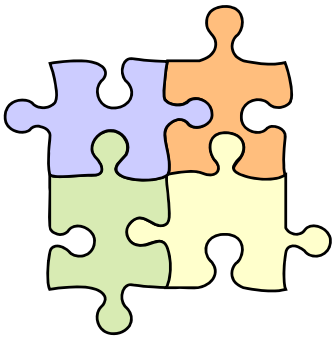
Luca Cabibbo

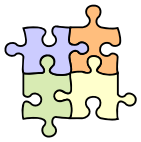


Architetture Software

POSA: Un catalogo di pattern architetturali

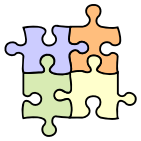
Dispensa PA 1

ottobre 2008



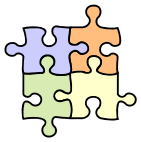
- Fonti

- [POSA] Pattern-Oriented Software Architecture – A System of Patterns
- [POSA4] Pattern-Oriented Software Architecture – A Pattern Language for Distributed Computing



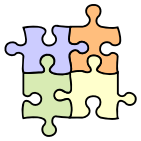
- Obiettivi e argomenti

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



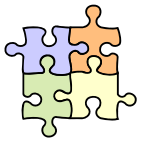
* Introduzione

- Un *pattern* (*software*)
  - la descrizione strutturata di una soluzione esemplare ad un problema (software) ricorrente
- Uno *pattern architetturale* – o *stile architetturale*
  - un pattern per descrivere un'architettura software
  - gli elementi descrivono sotto-sistemi o comunque macro-componenti
- Un *pattern language* (*linguaggio di pattern*)
  - una famiglia di pattern correlati
    - con una discussione sulle loro correlazioni
  - specifici per la progettazione di certi tipi di sistemi o per certi tipi di requisiti
    - ad es., per la sicurezza o per i sistemi distribuiti



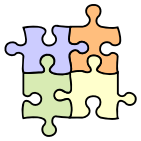
POSA: Un catalogo di pattern architetturali

- [POSA], pubblicato nel 1996, identifica e descrive quattro categorie di pattern architetturali
  - dal fango alla struttura
    - per sostenere una decomposizione controllata del sistema complessivo
  - sistemi distribuiti
    - per fornire un'infrastruttura per applicazioni distribuite
  - sistemi interattivi
    - per strutturare sistemi software che prevedono un'interazione uomo-macchina
  - sistemi adattabili
    - per sostenere l'adattamento del sistema – a fronte dell'evoluzione della tecnologia e/o dei requisiti funzionali



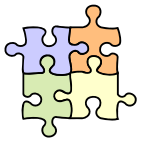
Categorie di pattern architetturali

- Esempi di pattern [POSA]
  - dal fango alla struttura
    - layers, pipes-and-filters, blackboard, ...
  - sistemi distribuiti
    - broker, ...
  - sistemi interattivi
    - model-view-controller, presentation-abstraction-control, ...
  - sistemi adattabili
    - reflection, microkernel, ...



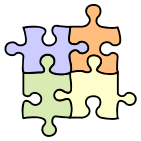
Altri cataloghi di pattern architetturali

- [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]
    - ...
- Chiaramente, esistono anche altri pattern ed altri cataloghi...



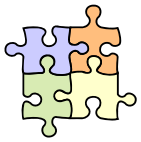
Benefici nell'uso degli stili architetturali

- 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



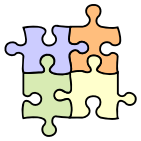
Osservazione sulla notazione

- 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 degli 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 comune che le interazioni tra elementi siano mostrati da uno scambio di messaggi (sincroni oppure asincroni)
- Dunque, una notazione ad oggetti è adeguata
  - ma i rettangoli indicano elementi architetturali, non oggetti



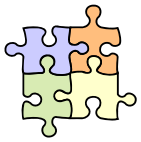
* Domain Model [POSA4]

- 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



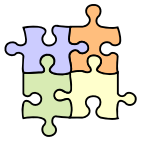
Domain Model

- 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 guidi lo sviluppo
  - senza un' "intuizione" precisa e ragionata del dominio applicativo e della portata dell'applicazione, la sua realizzazione rischia di essere una "grossa palla di fango" ("a big ball of mud") – difficile da comprendere, da comunicare, da valutare e da usare come base per la costruzione del sistema



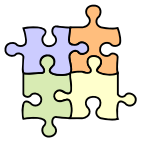
Domain Model

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



Domain Model

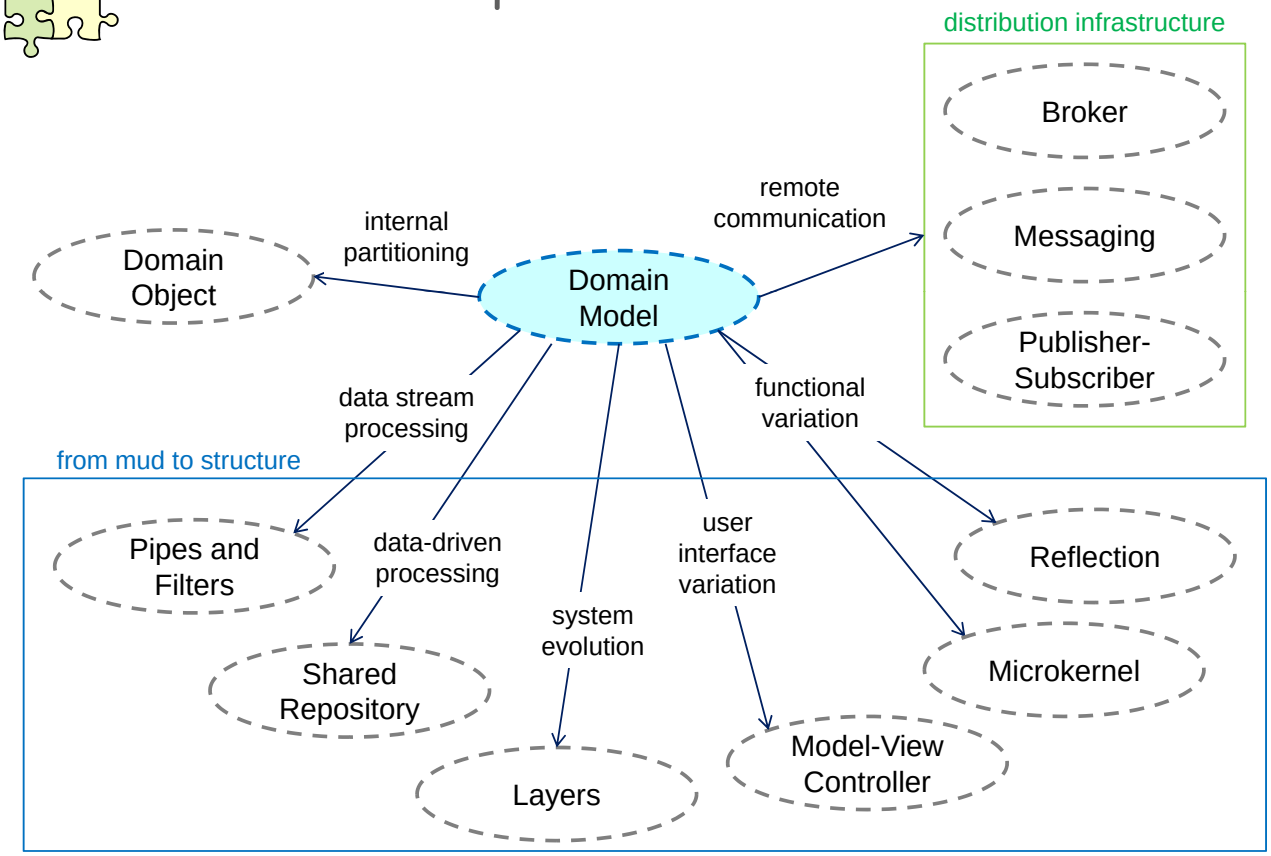
- Discussione
  - il pattern Domain Model accetta molte delle idee proposte da Evans nel libro Domain-Driven Design (DDD)
    - un modello di dominio non è solo un diagramma – è l'idea stessa che il diagramma vuole comunicare
    - il modello di dominio ed il progetto danno forma l'uno all'altro
    - il modello di dominio è la struttura portante del linguaggio usato nella comunicazione tra le parti interessate
    - il modello di dominio è conoscenza distillata



Domain Model

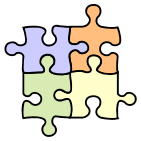
- Discussione
  - il pattern *Domain-Driven Design* dal libro Domain-Driven Design (DDD) di Evans
    - se il progetto, o qualche sua parte importante, non è in corrispondenza con il modello di dominio, allora il valore del modello è scarso, oppure la correttezza del progetto è sospetta – se invece una corrispondenza c'è 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 ed il progetto in modo che entrambi riflettano una profonda comprensione del dominio

Relazioni tra pattern



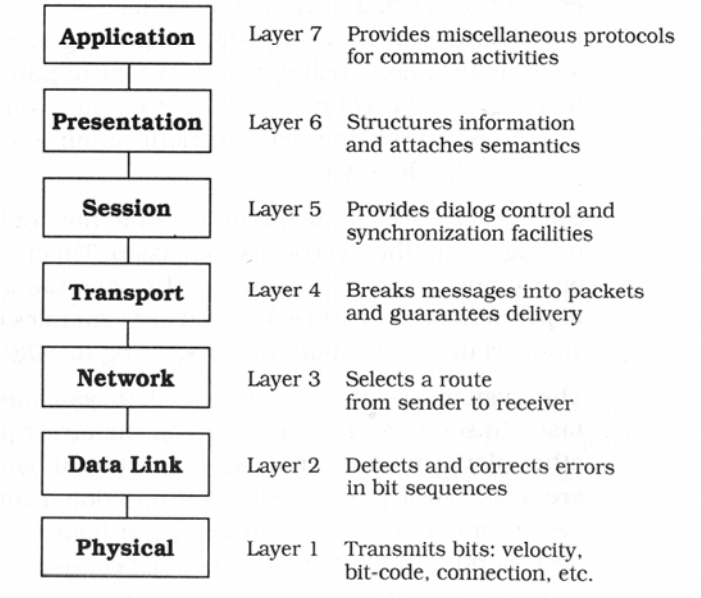
* Layers [POSA]

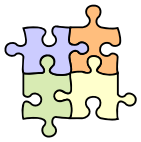
- Il pattern architetturale **Layers**
  - aiuta a strutturare applicazioni che possono essere decomposte in gruppi di compiti
  - in cui ciascun gruppo di compiti è ad un particolare livello di astrazione



Esempio

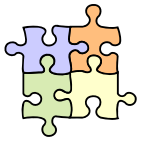
- L'esempio più noto di architettura a strati – protocolli di rete
  - ogni strato si occupa di uno specifico aspetto della comunicazione – di compiti ad uno specifico livello di astrazione





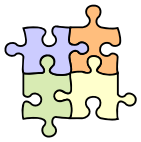
Layers

- Contesto
  - un sistema grande – richiede di essere decomposto
  - è necessario sostenere l'indipendenza di sviluppo ed evoluzione delle diverse parti del sistema
- Problema
  - il sistema si deve occupare della gestione di diversi aspetti – a differenti livelli di astrazione
    - ad es., un sistema per la tariffazione di telefonate
    - da una parte deve gestire dati che vengono da sensori – dall'altra politiche di alto livello come le tariffe telefoniche
  - è richiesta modificabilità e/o portabilità – più in generale, le diverse parti del sistema devo poter essere sviluppate ed evolvere in modo indipendente



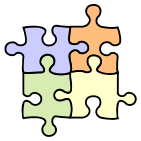
Layers

- Soluzione
  - partiziona il sistema in una gerarchia di *strati* verticali
  - ciascuno strato ha una responsabilità distinta e specifica
    - tutti i componenti nell'ambito di ciascuno strato lavorano ad uno stesso livello di astrazione
  - costruisci le funzionalità di uno strato in modo che dipendano solo dallo stesso strato o da strati inferiori
    - uno strato può comunicare solo con lo/gli strato/i sottostante/i
  - 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



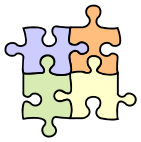
Layers

- Soluzione – ecco le caratteristiche del generico strato J
  - gestisce un certo tipo di responsabilità – quelle al livello di astrazione J
  - fornisce (tramite un'interfaccia) servizi allo strato J+1
  - richiede servizi/delega compiti allo strato J-1 (tramite la sua interfaccia)
  - collabora con lo strato J-1



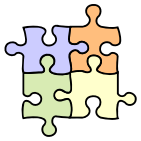
Osservazione

- Si noti che niente viene detto sulla natura degli strati
  - potrebbero essere moduli
  - ma potrebbero anche essere componenti runtime, processi o componenti di deployment
- Ad esempio
  - se gli strati sono moduli
    - la comunicazione tra strati potrebbe essere realizzata come chiamata di procedure (locali)
  - se gli strati sono processi
    - la comunicazione tra strati deve essere realizzata mediante un meccanismo di comunicazione interprocesso – ad es., RPC
  - ...



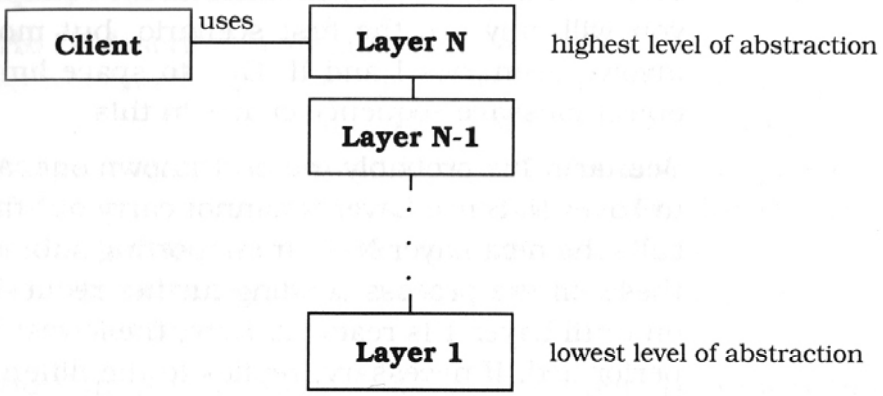
Pattern, soluzione e scenari

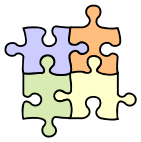
- La *soluzione* di un pattern descrive il *principio* o l'*idea risolutiva* fondamentale del pattern
  - la soluzione comprende almeno una descrizione della *struttura statica* e del *comportamento dinamico* del pattern
- In particolare, la dinamica di un pattern può essere descritta tramite un insieme 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)
  - talvolta gli scenari sono utilizzati per descrivere modi diversi di applicare uno stesso pattern (inteso come idea risolutiva)



Scenario 1 – comunicazione top-down

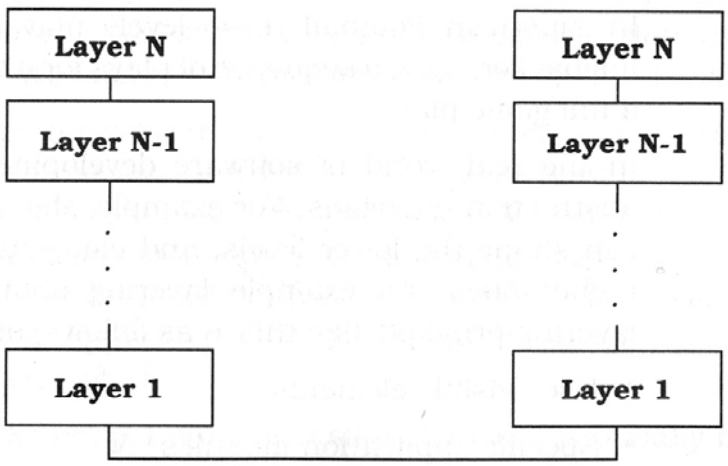
- Lo scenario più comune per Layers – un client effettua una richiesta – al livello N
  - la richiesta viene via via decomposta in sotto-richieste di livello N-1, N-2 – scendendo tra gli strati
  - le eventuali risposte alle sotto-richieste vengono via via combinate – risalendo tra gli strati

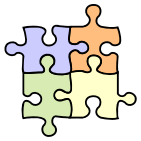




Scenario 2 – comunicazione bottom-up

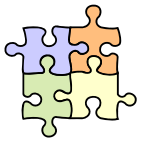
- Il secondo scenario più comune per Layers – le richieste arrivano dal livello più basso
  - i dati risalgono, venendo interpretati opportunamente
  - più corretto parlare di notifiche – anziché di richieste
- Ad esempio, parte nella comunicazione con la pila ISO-OSI





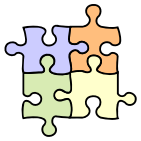
Altri scenari

- 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., una ricezione di un messaggio ripetuto ma già ricevuto, ed il messaggio viene scartato senza arrivare al livello N
  - una richiesta viene gestita da un sottoinsieme degli strati di due pile
    - ad es., se viene trovato un errore in un pacchetto, viene effettuata una richiesta di ritrasmissione nell'ambito di uno strato intermedio



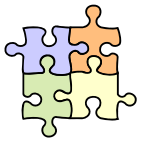
Layers – Applicazione (1/2)

- Definisci il criterio di astrazione
  - ad es., possibili livelli di astrazione in un'applicazione per il gioco degli scacchi
    - pezzi (alfiere), mosse (arrocco), tattiche (difesa siciliana), strategia di gioco complessiva
  - più comunemente
    - presentazione, logica applicativa, servizi tecnici – anche se sono possibili ulteriori livelli intermedi
- Determina gli strati – uno strato per livello di astrazione
  - con i compiti (responsabilità) per ciascuno strato
- Specifica i servizi offerti da ciascuno strato
- Raffina la definizione degli strati – in modo iterativo



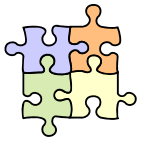
Layers – Applicazione (2/2)

- Definisci l'interfaccia di ciascuno strato
  - quali i servizi offerti da ciascuno strato? quali le notifiche accettate?
- Struttura individualmente gli strati – descritto dopo
- 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 o 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



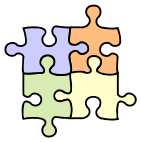
Conseguenze

- Manutenibilità
  - ☺ può essere alta – se le dipendenze sono localizzate
  - 😐 in realtà, dipende da come i cambiamenti si ripercuotono sul sistema – la manutenibilità è alta se i cambiamenti sono localizzati in singoli componenti/strati
  - ☹ tuttavia, se le funzionalità del sistema non sono state organizzate rispetto ad alcuni cambiamenti attesi, altri cambiamenti potranno riguardare molti strati – con un effetto domino



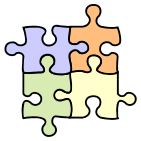
Conseguenze

- 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



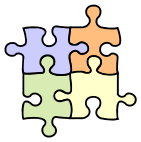
Conseguenze

- 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



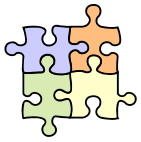
Conseguenze

- Sicurezza
  - ☺ è possibile inserire strati che introducono opportuni meccanismi di sicurezza – ad es., autenticazione, autorizzazioni, crittografia, ...

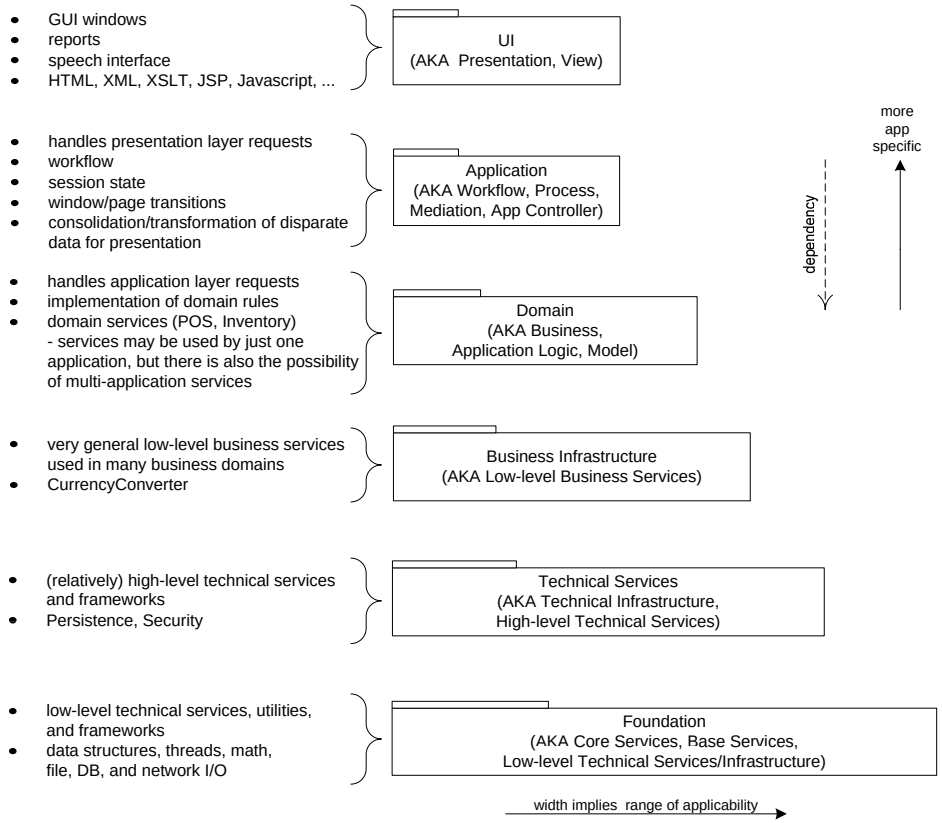


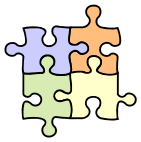
Conseguenze

- Altre conseguenze
  - ☺ possibilità di riusare strati
  - ☺ portabilità, usando strati dedicati alla piattaforma di esecuzione
  - ☹ può essere difficile stabilire la granularità/il numero/il livello di astrazione degli strati



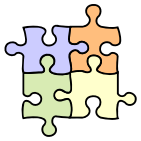
Esempio – una possibile scelta degli strati





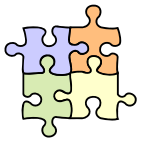
Discussione

- Una scelta comune nel partizionamento di un sistema informatico
  - presentazione, logica applicativa, accesso ai dati persistenti
  - è motivata da
    - decomposizione relativa ad interessi diversi (criterio di astrazione)
    - tasso di cambiamento
      - le interfacce utente tendono a cambiare più rapidamente della logica applicativa
      - la logica applicativa tende a cambiare più rapidamente della struttura dei dati persistenti



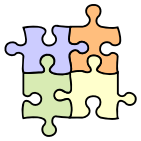
Discussione

- Il pattern Layers è basato su alcuni principi architetturali fondamentali della progettazione del software
  - *separazione degli interessi*
    - mantieni separati gli aspetti relativi a interessi (responsabilità a grana grossa) diversi
  - *modularità* – accoppiamento basso e coesione alta
    - le responsabilità di ciascuno elemento devono essere coese (ovvero, logicamente correlate) tra loro e non mischiate con responsabilità di altri elementi



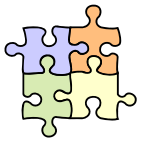
Discussione

- Ancora sulla modularità
  - 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]
  - ovvero, se la modificabilità è la qualità più importante, allora deve essere alla base del criterio di decomposizione utilizzato
    - il principio di modularità suggerisce di incapsulare ciascuna dimensione di cambiamento (atteso) in un elemento diverso
    - in modo che ciascun cambiamento (atteso) possa essere gestito modificando un singolo elemento – ma non gli altri
  - Layers è chiaramente basato su questo principio



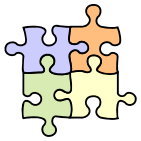
Discussione

- Ancora sul criterio di decomposizione in strati – [POSA4] suggerisce che l'applicazione di Layers sia basata 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
  - il criterio di partizionamento specifico può essere definito secondo varie dimensioni – ad esempio
    - astrazione – presentazione, logica applicativa e dati persistenti
    - 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



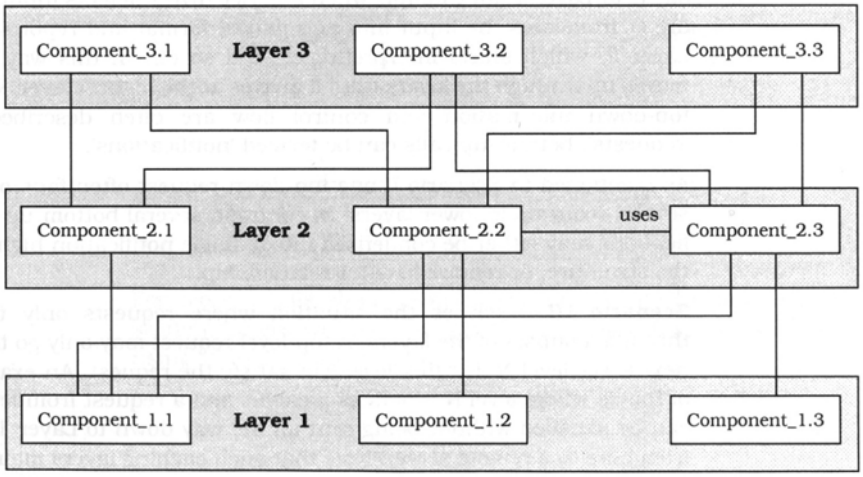
Discussione

- 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

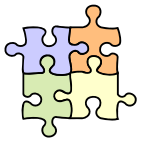


Discussione

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

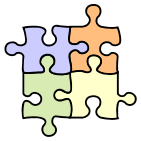


  - quale/i pattern per questa ulteriore decomposizione?



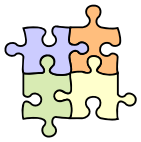
* Domain Object [POSA4]

- Il pattern architetturale **Domain Object**
  - guida la decomposizione di elementi architetturali più grandi
    - ad es., uno strato di un'architettura secondo Layers
  - i criteri base sono, ancora una volta, il Principio di Separazione degli Interessi e la Modularità



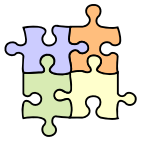
Domain Object

- Contesto
  - decomposizione di un intero sistema, oppure
  - decomposizione di un elemento architetturale
- Problema
  - le parti che formano un sistema software sono spesso caratterizzare da correlazioni e collaborazioni molteplici e variegate
  - realizzare queste funzionalità correlate senza cura può portare ad un progetto con un'elevata complessità strutturale



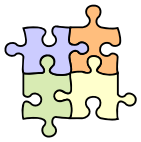
Domain Object

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



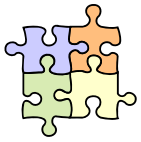
Domain Object

- Discussione
  - il pattern Domain Object è un'ulteriore applicazione del principio di separazione degli interessi e del principio di modularità
  - 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
  - la decomposizione, coerentemente con il pattern Domain Model, può essere guidata da un opportuno modello del dominio
    - questo è la motivazione del nome del pattern
  - la decomposizione, anche se relativa a responsabilità *funzionali*, non può essere indipendente dalle qualità (*non funzionali*) che il sistema deve possedere
    - ad es., prestazioni, affidabilità, ...

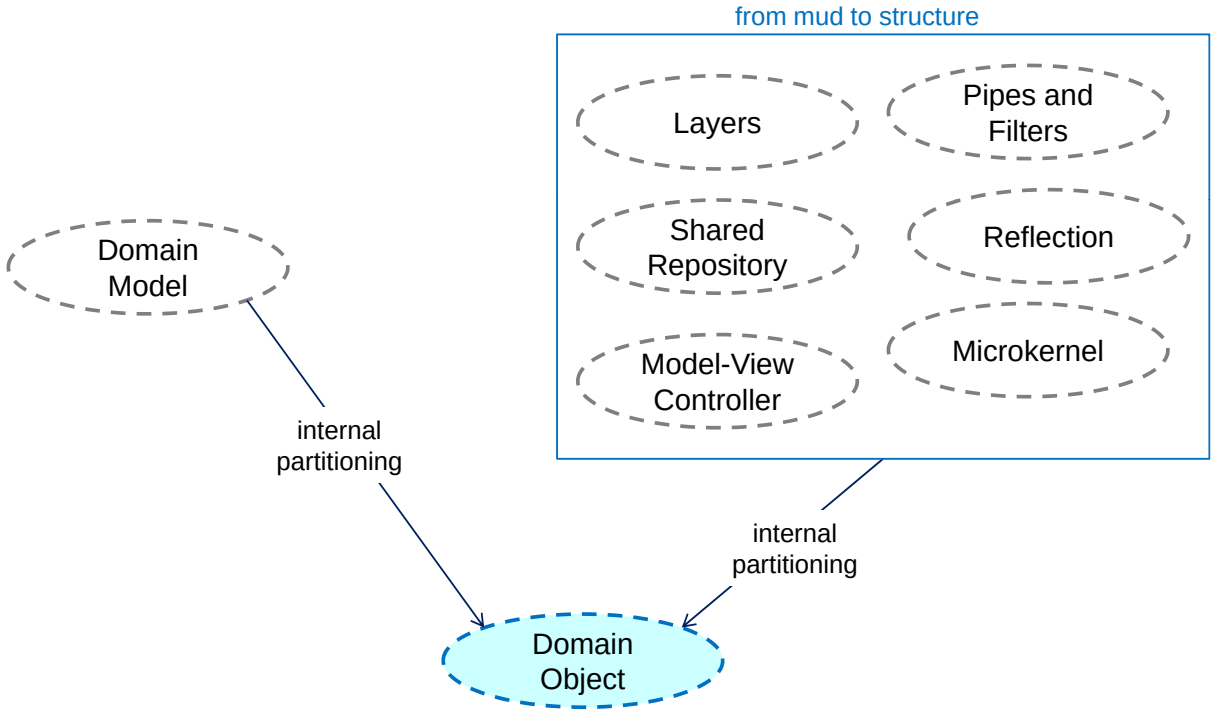


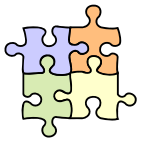
Domain Object

- La decomposizione va guidata da un qualche modello del dominio
  - in alcuni casi, la decomposizione è 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
  - in altri casi, il criterio può essere la decomposizione rappresentazionale delle informazioni del dominio
    - ad es., lo strato della logica applicativa (vedi Larman)
  - oppure, il criterio può essere la 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



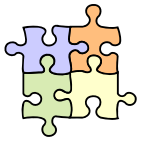
Relazioni tra pattern





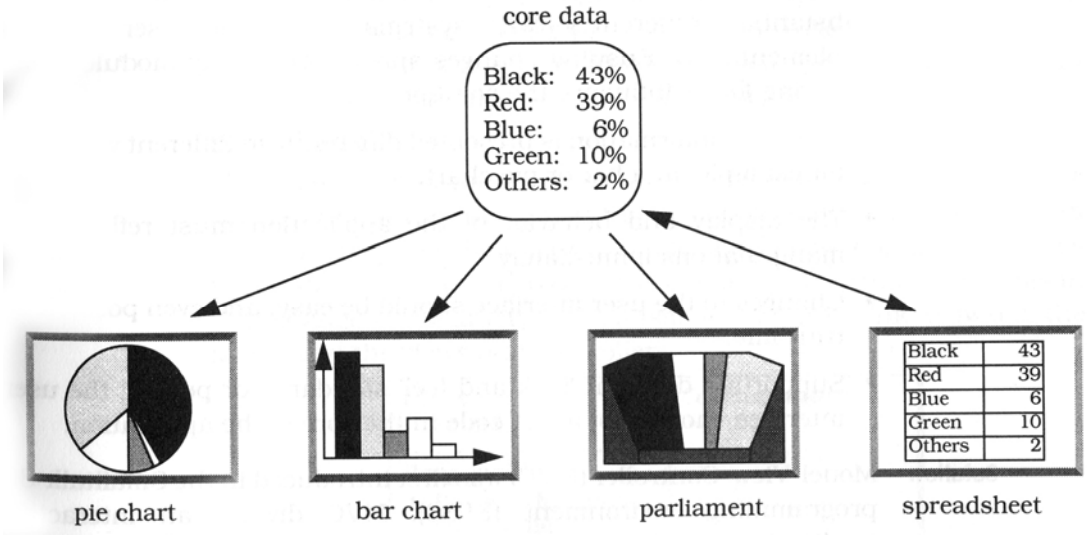
* Model-View-Controller [POSA]

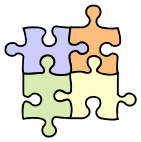
- Il pattern architetturale **Model-View-Controller** (**MVC**) divide un'applicazione interattiva in tre componenti
  - *modello*
    - contiene i dati e le funzionalità di base – si occupa dell'elaborazione dei dati
  - *viste*
    - mostrano informazioni agli utenti – si occupa della gestione dell'output
  - *controller*
    - gestiscono le richieste degli utenti – si occupa della gestione dell'input
  - una vista e un controller, insieme, formano un'interfaccia utente
  - un meccanismo di propagazione dei cambiamenti garantisce la consistenza tra modello e dati visualizzati



Esempio

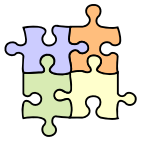
- Sistema informativo con risultati elettorali
  - diverse rappresentazioni grafiche dei risultati





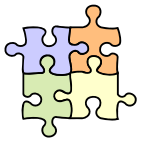
Model-View-Controller

- Contesto
  - applicazione interattiva – con interfaccia utente flessibile
- Problema
  - le interfacce utente sono soggette a richieste di cambiamenti
    - cambiano più rapidamente della logica applicativa e della struttura dei dati persistenti
  - sono richiesti
    - modificabilità dell'interfaccia utente
    - utenti diversi vogliono interfacce utente diverse
    - modi diversi di interagire con il sistema – ad es., mouse vs. tastiera
  - cambiamenti nell'interfaccia utente non devono ripercuotersi sulle funzionalità fondamentali (logica applicativa) dell'applicazione

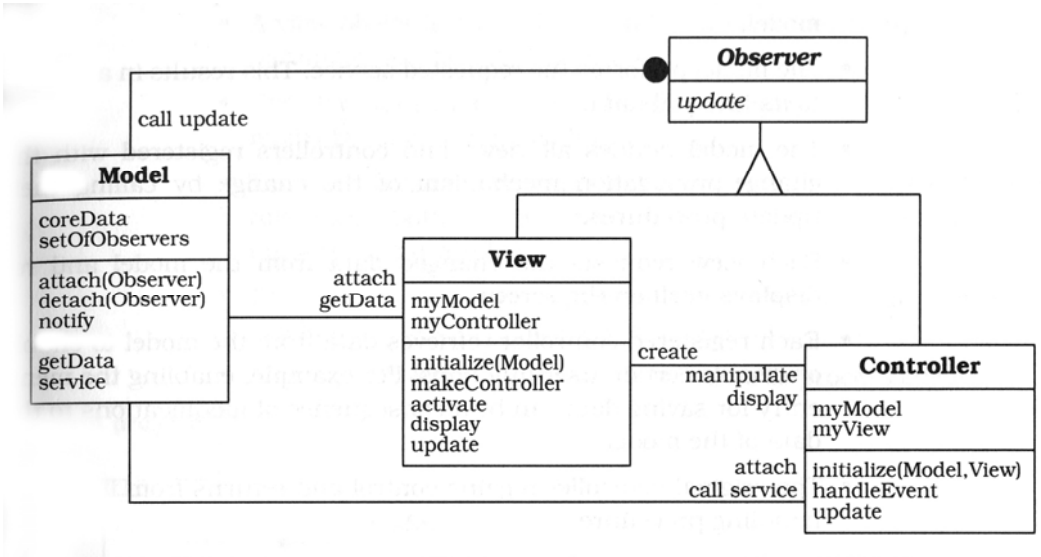


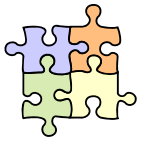
Model-View-Controller

- Soluzione
  - dividi l'applicazione interattiva in tre parti disaccoppiate – input, elaborazione e output
    - il *modello* (elaborazione) contiene i dati e le funzionalità di base
    - le *viste* (output) mostrano informazioni agli utenti
    - i *controller* (input) gestiscono le richieste degli utenti
    - ogni vista ha un suo controller
  - garantisci la consistenza delle tre parti con l'aiuto di un meccanismo di propagazione dei cambiamenti
    - quando un modello cambia il suo stato, notifica tutte le sue viste ed i suoi controller del cambiamento – in modo che questi possano aggiornare il loro stato in modo appropriato



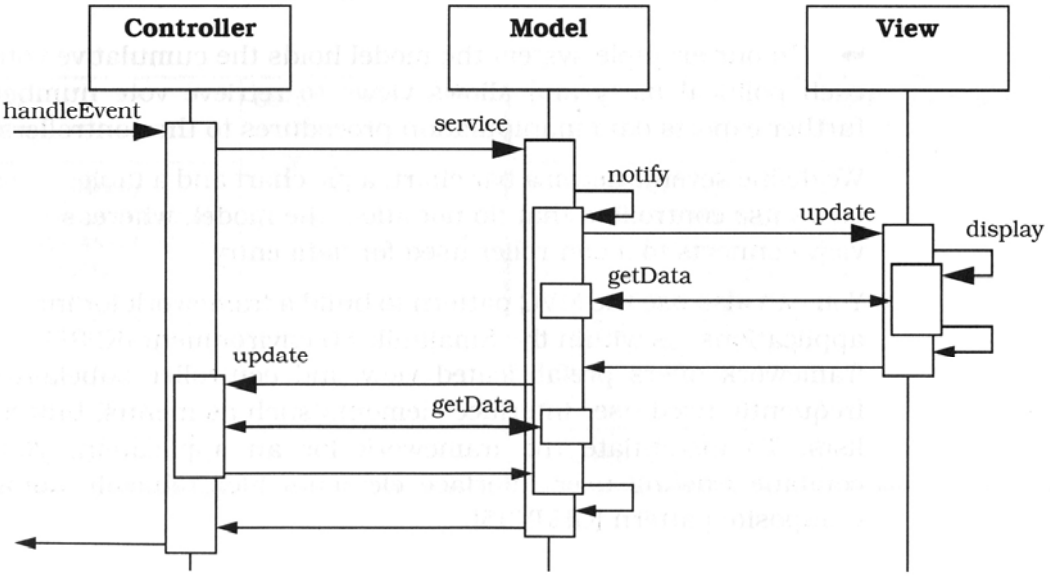
Struttura

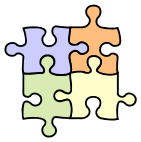




Scenario 1

- Propagazione dell'input

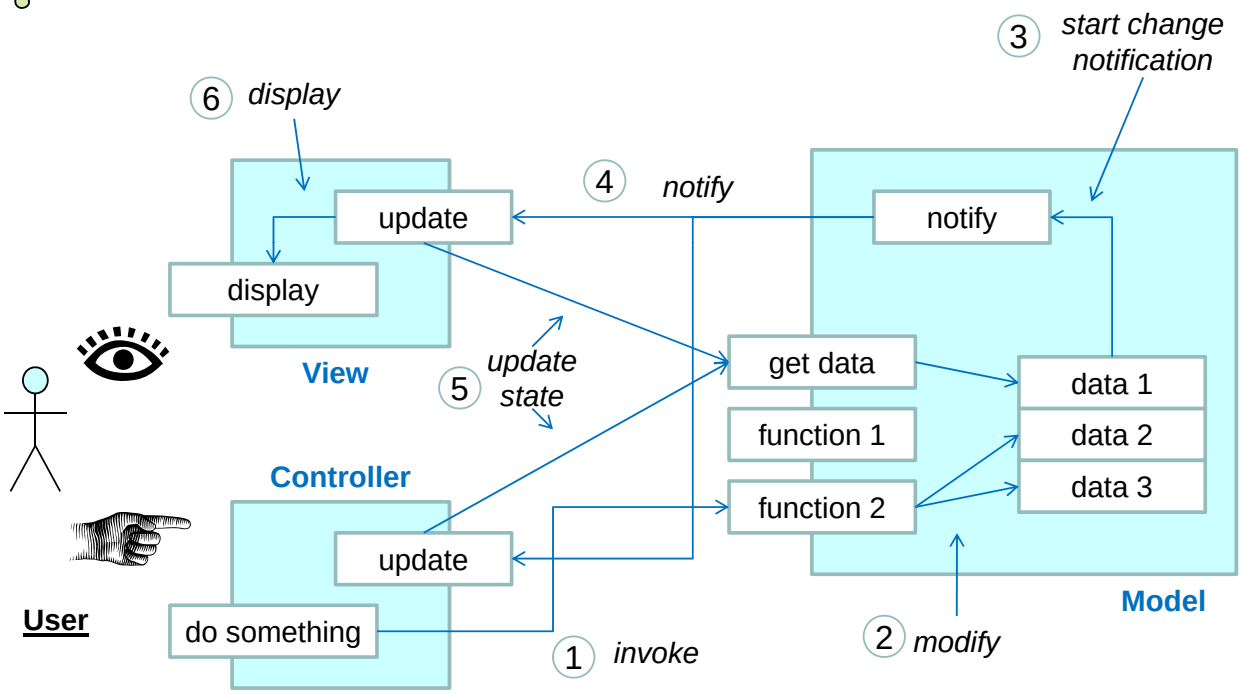




Scenario 1

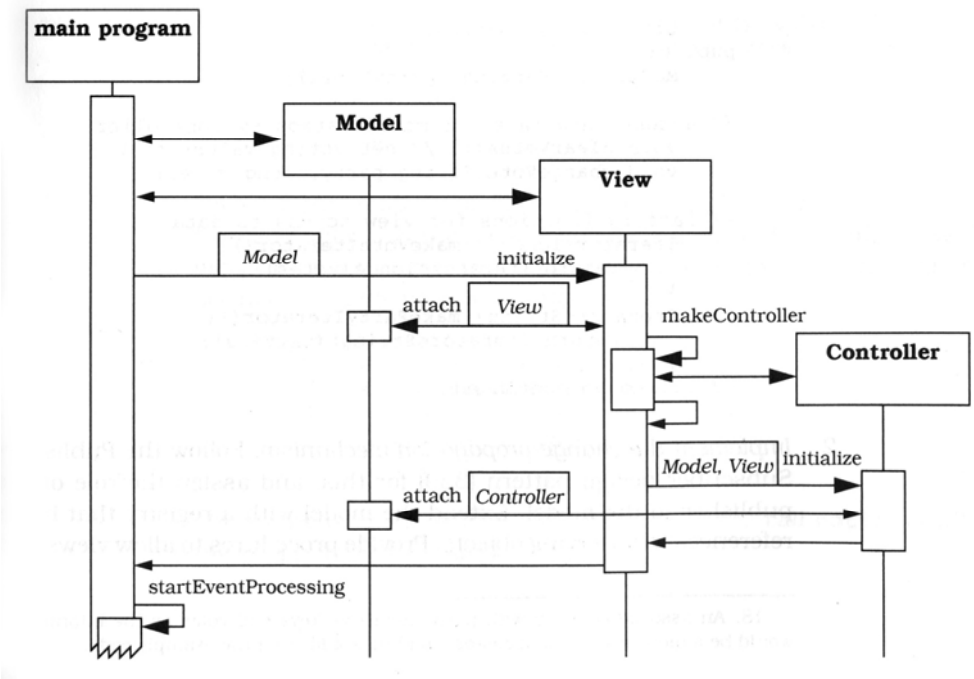
- Propagazione dell'input
  - il controller accetta una richiesta di input tramite la sua procedura di gestione degli eventi – interpreta l'evento e chiede al modello l'esecuzione di un servizio
  - il modello esegue il servizio richiesto – questo può portare ad un cambiamento del suo stato interno
  - il modello notifica tutte le sue viste e i suoi controller dei cambiamenti – mediante il meccanismo di propagazione dei cambiamenti – ad es., mediante Observer
    - ogni vista chiede al modello i dati cambiati di interesse – ed aggiorna la sua visualizzazione
    - ogni controller interroga il modello – ad es., per capire se deve abilitare o disabilitare certe funzionalità
  - in controllo torna al controller originale, considerando conclusa la gestione dell'evento di input

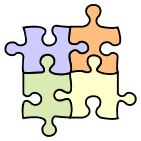
Scenario 1



Scenario 2

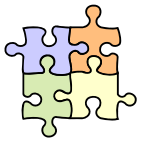
- Inizializzazione della triade MVC





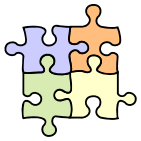
Conseguenze

- Benefici
  - ☺ possibili viste multiple sullo stesso modello
  - ☺ sincronizzazione delle viste
  - ☺ viste e controller plug-and-play
  - ☺ look-and-feel plug-and-play
  - ☺ possibilità di sviluppare/riusare framework – in effetti, MVC è alla base di molti framework per lo sviluppo di applicazioni interattive



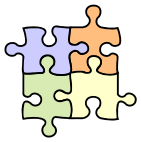
Conseguenze

- Inconvenienti
  - ☹ aumento della complessità
  - ☹ rischio di numero eccessivo di aggiornamenti
  - ☹ accoppiamento tra vista e controller – e di vista e controller con il modello
  - ☹ esistono varianti considerate più flessibili/portabili di MVC – PAC, PCMEF, ...



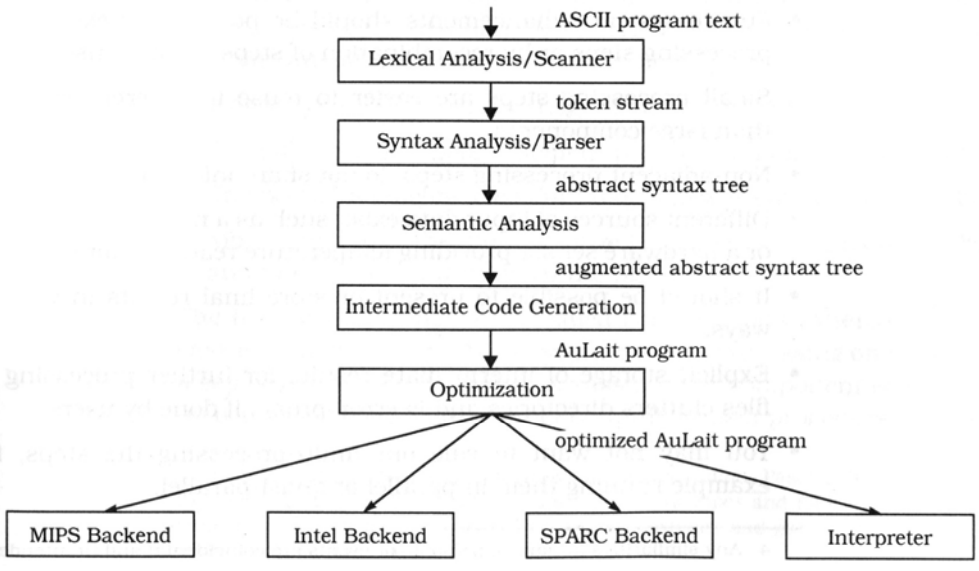
* Pipes and Filters [POSA]

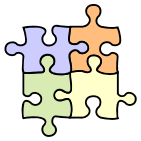
- Il pattern architetturale **Pipes and Filters**
  - fornisce una struttura per sistemi che devono elaborare flussi di dati
  - l'elaborazione complessiva è decomposta in passi di elaborazione
  - ciascun passo di elaborazione è incapsulato in un componente *filtro*
  - i dati sono trasferiti tra filtri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



Esempio

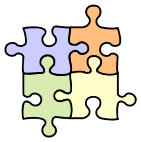
- Si vuole definire un nuovo linguaggio di programmazione (Mocha) – e costruire un compilatore portabile – basato tra l'altro su un linguaggio intermedio (AuLait)





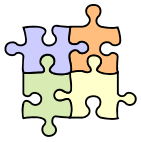
Pipes and Filters

- Contesto
  - un sistema (o componente) deve elaborare flussi di dati
- Problema
  - l'applicazione (o componente) deve elaborare flussi di dati – flussi di ingresso sono trasformati in flussi di uscita
  - una decomposizione in elementi che comunicano mediante meccanismi di tipo richiesta/risposta è considerata inadeguata
  - è possibile ragionare in termini di un modello che descrive il flusso di trasformazione dei dati
    - 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



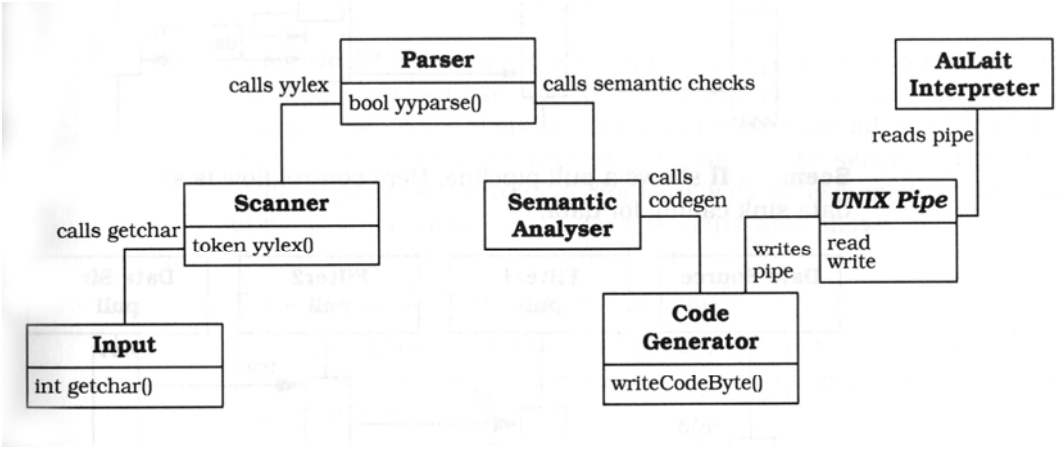
Pipes and Filters

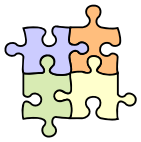
- Soluzione
  - suddividi il compito complessivo di elaborazione in una sequenza di passi di elaborazione di dati successivi ed auto-contenuti
    - ciascun passo è implementato da un *filtro* – che consuma dati di input e genera dati di output in modo incrementale
    - filtri successivi sono collegati da *pipe* – buffer di dati intermedi – che collegano i filtri, mantenendone basso l'accoppiamento
  - la *pipeline* complessiva modella il flusso dei dati nell'applicazione



Esempio

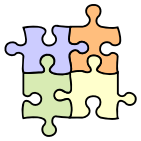
- Il compilatore usa
  - strumenti come lex e yacc – come filtri
  - le pipe di Unix – come pipe





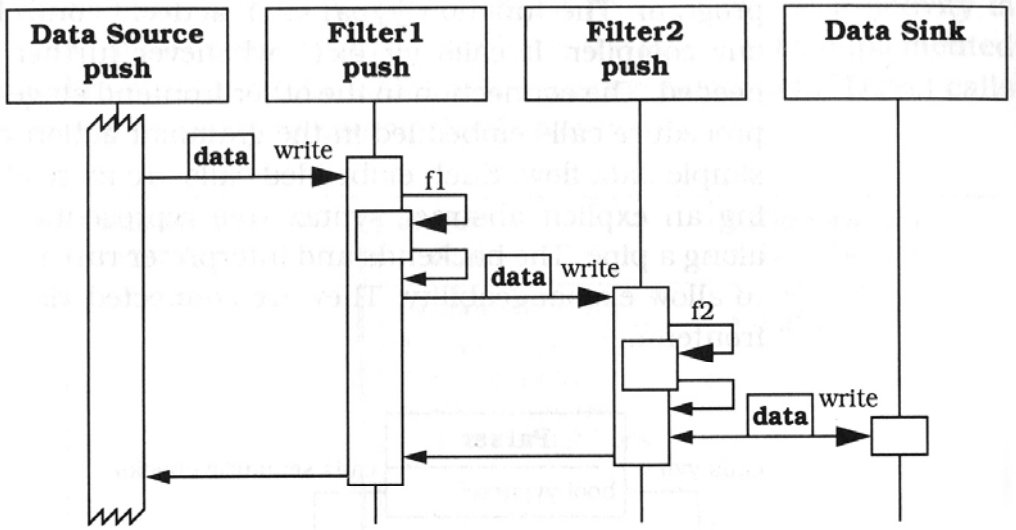
Dinamiche

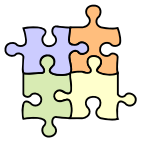
- Sono possibili diverse realizzazioni (scenari) di Pipes and Filters
  - il flusso di dati è sempre unidirezionale, dalla sorgente alla destinazione
    - la gestione del controllo è diversa da scenario a scenario
  - in alcuni scenari, c'è un solo elemento attivo – i filtri potrebbero essere moduli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



Scenario 1

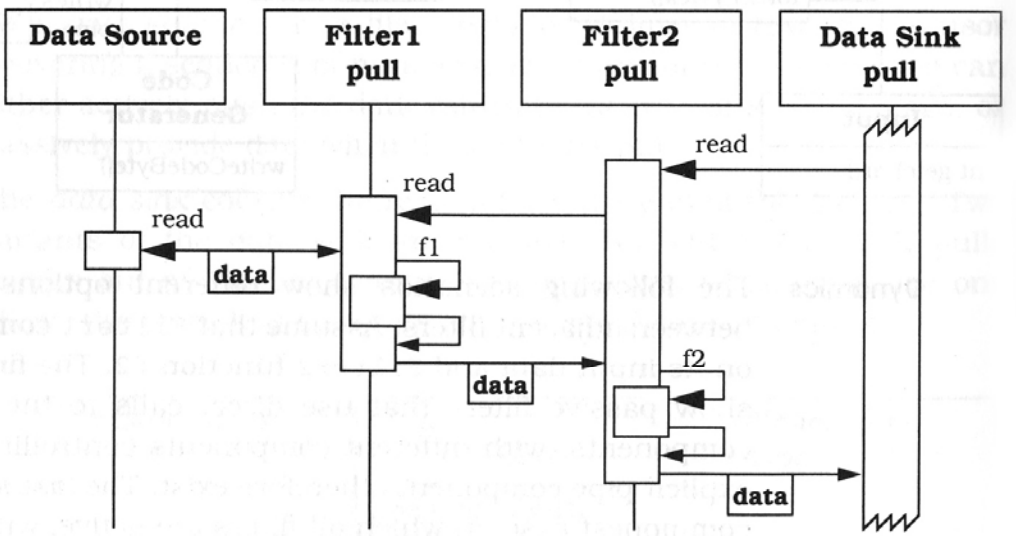
- Pipeline di tipo push
  - l'elaborazione inizia nella sorgente dei dati
  - filtri passivi – potrebbero essere chiamate dirette

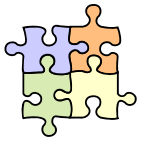




Scenario 2

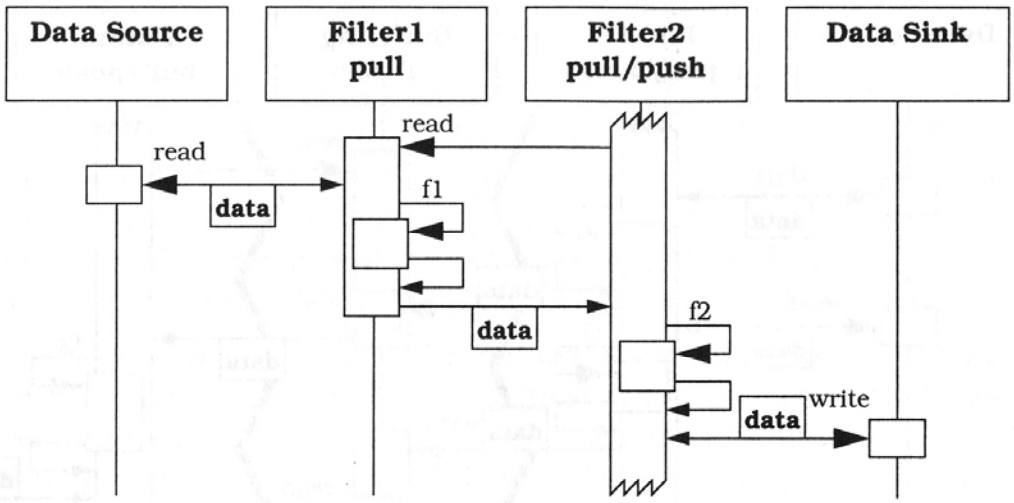
- Pipeline di tipo pull
  - l'elaborazione inizia dalla destinazione dei dati
  - filtri passivi – potrebbero essere chiamate dirette

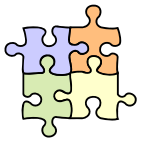




Scenario 3

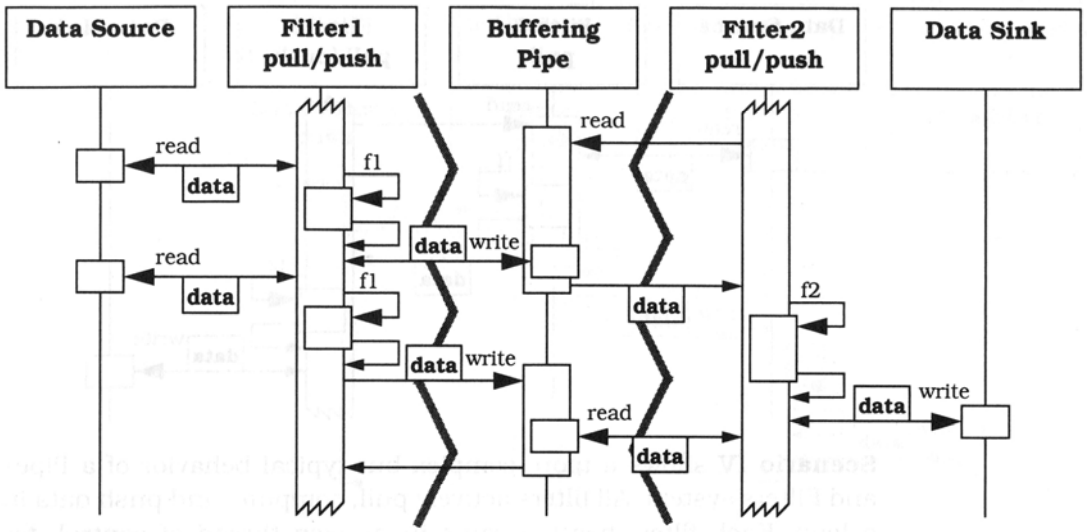
- Uno scenario misto
  - il filtro 2 ha un ruolo attivo
  - filtro 1, sorgente e destinazione sono passivi – potrebbero essere chiamate dirette

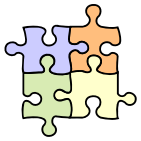




Scenario 4

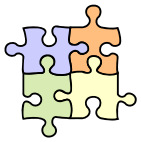
- Uno scenario tipico – e più complesso
  - più filtri attivi – ciascuno con un suo thread/processo
  - la pipe svolge anche il ruolo di buffer – sincronizza i filtri





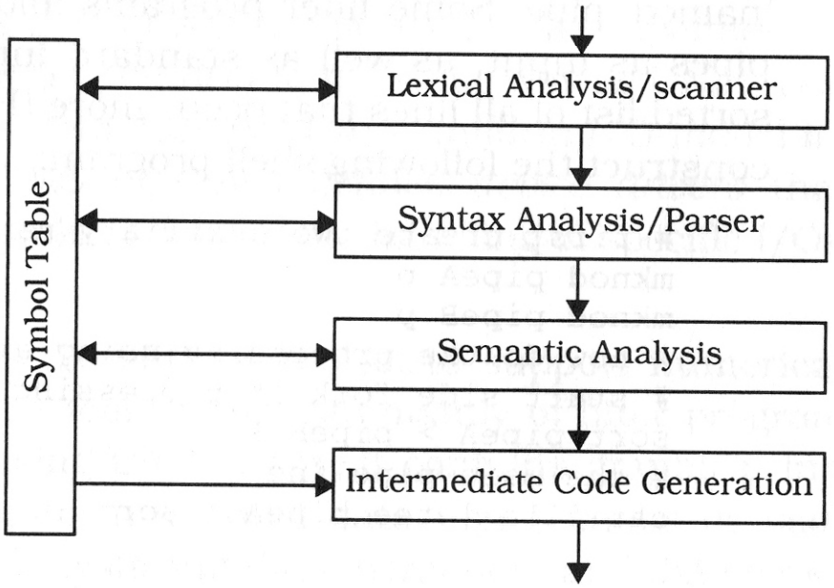
Pipes and Filters – Applicazione

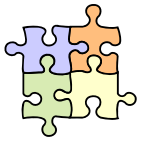
- Suddividi il compito da svolgere in un gruppo di passi di elaborazione
- Definisci il formato dei dati scambiati dai filtri
- Decidi come implementare le connessioni pipe
  - quali filtri sono attivi?
  - i filtri passivi sono attivati in modo push o pull?
  - potrei avere chiamate dirette tra filtri – ma per cambiare la pipeline dovrei cambiare il codice
  - più flessibile la soluzione che usa meccanismi di pipe
  - 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

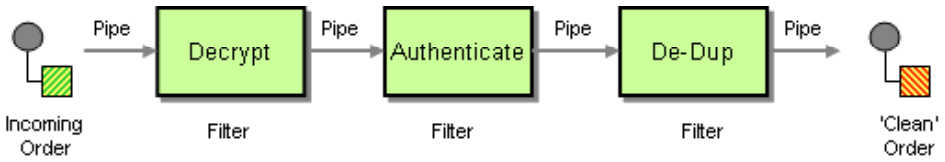
- Nei compilatori, il pattern Pipes and Filters non viene seguito in modo stretto
  - viene utilizzata anche una tabella dei simboli condivisa

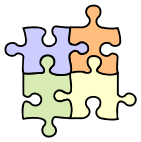




Esempio

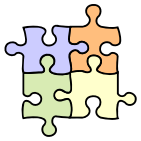
- Si supponga che un sistema riceva ordini sotto forma di messaggi
  - gli ordini sono cifrati – per motivi di sicurezza
  - gli ordini contengono un certificato che garantisce l'identità del cliente
  - è possibile che arrivino messaggi ripetuti – i duplicati vanno eliminati
  - vogliamo trasformare 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





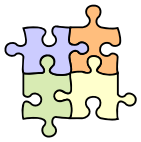
Conseguenze

- Manutenibilità
  - ☺ flessibilità basata sulla sostituzione di filtri
  - ☺ flessibilità basata sulla ricombinazione della rete/pipeline – filtri e pipe possono addirittura essere riorganizzati a runtime
  - ☺ la manutenibilità è buona nella misura in cui i nuovi requisiti possono essere implementati da nuovi filtri o come riorganizzazione della rete
  - ☺ le pipe sono un'indirezione che può ridurre l'accoppiamento tra filtri – può essere positivo il fatto che i filtri non conoscono né gli altri filtri né le pipe
  - ☹ tuttavia, spesso i cambiamenti possono coinvolgere più filtri – per questo, le architetture basate su Pipes and Filters sono talvolta poco mantenibili



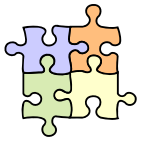
Conseguenze

- Prestazioni
  - ☺ i filtri costituiscono delle unità eccellenti 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 cambiamento di contesto e trasferimento continuo dei dati – perché i filtri normalmente svolgono una piccola quantità di lavoro
  - 😐 il guadagno di efficienza potrebbe essere solo un'illusione



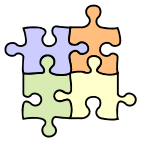
Conseguenze

- Affidabilità
  - 😐 difficile fare considerazioni generali – dipende dalla topologia della rete
  - ☹ 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'infrastruttura di messaging affidabile



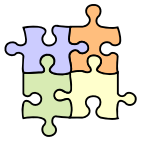
Conseguenze

- Sicurezza
  - ☺ i sistemi Pipes and Filters normalmente sono 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



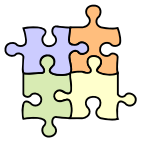
Conseguenze

- Altro
  - ☺ possibilità di riusare componenti filtro
  - ☹ la condivisione di dati tra più elementi è difficile – costosa o poco flessibile
  - ☹ gestione degli errori difficoltosa



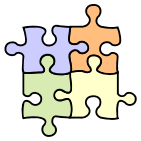
Una variante – Tee and join pipeline system

- *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



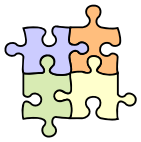
Esempio

- 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 (processi) nell'esecuzione di un'interrogazione, e collegati da pipe
    - sulla base della struttura dell'albero per l'interrogazione



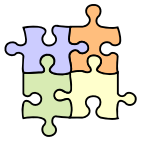
Discussione

- Pipes and Filters suggerisce una decomposizione dell'elaborazione in passi di elaborazione differenti
  - i passi identificati possono evolvere in modo indipendente
  - sostiene un approccio di elaborazione incrementale
- L'organizzazione della pipeline può essere guidata sulla base di una modellazione del dominio in termini di flusso di dati e attività
  - i filtri corrispondono ad unità di elaborazione specifici 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 identificate come Domain Object
  - i pattern Messaging e Publisher-Subscriber definiscono una possibile infrastruttura distribuita per architetture Pipes and Filters



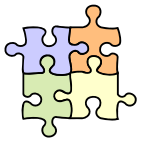
* Shared Repository [POSA4]

- Il pattern architetturale **Shared Repository**
  - guida la connessione tra elementi architetturali o applicazioni che operano su un insieme di dati condivisi
  - il coordinamento tra i diversi elementi o applicazioni avviene tramite questi dati condivisi – e non tramite interazioni dirette tra gli elementi o applicazioni



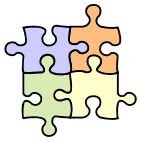
Shared Repository

- Contesto
  - un'applicazione data-intensive
- Problema
  - alcune applicazioni (e i loro componenti) sono inerentemente guidate dai dati
  - le interazioni tra questi componenti non sono guidate da processi specifici
    - oppure, non si vogliono cablare questi processi nel codice, ad esempio perché soggetti a cambiamenti frequenti
  - questi componenti devono comunque interagire in modo controllato – anche in mancanza di un meccanismo funzionale esplicito che governa le loro interazioni ed interconnessioni
  - è possibile coordinare questi componenti con riferimento a dati condivisi su cui operano



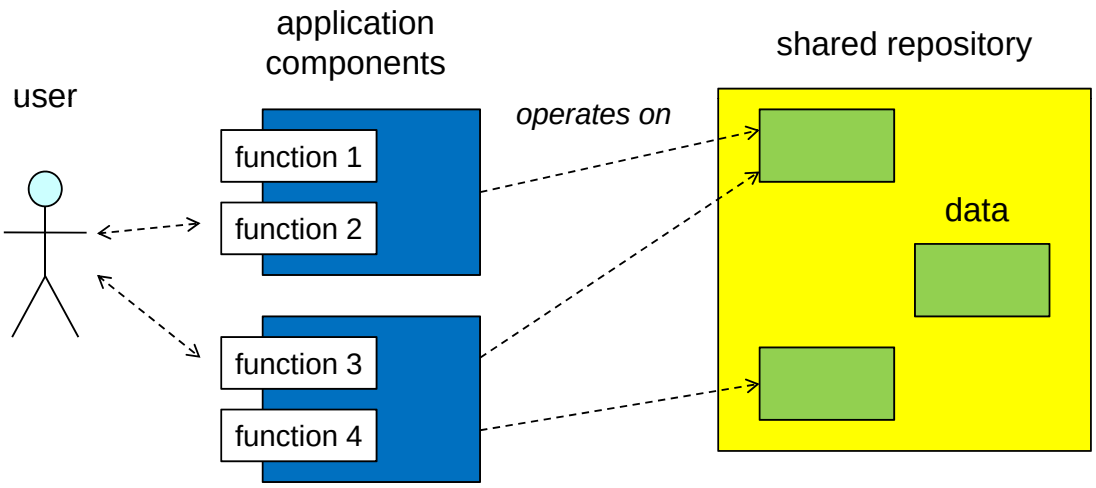
Shared Repository

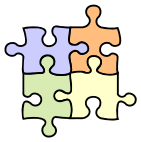
- Soluzione
  - mantieni i dati in un *repository* centrale *condiviso* da tutti i componenti funzionali dell'applicazione
  - fa guidare e coordinare il flusso di controllo della logica applicativa dalla disponibilità, qualità e stato dei dati nel repository
    - i componenti lavorano direttamente con i dati mantenuti nel repository condiviso



Shared Repository

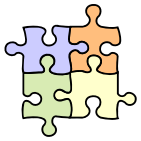
- Struttura della soluzione





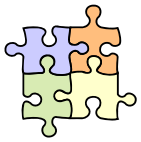
Shared Repository

- Discussione
  - l'architettura Shared Repository consente l'integrazione elementi funzionali (o applicazioni) con un flusso di controllo guidato dai dati
  - il repository condiviso è un punto di accesso a dati condivisi
    - potrebbe essere una base di dati relazionale
    - potrebbe essere una collezione di oggetti in memoria



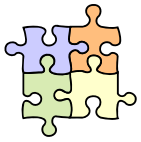
Shared Repository

- Discussione
  - l'integrazione è basata sull'accesso ai dati nel repository condiviso – e non su interazioni esplicite e dirette tra i vari elementi
    - intuitivamente, l'integrazione tra i vari elementi avviene accoppiando ciascun elemento con il repository condiviso – non accoppiando gli elementi direttamente tra di loro
  - è possibile integrare un'ulteriore applicazione?
    - in linea di principio sì, senza peraltro modificare le applicazioni pre-esistenti
    - purché il repository condiviso consenta questa ulteriore integrazione
    - ma il repository condiviso si presta sempre ad ulteriori integrazioni?



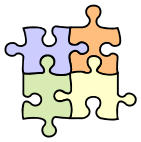
Shared Repository

- Discussione
  - alcuni potenziali cambiamenti nella struttura/schema del repository condiviso potrebbero richiedere cambiamenti anche nelle applicazioni che utilizzano i dati corrispondenti
    - pertanto, alcune organizzazioni sono restie a cambiare lo schema del repository condiviso – se non mediante operazioni di pure "estensioni"
    - come conseguenza, il sistema informatico (nel suo complesso) potrebbe non essere in grado di sostenere cambiamenti dovuti anche a necessità di business effettive ed urgenti



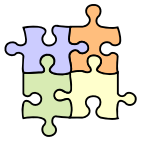
Shared Repository

- Discussione
  - l'accesso ai dati gestiti dal repository condiviso dovrebbe essere opportunamente sincronizzato
    - nei casi più semplici, è sufficiente un'interfaccia formata da operazioni thread-safe
    - nei casi più complessi, sono richieste capacità transazionali
  - talvolta è necessario fornire anche un meccanismo di notifica dei cambiamenti a componenti interessati
  - in generale, nelle applicazioni data-intensive, è un interesse rilevante quello di comprendere "il modo in cui l'architettura memorizza, manipola, gestisce e distribuisce informazioni"



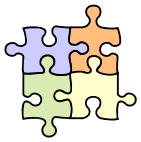
* Database Access Layer [POSA4]

- Il pattern architetturale **Database Access Layer**
  - guida la connessione tra elementi architetturali sviluppati con tecnologia orientata agli oggetti e una base di dati relazionale
  - ci limitiamo a descrivere le idee base di questa soluzione
    - non consideriamo le possibili modalità di realizzazione interna dello strato per l'accesso ai dati



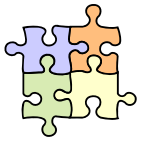
Database Access Layer

- Contesto
  - applicazione orientata agli oggetti che gestisce i suoi dati persistenti in una base di dati relazionale
- Problema
  - i sistemi software sono sempre più implementati con tecnologie orientate agli oggetti – la persistenza di queste applicazioni viene spesso gestita mediante basi di dati relazionali
    - le tecnologie ad oggetti sostengono progettazione ed implementazione
    - le tecnologie relazionali forniscono accesso efficiente, efficace, affidabile e sicuro a basi di dati grandi, persistenti e condivise
  - ci sono problematiche di accoppiamento tra queste tecnologie
    - ad es., modello dei dati, paradigma di accesso, ...



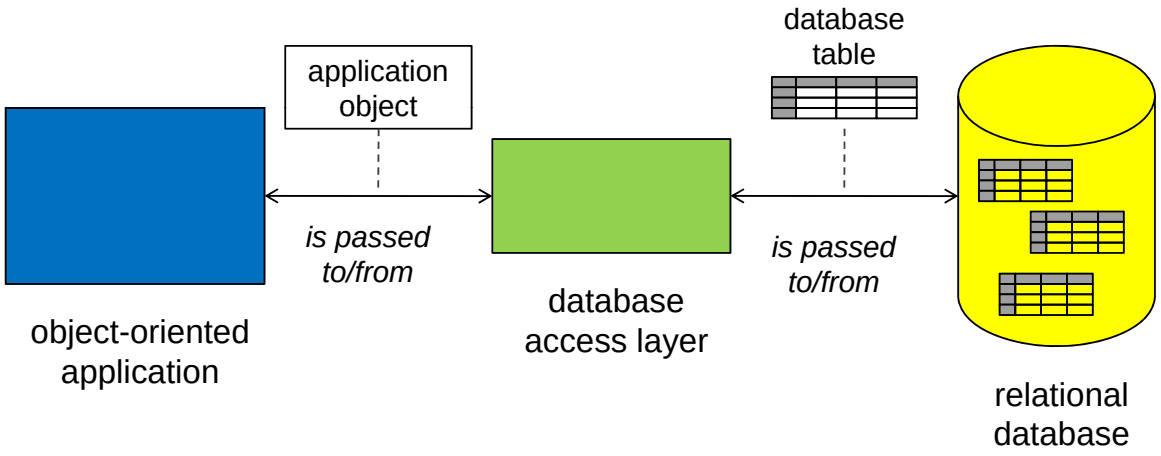
Database Access Layer

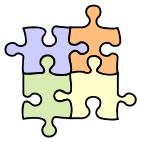
- Soluzione
  - introduci uno *strato* separato *per l'accesso alla base di dati* (*database access layer*) – tra l'applicazione e la base di dati relazionale
    - questo strato fornisce all'applicazione un'interfaccia per l'accesso ai dati stabile ed orientata agli oggetti
      - operazioni CRUD (Create, Read, Update, Delete) per l'accesso agli oggetti delle classi (supposte persistenti) dell'applicazione
    - l'implementazione del Database Access Layer tiene conto degli aspetti relativi alle basi di dati relazionali
      - traduce operazioni CRUD in istruzioni SQL



Database Access Layer

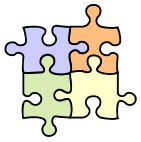
- Struttura della soluzione





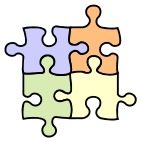
Database Access Layer

- Discussione
  - necessario un linguaggio di alto livello per la descrizione delle corrispondenze tra classi/oggetti dell'applicazione e tabelle/righe della base di dati relazionale
  - il Database Access Layer si può occupare di numerosi aspetti relativi alla gestione di dati persistenti
    - concorrenza, transazioni, caching, accesso a DBMS diversi, ...
  - la struttura interna di un Database Access Layer è guidata anche da motivazioni legate alle tecnologie



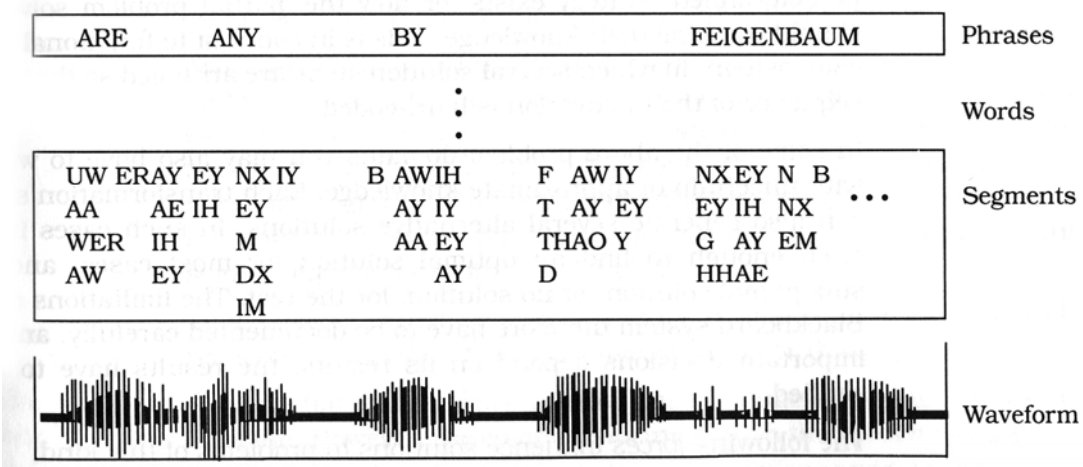
* Blackboard [POSA]

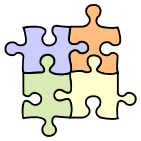
- Il pattern architetturale **Blackboard**
  - utile per problemi per cui non è nota nessuna strategia di soluzione deterministica
  - diversi sotto-sistemi cooperano – integrando la loro conoscenza – per costruire una soluzione di un problema – eventualmente parziale o approssimata



Esempio

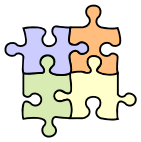
- Riconoscimento della voce
  - dal segnale vocale, ricostruire fonemi, parole, frasi





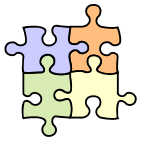
Blackboard

- Contesto
  - problema in un dominio immaturo – in cui non ci sono soluzioni note o fattibili
- Problema
  - il problema non ha una soluzione deterministica nota
  - il problema può essere decomposto in sotto-problemi
  - i sotto-problemi possono essere eterogenei
  - sono note diverse euristiche per ricombinare le sotto-soluzioni

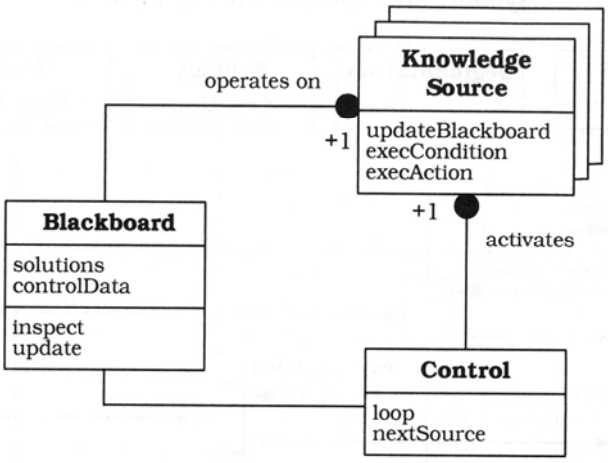


Blackboard

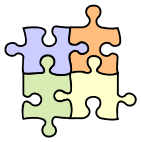
- Soluzione
  - una collezione di componenti indipendenti
  - in grado di cooperare su una struttura di dati comune – *blackboard*
  - il sistema lavora su ipotesi di soluzione intermedie – e le migliora gradualmente
    - un componente di controllo gestisce l'euristica per combinare le soluzioni parziali



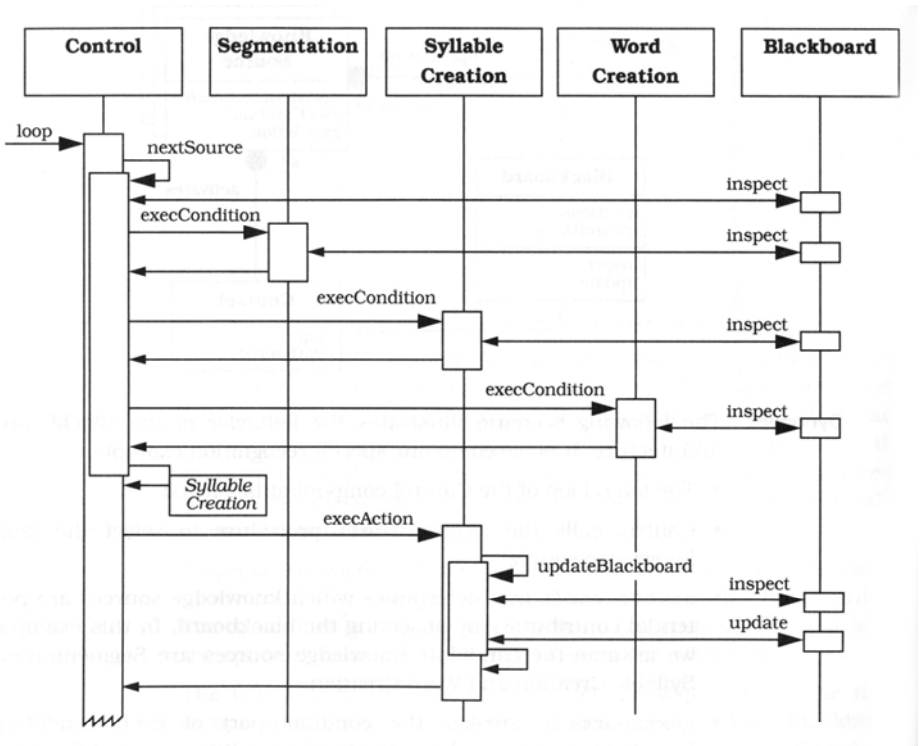
Struttura

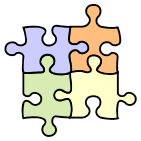


- La *Blackboard* gestisce i dati condivisi
  - è uno Shared Repository
- Le *Knowledge Source* risolvono sotto-problemi parziali
  - consultando e aggiornando la *Blackboard*
- Il *Control* schedula l'attivazione delle *Knowledge Source*



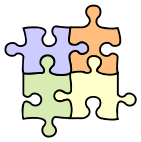
Esempio





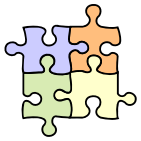
Conseguenze

- Prestazioni
  - 😐 in genere, non è un obiettivo fondamentale di questo stile
  - ☺ tuttavia, è possibile ottenere buone prestazioni in contesti in cui i dati rappresentati nella blackboard sono altamente strutturati – e la generazione e l'uso di questi dati sono separati – nel caso in cui è sufficiente avere dati "abbastanza aggiornati"



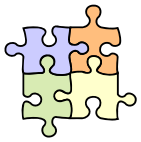
Conseguenze

- Manutenibilità
  - ☺ modello computazionale semplice
  - ☺ incoraggia un accoppiamento debole tra le sorgenti di conoscenza
  - ☺ possibile sperimentare soluzioni diverse
  - ☺ possibile usare meta-dati per descrivere la struttura della blackboard
  - ☺ può consentire una modifica, anche dinamica, delle informazioni gestite dalla blackboard e dei componenti che la aggiornano – se i componenti vedono la struttura logica (e non fisica) della blackboard
  - ☹ tuttavia, in alcuni casi potrebbe essere poco manutenibile – se non è accettabile una modifica del modello dei dati condiviso



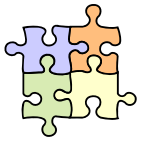
Conseguenze

- Affidabilità
  - ☺ l'indipendenza dei componenti e l'approccio iterativo al calcolo della soluzione potrebbero favorire l'affidabilità
  - ☺ possibilità di centralizzare funzionalità di backup/recovery
  - ☹ tuttavia, la decentralizzazione del controllo può rendere difficile capire se certe responsabilità sono soddisfatte e valutare l'affidabilità



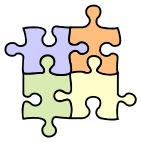
Conseguenze

- Sicurezza
  - ☹ la centralizzazione dei dati può causare problemi di sicurezza – se non vengono prese precauzioni



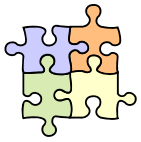
* Microkernel [POSA]

- Il pattern architetturale **Microkernel**
  - sostiene lo sviluppo di un insieme di applicazioni – che sono variazioni l'una dell'altra
  - tutte le diverse applicazioni sono basate sulla stessa architettura ed hanno un unico nucleo funzionale
  - le diverse applicazioni sono costruite in sede di deployment



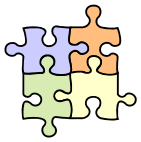
Microkernel

- Contesto
  - applicazione adattabile a diversi scenari di deployment
- Problema
  - alcune applicazioni devono esistere in versioni multiple
    - le diverse applicazioni si differenziano, ad esempio, nelle funzionalità specifiche offerte o nell'interfaccia utente
  - malgrado queste differenze, tutte le versioni dell'applicazioni dovrebbero essere basate su una stessa architettura comune ed uno stesso nucleo funzionale comune
  - alcuni obiettivi di progetto
    - evitare variazioni architetturali – minimizzare lo sforzo di sviluppo ed evoluzione delle funzioni comuni – possibilità di definire nuove versioni o di variare le versioni esistenti

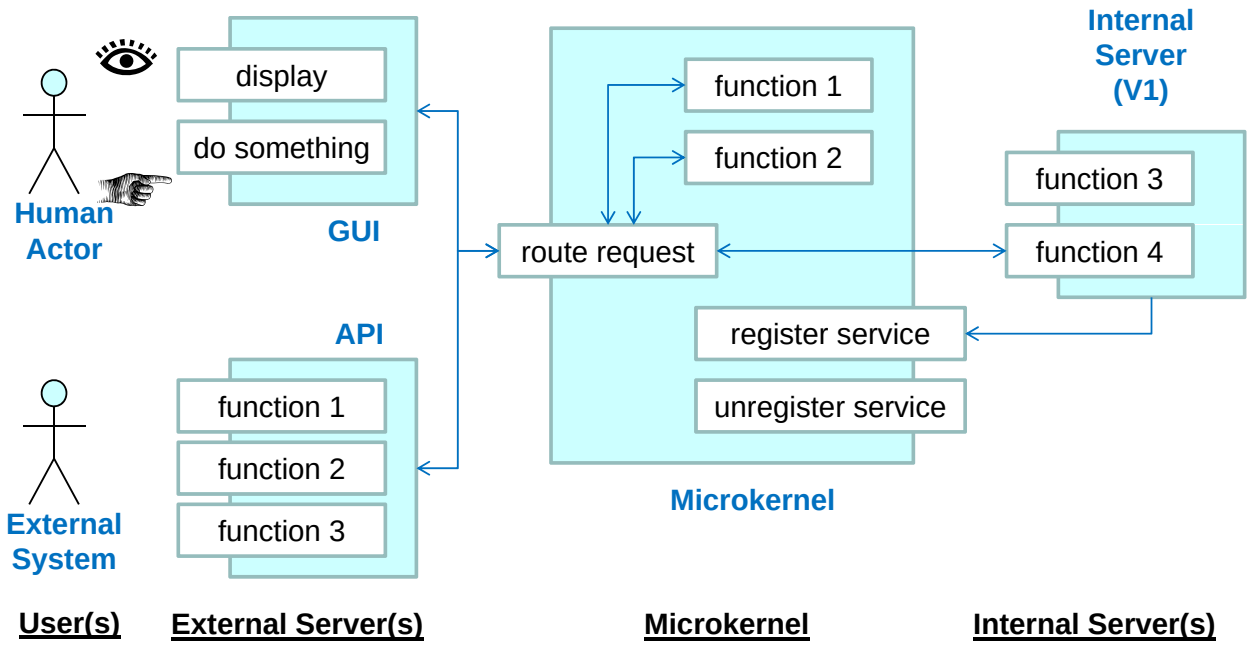


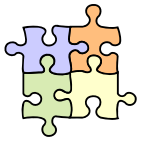
Microkernel

- Soluzione
  - componi le diverse versioni dell'applicazione estendendo un nucleo comune ma minimale, tramite un'infrastruttura plug-and-play
  - il *microkernel* implementa le funzionalità condivise da tutte le versioni, e fornisce l'infrastruttura per integrare le funzionalità specifiche delle diverse versioni
  - un *server interno* implementa delle funzionalità autocontenute, ma specifiche per una versione
  - un *server esterno* implementa un'interfaccia utente o un'API specifica per una versione
  - una versione dell'applicazione è ottenuta connettendo il microkernel con i corrispondenti server interni ed esterni
  - la richiesta di un client, fatta tramite un server esterno, viene propagata, tramite il microkernel, al server interno specifico



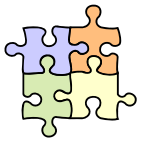
Struttura





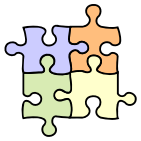
Discussione

- Un'architettura basata su Microkernel
  - sostiene lo sviluppo, l'evoluzione e la gestione di versioni multiple di un'applicazione
  - garantisce che ciascuna versione possa essere effettivamente definita con riferimento al suo scopo specifico
  - la struttura interna è complessa – richiede l'adozione oculata di un insieme opportuno di ulteriori pattern architetturali



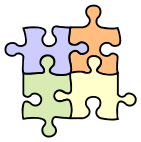
* Reflection [POSA]

- Il pattern architetturale **Reflection**
  - fornisce un meccanismo per cambiare la struttura e il comportamento di un sistema in modo dinamico
  - consente la modifica di aspetti fondamentali – ad es., delle strutture di dati e dei meccanismi di comunicazione



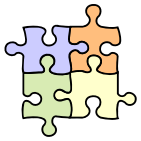
Reflection

- Contesto
  - un sistema che deve consentire delle variazioni – in qualunque momento, anche quando il sistema è attualmente "in operazione"
- Problema
  - il sistema deve evolvere nel tempo – ad es., a causa di cambiamenti nei requisiti o del contesto di utilizzo
  - è difficile prevedere a priori tutte le modifiche
  - le modifiche possono avvenire quando il sistema è in operazione – non è accettabile realizzare le modifiche intervenendo sul codice



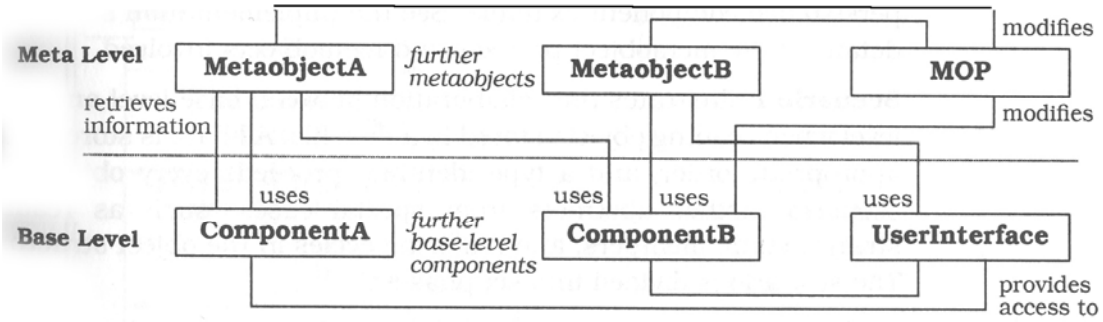
Reflection

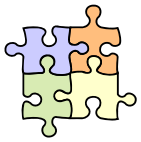
- Soluzione
  - rappresenta esplicitamente le informazioni sulle proprietà e gli aspetti variabili della struttura, del comportamento e delle informazioni di stato dell'applicazione – mediante un insieme di meta-dati o meta-oggetti
  - suddividi il sistema in due parti – mediante un'architettura a due livelli che separa i metadati della logica applicativa fondamentale dell'applicazione
    - un *meta-livello* – contiene i meta-dati
    - un *livello base* – comprende la logica applicativa – la cui implementazione è basata sul meta-livello
  - fornisci un protocollo per amministrare e configurare *dinamicamente* gli oggetti del meta-livello
    - cambiamenti di informazioni nel meta-livello influiscono sul comportamento effettivo al livello base



Struttura

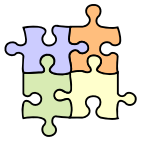
- Struttura
  - due livelli – meta e base
  - protocollo per accedere/cambiare i meta-dati – Meta Object Protocol (MOP)





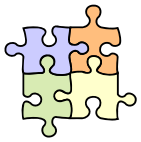
Esempio

- Il catalogo di una base di dati relazionale
  - lo schema di una base di dati relazionale è rappresentato mediante un *catalogo* – un insieme di relazioni con un ruolo specifico
    - il catalogo costituisce il meta-livello – descrive anche le corrispondenze tra livello logico e livello fisico
    - lo schema della base di dati costituisce il livello base
  - le meta-informazioni vengono usate per realizzare il comportamento richiesto
    - ad es., la richiesta “INSERT INTO R VALUES (...)” potrebbe essere realizzata come “memorizza dei record nel file in cui è memorizzato R” e “aggiorna gli indici per R”



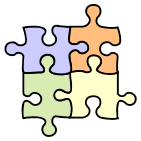
Esempio

- Un Database Access Layer
  - può gestire le corrispondenze tra classi/oggetti dell'applicazione e tabelle/righe della base di dati relazionale mediante un meccanismo di riflessione
  - al meta-livello vengono descritti – mediante meta-oggetti
    - la struttura delle classi
    - lo schema della base di dati
    - le corrispondenze tra classi e relazioni
  - al livello base – le meta-informazioni vengono usate per realizzare il comportamento richiesto
    - ad es., la richiesta "salva un oggetto" viene realizzata come "memorizza righe nelle tabelle che sono in corrispondenza con la classe dell'oggetto"



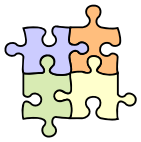
Conseguenze

- Benefici
  - ☺ facile modificare il sistema
  - ☺ è possibile effettuare modifiche al sistema senza cambiare il codice sorgente
- Inconvenienti
  - ☹ aumenta il numero di componenti – maggior complessità
  - ☹ minor efficienza
  - ☹ non tutte le modifiche sono possibili – solo quelle previste dalla meta-modellazione



* Discussione

- I pattern architetturali che sono stati mostrati – ed in particolare quelli “dal fango alla struttura”
  - guidano la decomposizione architetturale di un sistema – o di un componente di un sistema
  - ciascun pattern
    - identifica un particolare tipo di elementi ed una particolare modalità di interazione tra questi elementi
    - descrive criteri per effettuare la decomposizione sulla base di questi elementi
    - 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



Discussione

- Altri pattern architetturali tra quelli mostrati, viceversa, si concentrano su aspetti più specifici
  - ad es., interfacce grafiche e persistenza
  - in alcuni casi, portano ad identificare elementi di natura più tecnica – elementi infrastrutturali per consentire la cooperazione di altri elementi
  - sono comunque descritti ad un livello generale
    - sicuramente in modo indipendente dalle possibili implementazioni e piattaforme
    - normalmente, anche in modo indipendente dalle particolari tecnologie