UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Ingegneria dell'Informazione ed Elettrica e Matematica applicata

Corso di Laurea in Ingegneria Informatica

APPUNTI DI INGEGNERIA DEL SOFTWARE

DI FRANCESCO PIO CIRILLO

https://github.com/francescopiocirillo

“Non ti abbattere mai”

DISCLAIMER

Questi appunti sono stati realizzati a scopo puramente educativo e di condivisione della conoscenza. Non hanno alcun fine commerciale e non intendono violare alcun diritto d'autore o di proprietà intellettuale.

I contenuti di questo documento sono una rielaborazione personale di lezioni universitarie, materiali di studio e concetti appresi, espressi in modo originale ove possibile. Tuttavia, potrebbero includere riferimenti a fonti esterne, concetti accademici o traduzioni di materiale didattico fornito dai docenti o presente in libri di testo.

Se ritieni che questo documento contenga materiale di tua proprietà intellettuale e desideri richiederne la modifica o la rimozione, ti invito a contattarmi. Sarò disponibile a risolvere la questione nel minor tempo possibile.

In quanto autore di questi appunti non posso garantire l'accuratezza, la completezza o l'aggiornamento dei contenuti p e non mi assumo alcuna responsabilità per eventuali errori, omissioni o danni derivanti dall’uso di queste informazioni. L’uso di questo materiale è a totale discrezione e responsabilità dell’utente.

Design

PROGETTAZIONE ARCHITETTURALE vs PROGETTAZIONE DETTAGLIATA (ARCHITECTURAL vs DETAILED DESIGN)

I Requisiti gestiscono il "Cosa"; cosa si vuole che faccia il sistema, quali sono i suoi vincoli ecc.

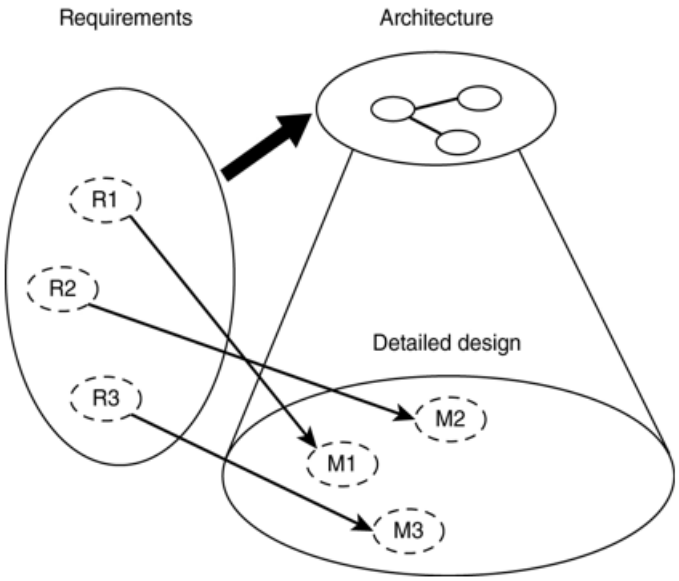
Il Design (progettazione) gestisce il "Come":

- come il Sistema è **decomposto in componenti**
- come questi componenti si **interfacciano e interagiscono tra loro**
- come ogni singolo componente **funziona** ecc.

Le fasi del Design sono tendenzialmente divise in:

- fase di **Design Architetturale**: una panoramica ad alto livello del sistema che va sviluppato che permette di identificare le componenti principali e definire le relazioni tra loro, il design architetturale è guidato dai requisiti (principalmente non funzionali);
- fase di **Design Dettagliato**: i componenti sono decomposti ad un livello di dettaglio molto più raffinato, il design dettaglio è guidato dai requisiti funzionali e dall'architettura.

Nei piccoli progetti è spesso assente una architettura esplicita e ci si concentra direttamente sul Design Dettagliato.



Nei **Processi Software tradizionali** il Design è quanto più dettagliato possibile e i programmatori si limitano a tradurre il Design in Codice.

Nelle **Metodologie Agili** invece il Design è definito ad un alto livello di astrazione e i programmatori si curano anche del design dettagliato.

DESIGN ARCHITETTURALE (ARCHITECTURAL DESIGN)

> 💡 Definire l'Architettura di un Software consiste nel definire la/e Struttura/e della Soluzione, che comprende gli elementi Software principali (moduli, componenti), le loro proprietà esternamente visibili (interfacce) e le relazioni tra loro (interazioni).

Ogni Sistema Software ha una **architettura** (anche se implicita) e addirittura un Sistema può avere multiple Strutture, vale a dire multipli modi di organizzare elementi sulla base della prospettiva.

Architectural Views (Visioni Architettoniche)

Kructhen Architectural Views (1995):

- **Logical View** (visione logica): definisce i moduli software, le componenti e le loro relazioni (gerarchia dei moduli, diagrammi di classe);
- **Process View** (visione di processo): come le componenti del sistema interagiscono e come sono distribuite su diversi processi (sequence diagrams, activity diagrams);
- **Development View** (visione di sviluppo): descrive l'organizzazione del team di sviluppo e come è gestito il processo di sviluppo (build scripts (non so cosa siano), version control systems (esempio: GIT));
- **Physical View** (visione fisica): informazioni sulle componenti hardware, sulla topologia della rete e su come le componenti software sono distribuite tra server e nodi (deployment diagrams).

Bass, Clements and Kazman (2012):

- **Module Views** (visione a moduli): rappresenta la decomposizione del sistema in moduli e sottosistemi e come i moduli dipendono l'uno dall'altro;
- **Run-time Views** (visione a tempo di esecuzione): indica come i moduli e/o i processi in esecuzione comunicano tra loro;
- **Allocation Views**: mappa i moduli software ad altri sistemi come strutture hardware, file sorgente, persone o team responsabili.

Le differenti visioni sono importanti e utili per diversi Stakeholders.

Meta-Architecturale Knowledge (conoscenza meta-architetturale)

Molte architetture hanno caratteristiche comuni, i Software Engineers hanno definito:

1. Architectural Styles or Patterns (Stili o pattern architetturali);
2. Architectural Tactics (tattiche architetturali);
3. Reference Architectures (architetture di riferimento, un documento o set di documenti che forniscono strutture consigliate e integrazioni di prodotti e servizi informatici per creare una soluzione).

Questi strumenti sono utilizzabili come punti di partenza per la definizione dell'architettura di un sistema e come mezzo di comunicazione efficace per fornire un'idea veloce della struttura ad alto

livello del sistema.

1. **ARCHITECTURAL STYLES (STILI ARCHITETTURALI)**

Gli stili architetturali sono dei punti di partenza riutilizzabili per le attività di Design Architetturale, importanti sono:

- Pipes-and-Filters (Pipeline)
- Event driven
- Client-Server
- Model-View-Controller (MVC)
- Layered (Three-Tier, Jakarta EE)
- Service Oriented (SOA, ESB)

Pipes-and-Filters

Si tratta di una architettura incentrata sui dati e strutturata intorno a come i dati si muovono attraverso l'applicazione.

L'applicazione prende i dati in input dopodiché i dati subiscono una serie di trasformazioni sequenziali e infine l'applicazione restituisce i dati processati come output.

Consiste di una serie di processi connessi da pipes dove l'output di un processo è l'input del successivo, è uno stile architettonico ampiamente usato nelle shell dei Sistemi Operativi e negli Scripts Unix.

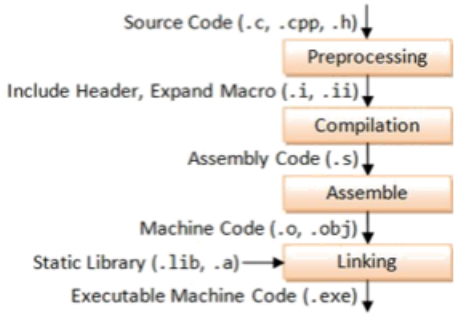
```
cat book.txt | sort | uniq
```

Le componenti di questa architettura sono:

- **Data Source** (sorgente dei dati): responsabile per la recezione dei dati in input e il trasferimento alla pipeline;
- **Pipe**: serve a trasferire e mettere in buffer i dati tra le altre componenti;
- **Filter**: sono degli step di processing dei data auto-contenuti che effettuano una funzione di trasformazione sui dati;
- **Data Sink**: riceve i dati processati alla fine della pipeline e li fornisce come output dell'applicazione.

I vantaggi di questo stile sono la **riutilizzabilità, verificabilità, manutenibilità e modificabilità** ma non è lo stile migliore a livello di prestazioni.

> Un esempio di utilizzo di Pipes-and-Filtes è il processo di Compilazione e Linking del Codice Sorgente.



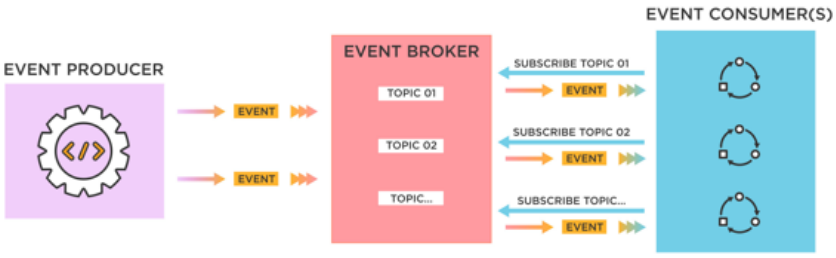
Event Driven

Il Software è composto di numerosi attori relativamente indipendenti che comunicano tra loro attraverso eventi al fine di ottenere un obiettivo coordinato.

Le Componenti sono:

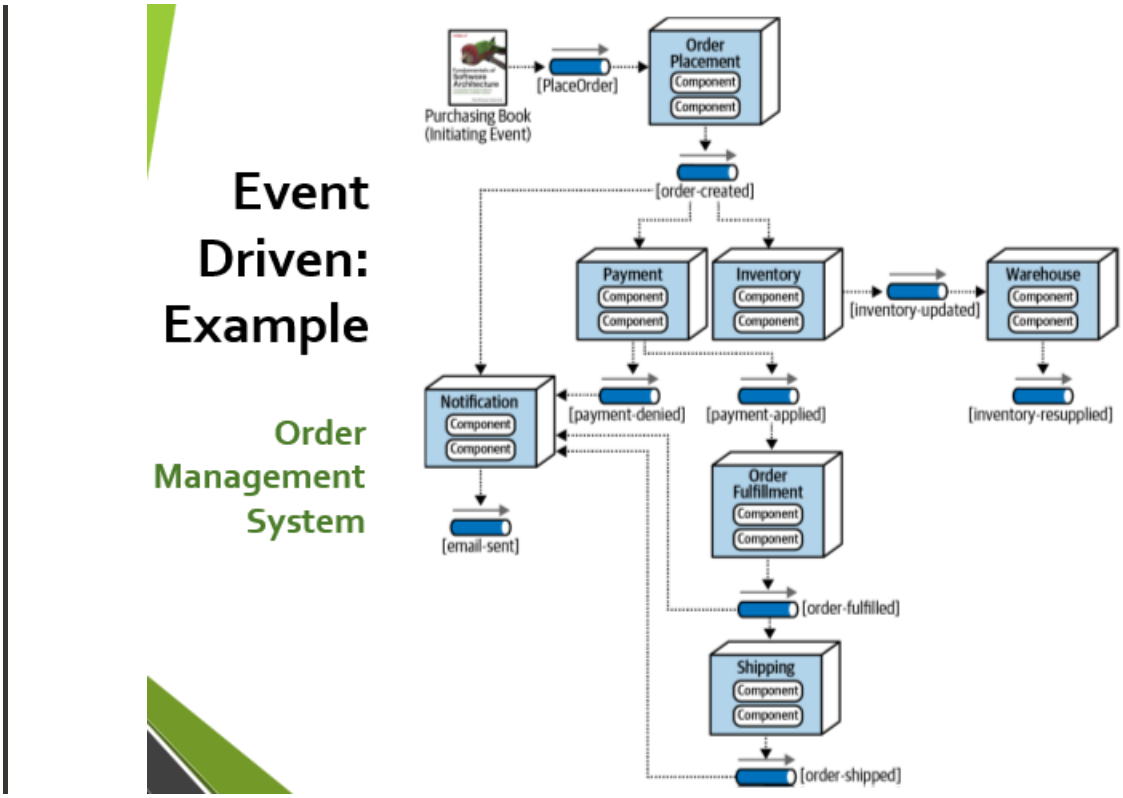
- **Event Producers**: generano o rilevano eventi e li trasmettono ai canali degli eventi. Non sanno nulla dei Consumers;
- **Event Brokers**: sono pipes nelle quali gli eventi sono trasmessi dagli Event Producer agli Event Consumer;
- **Event Consumers**: ricevono eventi e agiscono su di essi.

Lo Stile Event Driven si basa sul Paradigma publish/subscribe (pubblica/iscriviti) cioè i Producer pubblicano eventi sui canali e i Consumer si iscrivono ai canali per ricevere gli eventi.



Il **vantaggio importante è il disaccoppiamento tra Consumer e Producer.**

> Ex:

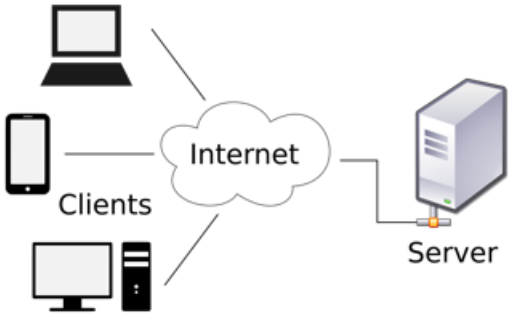


Client-Server

I Task o i Workloads sono divisi tra:

- i **providers** (fornitori) di una risorsa o servizio, che prendono il **nome di servers;**
- i **requesters** (richieditori) di una risorsa o servizio, che prendono il **nome di clients.**

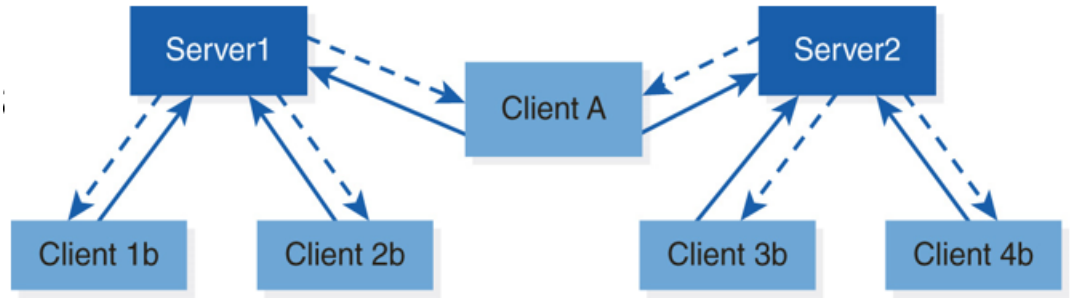
Il Client invia la sua richiesta attraverso un dispositivo abilitato alla rete (spesso la comunicazione viaggia su reti di calcolatori), il Server della rete accetta e processa la richiesta dell'utente e infine spedisce la risposta al Client.

> Ex: Browsers e Web Servers
>
> 
>

In questo stile si fa differenza tra:

- i **Thin Client** (clienti magri) che hanno potenza di calcolo e risorse limitate e sono principalmente usati come interfacce utente per interagire con applicazioni e dati che sono ospitate dai server;
- i **Fat Client** (clienti grassi) che possono effettuare una quantità sostanziale di computazioni localmente ed eseguono GUI ricche e anche una parte della logica dell'applicazione.

Potrebbero essere coinvolti molti server.



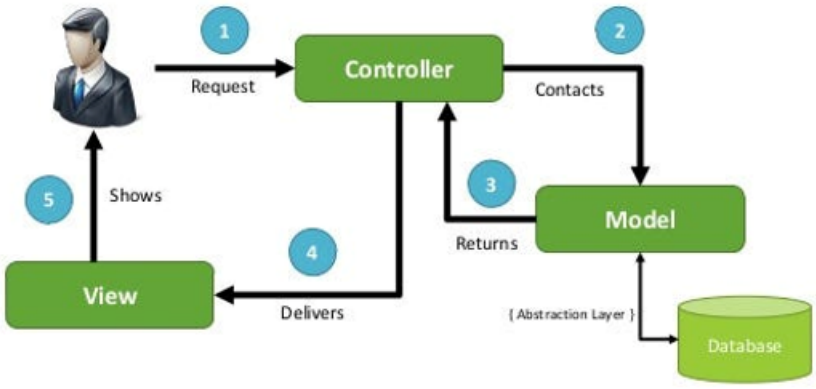
▼ Esempi di Client-Server:

Web Browsing: il web server fornisce le pagine web richieste, immagini e altre risorse. Il browser renderizza il contenuto per farlo visualizzare all'utente.
Email: il client email (webmail o email app) si connette all'email server (SMTP) per inviare e ricevere email e accedere alla mailbox.
File Sharing: i clients (computer o altro) prendono e mettono file sul server.
Database Management: applicazioni software inviano richieste ad un server database che recupera, aggiorna o manipola i dati.
Remote Desktop: un client può connettersi ad un remote server o ad un altro computer per avere accesso al suo ambiente desktop e alle sue applicazioni.

Model-View-Controller

Questo stile separa l'applicazione in tre componenti interconnesse:

- **Model**: incapsula i dati e fornisce metodi per interagire con i dati e manipolarli;
- **View**: è responsabile per la presentazione dei dati all'utente e per la gestione dei componenti dell'interfaccia utente;
- **Controller**: agisce da intermediario tra Model e View, nello specifico comunica con il modello per richiedere dati o causare cambiamenti basati sulla azioni dell'utente, la vista viene poi aggiornata per riflettere i cambiamenti del modello o le interazioni dell'utente.

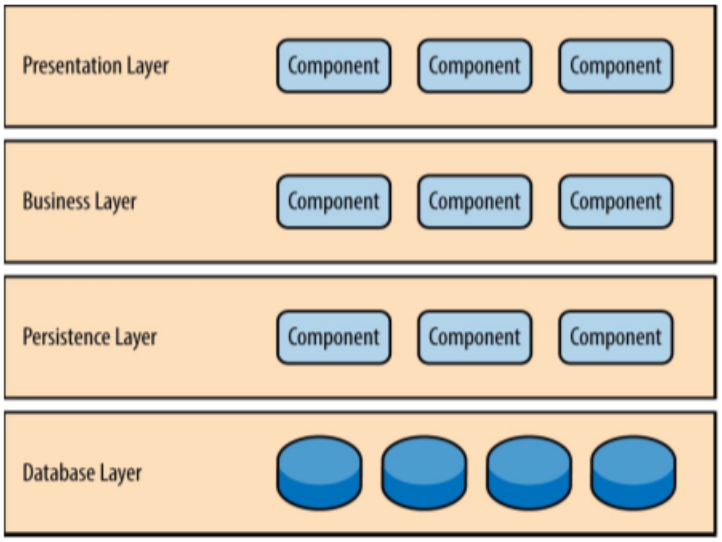


Ad un singolo Model possono essere associate diverse Views per divere presentazioni degli stessi dati.

MVC è uno stile architetturale popolare per lo sviluppo di Web Applications (di solito in congiunzione a client-server), Desktop Applications e Mobile Apps.

MVC permette lo sviluppo di software ben strutturato, manutenibile ed estensibile.

Layered



Quello a strati è uno dei più comuni stili architetturali.

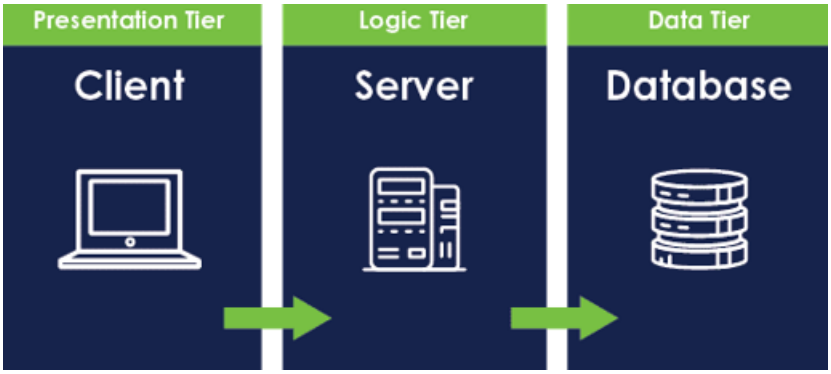
Moduli o componenti con funzionalità simili vengono organizzati in strati orizzontali (>=2), ogni strato svolge uno specifico ruolo e i componenti possono comunicare solo con componenti di strati adiacenti. La divisione in strati permette ai team di dividere il carico di lavoro.

L'architettura Layered è **tecnicamente partizionata**, cioè le componenti sono raggruppate sulla base del loro ruolo tecnico (come presentazione o business) e non sulla base del dominio (come ad esempio cliente).

La principale criticità delle architetture Layered è che ogni Business Domain è sparpagliato su tutti i livelli e questo rende difficile fare modifiche ad un Domain.

Layered: Three Tiers

Si tratta di un caso speciale di architettura Layered basata su MVC, la divisione in strati è la seguente:



- **Presentation Tier (View):** responsabile della gestione dell'interazione con l'utente e della presentazione dell'interfaccia utente dell'applicazione;
- **Business Logic Tier (Controller):** contiene il cuore della logica dell'applicazione, le business rules e i processi che guidano la funzionalità dell'app;
- **Data Tier (Model)**: responsabile della gestione e dell'immagazzinamento dei dati dell'applicazione.

Layered: Jakarta EE

> 💡 Jakarta EE (ex Java EE) è un set di specifiche che estendono Java per sviluppare applicazioni di livello enterprise (aziendale).

La sua architettura di riferimento è basata su quattro livelli:

- **Client Tier:** rappresenta l'interfaccia utente renderizzata su web browsers, mobile applications e desktop applications (in un certo senso View);
- **Web Tier:** responsabile per la gestione di interfacce utente web-based e sui contenuti web dinamici (Java Server Pages) (in un certo senso View);
- **Business (Application) Tier:** contiene il cuore della Business Logic dell'applicazione (Enterprise java Beans) (in un certo senso Controller);
- **Persistence (Data) Tier:** è responsabile per lo storage e il recupero dei dati (in un certo senso Model).

Service Oriented

L'architettura viene divisa in servizi indipendenti, ogni servizio incapsula una **business activity** e comunica con gli altri servizi attraverso **network calls** (chiamate di rete).

- Accoppiamento allentato (Loose Coupling):

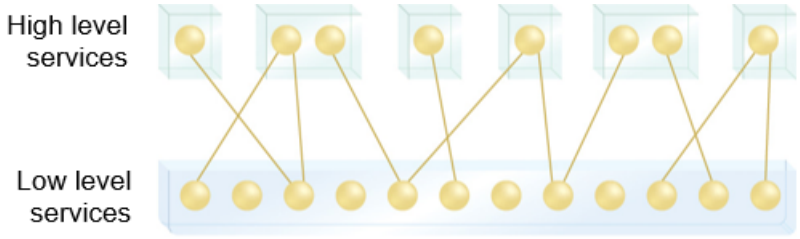
  I servizi non hanno dipendenze che li legano strettamente tra loro, questo permette flessibilità e

riusabilità dei servizi e permette di modificare o rimpiazzare un servizio senza avere effetto sugli altri.

- Interoperabilità:
  I
  Servizi lavorano insieme indipendentemente da tecnologie, piattaforme o linguaggi di programmazione utilizzati attraverso l'uso di protocolli di comunicazione standard e formati dei dati standard.

Service Oriented: Orchestrazione (Orchestration)

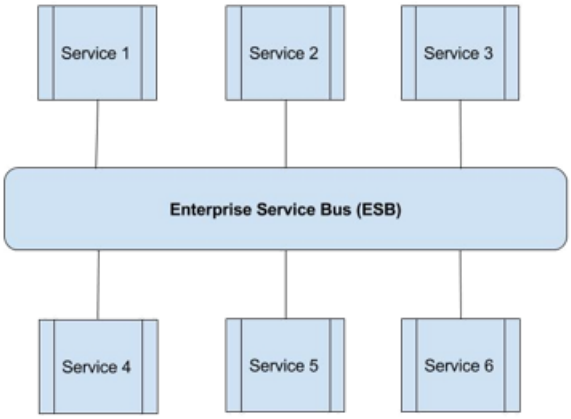
> 💡 SOA (service-oriented architecture) permette la composizione di servizi in processi di business o workflows più complessi.



Service Oriented: Enterprise Service Bus

Tutte le comunicazioni avvengono su un singolo canale (invece di far usare ai servizi delle chiamate su rete).

Questo permette a ESB di ottenere metriche su ogni servizio e direzionare le chiamate a diverse versioni.



2. **Tattiche Architetturali**

Risolvono specifici problemi architetturali senza avere effetto sulla struttura generale

EX:

> in una architettura three-tiers si teme che **un componente possa andare offline senza che nessuno se ne renda conto** allora si può implementare un nuovo componente dedito al failure detection.
>
> - Tattica 1 (heartbeat): ogni componente invia un messaggio al fault detector a intervalli prestabiliti.
> - Tattica 2 (Ping/Echo): il fault detector invia un messaggio alle altre componenti e attende risposta.

Segue una lista di alcune popolari Tattiche Architetturali:

- **Caching:** conservare e riutilizzare dati acceduti di frequente al fine di ridurre la latenza e migliorare le performance del sistema;
- **Load balancing:** distribuire le richieste in arrivo o il traffico di rete su multipli server o risorse per migliorare la scalabilità, la fault tolerance (tolleranza al fallimento) e la disponibilità (availability);
- **Compression**: ridurre le dimensioni dei dati al fine di ridurre i requisiti sullo storage e sulla bandwidth;
- **Encryption**: proteggere i dati codificandoli in un modo che permette l'accesso solo agli utenti autorizzati;
- **Connection Pooling**: riutilizzare le connessioni sui database per evitare l'overhead di creare una nuova connessione per ogni richiesta.

3. **Architetture di riferimento (reference architectures)**

> 💡 Le reference architectures sono blueprint o template dettagliati e specifici per il design di uno specifico tipo di sistema o applicazione.

Le Reference Architecture:

- si riferiscono ad un particolare tipo di dominio o applicazione (e-commerce, content management, IoT, ...)
- definiscono le componenti e le loro relazioni, includendo le tecnologie raccomandate e le best practices.

Mentre gli Stili Architetturali forniscono un framework di alto livello, le architetture di riferimento invece forniscono istruzioni pratiche per il system design e l'implementazione.

DETAILED DESIGN

Il Design Architetturale necessita di essere rifinito per produrre un design dettagliato, ma quanto dettagliato?

- **Livello di Dettaglio più fine possibile**: l'implementazione diventa una mappatura di quel design rispetto al linguaggio scelto per l'implementazione.
- **Bilanciamento dei dettagli:** in pratica raramente è necessario specificare ogni dettaglio durante la fase di Design anche perché raggiungere il livello di dettaglio più fine possibile è costoso e molto time-consuming, inoltre un design più semplificato è più flessibile in caso di cambiamento.

Module Decomposition

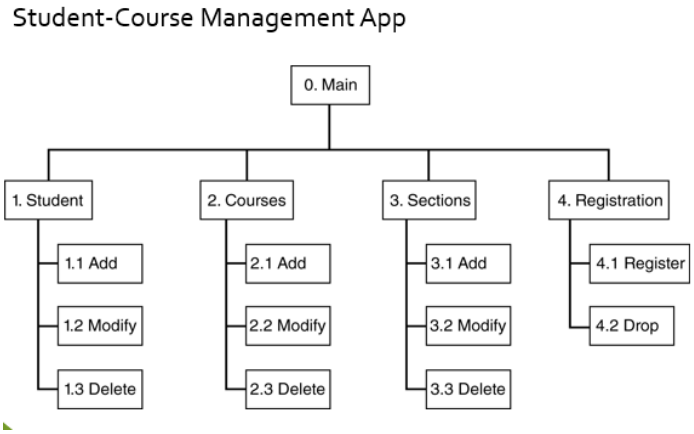
Consiste nel decomporre una funzione o un modulo in moduli più piccoli dove i moduli più piccoli incapsulano specifici tasks o responsabilità che sono chiamati dal modulo principale.

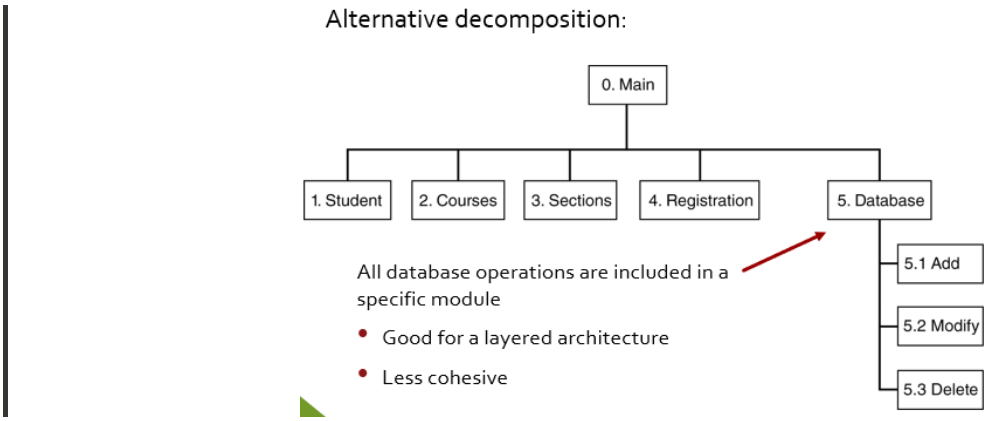
La Module Decomposition è principalmente utilizzata:

- nel **programming strutturato**, poiché promuove la produzione di codice modulare e ben organizzato e facilita debugging e manutenzione;
- nella **programmazione OO**, viene inizialmente decomposto un sistema in sottosistemi gestibili e vengono anche decomposti metodi di difficile implementazione.

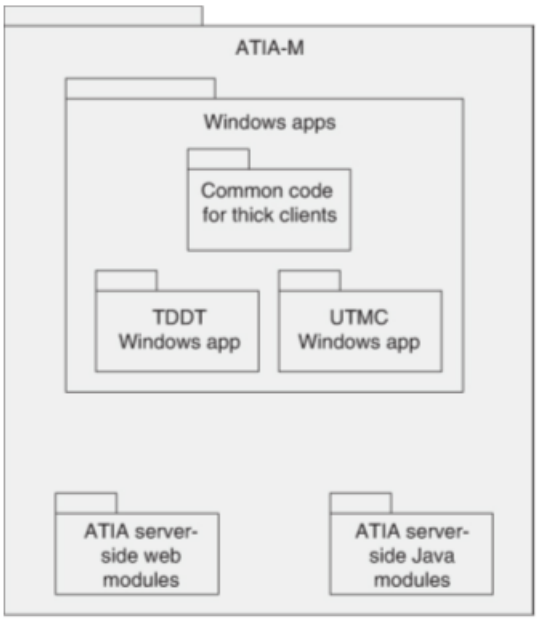
I moduli dovrebbero essere granulari abbastanza da renderli chiari ma non troppo al punto da renderli eccessivamente frammentati.

Seguono due esempi di **decomposition tree** (il secondo esempio mostra tutte le operazioni relative a Database in uno specifico modulo, questa divisione tecnica è buona per una architettura Layered ma fa perdere di coesione):

> Ex:
>
> 
>



Alternativamente agli alberi di decomposizione è possibile utilizzare un **UML Package Diagram** come visibile in basso.



Object Oriented Design

Molti moderni sistemi software sono sviluppati usando OO e la maggior parte della documentazione è realizzata per mezzo dei diagrammi UML.

Quali diagrammi UML vanno usati in quali fasi?

Durante il **Requirements Engineering**:

- Use Cases and Use-Case Diagrams;
- Preliminary (Conceptual) class diagrams (opt.)
- Preliminary sequence diagrams (opt.)

Durante il **Design**:

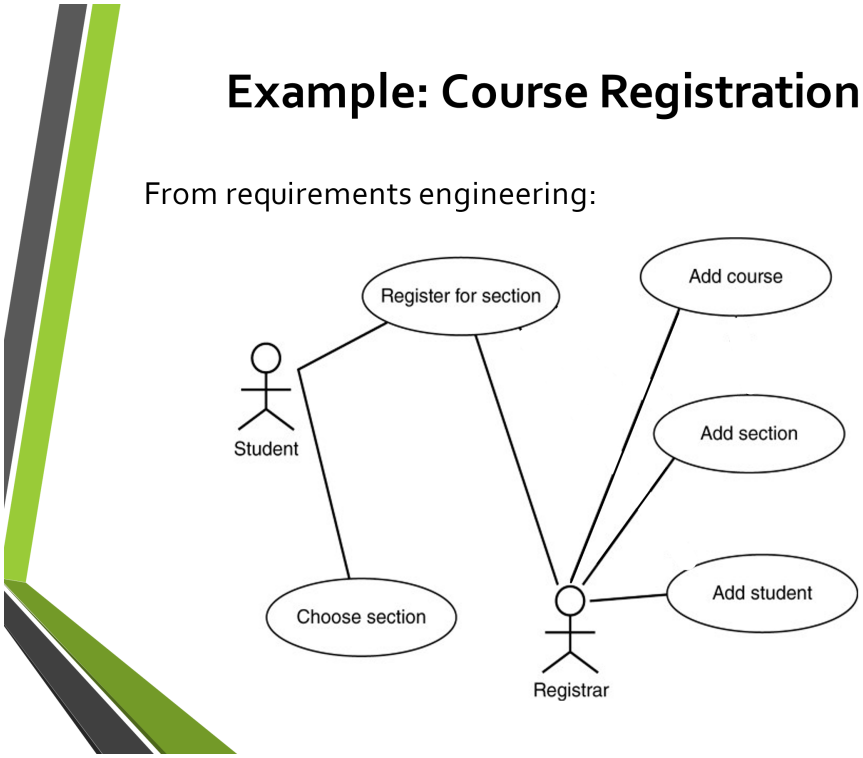
-Use Cases refinement (opt.) cioè System Use Cases

-Class diagrams (detailed)

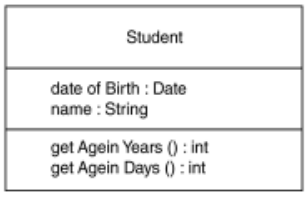
-Sequence/Collaboration diagrams

-Other diagrams when needed

Segue un esempio di utilizzo di vari diagrammi UML per realizzare la Registrazione ad un Corso di Laurea:

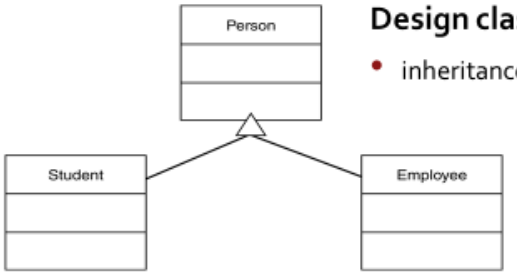


Example: Course Registration



Design classes:

- names
- attributes
- operations



Design class relations:

- inheritance

Example: Course Registration

Design class relations:

- associations

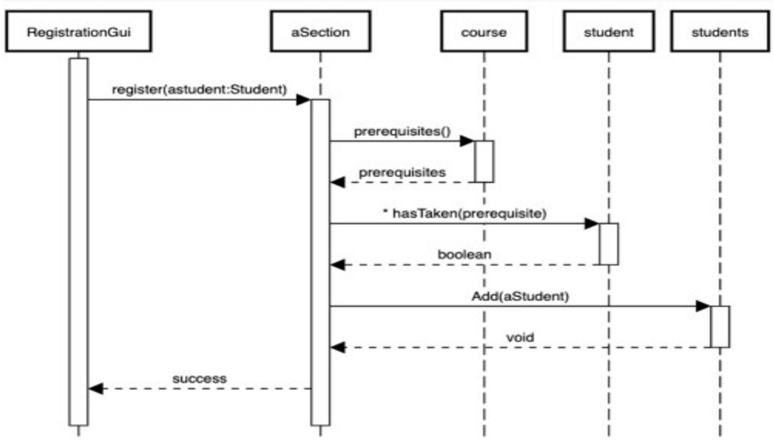
Design class relations:

- composition

Example: Course Registration

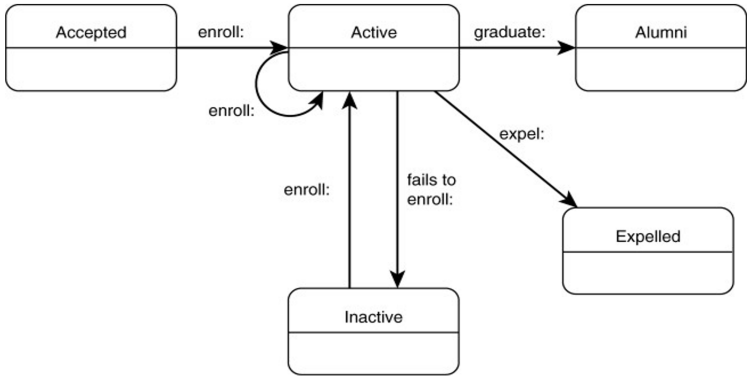
Design object interactions:

- for all (main) use cases and tasks within use cases
- granularity matters!



Example: Course Registration

Show the **lifetime behaviour** of some interesting objects (if relevant)



I Diagrammi elencati negli esempi di cui sopra sono più che necessari per la maggior parte dei piccoli progetti ma (se necessario) si possono anche utilizzare tra i tanti i seguenti diagrammi:

- **Activity Diagrams**, per descrivere alcuni algoritmi importanti;
- **Object Diagrams**, per semplificare alcune relazioni complesse tra classi;
- **Package Diagrams**, per mostrare come le classi sono organizzate in packages (nei grandi sistemi);
- **Component** e **Deployement Diagrams**, per i sistemi distribuiti;
- **Timing Diagrams**, se il tempo è importante (ad esempio nei sistemi real time).

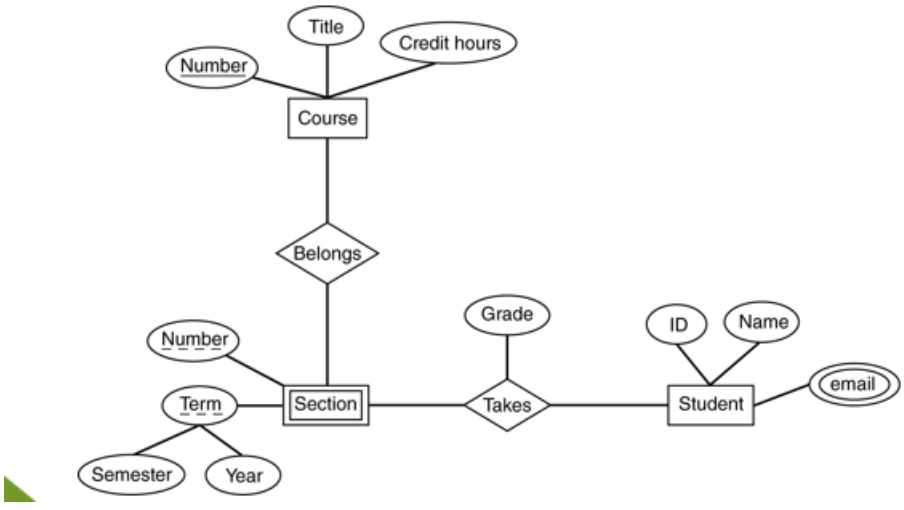
Database Design

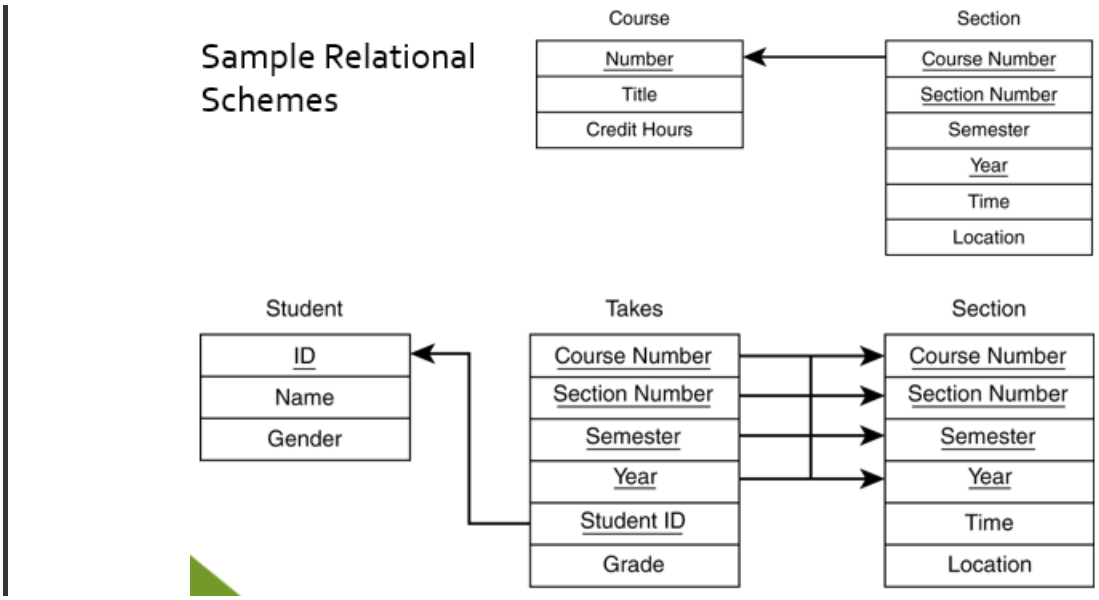
Un database può essere usato per ottenere **persistenza**.

I passi per il Design di un **Database Relazionale** sono:

- **Data Modelling**: idenificare entità, attributi e relazioni (entity-relationship diagram);
- **Logical Database Design**: definire database tables con foreign key relationships (relazioni chiave estranea) (relational scheme, schema relazionale);
- **Physical Database Design**: vengono prese decisioni sui tipi di dati e sugli indici.

Il Design di Database non è approfondito in questo corso.

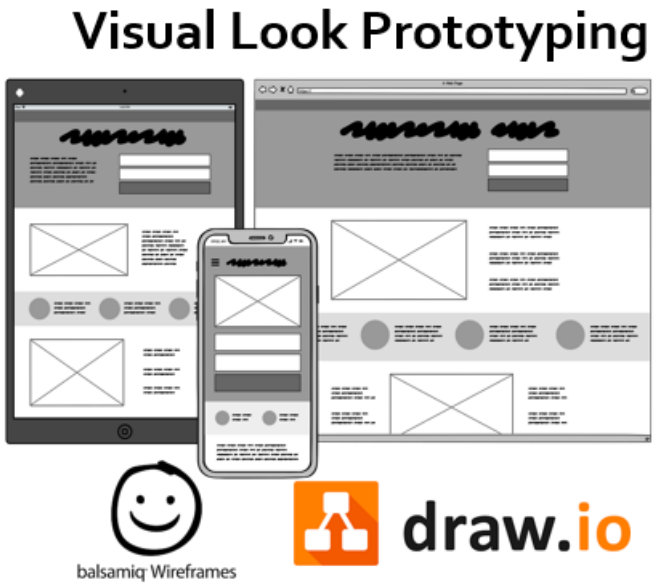
> Segue un esempio di Database Design:
>
> A sample Entity-Relationship (ER) diagram
>
> 
>



User Interface Design

L'interfaccia utente (UI) è parte più visibile del Software.

È fondamentale farla nel modo giusto tuttavia lo UI design è molto diverso dalla programmazione, si basa sulla **psicologia**, le **scienze cognitive**, l'**estetica** e l'**arte** e può essere un campo difficile per gli ingegneri del software, tanto che nella maggior parte dei casi il design dell'interfaccia grafica è lasciata a degli specialisti.

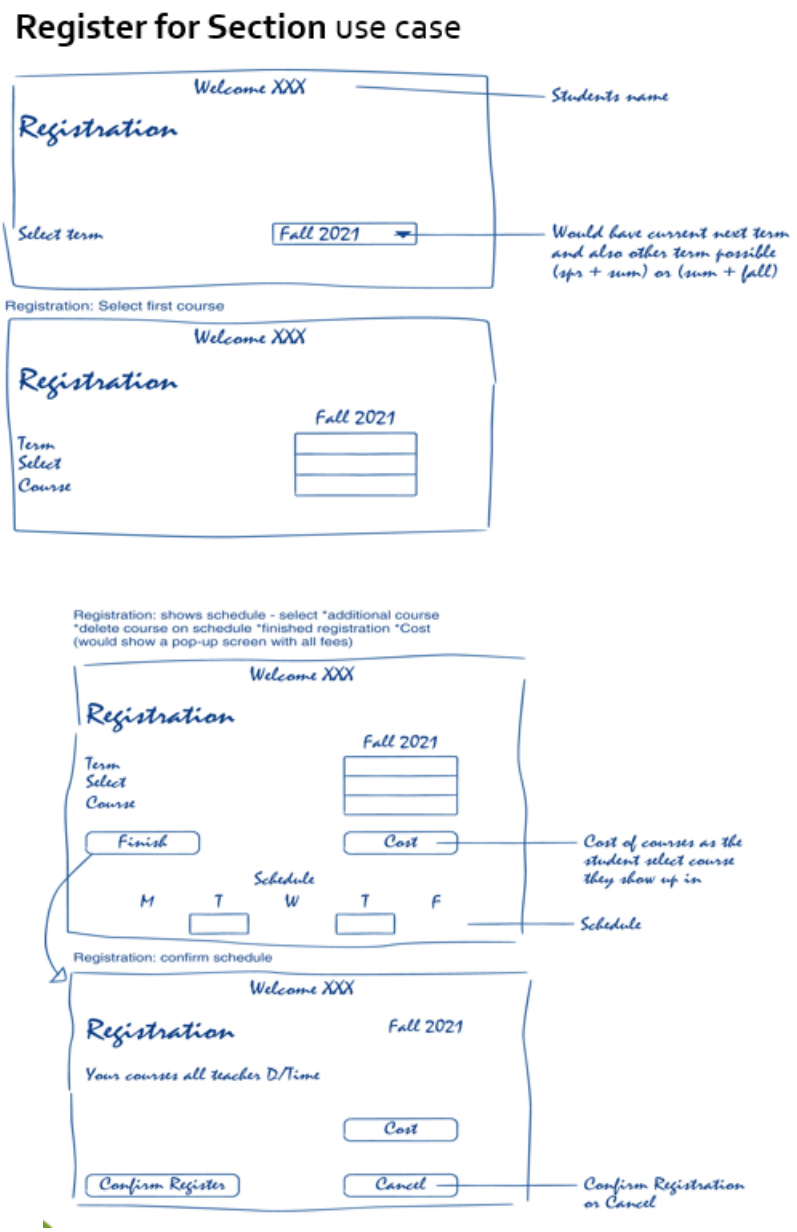


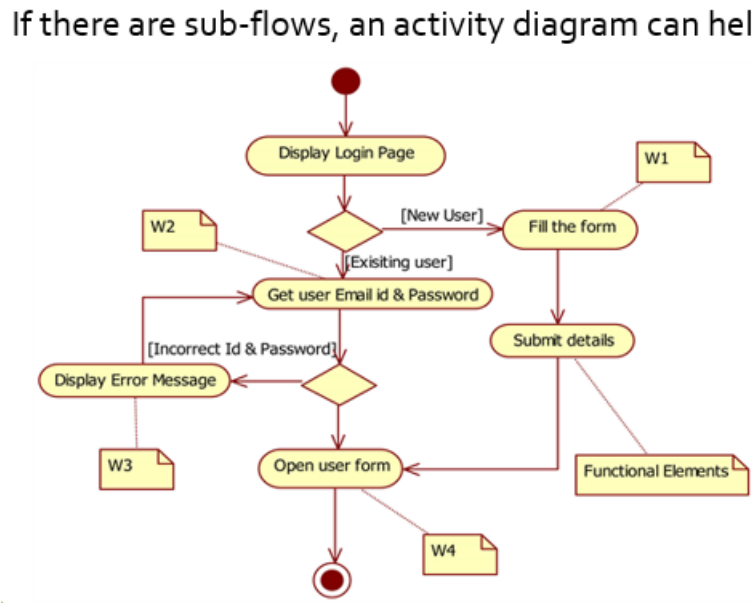
Durante la fase di **Requirements Engineering** è possibile (optional) produrre un prototipo dell'aspetto visivo (Visual Look prototype) dell'interfaccia utente da usare per la user validation.

Durante la **fase di Design**:

- si aggiorna e si arricchisce di dettaglio l'interfaccia utente (si producono i cosiddetti mock ups);
- viene definito il flusso di interazioni (Flow of Interactions) nell'interfaccia che comprende la sequenza di videate che l'utente attraversa per raggiungere un obiettivo, in caso ci siano sotto-flussi alternativi può essere di aiuto utilizzare gli Activity Diagrams.

> {Fra: "questa parte sul flusso di interazioni è più legata alla UX, user experience, che è diversa da UI ma non è una cosa approfondita nelle slide"}.
>
> Segue un esempio di "Flow of Interactions":





Modello di Norman per la User Interaction

Secondo il modello di Norman gli umani pensano seguendo fasi specifiche:

1. Formazione di un obiettivo;
2. Formazione di una intenzione;
3. Specificazione di un'azione;
4. Esecuzione dell'azione;
5. Percezione dello stato del sistema (feedback);
6. Interpretazione del feedback;
7. Valutazione.

Il feedback è critico per la comprensione dell'utente, un sistema con un bottone che non comporta nessun visibile cambiamento quando premuto genera esitazione e confusione nell'utente che infatti si aspetta un cambiamento ad ogni sua azione di "pressione" (di un tasto).

GOMS Model

Il Goals, Operators, Methods and Selection (GOMS) model è un modello classico per la User Interaction.

Quando si progetta una interfaccia utente bisogna:

- Sapere quali sono gli obiettivi (GOALS) dei propri utenti (use cases);
- Studiare le loro aspettative sulle azioni necessarie a raggiungere l'obiettivo;
- Fornire all'utente gli OPERATORI e METODI appropriati (elementi della UI) (i metodi sono la sequenza di operazioni di base necessarie per ottenere un goal);

- Definire le regole di SELEZIONE (quali metodi applicare per raggiungere un goal quando ne sono disponibili molti).

Caratteristiche di una buona UI

Una buona UI è la pietra fondante di una esperienza utente positiva, al fine di migliorare usabilità e soddisfazione degli utenti bisogna tenere in conto diversi aspetti.

Intuitività: gli elementi e le interazioni dell'interfaccia grafica devono essere naturalmente comprensibili e gli utenti devono essere in grado di usare l'interfaccia senza una preparazione estensiva.

Coerenza: bisogna mirare all'uniformità degli elementi e dei comportamenti dell'interfaccia, in questo modo gli utenti possono predire come il sistema risponderà ai loro input, riducendo così il carico cognitivo.

Efficienza: la UI deve ottimizzare il completamento dei task con lo sforzo minimo da parte dell'utente (non come i siti dell'amministrazione italiana). Una interfaccia "semplificata" e ricca di shortcuts aumenta l'efficienza.

Responsività: gli elementi dell'interfaccia devono reagire con prontezza agli input da parte dell'utente, ritardi e non responsività sono infatti causa di frustrazione.

Chiarezza e leggibilità: i contenuti e il testo devono essere chiari, leggibili e ben strutturati. Una buona scelta di tipografia e contrasto può aumentare la leggibilità.

Feedback: l'interfaccia deve fornire feedback informativi alle azioni dell'utente in modo da guidarlo e renderlo consapevole dello stato del sistema.

Gestione degli errori: i messaggi di errore permettono di guidare l'utente alla rettifica degli errroi senza frustrazione.

Personalizzabilità: le UI devono permettere di personalizzare la propria esperienza in modo da andare in contro alle necessità degli utenti.

Accessibilità: le UI sono progettate al fine di aiutare utenti con disabilità tramite ad esempio l'incorporazione di screen readers, navigazione da tastiera e "testo alternativo".

Estetica: l'esperienza utente è molto migliorata da una UI dal design e layout esteticamente piacevoli.

Scalabilità: una buona UI deve essere adattabile a varie dimensioni di schermi e risoluzioni, un design responsivo assicura usabilità coerente su diversi dispositivi.

BUONI ATTRIBUTI DEL DESIGN

Un Sistema Software dovrebbe essere facile da:

- Capire;
- Cambiare;
- Riutilizzare;
- Testare;
- Integrare;

- Programmare.

In pratica un sistema software dovrebbe essere semplice e un buon Design dovrebbe favorire ciò.

La "semplicità" può essere misurata da due metriche:

1. **Coesione** (che riguarda le caratteristiche intra-modulo);
2. **Accoppiamento** (che riguarda le caratteristiche inter-modulo).

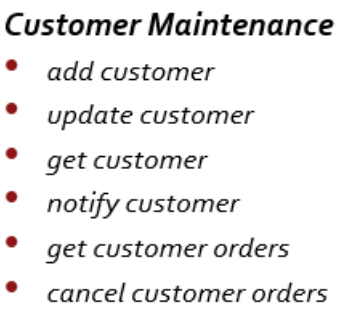
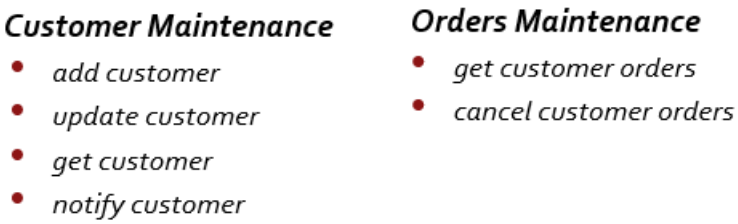
L'obiettivo è ottenere la Coesione più alta possibile e l'Accoppiamento più basso possibile.

1. **Coesione**

> 💡 La Coesione è legata a quanto le parti di un modulo dovrebbero essere nello stesso modulo, cioè quanto sono veramente in relazione tra loro.

Un modulo coeso dovrebbe essere "packaged together" (NON è CHIARO IL SIGNIFICATO DI QUESTA FRASE).

La rottura di un modulo coeso in parti più piccole dovrebbe richiedere, per ottenere risultati utili, un forte accoppiamento tra le parti ottenute attraverso "chiamate" tra questi nuovi moduli.

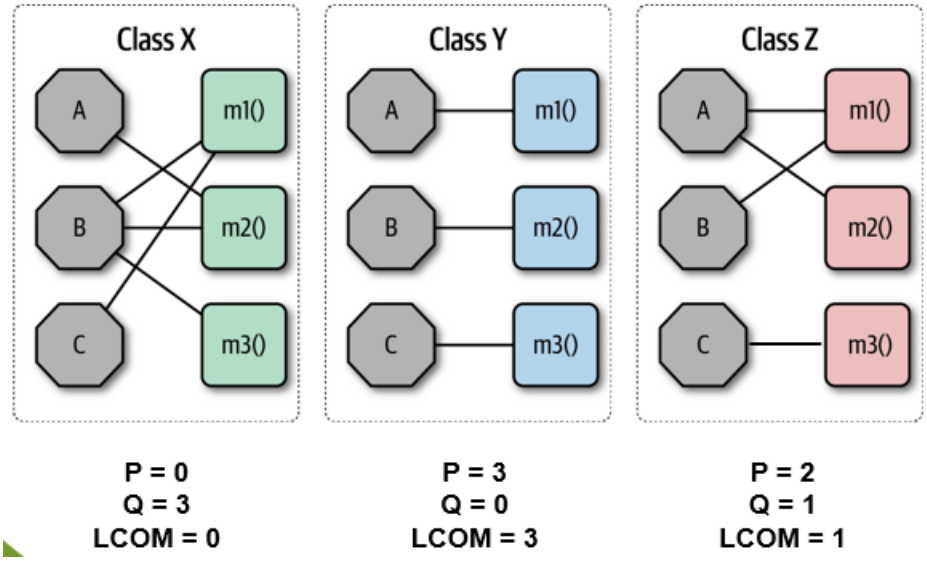
> Ex:
>
> 
>
>
> 
>
>
> Quale delle due partizioni garantisce la coesione migliore? Dipende, ad esempio se Order Maintenance richiede molta conoscenza sulle informazioni del Customer ha senso tenerli insieme, lo stesso vale se è noto che Order Maintenance ha le sole due operazioni mostrate in figura, se invece è noto che le due partizioni cresceranno può avere senso dividerle.

1.1 **Misurare la Coesione**

Al fine di misurare la coesione di una classe o di un modulo in linguaggio orientato agli oggetti si usa una metrica chiamata LCOM (Lack of Cohesion of Methods, mancanza della coesione dei metodi).

$$LCOM = max(P - Q, 0)$$

> 💡 **Q** è il numero totale di *paia di metodi* nella classe che accedono ad uno o più degli stessi attributi della classe.
>
> **P** è il numero totale di *paia di metodi* nella classe che non accedono a nessun attributo in comune.

> Ex:
>
> 
>
>
> La Classe X ha un punteggio LCOM basso il che è indice di una buona coesione strutturale.
>
> La Classe Y manca di coesione; ciascuna delle coppie campi/metodi potrebbe essere in una classe a sé senza effetti sul comportamento.
>
> La Classe Z ha una coesione mista, gli sviluppatori potrebbero fare refactoring e porre l'ultima combinazione campo/metodo in una classe a sé.

2. **Accoppiamento**

💡 L'accoppiamento è il grado di interdipendenza tra i moduli.

- Con un **accoppiamento basso o lasco** i moduli sono relativamente indipendenti l'uno dall'altro e interagiscono attraverso interfacce o astrazioni ben definiti; cambiamenti ad un modulo hanno impatto minimo sugli altri.
- Con un **accoppiamento alto o stretto** i moduli sono altamente dipendenti l'uno dall'altro, referenziano le parti interne l'uno dell'altro il che rende difficile fare cambiamenti ad un modulo senza causare ripercussioni sugli altri.

In generale le classi strettamente accoppiate sono difficili da capire (perché bisogna capire tutte le classi legate tra loro) e quindi anche difficile da modificare. Inoltre, sono difficili da riutilizzare se isolate (perché bisogna esportare anche tutte le altre classi accoppiate) e un errore in una può facilmente propagarsi in tutte le altre. Per questi motivazioni l'accoppiamento stretto è un tratto di Design non desiderabile.

2.1 **Misurare l'Accoppiamento**

Date due unità software x e y il "**pairwise coupling** " (trad: accoppiamento relativo alla coppia) relativo a x e y può essere calcolato come:

$$C(x, y) = I + N/(N + 1)$$

💡 **N** è il numero di relazioni di accoppiamento tra x e y.

I è il livello più stretto di accoppiamento tra x e y (Misurato da 1 a 5, approfondito di seguito).

💡 L'**accoppiamento globale del sistema** è il valore medio (o mediano, dipende da come si vuole tradurre) dell'accoppiamento di tutte le coppie.

Esistono 5 livelli di accoppiamento, dal peggiore al migliore sono:

- **Accoppiamento dei Contenuti** (*Livello 5*, Content Coupling)

  Due unità accedono l'una ai dati interni o alle informazioni procedurali dell'altra. Cambiamenti in un'unità spesso richiedono una review dell'altra il che porta ad un'alta probabilità di errori.

- **Accoppiamento Comune** (*Livello 4*, Common Coupling)

  Due unità fanno entrambe riferimento alla stessa variabile globale, questa può essere usata per gli scambi di informazioni e avere impatto sulla logica delle due unità.

- **Accoppiamento di Controllo** (*Livello 3*, Control Coupling)

  Una unità passa informazioni di controllo ad un'altra unità, direttamente influenzando la sua logica. Questo obbliga a considerare entrambi i moduli come una coppia.

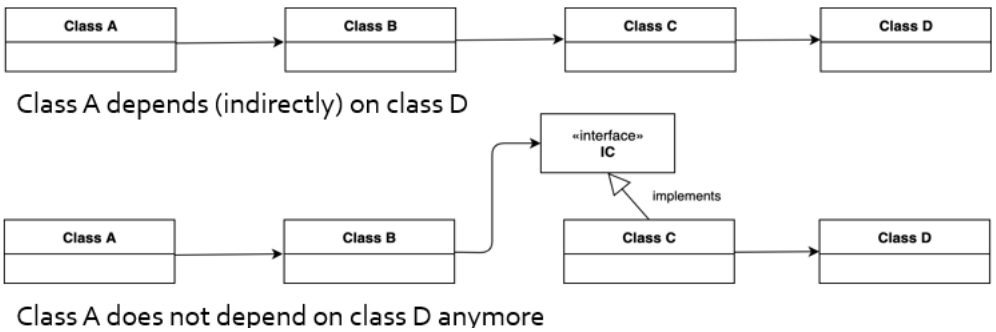
- **Accoppiamento Timbro** (*Livello 2*, Stamp Coupling)

Due unità software si scambiano più dati del necessario, il che rende più difficile comprendere l'interdipendenza tra queste.

- **Accoppiamento dei Dati** (*Livello 1*, Data Coupling)

  Vengono scambiati SOLO dati necessari, il che porta a delle unità accoppiate lascamente, questo è il migliore livello raggiungibile.

Si può ridurre l'accoppiamento con diverse tecniche, in OOP è molto utilizzata **l'introduzione di interfacce.**



2.2 **Legge di Demetra**

Secondo questa legge un oggetto dovrebbe inviare messaggi solo ai seguenti tipi di oggetti:

- Sé stesso;
- I propri attributi;
- I parametri dei metodi nell'oggetto;
- Ogni oggetto creato da un metodo nell'oggetto;
- Ogni oggetto restituito dalla chiamata a uno dei metodi dell'oggetto;
- Ogni oggetto in ogni collezione che sia una delle categorie sopra elencate.

La legge di Demetra può essere riassunta in "NON PARLARE CON GLI SCONOSCIUTI".