**INGEGNERIA DEL SOFTWARE**

* **CARATTERISTICHE DEL SOFTWARE**
* **QUALITÀ’ DEL SOFTWARE**
* **SPAGHETTI CODE**
* **PARADIGMA COMMAND AND QUERY**
* **MIGLIORAMENTI SUL CODICE**
* **PUNTI DA RICORDARE: Test Driven Development, principio di singola responsabilità**
* **REFACTORING** (Processo che modifica la struttura del codice migliorandola senza alterarne il comportamento)
  + **PERCHÉ’ FARE REFACTORING**
  + **ESTRAI METODO**
  + **SOSTITUISCI TEMP CON QUERY**
  + **DIVIDI VARIABILE TEMP**
* **DESIGN PATTERN** (Creazionali, Strutturali, Comportamentali)
  + **DESCRIZIONE** (i DP sono strutture software per un piccolo numero di classi che descrivono soluzioni di successo per problemi ricorrenti. essi permettono ai principianti di lavorare come esperti e agli esperti di avere un linguaggio comune).
  + **SINGLETON** (Assicurarsi che una class abbia una sola istanza e fornire un punto di accesso globale all’istanza)
  + **FACTORY METHOD** (Definire un’interfaccia per creare un oggetto, ma lasciare che le sottoclassi decidano quale classe istanziare. Questo DP permette ad una classe di rimandare l’istanziazione alle sottoclassi)
  + **DEPENDENCY INJECTION** (Factory Method può essere usato per inserire le dipendenze necessarie tramite il costruttore, in modo da rendere il codice più semplice)
  + **OBJECT POOL** (Un object pool è un deposito di istanze già create, un’istanza sarà estratta dal pool quando un client ne farà richiesta)
  + **ADAPTER** (Converte l’interfaccia di una classe in un’altra interfaccia che i client si aspettano. Adapter permette ad alcune classi di interagire, eliminando il problema di interfacce incompatibili)
  + **FACADE** (Fornire un’interfaccia unificata al posto di un insieme di interfacce in un sottosistema. Definire un’interfaccia di alto livello (**semplificata**) che rende il sottosistema più facile da usare
  + **STATE** (Permettere ad un oggetto di alterare il suo comportamento quando il suo stato interno cambia. Far sembrare che l’oggetto abbia cambiato la sua classe)
  + **OBSERVER** (Definire una dipendenza uno a molti fra oggetti, così che quando un oggetto cambia stato tutti i suoi oggetti dipendenti sono notificati e aggiornati automaticamente. Conosciuto anche come **Publish-Subscribe**)
  + **MVC** (Model View control, è considerato un design pattern architetturale per le applicazioni interattive, esso permette di cambiare facilmente le interfacce utente)
* **PROCESSI DI SVILUPPO DEL SOFTWARE**
  + **ANALISI DEI REQUISITI**
    - **fattibilità**
    - **specifiche**
    - **requisiti funzionali**
  + **PROGETTAZIONE E IMPLEMENTAZIONE**
    - l’implementazione non è altro che la **traduzione da modello a codice**
  + **TESTING (V & V)** questa fase mira a dimostrare che il software è conforme alle specifiche del cliente. Le due V stanno a indicare **Verifica (**il sw è conforme alle sue specifiche?) e **Validazione** (il sw fa quello che l’utente ha realmente richiesto?). Esistono diverse tipologie di test tra le quali:
    - **UNIT TEST:** testano le singole componenti indipendentemente e sono effettuate dallo sviluppatore
    - **TEST DI SISTEMA**: viene testato l’intero sistema
    - **TEST DI ACCETTAZIONE:** sono dei test effettuati dagli sviluppatori tramite dati forniti dal cliente, forniscono la versione **alpha**
    - **TEST BETA:** sono effettuati dal cliente sul prodotto quasi completo
    - **TEST DI INTEGRAZIONE:** servono a introdurre parti di progetto a quello già esistente e a localizzare i bug presenti sul codice. Possono essere effettuati seguendo due approcci:
      * **TOP-DOWN:** vengono prima integrati i sottosistemi di alto livello e poi quelli di basso livello, essi permettono di scoprire errori nella struttura e di presentare versioni demo.
      * **BOTTOM-UP:** vengono prima integrati i sottosistemi di basso livello e poi quelli di alto livello, questa pratica permette una scrittura di test più semplice
    - **TEST REGRESSIVI:** vengono avviati quando viene scoperto un difetto, in quel momento si cerca di creare un test che lo rilevi e lo localizzi.

Esistono varie strategie di test e principalmente si dividono in due categorie:

* **BLACKBOX:** I test vengono effettuati senza conoscere l’implementazione del sistema ma solo partendo dalla documentazione prodotta.
* **WHITEBOX:** I test sono effettuati da persone che conoscono l'implementazione e si basano sul testare la struttura interna.

I test possono essere eseguiti sia **manualmente** che **automaticamente**.

* **COPERTURA DEL CODICE**: mi dice quando è il momento di finire la fase di testing, questo dato è dato da diverse **metriche**, per trovare la copertura di un codice basta dividere il programma in unità, definirne la copertura e dividere il numero di **unità eseguite** per il numero di **unità totali**.

Il linguaggio Java offre diversi Framework per la scrittura di test e derivati, tra i più utilizzati troviamo:

* **JUNIT**: questo framework garantisce l’indipendenza tra i test e usa asserzioni per automatizzare il controllo dei risultati dei test, inoltre, ciascun test è inteso come un metodo a se stante dunque facile da interpretare e scrivere.
* **ASSERTJ**: questo framework convive con JUNIT e permette di fare la concatenazione di asserzioni e permette di creare asserzioni in stile **fluent.**
* **JACOCO**: E’ un framework per verificare la copertura del codice, infatti la qualità del codice della test suite è misurabile tramite la code coverage, esso fornisce diversi parametri come la percentuali di classi, metodi, linee di codice e istruzioni coperte.

**TEST DRIVEN DEVELOPMENT**

E’ una strategia di sviluppo agile in cui i test vengono scritti prima del codice, in essa si alternano tre fasi:

- **RED**: scrivere il test

**-GREEN**: scrivere il codice per superare test senza far fallire quelli precedenti

**-REFACTOR**: migliorare il codice che ha superato i test.

Ovviamente per utilizzare questa strategia bisogna avere molta esperienza.

**TEST DOUBLE**

Per effettuare test a volte è necessario simulare il comportamento delle componenti da cui il metodo testato dipende, queste operazioni sono effettuate principalmente dai test double di tipo:

* **STUB**: Permettono di definire metodi in fase di test, ovvero, permettono di testare un metodo anche quando la classe da cui dipende il metodo che stiamo testando non è stata implementata
* **MOCK**: Permette di sapere quante volte un metodo è stato eseguito
  + **MOCKITO**: è una libreria che permette di semplificare la creazione di mock

* **MANUTENZIONE:** L’ultima parte del processo di sviluppo è anche la più grande e prevede i processi di modifica del codice dopo la consegna al cliente. I processi di manutenzione sono diversi e si dividono in:
  + **CORRETTIVI**: rimuovono gli errori
  + **ADATTIVI**: aggiustano il codice in ambienti nuovi
  + **PERFETTIVI**: aggiungono migliorie
  + **PREVENTIVI**: modificano il codice per prevenire l’insorgere di problemi

**PROCESSI DI** **CAMBIAMENTO**, essi sono i processi di cambiamento di un sistema e sono definiti da studi empirici e sono raggruppati nelle **leggi di Lehman:**

* **CAMBIAMENTO CONTINUO:** i sistemi hanno bisogno di essere continuamente adattati altrimenti diventano complessivamente meno efficienti
* **AUMENTO DI COMPLESSITÀ:** quando un sistema evolve, la sua struttura diventa più complessa, per questo deve essere fatto del lavoro che semplifica la sua struttura
* **AUTO-REGOLAZIONE:** attributi come dimensione, intervallo di release e numero di errori rimangono invariati in ciascuna release
* **STABILITÀ’ ORGANIZZATIVA:** durante la vita di un sistema il suo tasso di sviluppo è costante e indipendente dalle risorse impiegate per lo sviluppo
* **CONSERVAZIONE DELLA FAMILIARITÀ’:** in media, l’incremento di crescita di un sistema tende a rimanere costante o a diminuire
* **CONTINUA CRESCITA:** Il contenuto di funzioni di un sistema deve continuamente essere incrementato per garantire la soddisfazione di un cliente
* **DIMINUZIONE DELLA QUALITÀ’:** la qualità di un sistema diminuisce se non viene gestita e adattata durante i cambiamenti.

**COSTO DELLA MANUTENZIONE**

Come detto prima il costo della manutenzione è maggiore rispetto al costo dello sviluppo, esso però può essere ammortizzato se il team di sviluppo è coinvolto nella manutenzione, infatti, se la prima realizzazione è predisposta ai cambiamenti, sarà più facile effettuare modifiche, ad esempio, i DP tengono sotto controllo i costi di manutenzione. Esistono diversi **modelli di manutenzione**, tra i più comuni abbiamo:

* **QUICK FIX:** essi degradano la qualità del codice e sono solo a livello di codice stesso
* **MIGLIORAMENTO ITERATIVO:** sono cambiamenti atti in base ad un’analisi del sistema esistente e si concentrano sul mantenere stabile la complessità e il design
* **RIUSO:** si tenta di riutilizzare il più possibile quello già presente

Tra le modifiche che possiamo effettuare, esistono:

* **REFACTORING**
* **REVERSE ENGENEERING:** viene analizzato il sistema per estrarre il suo comportamento
* **RE- ENGENEERING:** viene progettato da capo il sistema

**METRICHE**

Una metrica definisce un set di misure per un sistema software, adottando una metrica possiamo monitorare il prodotto durante la sua costruzione e nel corso della sua vita. Ovviamente, solo gli **attributi interni** possono essere misurati direttamente da metriche, quelli **esterni** invece sono ricavati indirettamente. Tra le varie metriche distinguiamo due grandi gruppi:

* **METRICHE TRADIZIONALI**
  + **COMPLESSITÀ’ CICLOMATICA:** cc è il numero di test necessari per valutare l’algoritmo
  + **DIMENSIONE:** 
    - **LOC**: linee di codice
    - **NCNB**: non comment non blank
  + **COMMENT PERCENTAGE**: percentuale di commenti rispetto a LOC
* **METRICHE CK (solo per sistemi a oggetti)**
  + **WMC (**Weighted methods per class): la somma delle complessità dei metodi per una classe  **[bassa]**
  + **DIT** (Depth of Inheritance Tree): massimo numero di livelli dalla classe alla radice della gerarchia  **[bassa]**
  + **NOC** (Number of Children of a Class): numero di sottoclassi di una classe della gerarchia  **[alto]**
  + **CBO** (Coupling Between Object): numero di classi con cui una classe interagisce  **[bassa]**
  + **RFC** (Response for a class: numero di metodi eseguiti al ricevimento di un messaggio  **[bassa]**
  + **LCOM** (Lack of Cohesion of Methods): per ogni campo della classe, si calcola la percentuale di metodi che usano tale campo  **[bassa]**
* **TIPOLOGIE DI PROCESSI DI SVILUPPO**
  + **WATERFALL** (Si comincia la fase successiva solo se la fase precedente è completata)
  + **PROCESSO EVOLUTIVO**
    - **BUILD E FIX** (Documentazione inesistente, comprensione limitata del sistema da produrre)
    - **SVILUPPO PER ESPLORAZIONE** (Gli sviluppatori lavorano con i clienti e dalle specifiche iniziali si arriva fino al sistema software finale)
    - **SVILUPPO INCREMENTALE** (Sono implementate prima le funzionalità di base e poi al codice viene aggiunto altro codice per un altro gruppo di funzionalità)
    - **PROCESSO CBSE o BASATO SU COTS** (COTS = componenti esistenti, si cerca di usare quello che si ha già)
    - **A SPIRALE** (Focalizza su tanti prodotti parziali e ogni loop della spirale è una fase. E’ considerato un processo agile, ogni ciclo della spirale si divide in 4 parti: **identificazione degli obiettivi, valutazione dei rischi, produzione di una parte e convalida, revisione e pianificazione della prossima fase.)**
    - **SVILUPPO AGILE** (Sono più importanti auto-organizzazione, collaborazione e comunicazione rispetto a ordine e coerenza delle attività del progetto)
    - **EXTREME PROGRAMMING** (E’ un approccio basato sullo sviluppo e la consegna di piccoli incrementi di funzionalità, è adatto per progetti in cui i requisiti non sono stabili perché si adatta bene ai cambiamenti anche in fase avanzata. XP si focalizza sul codice ed è leggero). Il processo XP si basa su 12 pratiche:
      * **STORY CARD** (Scritte dal cliente permettono di ricavare una stima del loro tempo di sviluppo)
      * **STORYBOARD**
      * **GIOCO DI PIANIFICAZIONE** (Gli utenti scrivono le storie e se sono troppo difficili le separano, in genere vengono organizzate 3 settimane di lavoro)
      * **PICCOLE RELEASE**
      * **METAFORA** (Bisogna guidare il progetto con una singola metafora comprensibile dal cliente)
      * **DESIGN SEMPLICE** e comprensibile dal cliente
      * **CRC CARD** (Class Responsibility Collaboration, permettono di ragionare meglio in termini di oggetti e danno una visione complessiva del sistema)
      * **TESTING**
      * **PAIR PROGRAMMING** (Due programmatori lavorano insieme, uno scrive il codice, l’altro cerca soluzioni migliori e scrive i test). Un programmatore **driver** scrive il codice e un altro **navigator** suggerisce le modifiche. I ruoli si invertono spesso
* **POSSESSO DEL CODICE COLLETTIVO** (tutti possono modificare qualunque parte del sistema anche se esso è protetto da unit test. Chiunque è responsabile di tutto il sistema)
* **40 ORE A SETTIMANA** non è contemplata l’idea di fare ore extra, se necessarie sono dettate da cattiva pianificazione. Ogni giornata lavorativa inizia con 10 minuti di riunione in piedi.
* **CLIENTE SUL SITO** il cliente deve essere in sede quando serve, proprio a causa della scarsa documentazione.
* **REFACTORING** essendo che la progettazione iniziale è abbastanza scarna, sarà necessario fare modifiche strutturali in corsa tramite tecniche di refactoring per aumentare la qualità del codice.