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

PRINCIPI GENERALI DEL BUON DESIGN

KISS: Keep It Simple, Stupid!

La semplicità dovrebbe sempre essere l'obiettivo chiave, infatti meno codice si scrive in meno tempo, ha meno bug ed è più facile da modificare.

> La Semplicità è la suprema sofisticazione. -Leonadro Da Vinci.
> La Perfezione è raggiunta non quando non c'è più nulla da aggiungere ma quando non c'è nulla più da togliere. -Antoine de Saint-Exupéry.

YAGNI: You Aren't Going to Need It (non ti servirà)

Bisognerebbe provare a non aggiungere funzionalità fino a che non sono necessarie. Tutto lo sforzo usato per una feature che serve domani e sforzo tolto alle features (funzioni) che devono essere sviluppate per la versione corrente.

Aggiungere codice prima che sia necessario porta ai cosiddetti code bloat (codici gonfiati), vale a dire che il software diventa sempre più grande e complicato inutilmente.

Fai la cosa più semplice che può funzionare

Una buona domanda da chiedersi quando si programma è "Qual è la cosa più semplice che potrebbe funzionare?", questo aiuta a mantenersi su un percorso di semplicità nello sviluppo.

Separazione delle preoccupazioni (Separation of Concerns)

Diverse aree di funzionalità dovrebbero essere gestite da moduli di codice distinti e sovrapposti il minimo possibile. Questo aiuta a semplificare lo sviluppo e la manutenzione del codice.

Quando le preoccupazioni sono ben separate le sezioni individuali possono essere riutilizzate, sviluppate e aggiornate in maniera indipendente.

DRY – Don't Repeat Yourself

Ogni funzionalità significative dovrebbe essere implementata in un unico punto del codice sorgente, appena ci si rende conto che ci si sta ripetendo bisognerebbe creare una nuova astrazione.

La Duplicazione (sia essa fatta intenzionalmente o inavvertitamente) può portare a incubi di manutenzione, "poor factoring" e contraddizioni logiche.

Quando funzionalità simili sono svolte da pezzi di codice distinti è generalmente benefico combinarle astraendo le varie parti.

Scrivere il Codice per il Manutentore

“Scrivi il codice come se la persona che lo manuterrà fosse un violento psicopatico che sa dove vivi”.

Ogni codice degno di essere scritto è degno di essere manutenuto.

Bisognerebbe scrivere il codice e aggiungere commenti in modo renderlo comprensibile anche un programmatore junior.

Bisognerebbe seguire il principio di minore stupore, vale a dire il lettore non dovrebbe sentirsi sorpreso leggendo il codice.

Evita Ottimizzazioni Premature

L’ottimizzazione deve iniziare dopo aver ottenuto un codice funzionante e con l’aiuto di dati empirici.

> “Prima fallo funzionare, poi rendilo giusto e poi alla fine rendilo veloce”.
> “L’ottimizzazione prematura è la radice di tutti i mali”.

Regola del Boy-Scout

Lascia il campo più pulito di quando lo hai trovato. I cambiamenti al codice tendono ad abbassarne la qualità, bisogna evitarlo e focalizzarsi proprio sulla qualità. Ogni volta che viene individuato un pezzo di codice poco chiaro bisognerebbe subito correggerlo.

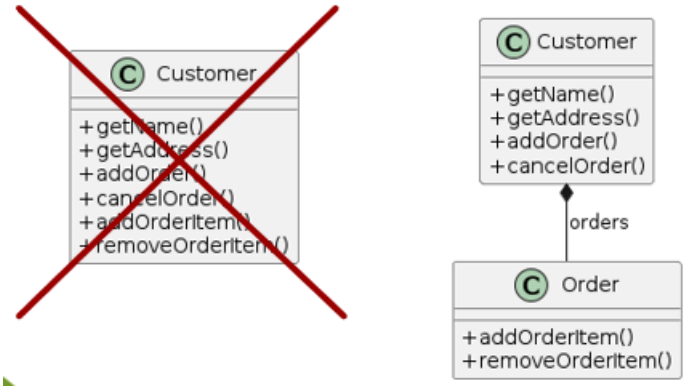
PRINCIPI INTRA-CLASSE

SOLID:

- **S**ingle Responsability Principle (principio della singola responsabilità);
- **O**pen-Closed Principle (principio aperto-chiuso);
- **L**iskov Substitution Principle (principio di sostituzione di Liskov);
- **I**nterface Segregation Principle (principio di segregazione dell’interfaccia);
- **D**ependency Inversion Principle (principio di inversione della dipendenza).

Single Responsability Principle

Un componente dovrebbe svolgere un unico task ben definito, in questo modo i cambiamenti a quel task coinvolgeranno solo un modulo o classe.

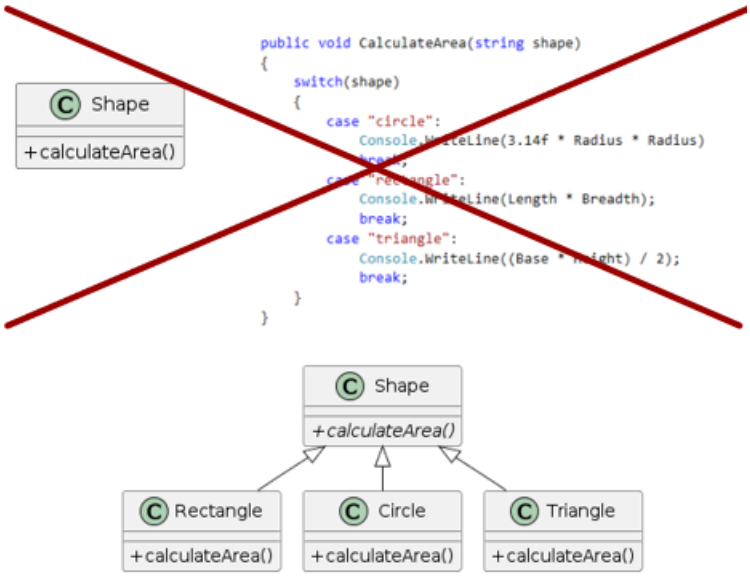


Open-Closed Principle

Le entità software (classi, moduli, funzioni ecc) dovrebbero essere **aperte per l'estensione ma chiuse per la modifica.**

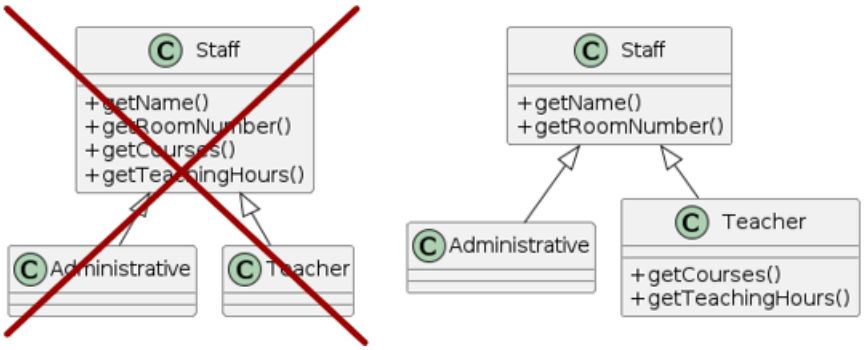
In altre parole le classi dovrebbe essere scritte in un modo che non invoglia a modificarle ma a **estenderle.**

Per ottenere questo risultato bisogna esporre solo le parti mobili che necessitano di cambiare e nascondere tutto il resto.



Liskov Substitution Principle

Gli oggetti dovrebbero essere **rimpiazzabili con istanze dei loro sottotipi** senza alterare la correttezza del programma.

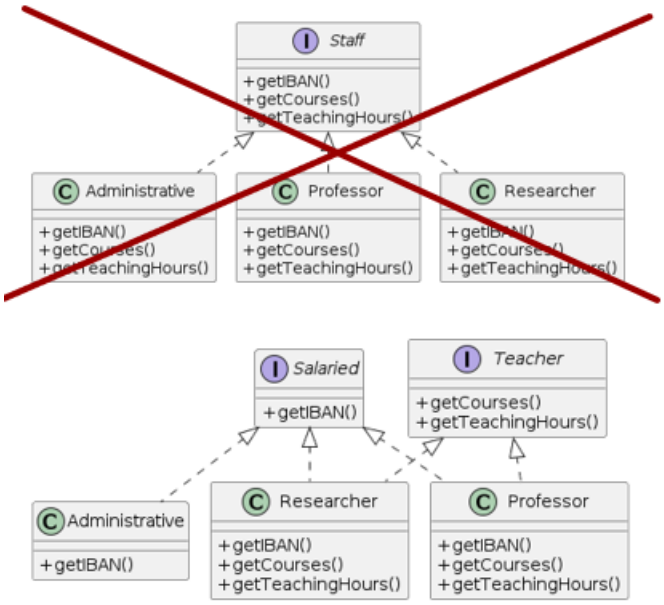


Nello schema a sinistra se sostituissimo Staff con Administrative, ci sarebbe incoerenza perché un amministrativo non ha *corsi* o *ore di insegnamento*. Nello schema a destra invece Staff può essere sostituito sia da Teacher sia da Administrative senza alcun problema perché entrambi hanno un *nome* e una *numero di stanza*.

Interface Segregation Principle

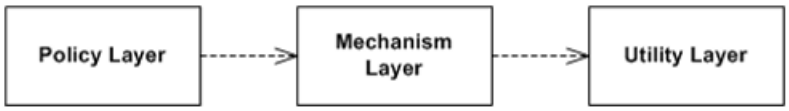
Un cliente non dovrebbe dipendere da metodi che non usa, di conseguenza è preferibile avere tante interfacce piccole e specifiche (che consistono di pochi metodi) piuttosto che tante ma grandi e generali. Questo permette al cliente di dipendere dal minimo numero di metodi possibile.

Un oggetto dovrebbe implementare tante interfacce, una per ogni ruolo che gioca in un diverso contesto.

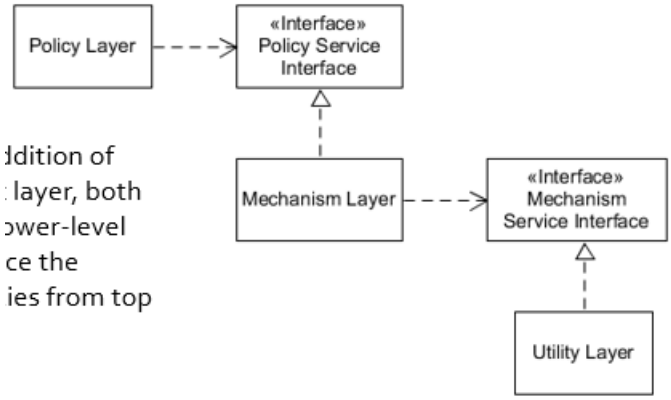


Dependency Inversion Principle

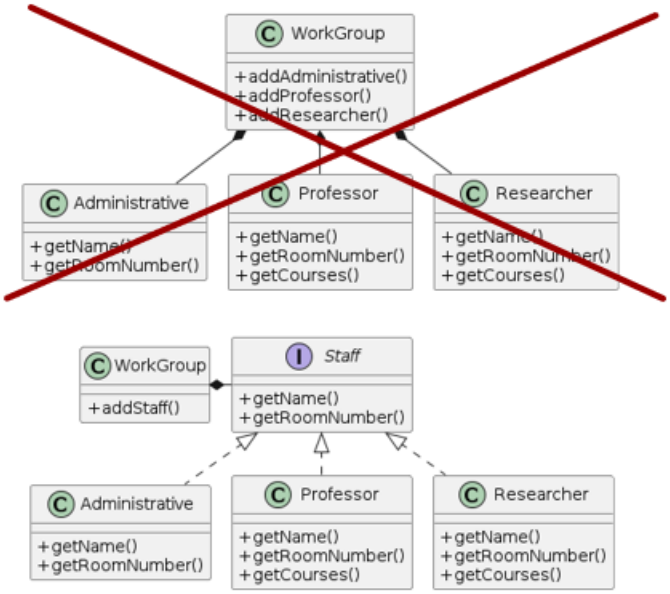
Nelle architetture convenzionali le componenti di basso livello sono progettate per essere “consumate” dalle componenti di alto livello il che favorisce la complessità del sistema. Con questa struttura le componenti di alto livello dipendono da quelle di basso livello per compiere un task e questo limita la loro riutilizzabilità.



💡 Il principio di inversione della dipendenza detta che i moduli di alto livello non debbano essere dipendenti da quelli di basso livello ma che debbano tutti invece essere dipendenti da delle astrazioni.



Con l'aggiunta di uno strato astratto, gli strati di alto e basso livello riducono entrambi le dipendenze dall'alto in basso.



rombo pieno COMPOSIZIONE e il simbolo sta vicino al composto. Un WorkGroup è composto da più Professor nel primo schema

Altri importanti principi

Bisogna aumentare il più possibile la coesione (per ragioni e con modalità spiegati nel documento 7.1) nascondere i dettagli implementativi in modo da consentire cambiamenti all'implementazione con un effetto minimo su altri moduli che usano il componente cambiato, si può ottenere ciò:

- **Minimizzando l'accessibilità** delle classi e dei membri;
- **Non esponendo** al pubblico i dati;
- **Evitando di integrare dettagli** implementativi nell'**interfaccia** della classe;
- **Diminuendo l'accoppiamento** in modo da nascondere meglio i dettagli.

È buona norma inoltre separare metodi Command e metodi Query, rendendoli anche riconoscibili con una convenzione sui nomi.

Ogni metodo dovrebbe essere o un Command (che esegue una azione) o una Query (che restituisce dati al chiamante) ma mai entrambi. Si nota che l'ordine di chiamata di dei metodi query è irrilevante visto che questi non modificano lo stato.

PRINCIPI INTER-CLASSI

Minimizzare l'accoppiamento

Minimizzare l'accoppiamento come detto nel documento 7.1.

Ortogonalità

Cose che non sono concettualmente legate non dovrebbero essere legate nel sistema. Questo principio è associato alla **semplicità** e **rende più semplice l'apprendimento, la lettura e la scrittura** di programmi in un linguaggio di programmazione.

Più è ortogonale il design meno eccezioni ci sono.

Il significato di una feature ortogonale è indipendente dal contesto.

Legge di Demetra

Applicare la Legge di Demetra come detto nel documento 7.1.

Favorire la Composizione all'Ereditarietà

L'ereditarietà è una forma di accoppiamento più forte della Composizione e può portare a violazioni del principio di sostituzione di Liskov (poiché le sottoclassi fanno facilmente delle assunzioni (?)).

La Composizione implica meno accoppiamento e quindi è preferibile. In definitiva bisogna usare la **Composizione se c'è una relazione di tipo "has a" o "uses a" e relegare l'ereditarietà a relazioni "is a".**

Principio di Robustezza

Bisogna essere conservatori in ciò che si fa e liberali in merito a ciò che fanno gli altri.

Servizi in collaborazione che dipendono dalle rispettive interfacce dovrebbero inviare comandi e/o dati che aderiscono il più possibile alle specifiche ma dovrebbero accettare anche input non conformi fintanto che sono comprensibili.

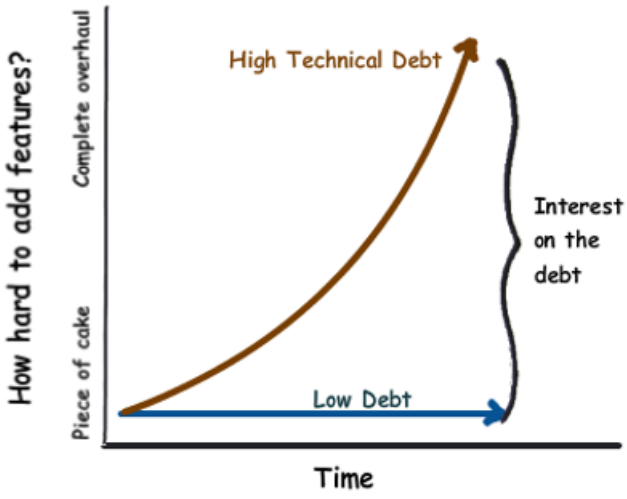
DEBITO TECNICO

Codice Pulito / Design Pulito

Il codice pulito è:

- **Leggibile**: scritto in modo conciso, chiaro e con nomi di variabili e funzioni descrittivi;
- **Manutenibile**: il codice deve essere facile da modificare e manutenere nel tempo;
- **Testabile**: il codice deve essere progettato per essere testato automaticamente;
- **Riusabile**: il codice deve essere sviluppato per essere riusato in altre parti del progetto o in progetti futuri.

Esattamente come contrarre un debito facilita gli acquisti oggi ma li rende più difficili se il debito si accumula scrivere Codice Sporco permette di scrivere e rilasciare codice più rapidamente ma al costo di renderlo più difficile da capire e manutenere.



💡 Questo è detto Debito Tecnico, il processo diventa più veloce temporaneamente ma poi rallenta mano a mano che il debito si accumula.

Esistono diversi fattori che peggiorano il debito tecnico

Non conformità ai principi del “buon design”

- ogni cambiamento ad una parte del progetto ha effetto sulle altre parti;
- è difficile isolare il lavoro di un singolo membro di un team.

Mancanza di testing

- la mancanza di feedback immediato introduce errori che possono essere scoperti dopo molto tempo.

Mancanza di documentazione

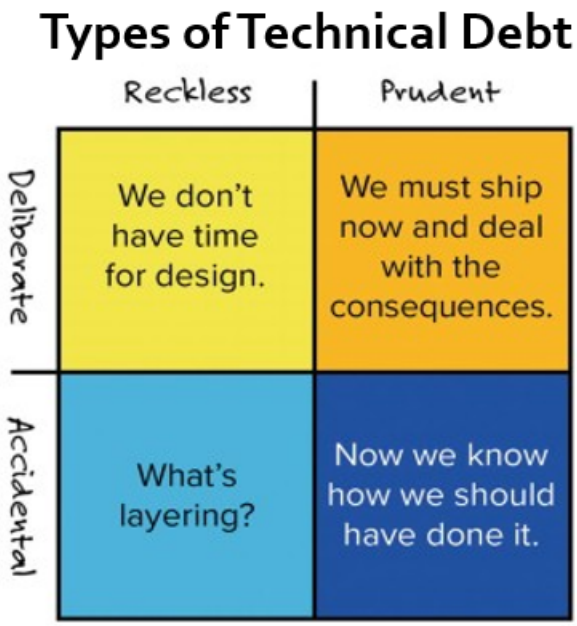
- rallenta l'integrazione di nuovi sviluppatori al progetto;
- può arrestare lo sviluppo se delle persone chiave lasciano il progetto.

Mancanza di integrazione tra i membri del team

- se la conoscenza non è condivisa le persone finiscono per lavorare basandosi su informazioni vecchie;
- i nuovi sviluppatori sono addestrati male dai loro mentori.

Sviluppo in parallelo

- può portare all'accumulazione di debito tecnico su più fronti;
- il debito esplode durante la fase di integrazione.



Effetti del debito tecnico:

- Deterioramento delle **performance** del software;
- Aumento della **difficoltà di manutenzione e miglioramento** del software;
- **Aumento dei costi** a lungo termine;
- **Riduzione della qualità** del software;
- **Soddisfazione dei clienti** e stakeholders più bassa.

Refactoring

Il Refactoring può essere utile per mettere una pezza ai problemi introdotti dal debito tecnico.

> 💡 Il **Refactoring** è un processo sistematico di miglioramento del codice senza aggiunta di nuove features allo scopo di ottenere codice pulito con un design semplice.

Le tecniche di Refactoring descrivono i passi necessari a risolvere uno specifico problema con il codice.

> 💡 I **Code Smells** (puzze del codice) sono indicatori di problemi che possono essere risolti tramite il refactoring, rivelano delle debolezze del design che riducono la qualità.

È consigliato consultare il sito: refactoring.guru