DOMANDE ORALE IS

TEST WHITE BOX:

Noto il funzionamento interno dei vari componenti, si effettuano test per verificare che le operazioni interne siano svolte correttamente e si esaminano i dettagli del codice.

1. **Quali errori vengono rilevati dal test WB che il test BB non rileva?**
   Il test strutturale valuta la completezza della test suite in base alla struttura del programma stesso. Si distingue dal test funzionale poiché:
   - La *sorgente di informazione* per derivare i casi di test è la struttura del codice anziché le specifiche
   - La misura di *completezza della test suite* cambia, ed è basata sulla copertura degli elementi della struttura anziché delle specifiche
   - Implica dunque l'utilizzo di tecniche diverse per derivare i casi di test

   Una test suite derivata dal testing BB non può rilevare errori di una parte del programma che non viene mai eseguita (il *control flow*). Il test WB *complimenta* dunque il BB andando a rilevare *errori logici e le assunzioni scorrette* che hanno maggiore probabilità di accadere su *path poco eseguiti*. Spesso capita di ritenere che un path poco eseguito in realtà venga eseguito regolarmente. Ciò può comportare errori di progettazione rilevati solo quando si collauda quel path, ad esempio errori banali di digitazione che non vengo rilevati in fase di compilazione.

2. **Cos'è un Control Flow Graph (CFG)?**
   È un grafo orientato che rappresenta il trasferimento di flusso di controllo tra due blocchi di istruzioni. Un *nodo* rappresenta un blocco di istruzioni e un *arco* rappresenta il trasferimento del flusso di controllo. Un nodo non può contenere istruzioni che possono interrompere l'esecuzione di quel determinato blocco, eventuali ramificazioni o cicli possono incorrere solo alla fine del nodo.
   Il CFG serve a semplificare il testing di quei programmi contenenti molti if e cicli che porterebbero ad una moltitudine di cammini impossibili da

testare. Esiste anche il *Call Graph* che rappresenta le relazioni chiamante-chiamato tra procedure.

3. Come si progettano i test-case?

La creazione di una test suite è guidata da un *fattore di coverage* che identifica quanti elementi sono stati eseguiti, il che rende anche automatizzabile la creazione stessa. Ci sono vari criteri di progettazione:

- Percorrere tutti i cammini indipendenti all'interno di un modulo almeno una volta
- Esercitare tutte le decisioni logiche nei casi booleani o negli switch
- Eseguire tutti i cicli fino al loro valore limite ed entro i valori ammessi
- Esercitare tutte le strutture dati per assicurarne la validità

Questi possono essere rispettati, non tutti contemporaneamente, attraverso diversi criteri di adeguatezza:

1) *Copertura degli statement*: Richiede che ogni istruzione sia eseguita almeno una volta. Non sempre però comporta l'esecuzione del programma per tutti i valori delle decisioni(ad es. negli if). La misura di *coverage* è data da $\frac{n.statement\ eseguiti}{n.\ statement}$, e necessita di 1 caso di test minimo.
2) *Copertura delle decisioni*: Richiede che ogni arco del CFG sia percorso almeno una volta. Un test dovrà contenere almeno N dati di test per ciascuna delle N decisone, cioè degli N rami uscenti dal nodo. Questo criterio include quello degli statement. Non può però rilevare malfunzionamenti dovuti a valori non ammessi. La misura di *coverage* è data da $\frac{n.\ archi\ eseguiti}{n\ .archi}$, e necessita di 2 casi di test minimo.
3) *Copertura delle condizioni*: Richiede che ogni singola condizione che compare nelle decisioni del programma valga sia vero che falso per diversi dati di test. Non è detto che soddisfi il criterio delle decisioni. La *coverage* è data da $\frac{n.\ verità\ presi\ dalle\ singole\ condizioni}{n.\ condizioni\ singole}$, e necessita di 3 casi di test minimo.

Spesso si opta dunque per la combinazione 'condizioni + decisioni', che necessita però di minimo 4 casi di test. Ciò causa una crescita esponenziale del numero di casi di test. Esiste dunque il *MC/DC criterion.*

4. **Criterio MC/DC delle condizioni semplici e composte:**
   Serve a testare combinazioni rilevanti di condizioni, evitando però la crescita esponenziale dei casi di test. Con rilevanti si intende ogni condizione singola che influenza indipendentemente l'esito di una decisione. Per ogni condizione C sono richiesti 2 casi di test:
   1) I valori di tutte le condizioni valutate tranne C sono gli stessi
   2) La condizione composta vale *true* per un test e *false* per l'altro

   Ha una complessità lineare: N+1 casi di test per N condizioni.
   È composto allora da: copertura delle condizioni singole (C); copertura delle decisioni (DC); più una condizione (M).
   Ogni condizione deve influenzare indipendentemente l'output della decisione. È un buon compromesso tra completezza e numero di casi di test.

5. **McCabe e copertura dei cammini di base:**
   È un criterio che richiede che ciascun cammino di un insieme di cammini di base sia eseguito almeno una volta.
   La copertura dei cammini è una misura della complessità logica, è usata come guida per definire un insieme di base per l'esecuzione dei cammini e i test case effettuati sull'insieme di base garantiscono l'esecuzione di ciascuna istruzione almeno una volta.
   - *La complessità ciclomatica di McCabe* definisce il numero dei cammini indipendenti nell'insieme di base di un programma e rappresenta il limite superiore del numero di test da effettuare
   - *Un cammino indipendente* è qualunque cammino in un programma che introduce almeno un nuovo gruppo di istruzioni o una nuova condizione, che percorrere un arco nel CFG non percorso già in precedenza

   L'insieme di base ovviamente non è unico, ma la complessità ciclomatica indica quanti cammini devono essere ricercati per comporre un insieme.
   Il numero ciclomatico è calcolato in tre diversi modi:
   1) Numero delle regioni del grafo (tutte le interne + quella esterna)
   2) A-N+2*C dove: A è il numero di archi; N numero di nodi; C numero delle partizioni del grafo (quasi sempre 1)

3) P+1 dove P sono i ‘nodi predicati’, cioè quei nodi dove viene effettuata una decisione o dove c’è una condizione (if e while).

TEST BLACK BOX:

Note le specifiche funzionali, si effettuano test per dimostrare che ciascuna funzione è completamente operativa, ignorando i dettagli interni

1. **Quali sono gli obbiettivi del testing?**
   L'obbiettivo principale della fase di testing è il collaudo del software, non devo trovare i difetti, bensì i malfunzionamenti. Soltanto successivamente con la fase di debugging andrò a trovare l'errore che causava quei determinati malfunzionamenti.

2. **Tesi di Dijkstra e inaccuratezza ottimistica:**
   Il testing indirettamente dimostra che il software rispetta le specifiche ed i requisiti e fornisce un'indicazione sulla sua affidabilità e qualità globale, ma *non può dimostrare l'assenza di difetti, può solo dimostrare la loro presenza.*
   Essendo impossibile fare un testing esaustivo al 100% ci sono vari approcci alla verifica di un software:
   1) *Inaccuratezza ottimistica* (testing): si accettano ottimisticamente programmi, pur sapendo che possono esserci dei malfunzionamenti non rilevati nella fase di testing
   2) *Inaccuratezza pessimistica* (analisi): se nell'analisi, che verifica una sola proprietà alla volta, si trova un malfunzionamento non è detto che questo sia un problema reale ma potrebbe essere legato alla presenza di variabili non dichiarate (warning). Si parla allora di pessimismo perché il programma non viene accettato anche se magari il software finale non avrà errori legati a quei warning
   3) *Proprietà semplificate*: riduce il grado di libertà per semplificare la proprietà da controllare

   Inoltre, il testing viene diviso in più fasi: d'unità, d'integrazione, di sistema e di accettazione.

3. **Differenze tra le fasi del testing:**
   - *Il testing di unità* verifica i moduli. Ciascun modulo viene verificato sulla base della documentazione relativa alla progettazione dettagliata

ed è effettuato subito dopo la codifica e usa *driver* (moduli chiamanti) e *stub* (moduli gerarchicamente inferiori). Questo ha un costo oneroso, in quanto *driver e stub* vengono realizzati ma non sono poi presenti nel prodotto finale, e inoltre alcuni moduli non possono essere comunque testati.

- *Il testing di integrazione* rileva errori relativi all'interazione tra moduli. Prende i moduli testati singolarmente e li aggiunge pochi alla volta costruendo in maniera incrementale la struttura del programma. È necessario per scovare errori di comunicazione, effetti collaterali e problemi per i dati globali. Può essere di tipo: *top-down*, iniziando dal *main* (usato come *driver*) e procedendo *depth-first* o *breadth-first* (profondità o larghezza), sostituendo gli *stub*; *bottom-up*, dove *cluster* di moduli senza figli eseguono come *driver* e non c'è bisogno di *stub*.
- *Nel testing di sistema* il software viene integrato con altri elementi del sistema (hardware) ed il tutto va testato e validato. Lo scopo è la verifica che tutti gli elementi del sistema siano stati correttamente integrati e che svolgano bene le funzioni loro assegnate.
- *I test di accettazione* sono condotti dall'utente finale per consentire la validazione di tutti i requisiti. Può richiedere molto tempo e portare alla rilevazione di errori cumulativi che non vengono corretti e possono portare al degrado del sistema. Si scinde in *α-test*, condotto dal cliente presso lo sviluppatore in ambiente controllato, e *β-test*, condotto da uno o più clienti senza lo sviluppatore, che lavorerà solo alle opportune modifiche per il rilascio finale.

4. Differenze tra verifica e validazione:

- *La validazione* risponde alla domanda: il software soddisfa i bisogni reali dell'utente? Stiamo costruendo *il software adeguato*? (right sw)

Riguarda i requisiti informali d'utenza e si fa a fronte delle funzionalità richiese. Esamina se il sistema software risolve il problema per cui viene prodotto, e non garantisce la correttezza del software.

- *La verifica*: il software soddisfa la specifica dei requisiti? Stiamo costruendo *il software in modo giusto*? (sw right)

Riguarda specifiche formali prodotte dall'analista e intende l'insieme delle attività volte a stabilire se il software è corretto rispetto alle specifiche dell'analista. Viene svolta attraverso *testing e debugging*.

5. **Principio di partizionamento e Partition-testing:**
   Bisogna sfruttare le conoscenze per scegliere campioni 'speciali' che più probabilmente includono regioni dello spazio degli input, o che sono più propensi a causare *failures* (*failure-causing inputs*). Basta dunque questo singolo input per simulare il comportamento che il modulo avrebbe rispetto ad un'intera classe. Ciò riduce significativamente i casi di test e dunque il costo.
   Si definisce poi *(Quasi-)Partition testing* un metodo di testing che divide l'insieme 'infinito' di casi di test in un insieme finito di classi, che possono sovrapporsi, la cui unione è l'intero spazio.
   Quando si adopera un *partition testing* nel contesto di un *test funzionale*, cioè quando le partizioni sono scelte sulla base delle specifiche funzionali, si parla di *specification-based partition testing* o *functional partition testing*. Il test funzionale usa la specifica per partizionare lo spazio degli input e per trovare quelli maggiormente *failure-causing*, che solitamente, anche se non c'è la garanzia, si trovano ai limiti delle classi.

6. **Quali sono i vincoli del Category-Patition testing (CPT)?**
   Il CPT è diviso in tre fasi:
   1) Decomposizione delle funzionalità che possono essere testate indipendentemente. Per ogni funzionalità si identificano i *parametri* che la descrivono e gli *elementi* da cui dipende. Per ogni parametro ed elemento poi si individuano una serie di caratteristiche elementari, dette *categorie*
   2) Identificazione di 4 tipi di valori rappresentativi per ogni categoria: *normal, special, boundary* ed *erroneus*.
   3) Si introducono *vincoli* sui valori individuati, si impone che le combinazioni possono avere più di un errore e si valuta se i valori sono compatibilmente combinati.

   I vincoli sono fondamentali perché riducono di molto il numero dei casi di test. Possono essere: vincolo *error, single* e *property*.

1) *Il vincolo errore* indica una classe di valori che da luogo ad un errore. Ad esempio, si può ipotizzare che un solo errore per combinazione sia sufficiente al fallimento del sistema
2) *Il vincolo single* indica una classe di valori che si desidera testare una sola volta e non su tutte le possibili combinazioni per ridurre il numero di test-case. Il calcolo del numero di test è uguale a quello di error, ma ha motivazioni diverse
3) *Il vincolo proprietà* elimina le combinazioni invalide dei valori delle categorie: [property] raggruppa i valori di un componente che hanno una proprietà in comune; [if-property] il valore può essere in combinazione solo con valori di altre categorie che hanno la label [property]

Talvolta non è facile inserire dei vincoli e si rischia di inserirne alcuni senza una motivazione valida. È in questi casi che si usa il test combinatoriale.

7. Quando si usa il CPT e quando il Pairwise combinatorial testing?

Data una specifica ci possono essere più approcci per derivare i casi di test. Ad esempio, la presenza di molti vincoli nel domino di input suggerisce un metodo di partizionamento con vincoli (category-partition testing). Al contrario, valori di input con pochi o nessun vincolo suggeriscono un approccio combinatoriale (pairwise testing).

Con il CPT si identificano manualmente gli attributi che caratterizzano lo spazio di input, che variano all'interno di un set di valori noti e si applicano dei vincoli per ridurre il numero dei casi di test.

Con il Pairwise Testing l'identificazione di attributi che caratterizzano lo spazio di input è automatica. Gli input che causano failure variano su un set di valori opportunamente ridotto, considerando combinazioni in coppie o triple di valori. Ad esempio, in un database se si scelgono le tuple in modo tale da avere più coppie nella stessa tupla e si ripete la tupla una sola volta, si può ridurre significativamente il numero di test. In genere si usano degli algoritmi per calcolare le tuple che non ripetano più volte la stessa coppia.

8. **Test sistematico vs test randomico:**
   Il test sistematico prova a selezionare input particolarmente rilevanti, scegliendo di solito valori rappresentativi di classi che tendono a fallire spesso o a non fallire mai. Il test Funzionale è un sistematico.
   Nel test randomico si scelgono gli input campionandoli dallo spazio degli input secondo una distribuzione di probabilità (uniforme o non). Ciò evita una possibile influenza del progettista nella scelta degli input, ma tratta tutti allo stesso modo e può essere svantaggioso perché la distribuzione dei fault non è uniforme.

9. **Come faccio a stabilire nel testing quando devo fermarmi?**
   Quando vengono raggiunti i requisiti di verifica e validazione del software, cioè se dopo aver ricevuto tutti i test funzionali e strutturali risulta corretto e pronto per il rilascio. Questo avviene quando effettuo un test per ogni requisito e questi funzionano tutti. È detto '*criterio di adeguatezza'*.

IMPLEMENTAZIONE JAVA E MODELLAZIONE

1. **Quali sono le tipologie di *collection* presenti nel *collection framework* di Java?**
   Le *collection* sono delle strutture dati dinamiche presenti nel package *java.util* e fanno uso della programmazione gerarchica per astrarre il tipo dei dati da loro manipolati, nota in Java come '*generics*'.
   Esistono quattro tipi di classi 'container' in Java: List, Set, Queue e Map. Ognuna di esse ha varie implementazioni: Array List, Linked List, Hash Set, Hash Map, etc.
   Le classi a cui appartengono gli oggetti contenuti nella struttura dati sono messe tra parentesi angolari. A differenza degli array, queste classi si ridimensionano automaticamente ed è possibile inserire un numero arbitrario di oggetti senza definirne la dimensione. Un container è un oggetto che raggruppa elementi multipli di una singola unità.
   Il collection framework, cioè il contesto dei container, è formato da: *interfacce*; *tipi di dato astratto* che rappresentano i container e ne permettono la manipolazione indipendente dalle caratteristiche; *implementazioni*; *classi concrete* che implementano le interfacce e sono dati riusabili; *algoritmi*; *metodi* coi quali si effettuano operazioni sui container. Esse rendono lo sviluppo migliore, più facile e veloce e con più API(application programm interface), che rendono il software riusabile, ma sono molto complesse e poco articolate.
   Delle collection fanno parte *List e Set*, e sono raccolte sequenziali di singoli elementi.
   - Set: collezione non ordinata e senza duplicati. È l'astrazione dell'insieme matematico.
   - List: collezione ordinata con possibili duplicati. Si accede agli elementi mediante un numero intero rappresentante la posizione.

2. **Cosa si intende con '*generics*' in Java?**
   Generics è il meccanismo che java offre per effettuare la programmazione parametrica. I tipi parametrizzati rivestono particolare importanza poiché consentono la creazione di classi, interfacce e metodi per i quali il tipo di dato sul quale si opera può essere specificato come

parametro. Parliamo quindi di classi, interfacce e metodi generici.

3. **Come si traducono in Java le associazioni e le aggregazioni?**
   Associazioni: un'istanza di una classe è provvista di riferimenti alle classi associate a seconda della cardinalità. Per le relazioni 'a 1' sarà contenuta un singolo riferimento; per quelle 'a MOLTI' ci sarà un vettore di riferimenti. Ad esempio, in un'associazione '1 a MOLTI' un'istanza di una classe è associata a più oggetti di un'altra. Nella prima sarà presente un attributo della seconda classe, mentre la relazione inversa può essere realizzata tramite un vettore di riferimenti, e non con variabili membro.
   La traduzione dipende anche dalla direzione: unidirezionale o bidirezionale (sottintesa).
   Aggregazione: oltre a contenere i/il riferimenti/o alla classe collegata, il costruttore di quella determinata classe dovrà ricevere in ingresso:
   - per l'aggregazione *lasca*: un puntatore all'oggetto contenuto
   - per l'aggregazione *stretta*: i parametri per costruire il contenuto. Questo si realizza aggiungendo una variabile membro alla classe contenitore, del tipo della classe contenuto, oppure implementando il costruttore in modo da richiamare il costruttore del contenitore

PATTERN:

I pattern sono soluzioni a problemi ricorrenti che si incontrano in applicazioni reali, che spiegano come realizzare oggetti e le interazioni tra essi, costruendo codice riusabile. Sono applicabili in ogni fase del ciclo di vita di un software e sono: di analisi; *architetturali*; *di progettazione*; di codifica.

Permettono il riuso della conoscenza/esperienza di progettazione, dato che raramente i problemi sono nuovi e unici, e forniscono anche indicazioni su "dove cercare le soluzioni ai problemi". Stabiliscono una terminologia comune e condivisa e forniscono una prospettiva di alto livello, liberandoci dal dover gestire troppo presto i dettagli della progettazione.

PATTERN ARCHITETTURALI:

Sono architetture software, cioè l'organizzazione di base di un sistema espressa dai suoi componenti, astratte e riusabili. Permettono la decomposizione gerarchica di un sistema in un insieme ordinato di *layer*.

1. Il pattern MVC: Model-View-Control

È stato uno dei primi pattern che nacque dalla necessita di visualizzare dati generici tramite interfaccia grafica in diverse rappresentazioni degli stessi dati.

Vengono strutturati vari modelli per risolvere i vari problemi posti dal progetto.

- *Il Controller*: trasforma gli input dell'utente all'interno della *View* in azioni da eseguire sul *Model*, e selezionando le schermate della *View* richiese implementa la logica di controllo dell'applicazione.
- *Il Model*: si occupa della gestione dei dati e offre alla *View* le funzionalità per l'accesso, e al *Controller* le funzionalità per l'aggiornamento. Ha la responsabilità di notificare alla *View* eventuali aggiornamenti verificatisi dopo richiese del *Controller*, al fine di presentare sempre dati aggiornati.
- *La Vista*: gestisce la logica di presentazione dei dati. Si comporta come l'Observer (non osserva, ma resta in attesa che gli venga detto di osservare) rispetto ai *Model*, a cui possono richiedere aggiornamenti in tempo reale.

2. Approccio BCE: Boundary-Control-Entity

Il BCE è una variante del MVC che nasce con l'obbiettivo di separare le responsabilità tra gli elementi e la business logic dal resto dell'applicazione, mentre il MVC separa la selezione dell'interfaccia utente dal resto dell'applicazione ed è più focalizzato proprio sull'interfacciamento.

Il package Boundary contiene gli oggetti responsabili dell'interfaccia utente e della logica di presentazione.

Il package Control contiene oggetti che percepiscono gli eventi generati dall'utente e controllano appunto l'esecuzione di un processo di business, rappresentando azioni e attività degli use-case.

Il package Entity contiene oggetti che rappresentano la semantica delle entità del dominio applicativo, e corrisponde alle strutture dati nel database del sistema e sono sempre oggetti persistenti.

Da questo si evolve poi il BCED (D=database), che lascia le classi di sistema in *Entity* e sposta quelle responsabili dell'estrazione dei dati nel package *Database*. L'*Entity* inoltre memorizza i dati che ottiene dal DB invocando *Carica* e *Salva* (unOggetto).

Il package Database è un'interfaccia verso il DB che apre e chiude connessioni verso esso, che estrae e memorizza i metadati degli oggetti contenuti e informa il DB stesso di richieste di commit e rollback.

Tutti e tre i pattern necessitano della persistenza dei dati.

L'accesso al database può essere gestito con tre strategie:

- Accesso diretto: ogni Entity si occupa autonomamente della gestione e dell'accesso al database. È poco efficiente.
- Classi DAO (direct access objects), cioè oggetti per accesso ai dati.
- *Persistence framework*: si libera il programmatore dalla scrittura di codice SQL, che viene generato automaticamente. *Ipernet e Dotnet*.

3. Perché classi DAO?

Le classi DAO rappresentano la strategia più vantaggiosa per gestire l'accesso al DB nel caso di utilizzo del pattern BCED. La strategia è quella di creare delle classi DAO dedicate per ogni entità che si occupano dell'interazione con la base di dati. Si crea una classe DAO per ogni classe che rappresenta un'entità del dominio che si vuole memorizzare.

Questa classe DAO conterrà i metodi di interrogazione e manipolazione della corrispondente classe di dominio. In particolare, conterrà le funzionalità CRUD: *create, read, update* e *delete.* Possono anche gestire rollback e ripristino.

4. **Pattern Broker:**

   Il Broker è un pattern architetturale, con schema logico di tipo publish/subscribe, cioè: gli oggetti serventi rendono noto i servizi forniti, gli oggetti clienti che sono interessati ad a uno o più servizi si iscrivono per utilizzare il servizio e le pubblicazioni e le iscrizioni avvengono presso il Broker.

   Ciò conferisce al sistema caratteristiche di trasparenza alla locazione, al linguaggio di programmazione e alla piattaforma target.

   L'architettura con Broker rappresenta una sorta di bus virtuale a cui ogni oggetto affida i suoi messaggi e il Broker è l'elemento di integrazione fra i "moduli" che compongono il sistema software. È molto utilizzato nelle architetture dei sistemi middleware come: IMB SOM o Microsoft OLE.

   Con questo pattern un'applicazione può accedere a servizi distribuiti semplicemente inviando messaggi all'oggetto appropriato, piuttosto che doversi impegnare in una comunicazione inter-processo di basso livello.

   Si fa uso del '*bridge'* per rendere le comunicazioni col *broker* trasparenti e nascondere i dettagli di implementazione della comunicazione.

   La registrazione di un server avviene cosi: il broker viene eseguito nella fase di inizializzazione del sistema e resta in attesa, aspettando che arrivino messaggi. L'utilizzatore esegue l'applicazione server, che esegue una fase di inizializzazione dopo la quale si registra presso il broker. Il broker riceve il messaggio di richiesta registrazione, estrae le informazioni necessarie e le memorizza in un repository, che serve per trovare e attivare i server, e invia un messaggio di avvenuta registrazione al server. Ricevuta la conferma, il server è attesa di richieste dai client.

5. **Approccio Client/Server:**
   È un'architettura distribuita dove dati ed elaborazione sono distribuiti su una rete di nodi di due tipi:
   - *Server*: processori potenti e dedicati che offrono servizi specifici come stampa, gestione di file system, compilazione, gestione traffico di rete.
   - *Clienti*: macchine meno prestazionali sui quali girano le applicazioni utente, che utilizzano i servizi dei server.

   Ogni applicazione può essere suddivisa logicamente in 3 parti:
   - *Presentazione*: che gestisce l'interfaccia utente (gestione eventi grafici, controlli formali sui campi in input)
   - *Logica applicativa*
   - *Gestione dei dati persistenti*

   La politica di allocazione di queste componenti porta alla classificazione dell'architettura su diversi livelli (2-tiered, 3-tiered, n-tiered).
   La *Business logic* trattata in modo esplicito:
   - Livello 1: gestione dei dati (DBMS, file XML)
   - Livello 2: business logic (processamento dati)
   - Livello 3: interfaccia utente (presentazione dati, servizi)

   Ogni livello ha obiettivi e vincoli di design propri e nessun livello fa assunzioni sugli altri: il 2 non fa assunzioni su rappresentazione dei dati, né sull'implementazione dell'interfaccia utente; il 3 non fa assunzioni su come opera la *business logic*.
   Ha molti vantaggi in termini di flessibilità, manutenibilità, testabilità, estensibilità. Un esempio sono i Web Services come AWS o le infrastrutture cloud.

PATTERN DI PROGETTAZIONE:

È una soluzione comprovata a un problema tipico che sorge nello sviluppo software in uno specifico contesto. Sono catalogati e descritti da: *nome*, simbolico; *problema*, descrive quando applicare il pattern; *soluzione*, descrive gli elementi che costituiscono il progetto e le loro interazioni; *conseguenze*, risultati e vincoli ottenuti applicandolo.
Vengono classificati in base a due caratteristiche:

- *Scopo* (purpose), cosa fa il pattern, che si divide a sua volta in:
  - *creazionali*: astraggono il processo di creazione di oggetti

    - *strutturali*: trattano la composizione di oggetti e classi per formare strutture più complesse
    - *comportamentali*: algoritmi di interazioni reciproche tra classi o oggetti e di distribuzione di responsabilità
- *Ambito* (scope), specifica se è relativo a:
    - *Classi*: trattano relazioni statiche, determinate a compile-time
    - *Oggetti*: trattano relazioni dinamiche, che variano a run-time

1. **Desing patterns creazionali:**
   In quelli relativi alle classi la creazione di oggetti è affidata a sottoclassi basandosi sull'ereditarietà, mentre per quelli elativi agli oggetti questa responsabilità è affidata ad altri oggetti. Ciò consente di rendere il sistema indipendente da come sono gli oggetti sono creati, rappresentati e organizzati. Fanno parte di questi DSC:
   - *Abstract factory*: un oggetto che serve a creare istanze di altri oggetti
   - *Factory method*: un oggetto che serve a creare istanze di diverse classi derivate
   - *Prototype*: istanza completa di un oggetto che serve per essere clonato
   - *Singleton*: un oggetto che restituisce una sola istanza di sé stesso
   - *Builder*: separa costruzione e rappresentazione per oggetti complessi

2. **Come si implementa il Singleton?**
   In primis assicura che una classe abbia una sola istanza e fornisce un punto di accesso globale ad essa, ed è la classe stessa responsabile di avere traccia della sua unica istanza e di fornire un modo per accedervi.
   Il Singleton va usato quando deve esserci esattamente un'istanza e deve essere accessibile da un unico punto globale, e questa istanza deve essere estendibile con subclassing e i clients devono essere capaci di usare le istanze estese senza modificarne il codice. Il tutto permette un accesso controllato e una maggiore pulizia rispetto all'uso delle variabili esterne.
   Viene implementato nascondendo l'operazione di creazione in un'operazione di classe *static* e rendendo il costruttore *protected*:

- Class Singleton{
  public:
  static Singleton* Instance(); -> return uniqueInstance
  protected:
  Singleton();
  private:
  static Singleton* _instance();
  };
- Singleton* Singleton::_instance = 0;
- Singleton* Singleton::Instance(){
  if (_instance == 0) {
  _instance = new Singleton;
  }
  return _instance;
  };

Anche se nasce per l'unicità dell'istanza, può essere ampliato al permettere un determinato numero di istanze.
Static è un qualificatore che indica che un membro è un membro di classe e non di oggetto.

3. Come si implementa l'Abstract factory?

Offre una interfaccia per creare famiglie di oggetti in relazione o dipendenti, senza specificare le loro classi concrete. L'abstract factory va usato quando:

- Un sistema deve essere indipendente da come i suoi prodotti sono creati, composti e rappresentati.
- Un sistema deve essere configurato per più famiglie di prodotti.
- Una famiglia di prodotti è progettata per servirsi di un solo insieme di classi per volta, ed occorre garantire questo vincolo.
- Occorre offrire una libreria di prodotti, ma si vuole esporre solo le loro interfacce e non le loro implementazioni.

In questo modo si isolano le classi concrete, il che aiuta a controllare le classi di oggetti che una applicazione crea, isolandole nelle

‘*concrete factory’* piuttosto che nel codice del client, e rende semplice cambiare famiglie di prodotti scegliendo una di esse.
Promuove la coerenza se gli oggetti prodotti in una famiglia sono progettati per lavorare insieme, ma è difficile supportare nuovi tipi di prodotti, perché necessita di cambiare la ‘*Factory Interface’* e tutte le sottoclassi.

4. **Come si implementa il Builder?**
   Usiamo il pattern Builder quando l’algoritmo per creare un oggetto complesso dovrebbe essere indipendente dalle parti che lo costituiscono e da come sono assemblate, il procedimento di costruzione consente rappresentazioni differenti per l’oggetto che viene costruito e può essere separato in passi discreti, ad esempio facendo il parsing di un testo.
   La struttura di un Builder è:
   - *Builder*: specifica un’interfaccia astratta per creare parti di un oggetto Product
   - *ConcreteBuilder*: costruisce e assembla le parti di Product implementando l’interfaccia Builder
   - *Director*: costruisce un oggetto usando l’ interfaccia Builder
   - *Product*: rappresenta l’oggetto complesso in costruzione

   (1) Il *Client* crea l’oggetto *Director* a lo configura con un *Builder*, (2) poi il *Director* notifica il *Builder* di costruire ogni parte del prodotto. (3) Il *Builder* gestisce le richieste del *Director* e aggiunge parti al prodotto, (4) infine il *Client* recupera il prodotto dal *Builder*.

5. **Desing patterns strutturali:**
   Sono relativi a come classi e oggetti sono composti per formare strutture più grandi. Quelli basati su classi utilizzano l’ereditarietà per generare classi che combinano le proprietà delle classi base, quelli basati su oggetti mostrano come comporre nuovi oggetti per realizzare nuove funzionalità. Fanno parte dei DPS: *composite, façade*, *adapter, bridge* e *proxy.*

6. **Come si implementa il Façade?**
   Rende più semplice l’uso di un sottosistema fornendo un’unica interfaccia per un insieme di funzionalità sparse. L’utente deve

preoccuparsi di cosa vuole fare e non di come viene fatto, nasconde allora al cliente le componenti del sottosistema. È facile da implementare perché è un aggregato di interfacce, e una singola classe rappresenta un intero sistema. Può essere anche una classe astratta con sottoclassi concrete per diverse implementazioni. Ad esempio, un robot con quattro classi: *camera*, per identificare; *arm*, braccio mobile; *pliers*, per afferrare; *operator*, che usa le parti del robot. Senza façade l'operator deve conoscere la struttura ed il comportamento di ogni classe. Utilizzandola invece l'operatore si interfaccia solo con la façade, la quale conosce e controlla gli oggetti del sottosistema camera-arm-pliers.
Per sottosistemi stratificati, è possibile utilizzare il façade come entry point per ciascun livello.

7. **Come si implementa il Composite?**
   È una struttura ad albero volta al rappresentare oggetti complessi. Lo scopo è comporre oggetti in strutture ricorsive ad albero per rappresentare gerarchie parte-tutto e consentire ai clients di trattare in modo uniforme gli oggetti singoli (semplici) e composizioni di oggetti. Ad esempio, un programma di grafica deve consentire di creare oggetti semplici (punti, linee), raggrupparli in oggetti composti (triangoli, quadrati), per poi trattarli come se fossero oggetti semplici.

8. **Come si implementa l'Adapter?**
   Somiglia molto al façade ma ha utilizzi diversi. Il suo scopo è adattare l'interfaccia di una classe già pronta all'interfaccia che un cliente si aspetta. Talvolta capita che una classe riusabile non lo è solo perché la sua interfaccia non coincide con quella del dominio specifico di un'applicazione, e quindi basterebbe creare un adattatore per poter utilizzare la classe. Esistono due tipi di adapter:
   - *object adapter*: che è basato su delega o composizione
   - *class adapter*: che è basato sull'ereditarietà. Eredita sia dall'interfaccia attesa, sia dalla classe adattata, e inoltre non permette l'ereditarietà multipla perché l'interfaccia attesa deve essere un'interfaccia e non una classe

Un class adapter può essere implementato facendo ereditare a *Adapter* pubblicamente da *Target* e privatamente da *Adaptee*. In tal modo *Adapter* è un sottotipo di *Target* ma non di *Adaptee*.

9. Desing patterns comportamentali: Observer

L'Observer serve a definire una dipendenza "uno a molti" tra oggetti in modo che quando un oggetto cambia stato, tutti gli oggetti dipendenti da lui vengono notificati e aggiornati automaticamente. Si implementa tramite attributi pubblici, ma comporta dei problemi di sincronizzazione. Quindi gli osservatori si registrano presso l'oggetto osservato e quando l'oggetto osservato cambia stato, notifica tutti gli osservatori (invocando un metodo). Quando notificato, ogni osservatore decide se non fare niente o se richiedere all'osservato delle informazioni sullo stato. L'oggetto osservato non conosce l'identità precisa degli osservatori, che possono essere delle View sullo stesso oggetto e vengono aggiunti/tolti a runtime.
È applicabile quando:

- Quando un'astrazione ha due aspetti, una dipendente dall'altra, e incapsulando questi aspetti in oggetti separati è consentito modificarli e riutilizzarli indipendentemente
- Quando un cambiamento ad un oggetto comporta il cambiamento di altri e ma non si sa di quanti
- Quando un oggetto dovrebbe essere capace di notificare altri oggetti senza fare assunzioni su chi siano questi oggetti. In pratica non vogliamo che questi oggetti siano fortemente accoppiati.

Java fornisce due interfacce per questo pattern: l'interfaccia *Observer*, che fa l'*update*; e la classe *Observable*, che chiama i metodi *addObserver, deleteObserver e notifyObserver*.

MANUTENZIONE:

La manutenibilità è un fattore di qualità del software che indica la facilita e l'economicità con cui vengono eseguite le attività di manutenzione, che solitamente supera il 70% del costo del software. La manutenzione può essere di quattro tipi:

1) *Manutenzione correttiva*: riguarda la rimozione di errori presenti nel prodotto dopo il rilascio, o eventuali errori introdotti a seguito di un'altra manutenzione. (20%)
2) *Manutenzione adattiva*: riguarda le modifiche dell'applicazione in risposta ai cambiamenti dell'ambiente (hardware, OS, DBMS). (25%)
3) *Manutenzione perfettiva*: riguarda i cambiamenti nel software per migliorare alcune qualità. La richiesta può provenire dagli sviluppatori per tenere il software ben aggiornato, o dai committenti per rispondere a nuovi requisiti. (50%)
4) *Manutenzione preventiva*: riguarda modifiche che rendono più semplice la correzione, gli adattamenti e le migliorie. (5%)

Abbiamo poi due diverse qualità*: la riparabilità e l'evolvibilità*. Un sistema software è *riparabile* se i suoi difetti possono essere corretti con una quantità ragionevole di lavoro. Solitamente un software che consta di moduli ben progettati è più facile da analizzare e riparare di un sistema monolitico, in quanto una corretta strutturazione in moduli favorisce la riparabilità. Un software è *evolvibile* se facilita cambiamenti che gli permettono di gestire nuovi requisiti. È una qualità che richiede la capacità di anticipare i cambiamenti in fase di progettazione e tende a diminuire con i rilasci successivi.

1. **Per cosa serve il '*regression test*' e come si può ottimizzare?**
   Avviene durante la fase di manutenzione, in seguito all'implementazione delle modifiche e prima del test di accettazione, e testa se le modifiche che sono state appena effettuate vanno ad influire sulle funzionalità già precedentemente testate e funzionanti (validate), e verifica che non ci sia nessun malfunzionamento. Bisogna capire in seguito ad una modifica quali requisiti sono impattati. Nel caso del modello a rilasci incrementali, il test di regressione assume un'importanza notevole in quanto: lo sviluppatore dopo aver sviluppato un nuovo modulo da integrare fa prima

un test in locale, poi fa il push e successivamente esegue il test di integrazione per controllare se il modulo appena integrato funziona bene con l'intero sistema.

2. **Cosa si fa per 'ringiovanire' il software?**
   Si attuano degli interventi finalizzati a migliorare la manutenibilità di un software deteriorato dagli interventi di manutenzione subiti, e possono essere di diversi tipi: *Refactoring*; *Reverse Engineering*; *Restructuring*; Ridocumentazione; Reengineering. La *ridocumentazione* è un'analisi statistica del codice sorgente al fine di produrre documentazione del sistema, che genera in output grafi delle chiamate, data-flow o control-flow, che servono a capire se c'è bisogno della ristrutturazione. La *ristrutturazione* avviene dopo la ridocumentazione ed è un'attività che trasforma il codice esistente in un codice equivalente dal punto di vita funzionale, ma migliorato rispetto alle qualità.

3. **Cos'è il *Refactoring*?**
   È un insieme di tecniche usabili per migliorare il codice ed il design di sistemi object-oriented, dato che anch'essi sono soggetti al deterioramento e diventano *legacy*. Queste tecniche cercano di eliminare i cosiddetti '*Bad Smells*' dal codice, ad esempio: codice duplicato, metodi troppo lunghi, classi troppo grandi, cambiamenti divergenti in una classe. Esistono due tool per il refactoring: 'Net Beans' e quello di Eclipse.

4. **Cos'è la *Reengineering?***
   È l'esaminazione e l'alterazione di un sistema per ricostituirlo in una nuova forma e nella successiva implementazione del nuovo modulo. È un'attività di reimplementazione di un sistema software svolta per migliorare a manutenibilità di un software esistente. Essa può comprendere: Ridocumentazione, Ristrutturazione e riscrittura di parte del software (o anche di tutto) senza modificare l'insieme di funzionalità che esso realizza. Gli obiettivi della reengineering sono:
   - Modularizzazione del sistema: suddivisione di un sistema monolitico in parti riusabili.
   - Miglioramento delle performance

- Porting: migrazione verso altre piattaforme.
- Estrazione del progetto: per migliorarne la manutenibilità e la portabilità.
- Utilizzo di nuove tecnologie: riguardo caratteristiche del linguaggio, standards, librerie, etc.

È inoltre suddivisa in varie fasi: analisi dei requisiti; cattura del modello; individuazione del problema; soluzione del problema; trasformazione del programma. È una sorta di '*forward engineering*' su un sistema però già esistente.

5. Cos'è la *Reverse engineering*?

È un'attività che consente di ottenere specifiche e informazioni sul design di un sistema a partire dal suo codice, attraverso processi di estrazione ed astrazione di informazioni. L'analisi del sistema serve ad identificare i componenti e le loro interazioni e a creare rappresentazioni del sistema in altre forme o ad un livello di astrazione maggiore. Dopo l'estrazione delle informazioni si passa all'astrazione da effettuare.

1) *Estrazione*: è un'analisi del codice o di altri artefatti software, allo scopo di ottenere informazioni relative al sistema analizzato. Particolarmente utili sono quegli strumenti in grado di estrarre informazioni da un codice sorgente qualsiasi, a partire dalla conoscenza della grammatica del linguaggio di programmazione.
2) *Astrazione*: si esaminano le informazioni estratte e si cercano di astrarre diagrammi, o viste, ad un più alto livello di astrazione (diagrammi di progetto). I processi di astrazione non sono completamente automatizzabili poiché necessitano di conoscenza ed esperienza umana.

6. Cosa posso fare nel caso di un sistema legacy?

Un sistema legacy è vecchio, di grandi dimensioni (centinaia di migliaia di linee di codice), è scritto in assembly o in un linguaggio di vecchia generazione ed è stato probabilmente sviluppato prima che si diffondessero i moderni principi dell'ingegneria del software. La manutenzione è stata svolta in modo da seguire le modifiche nei requisiti, aumentando così l'entropia (il disordine) del sistema, il che rende la

successiva difficile e onerosa. Questo tipo di sistemi può però realizzare funzionalità cruciali e irrinunciabili per l'organizzazione, contenendo anni di esperienza accumulata nell'ambito del dominio specifico del problema.

Per la manutenzione occorre dunque valutare il sistema sia da una prospettiva Economica che Tecnica:

- *Prospettiva Economica* (Business Value): L'azienda ha realmente bisogno del sistema?
- *Prospettiva Tecnica*: Qual è la qualità del software applicativo, e del suo ambiente di utilizzo (sia software che hardware)?

Dopo un'accurata analisi, viene categorizzato in uno di questi quattro modi, e si decidono le modalità su come agire:

1) *Low-quality, low-business value*: dovrebbe essere abbandonato.
2) *Low-quality, high-business value*: realizza funzionalità importanti ma è costoso mantenerlo. Dovrebbe essere reingegnerizzato in modo da rendere le future operazioni di manutenzione necessarie più agevoli ed efficaci (cioè finire nel quarto caso).
3) *High-quality, low-business value*: Si può decidere sia di abbandonarlo (in quanto poco importante), sia di rimpiazzarlo con COTS (recupero componenti importanti e sostituzione del resto con moduli già fatti), oppure mantenerlo.
4) *High-quality, high business value*: Su di esso si eseguono le operazioni di manutenzione. In questo modo, però, la qualità diventerà via via più bassa, fino a finire nel secondo caso.

Le metriche per la prospettiva economica sono: *valore economico; valore dei dati; utilità; specializzazione*.

Le metriche per la prospettiva tecnica sono: *manutenibilità; modularità; deterioramento; obsolescenza.*

FUNCTION POINT:

Il calcolo dei *function point* riguarda la stima dei costi in fase di pianificazione del software. Sono una metrica per la stima della qualità delle funzionalità del software, e una volta calcolati si è in grado di definire la quantità di sforzo richiesto per realizzarlo. I FP sono standardizzati dall'ISO tramite l'uso del metodo UFPM. A partire dai requisiti funzionali si determinano quante sono le funzionalità da realizzare e il peso di ciascuna di esse, ovvero la complessità in base ai dati di input/output e ai dati da produrre e memorizzare. Dapprima si determina il tipo di conteggio (di sviluppo o di manutenzione), poi si determina il confine applicativo per determinare quale sia l'interno e l'esterno dell'applicazione.

Si passa poi all'individuazione delle funzioni dati, e distinguiamo tra:

- *Internal logical file* (**ILF**): aggregati logici di dati generati, gestiti e usati internamente dal sistema (tabelle di database, file di log). Affinché dei file siano tali devono essere identificabili all'utente, dunque i file temporanei non sono ILF. Ad esempio, su Amazon nella gestione del carrello dovremo memorizzare l'acquisto con le sue informazioni (costo, data, quantità ecc.), queste sono dati generati e gestiti internamente al sistema ma visibili al suo utilizzatore;
- *External interface file* (**EIF**): aggregati logici di dati, scambiati con altre applicazioni. Anche questi per essere tali devono essere identificabili dall'utente. Spesso per ottenere l'interoperabilità tra due applicazioni l'una scrive nei dati dell'altra. In generale gli ELF di un'applicazione sono ILF di un'altra.

In seguito, c'è la fase di identificazione delle funzioni transazionali:

- *External input* (**EI**): informazioni distinte fornite dall'utente o da applicazioni esterne, usate come dati di ingresso. Essi alterano gli ILF o agiscono sul funzionamento del sistema. Non sono da considerarsi tali: schemi di menù usati per navigare all'interno dell'applicazione o richieste di input da parte di una query, in quanto non alternano gli ILF.
- *External output* (**EO**): output distinti che il sistema restituisce all'utente come risultato delle proprie elaborazioni. Lo scopo è restituire i risultati provenienti da elaborazioni (e non da semplici recuperi di dati da un ILF), per fare ciò dev'esservi una formula o una funzione di calcolo applicata a un ILF.

- *External Inquiry* (**EQ**): interrogazioni in linea che producono una risposta immediata del sistema, senza necessità di elaborazione. Sono EQ: le query al database, che prelevano dati da un ILF o un EIF. Non sono EQ: le schermate di menù, i messaggi di errore e i messaggi di conferma. Generalmente hanno un peso minore rispetto agli EO proprio per la mancanza di elaborazione sui file.

Per effettuare il conteggio degli UFP contiamo il numero degli ILF, EIF, EI, EO e EQ e li pesiamo per un fattore, associato ad una stima di complessità (semplice, media o complessa). A parità di complessità, un ILF pesa meno di un EQ, conterrà una query che restituisca quel dato e, pur non compiendo un'elaborazione, verrà effettuata almeno la ricerca in un database.

INGEGNERIA DEI REQUISITI:

È una disciplina che cerca di sviluppare metodi standard e sistematici per la gestione dei requisiti, dalla loro nascita alla loro verifica, fino a dopo il rilascio del software. I requisiti sono di fondamentale importanza in ogni fase del ciclo di vita del software e vanno gestiti in ciascuna di esse. Prima di rilasciare il software dobbiamo verificare che esso sia corretto, cioè conforme ai requisiti specificati, che sono dunque necessari in questa fase finale. Per la stessa progettazione dei test abbiamo bisogno dei requisiti: i test vengono pianificati non appena abbiamo specificato i requisiti. Questi infatti, per essere definito "buon requisito", dev'essere facilmente verificabile. Possiamo distinguere tra:

- *Requisiti utente*, che riguardano le funzionalità e i servizi che il software deve offrire ed eventuali vincoli operativi da rispettare. Generalmente sono scritti per i clienti e per l'approvazione del contratto, in un linguaggio naturale o tramite diagrammi, così da essere facilmente comprensibili anche ai "non addetti ai lavori";
- *Requisiti di sistema*, cioè un documento strutturato che fornisce una descrizione dettagliata delle funzionalità e dei servizi del sistema. Sono in parte comuni e in parte diversi dai requisiti utenti. Nei requisiti utente, ad esempio, potrebbero esservi requisiti di interfacciamento, non presenti nei requisiti di sistema.

- *Requisiti funzionali*, riguardano le funzionalità che il software deve offrire e ciò che dovremo verificare in fase di testing, pertanto sono espressi mediante forme verbali come "deve".
- *Requisiti non funzionali*, impongono vincoli o obbiettivi di qualità, riguardo: tempi di risposta (sistemi real-time), uso delle risorse (efficienza e prestazioni), affidabilità, manutenibilità, etc. Inoltre, possono riguardare il prodotto (efficienza, portabilità, security, reliability), il processo e possono essere requisiti esterni (interoperabilità), oppure requisiti etici.

L'IdR è divisa in varie fasi:

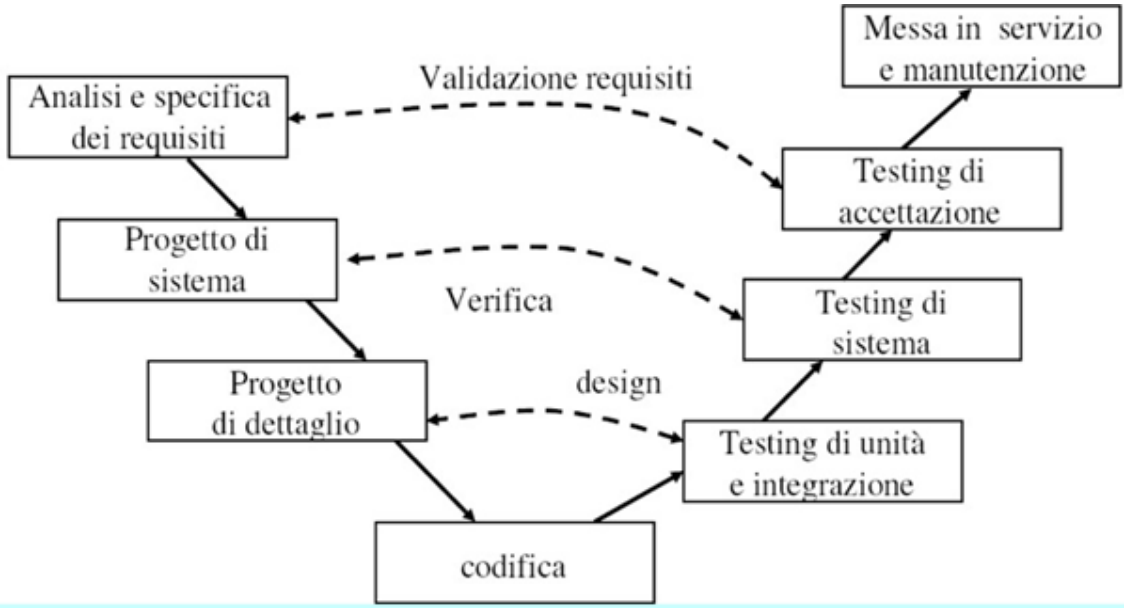
1) *Esplicitazione*, acquisizione dei requisiti;
2) *Analisi e negoziazione*, valutazione dei requisiti rispetto a completezza, conflittualità e compatibilità;
3) *Documentazione*, viene prodotto un documento tecnico (SRS) che descrive funzioni, prestazioni e vincoli del sistema;
4) *Validazione*, l'SRS viene passato al progettista che elabora una soluzione adeguata;
5) *Gestione:* tipicamente i requisiti sono soggetti a cambiamenti, che devono essere gestiti, non modificando semplicemente il documento SRS, ma anche considerando un *'analisi d'impatto'*, ovvero la stima degli impatti del cambiamento sui requisiti e sulle attività, possibile grazie alla *tracciabilità*. L'impatto di un cambiamento si può limitare con una buona modularità, difatti se un utente cambia un requisito ed è necessario modificare un'intera parte del progetto significa che la progettazione non è stata realizzata correttamente.

MODELLI DEI CICLI DI VITA DEL SOFTWARE:

Nella produzione di un sistema software sono coinvolte molte attività, come la fase di specifica, di progettazione, di codifica e di verifica. L'ordine in cui queste vengono affrontate definisce il ciclo di vita del software. Il più semplice è il *Code&Fix*, adatto solo a progetti piccoli e non a progetti di elevata complessità, in quanto non è ripetibile e strutturato (modulare).

1. **Com'è fatto il 'Modello a V' ?**

   E' una variante del modello a cascata, in cui tutte le attività del ramo di sinistra sono collegate con quelle del ramo a destra: durante le attività di sinistra, vengono progettati i test della fase a destra corrispondente (ad esempio dalla specifica dei requisiti corrisponde la progettazione dei test di accettazione). Se si trova un errore in una fase a destra (per esempio nel testing di sistema) si riesegue la fase a sinistra collegata.



   Nello sviluppo di sistemi HW/SW (Hardware/Software), si prevede l'utilizzo di un modello a V: dapprima esaminando i requisiti utente e di sistema (sia funzionali che non), poi identificando le unità del sistema assegnando requisiti alle unità (allocazione dei requisiti).

   In generale il modello a cascata è molto comprensibile e controllabile data la sua ampia documentazione ed è adatto ai contratti. D'altro canto, ha una difficile gestione per i requisiti variabili o incompleti, ritarda l'individuazione di problemi e ha una basta tempestività, che porta ad una eccessiva distanza tra specifica e consegna.

2. Com’è fatto il modello a spirale?

È un *meta-modello* iterativo che permette di scegliere in corso d’opera il modello generico da utilizzare. Il suo obiettivo è quello di fornire un quadro di riferimento per la progettazione dei processi. Consente di scegliere il modello più appropriato in funzione del livello di rischio. Il rischio è visto come una circostanza potenzialmente avversa in grado di pregiudicare il processo di sviluppo e la qualità del prodotto. Per gestire i rischi bisogna identificarli, affrontarli ed eliminarli prima che insorgano problemi seri o causa di re-implementazioni costose.

Su in piano cartesiano immaginiamo una spirale, e ogni giro della spirale rappresenta una fase del processo, ed il raggio del cerchio è grande in proporzione al costo. Quando si attraversa un settore, si cambia fase:

I. Definizione di obiettivi, vincoli e piano di gestione della fase.
II. Si analizzano i rischi della fase e si scelgono le attività necessarie a gestirli, ad esempio tramite simulazione o prototipazione.
III. Si sceglie un modello di sviluppo per il sistema tra i modelli generici. Si possono utilizzare:
    a. *Modello evolutivo*, se i requisiti sono incerti;
    b. *A cascata* se i requisiti sono chiari e ben definiti;
    c. *Trasformazionale* se la sicurezza è un requisito più importante
IV. Revisione dei risultati e pianificazione della prossima iterazione della spirale.

3. Cos’è un modello incrementale?

È una particolarizzazione del modello evolutivo che si applica durante la fase di implementazione. Vengono codificati i rilasci incrementali, che sono delle unità software indipendenti (moduli), ognuno dei quali è un’aggiunta al software di base e tiene conto dei feedback per l’affinamento dei requisiti. In questo modello scompare la fase di manutenzione, poiché la stessa fase è vista come un incremento. Gli incrementi vengono fatti per convenienza, perché ogni volta che modifico il software è come se aggiungessi nuove funzionalità, indipendenti dalle altre. Viene utilizzato quando i requisiti sono poco chiari e possono variare nel tempo. Un modello evolutivo è un modello prototipale che porta progressivamente al prodotto finale.

Il modello incrementale differisce da uno evolutivo in quanto nel secondo l'approccio incrementale avviene solo dalla fase di implementazione e rilascio in seguito ad un processo a cascata. Invece nel primo l'approccio incrementale è esteso a tutte le fasi del ciclo di sviluppo

4. **Modelli prototipali:**
   Oltre al prototipo evolutivo, esiste anche il prototipo '*usa e getta*': la prima versione dell'applicazione, cioè il prototipo, viene cestinata e la seconda versione poi può essere sviluppata seguendo un modello a cascata.
   In generale il modello prototipale/evolutivo è vantaggioso per l'utilizzo dei feedback, per una gestione efficiente della variabilità dei requisiti ed è molto tempestivo. Tuttavia, necessita elevate competenze sia nello sviluppo che nella gestione dei rischi.

5. **Metodologie agili: vantaggi e svantaggi**
   Le metodologie agili sono un sottoinsieme dei modelli evolutivi, nati come modelli di sviluppo lightweight: si cerca di limitare l'attività di pianificazione del processo, scrivendo e documentando il meno possibile e sviluppando il più rapidamente possibile, anche per piccoli incrementi. Sono metodologie people-oriented anziché process-oriented.
   Si comincia a lavorare ad un prodotto subito dopo essere stato commissionato, si lavora nel proprio laboratorio e poi il software viene messo in esecuzione sul campo.
   Ambiente di sviluppo e ambiente di esercizio sono momenti e luoghi differenti. Ad esempio, Netflix ha rimosso l'ambiente di prova, in quanto il software viene messo direttamente in esercizio: in questo modo si riducono i tempi e dallo sviluppo si passa direttamente all'esercizio.
   A volte ci sono addirittura più rilasci in un giorno, e in questo caso le finestre temporali sono ore. Non c'è nemmeno il tempo di effettuare pianificazione o documentazione, poiché dai requisiti si passa direttamente alla stesura di codice.
   Alcune delle più famose sono: *XP*, *Scrum* *e DevOPS*.
   In definitiva queste metodologie sono adatte a progetti innovativi o con requisiti non ben definiti, ma a causa della poca strutturazione, necessitano di una grande abilità di programmazione e pochi membri per

team di sviluppo. Non sono molto adeguate per i contratti, poiché i clienti non possono partecipare a tempo pieno allo sviluppo. E inoltre può richiedere del lavoro extra per mantenere la semplicità del sistema.

6. **XP (Extreme programming):**
   È la metodologia agile più diffusa. Si sviluppa per iterazioni molto veloci, che rilasciano piccoli incrementi, fatti con il '*pair programming'*. Il committente partecipa attivamente al team di sviluppo, il software viene testato continuamente per verificare cosa funziona e cosa no. È detto anche '*Test Driven Development*' (TDD), poiché lo sviluppo è interamente guidato dal testing. Inoltre, il software è modificabile da chiunque in qualunque punto del codice (*collective ownership*).

7. **SCRUM:**
   Significa letteralmente "mischia", ed è un modello iterativo ed incrementale, adottato negli anni '90, per lo sviluppo e gestione di ogni tipologia di prodotto. È suddiviso in tre fasi:
   - *Il pregame*: si fa il planning e si stabilisce l'architettura del rilascio, in maniera rapida;
   - *Development* (game): si sviluppa tramite piccole applicazioni, dette *sprint*;
   - *Il Post-game*: contiene la chiusura definitiva della release.

   Lo *sprint* è un ciclo iterativo in cui vengono sviluppate o migliorate una serie di funzionalità. Ogni *sprint* include le tradizionali fasi di sviluppo del software e può durare da una settimana ad un mese. Ognuno di essi è gestito da uno Scrum master (gestore). Esiste anche il ruolo del *product owner*, che rappresenta gli stakeholders, e infine c'è il team, ovvero il gruppo che esegue analisi, la progettazione, l'implementazione e il test. Giornalmente si fanno dei meeting che durano al massimo un quarto d'ora e c'è un backlog, una coda di cose da svolgere che viene assegnata, sequenzialmente o secondo altri criteri, a dei team che fanno gli sprint e vengono monitorati ogni 24 ore. Ogni sprint non deve durare più di 30 giorni.

8. **DevOPS:**

   È la frontiera più recente, sta per *development operations*. Fa parte delle metodologie evolutive, più che delle agili. Insieme di pratiche per colmare il gap tra lo sviluppo (agile) del software e la fase operazionale (IT operations). L'idea è che i team di sviluppo e di IT operations (che si preoccupano di garantire l'operatività del software dopo il rilascio) collaborino durante tutto il ciclo di vita, dal design al processo di sviluppo fino al supporto alla produzione. I principi fondamentali sono: il *system thinking,* l'enfasi sui feedback, CAMS (culture-automation-measurement-sharing): enfasi alla comunicazione e cooperazione, automatic build e test, supporto alla pianificazione e all'analisi dei trend, collaborazione e feedback.

   C'è continua interazione, sviluppo e rilascio. Oggi è molto utilizzato per le tecnologie in *cloud*, con *containers* o per la *virtualizzazione*. Su un nodo possiamo mettere più macchine virtuali in cui viene confinato un singolo software utente. Di conseguenza, se un software fallisce, è la macchina virtuale a fallire ma non l'intero nodo, non intaccando l'operato delle altre VM sul nodo. Un altro vantaggio è la sicurezza: bisogna garantire che i dati immessi da un utente non vengano visti dagli altri clienti del nodo (sfruttando sempre la virtualizzazione e i containers).

PRINCIPI DELL'INGEGNERIA DEL SOFTWARE:

Sono necessari, ma non sufficienti, a garantire la qualità di un software, e descrivono delle proprietà desiderabili di processi e prodotti in termini astratti. Sono diversi i principi cardine, tra questi compaiono:

1) *Rigore*: necessario a strutturare le attività di sviluppo. Il livello più alto di rigore è la *Formalità*, in cui si viene guidati da procedimenti matematici e che favorisce l'automazione. Favoriscono *affidabilità* e *verificabilità*, ma anche *manutenibilità* e *riusabilità* grazie ad una documentazione minuziosa.
2) *Separazione degli interessi*: permette di affrontare aspetti diversi del problema, focalizzandosi separatamente su ciascuno di essi. La separazione avviene in modo: temporale (ciclo di vita); dei fattori di qualità (progettare prima per assicurare la *correttezza* e poi verificare l'*efficienza*); di viste diverse (*data-flow e control-flow*); di parti diverse del sistema (con l'uso di moduli e sottoinsiemi); di domini (dell'implementazione, del problema e della soluzione).
3) *Differimento delle decisioni*: ogni decisione va presa al momento giusto, senza anticipare il momento della decisione rispetto a quando prenderla è effettivamente improcrastinabile. Ad esempio, la scelta del linguaggio di programmazione non va anticipata alla fase di analisi, così come la scelta degli strumenti di sviluppo o dell'architettura.
4) *Astrazione*: permette di identificare proprietà rilevanti di un'entità e ignorare le inessenziali, per definire poi una (o più) *Vista* dell'entità.
5) *Modularità*: (più importante) è la suddivisione in moduli di un sistema, in modo tale da farlo risultare più semplice da comprendere, manipolare e progettare. Successivamente al trattare i dettagli dei singoli moduli, si può poi studiare l'interoperabilità tra essi. Ad esempio, il sistema 'automobile' è formato dai moduli: motore, carrozzeria, trasmissione, etc.
6) *Anticipazione al cambiamento*: è la capacità di far evolvere il prodotto in seguito a cambiamenti di vario genere, ed è realizzata attraverso una corretta Modularizzazione.
7) *Generalità*: valuta la risolvibilità di un problema in modo generale per favorirne il riuso, ma spesso aumenta i costi.

8) *Incrementalità*: raggiungere un obbiettivo attraverso una serie di passaggi incrementali evolutivi, anche per un miglioramento dei fattori di qualità.

1. Modularità di un software:

Un modulo di un sistema software è un componente che realizza una astrazione ed è dotato di netta separazione tra Interfaccia, che specifica cosa fa, e Corpo, che descrive come viene realizzata. Grazie ad essa viene favorito anche l'incapsulamento e l'*information hiding*.

Le astrazioni più tipicamente realizzate da un modulo sono:

- *Sul controllo*: Consiste nell'astrarre una data funzionalità dai dettagli della sua implementazione. È ben supportata dai linguaggi di programmazione tradizionali tramite il concetto di sottoprogramma.
- *Sui dati*: Consiste nell'astrarre gli oggetti costituenti il sistema, descritti da una struttura dati e dalle operazioni possibili su di esso. Può essere realizzata con un uso opportuno delle tecniche di programmazione modulare nei linguaggi di programmazione, nei quali è supportata da appositi costrutti di programmazione ad oggetti.

Le tipologie di astrazioni danno luogo a diverse categorie di moduli:

- *Funzione*: Astrazione procedurale (sul controllo) che fornisce una funzione che implementa delle operazioni, in cui viene incapsulato un algoritmo;
- *Libreria*: un gruppo di astrazioni procedurali correlate, cioè un insieme di funzioni;
- *Oggetti*: Astrazione sui dati, che hanno una struttura dati interna permanente, visibile solo alle funzioni interne, e che fornisce uno stato del modulo;
- *Tipi di dati astratti*: oltre alle operazioni sui dati, esporta un *tipo* e le derivazioni del tipo sono *oggetti astratti*;
- *Pool*: di dati comuni. È di basso livello e non fornisce astrazione;
- *Moduli generici*: sono template di moduli parametrizzati rispetto ad un tipo, che devono essere istanziati con parametri reali prima di essere utilizzati.

La modularità favorisce il *dominio sulla complessità* di un sistema, *separazione degli interessi*, *suddivisione del lavoro* (anche in parallelo), *composizione* di un sistema(nel caso di un approccio di progettazione bottom-up), *riuso*, *analisi* e *manutenzione*.
I moduli devono avere un'*alta coesione*, se gli elementi sono raggruppati per un motivo logico e cooperativo, ed un *basso accoppiamento*, misura l'indipendenza tra i moduli.
La progettazione modulare ha due approcci:

1) *Top-down*: è basata su un approccio di *decomposizione funzionale* nella definizione del sistema software, cioè sull'individuazione delle funzionalità del sistema da realizzare e su raffinamenti successivi, da iterare finché la scomposizione del sistema individua sottosistemi di complessità accettabile.
2) *Bottom-up*: è basata sull'individuazione delle entità (classi e/o oggetti) facenti parte del sistema, delle loro proprietà e delle interrelazioni tra di esse. Il sistema viene costruito assemblando componenti con un approccio "dal basso verso l'alto".

Anche le relazioni tra i moduli sono categorizzate:

- *Uses* : un modulo usa un altro per portare a termine il suo compito;
- *IsComponentOf* : un modulo fa parte insieme ad altri di un modulo più grande;
- *Comprises* : A comprende B se e solo se B fa parte di A
- *IsComposedOf*: relazione allargata di *IsComponent*, e l'inverso è *Implements*

METRICHE DEL SOFTWARE:

La gestione di un progetto software richiede di monitorarne l'avanzamento e di valutarne quantitativamente caratteristiche relative non solo al prodotto finale, ma anche ad artefatti intermedi, anche al fine di decidere eventuali correttivi da apportare al processo di produzione. Si necessitano dunque di tecniche per "misurare" opportune grandezze, al fine di rispondere a domande quali: Ci sono ritardi nel progetto? Sono state consumate più risorse del previsto(in particolare il tempo)? Il software prodotto rispetta i requisiti di qualità?

Una *metrica* caratterizza in modo quantitativo e misurabile attributi semplici di una entità e sono lo strumento attraverso cui un ingegnere predice o misura i processi e le risorse. Servono inoltre: da guida nella ricerca di soluzioni efficienti ed efficaci a problemi complessi; a pianificare l'andamento di un progetto; alla valutazione della qualità; alla stima dello sforzo richiesto per lo sviluppo. Sono di diverso tipo: *di prodotto* (dimensionali (LoC), funzionali (FP), di complessità (McCabe)), *di processo* (modularità, gestione, ciclo di vita), *di qualità* (dependability (MTTF, MTBF), manutenibilità, usabilità), *object-oriented* (struttura interna, complessità esterna, strutturali, di pianificazione).

Una *misura* è una assegnazione oggettiva ed empirica di un valore ad un attributo.

Gli attributi possono essere *esterni*, di interesse per l'utente (facilita d'uso, efficienza, prestazioni), o *interni*, di interesse per gli sviluppatori (modularità, complessità). Se le metriche sono associate ad attributi interni, gli attributi esterni dipendono da quelli interni e sono più complessi.

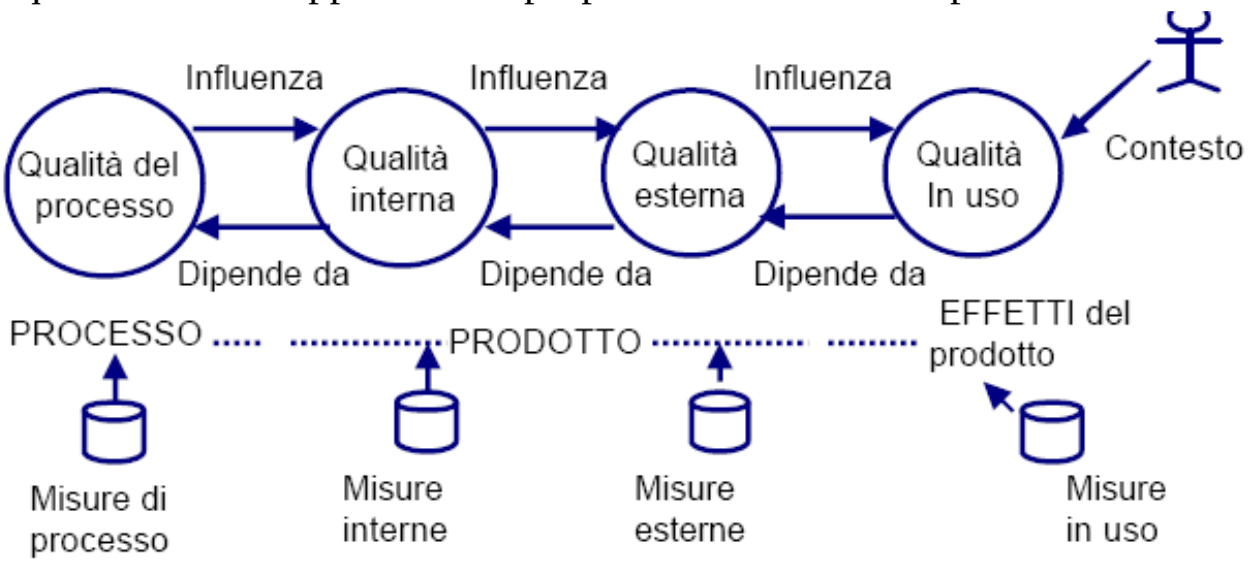
FATTORI DI QUALITÀ DI UN SOFTWARE:

Sono suddivisi in base a tre punti di vista: d'uso(percepiti), interni ed esterni.

La qualità in uso riguarda il livello con cui il prodotto si dimostra utile all'utente nel suo effettivo contesto di utilizzo.

La qualità esterna è quella rappresentata dalle prestazioni del prodotto e dalle funzionalità che offre.

La qualità interna rappresenta le proprietà intrinseche del prodotto.



Le caratteristiche esterne sono:

1) *Funzionalità*: il software fa ciò per cui è stato progettato? È la capacità di fornire servizi tali da soddisfare, in determinate condizioni, requisiti funzionali espliciti o impliciti. Le sottocaratteristiche correlate sono:
    a. *Adeguatezza*: presenza di funzioni appropriate per compiti specifici che supportano obiettivi dell'utente.
    b. *Accuratezza*: capacità di fornire risultati corretti in accordo con i requisiti dati dall'utente.
    c. *Interoperabilità*: capacità di interagire con altri sistemi.
    d. *Sicurezza*: capacità di proteggere programmi e dati da accessi non autorizzati e di consentire quelli autorizzati.

2) *Affidabilità*: il software reagisce bene a variazioni esterne? È la capacità di mantenere le prestazioni stabilite nelle condizioni e nei tempi fissati. Le sottocaratteristiche correlate sono:
    a. *Maturità* (robustezza): capacità di evitare malfunzionamenti della applicazione (failure) a seguito di condizioni di guasto (error) dovute ad errori (fault).
    b. *Tolleranza*: capacità di mantenere determinati livelli di prestazione in caso di malfunzionamenti.
    c. *Recuperabilità*: capacità e velocità, in caso di malfunzionamenti, di ripristinare dei livelli di prestazione predeterminati e di recuperare i dati.

3) *Efficienza*: il software usa bene le risorse disponibili? È il rapporto fra prestazioni e quantità di risorse utilizzate, in condizioni definite di funzionamento. Le sottocaratteristiche correlate sono:
    a. *Comportamento rispetto al tempo*: adeguati tempi di risposta, di elaborazione e throughput, per eseguire le funzioni richieste.
    b. *Uso di risorse*: utilizzo di una quantità e di una tipologia di risorse adeguate, per eseguire le funzioni richieste.

4) *Usabilità*: è *user-frendly*? È la capacità di essere compreso e usato con soddisfazione dall'utente in determinate condizioni d'uso. Le sottocaratteristiche correlate sono:
    a. *Comprensibilità*: capacità di ridurre l'impegno richiesto agli utenti per capirne il funzionamento.
    b. *Apprendibilità*: ridurre l'impegno richiesto per impararlo ad usarlo.

5) *Manutenibilità*: Quanto è facile modificare il software? È la capacità di essere modificato con un impegno contenuto. Le sottocaratteristiche correlate sono:
    a. *Analizzabilità:* capacità di limitare l'impegno richiesto per diagnosticare cause di malfunzionamenti, o per identificare parti da modificare.
    b. *Modificabilità*: capacità di limitare l'impegno richiesto per modificare, rimuovere errori o sostituire componenti.

c. *Stabilità*: capacità di ridurre il rischio di comportamenti inaspettati a seguito della effettuazione di modifiche.
d. *Testabilità*: capacità di essere facilmente testato per validare le modifiche apportate.

6) *Portabilità*: Quanto è facile trasferire il software su un'altra piattaforma? Le sottocaratteristiche correlate sono:
   a. *Adattabilità*: capacità di adattarsi a nuovi ambienti operativi limitando la necessità di apportare modifiche.
   b. *Installabilità*: capacità di ridurre l'impegno richiesto per installarlo in un particolare ambiente operativo.
   c. *Coesistenza*: capacità di coesistere con altri software nel medesimo ambiente, condividendo risorse.
   d. *Sostituibilità*: capacità di essere utilizzato al posto di un altro software per gli stessi compiti nello stesso ambiente.