RIASSUNTO ORALE DI INGEGNERIA DEL SOFTWARE

Ingegneria del Software

Università degli Studi di Napoli Federico II (UNINA)

47 pag.

Fattori di qualità del software:

Distinguiamo qualità interne ed esterne. Le qualità esterne riguardano il prodotto software, visto come una black-box, sono le qualità visibili agli utenti del sistema; le qualità interne riguardano gli sviluppatori, che guardano al software come a una white box.

Oltre a questa prima distinzione vi è una classificazione tra qualità del prodotto e qualità del processo. Le qualità interne influiscono su quelle esterne, o meglio permettono agli sviluppatori di raggiungere le qualità esterne. Allo stesso modo la qualità del processo, che comprende anche la conoscenza e le esperienze delle persone che lavorano al progetto, influisce sulla qualità del prodotto finale.

• **Correttezza**: un software è corretto se soddisfa i requisiti funzionali (ovvero le specifiche). Questa definizione assume che le specifiche siano disponibili e non ambigue. La correttezza è una proprietà matematica che stabilisce l'equivalenza tra il software e la sua specifica. Ovviamente possiamo essere tanto più sistematici e precisi nel valutare la correttezza quanto più rigorosi siamo stati nello specificare i requisiti funzionali. Possiamo verificarla attraverso il testing, mediante un approccio analitico o la verifica formale.

• **Affidabilità**: è la probabilità che un software operi come atteso in un intervallo di tempo determinato. La correttezza è una qualità assoluta, un qualunque scostamento da ciò che è stato stabilito nelle specifiche risulta non corretto; l'affidabilità è invece una qualità relativa, gli scostamenti sono tollerabili. La relazione per cui la correttezza è inclusa nell'affidabilità in realtà è valida sotto l'ipotesi che la specifica dei requisiti funzionali colga esattamente tutte le proprietà desiderate dall'utente di un'applicazione, senza contenere erroneamente proprietà indesiderabili. Nella pratica non è così: il software può essere corretto, e dunque soddisfare i requisiti funzionali espressi dalle specifiche, senza soddisfare i bisogni dell'utilizzatore. Possono esservi applicazioni corrette sviluppate sulla base di requisiti non corretti. Questo è dovuto al fatto che l'utente può aspettarsi che il software operi in maniera diversa da quanto non accada, sebbene il software verifichi le specifiche. (Anche in fase di testing si distingue la verifica, ovvero il controllo della corrispondenza con le specifiche, dalla validazione, dove si controlla invece che il prodotto rispetti le aspettative dell'utente). Una metrica per l'affidabilità è la **MTBF** (mean time between failures), l'intervallo medio di tempo tra i fallimenti. Questa tende a diminuire nel tempo. Un altro parametro importante è la **MTTF** (mean time to failure), intervallo di tempo per il primo fallimento

• **Robustezza**: un software è robusto se si comporta in maniera accettabile anche in situazioni non specificate nei requisiti, come input non corretti o non attesi, malfunzionamenti hardware. Un software può essere non robusto sebbene sia corretto, se la specifica non indica cosa il programma dovrebbe fare in caso di input scorretto. Se potessimo definire esattamente cosa occorrerebbe fare per rendere un'applicazione effettivamente robusta saremmo in grado di specificarlo, dunque la robustezza diventerebbe equivalente alla correttezza. Nel momento in cui un requisito diventa parte della specifica, il suo soddisfacimento è inerente alla proprietà di correttezza; se il requisito non fa parte delle specifiche allora sarà un problema di robustezza. La linea di demarcazione tra le due qualità è la specifica.

• **Prestazioni:** è una qualità esterna basata sui requisiti dell'utente e sull'utilizzo efficiente delle risorse. Differisce dall'efficienza in quanto quest'ultima è una qualità interna, si riferisce all'adeguato uso delle risorse da parte del software. L'efficienza influenza e spesso determina le prestazioni. Per valutare le prestazioni di un algoritmo utilizziamo la teoria computazione. Per valutazioni più specifiche possiamo effettuare misurazioni su sistemi reali, costruire modelli analitici o costruire un modello simulativo (queste tecniche possono anche essere complementari).

• **Usabilità**: un sistema è usabile (user friendly) se i suoi utenti lo reputano facile da utilizzare. Dalla definizione stessa si evince la natura soggettiva di tale proprietà. L'interfaccia utente influisce molto sull'

“amichevolezza” di un’applicazione. In generale l’usabilità è studiata dalla disciplina scientifica che si occupa del “fattore umano” (human factor), in informatica HCI (human-computer interaction). In molte discipline l’usabilità si ottiene standardizzando le interfacce uomo-macchina, così che un utente che abbia imparato a far funzionare un apparecchio sia in grado di far funzionare gli apparecchi analoghi, anche di altri produttori.

• **Manutenibilità:** è la facilità con cui le attività di manutenzione vengono eseguite e l’economicità dei relativi processi. Solitamente la manutenzione supera il 60% dei costi totali del software. La manutenzione può essere di tre tipi: correttiva, adattativa e perfettiva:

1. La **manutenzione correttiva** riguarda la rimozione di errori residui presenti nel prodotto al momento del rilascio (o eventualmente l’eliminazione di errori introdotti nel software durante l’attività di manutenzione stessa).
2. La **manutenzione adattativa** riguarda le modifiche dell’applicazione in risposta a cambiamenti dell’ambiente (hardware, sistema operativo, DBMS). Non è una necessità nata dal software stesso ma dall’ambiente in cui esso è inserito.
3. La **manutenzione perfettiva** riguarda i cambiamenti nel software per migliorare alcune qualità. La richiesta può provenire dagli sviluppatori, con l’obiettivo di migliorarne la presenza sul mercato, o dal committente, al fine di rispondere a nuovi requisiti.

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: la corretta strutturazione in moduli favorisce la riparabilità. Un software ha evolvibilità 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 (con le modifiche aumenta il rischio di introdurre nuovi errori).

1. **Riusabilità**: è la possibilità di poter riutilizzare delle parti del software. È una delle maggiori finalità della programmazione object-oriented. È una proprietà applicabile non solo ai componenti software ma anche alle parti del processo, può essere riutilizzato qualunque artefatto intermedio, ma può essere riutilizzato anche il processo per costruire diversi prodotti (metodologie del processo).
2. **Portabilità**: un software è portabile se può essere eseguito in ambienti diversi. Può essere ottenuta modularizzando il software, in modo tale che le dipendenze dall’ambiente vengano isolate in pochi moduli, modificabili in caso di trasporto del software in ambiente differente.
3. **Comprensibilità**: il software dev’essere comprensibile al fine di verificarne la correttezza, apportare modifiche e riusarlo. Questa dipende sia da come il software è progettato sia dal problema affrontato dal software (più il problema è semplice più facile risulterà produrre un software comprensibile).

• **Dependability**: per i sistemi critici si enfatizzano anche altri attributi di qualità, la dependability è cosistuita da:
Disponibilità: è la probabilità che il software operi come atteso in un dato istante di tempo.
Manutenibilità
Safety: ovvero l’assenza di conseguenze catastrofiche su utenti e ambiente.
Integrità: ovvero l’assenza di alterazioni improprie.

Un sistema safety critical non deve creare danni catastrofici all’ambente e alle persone circostanti. Abbiamo visto che il software è un prodotto ingegneristico particolare, in primo luogo perché è immateriale. Data la

sua immaterialità un software di per sé non potrebbe danneggiare l'ambiente o gli utenti ma nell'ambito di un sistema critico è diverso, basti pensare alle centrali nucleari. Nei sistemi critici alcune qualità del prodotto, tra cui proprio la safety, sono garantite a partire dalla qualità del processo: il rispetto dei vincoli sul processo aumenta la confidenzialità con le qualità del prodotto. È difficile definire degli standard per la qualità del processo, ve ne sono in alcuni campi, ad esempio, nei sistemi ferroviari, avionici o automobilistici. Gli standard sul processo possono prescrivere cosa debba essere prodotto nel suo corso, in che modo vadano prodotti i documenti di SRS o i test. Possono inoltre dettare regole di programmazione, da osservare in fase di implementazione. La programmazione strutturale impone regole: i blocchi devono avere un solo punto d'entrata ed un solo punto d'uscita (paradigma one in one out), non bisogna fare uso di break o di go to. Altre regole prescrivono che le variabili vengano inizializzate, di non basarsi sull'ordine di precedenza degli operatori (che potrebbe variare al variare dei contenuti) e di disattivare eventuali ottimizzazioni automatiche del compilatore, che potrebbero portare a errori molto gravi, in particolare nei sistemi critici. Un altro vincolo potrebbe riguardare la competenza del personale, esclusivamente dotato di alta qualificazione o formazione.

• **Sicurezza**: composta da disponibilità, integrità e confidenzialità, ovvero l'assenza di diffusione non autorizzata di informazioni.
Non c'è un completo accordo sulle definizioni degli attributi delle qualità del software.

Principi dell'ingegneria del software:

I principi che studiamo sono sufficientemente generali da essere applicabili lungo l'intero processo di costruzione del software ma non sono sufficienti a guidare lo sviluppo del software (sono necessari ma non sufficienti a garantirne la qualità). Descrivono proprietà desiderabili del processo e dei prodotti in termini astratti. Per poterli applicare dobbiamo disporre di metodi e tecniche che incorporino le proprietà desiderate nei processi e nei prodotti. L'individuazione dei metodi e delle tecniche da utilizzare è lo scopo di una metodologia. Infine, per aiutare l'applicazione dei metodi e delle tecniche ci serviamo di strumenti. La metodologia promuove un certo approccio alla soluzione di un problema, valutando alternative e scegliendo tra di esse la più opportuna.

• **Rigore:** lo sviluppo del software è un'attività creativa, un complemento dev'essere il rigore: attraverso un approccio rigoroso possiamo realizzare prodotti affidabili, controllandone il costo. Paradossalmente il rigore è un concetto intuitivo che non può essere definito in maniera rigorosa ma è necessario a strutturare le attività di sviluppo, fornendo precisione e accuratezza. La progettazione procede come una sequenza di passi ben definiti e specificati, ad ogni passo corrispondono metodi e tecniche secondo un approccio rigoroso e sistematico: la metodologia. Esistono diversi livelli di rigore, il più alto è la formalità: richiede che il processo di sviluppo del software sia guidato e valutato mediante leggi matematiche. Non sempre conviene essere formali in quanto è qualcosa di estremamente oneroso. È formale qualcosa di matematicamente trattabile: non ambiguo e dal significato univoco. Possiamo quindi automatizzare qualcosa di formale. Il processo di produzione del software ci porta da qualcosa di estremamente informale (l'esigenza iniziale) a qualcosa di estremamente formale: la soluzione. Ciò avviene attraverso le varie fasi di produzione (analisi, progettazione, codifica, test). Il vantaggio della formalità è che essa può essere alla base dei processi automatizzati. La fase in cui tradizionalmente si applica l'approccio formale è la fase di programmazione: i programmi sono oggetti formali. Rigore e formalità favoriscono affidabilità e verificabilità ma hanno effetti benefici anche su manutenibilità, riusabilità ecc… la documentazione rigorosa aiuta nel processo di manutenzione.

• **Separazione degli interessi:** ci consente di affrontare differenti aspetti del problema, concentrando la nostra attenzione su ciascuno di essi in maniera separata. Ciò è essenziale per dominare la complessità: isoliamo gli aspetti scarsamente correlati tra loro e, successivamente, consideriamo ciascun aspetto separatamente, approfondendo i dettagli importanti per il suo trattamento. Gli aspetti possono essere separati in differenti modi:

- ▪ Separazione temporale: nello stesso ciclo del software si separano diverse attività in diversi periodi temporali (ciclo di vita).
- ▪ Separazione di fattori di qualità: separare, ad esempio, correttezza ed efficienza, progettando prima per assicurare la correttezza e poi focalizzandosi sull'efficienza
- ▪ Separazione di viste diverse: ad esempio, quando analizziamo i requisiti di un'applicazione, può essere utile concentrarsi separatamente sul flusso di dati da un'attività all'altra all'interno del sistema e sul flusso di controllo che governa il modo con il quale le diverse attività sono sincronizzate.
- ▪ Separazione di parti del sistema: affrontiamo le parti del sistema separatamente
- ▪ Separazione di domini: è fondamentale distinguere gli aspetti relativi al dominio del problema da quelli relativi al dominio dell'implementazione. Le proprietà specifiche del dominio del problema valgono indipendentemente dagli aspetti implementativi.

• **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 le scelte di progetto (architettura software).

• **Astrazione:** ci consente di identificare gli aspetti fondamentali di un fenomeno, trascurando i suoi dettagli. È un caso particolare della separazione degli interessi,11 ci permette di separare gli aspetti importanti da quelli che contengono dettagli secondari. A seconda dello scopo una stessa realtà di interesse può essere modellata da molteplici astrazioni, ciascuna delle quali fornisce uno specifico punto di vista, per uno specifico scopo. Ciascun modello usato nell'ingegneria per descrivere i fenomeni è un'astrazione della realtà. Nell'esprimere i requisiti stessi forniamo un modello che astrae da numerosi dettagli che i progettisti decidono di poter ignorare senza conseguenze.

• **Anticipazione del cambiamento**: i cambiamenti del software sono dovuti alla capacità di riparare il software e alla necessità di supportare l'evoluzione dell'applicazione. La capacità del software di evolvere deve essere pianificata e anticipata con estrema cura. I progettisti devono riuscire in qualche modo a prevedere i cambiamenti e pianificare il progetto così da renderli agevoli. I probabili cambiamenti devono essere attribuiti a specifiche porzioni del software e il loro effetto circoscritto a esse: l'anticipazione del cambiamento dev'essere alla base della strategia di modularizzazione. Anche la riusabilità è favorita da questo principio, un componente è facilmente riusabile se deve essere sottoposto solo a limitati cambiamenti per poter essere riusato, dobbiamo quindi progettare il componente in modo che questi cambiamenti possano essere facilmente ottenuti. Questo principio viene applicato anche al processo: un cambiamento è ad esempio il ricambio di personale.

• **Modularità**: è la suddivisione/organizzazione di un modello o di un sistema in parti (moduli), in modo che esso risulti più semplice da comprendere e manipolare. Il vantaggio principale è che in un primo momento ci consente di trattare i dettagli di un singolo modulo separatamente, successivamente possiamo esaminare i moduli e le loro relazioni, in modo da integrarli in un sistema coerente.

Un modulo è un componente del sistema software che realizza un'astrazione: abbiamo due tipi di astrazione (sul controllo e sui dati).

L'astrazione sul controllo avviene mediante il meccanismo dei sottoprogrammi. Nei sottoprogrammi si affronta un problema e quando si ritorna al programma chiamante abbiamo un problema minore: non dobbiamo ragionare su come è implementato il problema risolto dal sottoprogramma. Il modo con cui il modulo interagisce con il resto è attraverso l'interfaccia. Nel caso dei sottoprogrammi l'interfaccia è il prototipo ed esprime cosa fa il modulo. Il corpo è il codice implementativo, che descrive come lo fa. Attraverso il meccanismo dell'information hiding nascondiamo nel corpo l'algoritmo, il procedimento con cui realizziamo ciò che si fa. Per far funzionare il sottoprogramma gli passiamo dei dati (il sottoprogramma ha i suoi dati locali).

L'astrazione sui dati ci permette di astrarre le entità costituenti il sistema, descritte in termini di una struttura dati e delle operazioni possibili su di esse. Un'informazione è descritta in generale dalla tripla tipo-valore-attributo, dove il tipo identifica il particolare insieme entro cui si effettua la scelta, il valore la scelta effettuata e l'attributo è un identificato che attribuisce un significato. In realtà il tipo non è solo un insieme di valori, è anche l'insieme delle operazioni che possiamo effettuare su di essi: ciò definisce il tipo di dato astratto (ad esempio, se consideriamo il tipo int e il tipo string è ovvio che alcune operazioni, come la somma o il prodotto, definite per il tipo int non avrebbero senso per il tipo string: la struttura dati concreta è incapsulata nelle operazioni su di essa consentite). Un oggetto è una struttura dati unita alle operazioni possibili su di essa Se abbiamo necessita di utilizzare più oggetti definiremo allora un tipo (in quanto vogliamo una molteplicità di istanze): in questo caso l'oggetto sarà un'istanza del tipo di appartenenza. Un modo per realizzare un tipo di dato astratto è la classe, altri sono costrutti come il typedef e lo struct. La

differenza è che utilizzando typedef/struct adottiamo una disciplina di programmazione, eventuali comportamenti illeciti saranno noti solo a tempo di esecuzione (non è appropriato per la progettazione di un software). Attraverso la classe invece obblighiamo l'utente ad accedere all'oggetto solo attraverso le operazioni consentite: il compilatore impedirà all'utente eventuali usi/accessi illeciti. Per fare ciò il linguaggio deve essere tipizzato e compilato. Il compilatore controlla che le operazioni siano coerenti con il tipo e ciò è fondamentale per ottenere un software di qualità. In un linguaggio tipizzato l'oggetto allora è istanza di un tipo, altrimenti è solo una struttura dati unita alle sue operazioni. Nelle classi l'interfaccia è costituita dalle operazioni consentite sugli oggetti istanziati. Secondo il meccanismo

Un altro tipo di moduli sono i **moduli generici** (in C++ li otteniamo tramite i template). Sono parametrici (valori che non sono costanti né variabili) e trattati in maniera generica. Se di un oggetto creiamo un tipo otteniamo un tipo di dato astratto (ADT), se un oggetto lo rediamo generico creiamo un oggetto generico. Possiamo anche rendere generico un tipo, ottenendo un tipo di dato astratto generico. Tipizzazione e genericità sono ortogonali. I moduli generici possono essere template di classe e di funzione, parametrizzati rispetto a un tipo. Per essere usati devono essere prima istanziati fornendo parametri reali (tipi). La genericità di una funzione è nella lista dei parametri. Quando della classe astratta specifichiamo il tipo (all'atto dell'istanziazione) otteniamo la classe del tipo specificato e da questa l'oggetto istanziato.

Un esempio di modulo generico sono gli iteratori che navigano sui contenitori (pila, lista, coda) a prescindere da come questi siano realizzati.

Coesione e accoppiamento:

Possiamo aggregare più moduli per fare un modulo composto, un esempio è la libreria. La coesione rappresenta il livello di omogeneità tra i componenti di un modulo, dev'essere alta. L'accoppiamento è l'interdipendenza tra moduli, dev'essere bassa. L'accoppiamento alto è indice di una struttura software non fatta bene. Dobbiamo avere dei criteri di coesione. In una libreria matematica tutti gli elementi saranno funzioni che svolgeranno operazioni matematiche. Il criterio può essere logico (come in questo caso), temporale (aggreghiamo le funzioni che vengono usate allo stesso tempo, un esempio è durante la fase di boot del computer). Possiamo anche organizzare il sistema a strati, dove ogni livello può accedere solo al successivo, ad esempio l'applicazione può essere realizzata molto più facilmente se realizziamo diversi livelli di astrazione. Ogni strato aumenta l'astrazione: un criterio di coesione può essere proprio quello del livello di astrazione, ragionando a strati, il che è un modo di realizzare l'architettura software (in fase di progettazione).

Esistono due tipologie di approccio di progettazione:

1**. Top down** : basata su un approccio di decomposizione funzionale, cioè sull'individuazione delle funzionalità del sistema da realizzare e su raffinamento successivi, da iterare finchè la scomposizione del sistema individua sottoinsiemi di complessità accettabile. procediamo dall'alto verso il basso. Può essere fatta in due modi, secondo l'astrazione funzionale o sui dati. L'astrazione funzionale si sposa bene con l'approccio top-down.

2. **Bottom-up**: dal basso verso l'alto. Solitamente attraverso l'astrazione sui dati partiamo dal basso, dall'individuazione delle entità facenti parte del sistema, assemblando poi i pezzi individuati.

Nella produzione di un sistema software sono coinvolte molte attività, come la fase di specifica, di progettazione, di verifica. L'ordine in cui queste vengono affrontate definisce il ciclo di vita del software. in generale, il processo di produzione del software è il processo che seguiamo per costruire, consegnare ai clienti e far evolvere il prodotto software, dalla nascita dell'idea fino alla consegna ed al ritiro del prodotto quando diviene obsoleto. Esaminiamo diversi modelli che cercano di catturare l'essenza di questo processo, tenendo ben presenti due cose: la produzione del software è un'attività prevalentemente intellettuale, quindi non facilmente automatizzabile; in secondo luogo il software è caratterizzato da un alto grado di instabilità (i requisiti cambiano in continuazione e quindi i prodotti devono essere evolvibili).

Code and fix : Un approccio primitivo alla produzione del software, tipico del programmatore singolo, consiste nello scrivere codice ed aggiustarlo (per correggere errori, migliorarne le funzionalità o per aggiungere nuove caratteristiche): code and fix. Questo modello è stato causa di molte difficoltà e carenze, in primis dovute al fatto che, dopo una serie di cambiamenti, il codice diventava disorganizzato rendendo le modifiche successive più difficili. Oggi, inoltre, il software non viene sviluppato per uso personale, ma per utenti che in generale non possiedono un background informatico. Diventano, quindi, importanti questione economiche, organizzative e psicologiche; è cresciuta la domanda per livelli qualitativi molto più elevati e i requisiti di affidabilità sono diventati più stringenti. Un'altra differenza sostanziale con il passato è che lo sviluppo del software è diventato un'attività di gruppo. Appare chiaro che il modello code and fix sia inadeguato per lo sviluppo odierno del software: - non c'è una gestione pianificata della complessità (modularità, separazione degli interessi, anticipazione del cambiamento); - la gestione del personale risulta difficile (problemi di ricambio del personale per mancata documentazione, che del resto è importante proprio per comunicare. La conoscenza deve essere scritta, anche mediante artefatti software, purché opportunamente documentata, per stabilire il linguaggio di comunicazione, in termini di metodologie, glossario ecc.); - risulta difficile aggiustare/modificare/ristrutturare il codice; - interpretazione sbagliata dei requisiti utente; - processo non prevedibile (tempi/costi, manutenibilità), qualità non misurabile.

Le tre caratteristiche fondamentali di un sistema software sono qualità, tempi e costi (triangolo delle qualità).

Questi problemi portano a riconoscere la necessità di processi predicibili e controllabili, dunque strutturati. La qualità del processo influenza: la qualità dei prodotti, i tempi per portare il prodotto sul mercato, i costi, le prestazioni sui diversi progetti.

1. **Acquisizione, analisi e specifica dei requisiti**: Questa è una delle attività più critiche del processo di ingegnerizzazione del software: se si pensa al software come passaggio da qualcosa di informale a qualcosa di estremamente formale, è proprio questa la fase in cui deve avvenire la transizione. L'ingegneria dei requisiti sviluppa metodi per raccogliere (estrarre l'informazione dal committente, che non necessariamente corrisponde ad una persona fisica, ma potrebbe anche essere il mercato), documentare, classificare ed analizzare i requisiti. C'è la necessità di elicitare i requisiti latenti, oltre ai requisiti funzionali. In questa fase bisogna svincolare il "cosa" dal "come": un esempio potrebbe essere la separazione degli interessi, specificando le funzionalità e le qualità che il software deve possedere, senza vincolare la progettazione e l'implementazione. I compiti dell'analista sono: identificare i portatori di interesse (stakeholders), esplicitare i requisiti, conciliare i vari punti di vista, specificare i requisiti. Il risultato di questa fase è la produzione di un Documento di specifica dei Requisiti (DSR), un glossario comune comprensibile, preciso, completo, coerente, non ambiguo, modificabile. D'altro canto, abbiamo detto che la linea di demarcazione tra correttezza e robustezza del software è proprio la specifica. Il DSR contiene: - descrizione del dominio. Le domande principali a cui rispondere sono: chi sono gli utenti interessati e quali sono i loro obiettivi e le loro aspettative? Quali sono le principali entità che caratterizzano il dominio?

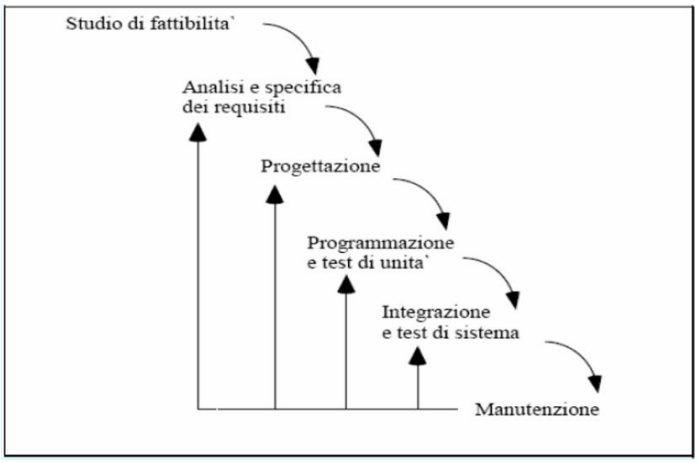
Quali sono le principali relazioni? Che influssi ha il sistema che si vuole sviluppare su di esse? - requisiti funzionali: descrivono che cosa dovrà fare il prodotto, usando notazioni informali, semiformali o formali. Un esempio che vediamo è lo standard UML. - requisiti non funzionali: affidabilità, disponibilità, integrità, sicurezza, ecc. - requisiti del processo di sviluppo-manutenzione: procedure per il controllo della qualità (procedure di test del sistema), possibili cambiamenti del sistema. Quando si specificano i requisiti, a fronte di essi, vanno specificate anche le modalità di test, mediante un Piano di Test di Sistema (PTS). È una modalità per ottenere feedback: se non si riesce a capire come testare i requisiti, significa che questi sono scritti male, non in maniera testabile. Bisogna poter confrontare l'output esterno con l'output atteso.

Attenzione: la scelta in questa fase di utilizzare uno specifico linguaggio di programmazione costituisce un vero e proprio vincolo, anzi può essere dannosa in quanto viene a mancare il principio di differimento delle decisioni (le decisioni vanno prese solamente a tempo debito). Potrebbe essere però un vincolo se necessito di garantire alcuni fattori come la portabilità (ad esempio sviluppare in Java garantisce una buona portabilità del sistema) o se esplicitamente richiesto dal committente. Come ciascuna fase, l'analisi presenta come input una raccolta di requisiti,

2. **Progettazione**: La progettazione è l'attività attraverso la quale i progettisti strutturano l'applicazione a diversi livelli di dettaglio. È possibile cominciare ad un livello alto definendo un'architettura che separi le funzionalità tra i componenti client e server. Si può procedere poi con una progettazione dettagliata che si occupi della scomposizione di client e server in componenti modulari, definendo le loro interfacce in maniera precisa. Ogni componente può essere a sua volta scomposto in ulteriori sottocomponenti. Il risultato è un documento di specifica di progetto (DSP) che contiene una descrizione dell'architettura software: descrive i componenti, le loro interfacce, le relazioni tra di loro; registra le decisioni significative e ne spiega le motivazioni. Una documentazione di questo tipo è importante in vista di possibili richieste future di cambiamenti che potrebbero comportare modifiche all'architettura. La forma esatta del documento di specifica di progetto è solitamente definita come parte degli standard adottati dall'azienda.

3. **Codifica, test e rilascio**: Questa fase si articola nella produzione del codice in un determinato linguaggio di programmazione e nel testing, in particolare si distinguono tre tipologie di test: test di unità, test di integrazione e di sistema, test di accettazione. Dopo aver completato lo sviluppo di un applicativo, rimangono da portare a termine diverse attività post-sviluppo. In primis, il software deve essere consegnato ai clienti. Ciò avviene, in genere, in due fasi: nella prima l'applicazione viene distribuita ad un gruppo selezionato di clienti prima del suo rilascio. Lo scopo è quello di eseguire una sorta di esperimento controllato per determinare, sulla base del feedback ricevuto dagli utenti, se sono necessari cambiamenti al software prima del rilascio: beta test. Nella seconda fase, invece, il prodotto viene rilasciato a tutti i clienti. Infine, la manutenzione può essere definita come l'insieme di attività svolte per modificare il sistema dopo che è stato rilasciato al cliente. La manutenzione può essere adattiva, correttiva o perfettiva.

Modello a cascata – Waterfall Model



Nel modello a cascata, il processo struttura le attività come una cascata lineare di fasi in cui l'output di una fase diventa l'input della seguente. Ogni fase, a sua volta, è strutturata, come un insieme di sottoattività che possono essere svolte da diverse persone in maniera concorrente. La fine di ogni fase è una milestone, l'output di ogni fase è detto deliverable, il quale deve essere ben definito per poter misurare la qualità del progetto. Il modello ha fornito due importanti contributi alla comprensione dei processi software: il processo di sviluppo del software deve essere soggetto a disciplina, pianificazione e gestione; l'implementazione del prodotto dovrebbe essere rimandata fino a quando non sono perfettamente chiari gli obiettivi.

Visto che è un modello ideale, il modello a cascata può essere solo approssimato nella pratica. È possibile caratterizzarlo mediante tre proprietà: linearità, rigidità e monoliticità. Il modello a cascata si basa sull'assunzione che lo sviluppo del software proceda linearmente dall'analisi alla produzione di codice. Nella pratica ciò non può succedere ed è necessario prevedere forme disciplinate di cicli di feedback. Lo scopo dell'alpha e del beta testing è infatti quello di fornire feedback alle fasi precedenti. La pianificazione del progetto è basata su un'assunzione di linearità e qualsiasi deviazione dalla progettazione lineare è sconsigliata in quanto rappresenta una deviazione dal piano originale e richiede una nuova pianificazione. Un'altra assunzione alla base del modello a cascata è quello della rigidità delle fasi, ovvero il fatto che i risultati di ogni fase vengono congelati prima di procedere alla fase successiva. Il modello, di conseguenza, assume che i requisiti e le specifiche di progetto possono essere congelati nelle prime fasi di sviluppo, quando le conoscenze dell'area applicativa e l'esperienza sono ancora preliminari e soggette a cambiamento. Questa assunzione non riconosce la necessità di un'interazione tra i clienti e gli sviluppatori in modo da far evolvere i requisiti durante il ciclo di vita. Infine, il modello a cascata è monolitico, nel senso che tutta la pianificazione è orientata ad una singola data di rilascio. Tutta l'analisi viene completata prima che cominci la progettazione, e il prodotto viene rilasciato mesi o addirittura anni dopo che i requisiti sono stati raccolti, analizzati e specificati. Se eventuali errori commessi in questa fase che non sono subito scoperti, verranno identificati solo dopo che il sistema è stato rilasciato ai clienti. Inoltre, siccome il processo di sviluppo può durare a lungo, il prodotto potrebbe essere rilasciato quando ormai i requisiti dei clienti sono cambiati e richiedere subito altri adattamenti. Il modello a cascata soffre di punti deboli: non permette una visibilità sufficiente del prodotto in sviluppo fino a quando il prodotto completo non è pienamente implementato.

Vediamo alcuni dei problemi associati alla rigidità del modello:

- è difficile stimare le risorse in maniera accurata quando sono disponibili solo informazioni limitate. La stima dei costi e la pianificazione del progetto, con il modello a cascata, sono spesso possibili solo dopo che è stata effettuata una prima fase di analisi;
- la specifica dei requisiti produce un documento scritto che guida e vincola il prodotto da sviluppare. Indipendentemente dalla leggibilità, esso rimane pur sempre un documento inanimato, ben diverso dallo strumento attivo che verrà rilasciato;
- l'utente spesso non conosce i requisiti esatti dell'applicazione, in alcuni casi non può conoscerli.

- non sottolinea sufficientemente il bisogno di anticipare possibili cambiamenti. Al contrario, la filosofia di base del modello è che si debba puntare alla linearità congelando più cose possibili nelle prime fasi.

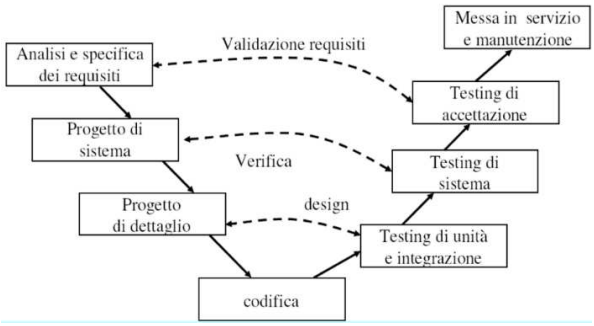
il processo è document driven, guidato dai documenti, obbligano a standard pesantemente basati sulla produzione di una data documentazione in determinati momenti.

Modello V&V

Il modello a cascata è meno utilizzato per via della sua eccessiva rigorosità, basandosi sull'assunzione che lo sviluppo del software proceda linearmente dall'analisi alla produzione del codice. Nella pratica ciò non può succedere ed è necessario prevedere forme disciplinate di feedback. Lo scopo, infatti, dell'alpha e del beta testing è quello di fornire feedback alle fasi precedenti. Un modello di questo tipo permette un feedback disciplinato ed esplicito, limitando i cicli di feedback da una fase a quella immediatamente precedente, minimizzando la quantità di lavoro da rifare nel ripercorrere le fasi precedenti. Il fondamento logico che si dovrebbe cercare di ottenere la linearità del ciclo di vita in modo da mantenere il processo il più possibile prevedibile e facile da monitorare. Questo viene chiamato modello a cascata **verification&validation**, con retroazione: variazione del modello a cascata con l'inserimento della retroazione, verifica la correttezza del risultato dell'attività con la possibilità di ripetere l'attività precedente. con verifica si intende la corrispondenza tra prodotto software e specifica, mentre con validazione si intende l'appropriatezza di un prodotto software rispetto ai requisiti utente.

E' una variante del modello a cascata: introduce un feedback in ogni fase, si possono così rilevare errori prima del rilascio, ma resta comunque un modello che non anticipa i possibili cambiamenti, quindi può essere utile quando a priori si sa che il sistema sarà poco soggetto ai cambiamenti.

Modello a V



Nel modello a V 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. Ciò si può iterare migliorando requisiti, progetto e codice.

Sulla parte sinistra è mostrata la fase in cui si passa dai requisiti alla codifica. Nella parte destra tutto ciò che riguarda il test, quindi la verifica. Inoltre nella parte di sinistra deve essere generato un documento che specificano come andare a testare i requisiti , in quanto non è detto che chi scrive i requisti sarà lo stesso a svolgere il test. Durante la fase di analisi, compie un'attività di test di accettazione che si fa a fronte dei requisiti d'utente (serve a verificare che il software soddisfi ciò che è richiesto.). Poi si passa al test di sistema che si fa a fronte del progetto di sistema e serve per verificare la specifica dei requisiti. E Infine il testing di unità e integrazione che si fa a fronte del progetto di dettaglio che mi dà una specifica di tutti i moduli del sistema.

Nello sviluppo di sistemi HW/SW (Hardware/Software), di cui il software è una parte, si prevede l'utilizzo di un modello a V: dapprima si esaminano i requisiti utente di sistema (sia funzionali, sia non funzionali), poi si identificano le unità del sistema assegnando requisiti alle unità (allocazione dei requisiti). L'analisi dei requisiti hardware/software esamina le risorse di ogni unità, decomponendole in CSCI (Computer Software Configuration Items) e HCI (Hardware Configuration Items).

Le varianti del modello a cascata sono adatte soprattutto quando si prevede che il sistema (e/o l'ambiente) sarà poco soggetto a cambiamenti, oppure vi sono requisiti chiari e completi sin da subito con poca possibilità di cambiare (ad esempio sistemi non interattivi).

Modelli evolutivi

I modelli prototipali utilizzano i prototipi, quando i requisiti non sono chiari o possono cambiare (abbiamo visto che nel modello a V e nei modelli a cascata i requisiti sono congelati), con l'obiettivo di raffinare i requisiti. Il prototipo non è il prodotto finale, ma una versione inziale, utilizzata solo temporaneamente, fino a quando fornisce al progettista un feedback sufficiente. Quindi, si realizza un prototipo, un modello dell'applicazione, con lo scopo di validare i requisiti, ricevendo feedback dagli stakeholders e/o committenti. Si finisce di utilizzare il prototipo quando sono stati chiariti i requisiti: ad esempio, un prototipo potrebbe essere un'interfaccia utente dell'applicazione, in cui non è necessario che le funzionalità siano effettivamente implementate, ma è utile per stabilire come si deve presentare l'applicazione (erano i requisiti di interfacciamento uomo-macchina). Logicamente un prototipo deve costare poco, facile dea realizzare, ad esempio mediante applicazioni web, interfacce come ambienti di programmazioni visuali (Visual Basic). Il prototipo può essere **usa e getta** (throw-away): la prima versione dell'applicazione viene cestinata, la seconda versione può poi essere sviluppata seguendo un modello di processo a cascata. Questo approccio (ispirato al principio "Do it twice") fornisce una soluzione parziale ad alcuni problemi discussi precedentemente, come l'eliminazione di errori nei requisiti. Non elimina comunque la distanza temporale che esiste tra la definizione dei requisiti e il rilascio dell'applicazione e non sottolinea la necessità di anticipare i cambiamenti. Se invece il prototipo porta progressivamente al prodotto finale si parla di **prototipo evolutivo**: il prototipo è uno strumento di identificazione dei requisiti di utente; è incompleto, approssimativo, realizzato utilizzando parti già possedute o routines stub.

Bohem definisce il modello di processo evolutivo come "un modello le cui fasi consistono in versioni incrementali di un prodotto software operazionale con una direzione evolutiva determinata dall'esperienza pratica".

MODELLO INCREMENTALE

Il modello evolutivo per implementazioni incrementali è un processo a cascata fino all'implementazione, solo l'implementazione è rilasciata incrementalmente, le fasi precedenti sono come da modello a cascata. Permangono, perciò, problemi tipici del processo a cascata come l'eccessiva distanza tra l'analisi dei requisiti dell'intero sistema al primo rilascio, il "congelamento" dei requisiti (la specifica dei requisiti soggetta a possibile successiva invalidazione). Il **modello incrementale** più generale si ha quando l'approccio incrementale è esteso a tutte le fasi del ciclo di sviluppo si parla di modello a rilasci incrementali: si inizia con una fase che copre gli obiettivi del sistema complessivo e l'architettura complessiva, si procede con l'analisi dei requisiti di un incremento. Ogni incremento è progettato, implementato, testato, integrato e rilasciato. Gli incrementi tengono conto dei feedback (affinamento dei requisiti). È sicuramente più facile pianificare le risorse necessarie, dato che si procede per un incremento alla volta. In questo modo si può far fronte alle esigenze del cliente che almeno riesce ad avere subito a disposizione alcune funzionalità, ed è utile quando lo sviluppatore viene pagato a rate. In un modello evolutivo, la manutenzione scompare come fase del ciclo di vita, il ciclo di vita diventa un'evoluzione continua.

Modello trasformazionale

Il modello trasformazionale si basa su due concetti: prototipazione e formalizzazione, ottenendo un codice eseguibile di basso livello. Si specificano i requisiti formalmente (devono essere convalidate prima di essere trasformata), si procede trasformando man mano la descrizione formale in una meno astratta e più dettagliata, fino a che diviene eseguibile da un processo astratto. Le specifiche eseguibili possono essere viste come un prototipo evolutivo.

L’idea di base è che a giudicare lo sviluppo non è la documentazione, ma il modello. Lo sviluppo è la trasformazione di modelli, da uno più astratto a uno più raffinato. Avviene sotto due punti di vista: trasformare il modello in codice dal punto di vista strutturale e la generazione automatica comportamentale.

Le trasformazioni possono essere eseguite manualmente o supportate da appositi strumenti. Uno dei vantaggi del processo di trasformazioni è la riusabilità: i componenti sono riusabili, del resto abbiamo detto che otteniamo un semilavorato di livello i+1 da uno di livello i. Attualmente non è un paradigma praticabile, ma è guardato con interesse per l’approccio formale allo sviluppo del software. Per esempio, fatto un modello di analisi e considerate delle regole di trasformazione, sarebbe semplice ottenere la realizzazione del progetto mediante i principi di questo modello, ma questo implicherebbe una formalità dei requisiti. La formalità è molto onerosa.

Modello a spirale

E’ un metamodello, non uno specifico modello , ma uno schema generale. 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. Il processo è rappresentato come una spirale, piuttosto che come una sequenza di attività con retroazione. Ogni giro della spirale rappresenta una fase del processo

- **Primo settore**: Definizione di obiettivi, vincoli e piano di gestione della fase.
- **Secondo settore** : Si analizzano I rischi della fase e si scelgono le attività necessarie a gestire I rischi (ad esempio tramite simulazione o prototipazione)
- **Terzo settore:** Si sceglie un modello di sviluppo per il sistema tra I modelli generici Nel terzo settore si può utilizzare:
  - Modello evolutivo, se i requisiti sono incerti
  - A cascata se i requisiti sono chiari e ben definiti
  - Trasformazionale se la sicurezza è un requisito più importante
- **Quarto settore:** revisione dei risultati e pianificazione della prossima iterazione della spirale

Metodologie agili

Le metodologie agili sono un sottoinsieme dei modelli evolutive, nate in alternativa alle metodologie tradizionali. Sono nati come modelli di sviluppo leggeri (lightweight): più adattivi che predittivi, nel senso si progettano programmi pensati per cambiare nel tempo. Negli approcci tradizionali, come quello a cascata, un aspetto delicato è proprio la pianificazione: dall'inizio dobbiamo cercare di sapere con precisione cosa e come vogliamo fare, sia come funzionalità che come fattori di qualità, poi pianifichiamo il processo di lavorazione, fino al rilascio ed alla manutenzione successiva. La pianificazione riguarda anche le risorse, il personale, le skills del personale, monitorare la produzione. Con le metodologie agili, si cerca di limitare l'attività di pianificazione del processo, anzi si scrive e si documenta il meno possibile, mentre si sviluppa, anche per piccoli incrementi, il più rapidamente possibile. Sono metodologie people-oriented anziché process-oriented, (dove per people si considerano anche gli stessi sviluppatori) l'approccio prevede di adattare il processo alla natura dell'uomo. Lo sviluppo software deve diventare un'attività piacevole per chi lo opera. Anche per le metodologie agile l'ambito d'uso è che i requisiti siano pochi chiari o instabili, addirittura variabili nel tempo. L'approccio è flessibile (quindi è un modello evolutivo) e "leggero", poiché si parte dal presupposto che le numerose prescrizioni da seguire, la quantità di documenti richiesta e un'eccessiva rigidità rendono "pesante" il processo di sviluppo, aumentando il rischio di fallimento.

La priorità massima delle metodologie agili è soddisfare il cliente, pertanto il cambiamento dei requisiti è ben accetto. Le persone del business e gli sviluppatori devono lavorare insieme giorno per giorno. In certi ambiti, avere il cliente vicino può convenire piuttosto che fare "interviste", pianificare, e solo dopo produrre e rilasciare. Un aspetto tipico è lavorare a coppie (pair programming), uno programma e l'altro fa il test, poi il contrario: in questo modo il test si fa subito e viene fatto da un occhio diverso rispetto al programmatore.

Agile	Adatto a progetti particolarmente innovativi e con requisiti non ben definiti	A causa della poca strutturazione, necessità di una grande disciplina e motivazione; non molto adeguato ai contratti; difficilmente comprensibile dal cliente

Ci sono varie metodologie agili, tra le più diffuse ci sono **XP** e **Scrum, DevOps**.

XP (eXtreme Programming)

è di Kent Beck (uno degli autori del manifesto) ed è la metodologia agile più diffusa. Si sviluppa per iterazioni molto veloci, che rilasciano piccoli incrementi, magari fatti con pair programming, mettendo più coppie a lavorare. Il committente partecipa attivamente al team sviluppo, il software viene testato continuamente per verificare cosa funziona e cosa no. Il risultato è un processo che combina disciplina e adattività. Le pratiche sono: **planning game** (determinare obiettivo e tempi della prossima release; durano poco), **small releases** (rilasciare velocemente un piccolo incremento, quindi saranno necessari programmatori esperti), **simple design** (il sistema è concepito nel modo più semplice possibile, non perdendo tempo nelle grosse formalizzazioni del design), refactoring (ristrutturare il sistema senza cambiarne il comportamento), **testing** : Da effettuare costantemente durante lo sviluppo (**Test Driven Development, TDD**)., il test-driven development è un modello di sviluppo del software che prevede che la stesura dei test automatici avvenga prima di quella del software che deve essere sottoposto a test, e che lo sviluppo del software applicativo sia
orientato esclusivamente all'obiettivo di passare i test automatici precedentemente predisposti.

Più in dettaglio, il TDD prevede la ripetizione di un breve ciclo di sviluppo in tre fasi, detto "ciclo TDD". Nella prima fase (detta "fase rossa"), il programmatore scrive un test automatico per la nuova funzione da sviluppare, che deve fallire in quanto la funzione non è stata ancora realizzata. Nella seconda fase (detta "fase verde"), il programmatore sviluppa la quantità minima di codice necessaria per passare il test. Nella

terza fase (detta "fase grigia" o di refactoring), il programmatore esegue il refactoring del codice per adeguarlo a determinati standard di qualità

Un'altra pratica è il **collective ownership**: chiunque può cambiare qualsiasi parte del codice quando vuole, il software non viene visto come un prodotto artigianale. Tramite l'Ingengeria del Software si passa dal software come artigianato al software come industria. Del resto, il software non è brevettabile, gli algoritmi non sono brevettabili ma sono solo soggetti a copyright, dato che ci si chiede se esso sia effettivamente frutto dell'ingegno.

Scrum è un'altra metodologia agile, dove scrum significa mischia nel rugby. E' un modello iterativo ed incrementale, adottato negli anni 90, per lo sviluppo e gestione di ogni tipologia di prodotto. E' 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;
- **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 tradizionali fasi di sviluppo del software e può durare da una settimana ad un mese. Uno sprint è gestito da uno scrum master (ogni gruppo ha il suo gestore). Esiste anche il ruolo del product owner, che rappresenta gli stakeholders, e infine c'è il team, ovvero il gruppo che esegue analisi, progettazione, implementazione e test. Giornalmente si fanno i meeting e durano al massimo un quarto d'ora. La backlog è la coda di cose da svolgere, questa coda 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.

DevOps è la frontiera più recente, sta per development operations. Fa parte delle metodologie evolutive, più che delle agili. E' un particolare modello di ciclo di vita di un software in cui l'obbiettivo è unire la parte di sviluppo con quella di manutenzione, in quanto vi è stretta correlazione tra le due parti. Viene raffigurato con il simbolo di infinito per sottolineare la stretta collaborazione fra i due team. Queste parti che vengono messe insieme vengono testate periodicamente. Prima che venga rilasciato se vi sono degli errori, viene notificato allo sviluppatore , che può ri-lavorare sulle cose scritte. Nel ciclo di vita vi è una parte che è il CONTINOUS INTEGRATION, rappresentato da un cerchio. Corrisponde al fatto che viene fatto sempre il test d'integrazione dovuto al motivo che si lavora in parallelo. Ogni sviluppatore lavora sulla propria parte e quando si trova un problema o un'incongruenza viene notificato allo sviluppatore che l'ha scritto. Quindi, quello che scrive una persona deve essere congruente con quello che scrivono gli altri, in quanto è una parte dell'intero sviluppo del software.

Un vantaggio è la sicurezza: dobbiamo garantire che i dati immessi non siano visti dagli altri clienti del nodo. Ciò si fa con tecnologie di virtualizzazione e di contenitore. Ognuno ha un contenitore, un ambiente che contenga sul nodo. In questo modo utilizziamo la macchina virtuale isolatamente dal resto.

Stima dei costi:

La stima dei costi è fondamentale in fase di pianificazione per stimare il prezzo del prodotto finale, dimensionare. È applicabile anche ai prodotti software già esistenti, nel momento in cui dobbiamo valutare se effettuare una manutenzione (correttiva, adattativa o perfettiva, a seconda dell'esigenza) sia effettivamente più conveniente di rifare completamente il prodotto (in termini di risorse). I metodi non sono molti e sono tutti basati su principi empirici, il principale è l'analisi dei punti di funzione.

La stima dei costi è influenzata dalla produttività degli sviluppatori, serve una **metrica di produttività**. La metrica ideale dovrebbe misurare quale sia il valore prodotto per unità di sforzo, ad esempio, se per unità di sforzo consideriamo il mese/uomo il costo dipende da cose note, come gli stipendi da pagare, ma il valore è diverso. I **punti funzione** (function points) sono una metrica per la stima della quantità di funzionalità del software, il punto f corrisponde al valore della funzionalità. Calcolati tali punti siamo in grado di derivare la quantità di sforzo richiesto per realizzarlo, che non dipende dalla tecnologia.

I function point sono standardizzati dall'ISO e dal gruppo di lavoro IFPUG (International Function Point User Group). Il metodo 4.1 è l'UFPM (il metodo degli unadjusted function points) e si applica in due stati:

1. Calcoliamo la stima delle funzionalità, senza aggiustamenti
2. Applichiamo dei valori correttivi in base alla letteratura dell'ingegneria del software o all'esperienza nella nostra realtà specifica.

A partire dai requisiti funzionali determiniamo quante sono le funzionalità da realizzare (cosa che siamo in grado di evincere già dai casi d'uso) e il peso di ciascuna di esse, ovvero la complessità in base ai dati in ingresso, in uscita e ai dati da produrre e memorizzare.

- **Passo 1** - valutazione degli unadjusted function points:

1. Determinazione del tipo di conteggio;
2. Identificazione dell'ambito di conteggio e dei confini applicativi;
3. Identificazione delle funzioni dati;
4. Identificazione delle funzioni transazionali;
5. Conteggio degli UFP;

al termine di questa fase applicheremo i fattori di aggiustamento e otterremo i function points.

Per prima cosa dobbiamo individuare il tipo di conteggio tra:

- **Conteggio per sviluppo di progetto**, nel caso in cui stiamo sviluppando un nuovo software;
- **Conteggio per manutenzione evolutiva**, viene applicato per misurare le modifiche ad un'applicazione esistente nel momento in cui vengono presentati requisiti aggiuntivi;
- **Conteggio applicativo**, misura un'applicazione già esistente (può essere utile per scegliere se rifare il prodotto exnovo o effettuare una manutenzione).

Successivamente identifichiamo il confine applicativo, la linea di demarcazione tra l'utente (o altri sistemi) e l'applicazione da utilizzare. Tale individuazione va effettuata dalla percezione dell'utente in merito alle funzionalità, dunque in base ad aspetti di business e non tecnologici. Dobbiamo individuare gli input provenienti dall'utente esterno, gli output ad esso forniti e gli external inquiry (input e output scambiati con applicazioni esterne). Avremo uno scambio di file con altre applicazioni ma anche file interni, non intesi come file fisici ma come file logici.

Nello step di identificazione delle funzioni dati distinguiamo tra:

• **Internal logical file (ILF):** aggregati logici di dati generati, gestiti e usati internamente dal sistema (ad esempio, tabelle di database, file di errore, password trattenute dal sistema ecc.). Affinchè dei file siano tali devono essere identificabili all'utente, dunque i file temporanei non sono ILF. Nell'esempio prima visto 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 nel deposito dati dell'altra, in generale gli ELF di un'applicazione sono ILF di un'altra.

Nella fase di identificazione delle funzioni transazioni invece:

• **External input (EI):** informazioni distinte fornite dall'utente o da altre applicazioni, usate come dati di ingresso. Generalmente alterano gli ILF o agiscono sul funzionamento del sistema. Non sono da considerarsi tali schemi di menù usati per navigare all'interno dell'applicazione, in quanto non alternao gli ILF, richieste di input da parte di una query o meccanismi per il refresh delle schermate a video.

• **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, tramite cui vengono dati risultati all'utente prelevando dati da un ILF o un EIF. Avranno un peso mine rispetto agli EO proprio per la mancata elaborazione. Non sono EQ le schermate di menù, i messaggi di errore e i messaggi di conferma.

Per effettuare il conteggio degli UFP contiamo il numero degli ILF, EIF, EI, EO e EQ (rispettivamente NILF, NEIF, NEI, NEO e NEQ) e li pesiamo per un fattore, associato ad una stima di complessità (semplice, media o complessa). Gli UFP li otteniamo come sommatoria dei ViPi. I dati ottenuti provengono da stime empiriche, possiamo aggiustarli aggiungendo la nostra conoscenza. Notiamo che, a parità di complessità, il numero di internal logical file (ILF) pesa meno degli EQ. Questo perché se NIFl è il file di scrittura del record corrispondente al prodotto comprato su Amazon, l'EQ conterrà una query che restituisca quel dato e, pur non compiendo un'elaborazione, verrà effettuata almeno la ricerca. Tanto più è complessa la scrittura tanto più sarà complessa la query. Al crescere della complessità il rapporto tra NIFL e NEQ rimane pressoché invariato. Il parametro più pesante è il NEO, in quanto richiede un'elaborazione.

• **Passo 2** - applicazione dei fattori correttivi:

Distinguiamo 14 fattori correttivi, il cui valore varia tra 0 e 5:

- 0 ininfluente
- 1 incidenza scarsa
- 2 incidenza moderata
- 3 incidenza media
- 4 incidenza significativa
- 5 incidenza essenziale

Il parametro prestazioni pesa gli obiettivi prestazionali dell'operazione. Se non vi sono tali vincoli sarà un fattore ininfluente (0). Se i vincoli non funzionali impongono il rispetto dei tempi ma in maniera non troppo preoccupante varrà 1. Se i tempi di risposta sono critici durante le ore di picco 2, se lo sono durante l'intera giornata 3. C'è infatti differenza se sono critici solo in presenza di traffico o sempre, gestiremo le situazioni differentemente, in un caso va gestito il caso medio, nell'altro il picco (nei sistemi real time, ad esempio, distinguevamo gli hard real time e i soft real time, a seconda che la scadenza fosse mandatoria o da rispettare in termini probabilistici). Se l'ottimizzazione delle prestazioni riveste estrema criticità varrà 5.

Efficienza per l'utente finale: esprime il grado di considerazione per i fattori umani e la facilità di utilizzo per l'utente. Alcuni requisiti possono riguardare la possibilità di fornire gli help, la documentazione online, il menù, lo scrolling ecc. Ciò che cambia è la logica di interfacciamento e i pesi vengono stabili in base a quanti di questi elementi (sopracitati) sono importanti.

Riusabilità: tendenzialmente non è un fattore che interessa l'utente, può in situazioni come le pubbliche amministrazioni. Pesiamo tale fattore in base alla quantità di moduli che potranno essere riutilizzati.

Il valore finale del conteggio FP è il prodotto tra il conteggio UFP e l'aliquota introdotta dai fattori correttivi, secondo la formula:

$$FP = UFP \times \left(0.65 + 0.01 \times \sum_{i=1}^{14} F_i \right)$$

Tramite gli FP possiamo calcolare il costo del progetto. Ad esempio, se dobbiamo realizzare 12 function point e ciascuno costa 1000 il costo finale allora sarà banalmente 12000. Per calcolare il costo di ciascun function point applichiamo delle stime tramite le tabelle, i valori dipenderanno da fattori come il linguaggio di programmazione adoperato, il numero di linee di codice (che varia di linguaggio in linguaggio). Più è elevato il livello di astrazione è meno saranno le linee di codice necessarie tra i rispettivi linguaggi.

Ingegneria dei requisiti:

Per ogni modello di ciclo di vita del software è fondamentale un'attenta analisi dei requisiti. In questa fase ci occupiamo di capire le funzionalità e le qualità che devono essere assicurate dal software, tralasciando come queste saranno realizzate. Prima di rilasciare il software dobbiamo verificare che esso sia corretto, dobbiamo verificare che sia conforme ai requisiti specificati, che dunque ritornano necessari in questa fase finale. Per la stessa progettazione dei test abbiamo bisogno dei requisiti: pianifichiamo i test non appena abbiamo specificato i requisiti, ciascun requisito per essere un "buon requisito" dev'essere infatti facilmente verificabile. I requisiti verranno successivamente aggiunti o modificati in fase di manutenzione. L'ingegneria dei requisiti è la 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.

Possiamo effettuare una distinzione tra:

- **Requisiti utente**: riguardano le funzionalità e i servizi che il software deve offrire ed eventuali vincoli operativi da rispettare. Generalmente sono scritti per i clienti (clienti, client manager, system architects..) e per l'approvazione del contratto. Sono scritti in linguaggio naturale o tramite diagrammi, così da essere facilmente comprensibili anche ai "non addetti ai lavori".

- **Requisiti di sistema**: si tratta di un documento strutturato che fornisce una descrizione dettagliata delle funzionalità e dei servizi del sistema. I requisiti di sistema sono in parte comuni e in parte diversi dai requisiti utenti. Nei requisiti utenti, ad esempio, potrebbero esservi requisiti di interfacciamento, non presenti nei requisiti di sistema.

I **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". Essi devono descrivere:

- Quali input il sistema deve accettare
- Quali output deve produrre
- Quali dati bisogna gestire
- Quali elaborazioni bisogna svolgere
- Eventualmente tempificazione e sincronizzazione

Modelliamo i requisiti funzionali tramite il diagramma dei casi d'uso e i relativi scenari, questi ultimi indicano input, output, precondizioni e post-condizioni.

I requisiti non-funzionali impongono vincoli o qualità, riguardano tempi di risposta, uso delle risorse (efficienza e prestazioni), affidabilità, manutenibilità ecc..

I **requisiti non funzionali** possono riguardare il prodotto (efficienza, portabilità, security, reliability..), il processo e possono essere requisiti esterni, quale ad esempio l'interoperabilità, oppure requisiti etici (i dati non vanno esposti ad usi impropri). Definiscono è limitano le proprietà del sistema. Sono più critici dei requisiti funzionali e possono vincolare anche il processo di sviluppo da adottare.

Le fasi dell'ingegneria dei requisiti L'ingegneria dei requisiti sviluppa metodi per raccogliere, documentare, classificare, analizzare e gestire i requisiti. E' essenziale, in quanto errori in questa fase si propagano nei passi successivi e il costo per porvi rimedio cresce col passare del tempo. I requisiti non corretti possono portare a consegne ritardati, costi maggiori, insoddisfazioni dell'utente, comportamenti errati ed imprevisti, un prodotto software corretto (conforme alle specifiche) ma che non soddisfi i bisogni dell'utente, altri costi di manutenzione (appena finita l'analisi delle specifiche, si può controllare la correttezza, la verificabilità, la completezza mediante una review da parte di altri).

Le fasi principali della Requirements Engineering sono:

- Elicitation (esplicitazione): in questa fase, si acquisiscono i requisiti tramite consultazioni con gli stakeholders, documenti, conoscenza del dominio e studi del mercato: si cerca di far emergere tutti i requisiti funzionali e tutte le caratteristiche dei requisiti non funzionali.

- Analisi e negoziazione: vengono valutati i requisiti rispetto a completezza, conflitti (tra viste diverse dei vari stakeholders), compatibilità. Questa fase si articola, a sua volta, in:

1. Checklist: si verifica che i requisiti abbiano le caratteristiche desiderate;
2. Requirements prioritization: si conferiscono le priorità ai requisiti;
3. Interaction matrix: si controlla l'interazione tra i diversi requisiti, per evitare inconsistenze;
4. Risk Analysis: analisi dei rischi.

Successivamente, i requisiti vengono elaborati e si procede alla negoziazione con gli stakeholders.

- Documentazione: si produce un documento tecnico che descriva le funzioni, prestazioni e vincoli del sistema. Si possono distinguere diversi documenti (testuale, insieme di modelli, insieme di scenari utente, un prototipo) anche a seconda degli stili: approccio informale (le specifiche sono espresse in linguaggio naturale, strutturato), approccio formale (automa a stati finiti), approccio semi – formale (UML). La specifica dei requisiti è un documento strutturato che contiene descrizioni dettagliate dei servizi del sistema (specifica funzionale), può servire come contratto tra committente e sviluppatore. La specifica del software è una descrizione astratta del software che è la base del progetto e aggiunge ulteriori dettagli alla specifica dei requisiti. Il documento di specifica dei requisiti software (SRS) costituisce il punto di convergenza di tre diversi punti di vista: cliente, utente, sviluppatore. È un punto di riferimento per la validazione del prodotto finale: un documento SRS di qualità è il pre -requisito per un software di alta qualità e riduce i costi di sviluppo.

- Validazione: l'SRS viene passato al progettista, che elabora una soluzione. Per controllare che i requisiti specificati vengano elaborati realmente, i requisiti devono essere allocati alle parti della soluzione (allocazione dei requisiti al sistema): ad esempio, se sto utilizzando il principio di modularità, bisogna specificare quali requisiti vengono elaborati in ogni modulo. Successivamente si passa alla progettazione di basso livello, scomponendo ogni componente fino ad arrivare a componenti sufficientemente piccoli da dominare la complessità. Durante la scomposizione, è necessario continuare ad allocare ciascun requisito al sottocomponente particolare. Si passa, poi, alla codifica, la stesura del programma: ciascun modulo software implementa qualche componente, e grazie all'allocazione dei requisiti, di ciascun modulo software è possibile conoscere i requisiti che esso deve realizzare. Quindi, la specifica dei requisiti viaggia lungo tutto il processo, dall'analisi fino alla fase di test: tracciabilità dei requisiti.

- Gestione: tipicamente i requisiti sono soggetti a cambiamenti, che devono essere gestiti (requirements management), non modificando semplicemente il documento SRS, ma anche considerando un impact analysis (analisi d'impatto), ovvero la stima degli impatti del cambiamento sui requisiti e sulle attività, possibile grazie alla tracciabilità.

L'ingegneria dei requisiti non si limita soltanto alla prima fase del processo di realizzazione di un sistema software, ma al contrario interessa tutte le fasi: ciascun requisito va seguito almeno fino alla fase di realizzazione (collaudo e test), motivo per cui il software test plan, documento che specifica quali saranno i test da effettuare, va steso subito dopo l'SRS.

PATTERN ARCHITETTURALI

Un **pattern** è la descrizione di un problema ricorrente e della soluzione che possiamo mettere in atto per risolverlo. I pattern esistono in ciascuna fase del ciclo di vita del software, dall'analisi alla codifica. I pattern di progettazione sono detti **pattern architetturali**, i pattern di progettazione di dettaglio sono detti **design patterns**. Essi favoriscono il riuso del software e delle sue componenti ma allo stesso tempo forniscono un vocabolario comune.

In fase di architettura organizziamo il sistema in componenti e, mappiamo le funzionalità sui moduli di cui si compone il software. Oltre ai requisiti funzionali vanno mappati anche i requisiti non funzionali, che risulta essere un'operazione più difficile.

Per decomporre un sistema in sottosistemi abbiamo due approcci tipici, **l'architettura a strati (layer)** o a **partizioni (tier)**.

1. Le architetture a strati sono gerarchie, ogni strato fornisce dei servizi, si manifesta tramite operazioni che altri strati possono invocare. Il livello di interfaccia tra due strati è una **API** (**Application Programming Interface**): l'interfaccia di uno strato che fornisce servizi ad altri strati che possono accedervi invocando le operazioni definite nell'API. Tra le architetture a strati differenziamo le architetture chiuse da quelle aperte. In **un'architettura chiusa** ogni strato può accedere solo a quello immediatamente sottostante. In **un'architettura aperta** gli strati possono usufruire dei servizi offerti anche da strati più in basso. Una tipica architettura chiusa è quella dei protocolli, un sistema operativo è invece aperto: c'è un nucleo, kernel, dove è descritta la schedulazione dei processi e la gestione della memoria, ogni componente lo utilizza. Progettare un'architettura chiusa fa sì che quando viene cambiato uno strato non cambia niente se l'API non viene impattata, in caso contrario cambia solo lo strato immediatamente superiore, in quanto è l'unico a utilizzarla. Se l'architettura è aperta nel momento in cui cambia un'API cambieranno tutti gli strati che vi hanno accesso.

2. L'architettura a partizioni organizza il sistema in moduli tra loro pari (peer). Solitamente si utilizza questo approccio per raggruppare servizi secondo qualche criterio di coesione. Nelle architetture a strati il criterio era per lo più il differente livello di astrazione. Nelle singole partizioni i servizi sono coesi tra loro ma diversi dagli altri.

Esempi di architectural patterns:

- **Pattern MCV (Model-View-Controller):** È utilizzato in particolar modo in alcune tecnologie legate a Java. Questo pattern nasce dalla necessità di visualizzare dati generici tramite interfaccia grafica mediante l'utilizzo di rappresentazioni diverse dei dati stessi. Il sistema viene strutturato in tre componenti che interagiscono tra loro: **il Controller, la Vista e il Modello**. Il modello incapsula i dati, la vista si occupa della loro presentazione e il controller incapsula la business logic. L'utente riceve quello che mostra la vista e imposta ciò che vuole vedere nello strato View. Se vi sono delle modifiche, l'utente deve visualizzare i dati modificati. Quando l'utente richiede di vedere un dato la Vista lo deve prelevare dal Modello. Questo passaggio non avviene direttamente tra la Vista e il Modello, altrimenti la View implementerebbe anche la business logic, avviene tramite il Controller che trasforma gli input utente (nella View) in azioni del Model, comunicando al Modello ciò che dev'essere fornito alla Vista per cambiare la visualizzazione dei dati. L'MVC disaccoppia Vista e Modello secondo il modello sottoscrizione-notifica. I dati si trovano nel Modello e quando cambiano occorre cambiarne la rappresentazione, per preservarne la

coerenza. Quando il Modello cambia avvisa la Vista tramite una notifica (modalità “push”), in tal caso il Modello fa da notificatore e la Vista dovrà aver effettuato una sottoscrizione, richiedendo di essere aggiornata. Il Modello incapsula lo stato dell’applicazione, cambia il dato in quanto cambia lo stato e dunque va modificata la Vista. La Vista a sua volta interpreta il Modello e può voler modificare un dato, così come può richiedere un nuovo stato. La Vista comunicherà al Controller l’input ottenuto e questo comunicherà al Modello di cambiare lo stato.

- **Pattern BCE (Boundary – Control – Entity):** Si basa sulla presenza di tre gruppi di classi package:

  1. il **package boundary** contiene le classi che gestiscono l’interfacciamento con l’utente e la logica presentazione;
  2. il **pakcage control** contiene gli oggetti che recepiscono gli eventi generati dall’utente, rappresentano le azioni dei casi d’uso, implementando la business logic;
  3. il **package entity** raggruppa le classi che rappresentano i dati, entità del dominio applicativo, e si occupa dell’interfacciamento tra questi e la base di dati in cui sono salvati .

  Per certi versi è un pattern simile all’MVC ma nasce con obiettivi diversi. A differenza dell’MVC nasce con l’obiettivo di assegnare a oggetti diversi responsabilità diverse. Siamo in fase di progettazione perché stiamo modellando l’architettura del software, in analisi infatti la persistenza dei dati non è un problema presente. BCE nasce con l’obiettivo di separare le responsabilità di elementi nell’OOD , mentre MVC nasce con l’obiettivo di separare la selezione delle interfacce utente dal resto dell’applicazione. Il primo è più orientato a separare chiaramente la business logic dal resto in applicazioni OOD, il secondo è più focalizzato sull’interfacciamento con l’utente.

  Se la persistenza si realizza tramite database si parla di **pattern BCED**, nel **package database** avremo le classi che si occupano della gestione del database. L’interfacciamento con il database viene gestito da quest’ultimo package. Nel progettare la soluzione dobbiamo gestire l’apertura e la chiusura delle connessioni con il database, l’autenticazione, la sicurezza, il concetto di cursore ecc.. L’accesso a DB può essere gestito con tre strategie:
  • Accesso diretto (forza bruta)
  • Oggetti per accesso ai dati (Data Access Objects, DAO)
  • Persistence framework

- **Pattern Broker**: Il Broker è un pattern architetturale con schema logico di tipo publish/subscribe. Se consideriamo un’applicazione client/server ,i **server** sono coloro che rendono noti i servizi forniti (publisher) e i **client** utilizzano quei servizi (subscriber). Client e server, ovvero chi richiede e chi offre, sono disaccoppiati da un intermediario, **il broker**. Il broker rappresenta un bus software che permette ai produttori di porre tutti i dati sul bus, così da non dover comunicare direttamente con ciascun consumatore. Un broker è un componente responsabile della trasmissione delle richieste dai clienti ai serventi, e della trasmissione ai clienti delle risposte o delle eccezioni rilevate. I **bridge** sono componenti opzionali del pattern, utilizzati per nascondere i dettagli di implementazione della comunicazione tra i broker in modo che, in una rete eterogenea, i broker comunichino in maniera trasparente, rispetto all’architettura di rete e ai sistemi operativi in uso. Il **proxy lato cliente**, è lo strato software addizionale tra cliente e broker, che rende trasparente al cliente l’accesso ad un oggetto remoto, che appare così come un oggetto locale. Il **proxy lato servente**, è il duale del proxy lato cliente.

Per registrare un server sul broker locale:

1. Il broker viene eseguito nella fase di inizializzazione del sistema. Entra così in un ciclo di attesa eventi, aspettando che arrivino messaggi.
2. L'utilizzatore esegue l'applicazione server, che esegue una fase di inizializzazione dopo la quale si registra presso il broker.
3. Il broker riceve il messaggio di richiesta registrazione, estrae le informazioni necessarie e le memorizza in un'area dati, il repository, che serve per trovare e attivare i server, e invia un messaggio di avvenuta registrazione al server.
4. Ricevuta la conferma, il server entra nel ciclo di attesa delle richieste provenienti dai clienti.

Quando un cliente invia una richiesta ad un broker locale:

1. L'applicazione cliente viene eseguita. Durante l'esecuzione del programma il cliente invoca un metodo di un oggetto server remoto.
2. Il proxy lato cliente impacchetta tutti i parametri ed altre informazioni supplementari, in un messaggio che viene trasferito al broker locale.
3. Il broker ricerca la posizione del server richiesto all'interno del suo repository. Se il server è disponibile localmente, il broker trasferisce il messaggio al corrispondente proxy lato server.
4. Il proxy lato server spacchetta i parametri e le informazioni supplementari, tra cui il metodo che bisogna invocare. A questo punto il proxy richiama il servizio appropriato.
5. Dopo che il servizio è stato eseguito, il server restituisce il risultato al proxy lato server, che lo impacchetta, insieme ad informazioni supplementari, in un messaggio che trasferisce poi al broker.
6. Il Broker trasferisce la risposta al proxy lato client.
7. Il proxy lato client riceve la risposta, spacchetta il messaggio e lo invia all'applicazione cliente, che continua la sua esecuzione normalmente.

- **Pattern Pipe & Filter**: è una concatenazione di elementi, cioè l'output di un'elaborazione va in ingresso ad un altro modulo

- **Pattern Client/Server**: anche questo pattern ha lo scopo di disaccoppiare l'interazione tra due entità comunicanti. La differenza rispetto al pattern Broker è che ora la comunicazione non è tra pari. Solo uno può iniziare la comunicazione, solo uno può essere attivo: il client. Il server è del tutto passivo. Nel caso della programmazione parliamo di oggetti, nel senso di programmi o applicazioni distribuite su rete, invece, si parla di client/server. I client/server possono essere distribuiti su uno, due, o n livelli.
  Ogni applicazione può essere suddivisa logicamente in 3 parti:
  - Presentazione,che gestisce l'interfaccia utente (gestione eventi grafici, controlli formali sui campi in input, help, …)
  - Logica applicativa
  - Gestione dei dati persistent

- **Pattern SOA (Service-oriented architectures)**: Le SOA rappresentano un'evoluzione delle architetture client-server. Si ha da un lato l'**utente** che a cui serve richiedere un servizio, i **provider** che si occupano di implementare un servizio e il **registro** uno spazio logico in cui sono elencati i servizi disponibili che consente ai provider di pubblicizzare i servizi e agli utenti di cercarli e interrogarli. C'è una flessibilità molto alta dovuta al registro, che disaccoppia client e server. Questa architettura serve per spiegare meglio il concetto di indipendenza dei servizi e possono essere installate sui containers, che sono delle macchine virtuali leggere.

Design Patterns

A differenza dei pattern architetturali sono pensati in fase di progettazione e non in fase di definizione dell'architettura (progettazione ad alto livello). Classifichiamo i pattern di design secondo due criteri:

lo **scopo (purpose)** che riflette cosa fa il pattern. Lo scopo può essere:

- **Creazionale**
- **Strutturale**
- **Comportamentale (behavorial)**

L'**ambito (scope)** : L'ambito specifica se il pattern è relativo a classi o ad oggetti

- **Classi**: trattano la relazioni statiche, determinate a tempo di compilazione, tra classi e sottoclassi;
- **Oggetti**: trattano relazioni dinamiche, che variano a tempo di esecuzione, tra oggetti.

PATTERN CREAZIONALI:

I design pattern creazionali astraggono il processo di istanziazione. Consentono di rendere il sistema indipendente da come gli oggetti sono creati e rappresentati. Se basati su classi, utilizzano l'ereditarietà per modificare la classe istanziata. Se basati su oggetti, delegano l'istanziazione ad altri oggetti. Incapsulano la conoscenza relativa alle classi concrete utilizzate dal sistema e nascondono come le istanze delle classi sono create e assemblate. Esempi di creational patterns:

- **Singleton**: Assicura che una classe abbia una sola istanza e fornisce un punto globale di accesso ad essa. In alcuni casi è importante che una classe abbia esattamente un'istanza. Una variabile esterna fa sì che l'oggetto sia accessibile, ma non impedisce che siano istanziati più oggetti.
  La classe stessa è responsabile di avere traccia della sua unica istanza e fornire un modo per accedere a tale istanza. Il Singleton può essere usato quando l'unica istanza deve essere estensibile con subclassing e i clients devono essere capaci di usare una istanza estesa senza modificare il loro codice.
  Il singleton definisce un'operazione **Instance()** che consente ai clients di accedere alla sua unica istanza; Instance() è un metodo di classe e può essere responsabile di creare la sua propria unica istanza. Il Singleton può essere esteso anche per consentire un numero variabile di istanze.
  Una tecnica per garantire che vi sia una sola istanza della classe consiste nel nascondere l'operazione di creazione in una operazione di classe (funzione static) e rendendo il costruttore protected.

```
class Singleton {
 public:
   static Singleton* Instance();
 protected:
   Singleton();
 private:
   static Singleton* _instance; };
Singleton* Singleton::_instance = 0;
Singleton* Singleton::Instance () {
if (_instance == 0) {
_instance = new Singleton; }
return _instance; }
```

- **Abstract Factory**: Offre una interfaccia per creare famiglie di oggetti in relazione o dipendenti, senza specificare le loro classi concrete. Ci sono casi in cui vogliamo creare differenti elementi tra loro in relazione senza legarci alle loro istanze 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.

- **Builder:** Separa la costruzione di un oggetto complesso dalla sua rappresentazione così che lo stesso processo di costruzione può creare differenti rappresentazioni.
  Il Client crea l'oggetto Director e lo configura con un Builder. Il Director notifica il Builder di costruire ogni parte del prodotto. Il Builder gestisce le richieste del Director e aggiunge parti al prodotto. Il Client recupera il prodotto dal Builder. Usiamo il pattern Builder quando:
  - L' algoritmo per creare un oggetto complesso dovrebbe essere indipendente dalle parti che costituiscono l'oggetto e da come sono assemblate
  - Il procedimento di costruzione deve consentire differenti rappresentazioni per l'oggetto che viene costruito
  - Il processo di costruzione può essere separato in passi discreti (differenza tra Builder e Abstract Factory)

PATTERN STRUTTURALI:

I pattern strutturali sono relativi a come classi e oggetti sono composti per formare strutture più grandi. I design pattern strutturali basati su classi, utilizzano l'ereditarietà per generare classi che combinano le proprietà di classi base. I design pattern strutturali basati su oggetti , mostrano come comporre oggetti per realizzare nuove funzionalità.

Esempi di pattern strutturali:

- **Façade**: (fasad) Rende più semplice l'uso di un (sotto)sistema. Fornisce un'unica interfaccia per un insieme di funzionalità sparse su più interfacce/classi. La suddivisione di un sistema in sottosistemi aiuta a ridurre la complessità. Tuttavia, occorre diminuire comunicazione e dipendenze tra i sottosistemi. Non c'è incapsulamento: L'operatore deve conoscere la struttura ed il comportamento. Utilizzare il pattern Façade:
  - Per fornire una vista semplice e di default di un sottosistema complesso;
  - Nel caso di sottosistema stratificato, è possibile utilizzare façade come entry point per ciascun livello.

  Il Facade conosce quali classi di un sottosistema sono responsabili per una richiesta e delega le richieste del client agli oggetti appropriati del sottosistema.

  Le classi del sottosistema:

  - Implementano le funzionalità del sottosistema;
  - Gestiscono il lavoro assegnato da Façade;

Il Facade promuove un accoppiamento debole fra cliente e sottosistema e nasconde al cliente le componenti del sottosistema. Il cliente può comunque, se necessario, usare direttamente le classi del sottosistema. L'accoppiamento tra i client ed il sottosistema può essere ridotto anche rendendo Façade una classe astratta con sottoclassi concrete per diverse implementazioni di un sottosistema. In tal caso, i client possono comunicare con il sottosistema attraveso l'interfaccia della classe astratta Façade.

- **Adapter**: Adatta l'interfaccia di una classe già pronta all'interfaccia che il cliente si aspetta. Talvolta una classe progettata per il riuso non è riusabile solo perché la sua interfaccia non coincide con quella del dominio specifico di un'applicazione. Usiamo Adapter quando:
  - Vogliamo utilizzare una classe esistente la cui interfaccia non risponde alle nostre necessità;
  - Si vuole realizzare una classe riusabile che coopera con classi che non necessariamente hanno un'interfaccia compatibile;
  - Occorre utilizzare sottoclassi esistenti le cui interfacce non possono essere adattate sottoclassando ciascuna di esse. Un object adapter può adattare l'interfaccia della classe base.

  Vi sono due tipi di adapter:

  1. **Object Adapter** : Basato su delega/composizione.
  2. **Class Adapter** : Basato su ereditarietà; L'adattatore eredita sia dall'interfaccia attesa sia dalla classe adattata;

La differenza fra Adapter e Facade è che entrambi sono wrapper (involucri), entrambi si basano su un'interfaccia, ma Façade la semplifica, mentre Adapter la converte.

- **Composite**: Compone oggetti in strutture ricorsive ad albero per rappresentare gerarchie parte-tutto e consentire ai clienti di trattare in modo uniforme oggetti singoli/semplici

PATTERN COMPORTAMENTALI:

si occupa del modo in cui classi o oggetti interagiscono reciprocamente

Esempi:

- **Observer**: Definiscono una dipendenza uno a molti tra oggetti in modo che quando un oggetto cambia stato, tutti gli oggetti che dipendono da lui sono notificati e aggiornati automaticamente. Gli osservatori si aggiungono/tolgono a runtime e potrebbero essere varie viste sullo stesso oggetto. L'oggetto osservato non conosce l'identità precisa degli osservatori .

  Quindi gli osservatori si registrano presso l'oggetto osservato. Quando l'oggetto osservato cambia stato, notifica tutti gli osservatori (ne invoca un metodo) . Quando notificano, ogni osservatore decide cosa fare.

GESTIONE DELLE ECCEZIONI

Sintesi:

Un anomalia è una condizione anormale che può verificarsi durante la messa in esecuzione di un programma che può determinare conseguenze fallimentari per l'intero sistema.
Un approccio per la gestione di tali anomalie è il meccanismo di gestione delle eccezioni. Il normale funzionamento del sistema viene interrotto nel momento in cui viene rilevata l'eccezione e passando tale eccezione al meccanismo di gestione delle eccezioni. In C++ gestiamo le eccezioni attraverso tre istruzioni: **try,chatch e throw**.
Try e Catch: mi segue su un blocco di determinate istruzioni. Per dichiarare l'istruzione stessa si usa il catch. Come catch gli passiamo i valori Tx e l'argomento di x, dove Tx è il tipo dell'istruzione e l'argomento è il valore dell'eccezione stessa. In un try possiamo avere anche più catch nel momento in cui si sollevano più eccezioni.
Trhow: serve per lanciare intenzionalmente un'eccezione . Se un metodo può sollevare un'eccezione il compilatore mi dice che per fare questo , essa deva essere per forza esplicitata, attraverso il trhow. Se trhow non ritrova nessun catch, richiama la funzione terminated() che determina un abort, vi è un arresto anomalo del programma. Nelle intestazioni può capitare di avere trhow(elenco dei tipi): nel momento in cui vi è un elenco tipi non ammesso si richiama la funzione unexpested() che determina allo stesso modo un abort del programma.

Per le eccezioni abbiamo un costruttore e un distruttore: il costruttore istanzia un oggetto ma non abbiamo output: nulla ci dice se l'oggetto è istanziato o meno se non è messo in esecuzione nel programma. L'eccezione permette di avere , appunto, un output. Ci dice se l'oggetto è istanziato in maniera corretta.
Per invocare un distruttore vi sono due metodi: ossia per uscire da uno scope o attraverso una delete o attraverso l'eccezione. L'eccezione non può uscire dal distruttore mentre se ne passa a quella successiva, perché si avrebbe un fallimento del sistema.

In Java, invece, si gestiscono le eccezioni in maniera diretta o indiretta.
Diretta: Quando un'eccezione viene gestita nello stesso metodo con cui è stata rilevata (try e catch)
In questo caso possiamo utilizzare tre blocchi: try catch e finally. Try mi esplicita qual è l'istruzione da controllare che mi lancia un'eccezione. Catch mi determina corpo,modulo dell'eccezione stessa lanciata. Il finally è un blocco che mi dice che quell'istruzione deve essere seguita indipendentemente che vi sono delle eccezioni, che venga o meno gestita
Indiretta: Quando non viene gestita dal metodo con cui è rilevata ma attraverso il metodo chiamante che la gestisce o la reinvia a sua volta (throw). Il sistema di gestione delle eccezioni in java prevede la loro suddivisione in due tipologie così sintetizzate:

1. **checked:** esprimono eccezioni che si riferiscono a condizioni recuperabili e che quindi possono essere gestite dal metodo chiamante; Per lanciare un'eccezione checked lo facciamo mediante il throw
2. **unchecked**: esprimono condizioni non recuperabili, generalmente dovute ad errori di programmazione e quindi non gestibili dal metodo chiamante.

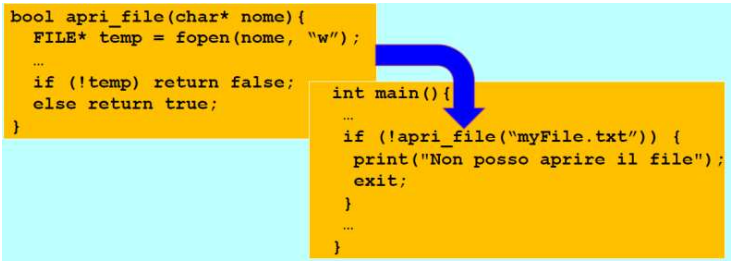
COMPLETO:

L'Anomalia (o anche errore) è una condizione anormale e inattesa che si manifesta durante l'esercizio di un sistema. Una anomalia può condurre al fallimento del sistema, cioè una deviazione dal comportamento atteso. Le anomalie devono essere opportunamente gestite dal programmatore

Idealmente le condizioni anomale vanno previste dal programmatore.
Esempio:
FILE* temp = fopen("test.txt", "r");
fscanf(temp, "%s", mystring);

Cosa succede se il file test.txt non esiste? Il programma termina in maniera anomala.
Si richiede che il blocco di codice che ha generato il fallimento restituisca delle informazioni su quanto è accaduto a chi lo ha invocato, in modo che si possa agire per gestire opportunamente l'evento (eccezionale), evitando che il programma termini in maniera anomala.

In linguaggi come il C, una condizione anomala è gestita da "codici d'errore" restituiti dalla funzione:
• un valore di tipo booleano (flag);
• uno tra *n* valori opportunamente codificati
La procedura chiamante deve controllare il valore restituito dalla funzione per conoscerne l'esito dell'esecuzione.
Tale approccio è poco efficiente:



• il controllo di eventi non standard è completamente a carico del programmatore attraverso la verifica dei valori di ritorno (cosa che non sempre viene fatta!);

• un semplice *flag* booleano contiene poche informazioni al fine di comprendere e poter gestire l'evento eccezionale;

• se i possibili valori di ritorno sono molti la procedura chiamante si "appesantisce".

Eccezioni

Una soluzione è di tipizzare le condizioni anomale, e forzare il programmatore ad invocare la soluzione dell'errore generato. Ciò è realizzato per mezzo di **eccezioni.**
• Una **eccezione** esprime una condizione anormale che si genera in una sequenza di codice durante l'esecuzione,
• Tale situazione pregiudica la regolare esecuzione e richiede un'opportuna gestione

Per gestire tali eventi è necessario gestire una alterazione occasionale (eccezionale) dell'invocazione usuale (standard) delle operazioni.
Un **meccanismo delle eccezioni** è una struttura di controllo di un linguaggio, che consente di esprimere che la continuazione standard di un'operazione debba essere sostituita da una continuazione eccezionale nel caso in cui una eccezione sia rilevata.
La continuazione eccezionale può essere definita nei vari linguaggi utilizzando apposite etichette.
Il meccanismo di gestione delle eccezioni fornisce un'alternativa alle tecniche tradizionali che sono inefficienti e ineleganti.
Viene esplicitamente separata la parte di codice che si occupa della gestione delle eccezioni dal codice ordinario, così da rendere il programma più leggibile
In un linguaggio ad oggetti, un ' eccezione può verificarsi nel corso dell'esecuzione di un metodo.

L'eccezione può essere gestita in due modalità:

- **Diretta**: l'eccezione viene intercettata e "processata" nel metodo stesso;
- **Indiretta**: il metodo che rileva l'eccezione non la gestisce e l'eccezione viene "rinviata" al (metodo) chiamante.

Istruzioni / Parole chiave

La gestione delle eccezioni del C++ utilizza le istruzioni:

- try {…} catch {…};
- throw.

```
try{
     //blocco try
}
catch(T1 arg1){
     //blocco catch
}
…
catch(TN argN){
     //blocco catch
}
```

try, catch, throw sono parole chiave del linguaggio. Le eccezioni sono tipizzate (sono oggetti istanza di un tipo). Le istruzioni che si desidera controllare devono essere contenute nel blocco try. Se all'interno del blocco try si verifica un'eccezione, nella gestione diretta essa va "catturata" tramite catch ed opportunamente gestita. Throw si utilizza sia per lanciare un'eccezione, sia per dichiarare che un metodo può generare un'eccezione

Un'eccezione lanciata ha un tipo (TX) ed è raccolta dalla corrispondente istruzione catch (catch(TX argX)), il cui blocco eseguibile gestisce l'eccezione; l'argomento (argX) riceve il valore dell'eccezione.
È possibile associare più istruzioni catch in un try. Ogni catch deve raccogliere un tipo diverso di eccezione. Se non viene generata alcuna eccezione nelle istruzioni racchiuse nel blocco try, non verrà eseguita alcuna istruzione catch.

Throw

L'istruzione throw serve a "lanciare" intenzionalmente un'eccezione:
throw *eccezione*; Affinchè un'eccezione sia raccolta, throw deve essere eseguita dall'interno dello stesso blocco try o in qualsiasi altra funzione richiamata (direttamente o indirettamente) dall'interno del blocco try. Il valore di *eccezione* è il valore "lanciato". Se si lancia un'eccezione per la quale non vi è alcuna istruzione catch verrà provocata la fine anormale del programma. Ciò avviene attraverso la chiamata della funzione terminate() che a sua volta richiama, tipicamente, la funzione abort() che termina il programma.

Try all'interno di funzioni: Un blocco try può essere inserito all'interno di una funzione. Ogni volta che il programma accede alla funzione verrà reinizializzato il sistema di gestione delle eccezioni rispetto a tale funzione.

```
void Xhandler(int test){
  try{
    if(test) throw test;
  }
  catch(int i) {
    cout << «Catturata eccezione: "
         << i << '\n';
  }
}
```

```
int main(){
  …

  Xhandler(1);
  Xhandler(2);
  Xhandler(0);
  Xhandler(3);

  …
}
```

Throw nella intestazione di funzioni. È buona norma aggiungere alla definizione di una funzione f. La clausola throw in modo da specificare le possibili eccezioni sollevabili dalla funzione stessa:
La funzione potrà lanciare eccezioni relative ai soli tipi di dati contenuti in *elenco-tipi*, separati da virgole. Ogni tentativo di lanciare un'eccezione non consentita (ovvero il cui tipo non è specificato in *elenco-tipi*) provocherà la chiamata della funzione standard unexpected() che a sua volta richiama abort() e causa la fine anormale del programma. Se si desidera indicare che una funzione non è in grado di lanciare eccezioni si dovrà utilizzare un elenco vuoto.

Tipizzazione delle eccezioni –

È possibile assegnare ad una eccezione un tipo d'utente (es.: classe, struct). In tal caso l'eccezione è un oggetto creato dal programmatore che descrive la condizione anomala verificatasi. L'oggetto eccezione può poi essere utilizzato da chi "cattura" l'eccezione.

Gerarchie di eccezioni –

Le classi definite per le eccezioni possono essere strutturate gerarchicamente (con l'ereditarietà)
In caso di eccezioni per i cui tipi esiste una gerarchia, una clausola catch con argomento una eccezione di una classe base di una gerarchia corrisponde anche alle eccezioni di ogni classe derivata da essa.

- Se si vogliono raccogliere eccezioni sia della classe base che della classe derivata occorre che nella sequenza di catch quelli relativi alla classe derivata siano posti prima di quelli relativi alla classe base;
- In caso contrario il catch della classe base raccoglierà anche le eccezioni delle classi derivate.

Try, catch, finally

La gestione delle eccezioni si basa sul meccanismo try–catch e la clausola throws nella dichiarazione di un metodo.

Nel blocco " try " , viene inserita la sequenza di istruzioni che potenzialmente lancia un'eccezione.

Se una eccezione è sollevata durante un blocco try, il resto del codice nel blocco try non è eseguito

```
try {
  …
  Istruzione;
  …
}
```

Nel blocco " catch " , si inserisce il codice per il trattamento dell ' eccezione (exception handler).

```
try {
  …
  Istruzione;
  …
} catch(TipoEccezione1 e1){
//trattamento eccezione 1
} catch(TipoEccezione2 e2){
//trattamento eccezione 2
}
```

Se un'eccezione non è intercettata dai blocchi "catch", essa sarà inoltrata al metodo chiamante.

Un blocco try può essere seguito da una successione di blocchi catch: in quale ordine collocarli?

Il flusso di controllo di un blocco try/catch dipende dalle eccezioni sollevate nel blocco **try;**
Il blocco **finally** permette al programmatore di scrivere istruzioni che verranno eseguite incondizionatamente, a prescindere dal verificarsi di eccezioni. Nel caso sia sollevata un ' eccezione, il blocco "**finally**" verrà eseguito dopo il "**try-catch**"**.** Consente di realizzare operazioni di clean-up.

Il sistema di gestione delle eccezioni in java prevede la loro suddivisione in due tipologie così sintetizzate:

3. **checked:** esprimono eccezioni che si riferiscono a condizioni recuperabili e che quindi possono essere gestite dal metodo chiamante;
4. **unchecked**: esprimono condizioni non recuperabili, generalmente dovute ad errori di programmazione e quindi non gestibili dal metodo chiamante.

VERIFICA E VALIDAZIONE

Tramite la **validazione** constatiamo se il software rispetti le esigenze dell'utente, mediante la **verifica** i requisiti.

Un software è corretto se rispetta i requisiti specificati ma non è detto che un software corretto, che quindi supera la verifica, risponda anche alle effettive esigenze dell'utente, superando anche la validazione.

Il **testing** è un processo di esecuzione del software al fine di scoprirne i malfunzionamenti , dietro ai quali vi sono difetti che vanno rimossi. Per eseguire il testing il sistema dev'essere disponibile ed eseguibile: il testing è una tecnica di programmazione dinamica. Talvolta può essere invece necessario effettuare delle verifiche prima del tempo di esecuzione, per ispezione. Tramite **l'analisi statica** del codice si prendono in esame le istruzioni del programma, senza eseguirle e si analizzano. Si cercano violazione delle regole di programmazione, che indicano la presenza di difetti. Viene analizzato un singolo modulo, controlliamo che ogni variabile sia stata inizializzata, che non ci siano variabili dichiarate ma non definite o definite ma mai usate.

Definizioni:

Il **debugging** è il processo con cui, sapendo che vi soni malfunzionamento dovuti a difetti, andiamo a identificare dove sia il difetto e a capire quale sia stato l'errore, per correggerlo.

Un **test case** è un insieme di input, condizione di esecuzione che devono verificarsi quando viene dato l'input e un criterio pass/fail in base al quale stabiliamo se il test è passato con successo (pass) o meno (fail). Un test-case ha successo se scopre un malfunzionamento non ancora scoperto

Un **test case specification** è un requisito che dev'essere soddisfatto da uno o più casi di test. Un esempio potrebbe essere una funzionalità di ricerca di uno studente tra molti altri: è logico che, essendovi più studenti, potremo cercarne vari.

Una **test suite** è un insieme di casi di test.

La **test execution** è effettivamente il test, ovvero l'esecuzione di un caso di test (definito anche dopo aver definito i requisiti) dopo l'implementazione.

Una **test obligation** è una specifica dei casi di test parziale, che richiede qualche proprietà ad alcuni casi di test che vogliamo effettuare.

Un **criterio di adeguatezza** è un predicato su una coppia (programma-test suite). Di solito, è espresso nella forma di una regola per derivare un insieme di test obligation. Il criterio è soddisfatto se ogni test obligation è soddisfatto da almeno un caso di test nella test suite.

Il testing (**tesi di Dijkstra**) non può dimostrare l'assenza totale di difetti in un software perché i test che bisognerebbe eseguire sarebbero in numero troppo elevato, impossibile da eseguire, anche nel caso di un programma semplice. Non esistono programmi senza difetti ma unicamente programmi di cui non abbiamo ancora scoperto tutti i difetti.

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

Fra le tecniche di verifica si ha:

- Testing : Volto a rilevare malfunzionamenti.
  Il testing inizia il prima possibile con la pianificazione dei test. La generazione dei casi test nelle prime fasi ha diversi vantaggi come i test generati indipendentemente dal codice, quando le specifiche sono ancora ben impresse nella mente dell'analista . La generazione dei casi di test può evidenziare inconsistenza e incompletezze delle specifiche corrispondenti

vi sono due approcci generali al testing:

BLACK-BOX TESTING : Conoscendo le specifiche funzionali, effettuare test per dimostrare che ciascuna funzione è completamente operativa, ignorando i dettagli interni .
I test sono condotti sulle interfacce software per verificare che

- I dati di input siano accettati in modo appropriato
- I dati di output siano corretti
- Sia mantenuta integrità delle informazioni esterne

Sono esaminati alcuni aspetti del modello fondamentale del sistema, senza preoccuparsi della struttura logica interna del software

WHITE-BOX TESTING : Conoscendo il funzionamento interno dei vari componenti, si effettuano test per verificare che il meccanismo interno funzioni bene, cioè che le operazioni interne siano svolte correttamente. Viene svolto un test dei cammini logici interni. (Test-case che esercitano insiemi specifici di condizioni e/o cicli). Vi è l'impossibilità in generale di eseguire un testing esaustivo di tutti i cammini, che può essere enormemente elevato anche per piccoli programmi.

Le tecniche di analisi includono **Tecniche di ispezione manuale** e **Analisi automatica**. L'analisi mira a verificare il soddisfacimento di una proprietà. Può essere applicata ad ogni fase del processo di sviluppo in particolare nella specifica e nella progettazione. L'ispezione può essere applicata ad ogni documento. Richiede una notevole quantità di tempo ed è usata principalmente :

- Quando le altre tecniche sono inapplicabili
- Quando le altre tecniche non forniscono una sufficiente copertura

- Analisi statica: prende in esame le istruzioni del programma (che sono un insieme finito), senza eseguirlo; è impossibile rilevare malfunzionamenti che dipendono dal valore assunto dinamicamente dalle variabili; è più facile rilevare anomalie legate alla presenza di variabili non dichiarate in linguaggi fortemente tipizzati quali Pascal, Ada, C++, etc.
- Analisi dinamica: prende in esame l'insieme delle esecuzioni di un programma (quasi mai limitato).

Test di unità:

Il test di unità viene effettuato al fine di rilevare i malfunzionamenti dovuti a difetti presenti nei singoli moduli. Il modulo può essere un sottoprogramma, per testarlo scriviamo un programma chiamante che lo invochi. Non è detto che i moduli chiamati da quello testato siano disponibili, dovremo allora scrivere dei moduli fittizi che ne simulino il comportamento: gli **stub**. Il driver simula i moduli chiamanti. I moduli fittizi (stub) simulano i moduli gerarchicamente inferiori del modulo da testare, stampano o restituiscono i dati nel modulo da testare. Driver e stub sono onerosi in quanto devono essere realizzati ma non fanno parte

del prodotto finale. Talvolta alcuni moduli non possono essere testati con il solo uso di driver e stub; in tal caso il testing è posposto sino al testing di integrazione.

Test di integrazione:

Il test di integrazione viene effettuato per individuare gli errori (in senso generico) relativi all'interazione tra i moduli. Le strategie adottabili sono varie, il principio generale è quello di aggiungere un numero ridotto di moduli alla volta. Faremo interagire un modulo, anziché con lo stub, con il modulo reale e vedremo se vi sono problemi nell'interazione: lo scopo è trovare gli errori dovuti alla comunicazione effettiva tra i moduli (comunicazione che nei test di unità, mediante gli stub, era fittizia). Gli eventuali problemi di comunicazione sono dovuti a problemi delle interfacce: un modulo fornisce un servizio e chi lo riceve se lo aspetta diverso.

Un'altra cosa che si vuole scoprire, tramite questo test, sono i difetti legati agli effetti collaterali. L'interazione tra due moduli può apparentemente avvenire senza problemi ma crearne altri che riscontreremo su altri moduli.

Le strategie per effettuare i test di integrazione sono

- **top-down** L'integrazione avviene seguendo la gerarchia di controllo iniziando dal MAIN ed aggregando man mano i moduli subordinati, con modalità **depth-first** o **breadthfirst.** Nell'approccio depth-first il cammino principale lungo cui effettuare l'integrazione è arbitrario. Nell'approccio breadth-first i moduli vengono aggregati muovendosi orizzontalmente in ciascun livello gerarchico. Il processo avviene in 4 passi
    1. Il main è usato come driver del test e gli stub sono costituiti da tutti i moduli direttamente subordinati. (Gli stub sono sostituiti man mano con i moduli reali)
    2. Dopo l ' integrazione di ciascun modulo, si eseguono gli opportuni test
    3. Al completamento di ciascun test, un altro modulo reale viene aggiunto, sostituendolo allo stub.
    4. Può essere condotto il testing di regressione per accettarsi che non siano stati introdotti nuovi errori. Il processo continua dal passo 2 finché non si ottiene la completa integrazione del sistema. Ad ogni sostituzione di uno stub devono essere effettuati test per verificarne l'interfaccia
- **bottom-up**: L'integrazione inizia partendo dai moduli al livello più basso (quelli senza figli) e viene eliminata la necessità di stub .
  Strategia in più passi:
    1. I moduli di basso livello vengono aggregati in gruppi (cluster) che eseguono una specifica sottofunzione software
    2. Viene realizzato un driver per coordinare l ' esecuzione
    3. Si esegue il test del cluster ! I driver sono sostituiti dai moduli reali che vengono aggregati nel programma

Test di sistema:

Scopo del test è la verifica che tutti gli elementi del sistema siano stati correttamente integrati e svolgano bene le funzioni loro assegnate.

Test di accettazione:

Test di accettazione 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 sono corretti e che possono portare al degrado del sistema. Il **beta-test** è un test di accettazione condotto dall'utente finale a cui lo

sviluppatore, generalmente, non è presente. Serve al produttore in quanto avrà un riscontro, a seguito del caso farà le opportune modifiche o meno, per poi rilasciare il prodotto. Ovviamente il cliente che lo segue ha un costo, a fronte del quale gli vengono concessi dei vantaggi. Questi possono essere o uno sconto sul prodotto finale o il vantaggio d'anteprima. Prima di fare ciò, dopo il test di sistema, facciamo utilizzare il software a uno dei nostri dipendenti, ottenendo altri riscontri. Parliamo di **alpha-test**. E' meno dispendioso in quanto il dipendente è pagato direttamente da noi. Avviene presso lo sviluppatore, in ambiente controllato. Non siamo sempre in grado di eseguire un alpha-test.

A & T:

Le tecniche di **analisi e testing (A & T)** sono moltissime. Abbiamo la fase di planning, di verifica delle specifiche, di generazione dei test, di esecuzione dei test e quella di miglioramento del processo.

- **Planning**: Pianifichiamo l'identificazione delle qualità e i test di accettazione all'inizio, i test di sistema invece quando abbiamo specificato i requisiti. i test da fare sono stabiliti appena fatte le specifiche ma vengono pensati già durante la specifica. I requisiti infatti devono essere testabili, se non siamo in grado di pianificare un test non sono requisiti buoni. Pianifichiamo i test di unità e di integrazione durante la progettazione.

- **Verifica delle specifiche**: validiamo le specifiche durante la fase di specifica e anche un po' oltre. L'ispezione della progettazione architetturale viene fatta passo per passo. Ispezioniamo i documenti della progettazione di basso livello, allo stesso modo, man mano che vengono prodotti. Controlliamo il codice mentre viene scritto.

- **Generazione dei test:** i test di sistema vengono generati dopo la specifica dei requisiti. In seguito alla progettazione ad alto livello generiamo i test di integrazione mentre quelli di unità dopo la progettazione a basso livello (riguardano infatti i singoli moduli, dunque più dettagliati).

-**Eseguiamo i test** di unità appena dopo averli sviluppati. Gli oracoli vengono progettati appena dopo aver definito i test. eseguiamo il test di integrità e dopo l'integrazione il test di sistema. Dunque, A & T accompagnano tutto il ciclo di vita del software.

Test Funzionale

Si tratta di derivazione dei casi di test dalle specifiche. Funzionale perché si riferisce alla sorgente di informazione usata nella progettazione dei casi di test.

Le specifiche funzionali sono descrizione del comportamento atteso del programma.

Perché il test funzionale?

- **E' la tecnica base per progettare i casi di test** : Spesso è utile nel raffinare le specifiche.
- **Efficiente** : Trova delle classi di fault che possono eludere altri approcci (e.g., missing logic)
- **Ampiamente applicabile** : A ogni descrizione del comportamento del programma che serve da specifica e a ogni livello di granularità, dal test di unità al test di sistema
- **Economico** : Tipicamente meno costoso da progettare ed eseguire rispetto ai casi di test strutturali (basati sul codice)

Progettazione dei test preliminare

Il codice del programma non è necessario , serve solo una descrizione del comportamento atteso, inoltre, possono essere usate anche specifiche informali ed incomplete. Posso scrivere casi di test nelle primissime fasi, quando ancora non ho sviluppato il codice.

La progettazione preliminare (già nelle prima fasi del ciclo di sviluppo) comporta benefici
• Spesso rileva ambiguità e inconsistenza nelle specifiche
• Utile per stimare la testabilità
• Per migliorare il piano di test ed il budget grazie al miglioramento delle specifiche
• Utile nel supportare la descrizione delle specifiche

Nel caso estremo, come nella metodologia XP(extreme programming), c'è la pratica di TDD(ossia non scrivo il codice ma il caso di test, cioè cosa vorrei che il codice facesse.

Random testing

Il test di tipo casuale è utilizzato, tipicamente, non per cercare dei difetti ma per fare delle stime di affidabilità. Consideriamo la ricerca di difetti in un software come la ricerca di aghi in un pagliaio.
Per avere una stima dell'affidabilità del software in operazione il random test vede il problema come un campionamento, ossia gli insieme degli input (la popolazione da campionare) e i casi di test sono il campione.

L'obiettivo del campione è quello di rappresentare una determinata quantità che si riferisce alla popolazione(es. la media del reddito dei cittadini)
La stima è la probabilità di fallire: il sistema accetta 10 mila input, tra questi ce ne sono 1000 che causano un fallimento—> si pesca un campione, se 2 su 100 falliscono ho la probabilita del 2%.

Svantaggi: La Distribuzione dei fault(fallimenti) non è uniforme !
Vantaggi: si ha una stima dell'affidabilità reale, inoltre non sfruttando nessuna conoscenza specifica è facilmente automatizzabile

Test Funzionale vs. Strutturale: Il test funzionale è migliore per fault di tipo missing logic
• Un problema comune: alcune parti della logica del programma vengono semplicemente dimenticate
• Il test strutturale (basato sul codice) non si focalizzerà mai su codice che non esiste

Cerca di rilevare difetti compresi nelle seguenti categorie:
• Funzioni incomplete o mancanti
• Errori di interfaccia (Il test strutturale non mostra errori di interfaccia perché è utilizzato sul singolo modulo)
• Errori nelle strutture dati o negli accessi a data base esterni
• Errori di presentazione
• Errori di inizializzazione e terminazione
Non è alternativo, ma complementare al test strutturale

Il test funzionale si applica a tutti i livelli di granularità, mentre il test strutturale si applica tipicamente solo all'unità

Partition Testing

Ossia il test che utilizza le partizioni: Si vede l'insieme dello spazio degli input come una partizione, ossia come un insieme di classi di input la cui totalità forma l'insieme stesso. L'obiettivo è quello di isolare regioni dello spazio degli input in cui ci sono i fallimenti. L'idea di fondo è che se riesco ad isolare queste classi, mi basta un solo input di quella classe per rilevare il difetto che causa i fallimenti.
Sfruttare conoscenza per scegliere campioni che più probabilmente includono regioni dello spazio degli input "speciali" o propense a causare failures (failure-causing inputs). I **failure-causing inputs** sono sparsi nell'intero spazio degli input ma possiamo trovare regioni in cui sono densi.

Si chiama **(Quasi-)Partition Testing** un metodo di testing che divide l'insieme ("infinito") di casi di test in un insieme finito di classi, la cui unione è l'intero spazio.
("Quasi-" perché le classi possono sovrapporsi.)

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** o spesso solo di functional testing . Il caso desiderabile è che ogni fault porta a failures che sono densi (facili da trovare) in alcune classi di input.

Test funzionale: sfruttare le specifiche

Il test funzionale usa la specifica (formale o informale) per partizionare lo spazio degli input . I test sono condotti sulle interfacce software per verificare che:
• I dati di input sono accettati in modo appropriato
• I dati di output sono corretti
• Mantenuta integrità delle informazioni esterne

Sono esaminati alcuni aspetti del modello fondamentale del sistema, senza preoccuparsi della struttura logica interna del software. Il dominio dei dati di input è suddiviso in classi: Una classe rappresenta un insieme di stati validi o non validi per le condizioni d'ingresso.
Un test-case ideale rileva da solo una classe di fault (es. una scorretta elaborazione di tutti i dati di tipo char), che altrimenti richiederebbe l'esecuzione di molti test, prima di intuire il fault generico.

Approcci al test funzionale: dalla specifica ai casi di test
1. Decomporre la specifica
2. Selezionare dei rappresentanti : Valori rappresentativi di ogni input oppure o Comportamenti rappresentativi di un modello .
3. Formare le specifiche dei test
4. Produrre ed eseguire i test effettivi

Test Combinatoriale

Il test combinatoriale consiste nella strutturazione (manuale) delle specifiche di test in un insieme di proprietà o attributi che possono essere sistematicamente variate in insiemi di valori.
Vantaggi:
• I test sono automatizzabili
• I test sono caratterizzati con attributi che possono essere combinati fra loro per ottenere delle combinazioni "rappresentative" per il test di una funzionalità.

L'idea di base è Identificare gli attributi distinti e generare delle combinazioni degli attributi per il testing. Data una specifica ci sono più approcci per derivare i casi di test: sono le tecniche di tipo combinatoriale (combinatorial testing):

- **Category-partition testing**: Nasce dalla presenza di molti vincoli nel domino di input che suggerisce un metodo di partizionamento con vincoli. Identifica manualmente gli attributi che caratterizzano lo spazio di input. Gli attributi variano all'interno di un set di valori. Applica i vincoli manualmente per ridurre il numero dei casi di test. Decompone le funzionalità che possono essere testate indipendentemente. Per ogni funzionalità si individuano i parametri che la descrivono e gli elementi da cui dipende. Per ogni parametro ed elemento si individuano un insieme di caratteristiche elementari, dette categorie . Identifica i valori rappresentativi. Per ogni categoria si identifica classi di valori rappresentativi: Normal values , Special Values , Boundary Values ,Error Values . Introduce i vincoli sui valori individuati: si impone che le combinazioni possono avere al più un errore.

  Tuttavia, talvolta inserire dei vincoli non è facile e si rischia si inserirne alcuni senza una motivazione razionale.

- **Pairwise testing**: Al contrario, nasce dalla presenza di valori di input con pochi o nessun vincolo che suggeriscono un approccio combinatoriale**.** Test di tipo combinatorio che usa un numero limitato dei valori di input (coppie, triple). L'idea alla base è che la maggior parte dei fallimenti software sono dovuti a combinazioni di pochi input (coppie, triple). Si copre "la maggior" parte dei fault con pochi test. L'individuazione di coppie (triple) di valori di input può essere proibitiva se esistono diversi parametri del sistema, tuttavia si usano euristiche che semplificano la ricerca dei parametri. Identificano gli attributi che caratterizzano lo spazio di input. Gli input variano su un set di valori opportunamente ridotto, considerando coppie dei valori degli attributi (a due a due). Si possono considerare anche combinazioni n-arie per n>2 (a tre a tre, a quattro a quattro, ecc.). Per il Pairwise Testing si considerano solo coppie e triple, a coppie cioè cerca di derivare un insieme di casi di test in cui ogni coppia è presente almeno una volta.

  Si possono introdurre dei vincoli che riducono notevolemente il numero di test case
  - **Vincolo error** : Il vincolo errore indica una classe di valori che dà luogo ad un errore. Si può ipotizzare che sia sufficiente un solo errore per combinazione per osservare un potenziale fallimento del sistema
  - **Vincolo single** : 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]
  - **Vincolo property** : Il vincolo single[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. Nel calcolo del numero di test case si comporta come il vincolo [error] ma ha una motivazione diversa che ne giustifica l'uso.

TEST STRUTTURALE

Valuta la completezza della test suite in base alla struttura del programma stesso. (è un white box)

Si distingue dal functional ("black-box") testing:

- 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 (basata sulla copertura di elementi della struttura anziché delle specifiche)
  Ciò implica l'utilizzo di tecniche diverse per derivare i casi di test

Il test strutturale viene utilizzato se parte di un programma non è eseguita (tipicamente un elemento del control flow) da alcun caso di test, i difetti in quella parte non possono essere rilevati e completa il test funzionale. Non basta solo il test funzionale per la tipologia degli errori che non possono essere rilevati dal test black-box.

Con il test strutturale viene svolto un test dei cammini logici interni , ma vi è difficoltà nell'eseguire un testing esaustivo di tutti i cammini, che può essere enormemente elevato anche per piccoli programmi.

Per decidere quali test effettuare passiamo attraverso dei modelli del programma:

Control Flow Graph (CFG): Grafo orientato che rappresenta il trasferimento del flusso di controllo tra blocchi di istruzioni . Un **nodo** rappresenta un blocco di 1 o più istruzioni, tutte eseguite se il flusso di controllo raggiunge la prima di esse. Un **arco** rappresenta il trasferimento del flusso di controllo tra la coppia di nodi che esso collega. Ogni nodo rappresenta una componente 1-in/1-out, in cui è possibile uno e solamente un cammino di controllo aciclico: nessuna decisione, né congiunzione di flussi, può interrompere la sequenza delle istruzioni incluse nel nodo. Solo alla fine può esservi una ramificazione e solo all'inizio può esservi una congiunzione

Call Graph (CG): Grafo orientato che rappresenta le relazioni chiamante-chiamato tra procedure. I **nodi** rappresentano le procedure e gli archi le relazioni di chiamata tra esse. Un **arco** è rappresentativo di tutte le chiamate che possono sussistere tra 2 nodi (nel caso si usi un arco per ogni chiamata si ha un multi-call graph). Vi sarà un nodo che non ha archi entranti, che corrisponde il programma principale

I casi di test sono derivati a partire dagli elementi strutturali(per es. del CFG), non dalle specifiche. La creazione è guidata da una misura di **coverage** che identifica quanti elementi sono stati eseguiti. Si progettano test-case in modo che: tutti i cammini indipendenti all'interno di un modulo siano stati esercitati almeno una volta e siano esercitate tutte le decisioni logiche nei casi vero e falso

COPERTURA DEGLI STATEMENT: Richiede che ogni istruzione sia eseguita almeno una volta. Coprire tutti gli statement non implica eseguire tutti i rami.

COPERTURA DELLE DECISIONI: Richiede che ogni arco del control flow graph sia percorso ameno una volta. Un test dovrà contenere almeno 2 dati di test per ciascuna decisione, uno per ogni ramo uscente da un nodo decisione. Il criterio di copertura delle decisioni include quello di copertura degli statement

COPERTURA DELLE CONDIZIONI: Richiede che ogni singola condizione che compare nelle decisioni del programma valga sia vero che falso per diversi dati di test

COPERTURA DEI CAMMINI DI BASE: Richiede che ciascun cammino di un insieme di cammini di base sia eseguito almeno una volta

Modified Condition/Decision (MC/DC):

Il criterio delle condizioni causa una crescita esponenziale del numero di casi di test. Il MC/DC non è di facile applicazione ma è vantaggioso in quanto riduciamo il numero di complessità rendendo lineare la crescita del numero di casi di test, rispetto al numero di percorsi. Secondo questo criterio non testiamo tutte le condizioni ma solo quelle rilevanti, ovvero le decisioni che influenzano indipendentemente l'esito di una condizione composta. Individuare tali condizioni è un'attività brain-intensive, non automatizzabile, di conseguenza non è un criterio di applicazione immediata ma permette di ridurre notevolmente il numero di casi di test.
Richiede per ogni condizione C due casi di test:

- I valori di tutte le condizioni valutate tranne C sono gli stessi
- La condizione composta vale true per un caso di test e false per l'altro

La copertura del criterio MC/DC è "implicata" da quella del criterio delle condizioni composte e implica tutti gli altri. Dunque è un buon compromesso tra completezza e numero di casi di test (e per questo ampiamente usato).

Numero ciclomatico di Mc Cabe:

il numero ciclomatico o numero di Mc Cabe, ovvero il numero dei cammini indipendenti nell'insieme di base di un programma, è una metrica di complessità. Tale numero è legato al numero di test che dobbiamo effettuare, ovvero almeno quanti i cammini indipendenti. Possiamo calcolarlo in tre modi:

1. V(g)= Numero di regioni del grafo
2. V(g)= F-N+2*C dove F=archi e N=nodi
3. V(g)= P+1 dove P=nodi predicati

Manutenzione del software

La manutenzione è la fase del ciclo di vita di un software che consente di intervenire sul software per migliorarne la qualità.

Il software si deteriora perché, a partire dal rilascio, cambiano le esigenze e quindi si mette mano alla struttura del codice: ma si introducono cambiamenti che non sono pianificati, quindi non sono integrati con il resto del sistema ma solo per risolvere quella particolare situazione.

Classi di manutenzione

• **Manutenzione correttiva** – Modifiche per correggere difetti
• **Manutenzione adattativa** – Modifiche per adattare il software a cambiamenti dell'ambiente operativo (hardware, software di base, interfacce, organizzazione, legislazione, ecc.)
• **Manutenzione perfettiva** – Estensione dei requisiti funzionali, o migliorie di requisiti non funzionali in risposta a richieste dell'utente *(agile)*
• **Manutenzione preventiva** – Modifiche che rendono più semplici le correzioni, gli adattamenti e le migliorie

Problemi della manutenzione

Il problema principale è che non c'è adeguata pianificazione, il software dovrebbe essere progettato per cambiare. Altri problemi possono essere: difficoltà nel comprendere un programma scritto da altri; mancanza di documentazione completa/ consistente ; software non progettato per modifiche future; difficoltà nel tradurre una richiesta di modifica di funzionamento del sistema in una modifica del software; valutazione dell'impatto di ciascuna modifica sull'intero sistema ; la gestione della configurazione del software ; la necessità di ritestare il sistema dopo le modifiche

Modelli di processo per la manutenzione del software

Modello di riparazione veloce (Quick-fix model): modifica il codice in termini di patches ('pezze') ; è veloce ed economico sul breve termine ; degrada le strutture; si ha una documentazione modificata a posteriori . Si parte dal codice. Tocco il codice e vado a vedere come questo codice impatta sul progetto; i requisiti e i casi di test.

Modello di miglioramento iterativo (iterative-enhancement model). si ha una valutazione preventiva dell'impatto della modifica; si decide se lavorare su componenti esistenti o sviluppare nuove componenti; va a preservare la struttura; è lento e costoso sul breve termine; si ha una documentazione modificata in anticipo. Si effettua un'analisi di ciò che devo cambiare e intervengo sul sistema a partire dai requisiti.

> **A 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.

Gli interventi **per 'Ringiovanire' il software** sono finalizzati a migliorare la manutenibilità di un software ormai deteriorato dagli interventi di manutenzione subiti.
Diversi tipi di intervento possibili:

- **Ridocumentazione**
- **Restructuring (o Refactoring)**
- **Reengineering**
- **Reverse Engineering**

Ridocumentazione:

Parto dal codice sorgente e devo applicare strumenti di analisi statica del codice. L'analisi statica è una tipologia di tecnica che mira a verificare delle proprietà del software.
Consiste in una analisi statica del codice sorgente (attraverso appositi strumenti) al fine di produrre documentazione del sistema.
Si analizzano: usi delle variabili, chiamate fra componenti, path del flusso di controllo, dimensioni dei componenti, parametri di chiamate, per capire cosa fa il codice e come lo fa.
Gli output di una attività di ridocumentazione possono essere: grafi delle chiamate,tabelle delle interfacce delle funzioni, dizionari dati, diagrammi del data-flow o control-flow, pseudo-codice, cross-reference fra componenti o variabili. Tali output si possono usare per verificare se il software ha bisogno di ristrutturazione.

Restructuring o Refactoring

Attività che trasforma il codice esistente in codice equivalente dal punto di vista funzionale, ma migliorato dal punto di vista della sua qualità.
In genere si esegue in tre passi:
1. Analisi statica del codice per ottenerne una rappresentazione interna (es. call graph, control-flow graph…)
2. Semplificazione della rappresentazione interna attraverso tecniche di trasformazione automatiche.
3. La nuova rappresentazione viene usata per generare una versione strutturata (migliorata) ed equivalente al codice originario

Refactoring

Il refactoring è un insieme di tecniche usabili per migliorare il codice ed il design di sistemi objectoriented. Tali tecniche cercano di eliminare i cosiddetti **Bad Smells** dal codice. Si utilizzano dei pattern per capire cosa non va nel codice e correggerlo. I pattern sono soluzioni preconfezionate a problemi ricorrenti.

Bad Smells\Code Smells : ossia "codice che puzza".
si identificano delle parti del codice che potenzialmente sono scritti male, dunque, causeranno dei problemi. Non sono scritti secondi i migliori principi dell'ingegneria del software

Reengineering

La reingegnerizzazione (reengineering) è un'attività di re- implementazione di un sistema software svolta per migliorare la manutenibilità di un software esistente.
Essa può comprendere:
Ridocumentazione, Ristrutturazione e Rriscrittura di parte del software (o anche di tutto) senza modificare l'insieme di funzionalità che esso realizza

Obiettivi del Reengineering
• Modularizzazione del sistema – Suddivisione di un sistema monolitico in parti da riusare separatamente
• Miglioramento delle Performance – Migliorare le prestazioni di un sistema esistente
• Migrazione (o Porting) verso altre Piattaforme – Necessità di localizzare I componenti dipendenti dalla

piattaforma
• Estrazione del progetto – Per migliorare maintainability, portability, etc.
• Usare una nuova Tecnologia – quali nuove caratteristiche di un linguaggio, standards, librerie, etc

Il ciclo di vita del Reengineering: Parto dal codice per estrarre il modello, individuo i problemi di scarsa qualità e riparto da una nuova progettazione del sistema. Prevede un backword engineering, a partire del codice fa un recupero della documentazione e una forward engineering

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.

Prevede due fasi:

Estrazione – Analisi del codice o di altri artifatti software, allo scopo di ottenere informazioni relative al sistema analizzato. Particolarmente utili sono quelli strumenti in grado di estrarre informazioni da un codice sorgente qualsiasi, nota che sia la grammatica del linguaggio di programmazione (ad esempio JavaCC)

Astrazione – Si esaminano le informazioni estratte e si cercano di astrarre diagrammi, o viste, ad un più alto livello di astrazione (es.: diagrammi di progetto, architetturali, del dominio dei dati) – I processi di astrazione non sono completamente automatizzabili poichè necessitano di conoscenza ed esperienza umana

LEGACY SYSTEM:

Un **sistema legacy**, in informatica, è un **sistema** informatico, un'applicazione o un componente obsoleto, che continua ad essere usato poiché l'utente (di solito un'organizzazione) non intende o non può rimpiazzarlo. Un sistema legacy ("ereditato") é spesso vecchio (10 anni o più di vita); É di grandi dimensioni (centinania di migliaia di lineedi codice). É scritto in assembly o in un linguaggio di vecchia generazione. É 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. La manutenzione risulta ormai difficile e costosa. Realizza funzionalità cruciali e irrinunciabili per l'organizzazione. Contiene anni di esperienza accumulata nell'ambito del dominio specifico del problema

Un sistema legacy obbliga il management a cercare soluzioni e strategie di manutenzione vincenti! Svariate alternative disponibili:
– Eliminazione del sistema e sviluppo di un nuovo sistema
– Recupero dei componenti più preziosi del sistema, sostituendo i restanti con prodotti preconfezionati (COTS)
– Eliminazione dei componenti ormai inutili, del codice obsoleto e dei dati "morti"
– Congelamento del sistema as is, e riutilizzo mediante tecniche di wrapping
– Manutenzione Ordinaria
– Manutenzione Preventiva (re-documentation, restructuring o reengineering)
– Migrazione …

Occorre 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)?

Legacy system categories :

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 (necessarie) operazioni di manutenzione 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 (se realizza qualcosa di generale, indipendente dal dominio specifico) oppure mantenerlo (dato che i costi di manutenzione saranno limitati)

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

Valutazione del valore economico

L'Assessment del Valore di Business dovrebbe prendere in considerazione diversi punti di vista per valutare quanto è importante il processo aziendale per il raggiungimento degli obiettivi aziendali.
– Utenti del sistema, clienti, Line managers, IT managers, Senior managers.
É necessario condurre interviste e raccogliere I risultati su: – Utilizzo del sistema – Processi aziendali supportati – Fidatezza del sistema – Gli output prodotti

Valutazione della qualità tecnica

Richiede due tipi diversi di assessment:
• Assessment dell'ambiente (environment) – Quanto è efficace l'ambiente operativo del sistema e quanto costa mantenerlo?
• Application assessment – Qual è la qualità dell'applicazione software?

Metriche del software

Misure nel processo di produzione del software

La gestione di un progetto software richiede di monitorarne l'avanzamento e di valutarne quantitativamente caratteristiche relative al prodotto finale e non solo.
Si ha quindi necessità di tecniche per "**misurare**" opportune grandezze al fine verificare se vi sono sono ritardi nel progetto, se si è consumato o si sta consumando di più (o di meno) del previsto in termini di risorse e se il software prodotto rispetta i requisiti di qualità.
Queste verifiche riguardano 3 fattori, del cosiddetto triangolo di ferro (**qualità, costo e tempo**)

Ciò richiede:
• **Stima a priori** di tali grandezze.
• **Misurazione** di tali grandezze**, durante il processo** o **a posteriori**.

Una **metrica** caratterizza in modo quantitativo e misurabile attributi semplici di una entità
Una **misura** è una assegnazione oggettiva ed empirica di un valore ad un attributo.

Al modello di qualità è associata una distinzione tra:
Attributi esterni: visibili e di interesse per l'utente (facilità d'uso, prestazioni, efficienza, …)
Attributi interni: visibili e di interesse per i produttori/sviluppatori (modularità, complessità, …)
In generale accade che
- le metriche sono associate ad attributi interni
- gli attributi esterni dipendono da attributi interni e sono più complessi.

Le **metriche** del software sono lo strumento attraverso il quale l'ingegnere del software predice e/o misura gli aspetti di processo, di gestione delle risorse, e di prodotto, rilevanti nell'ingegnerizzazione del software. Servono da guida nella ricerca di soluzioni efficienti ed efficaci a problemi complessi.

Due classificazioni, non necessariamente in contrapposizione
- Metriche di prodotto
- Metriche di processo
- Metriche di qualità
- Metriche funzionali
- Metriche strutturali
- Metriche dimensionali

Metriche di Prodotto

Sono:

- **Metriche dimensionali**

La *dimensione del programma* è generalmente valutata in termini di ***numero di linee di codice (LOC)***
È un indicatore considerato tradizionalmente importante, specie se riferito a linguaggi di programmazione differenti, dove diverse sono le densità di significato per ogni linea di codice.
Non è ancora chiaro se nel conteggio siano da considerare anche i commenti (in genere no), utili per la documentazione, e la parte dichiarativa, che contribuisce in maniera rilevante a determinare la qualità del SW.
Altre metriche dimensionali legate alle LoC sono:

- **NCNB= no-comment no-blank:** calcola tutte le linee di codice eccetto commenti e linee vuote.
- **CLoC= Linee di commento:** CP= (CLoC) / (NLoC+CLoC) , utile anche come metrica di qualità

Altre misure legate alle LoC sono: produttività e qualità.
La produttività, in generale, indica quanto lavoro è stato portato a termine rispetto allo sforzo(lo sforzo può essere o il tempo o il numero di mesi uomo). **Produttività (P)** P = LOC / M (M = num. mesi uomo; tempo

occorso per sviluppare il programma)

La qualità, può essere calcolata come l'inverso, ossia quanti difetti per numero di linee di codice.
Qualità (Q) Q = LOC / E (E = numero errori rilevati. 1/Q dà la densità di errori rilevati nel codice)

LOC misura la lunghezza del codice, non va considerata come un indicatore della sua complessità
La metrica descritta permette di *misurare* a posteriori alcune caratteristiche di un programma e del suo sviluppo.

- **Metriche di complessità**

Numero ciclomatico: il numero ciclomatico o numero di Mc Cabe, ovvero il numero dei cammini indipendenti nell'insieme di base di un programma, è una metrica di complessità. Tale numero è legato al numero di test che dobbiamo effettuare, ovvero almeno quanti i cammini indipendenti. Possiamo calcolarlo in tre modi:

1. V(g)= Numero di regioni del grafo
2. V(g)= F-N+2*C dove F=archi e N=nodi
3. V(g)= P+1 dove P=nodi predicati

Ancora una metrica di complessità: Flusso di Informazione.
C = L * (Fan-in * Fan-out)2
L: metrica dimensionale (LoC)
Fan-in: numero di flussi entranti nel componente
Fan-out: numero di flussi uscenti dal componente

Metriche di processo

Servono per la valutazione di un progetto

- **Metriche di Modularità:** Organizzative, di risorse, di personale, di gestione tecnologica
- **Metriche di Gestione** Del progetto, di configurazione
- **Metriche del ciclo di vita**: Di definizione del problema, di analisi, di progetto, di implementazione

Metriche di Qualità

Metriche relative ai fattori di qualità; ad es.:

- **Metriche di dependability:**
- Affidabilità (probabilità di fallimento in [0,t])
- MTTF (Mean Time To Failure), MTBF (Mean Time Between Failures)
- Disponibilità (probabilità di fallimento in *t*)
- **Metriche di manutenibilità** (ad es., di correggibilità, di espandibilità)
- **Metriche di usabilità, di prestazioni**, ecc.

Metriche di Progetto

La diffusione dei linguaggi OO ha reso obsolete alcune metriche tradizionali che consideravano dati e funzioni
Si focalizzano sulla struttura interna dell'oggetto e sulla complessità esterna, che misura l'interazione tra le entità

Metriche di complessità di **Chidamber e Kemerer (CK)**:
Sei metriche:
• *Weighted Method per Class*: definita come numero di metodi per classe *oppure* come somma delle complessità di tutti i metodi delle classi
• *Response for a class*: cardinalità dell'insieme di tutti i metodi che possono essere chiamati in risposta ad un messaggio ricevuto da un oggetto della classe.
• *Coupling between object classes*: misura dell'accoppiamento. Numero di classi con la quale una data classe è accoppiata
• *Lack of Cohesion*: misura della (mancanza di) coesione. Misura il grado di diversità fra metodi in relazione alle variabili usate. Si può calcolare contando gli insiemi disgiunti prodotti dall'intersezione fra insiemi di attributi usati dai metodi. I metodi sono più simili se operano sugli stessi attributi
• *Depth of inheritance tree*: rappresenta la profondità dell'albero in una gerarchia di classi. Una metrica di supporto è il *Number of Methods Inherited*
• *Number of children*: numero di sottoclassi immediatamente subordinate a una certa classe della gerarchia Maggiore è tale numero, maggiore è il riuso. Tuttavia, maggiore è tale numero maggiore è la probabilità di avere una astrazione errata

Metriche di pianificazione

Application size:
• Danno una misura della quantità di lavoro necessaria
Ad es., *Number of key classes, Number of support classes, Number of subsystems*
Staffing size
• Danno una misura del personale necessario
Ad es., *Person-Days per Class, Classes per Developer*
S**cheduling**
• Forniscono uno strumento di pianificazione temporale dei processi nell'ambito della gestione dei progetti

- **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.

- **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.

- **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.

- **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