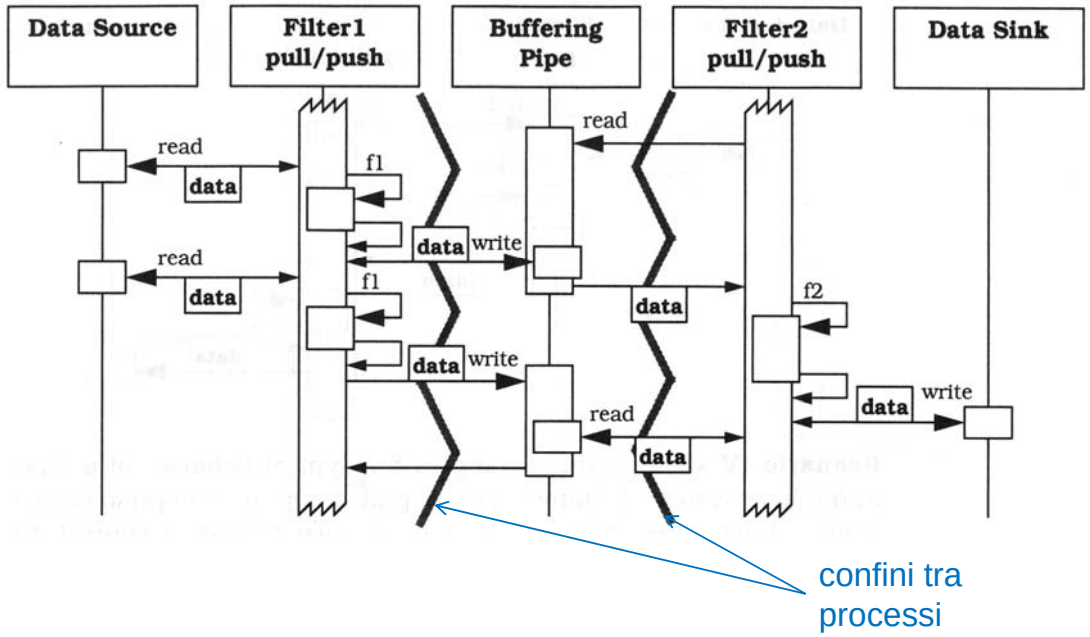
Scenario 3

- Uno scenario misto pull-push
  - nell'esempio solo il filtro 2 ha un ruolo attivo
  - filtro 1, sorgente e destinazione sono passivi – potrebbero essere chiamate dirette



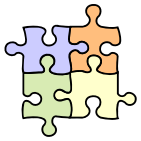
Scenario 4

- Un altro scenario – più complesso – è lo scenario più comune



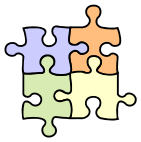
Scenario 4

- Un altro scenario – più complesso – è lo scenario più comune
  - sono presenti più filtri attivi
    - ciascun filtro attivo opera in modalità pull/push
    - ciascun filtro opera nell'ambito di un proprio thread/processo
    - dunque, i diversi filtri operano in modo asincrono e concorrente
  - le pipe
    - ciascuna pipe svolge anche il ruolo di buffer – per sincronizzare i filtri che collega
    - le pipe riducono l'accoppiamento tra i filtri che collegano – ad esempio, un filtro di solito non conosce l'identità dei filtri adiacenti



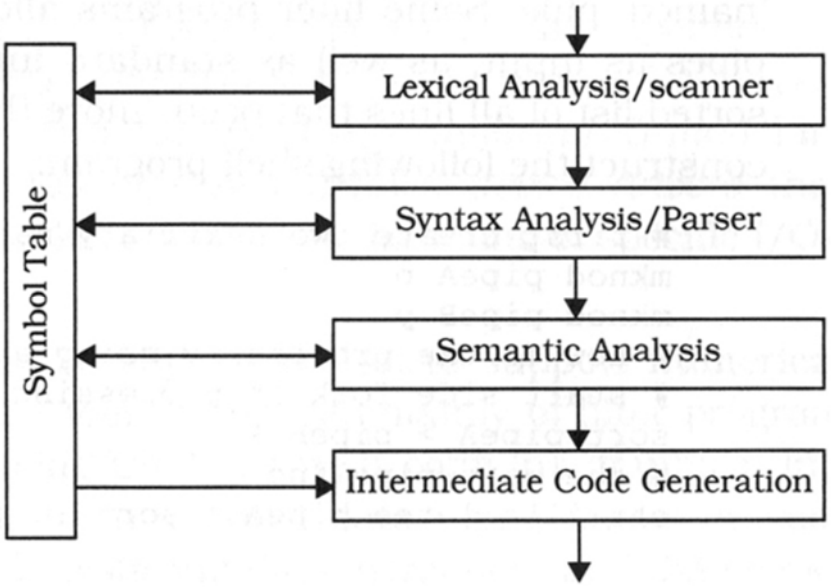
Pipes and Filters – Applicazione

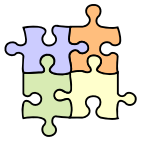
- Suddividi il compito da svolgere in un gruppo di passi di elaborazione
- Definisci il formato dei dati scambiati dai filtri
- Decidi come implementare le connessioni pipe
  - quali filtri sono attivi?
  - i filtri passivi sono attivati in modo push o pull?
  - potrei avere chiamate dirette tra filtri – ma per cambiare la pipeline dovrei cambiare il codice
  - la soluzione che usa meccanismi di pipe o altri connettori è in genere più flessibile
  - possibili soluzioni ad-hoc – ad es., filtri come thread diversi di uno stesso processo
- Progetta e implementa i filtri
- Progetta per la gestione degli errori
- Metti in piedi la pipeline di elaborazione



Esempio

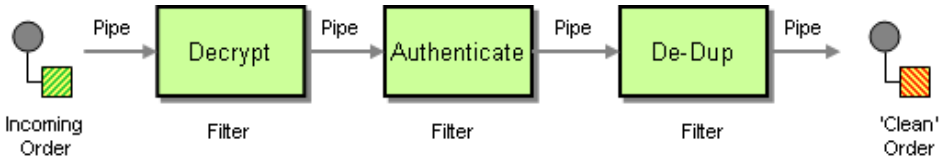
- Nei compilatori, il pattern Pipes and Filters non viene applicato in modo stretto
  - viene utilizzata anche una tabella dei simboli condivisa

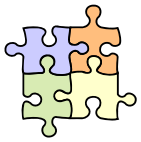




Esempio

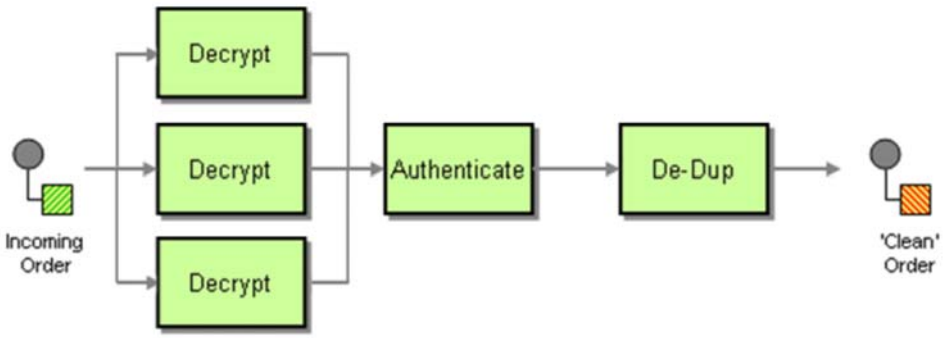
- Si supponga che un sistema riceva ordini sotto forma di messaggi
  - gli ordini sono cifrati – per motivi di sicurezza
  - gli ordini contengono un certificato che garantisce l'identità del cliente
  - è possibile che arrivino messaggi ripetuti – i duplicati vanno eliminati
  - vogliamo trasformare
    - un flusso di ordini cifrati, con dati aggiuntivi e con possibili duplicati
    - in un flusso di ordini "in chiaro", senza informazioni ridondanti e senza duplicati

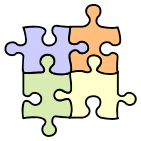




Esempio (cont.)

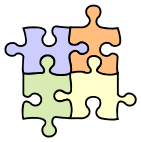
- Si supponga che un sistema riceva ordini sotto forma di messaggi
  - vogliamo trasformare – un flusso di ordini cifrati, con dati aggiuntivi e con possibili duplicati – in un flusso di ordini "in chiaro", senza informazioni ridondanti e senza duplicati
  - vogliamo evitare la penalizzazione delle prestazioni dovuta al fatto che l'operazione di decifratura è molto più lenta di quella di autenticazione





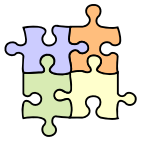
Conseguenze

- Modificabilità
  - ☺ la modificabilità (intesa come possibilità di cambiare la trasformazione operata dal sistema) è buona nella misura in cui la nuova trasformazione può essere realizzata come "ridefinizione" della pipeline
  - ☺ la flessibilità offerta da P&F è basata soprattutto su
    - la possibilità aggiungere/sostituire/rimuovere filtri – da una libreria di filtri preesistenti – oppure definendo nuovi filtri o modificando filtri esistenti
    - la presenza e l'uso di pipe – sono un'indirezione per ridurre l'accoppiamento tra filtri – in genere, infatti, i filtri non conoscono né gli altri filtri né le pipe che li collegano
    - la possibilità di riorganizzare la pipeline – in diversi momenti della vita del sistema, ad es., talvolta anche a runtime
  - ☹ tuttavia, in alcuni casi la modificabilità può essere bassa – se non è disponibile una libreria ricca di filtri, o se sono necessari cambiamenti in cascata su più filtri



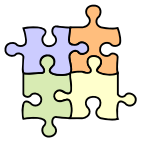
Conseguenze

- Prestazioni
  - ☺ i filtri costituiscono delle unità molto buone per la concorrenza
  - ☺ è possibile pensare di avere più istanze runtime di uno stesso filtro, che operano in parallelo
  - ☺ potrebbe non essere necessario usare file per i risultati intermedi
  - ☺ poiché i filtri sono connessi da pipe, l'interfaccia per ciascun filtro deve essere limitata – questo riduce il numero dei punti di sincronizzazione
  - ☹ tuttavia, c'è un overhead dovuto al trasferimento continuo dei dati tra i filtri (ciascuno dei quali esegue normalmente solo una piccola quantità di lavoro) – nonché al cambiamento di contesto e spesso anche del formato dei dati
  - 😐 il guadagno di efficienza potrebbe essere solo un'illusione



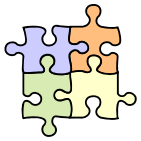
Conseguenze

- Affidabilità
  - 😐 difficile fare considerazioni generali – dipende dalla topologia della pipeline
  - ☹ tuttavia, spesso i dati devono essere elaborati da molti filtri – questo diminuisce l'affidabilità
  - ☺ d'altra parte, se i dati attraversano uno o pochi filtri, può essere facile verificare il sistema
  - ☺ l'affidabilità può essere sostenuta da un'opportuna infrastruttura – ad esempio, MapReduce oppure un middleware per il messaging affidabile



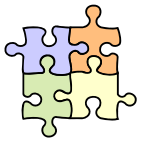
Conseguenze

- Sicurezza
  - ☺ i sistemi e i componenti Pipes and Filters sono normalmente basati su un'interfaccia piccola e ben definita
  - ☺ può essere semplice introdurre meccanismi di sicurezza (autenticazione, autorizzazioni, crittografia) sull'intero sistema e/o sui singoli componenti
- Altro
  - ☺ possibilità di riusare componenti filtro
  - ☹ la condivisione di dati tra più elementi è difficile – costosa o poco flessibile
  - ☹ la gestione degli errori è di solito difficoltosa
  - ☹ questo stile di solito non è adatto a sistemi interattivi



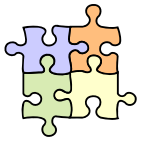
Una variante – Tee and join pipeline system

- *Tee and join pipeline system* è una variante di Pipe and Filters
  - i filtri possono avere più di un ingresso e/o più di una uscita
    - tee di Unix
  - l'elaborazione non è una pipeline, ma un grafo diretto
    - possibile anche avere cicli – ma la comprensione e la verifica diventano più difficili



Esempio: esecuzione di query relazionali

- Tee and join pipeline è usato per implementare l'esecutore di interrogazioni nei DBMS relazionali – implementazione dell'*algebra relazionale*
  - interrogazioni relazionali espresse come alberi
    - le foglie sono relazioni, i nodi sono operatori
  - operatori relazionali (effettivi)
    - scan, selezione, proiezione, rimuovi duplicati, ordina, join (hash-join, merge-join, nested loop, ...), semi-join, raggruppa, ...
  - un modulo per ciascun operatore
  - i moduli-operatori sono istanziati (tramite processi o thread) nell'esecuzione di un'interrogazione, e collegati da pipe
    - sulla base della struttura dell'albero per l'interrogazione

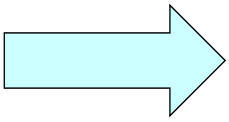


Esempio: esecuzione di query relazionali

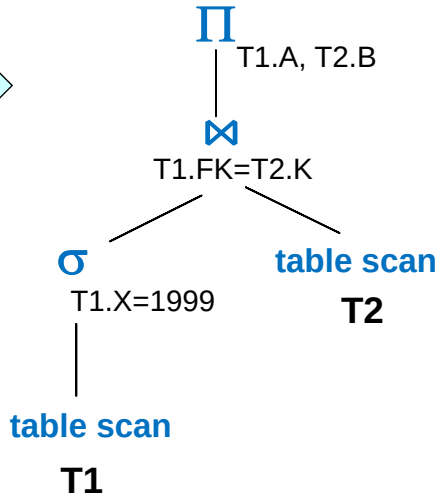
- Un esempio di query relazionale

SQL Query:

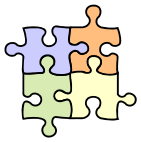
```
SELECT  T1.A, T2.B
FROM    T1, T2
WHERE   T1.FK=T2.K
AND     T1.X=1999;
```



Execution Plan:

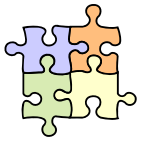


  - l'esecuzione della query potrà essere basata sull'allocazione di un thread/processo per operatore, collegati tramite una pipeline tee-and-join



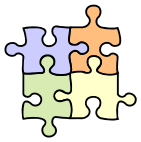
Discussione

- Pipes and Filters suggerisce una decomposizione dell'elaborazione in passi di elaborazione distinti – implementati da componenti filtro
  - sostiene un approccio di elaborazione incrementale
  - i passi identificati possono evolvere in modo indipendente
- Le pipe
  - svolgono il ruolo di connettori tra componenti filtro
    - comunicazione tra filtri
    - sincronizzazione tra filtri
  - sostengono l'accoppiamento debole tra i filtri
  - in un'architettura distribuita, possono essere realizzate mediante un'infrastruttura di messaging – sulla base dei pattern Messaging e Publisher-Subscriber (discussi successivamente)



Discussione

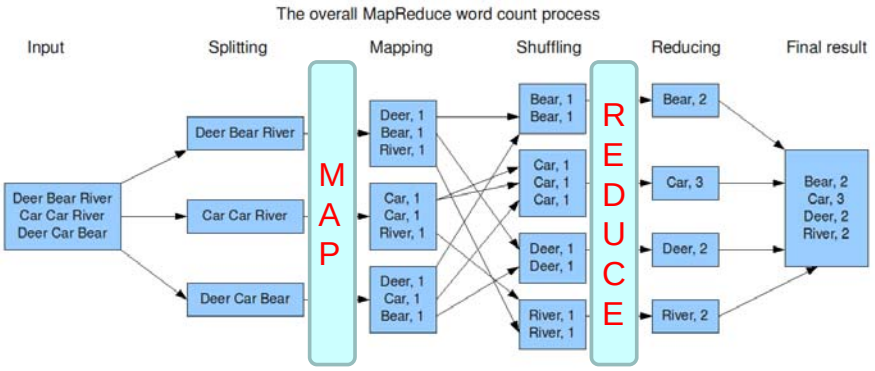
- L'organizzazione della pipeline può essere guidata da una modellazione del dominio di tipo data-flow – ovvero, effettuata in termini di flussi di dati e attività
  - i filtri corrispondono a unità di elaborazione specifiche del dominio – ciascun filtro può essere identificato come un Domain Object
  - una pipe implementa una politica di buffering e movimentazione di dati tra filtri – talvolta anche le pipe possono essere considerate come Domain Object

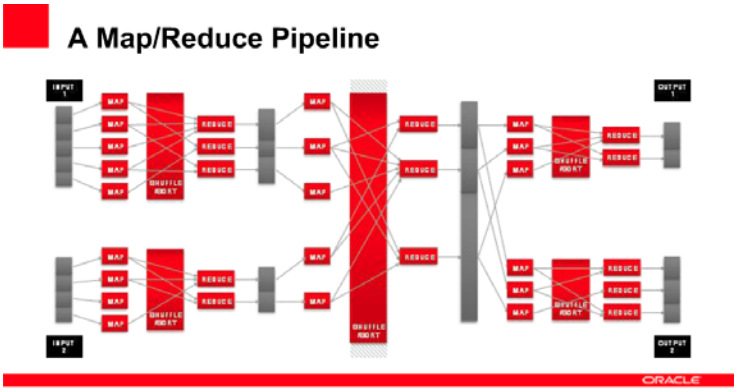


- Usi conosciuti

- Alcuni usi conosciuti del pattern Pipes and Filters
  - nello scripting in Unix, sulla base della disponibilità delle pipe | nonché di numerosi programmi filtro – ad esempio
    - grep HTTP_USER_AGENT httpd_state_log | cut –f 2- | grep –v 'via gateway' | grep –v 'via proxy gateway' | sort | uniq –c | sort –nr | head -5
    - i primi 5 browser (esclusi agenti proxy) da un log www
  - nel server web Apache, nell'elaborazione di richieste
  - in sistemi di calcolo scientifico
  - Pipes and Filters è alla base delle infrastrutture di messaging e dei sistemi di workflow – importanti nelle architetture a servizi
  - Yahoo!Pipes – a powerful composition tool to aggregate, manipulate, and mashup content from around the web
  - MapReduce – a programming model for processing large data sets with a parallel, distributed algorithm on a cluster

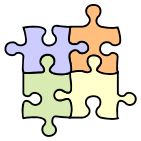
Esempio: MapReduce





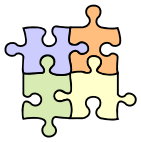
- P&F e tattiche per la modificabilità

- *Increase semantic coherence – increase cohesion*
  - ciascun filtro è definito in corrispondenza a un passo di elaborazione autocontenuto
  - le responsabilità di un filtro lavorano insieme per arricchire, raffinare o trasformare i dati di ingresso al filtro
  - inoltre, queste responsabilità operando indipendentemente e non dipendono dagli altri filtri
  - la coerenza semantica è dunque usata come criterio con cui sono raggruppate le responsabilità che sono allocate a un filtro, ovvero per il modo in cui esse manipolano dati o informazioni



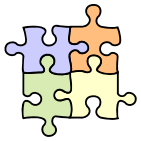
P&F e tattiche per la modificabilità

- *Encapsulate – reducing coupling*
  - le responsabilità pubbliche di un filtro sono relative al formato in cui il filtro è in grado di accettare o produrre dati
    - alcuni sistemi usano lo stesso formato di scambio per tutti i filtri – in altri casi sono usati più formati specializzati
  - le responsabilità private di un filtro sono relative alle elaborazioni che esso effettua



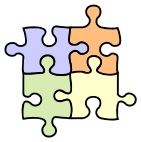
P&F e tattiche per la modificabilità

- *Use an intermediary – reduce coupling*
  - le pipe rompono alcune delle dipendenze (ma non tutte) tra filtri adiacenti – fornendo bufferizzazione e sincronizzazione
    - tuttavia, le pipe non rompono le dipendenze relative alla sintassi dei dati
  - se necessario, è possibile utilizzare degli intermediari per proteggere i filtri anche da cambiamenti nella sintassi dei dati



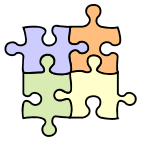
P&F e tattiche per la modificabilità

- *Restrict dependencies – reduce coupling*
  - una sequenza di filtri combinata tramite pipe forma una pipeline
  - le pipe sono intermediari che rompono le dipendenze di comunicazione tra filtri
  - inoltre, i percorsi di comunicazione sono ulteriormente limitati dal fatto che ciascun filtro ha un singolo ingresso e una singola uscita
  - questo vincolo limita il numero di dipendenze tra filtri e, dunque, il numero delle responsabilità a cui una modifica può propagarsi



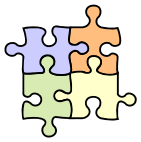
P&F e tattiche per la modificabilità

- *Defer binding*
  - la pipeline, ovvero il collegamento tra filtri e pipe, può essere creata quando vengono invocati i filtri
    - ad esempio, nello scripting in UNIX viene usato un binding al momento dell'avvio dello script
    - nei sistemi di elaborazione delle interrogazioni, il binding avviene a runtime



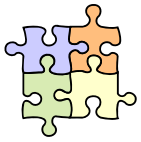
P&F e tattiche per la modificabilità

- Un commento sull'architettura tee & join pipeline
  - tee & join pipeline consente di avere filtri con più ingressi/uscite
    - si tratta di una rimozione di *Restrict dependencies*
  - tee & join pipeline consente di definire trasformazioni più complesse
    - probabilmente a scapito della modificabilità del sistema



* Discussione

- Alcuni dei pattern architetturali che sono stati mostrati – in particolare, quelli nella categoria [POSA] "dal fango alla struttura"
  - guidano la decomposizione architetturale "fondamentale" di un sistema – o di un componente di un sistema
  - ciascun pattern/stile
    - identifica alcuni particolari tipi di elemento e delle particolari modalità di interazione tra questi elementi
    - descrive criteri per effettuare la decomposizione sulla base di questi tipi di elemento e delle possibili relazioni tra essi
    - discute il raggiungimento (o meno) di proprietà di qualità
  - il criterio di identificazione degli elementi/componenti fa comunemente riferimento a qualche modalità di modellazione del dominio del sistema



Punto della situazione