Progettare per preservare (riepilogo e usi dei Design Pattern)

Slides che non erano caricate

> *I design pattern servono per ragionare bene per la progettazione di un sistema software.*

Per orientarsi su quale pattern vedere e utilizzare, prima di valutarli singolarmente, si possono **considerare le seguenti possibili esigenze**:

- **Non ci si vuole legare alle singole istanze** ma ad un'**interfaccia più generica.** Serve più **flessibilità** quando si usa una classe. Allora si usa ***Factory Method e Prototype***
- SI vuole usare il software per il futuro e deve essere pronto per prossimi aggiornamenti- Non si sa chi è il destinatario della nostra chiamata. quindi non si legge il richiedente con un particolare ricevente. Quando si modifica il software, si ha l'esigenza di **adeguare il software ad ogni tipo di ricevente**. Allora si usa ***Chain of Responsability e Command***
- Il software, in genere, dura molti anni. In tutto questo tempo **cambiano le caratteristiche della macchina** sulla quale questo software "gira" (cambiano le librerie, ***tutto si evolve***). Non bisogna legarsi fortemente alle librerie e alle piattaforme hardware e quindi per permettere future modifiche. Se nel software si capisce che, nel corso del tempo, sarà utile utilizzare nuove librerie, allora vuol dire che bisogna fare adattamenti. Si possono usare ***Factory Method (ma anche Abstract Factory), Bridge*** permettono il cambiamento di librerie e adattamenti alle nuove tecnologie
- Si ha una **dipendenza di implementazione**, quindi c'è un algoritmo che sa prendere i dati dalle varie parti e li sa elaborare. Se si pensa che ci saranno **varianti di questo algoritmo** allora non bisogna legarsi troppo ad un'implementazione. Si può usare il ***Factory Method*** cambiando all'interno dei *ConcreteProduct* le varie implementazioni. Oppure ***Bridge***, *Memento, Proxy*.
- Una classe, in un determinato momento, sa quello che deve fare (***in base al suo stato***). L'algoritmo con cui operare è diverso in vari momenti del software in base al suo stato. In questo caso si può usare ***State***. Simile a State esiste ***Strategy*** e, in particolare:
  - E' come uno *State*. I ruoli (nomi) sono molto simili allo State e le relazioni fra le classi sono identiche. Questi due design pattern sono identici e i ruoli sono mappati allo stesso modo.
  - **Per State**: si ha un sistema che attraversa vari stati.
  - **Per Strategy:** si ha un sistema che vuole applicare diversi algoritmi, a seconda di un'impostazione che c'è in un determinato momento. Questi diversi algoritmi li implemento sui *ConcreteStrategy* (analogamente ai *ConcreteState* di State)
  - Si può anche ***usare Template Method*** (che usa *dispatch*). Si diminuiscono le dipendenze con l'algoritmo
  - Altri Design Pattern sono: Iterator (scorre in maniera flessibile liste di dati), Visitor (Non serve spesso ma solo quando si vuole eseguire una navigazione del codice già scritto)
- Nel progetto si ha uno **stretto accoppiamento** che incrementa la probabilità che una classe sia riusata da sola e che il sistema sia compreso, modificato ed esteso più facilmente. Si incapsulano vari algoritmi in un macro sistema in sottoclassi. ***Si usa il Façade*** per **disaccoppiare le singole classi dai chiamanti**. Alternativamente si possono usare ***Factory, Bridge, Chain, Command, Mediator, Observer*** che eseguono questo **disaccoppiamento fondamentale**.
- Sebbene l'**ereditarietà permette di riusare codice già scritto (ed estenderlo), non si deve abusare**. Se nel sistema si hanno **gerarchie molto profonde** (+10 livelli classi), allora i livelli più

profondi sono troppo complessi perché contengono tutti i metodi delle classi che stanno al livello superiore. Testare questo sistema vuol dire valutare tutti i possibili classi. Si può riusare il codice tramite **composizione**: all'interno di una classe si crea un'**istanza di un'altra classe e determinate chiamate verranno rimandate a tale istanza**. L'interfaccia sulla quale si chiamano i **metodi rimandati, è molto più specifica, ristretta e limitata**.

- E' necessario **aggiungere funzionalità senza essere abilitati a modificare** un determinato codice (quindi **non si hanno i permessi sul codice**). Si può usare ***Adapter (cambia l'interfaccia) e Decorator (non cambia l'interfaccia)***

Metriche su evoluzione dei software

Dopo il completamento del prodotto viene fornito al cliente in modo definitivo. In questo caso vuol dire che è stato testato ed è pronto per la consegna. La consegna definisce il **completamento del processo di sviluppo**

- Dopo questo momento intervengono ulteriori notifiche per vari motivi: per esempio -> i programmatori vogliono fare delle correzioni prima che il cliente lo dica

Tutte le modifiche che seguono la consegna del software fanno parte della procedura di ***MANUTENZIONE***.

> ***Manutenzione** ed **evoluzione** sono spesso equivalenti. Se si vuole essere precisi: manutenzione (modifica del softare) ed evoluzione(nuovo software, rilascio di patch, rimozione sistema)*

Mediamente i costi di manutenzione sono abbastanza alti perchè potrebbe potenzialmente durare molto di più rispetto allo sviluppo. Ci sono delle **categorie per cambiamenti di evoluzione** del software:

- **Correttivo** (*statisticamente 17% fra tutti i cambiamenti*) = **Rimozione di difetti**. Vengono rimossi il prima possibile per evitare problemi futuri
- **Adattivi** (*18% fra tutti i cambiamenti*) = **cambiamento dovuto per farlo girare su un'altra piattaforma software**. Può essere eseguito su un nuovo SO, usa una nuova libreria ecc...
- Perfettivi (*60% fra tutti i cambiamenti*) = Ogni volta che cambio il software per **aggiungere funzionalità migliorandole** oppure **miglioro le prestazioni**. Se il software è utile al cliente, egli vorrà farci sempre più cose e, quindi, richiede sempre nuove funzionalità. Quindi il software ha *vita lunga*
- Preventivi (*5% fra tutti i cambiamenti*) = modifica internamente la struttura (refactoring) senza incidere sulle funzionalità e correttezza di esecuzione. Prima della consegna alcune cose potevano essere progettate meglio (*magari serviva qualche design pattern non utilizzato*). Queste modifiche **ci si prepara per poter intervenire in maniera più semplice** in futuro sul software.

> *E' buona pratica anticipare i cambiamenti a design time (tramite parametrizzazione, incapsulamento, etc.)*

Dinamiche di evoluzione

Lehman e Belady studiavano i comportanti dei sistemi software:
"I sistemi materiali sono sottoposti a leggi fisiche ma i sistemi software no. Ci sono delle leggi che possono regolare i sistemi software e che fanno vedere le leggi soddisfatte da molti sistemi software? "

Vi sono delle **leggi** (*che sono state dimostrate*) che si sono rilevate veritiere per i sistemi software. Queste leggi riguardano sistemi software di ***GRANDI DIMENSIONI*** sviluppati da **grandi aziende**.

Leggi di Lehman

- **Cambiamento continuo**: I sistemi hanno bisogno di essere continuamente adattati altrimenti diventano progressivamente meno soddisfacenti
  - Se un software è ritenuto di successo, allora deve **continuamente cambiare** perché altrimenti sarà sempre meno soddisfacente.
- **Aumento della complessità**: Quando un sistema evolve, la sua struttura aumenta di complessità, a meno che del lavoro viene fatto per preservare o semplificare la sua struttura
  - Se **NON si riflette profondamente sulla struttura del sistema software**, si elimina la struttura iniziale del sistema, quindi **si degrada**
- **Auto-regolazione**: Attributi come dimensione, intervallo tra release e numero di errori trovati in ciascuna release sono approssimativamente invarianti
  - Se si misura la dimensione del software e il tempo necessario fra una release importante e un'altra anch'essa importante, si vede che gli errori fra le release sono circa uguali. Nonostante si facciano delle modifiche, certi parametri rimangono costanti nel tempo. La dimensione rimane circa uguali perché alcune parti vengono rimosse e, spesso, le aggiunte non sono di grandissime dimensione.
  - Gli errori sono sempre circa costanti fra le release. Le scelte sul design e progettazione sono quelle fatte all'inizio. La capacità di un programmatore di scrivere codice e progettare rimane sempre basata sui requisiti iniziali e per questo gli errori possibili saranno pressoché uguali
- **Stabilità organizzativa**: Durante la vita di un sistema il suo tasso di sviluppo è circa costante e indipendente dalle risorse impiegate per lo sviluppo
  - si legga alla terza legge ma si focalizza sul tasso di sviluppo: "*Quante modifiche si possono fare per fornire una nuova release?* ". Se si ha una nuova versione e si devono aggiungere delle funzionalità, esse saranno limitate nel range di funzionalità da poter sviluppare per release. Occorre un certo tempo per studiare un certo software indipendentemente dal numero di persone assegnata al team di sviluppo

A livello di Coordinazione

> *Più persone si aggiungono al team, più si rallenta lo sviluppo (si devono coordinare, si devono fare domande per le scelte fatte, devono studiare per un bel po' di tempo il software* GIA' SVILUPPATO*).*

Applicabilità delle leggi

- Sono in generale applicabili a grandi sistemi sviluppati da grandi organizzazioni
- Non è chiaro come si adattano a
  - Piccoli prodotti
  - Prodotti che incorporano un certo numero di COTS
  - Piccole organizzazioni

Costo di manutenzione

Occorre un **grande tempo per lo studio del software** prima di eseguire modifiche.

- Spesso gli sviluppatori della prima versione non sono mai gli stessi di una successiva versione. Man mano gli sviluppatori si allontanano dalla scrittura del codice perché, man mano che si diventa più bravi, si sale di livello e si diventa sempre più importanti fino a diventare coordinatori e progettisti.
- Il software **si degrada nel tempo** e quindi **apportare modifiche diventa sempre più dispendioso** di tempo e quindi richiede **più soldi**

- Gli sviluppatori che hanno scritto il software inizialmente erano spinti a produrre in poco tempo la prima release. Ne segue che chi scrive il codice, non è incentivato a scrivere codice che DURI NEL TEMPO proprio perché si **vuole consegnare il prima possibile**. Questa **mancanza**, si ripercuote su una cattiva versione del codice e quindi su **un maggior costo futuro di manutenzione**

Modelli di manutenzione

Ci sono 3 modi:

- **Quick-fix**: Si ha poco tempo per aggiustare. La visione è relativa al codice scritto senza avere visione della documentazione o dei test quindi **non esiste un'analisi del sistema**
- **Miglioramento iterativo**: Cambiamenti fatti in base ad un'**analisi del sistema esistente**. Si fa un **controllo sulla complessità** e sul **mantenimento del design**.
- **Riuso**: diventa impossibile modificare il software e quindi si riusano delle componenti. Si prendono parti dalla versione precedente. Stabilire i requisiti del nuovo sistema riusando il più possibile

Tipi di modifiche

- **Re-factoring o re-structuring**: Processo di cambiamento del software che non altera il comportamento del codice ma migliora la struttura interna. Ovvero: prendere un sistema fatto male e modificarlo per ottenere una struttura ben fatta
- **Reverse engineering**: Analizzare un sistema per estrarre informazioni sul suo comportamento o sulla sua struttura
  - Capire dal codice disponibile cosa è stato fatto nel sorgente
- **Re-engineering**: Alterare un sistema per ricostituirlo in un'altra forma. Cambiare totalmente il design e le varie interazioni fra le parti del sistema

↳ Prendiamo un codice e lo Adattiamo a richiesta del Cliente

Metriche

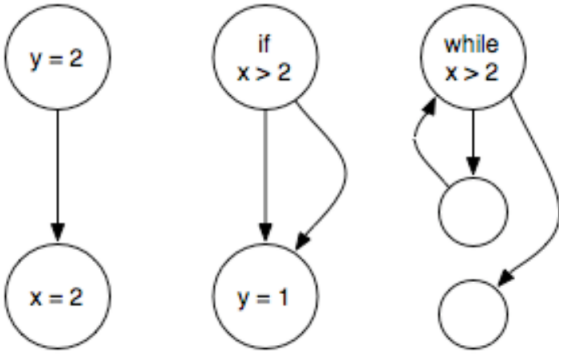
Si possono fare delle misurazioni sul software in sviluppo o già sviluppato per analizzare se si sta procedendo nella maniera corretta o no.
La metrica è solo un'indicazione con determinati limiti e a volte è utile avere uno strumento tale a disposizione.

Si distinguono:

- ***METRICHE TRADIZIONALI***-> nate prima del sistema ad oggetti (quindi specificate per tale campo) ma si applicano anche per sistemi non a oggetti

Metriche tradizionali

- **LOC** (*lines of code*) -> indica il **numero di linee** scritte nel codice sorgente. Si può misurare aprendo il codice e contando le linee. Si contano le linee vuote, le parentesi isolate su una singola riga ecc....Si conta tutto.
- **NCNB** (*Non comment - non blank*) -> si conta tutto tranne commenti o linee vuote
- Si potrebbero anche contare i punti e virgola (`;`) che separano le istruzioni
- **Complessità ciclomatica** (*cc*): si applica ad un algoritmo (non ad un oggetto). Si può rapprensetare l'algoritmo tramite un grafo dove:
  - linea di codice/istruzione = nodo del grafo
  - flusso fra le linee = arco del grafo



- Per calcolare cc si contano nodi/archi del grafo
- $cc = archi - nodi + 2$
- **Per istruzioni sequenziali** -> 2 nodi e 1 arco -> cc = **numero di test** = 1
- **Per istruzioni condizionali** -> 2 archi e 2 nodi -> cc = **numero di test** = 2
- **Per istruzioni iterative** -> 3 archi e 3 nodi -> cc = **numero di test** = 2
- In generale, ***più è alto cc, più condizioni nidificate ci sono e quindi più è complesso il codice (anche nella sua comprensione)***
- Se si vuole percorrere almeno una volta ciascun ramo del grafo, bisogna leggere proprio `cc` e questa "variabile" viene detta "*numero di test*" e ***serve per capire quanti test bisogna fare per testare un determinato codice***

Metriche CK (Chidamber e Kemerer)

Sono metriche per sistemi ad oggetti. I numeri che ci danno queste metriche sono testate a fondo e quindi sono molto affidabili. ***Ogni classe avrà un singolo valori fra le misure che seguono***.

> *VALORI ALTI=maggiora complessità -> **TRANNE NOC**=minor complessità*

Ci sono 6 modi per fare le misure:

Pesiamo i metodi
Gli Attribuiamo un peso.

- **WMC** (*Weighted Methods per Class*): per ogni classe si può ricavare la somma pesata dei metodi delle classi. Si sceglie un peso adatto per la misura e si vede quanto quella classe "pesa" e quanto è importante all'interno del sistema. **Non vengono considerate le superclassi**. Un peso si attribuitsce:
  - **peso = costante** al metodo: WMC sarà, quindi, il numero di metodi della classe
  - **peso = LOC** del metodo -> *i metodi più lunghi sono quelli più complessi*
  - **peso = complessità ciclomatica** per ogni metodo
  - **NOTE WMC**: se è alto, è **difficile riusarla** e quindi molto più **complessa da comprendere**.
- DIT (*Depth of Inheritance Tree*): profondità (***numeri di livelli***) albero della gerarchia. Alla radice della gerarchia si dà un valore di default (anche 0). Più è profonda la gerarchia, più è complessa quella classe.
  - **NOTE NOC**: se il valore è alto vuol dire che la complessità è alta e i test si fanno troppo in profondità

Valutiamo solo i figli del livello sottostante

- NOC (*(Number of Children of a Class*): conta il **numero di figli che si trovano immediatamente sotto** per una singola classe.
  - **NOTE NOC**: Possibile riuso se NOC è alto
- CBO (*Coupling Between Object Classes*): dà l'accoppiamento fra le classi. Si devono contare le associazioni fra le classi (quindi le interazioni) e non l'ereditarietà.

    - Es: `A -> B` `A -> C` `B -> D`: $CBO(A) = 2$ e $CBO(B) = 1$
    - **NOTE CBO**: Più è alto e più il sistema interagisce con altre classi. Quindi è molto più complesso
- RFC *(Response for a Class)* : quante **interazioni vengono fatte quando una classe riceve un'invocazione**. Quanti metodi diversi sono chiamati all'interno di un unico metodo. A tale numero viene sommato il numero totale di metodi. In altre parole: quanti messaggi distinti può inoltrare una classe (RFC). Più chiamate fa una classe, più è accoppiata con altre classi.
    - **NOTE RFC**: Più è alto, allora ci sono molte interazioni. Implica più test e più complessità
- LCOM (*Lack of Cohesion of Methods*): mancanza di coesione fra i metodi di una classe. Coesione = singola responsabilità per i metodi -> ogni metodo svolge un unico piccolo compito. ***Idealmente si deve avere: alta coesione e basso accoppiamento fra le classi***.
    - $$LCOM = 1 - \frac{\sum_i^a m_{A_i}}{m * a}$$
    - a=numero di campi, m=numero di metodi, m_{A_i}=numero di metodi che usano il campo A_i
    - ***Per CAMPO si intende una qualsiasi variabile/attributo.*** Il metodo 1 usa la variabile? E il metodo 2? ecc...
    - Esempio: *con 3 campi e 3 metodi*:
        - *ciascun metodo usa 3 campi*, allora $(3 + 3 + 3)/3 * 3) = 1$
        - *ciascun metodo usa 2 campi*, allora $(2 + 2 + 2)/3 * 3) = 2/3$
        - *ciascun metodo usa un campo*, allora $(1 + 1 + 1)/3 * 3) = 1/3$
        - *un solo metodo usa 3 campi*, allora $(1 + 1 + 1)/3 * 3) = 1/3$
    - **NOTE LCOM**: LCOM è basso -> coesione alta -> buona progettazione -> minor complessità. *Una classe si deve dividere se devo riusarla per altri scopi.*

Test

Un software è corretto perché, leggo i requisiti, progetto i test sulla base dei requisiti, valuto il risultato calcolato ed eseguo il software: se i risultati coincidono, allora il software è corretto

- Verificare tramite test serve per **CONVALIDARE il sistema software**.

Si distingue:

- **Verifica**: **Software inerente alle specifiche** (test approvati)
- **Validazione**: **Convalidare software**: il software è corretto. *Ma è anche utile?* Il cliente darà un feedback sul software prodotto. *"Il software fa quello che deve fare ma non mi soddisfa"*. Quindi c'è stata una mancanza di specifiche all'inizio.

Processo V & V (Verifica e Validazione)

Bisogna verificare che non ci siano contraddizioni e che il cliente sia soddisfatto (convalida).
La progettazione può avere un processo V&V: l'architettura , gli algoritmi scelti sono quelli voluti.

- V&V sul codice si fa tramite test. Si deve verificare se ci sono problemi a livello di codice: magari uso costrutti che generano risultati non voluti. Magari ci sono errori di scrittura nel codice. Il tipo di verifica che si fa (test) fa emergere situazioni di questo tipo che potrebbero essere sparsi su più punti del codice. Ne segue che potrebbero ***nascere certi problemi che prima ci erano sfuggiti***

Fase di Test: scrittura, esecuzione, analisi del risultato del test nei modi seguenti:

- **debug** se i risultati **NON** sono quelli aspettati
- **andare avanti** se i risultati sono quelli aspettati

> *Se il test esercita piccole parti di codice, allora risulta più semplice individuare l'errore*

Test: definizioni

I **dati dei test sono dati in input** (servono anche per portare ad uno certo stato determinate istanze/variabili) e vi sono degli **output stimati** (che serve per controllo del risultato dei test)

- **Caso di test** (*Test case*) è l'insieme dell'eseguibile, output stimato e dati in input
- **Test suite** = Insieme dei test case