Luca Cabibbo

Architettura dei Sistemi Software

Pattern architetturale Layers

dispensa asw330

ottobre 2024

Ogres are like onions.
Onions have layers. Ogres have layers.
You get it?
We both have layers.

Shrek

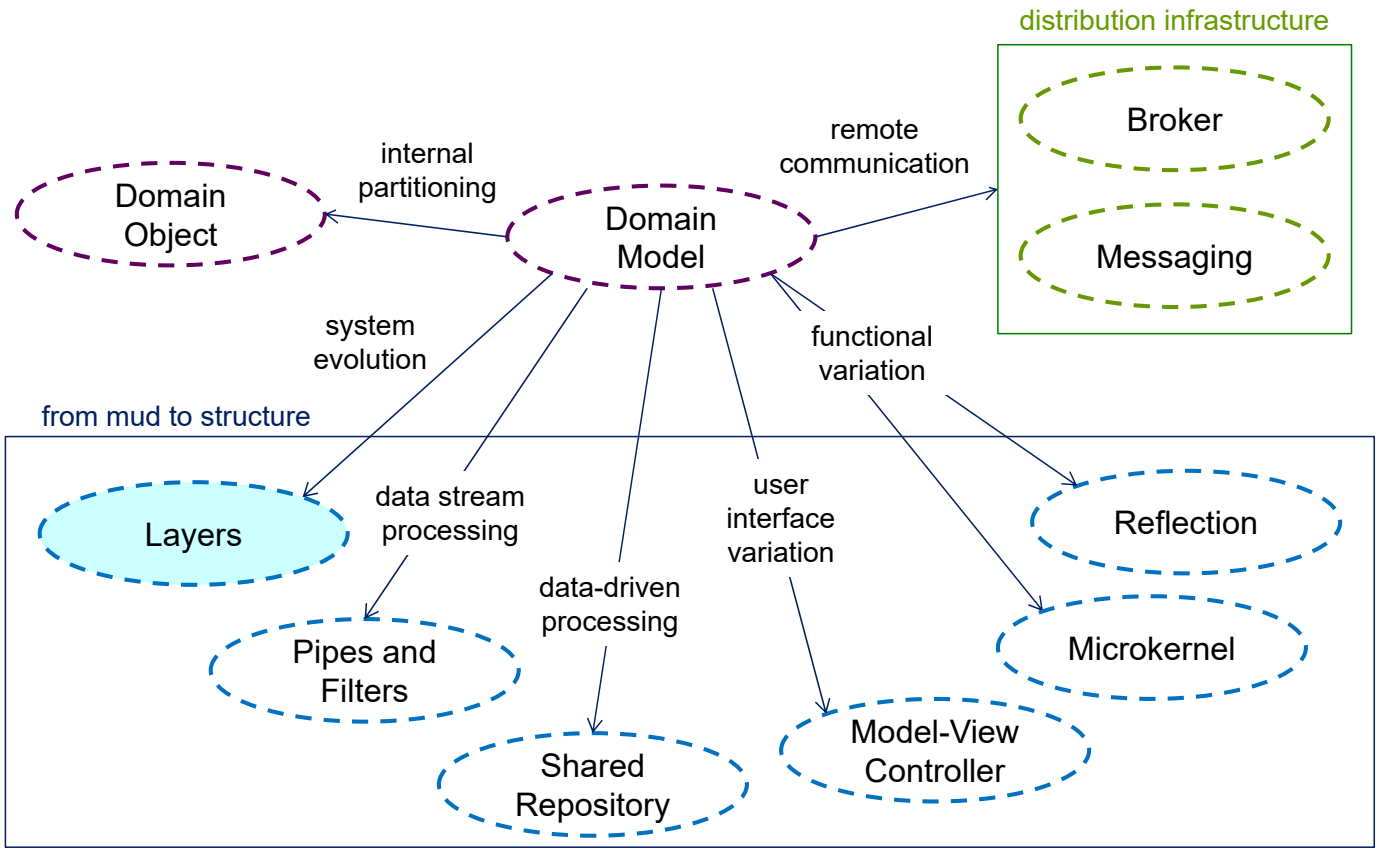
- Riferimenti

- Luca Cabibbo. **Architettura del Software**: Strutture e Qualità. Edizioni Efesto, 2021.
  - Capitolo 17, Pattern architetturale Layers
- [POSA1] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. **Pattern-Oriented Software Architecture (Volume 1)**: A System of Patterns. Wiley, 1996.
- [POSA4] Buschmann, F., Henney, K., and Schmidt, D.C. **Pattern-Oriented Software Architecture (Volume 4)**: A Pattern Language for Distributed Computing. Wiley, 2007.
- Bachmann, F., Bass, L., and Nord, R. Modifiability Tactics. Technical report CMU/SEI-2007-TR-002. 2007.

- Obiettivi e argomenti

- Obiettivi
  - presentare Layers – il pattern architetturale POSA più diffuso
  - fare qualche considerazione generale sui pattern architetturali – poiché questo è il primo pattern architetturale “concreto” che viene presentato
- Argomenti
  - Layers (POSA)
  - discussione

* Layers (POSA)



Layers

- Il pattern architetturale **Layers** – noto anche come *architettura a strati*
  - nella categoria “system evolution” di [POSA4]
  - aiuta a strutturare applicazioni che possono essere decomposte in gruppi di compiti, a diversi livelli di astrazione

Esempio

- Architettura dei protocolli di rete

Ogni strato si occupa di un protocollo diverso. Ogni volta ci si basa sulle operazioni fornite dallo strato in cui ci si trova.

Application	Layer 5	Provides network application protocols: HTTP, SMTP, FTP, etc.
Transport	Layer 4	Transmits messages from one host to another. Break messages into packets and guarantees delivery.
Network	Layer 3	Transmits packets from one host to another. Selects a route from sender to receiver.
Link	Layer 2	Transmits packets between adjacent hosts. Detects and corrects errors in bit sequences.
Physical	Layer 1	Transmits individual bits. Manages velocity, bit-code, connections, etc.

Layers

Il CONTESTO ci dà idea del protocollo che possiamo operare.
Il contesto è grande : devo decomporre
Il contesto ci dice anche come farlo : le parti in cui decomponiamo devono poter essere sviluppate in modo INDIPENDENTE (team diversi, poco dipendenti tra loro)

- Contesto
  - un sistema grande – richiede di essere decomposto
  - le diverse parti del sistema devono poter essere sviluppate e devono poter evolvere in modo indipendente
- Problema

Nota che non compare la parola strati. Il nome del pattern non compare MAI nella definizione del problema, che quindi non si pone MAI in termini della soluzione!

  - il sistema deve essere decomposto in parti che possano essere sviluppate e fatte evolvere in modo indipendente tra loro
    - è richiesta modificabilità, riusabilità e/o portabilità
  - la decomposizione deve essere basata su una separazione degli interessi ragionata – e non deve penalizzare altre qualità

Layers

- Soluzione
  - decomponi il sistema in una gerarchia verticale di elementi software, chiamati *strati*

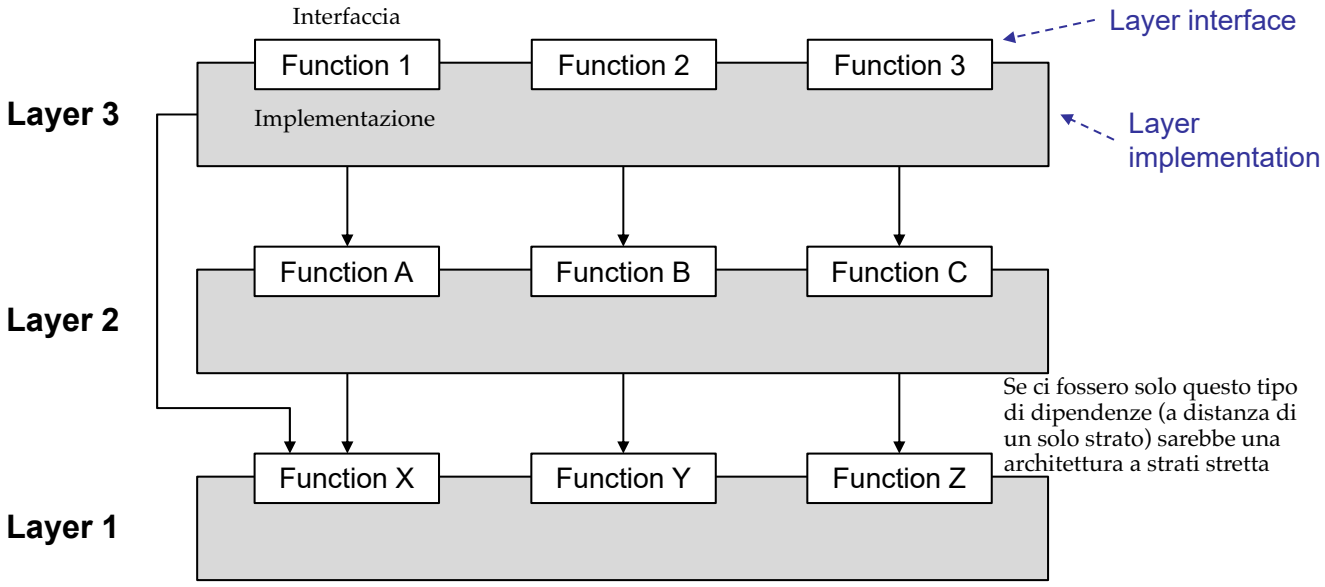
Definizione di un singolo elemento architetturale detto strato (ce ne possono, e devono, essere molteplici altrimenti non è un decomposizione)

  - ciascuno strato ha una responsabilità distinta e ben specifica
  - costruisci le funzionalità di ciascuno strato in modo che dipendano solo da se stesso o dagli strati inferiori
  - fornisci, in ciascuno strato, un'interfaccia che è separata dalla sua implementazione – e basa la comunicazione tra strati solo su queste interfacce

Le dipendenze sono solo dall'alto verso il basso, cioè un stato non può mai dipendere da quello che viene fatto in uno strato superiore

Layers

- Soluzione – struttura statica della soluzione



Se ci fossero solo questo tipo di dipendenze (a distanza di un solo strato) sarebbe una architettura a strati stretta

  - ma di che natura sono gli strati?

Che natura hanno? Si fa l'assunzione che gli elementi siano moduli, cioè codice.
Se invece gli elementi sono processi allora si chiama architettura client server.
Se invece gli elementi sono hardware allora si chiama architettura di deployment

Pattern, soluzione e scenari

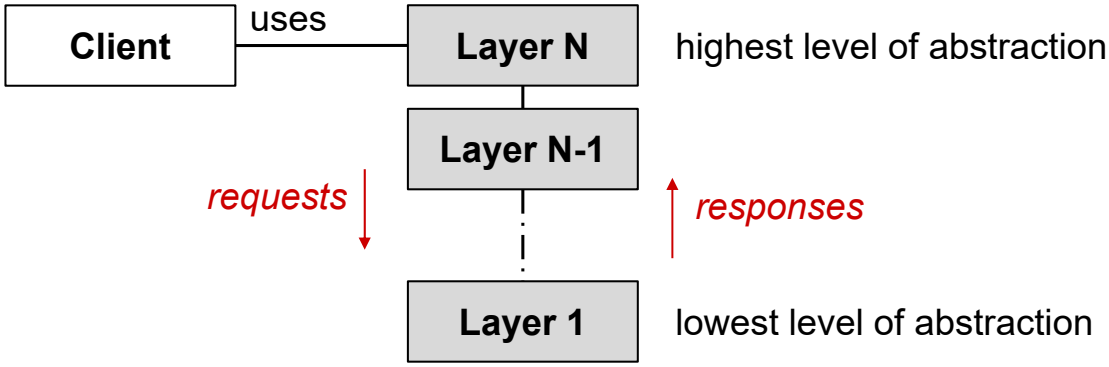
inizio ripasso

- La *soluzione* di un pattern descrive il "principio" o l'"idea risolutiva" fondamentale del pattern
  - comprende sia la *struttura statica* che il *comportamento dinamico* del pattern
- Gli aspetti dinamici di un pattern possono essere descritti sotto forma di scenari
  - ciascuno *scenario* descrive un possibile comportamento dinamico archetipale della soluzione
  - gli scenari possono anche descrivere modi diversi di applicare il pattern

Il pattern viene descritto in termini di SCENARI. Uno scenario è una situazione in cui potrà trovarsi il sistema insieme ad una descrizione di quella che dovrebbe essere la risposta del sistema stesso.
Gli scenari possono essere considerati comportamenti dinamici che siano archetipi della soluzione.

Scenario 1 – comunicazione top-down

- Lo scenario di Layers più comune è la *comunicazione top-down* – di tipo *richiesta-risposta*



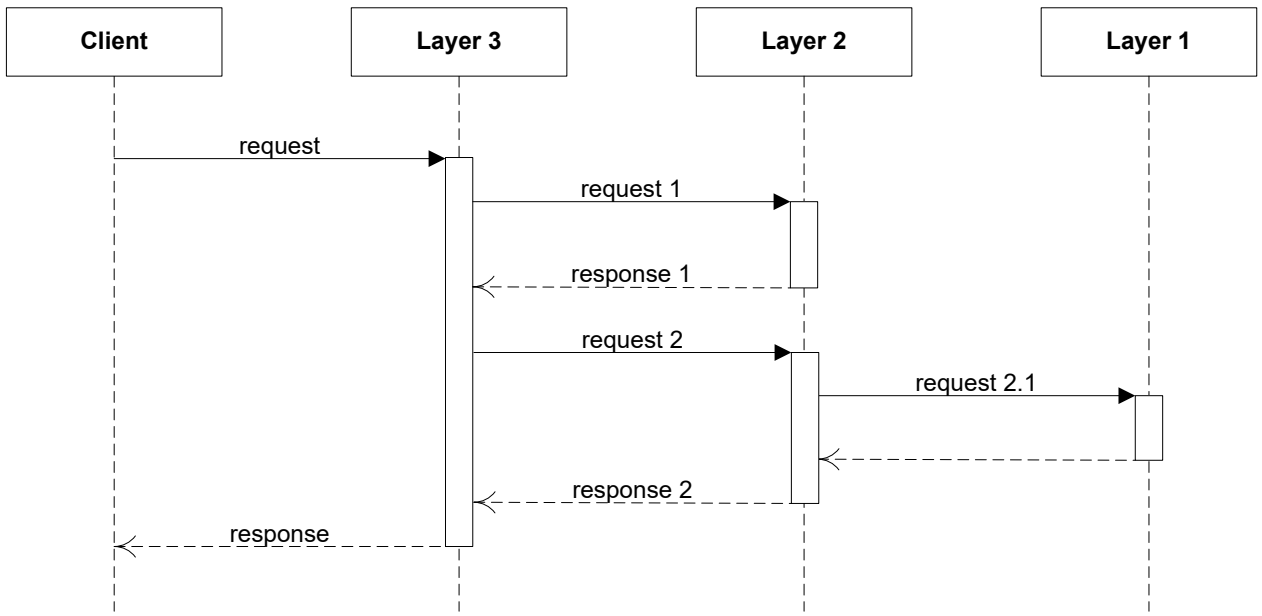
Si assume che ci sia un client del sistema che fa richieste allo stato più alto e si attende delle risposte. Queste richieste fatte allo strato più alto come vengono elaborate? Immaginiamo di trovarci in un'architettura standard di un certo tipo, con livello client, logica di business e strato dei servizi. Il client interagisce con lo strato di presentazione che ha la responsabilità di capire quale è la richiesta e a quel punto delega la gestione di questa richiesta all'altro strato sottostante che potrebbe essere quello della logica di business. Questo strato capisce chi interpellare e inoltra la richiesta, rielaborata, per esempio allo strato sottostante dei servizi tecnici della persistenza, che a sua volta interroga il db e estrapola delle informazioni. A questo punto le informazioni iniziano a risalire sotto forma di risposte, che cambiano forma in base allo stato in cui sono, fino ad arrivare al cliente.

è un tipo

Altro esempio richiesta http di tipo get

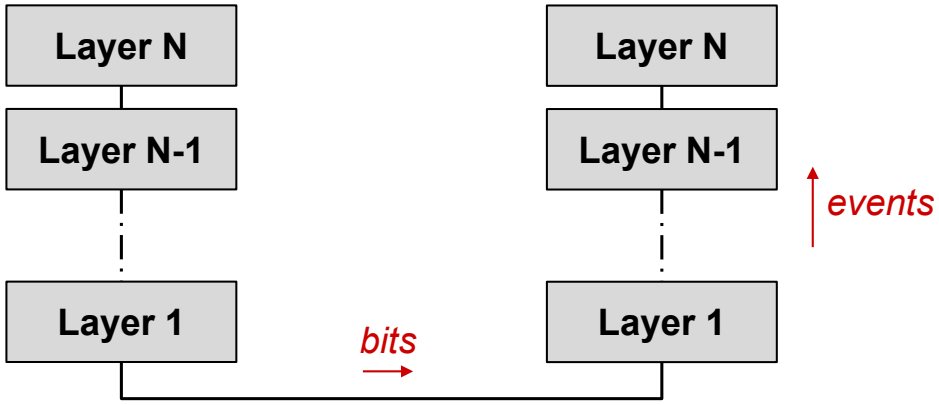
Scenario 1 – comunicazione top-down

- Lo scenario di Layers più comune è la *comunicazione top-down* – di tipo *richiesta-risposta*



Scenario 2 – comunicazione bottom-up

- Un altro scenario comune è la *comunicazione bottom-up* – basata sulla *notifica di eventi*



Questo scenario si verifica quando vengono ricevuti degli eventi nello strato più basso e si prevede che questi eventi risalgano l'architettura degli strati. Esempio, quando un server riceve una qualche richiesta, la riceve al livello più basso: riceve una sequenza di bit a livello 1. Questa informazione viene passata verso l'alto come una notifica di un evento, che strato dopo strato viene identificato nella sua interezza fino ad arrivare all'ultimo strato, per esempio identificata come una richiesta get http.

Altri scenari

- Sono possibili anche altri scenari
  - una richiesta allo strato N viene servita in uno strato intermedio – ad es., nello strato N-1 o N-2 Cioè non si deve arrivare fino al più basso (es l'info che serve è ancora in cache in qualche strato alto)
  - una notifica dallo strato 1 viene gestita in uno strato intermedio – ad es., nello strato 2 o 3 Cioè non si deve arrivare fino al più alto
  - una richiesta viene gestita da un sottoinsieme degli strati delle due pile

Scenari – discussione

- Due modalità principali di comunicazione per gli scenari di Layers – la comunicazione *richiesta-risposta* e la *notifica di eventi*
  - in corrispondenza, ciascuno strato può definire diverse interfacce
    - in uno strato più basso, verso l'alto
      - un'interfaccia (fornita) per ricevere l'invocazione di operazioni
      - un'interfaccia (fornita) per notificare eventi
    - in uno strato più alto, verso il basso
      - un'interfaccia (richiesta) per invocare operazioni
      - un'interfaccia (richiesta) per accettare notifiche di eventi

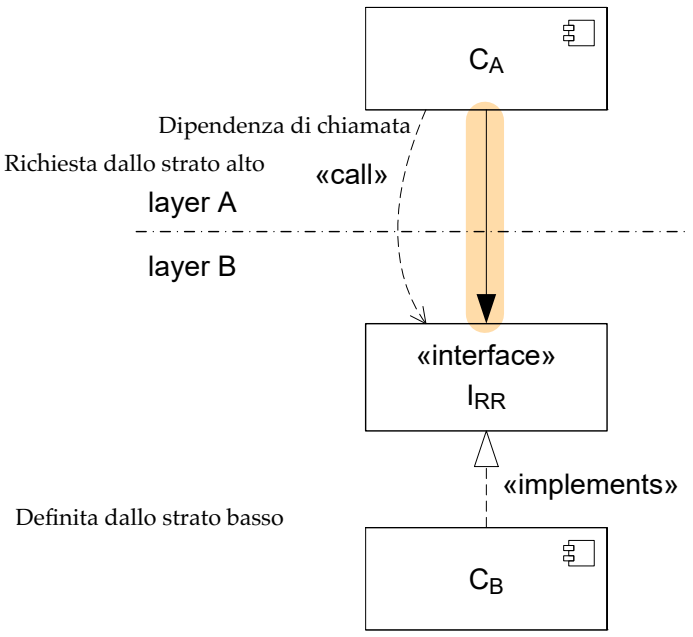
I nostri strati avranno interfacce sia verso l'alto che verso il basso. Verso l'alto saranno interfacce FORNITE (ti consento di eseguire un'operazione), verso il basso saranno interfacce RICHIESTE.

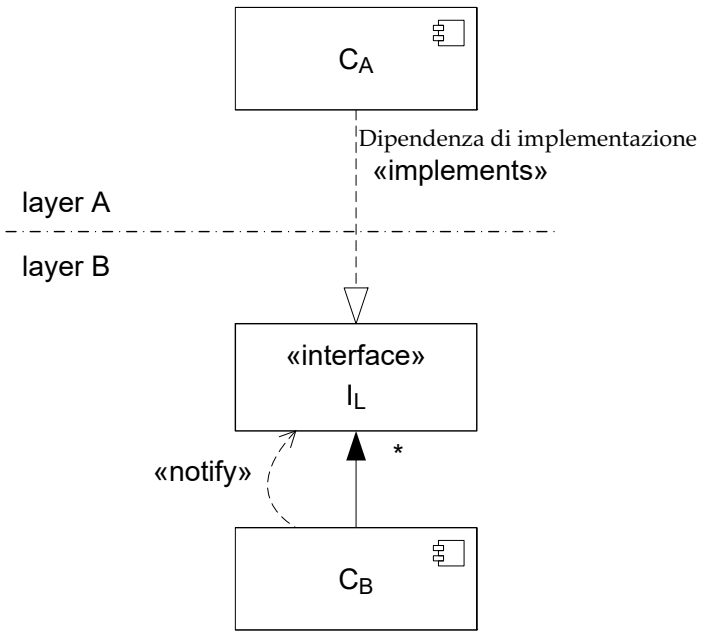
Nell'architettura a strati le dipendenze sono dall'alto verso il basso. Nello scenario di tipo richiesta-risposta io ho un'interfaccia di tipo richiesta-risposta che viene definita (quindi fornita) nello strato basso e richiesta dallo strato alto. Le dipendenze che ci interessano sono solo quelle che attraversano il confine tra strati, mentre all'interno di uno strato sono possibili tutte le dipendenze (qualsiasi verso).

Scenari – discussione

VEDI PROSSIMA SLIDE!

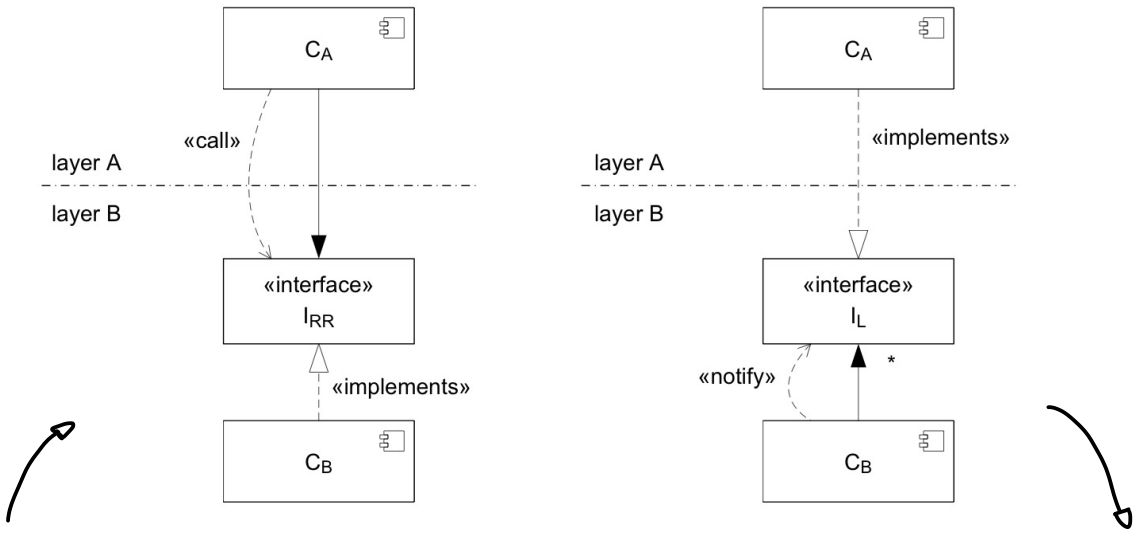
- Nell'architettura a strati, le dipendenze sono dall'alto verso il basso





Scenari – discussione

- Nell'architettura a strati, le dipendenze sono dall'alto verso il basso



In questo scenario questo fa una richiesta verso il basso quindi anche la DIPENDENZA è verso il basso. Ma nella notifica di eventi uno strato deve comunicare la ricezione dell'evento nello strato superiore, e quindi una notifica di evento sembrerebbe assimilabile ad una chiamata dal basso verso l'alto. Se fosse così però,non sarebbe soddisfatta la condizione delle dipendenze esclusive dall'alto verso il basso

Allora come viene gestito questo scenario nell'architettura a strati? Usando il principio di inversione delle dipendenze ed usando il pattern observer.
In particolare la gestione avviene in questo modo: c'è un'interfaccia listener; quando nel componente B si verifica un evento viene invocata un'operazione di questa interfaccia listener su tutti gli ascoltatori e questi ascoltatori, nel piano più alto, devono implementare questa interfaccia. Questa interfaccia è di proprietà dello strato più basso, e quindi l'unica dipendenza che attraversa gli strati è questa dell'implementazione tra il componente e l'interfaccia, ed è dall'alto verso il basso.

Quindi quando ci sono eventi che vanno notificati, QUALI sono gli eventi viene deciso in basso e poi lo strato (più alto) interessato a questi eventi deve implementare l'interfaccia per gestirli.

Gli eventi sono gestiti mediante chiamate dal basso verso l'alto, ma le dipendenze sono dall'alto verso il basso perché chi è interessato alla ricezione di questi eventi (strati più alti) deve riferirsi all'interfaccia in basso, rendendo soddisfatto così il vincolo

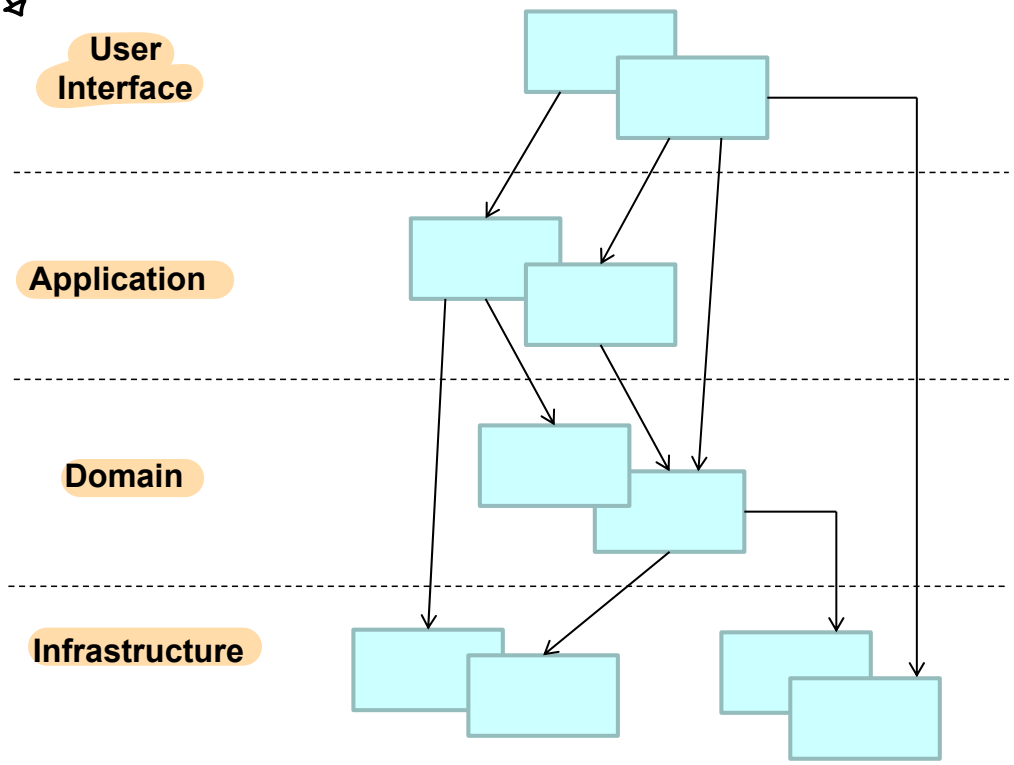
- Applicazione di Layers

- Definisci il criterio di partizionamento delle funzionalità
  - discusso più avanti
- Determina gli strati
  - determina il numero di strati e le loro responsabilità
- Specifica i servizi offerti da ciascuno strato
- Raffina la definizione degli strati – in modo iterativo
- Definisci l'interfaccia di ciascuno strato
  - quali i servizi offerti da ciascuno strato? quali le notifiche accettate?
- Struttura individualmente gli strati
  - discusso più avanti
- Specifica la comunicazione tra strati – top-down o bottom-up
- Disaccoppia gli strati – usa opportuni design pattern
- Progetta una strategia per la gestione degli errori

È importante capire che questo non è possibile per tutti i pattern. Nell'architettura a strati come si fa? Se si verifica un errore (che si verifica sotto forma di eccezione) o questo viene gestito all'interno dello strato, oppure lo strato si rende conto che non ha responsabilità sufficiente per gestirlo e chiede allo strato superiore di risolverlo (nota che quando l'errore viene inoltrato in alto viene riconvertito in un tipo di errore coerente con lo strato)

- Esempio: Layered Architecture [DDD]

Modo d'uso particolare dell'architettura a strati



Layered Architecture (DDD)

- Un esempio comune di architettura a strati è la *Layered Architecture* [DDD]
  - *presentation layer* – UI o presentazione
  - *application layer* – applicazione
  - *domain layer* – dominio
  - *infrastructure layer* – infrastruttura

} logica di business

Ogni responsabilità è di UNO strato

Layered Architecture

- Il *domain layer* rappresenta il modello di dominio
  - gli oggetti di dominio – liberi dalla responsabilità di visualizzarsi, di memorizzarsi e di gestire compiti dell'applicazione – sono focalizzati sulla rappresentazione del modello di dominio
  - è responsabile di gestire lo stato complessivo del sistema

Definisce le operazioni del dominio ma non i casi d'uso (quindi le singole operazioni ma non le funzionalità nell'insieme)

Layered Architecture

- L'*application layer* definisce i compiti che il sistema è chiamato a fare

  Implementa i casi di uso del sistema (es Elaborazione vendita, sono più funzionalità elementari combinate)

  - due varianti principali

    Controller GRASP

    - un insieme di facade sottili, che delegano lo svolgimento delle operazioni di sistema a oggetti opportuni dello strato del dominio
    - un insieme di classi più spesse che definiscono degli "operation script" – che implementano direttamente la logica applicativa, ma delegano la logica di dominio allo strato del dominio
  - è responsabile di gestire lo stato delle conversazioni/sessioni con i suoi client

- Criteri di decomposizione

- Nell'applicazione di Layers, è necessario scegliere un criterio di decomposizione delle funzionalità
  - in generale, questa decomposizione dovrebbe garantire che ciascun gruppo di funzionalità sia incapsulato in uno strato e che possa essere sviluppato ed evolvere indipendentemente dagli altri gruppi di funzionalità
  - in pratica, ci sono diversi criteri specifici comuni per la decomposizione delle funzionalità – astrazione, granularità, distanza dall'hardware e tasso di cambiamento

Criteri di decomposizione

- Criteri di decomposizione comuni
  - il criterio dell'*astrazione* può essere applicato nei sistemi che si devono occupare della gestione di diversi aspetti, a livelli di astrazione differenti
    - può essere utilizzato per realizzare una decomposizione di dominio
    - questo criterio viene spesso utilizzato anche per motivare una decomposizione tecnica in strati come presentazione, logica applicativa e infrastruttura/servizi tecnici – che sono responsabilità a livelli di astrazione differenti

Un sistema deve occuparsi di una serie di compiti. A ciascun compito del sistema viene assegnato un livello di astrazione. Metti i compiti con lo stesso livello di astrazione nello stesso strato. I livelli più alti si occupano dei compiti con livello più alto di astrazione. Viceversa gli strati più bassi si occupano di compiti di livello di astrazione più bassi. Nota che concettualmente i livelli più bassi non possono dipendere da quelli più alti. A volte astratto vuol dire più specifico, meno riutilizzabile.

esempio min 34.30

Criteri di decomposizione

- Criteri di decomposizione comuni
  - il criterio della *granularità* può portare a una suddivisione in uno strato con oggetti o servizi di business che vengono usati da uno strato di processi di business

    Cose più grandi stanno sopra, cose più piccole stanno sotto
  - il criterio della *distanza dall'hardware* può portare a una suddivisione con uno strato che definisce astrazioni del sistema operativo, uno strato di protocolli di comunicazione e uno strato di funzionalità applicative

    Più vicino all'hardware, più in basso
  - il criterio del *tasso di cambiamento atteso* suggerisce di separare le funzionalità mettendo in basso le cose più stabili e in alto quelle meno stabili (perché?)

    Perchè i cambiamenti si propagano dal basso verso l'alto, e quindi voglio evitare propagazioni a cascata

    Un'evidenza empirica suggerisce che ciò che cambia più frequentemente e velocemente è l'interfaccia utente, poi la logica di business e più raramente di tutti la struttura base di dati
  - la decomposizione in strati può anche fare riferimento alla combinazione di diversi criteri

Sul numero degli strati

- Qual è il numero “giusto” di strati da utilizzare?
  - l’obiettivo è favorire un’evoluzione indipendente degli strati
  - troppi pochi strati potrebbero non separare abbastanza i diversi aspetti che il sistema deve gestire
  - troppi strati potrebbero frammentare eccessivamente l’architettura del software – e rendere difficile la sua evoluzione
  - la scelta degli strati è critica, perché è difficile da cambiare

- Layers e team di sviluppo

- Se Layers viene applicato come decomposizione di primo livello, gli strati vengono poi in genere assegnati a team di sviluppo separati
  - per favorire un’evoluzione indipendente degli strati, la decomposizione in strati non può essere indipendente dall’organizzazione dei team
  - una considerazione importante riguarda il costo del coordinamento – perché quello inter-team è maggiore di quello intra-team
    - la maggior parte delle modifiche attese dovrebbero avere impatto solo su singoli strati
    - ogni strato dovrebbe dipendere solo in modo debole dagli strati inferiori

Una critica al pattern Layers

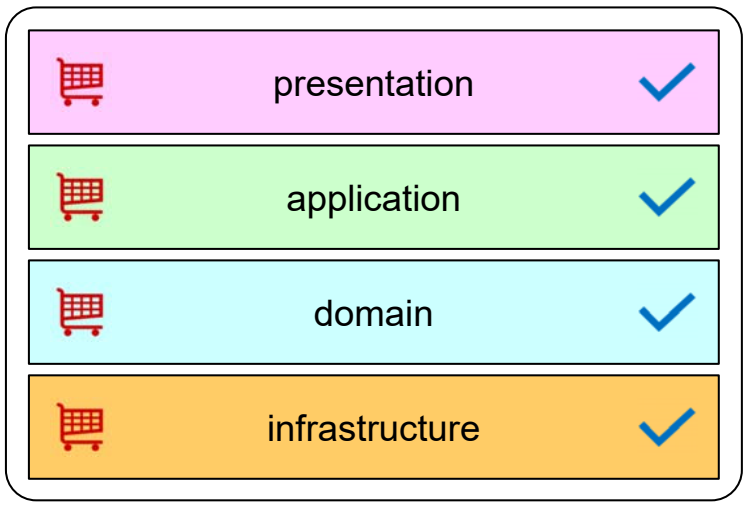
- Una critica al pattern Layers, legata alla legge di Conway
  - alcune organizzazioni applicano l'architettura a strati in modo "standard" – indipendentemente dal dominio applicativo dei sistemi software che sviluppano – e in corrispondenza adottano team di sviluppo mono-funzionali
  - in pratica, i sistemi software prodotti da queste organizzazioni risulteranno spesso caratterizzati da un accoppiamento elevato – anziché avere parti indipendenti
    - il problema non è nell'architettura a strati – ma nel modo in cui essa viene applicata!

Ad oggi questo tipo di architettura viene di solito applicata come decomposizione di secondo livello.

Una critica al pattern Layers

- Un esempio – un'applicazione di commercio elettronico – con un'architettura a strati "standard"
  - quali strati gestiscono la responsabilità del checkout?
  - quali strati devono essere modificati se cambia la modalità di gestione del checkout?

Cioè TUTTI gli strati si occupano dell'operazione di checkout, quindi se quello riguardante il checkout è un cambiamento atteso vale la pena sfruttare una decomposizione diversa



layered architecture

- Conseguenze

- Modificabilità
  - ☺ la modificabilità può essere alta
  - ☺ è possibile sostenere la portabilità

In realtà:

  - 😐 la modificabilità dipende da come i cambiamenti si ripercuotono sul sistema
  - ☹ alcuni cambiamenti potrebbero coinvolgere molti strati
  - ☹ è difficile cambiare la scelta degli strati e l'assegnazione di responsabilità agli strati

Quindi può essere alta ma in alcuni casi non lo è

Conseguenze

- Prestazioni
  - ☹ la comunicazione tra strati può penalizzare le prestazioni

Overhead se per fare una cosa devo coinvolgere tutti o tanti strati

  - 😐 allocare ciascuno strato a un diverso processo non migliora necessariamente le prestazioni
  - ☺ si possono migliorare le prestazioni associando un diverso thread di esecuzione a ciascun evento da elaborare

Cioè le cose possono andare meglio se ogni richiesta viene gestita da un thread separato

Conseguenze

È riduttivo dire che siccome ci sono tante parti diverse il sistema non è affidabile perché se si rompe una parte si rompe tutto, va studiato meglio nei vari aspetti

- Affidabilità (verificabilità) e disponibilità
  - ☹ se le richieste devono essere elaborate da molti strati, la verifica del sistema può essere più difficile
  - ☺ uno strato più alto può gestire guasti che si verificano negli strati inferiori
  - ☺ è possibile introdurre degli strati intermedi per effettuare il monitoraggio del sistema

Di positivo c'è la gestione degli errori

Conseguenze

- Sicurezza
  - ☺ è possibile inserire strati per introdurre opportuni meccanismi di sicurezza
- Altre conseguenze
  - ☺ possibilità di riusare strati
  - 😐 può aumentare gli sforzi iniziali e la complessità del sistema – ma questi sforzi sono poi di solito ripagati
  - ☹ può essere difficile stabilire la granularità/il numero/il livello di astrazione degli strati

- Discussione

- [POSA4] colloca Layers nella categoria “system evolution”
  - per sostenere la modificabilità, è basato sui principi di modularità e di separazione degli interessi
    - un progetto è *modulare* se è caratterizzato da accoppiamento basso e coesione alta
    - il principio di *separazione degli interessi* tende a separare interessi diversi in elementi differenti

- Usi conosciuti

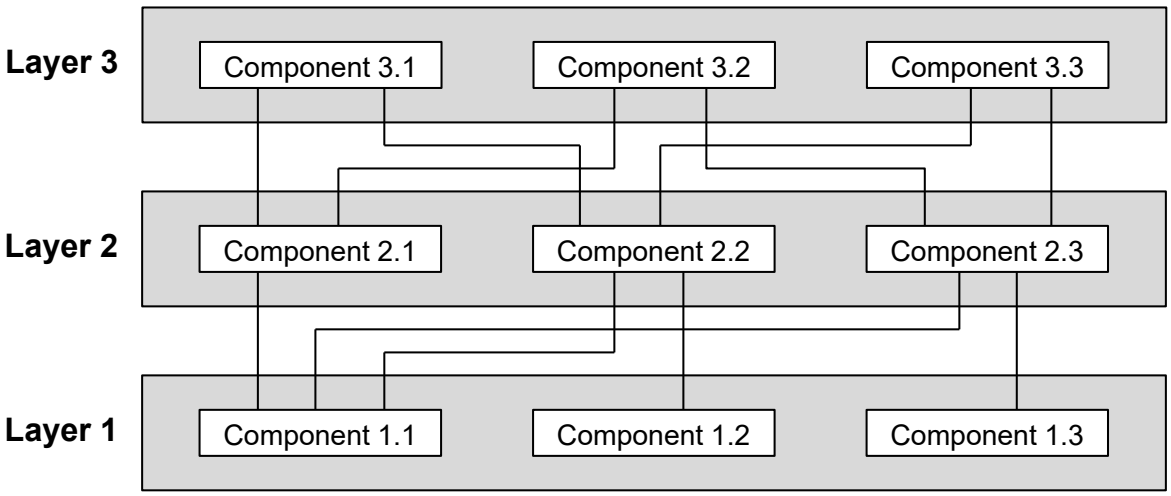
- Il pattern Layers è applicato in modo pervasivo
  - ad es., nei sistemi informativi e nei sistemi basati su macchine virtuali
  - nell’architettura client-server, a due o più livelli
  - nell’architettura ANSI a tre livelli dei DBMS (esterno, logico, interno)
    - che sostiene l’indipendenza dei livelli
  - nell’architettura a componenti e nell’architettura orientata ai servizi
  - nell’architettura del cloud
  - ...

- Sull'applicazione di Layers

- Il pattern Layers può essere applicato
  - come decomposizione di primo livello, per la strutturazione di un intero sistema
  - come decomposizione di secondo livello, per un componente o un sotto-sistema – infatti è comune organizzare a strati il codice dei singoli componenti architetturali

Sull'applicazione di Layers

- Gli strati possono anche essere strutturati internamente



* Discussione

- Layers è un pattern architetturale fondamentale, che viene usato nell'organizzazione di molti sistemi software
  - Layers, come gli altri pattern architetturali
    - identifica alcuni tipi specifici di elementi e di possibili modalità di interazione tra questi elementi
    - descrive criteri per effettuare la decomposizione sulla base di questi tipi di elementi e delle possibili relazioni tra essi
    - il criterio specifico di identificazione degli elementi/ componenti può far riferimento a qualche modalità di modellazione del dominio del sistema
    - discute la possibilità di raggiungere (o meno) alcuni attributi di qualità