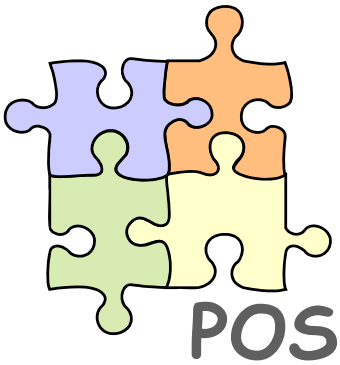
Luca Cabibbo



Architetture Software

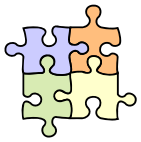
POSA: Un catalogo di pattern architetturali (seconda parte)

Dispensa ASW 361

ottobre 2014

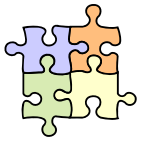
Quando una decisione ha senso
in molte circostanze diverse,
probabilmente è una buona decisione.

J.E. Russo



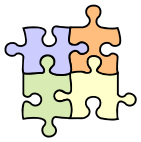
- Fonti

- [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
- [Bachmann, Bass, Nord] Modifiability Tactics, Technical report CMU/SEI-2007-TR-002, 2007

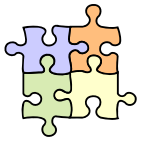


- Obiettivi e argomenti

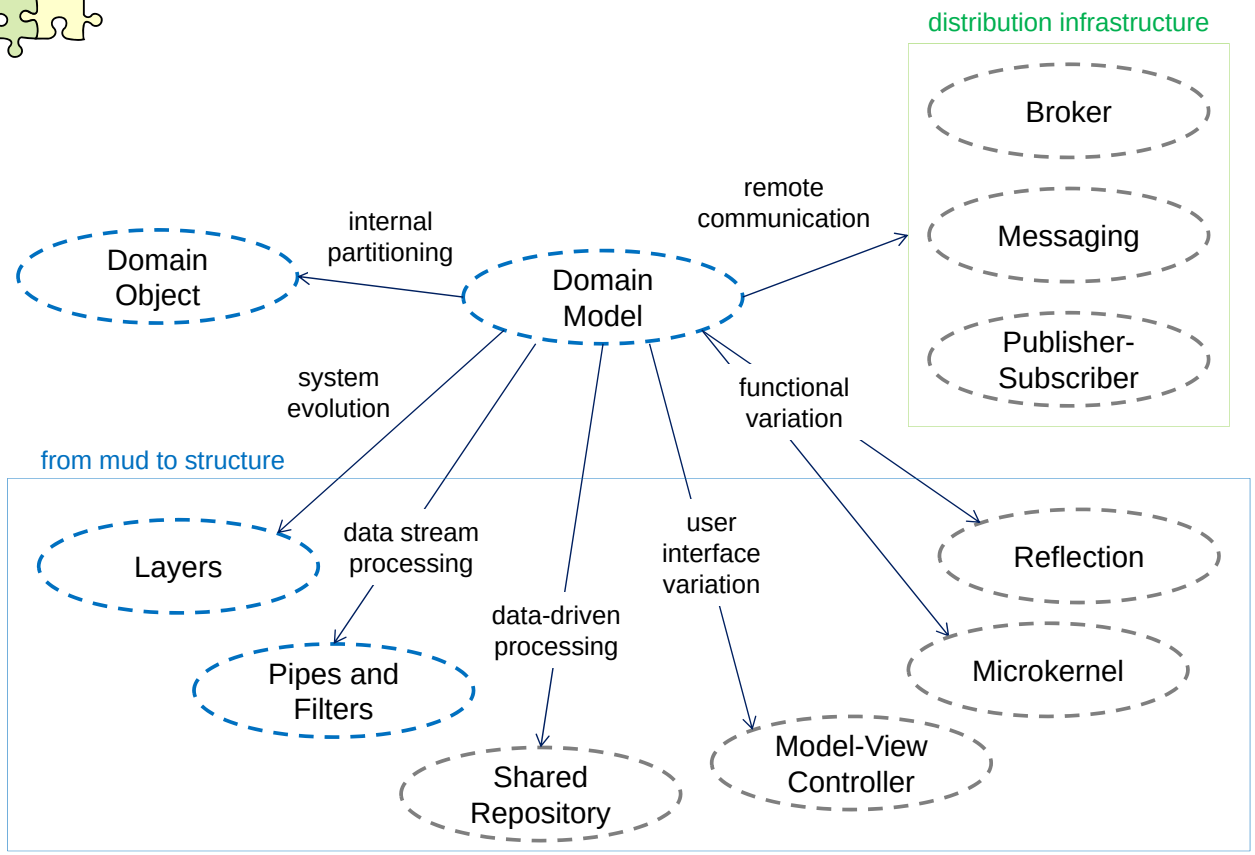
- Obiettivi
  - conoscere alcuni pattern architetturali [POSA] diffusi
  - esemplificare le relazioni tra pattern architetturali e tattiche
- Argomenti
  - introduzione
  - Domain Model [POSA4]
  - Layers [POSA]
  - Domain Object [POSA4]
  - Pipes and Filters [POSA]
  - Model-View-Controller [POSA]
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

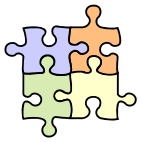


- Wordle



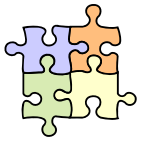
Punto della situazione





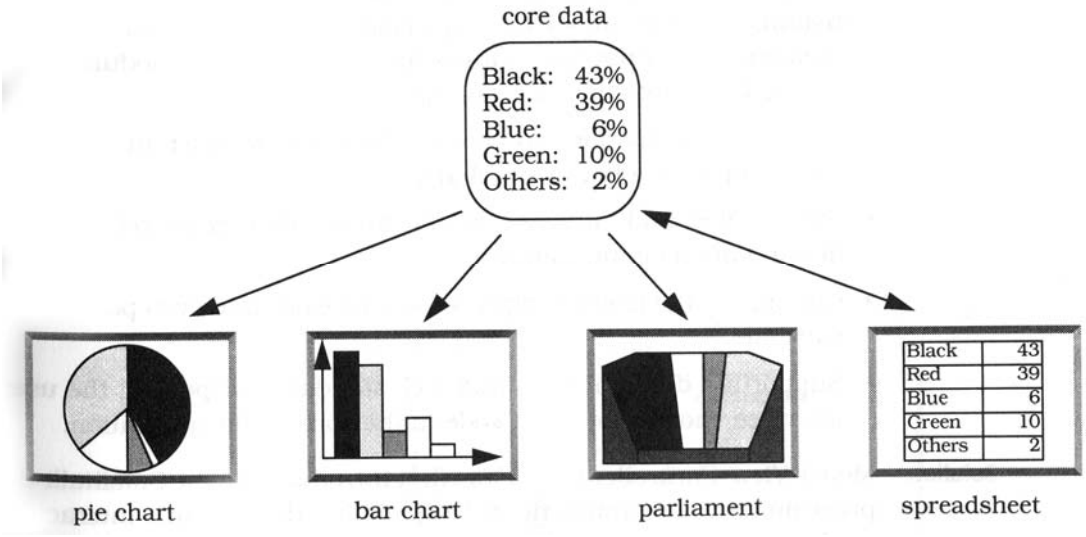
* Model-View-Controller [POSA]

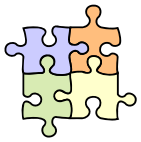
- Il pattern architetturale **Model-View-Controller** (**MVC**)
  - nella categoria [POSA4] “user interface variation”
  - divide un’applicazione interattiva in tre tipologie di componenti
    - *modello* – contiene i dati e le funzionalità di base
      - il modello si occupa dell’elaborazione dei dati
    - *viste* – mostrano informazioni agli utenti
      - una vista si occupa della gestione dell’output
    - *controller* – gestiscono le richieste degli utenti
      - un controller si occupa della gestione dell’input
  - un’*interfaccia utente* è formata da una vista e un controller
  - la coerenza tra modello e dati visualizzati è garantita da un meccanismo di propagazione dei cambiamenti



Esempio

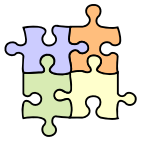
- Sistema informativo con risultati elettorali
  - diverse rappresentazioni grafiche dei risultati





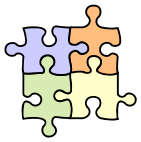
Model-View-Controller

- Contesto
  - applicazione interattiva – con interfaccia utente (UI) flessibile
- Problema
  - le interfacce utente (UI) sono soggette a richieste di cambiamento frequenti
    - cambiano più rapidamente della logica applicativa, della logica di dominio e della struttura dei dati persistenti
  - cambiamenti nell'interfaccia utente non devono ripercuotersi sulle funzionalità fondamentali (logica applicativa e logica di dominio) dell'applicazione
  - inoltre
    - utenti diversi richiedono interfacce utente differenti
    - interfacce utente diverse possono essere basate su modalità di interazione differenti – ad es., mouse vs. tastiera vs. touch

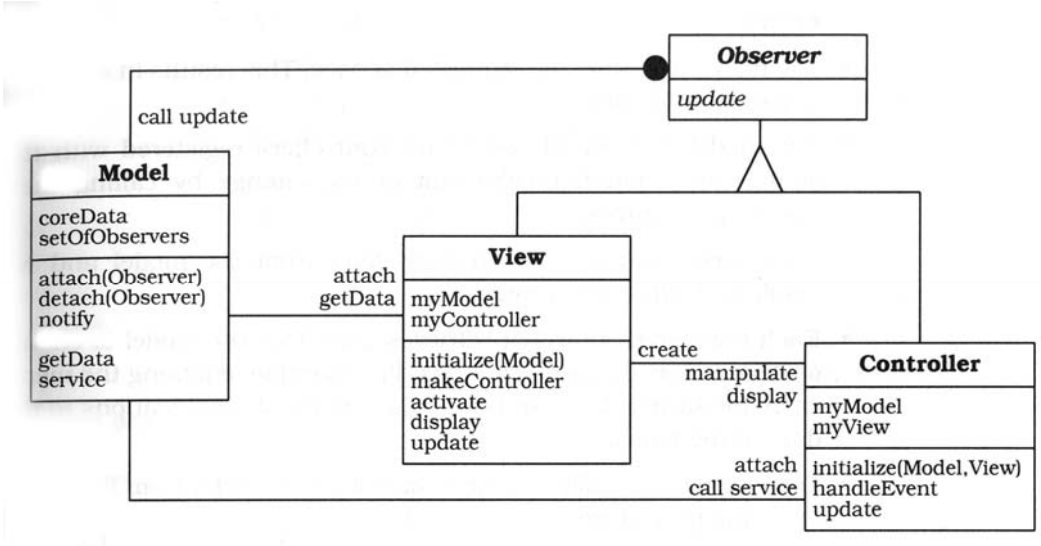


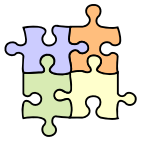
Model-View-Controller

- 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 inoltre la coerenza delle tre parti con l'aiuto di un meccanismo di propagazione dei cambiamenti
    - quando il modello cambia il suo stato, notifica tutte le viste e i relativi controller del cambiamento – in modo che questi possano aggiornare il loro stato in modo appropriato



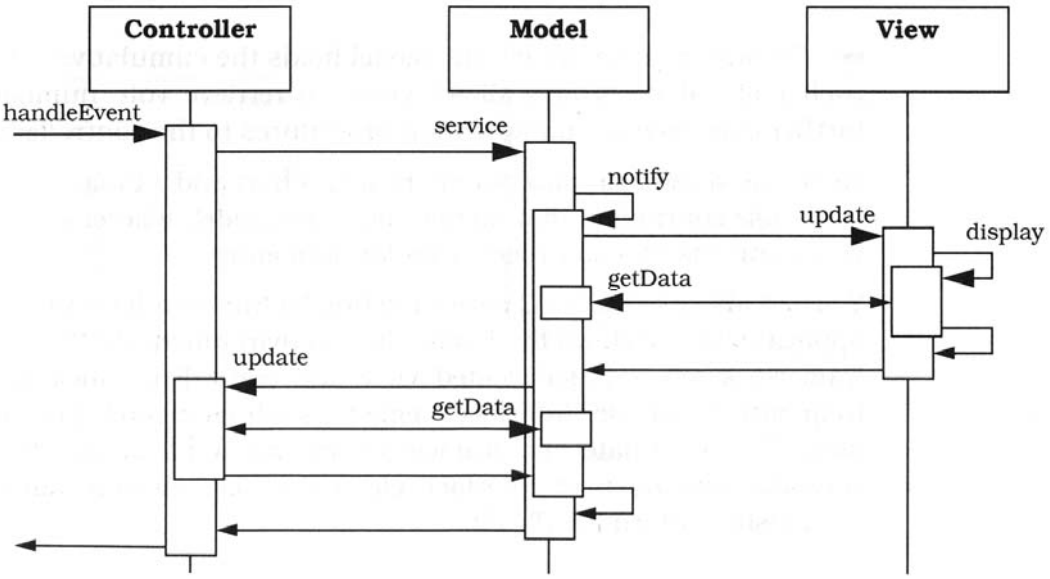
Struttura

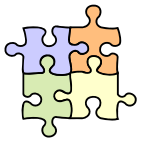




Scenario 1

- Scenario relativo alla propagazione dell'input

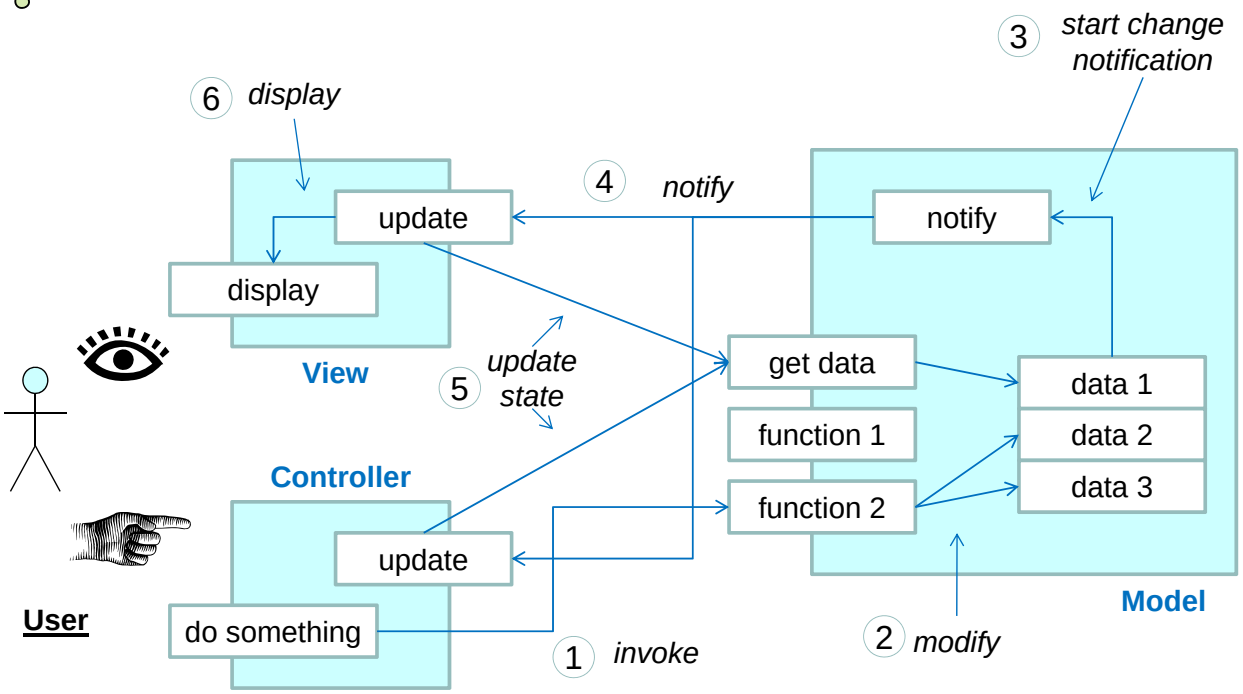




Scenario 1

- Scenario relativo alla propagazione dell'input
  - uno dei 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 a un cambiamento del suo stato interno
  - il modello notifica tutte le viste e i relativi controller dei cambiamenti di stato significativi – tramite un meccanismo di propagazione dei cambiamenti – ad es., mediante Observer
    - ogni vista chiede al modello i dati di interesse che sono cambiati – e aggiorna la sua visualizzazione
    - anche i controller interrogano il modello – ad es., per capire se devono abilitare o disabilitare certe funzionalità
  - in controllo torna al controller originale, considerando conclusa la gestione dell'evento di input

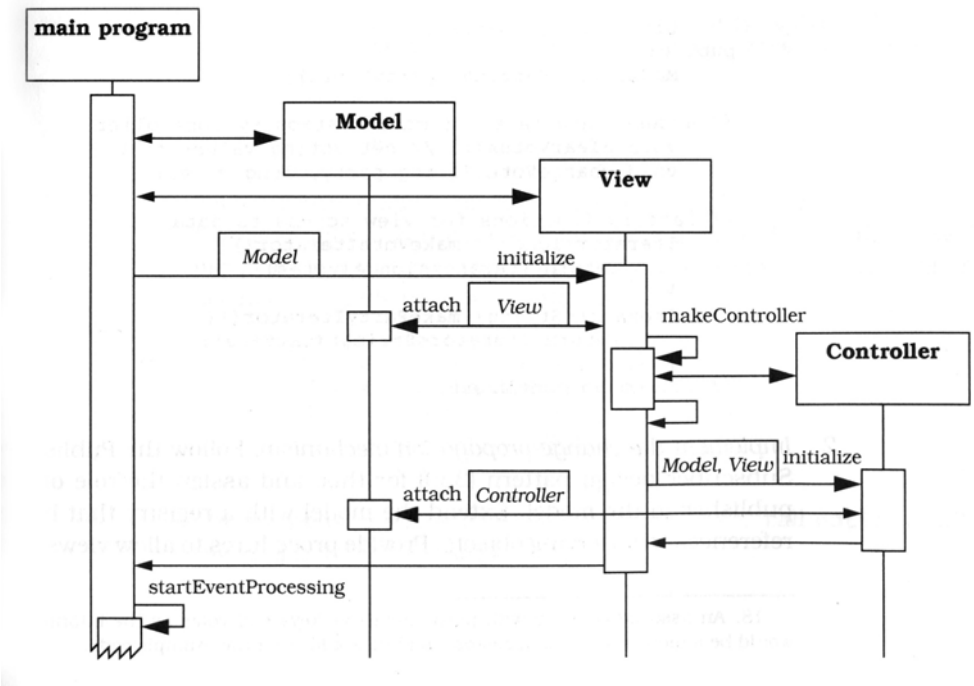
Scenario 1

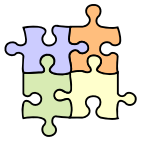


User Interface **Application Functionality**

Scenario 2

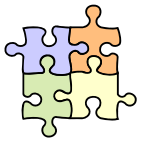
- Scenario relativo all'inizializzazione della triade MVC





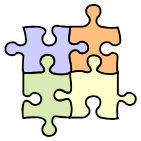
Scenario 2

- Scenario relativo all'inizializzazione della triade MVC
  - il programma principale crea il modello – ce n'è uno solo
  - poi, il programma principale crea le interfacce utente dell'applicazione – possono essere più di una – ciascuna è composta da una vista e da un controller
  - per ciascuna interfaccia utente
    - il programma principale crea la vista – gli passa come parametro il (riferimento al) modello
    - la vista si registra agli eventi generati dal modello
    - la vista crea il controller – gli passa come parametri modello e vista
    - anche il controller si registra agli eventi generati dal modello
  - infine, viene abilitata l'elaborazione degli eventi



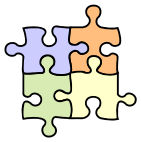
Model-View-Controller

- Alcuni linee guida
  - incapsula le funzionalità fondamentali dell'applicazione nel modello
    - con un'implementazione indipendente da ogni specifica modalità di visualizzazione o meccanismo di interazione – quindi da ogni formato o API di I/O
  - il modello, di solito, può essere partizionato in un insieme di Domain Object
  - associa ogni "pezzo coerente di informazioni" del modello a una o più viste auto-contenute
  - associa ciascuna vista con un insieme separato di controller
  - consenti all'utente di interagire con l'applicazione solo tramite i controller
  - collega modello, vista e controller con un meccanismo di propagazione dei cambiamenti – ad es., basato su Observer



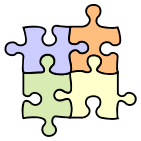
Conseguenze

- Benefici
  - ☺ possibili viste multiple sullo stesso modello
  - ☺ possibili modalità di interazioni multiple su una vista
  - ☺ 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



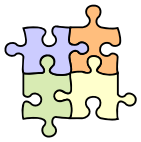
Conseguenze

- Possibili inconvenienti
  - ☹ aumento della complessità
  - ☹ rischio di numero eccessivo di aggiornamenti
  - ☹ accoppiamento tra vista e controller – e di vista e controller con il modello
  - ☹ inefficienza nell'accesso ai dati da parte delle viste
  - ☹ dipendenza dalla piattaforma – il porting di un sistema MVC su una piattaforma diversa può essere difficoltoso
  - ☹ esistono varianti di MVC considerate più flessibili/portabili o più adeguate per gli strumenti attuali per lo sviluppo di interfacce utente – ad es., PAC, MVP, ...



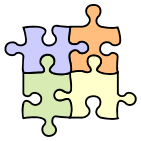
- Usi conosciuti

- Alcuni usi conosciuti del pattern MVC
  - MVC è alla base di molti framework per la definizione di interfacce utente
    - ad es., quello di Smalltalk
    - anche le applicazioni web in Java possono essere strutturate secondo MVC
  - più spesso, sono usati degli stili derivati da MVC
    - ad es., Model-View-Presenter in .NET



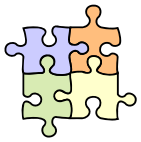
MVC e tattiche per la modificabilità

- *Encapsulate – reduce coupling*
  - il modello incapsula le funzionalità fondamentali dell'applicazione
- *Increase semantic coherence – increase cohesion*
  - le funzionalità fondamentali sono collocate nel modello
  - anche a ciascuna vista e a ciascun controller sono associate responsabilità semanticamente coerenti
- *Use an intermediary – reduce coupling*
  - un controller è un intermediario tra dispositivi di input e modello – una vista è un intermediario tra modello e dispositivi di output
- *Use runtime binding – defer binding*
  - durante l'esecuzione, è possibile aprire e chiudere dinamicamente le viste, così come legare viste differenti ai dati



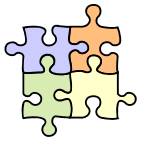
* Shared Repository [POSA4]

- Il pattern architetturale **Shared Repository**
  - nella categoria [POSA4] “data-driven processing”
  - 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 – anziché tramite interazioni dirette tra gli elementi o applicazioni



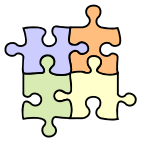
Esempio

- Si vogliono definire un gruppo di applicazioni relative ad attività diverse per la gestione degli ordini
  - ad es., ricezione ordini, evasione ordini, fatturazione, ricezione pagamenti, ricezione merci da fornitori, ...
- Queste applicazioni devono agire in modo coordinato
  - ad es., la ricezione di alcune merci potrebbe consentire l’evasione di alcuni ordini sospesi
- Il coordinamento diretto (point-to-point) tra applicazioni non è desiderato
  - può essere difficile esplicitare le modalità di coordinamento tra applicazioni
  - fino a N*N “coordinamenti diretti” tra N applicazioni
  - aggiungere un’applicazione può richiedere di modificare tutte le N applicazioni esistenti



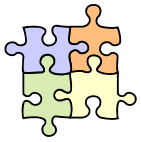
Shared Repository

- 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 e interconnessioni
  - è possibile coordinare questi componenti con riferimento a dati condivisi su cui operano



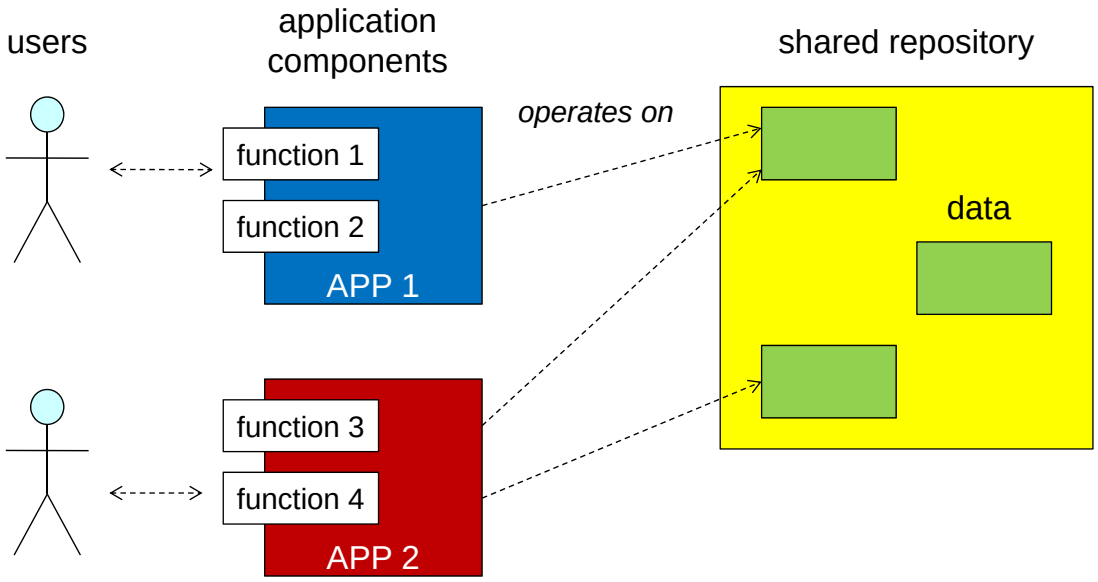
Shared Repository

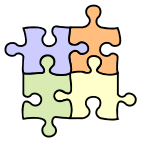
- Soluzione
  - mantieni i dati in un *repository* centrale *condiviso* da tutti i componenti funzionali dell'applicazione
  - fa guidare e coordinare il flusso di controllo della logica applicativa dalla disponibilità, dalla qualità e dallo stato dei dati nel repository
    - i componenti lavorano direttamente con i dati mantenuti nel repository condiviso
    - quando un componente crea, modifica o distrugge dei dati, questi cambiamenti sono accessibili anche agli altri componenti, che possono reagire di conseguenza
  - un caso particolare – molto comune – è quello in cui il repository condiviso è una base di dati
    - si parla in questo caso di *shared database*



Shared Repository

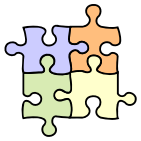
- Struttura della soluzione





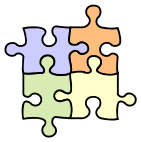
Shared Repository

- Discussione
  - l'architettura Shared Repository consente l'integrazione elementi funzionali (o applicazioni) con un flusso di controllo guidato dai dati
    - sostiene l'integrazione di elementi che operano sugli stessi dati – ma che non partecipano in un processo di business comune
  - 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
  - i dati gestiti dal repository possono essere di solito considerati dei Domain Object



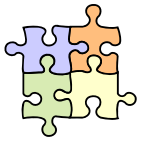
Shared Repository

- 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
  - può essere utile anche un meccanismo di notifica dei cambiamenti a componenti interessati – ad es., basato su Observer
  - 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"
    - di interesse per la "vista delle informazioni"



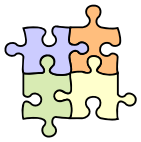
Shared Repository

- Discussione
  - l'integrazione è dunque basata sull'accesso a dati condivisi – 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 – spesso senza modifiche sul repository condiviso o con modifiche che hanno impatto limitato sulle applicazioni pre-esistenti
    - purché il repository consenta questa ulteriore integrazione
    - ma il repository si presta sempre ad ulteriori integrazioni?
  - per ora, è sufficiente sapere che esistono anche altri approcci all'integrazione di applicazioni



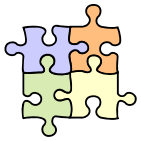
Conseguenze

- Modificabilità/evolvibilità
  - ☺ incoraggia un accoppiamento debole tra le varie applicazioni
  - ☺ possibile usare meta-dati per descrivere la struttura del repository
  - ☺ può consentire una modifica, anche dinamica, delle informazioni gestite dal repository e dei componenti che la aggiornano – se i componenti vedono la struttura logica (e non fisica) del repository
  - ☹ attenzione, le modifiche non sono sempre possibili/accettabili
- Affidabilità
  - ☺ possibilità di centralizzare funzionalità di backup/recovery
- Sicurezza
  - ☹ la centralizzazione dei dati può causare problemi di sicurezza – se non vengono prese precauzioni



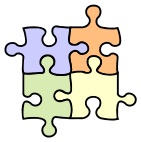
Sh. Rep. e tattiche per la modificabilità

- *Increase semantic coherence – increase cohesion*
  - le diverse applicazioni o componenti sono organizzate per coerenza semantica
- *Encapsulate – reduce coupling*
  - il repository può incapsulare la modalità di rappresentazione interna dei dati
- *Use an intermediary/Restrict dependencies – reduce coupling*
  - le applicazioni non comunicano tra loro direttamente – il repository è un intermediario
- *Use runtime registration – defer binding*
  - meccanismi di notifica dei cambiamenti, se presenti, sono basati su un meccanismo di registrazione a runtime



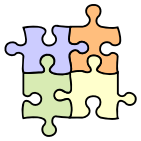
* Database Access Layer [POSA4]

- Il pattern architetturale **Database Access Layer**
  - guida la connessione tra elementi architetturali sviluppati con tecnologia orientata agli oggetti e una base di dati
    - chiamato anche *Object-Database Mapper* oppure *Object-Datastore Mapper*
  - un caso comune è il collegamento con una base di dati relazionale – è quello considerato qui di seguito
    - chiamato anche *Object-Relational Mapper*
  - 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
  - nota: questo pattern architetturale non appartiene alla categoria dei pattern fondamentali "dal fango alla struttura" di [POSA]
    - piuttosto, si tratta di un pattern di supporto per Shared Database



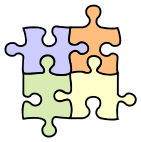
Database Access Layer

- 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 – e la persistenza di queste applicazioni viene spesso gestita mediante basi di dati relazionali
    - le tecnologie a oggetti sostengono progettazione e implementazione
    - le tecnologie relazionali forniscono accesso efficiente, efficace, affidabile e sicuro a basi di dati grandi, persistenti e condivise
  - ci sono però delle problematiche di accoppiamento tra queste tecnologie
    - ad es., modello dei dati, paradigma di accesso, ...



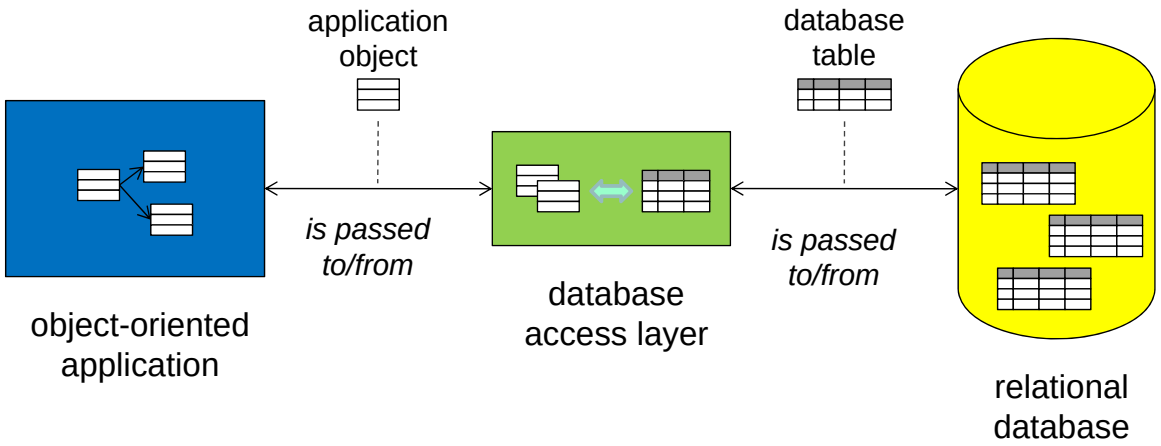
Database Access Layer

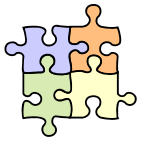
- Soluzione
  - introduci uno *strato* separato *per l'accesso alla base di dati* (*database access layer*) – tra l'applicazione e la base di dati relazionale
    - questo strato fornisce all'applicazione un'interfaccia per l'accesso ai dati stabile ed orientata agli oggetti
      - operazioni CRUD (Create, Read, Update, Delete) per l'accesso agli oggetti delle classi (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

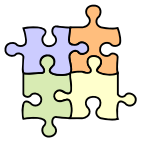
- Struttura della soluzione





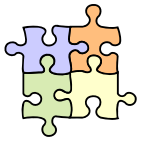
Database Access Layer

- Discussione – alcune motivazioni per l'uso di un Database Access Layer "generico"
  - risolvere il "disaccoppiamento d'impedenza"
    - i dati sono rappresentati in modo relazionale, ma la logica di business è meglio espressa con linguaggi ad oggetti
  - sostenere l'indipendenza dei dati
    - consentire che l'una o l'altra rappresentazione cambi – senza dover riscrivere tutta la logica d'accesso ai dati
  - indipendenza dai venditori di DBMS
    - i vari DBMS relazionali spesso parlano dialetti di SQL diversi – e comunque il loro uso "ottimale" è spesso diverso
  - lo sforzo della scrittura della logica d'accesso ai dati è spesso significativo – anche il 40% del costo di un intero progetto
    - un DAL generico può ridurre lo sforzo richiesto



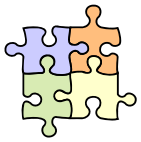
Database Access Layer

- Discussione
  - necessario un linguaggio di alto livello per la descrizione delle corrispondenze (ovvero, del mapping) 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
  - l'uso di un DAL o di un ORM "commerciale" è spesso una soluzione efficace – ma non sempre è anche una soluzione efficiente



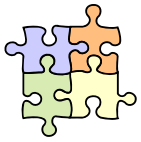
* Microkernel [POSA]

- Il pattern architetturale **Microkernel**
  - nella categoria [POSA4] “functional variation”
  - sostiene lo sviluppo di un insieme di applicazioni – che sono variazioni l’una dell’altra
    - non di una singola applicazione – piuttosto, di una “famiglia” (o “linea”) di prodotti software
  - tutte le diverse applicazioni sono basate sulla stessa architettura ed hanno un unico nucleo funzionale
  - le diverse applicazioni sono costruite in sede di compilazione o deployment



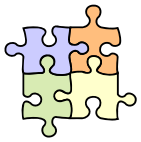
Microkernel

- 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 che offrono o nell’interfaccia utente
  - malgrado queste differenze, tutte le versioni dell’applicazioni dovrebbero essere basate su una stessa architettura comune e uno stesso nucleo funzionale comune
  - alcuni obiettivi di progetto
    - evitare variazioni architetturali
    - minimizzare lo sforzo di sviluppo ed evoluzione delle funzioni comuni
    - consentire nuove versioni e/o di variare le versioni esistenti

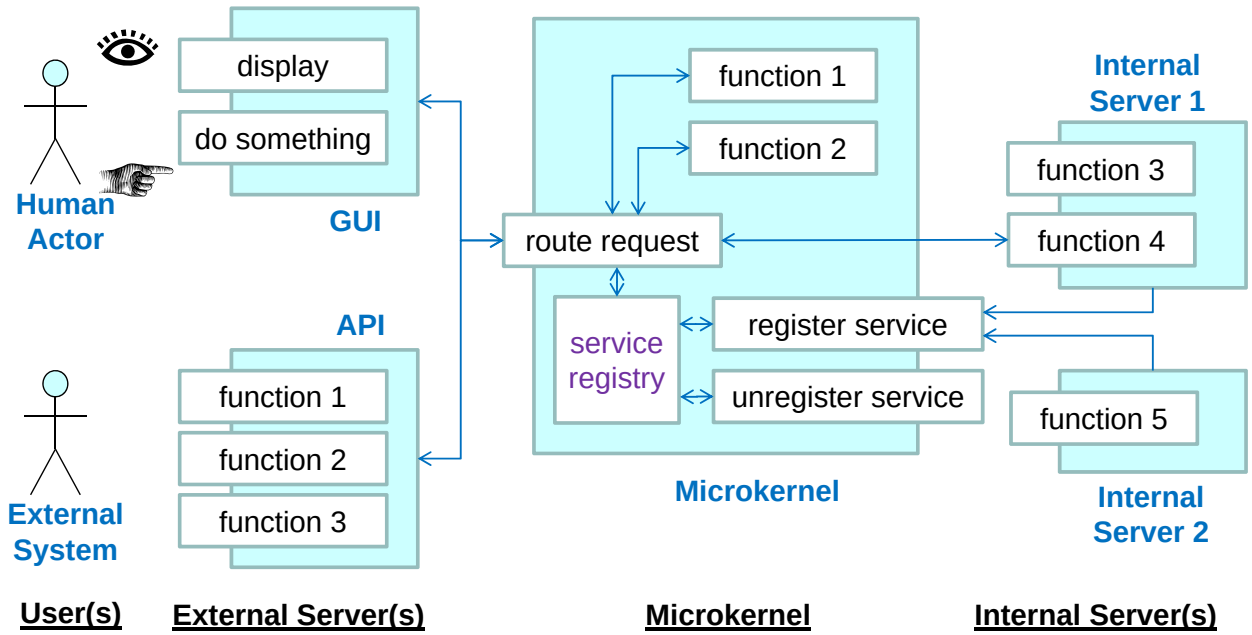


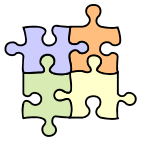
Microkernel

- Soluzione
  - componi le diverse versioni dell'applicazione estendendo un nucleo comune ma minimale, tramite un'infrastruttura plug-and-play – sulla base delle seguenti tipologie di elementi
    - un *microkernel*
      - implementa le funzionalità condivise da tutte le versioni
      - inoltre, fornisce l'infrastruttura per integrare le funzionalità specifiche delle diverse versioni
    - uno o più *server interni* (IS) – ciascun IS implementa delle funzionalità auto-contenute, ma specifiche per una versione
    - uno o più *server esterni* (ES) – ciascun ES implementa un'interfaccia utente o un'API specifica per una versione
  - ciascuna versione dell'applicazione è ottenuta connettendo il microkernel con dei corrispondenti server interni ed esterni



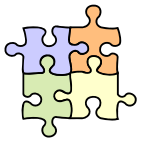
Struttura di una versione dell'applicazione





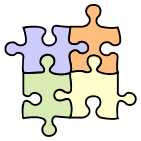
Microkernel

- Dinamica
  - in sede di deployment, il microkernel registra i server interni e il tipo di richieste che ciascuno di essi è in grado di soddisfare
    - varianti (per *Defer binding*) – collegamento durante la compilazione, oppure registrazione all'avvio del sistema
  - i client effettuano richieste al sistema solo tramite l'interfaccia o l'API di un server esterno
  - il server esterno propaga/delega le richieste al microkernel
  - gestione di una richiesta da parte del microkernel
    - se la richiesta è relativa a una funzionalità fondamentale, allora la funzionalità viene eseguita direttamente dal microkernel
    - altrimenti, la richiesta viene delegata dal microkernel al server interno in grado di soddisfare la richiesta



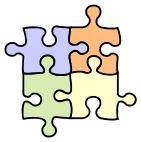
Discussione

- 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
  - l'evoluzione verso una nuova versione richiede la definizione di nuovi server interni ed esterni e la loro riconfigurazione
    - non sono richiesti cambiamenti né nel microkernel né nei server interni ed esterni pre-esistenti
  - lo sforzo di sviluppo e manutenzione è minimizzato – poiché ciascun servizio o UI è implementato una sola volta



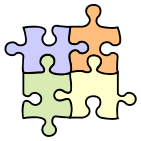
Discussione

- La struttura interna del microkernel è solitamente basata su Layers – ad es., potrebbe essere come segue
  - lo strato più basso consente di astrarre dalla piattaforma di sistema sottostante – per sostenere portabilità
  - il secondo strato implementa funzionalità di infrastruttura
  - lo strato successivo, le funzionalità di dominio condivise da tutte le versioni dell'applicazione – ad esempio, sulla base di Domain Object
  - lo strato più alto comprende i meccanismi per la configurazione dei server interni, come i meccanismi per effettuare il routing delle richieste ricevute dai server esterni
- Anche la struttura interna dei server interni può essere normalmente basata su Layers



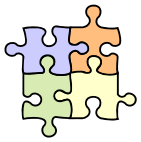
- Usi conosciuti

- Alcuni usi conosciuti del pattern Microkernel
  - nella realizzazione del kernel dei sistemi operativi
  - nella realizzazione di linee di prodotto
  - le *architetture a plug-in* possono essere considerate una variante di Microkernel
    - ad esempio, l'architettura a plug-in di Eclipse (E3) oppure quella di Mozilla Firefox



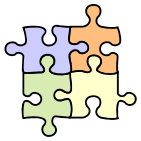
Microkernel e tattiche per la modificabilità

- *Increase semantic coherence – increase cohesion*
  - il microkernel implementa i servizi fondamentali
  - i server interni ed esterni estendono le funzionalità fornite dal microkernel
- *Abstract common services – reduce coupling*
  - il microkernel implementa dei servizi atomici (chiamati “meccanismi”) su cui si basa la costruzione di funzionalità più complesse – questi servizi sono resi astratti ai consumatori dei servizi
- *Encapsulate – reduce coupling*
  - le dipendenze specifiche per il sistema sono incapsulate nel microkernel



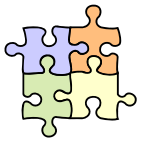
Microkernel e tattiche per la modificabilità

- *Restrict dependencies – reduce coupling*
  - i server interni sono accessibili solo al microkernel
  - il microkernel è accessibile solo dai server esterni
- *Use an intermediary – reduce coupling*
  - i server esterni fungono da intermediari nei confronti dei client esterni al sistema – proteggendo i client da dipendenze dirette
- *Defer binding*
  - nella formulazione originaria, una specifica versione dell’applicazione viene composta in sede di compilazione
  - tuttavia, è possibile prevedere che un’applicazione venga composta al momento del deployment, oppure anche al momento dell’avvio dell’applicazione stessa (architettura a plug-in)



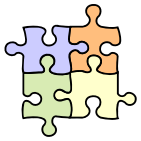
* Reflection [POSA]

- Il pattern architetturale **Reflection**
  - nella categoria [POSA4] “functional variation” – così come Microkernel
  - fornisce un meccanismo per cambiare la struttura e il comportamento di un sistema in modo dinamico – in particolare, anche a runtime
  - consente la modifica di aspetti fondamentali – ad es., delle strutture di dati e dei meccanismi di comunicazione



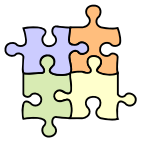
Reflection

- Contesto
  - un sistema che deve consentire delle variazioni – in qualunque momento, anche quando il sistema è attualmente “in operazione” – ovvero è rilasciato dal cliente o è a runtime
- Problema
  - il sistema deve poter evolvere nel tempo – ad es., un sistema la cui vita attesa è lunga, che deve rispondere a cambiamenti nei requisiti o del contesto di utilizzo (tecnologie e piattaforme che cambiano)
  - è difficile prevedere a priori tutte le modifiche e quando il sistema dovrà rispondere a richieste di cambiamento specifiche
  - per complicare le cose, le modifiche possono avvenire in qualunque momento – in particolare quando il sistema è “in operazione” e “a runtime” – pertanto non è accettabile realizzare le modifiche intervenendo sul codice



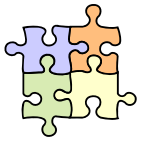
Reflection

- Soluzione
  - rappresenta esplicitamente le informazioni sulle proprietà e gli aspetti variabili della struttura, del comportamento e delle informazioni di stato del sistema – utilizzando un insieme di *meta-oggetti* o *meta-dati*
  - suddividi il sistema in due parti principali – in due strati
    - un *meta-livello* – contiene i meta-oggetti
    - un *livello base* – comprende la logica applicativa fondamentale del sistema
  - connetti il livello base con il meta-livello – in modo che cambiamenti di informazioni nel meta-livello influiscano sul comportamento effettivo del sistema
    - ovvero, fa sì che i componenti del livello base consultino gli appropriati meta-oggetti prima di eseguire qualunque comportamento soggetto a variazione



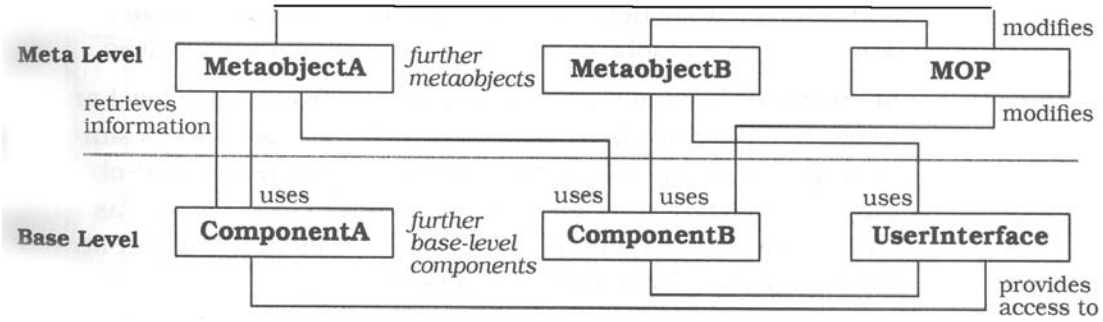
Reflection

- Soluzione (segue)
  - inoltre, fornisci un *protocollo* per amministrare e configurare *dinamicamente* gli oggetti del meta-livello
    - in pratica, si tratta un'interfaccia specializzata per amministratori e manutentori del sistema

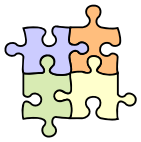


Struttura

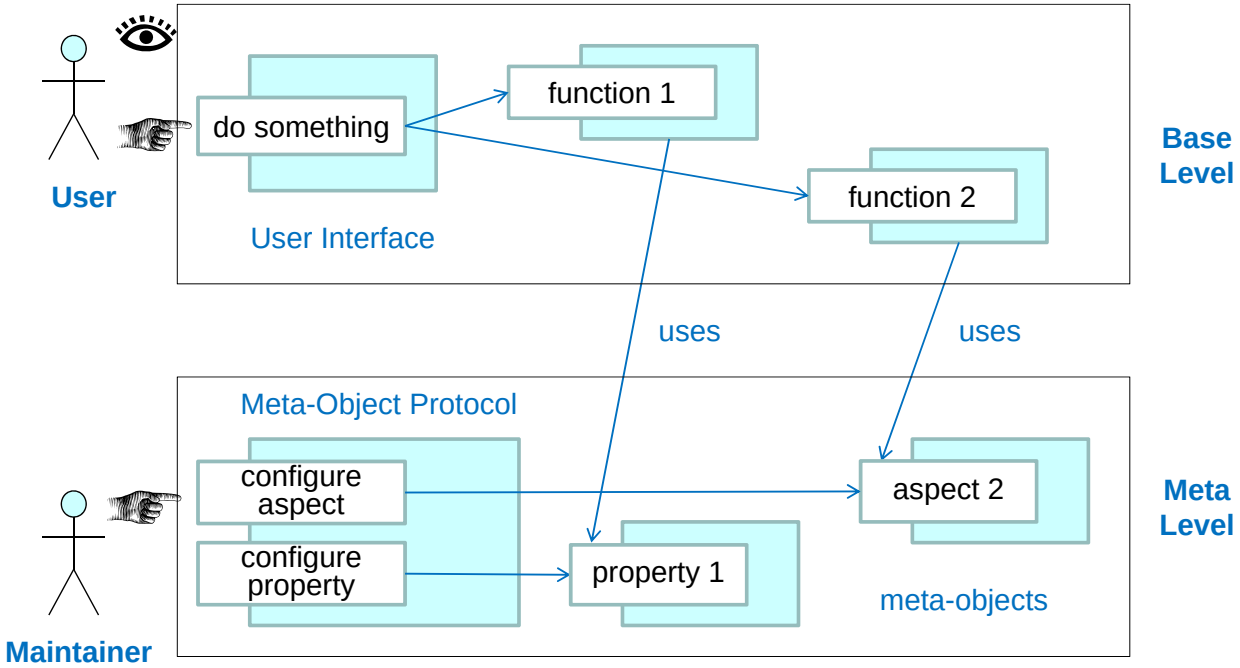
- Struttura
  - due strati chiamati livelli (levels) – meta e base
  - protocollo per accedere/cambiare i meta-dati – Meta Object Protocol (MOP)



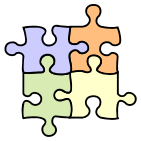
  - attenzione, la figura è fuorviante
    - intuitivamente è un'architettura a strati, ma è lo strato inferiore che dipende da quello superiore – e non viceversa



Struttura

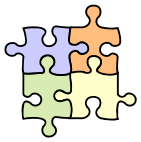


  - questa figura è "girata" rispetto alla precedente



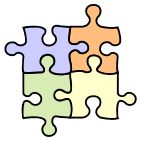
Esempio

- Il catalogo di una base di dati in un DBMS relazionale
  - lo schema di una base di dati è rappresentato mediante un *catalogo* – un insieme di relazioni con dei ruoli specifici
    - il catalogo appartiene al meta-livello – i meta-dati descrivono lo schema della base di dati in uso, nonché le corrispondenze tra livello logico e livello fisico
    - il livello base definisce la modalità di esecuzione dei comandi SQL – fa riferimento agli schemi descritti al meta-livello, nonché alle loro rappresentazione fisiche
  - le meta-informazioni vengono usate per realizzare il comportamento richiesto
    - le applicazioni usando il DBMS al livello base – in SQL
    - una richiesta, come ad es. “INSERT INTO R VALUES (...)”, potrebbe essere realizzata come “memorizza dei record nel file in cui è memorizzato R” e “aggiorna gli indici per R”



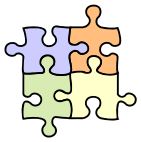
Esempio

- 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”



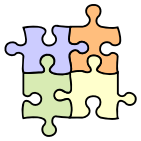
Reflection

- Alcune linee guida
  - inizia progettando un'applicazione in cui non è prevista alcuna variabilità – sulla base di un insieme di Domain Object
  - usa un metodo opportuno per identificare i punti di variazione nella struttura e nel comportamento del sistema – inoltre, determina tutte le informazioni che possono influire sul comportamento dell'applicazione
  - rappresenta ciascun comportamento o struttura variabile in un meta-oggetto separato – assegna tutti questi meta-oggetti al meta-livello
  - modifica l'implementazione di ciascun oggetto di dominio del progetto iniziale – in modo il suo comportamento sia determinato dai meta-oggetti – usando anche opportuni design pattern (ad es., factory, strategy, template method)
  - definisci un MOP per gestire i meta-oggetti



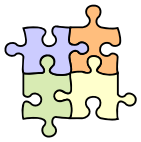
Conseguenze

- 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



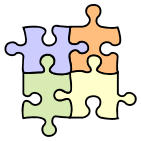
- Usi conosciuti

- Anche l'applicazione di Reflection è pervasiva
  - molti linguaggi di programmazione prevedono meccanismi di riflessione
  - molti sistemi sono basati su una nozione di meta-dati o meta-linguaggio
    - ad es., i DBMS
    - ad es., gli strumenti per lo sviluppo del software



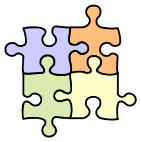
Reflection e tattiche per la modificabilità

- *Increase semantic coherence – increase cohesion*
  - i meta-oggetti rappresentano aspetti del software soggetti a cambiamento – responsabili di controllare tali cambiamenti
  - il livello base definisce la logica applicativa – usando i meta-oggetti per rimanere indipendente dagli aspetti soggetti a cambiamenti
- *Encapsulate – reduce coupling*
  - il MOP definisce un'interfaccia per gestire i meta-oggetti



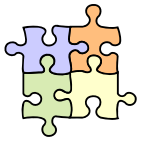
* Discussione

- 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



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
- Altri pattern architetturali – [POSA] e non – saranno discussi nel seguito del corso



Punto della situazione