Teoria

Introduzione

Un prodotto software è costituito da codice e documentazione, che includono anche artefatti intermedi come i documenti di requisiti, specifica e progetto. Il codice rappresenta il prodotto finale, e l'insieme organizzato di questi elementi costituisce il sistema software. Nel ciclo di sviluppo, il cliente ordina il prodotto, lo sviluppatore lo realizza, e l'utente lo utilizza; se cliente e sviluppatore coincidono, si parla di software interno, altrimenti di software a contratto.

L’affidabilità del software è un concetto complesso e dipende dall’utente e dal modo in cui utilizza il software: difetti che si manifestano per un utente possono non emergere per un altro, a causa di diversi profili operativi.

Un principio importante in questo ambito è la regola 10-90, che deriva da esperimenti su programmi di grandi dimensioni: il 90% del tempo di esecuzione del programma viene speso eseguendo solo il 10% delle istruzioni totali, che costituiscono il cosiddetto core (nucleo) del programma. Questo significa che gran parte dell’attenzione per ottimizzazione e affidabilità va concentrata proprio su questa parte centrale, che è quella maggiormente utilizzata e critica.

A differenza dell’hardware, i guasti software non sono causati da usura o deterioramento fisico, ma da difetti latenti nel codice che emergono solo in determinate condizioni. L’hardware si ripara sostituendo componenti danneggiati, mentre il software necessita di interventi di manutenzione per correggere errori, che possono influire sull’affidabilità in modo non sempre prevedibile.

Di conseguenza, le metriche di affidabilità hardware non si adattano direttamente al software. Mentre l’obiettivo hardware è mantenere stabile la frequenza di guasto, nel software si mira a far diminuire questa frequenza nel tempo attraverso miglioramenti continui.

Infine, la disponibilità del software rappresenta la percentuale di tempo in cui il software è utilizzabile, influenzata dalla frequenza dei guasti e dal tempo di riparazione. Queste metriche sono particolarmente critiche in sistemi dove l’interruzione può causare gravi perdite, come nei trasporti, nel controllo del traffico aereo, nella gestione del volo, nella distribuzione di energia e nelle comunicazioni.

Processo Software

Una serie di attività necessarie alla realizzazione del prodotto software nei tempi, con i costi e con le desiderate caratteristiche di qualità. Il processo software segue un ciclo di vita che si articola in 3 stadi (sviluppo, manutenzione, dismissione). Nel primo stadio si possono riconoscere due tipi di fasi:

- fase di **definizione** (OOA)

- fase di **produzione** (OOD)

Lo stadio di manutenzione è a supporto del software realizzato e prevede fasi di definizione e/o produzione al suo interno.
Durante ogni fase si procede ad efettuare il `testing` di quanto prodotto, mediante opportune techine di verifica e validazione. Ci sono vari tipo di manutenzione:

- **Manutenzione correttiva**, che ha lo scopo di eliminare i difetti che producono guasti del software.
- **Manutenzione adattiva**, che ha lo scopo di adattare il software ad eventuali cambiamenti a cui è sottoposto l'ambiente operativo per cui è stato sviluppato.
- **Manutenzione perfettiva**, ha lo spoco di estendere il software per accomodare nuove funzionalità.
- **Manutenzione preventiva**, consiste nell'eeffetuare modifiche che rendano più semplici le correzioni, gli adattamenti e le migliorie.

Dunque, per ciclo di vita del software indendiamo l'intervallo di tempo che intercorre tra l'istante in cui nasce l'esigenza di costruire un prodotto software e l'istante in cui il prodotto viene dismesso. Include le fasi di defizione dei requisiti, specifica, pianificazione, progetto preliminare, progetto dettagliato, codifica, integrazione, testing, uso, matunenzione e dimissione.

Modelli

Il modello del ciclo di vita del software specifica la serie di fasi attraverso cui il prodotto software progredisce e l'ordine con cui vanno eseguite, dalla definizione dei requisiti alla dismissione.

Modello Build & Fix

Il modello Build & Fix è uno dei modelli di sviluppo software più semplici e primitivi, spesso adottato in contesti non professionali o da sviluppatori alle prime armi. In questo approccio, il software viene costruito senza una vera analisi dei requisiti nè una progettazione preliminare, e viene continuamente corretto e modificato man mano che emergono problemi e richieste.

Consiste in un semplice ciclo:

1. Build: Si scrive il codice in base a una vaga idea di cosa debba fare il programma.
2. Fix: Quando emergono problemi o richieste, si corregge modificando direttamente il codice.
3. Si ripete il ciclo finché il cliente non si dichiara soddisfatto.

Modello a cascata (Waterfall)

Il modello a cascata è uno dei più classici e storici modelli di sviluppo del software. È stato uno dei primi ad essere formalizzato e si basa su un approccio sequenzale e lineare: ogni fase del progetto deve essere completata prima di passare alla successiva. Dopo ogni fase c'è la verifica, si tratta di controllare che il processo di sviluppo sia stato eseguito correttamente e che gli artefatti prodotti siano conformi alle specifiche. Oltre alla fase di verification c'è la fase di validation nella quale si verifica che il prodotto soddisfi effettivamente i bisogni dell'utente finale.

Modello a prototipi

IL modello di Rapid Prototyping è un approccio allo sviluppo del software che mette al centro la costruzione veloce di prototipi di sistema, con lo scopo di comprendere meglio i requisiti, validare le funzionalità con l'utente e ridurre il rischio di incomprensioni.

Un prototipo è una versione semplificata e spesso incompleta del sistema finale. Può essere:

- Interattivo ma non funzionante (solo interfaccia grafica)
- Funzionante in parte (solo alcune funzionalità)
- Creato con tecnologie diverse da quelle definitive.

Vantaggi	Svantaggi
Aiuta a chiarire meglio i requisiti, soprattutto quando inizialmente sono vaghi o incompleti.	Rischio di confondere il prototipo con il prodotto

Vantaggi	Svantaggi
Favorisce feedback e correzioni rapide, infatti il cliente può provare il prototipo e dare indicazioni prima dello sviluppo finale	Costi e tempi possono aumentare se gestito male
Coinvolge attivamente il cliente	Codice del prototipo spesso poco riutilizzabile
Migliora usabilità e soddisfazione finale	Non adatto a progetti critici o altamente regolati

Modello iterativo o incrementale

Il modello incrementale è un approccio allo sviluppo del software che prevede la realizzazione del sistema per gradi, cioè per incrementi (build) successivi. Ogni incremento aggiunge nuove funzionalità al sistema, partendo da una base semplice che cresce man man fino a diventare il prodotto completo.

A differenza del modello a cascata, dove si sviluppa tutto in un'unica volta, il modello incrementale prevede un ciclo ripetuto di:

1. Analisi dei requisiti (parziali)
2. Progettazione
3. Implementazione
4. Test
5. Consegna di un incremento (build) funzionante

Dopo il rilascio di una build, si raccolgono feedback e si passa allo sviluppo del successivo. È versatile, adattabile e orientato al feedback (simile a rapid prototyping).

Modello a spirale

Il modello a spirale è un modello pensato per integrare i punti di forza del modello a cascata e dello sviluppo iterativo, mettendo però al centro un elemento spesso

trascurato: la gestione del rischio.

Ogni ciclo della spirale è composto da 4 fasi fondamentali:

1. **Pianificazione**: si definiscono obiettivi, funzionalità da sviluppare, risorse necessarie.
2. **Analisi dei rischi**: si individuano i rischi tecnici o gestionali, si valutano alternative e si pianificano soluzioni.
3. **Sviluppo e verifica**: si implementa e si testa un prototipo o una parte del sistema.
4. **Revisione e feedback**: si valuta quanto prodotto, si ottiene feedback dal cliente e si decide se proseguire con un nuovo giro.

Ogni ciclo porta a una versione più raffinata de software, fino a giungere alla versione finale.

Risk Analysis

L'analisi dei rischi è un processo fondamentale per identificare, valutare e gestire i possibili problemi o incertezze che potrebbero influenzare negativamente il successo di un progetto. In questo contesto, l'analisi dei rischi aiuta prevenire problemi futuri, ottimizzare i costi e garantire che il progetto rimanga nei limiti previsti di tempo e qualità.

I passaggi fondamentli sono:

1. **Identificazione dei rischi**: Si riflette su tutte le possibili fonti di rischio: errori tecnici, mancanza di competenze, costi imprevisti, scandenze troppo strette, strumenti inadeguati, problemi organizzativi o esterni.
2. **Valutazione**: Per ciascun rischio si analizzano la probabilità che si verifichi (alta, media, bassa) e l'impatto che avrebbe sul progetto (grave, moderato, trascurabile).
3. **Pianificazione**: Si stabiliscono azioni e contromisure per gestire i rischi più rilevanti.
4. **Monitoraggio**: Durante il progetto, i rischi vengono tenuti sotto controllo e aggiornati: alcuni possono scomparire, altri nuovo possono emergere.

Senza un’analisi del rischio, un progetto può fallire improvvisamente a causa di

problemi prevedibili. Con una buona risk analysis, invece, il team è preparato: anche se il rischio si manifesta, sa cosa fare e come reagire, riducendo danni, costi e ritardi.

Modello Microsoft

La Microsoft ha dovuto affrontare problemi di:

- **Incremento della qualità** dei prodotti software;
- **Riduzione di tempi e costi** di sviluppo.

Per risolvere questi problemi si è adottato un processo che allo stesso tempo è **iterativo**, **incrementale**, **concorrente**, e che permette di esaltare la creatività delle persone coinvolte nello sviluppo software. L'approccio usato attualmente dalla Microsoft è noto come **syncronize-and-stabilize**. Con questo modello otteniamo un ciclo di sviluppo a 3 fasi:

1. **Planning**: definisce la visione del prodotto, le specifiche e la pianificazione.
2. **Development**: sviluppo di funzionalità in 3-4 sottoprogetti sequenziali, ognuno dei quali si traduce in un rilascio "milestone".
3. **Stabilizzazione**: test interni ed estero completi, prodotto finale e spedizione.

Le **caratteristiche principali** sono:

- Sviluppo software e testing eseguiti in parallelo.
- Features prioritizzate e integrate in 3-4 sottoprogetti cardine.
- Continuo feedback dei customer durante il processo di sviluppo.

Modello Agile

Il modello Agile non è un singolo modello di sviluppo, ma un insieme di principi e valori che guidano un modo di lavorare flessibile, collaborativo e iterativo. È nato come risposta ai limiti dei modelli tradizionali, che spesso si rivelano troppo rigidi in contesti dove le esigenze cambiano rapidamente o i requisiti non sono completamente chiari fin dall'inizio.

Alla base del modello Agile c'è il manifesto Agile che valorizza:

- Le persone e le interazioni più dei processi e degli strumenti.
- Il software funzionante più della documentazione esaustiva.
- La collaborazione col cliente più della negoziazione del contratto.
- La risposta al cambiamento più della pianificazione rigida.

Scrum

Scrum è uno dei framework più popolari all'intenro dell'approccio Agile. È una struttura organizzativa leggera e iterativa, pensata per gestire progetti complessi in modo flessibile, trasparente e collaborativo. Si basa sull'idea di sviluppare software a piccoli passi (`sprint`) e di apprendere costantemente da ciò che si è fatto per migliorare nel tempo.

Un progetto Scrum è suddiviso in **Sprint**, ovvero cicli di sviluppo a durata fissa (1-4 settimane), alla fine dei quali si rilascia un prodotto funzionante e potenzialmente consegnabile. Durante ogni Sprint, si eseguono le seguenti attività:

1. **Sprint Planning**: si pianifica il lavoro da svolgere nello sprint.
2. **Daily Scrum**: breve riunione giornaliera per allineare i team.
3. **Sprint Review**: si mostra il lavoro svolto al cliente.
4. **Sprint Retroprospective**: il team riflette su come migliorare il proprio processo.

Scrum definisce 3 ruoli principali, ognuno con responsabilità precise:

- **Product Owner**: definisce le priorità e mantiene aggiornato l'elenco ordinato delle funzionalità da implementare.
- **Scrum Master**: protegge il team da interferenze, rimuove ostacoli e fa rispettare le regole Scrum.
- **Dev Team**: gruppo di sviluppatori che lavora agli incrementi.

I principali artifatti Scrum sono:

- **Product Backlog**: lista prioritaria delle funzionalità richieste.
- **Sprint Backlog**: lista delle attività selezionate per lo sprint corrente.
- **Incremento**: il prodotto parzialmente completo e funzionante al termine dello sprint.

Software Requiments: funzionali, non funzionali e di dominio

I requisiti software sono la descrizione dei servizi che un sistema software deve fornire, insieme ai vincoli da rispettare sia in fase di sviluppo che durante la fase di opertività del software. Esistono due categorie di requisiti SW:

- **Requisiti utente**: Descrizione in liguaggio naturale dei servizi che il sistema deve fornire e dei vincoli operativi.
- **Requisiti di sistema**: Specificati mediante la stesura di un documento strutturato che descrive in modo dettagliato i servizi chhe il sistema SW deve fornire. Ci sono 3 categorie:

Requisiti Funzionali

I requisiti funzionali descrivono le funzionalità specifiche che il sistema deve offrire. Essi definiscono le azioni che il sistema deve compiere in risposta a determinati input o situazioni.

Requisiti non Funzionali

I requisiti non funzionali specificano le qualità e le caratteristiche che il sistema deve possedere, senza riferirsi a funzionalità specifiche. Essi influenzano l'esperienza dell'utente e le prestazioni del sistema. Per esempio:

- **Prestazioni**: Tempo di risposta, throuthput, utilizzo delle risorse.
- **Sicurezza**: Protezione dei dati, autenticazione, autorizzazione.
- **Usabilità**: Facilità d'uso, accessibilità, design dell'interfaccia cliente.

Requisiti di Dominio

Un requisito di dominio descrive un comportamento, un vincolo o una regola che non è universale per tutti i software, ma che vale solo in un certo contesto di applicazione (dominio). Spesso questi requisiti non sono espressi direttamente dall'utente, perché sono dati per scontati da chi opera nel settore, ma sono comunque essenziali affinché il sistema sia accettabile, corretto e conforme alle

aspettative.

Ignorare un requisito di dominio può rendere un sistema non conforme alla legge, inutilizzabile, o scartato dagli utenti esperti del settore. Per questo, durante l'analisi dei requisiti, è fondamentale conivolgere esperti del dominio, come avvocati, medici, funzionari pubblici etc... .

Tipo di requisito	Focus
Funzionali	Cosa il sistema fa (comportamento)
Non funzionali	Come il sistema si comporta (prestazioni, sicurezza...)
Di dominio	**Requisiti specifici del settore**, spesso imposti da regole esterne

OOA (Object Oriented Analysis) e OOD (Object Oriented Analisys)

- La fase di OOA definisce, secondo un approccio ad oggetti, **COSA** un prodotto software deve fare.
- La fase di OOD definisce, secondo un approccio ad oggetti, **COME** un prodotto software deve fare quanto specificato in fase di OOA.

Devono fornire, ciacuno dal proprio punto di vista, una rappresentazione corretta, completa e consistente:

- Struttura dei dati - **modello di dati**
- Aspetti funzionali del sistema - **modello comportamentale**
- Come le funzioni modificano i dati - **modello dinamico**

Notazione UML

Ciascun metodo di OOA (OOD) fa uso di una propria notazione per la rappresentazione de modelli di sistema. La notazione scelta è il linguaggio UML (Unified Modeling Language), è uno stardard per la descrizione di sistemi software.

Si compone di 9 formalismi di base e di un insieme di estensioni. UML è un linguaggio di descrzione, non è un metodo nè definisce un processo.

1. **Use Case Diagram**: Evidenziano la modalità con cui gli utenti utilizzano il sistema. Possono essere usati come supporto per la definizione dei requisiti utente.
2. **Class Diagram**: Consentono di rappresentare le classi con le relative proprità e le associazioni che le legano.
3. **State Diagram**: Rappresentano il comportamento dinamico dei singoli oggetti di una classe in termini di stati possibili e transizioni di stato per effetto di eventi.
4. **Activity Diagram**: Sono particolari state diagram, in cui gli stati rappresentati sono azioni in corso di esecuzione.
5. **Sequence Diagram**: Evidenziano le interazioni che oggetti di classi diverse si scambiano nell'ambito di un determinato caso d'uso, ordinate in sequenza temporale.
6. **Collaboration Digram**: Descrive le interazioni tra oggetti diversi, evidenziando le relazioni esistenti tra le singole istanze.
7. **Object Diagram**: Permettono di rappresentare gli oggetti e le relazioni tra essi nell'ambito di un determinato caso d'uso.
8. **Component Diagram**: Evidenziano la strutturazione e le dipendenze esisitenti tra componenti software.
9. **Deployment Diagram**: Evidenziano le configurazioni dei nodi elaborativi di un sistema real-time e i componeti, processi ed oggetti assegnati a tali nodi.

OOA - Modello dei dati

Rappresenta da un punto di vista statico e strutturale l'organizzazione logica dei dati da elaborare. Il modello dei dati viene specificato mediante il formalismo dei `class diagram` permettendo di definire classi, attributi, operazioni e associzioni tra classi. In questa fase si concentra sulla definizione delle cosidette `entity classes` ovvero quelle classi che definiscono il dominio applicativo e che sono rilevanti per il sistema. In seguito vengono introdotte le `control classes` che gestiscono la logica del sistema e le `boundary classes` che rappresentano l'interfaccia utente.

- **Alcune regole**:

- Ciascuna classe deve prevedere un insieme di istanze (oggetti), le cosidette singleton classes (classi per le quali si ha un' unica istanza) non sono di norma classificabili come entity classes.
- Ciascuna classe deve avere un nome significativo specifino del dominio applicativo. Adottare una convenzione standard (Pascal Case, Snake Case, Cammel Case).
- **Associazioni**:
- Gli attributi che non sono primitivi, quindi che si riferiscono ad un altra classe devono essere un associazione.
- Ogni associazione ternaria dovrebbe essere rimpiazzata con un ciclo di associazioni binarie.
- Assegnare nomi alle associazioni usando la stessa convenzione per gli attributi.
- Assegnare nomi di ruole alle estemità.
- Determinare la molteplicità delle associazioni.
- **Aggregazione**:

Rappresenta una relazione di tipo "whole-part" (contenimento) tra una classe composta e l'insieme di una o più classi componenti, senza possederlo completamente. La "parte" può esistere indipendentemente dal "tutto".

- L'ogetto contenuto è solo referenziato, non controllato o creato dal contenitore.
- Diamante vuoto in UML.
- Corrisponde a "has" e "member".

Una classe "Università" può avere più "Studenti". Gli studenti non appartengono esclusivamente a quell’università nel contesto del modello (potrebbero cambiare università o esistere indipendentemente da essa).

- **Composizione**:

È una relazione forte di "whole-part", in cui l' oggetto contenitore possiede completamente l'oggetto contenuto. La "parte" non può esistere senza il "tutto".

- L'ogetto contenuto è creato, gestito e distrutto assieme al contenitore.
- Diamante pieno in UML.
- Corrisponde a "owns" e "exclusive-owns".

Una classe "Casa" può avere oggetti "Stanza". Le stanze non hanno senso di

esistere al di fuori della casa. Se la casa viene distrutta, anche le stanze spariscono.

- **Ereditarietà**

Usata per rappresentare la condivisione di attributi ed operazioni tra classi. Le caratteristiche comuni sono modellate in una
classe più generica (`superclasse`), che viene specializzata nell'insieme di `sottoclassi` . Una sottoclasse eredita attributi ed operazioni della superclasse. Viene rappresentata in UML con una linea, che collega la sottoclasse con la superclasse, avente una freccia diretta verso la superclasse.

OOA - Modello Comportamentale

Rappresenta gli aspetti funzionali del sistema da un punto di vista operativo, evidenziando come gli oggetti collaborano ed interagiscono tra di loro. Fa uso di vari formalismi
(`use case digram, activity diagram, sequence diagram, collaboration diagram`).

Use case

Un Use Case rappresenta un'interazione tra un attore (utente o sistema esterno) e il sistema, relativa a una funzionalità specifica. Durante la fase di OOA si concentra su COSA il sistema deve fare (vari scenari di funzionamento). Ci sono 4 tipi di relazioni:

- **Associazione**: tra attoree caso d'uso
- **Include**: rappresenta una relazione obbligatoria tra due use case: uno include sempre l'altro durante la sua esecuzione. Si usa quando un comportamento è comune a più use case. Quando si vuole evitare duplicazioni. Quando si vuole modularizzare un comportamento ripetuto.
  Per esempio gli use case "Registra Utente" e "Recupera Password", entrambi includono "Inviare Email".
  Graficamente si rappresenta mediante una linea trateggiata con freccia vero il caso d'uso incluso.
- **Exclude**: rappresenta una relazione opzionale, cioè un caso d'uso può estendere un altro caso solo se una certa condizione è soddisfatta. Si usa

quando un comportamento è facoltativo. Quando si vuole aggiungere funzionalità opzionali senza complicare il caso d'uso principale.
Per esempio il caso d'uso generale "Inserire Risultato" -> `extend` "Segnalare Irregolarità", ovvero in alcuni casi particolari, durante l'inserimento del risultato, l'arbitro può decidere di segnalare un'irregolarità.
Graficamente si rappresenta mediante una linea trateggiata con freccia verso il caso d'uso principale.

- **Generalizzazione**

Activity Diagram

Rappresenta a vari livelli di astrazione il flusso di esecuzione, sia sequenziale che concorrente, in una applicazione object-oriented. In fase di OOA, viene usato per rappresentare il flusso delle attività nella esecuzione di un caso d’uso.

Si parte da un nodo iniziale (cerchio nero pieno) che identifica la partenza del flusso. Ogni attività è una fase del processo e viene rappresentata da un rettangolo. Le frecce che collegano le attività rappresentano l'ordine con cui avvengono. I nodi decisionali (rombi) permetteon di dirigere il flusso in base a condizioni. Infine, per idenficare la fine del flusso abbiamo un nodo finale (cercgio con un cerchio nero dentro).

Sequence e Collaboration Diagram

- I sequence descrivono lo scambio di messaggi tra oggeti in ordine temporale, è usato principalmente in fase di OOA
- I collaboration descrivono lo scambio di messaggi tra oggetti mediante relazioni tra gli oggetti stessi, è usato in fase di OOD.

Le attività dell'activity diagram vengono mappate come messaggi in un sequence diagram. Un messaggio può rappresentare:

- **Signal**: denota una chiamata di tipo asincrono, l'oggetto mittente continua l'esecuzione dopo aver inviato il messaggio asincrono.
- **Call**: Denota una chiamata ti tipo sincrono, l'oggetto mittente blocca la sua esecuzione in attesa di una risposta.

Durante questa fase, si definiscono l'insieme di operazioni che la classe mette a

disposizione, definendo l'interfaccia pubblica di classe, determinando per ogni operazione la sua `signature`, ovvero il nome, la lista degli argomenti formali e il tipo di ritorno. Durante la fase di OOD si definisce l' algoritmo.

Ogni messaggio inviato ad un oggetto identifica un metodo della classe a cui appartiene tale oggetto. Si possono usare criteri aggiuntivi, per esempio `CRUD`.

OOA - Modello Dinamico

Rappresenta il comportamento dinamico degli oggetti di una singola classe, in termini di stati possibili ed eventi e condizioni che originano transizioni di stato. Fa uso del formalismo `state diagram`. Viene costruito per ogni classe di controllo ed è usato principalmente per le applicazioni scientifiche real-time.

L'approccio **BCE (Boundary - Cotrol - Entity)** è una tecnica molto usata nell'analisi orientata agli oggetti, in particolare nella fase di OOA per organizzare logicamente le classi e le responsabilità in un sistema.

- **Entità**: Sono le classi che rappresentano i dati e il comportamento persistente. Corrispondono agli elementi del dominioop applicativo come `Utente`, `Ordine`, `Prodotto`.
- **Boundary**: Sono le classi che gestiscono l'interazione tra il sistema e gli attori esterni, come utenti o sistemi terzi. Si occupano dell'input e dell'output (UI, API etc...).
- **Control**: Sono le classi che coordinano il comportamento del sistema, gestendo il flusso di eventi fra boundary e entity. Contengono la logica applicativa o di business.

L’approccio BCE ti guida nel progettare un sistema che sia pulito, modulare e comprensibile, distinguendo chiaramente tra chi fornisce i dati (boundary), chi li elabora (control) e chi li rappresenta in modo stabile (entity).

OOD Software Architectures

OOD (Object-Oriented Design) è una fase nello sviluppo del software che si concentra su come realizzare il software, trasformando i concetti analizzati in un'architettura software concreta.

Un architettura software è un insieme di decisione strutturali e di linee guida che definiscono la forma generale di un sistema software: come è suddiviso, come comunicano le sue parti, e quali responsabilità ha ciascuna componente.

OOD Architetture distribuite (Distributed Software Systems)

Un sistema software distribuito è un sistema in cui componenti software localizzati su computer diversi comunicano e cooperano tra loro attraverso una rete per raggiungere un obiettivo comune.

Nei sistemi distribuiti, l'integrazione tra componenti eterogenei è complessa, infatti in mezzo troviamo un software chiamata `middleware`. È uno strato intermedio tra il sistema operativo e le applicazioni software e serve a facilitare la comunicazione, la gestione dei dati, la sicureza tra componenti software diversi, spesso distribuiti su più macchine o piattaforme.

Vantaggi	Svantaggi / Sfide
Scalabilità molto elevata	Complessità nella progettazione
Resilienza: tolleranza ai guasti	Problemi di sincronizzazione e coerenza
Prestazioni ottimizzabili su larga scala	Debugging più difficile
Flessibilità e modularità	Sicurezza più difficile da gestire
Adatto al cloud e ai servizi moderni	Latency e problemi di rete da considerare

1. Client/Server Architectures

È un modello in cui due entità principali – `client` e `server` – si dividono i compiti in modo cooperativo ma asimmetrico.
In questo modello:

- Il client è l’entità che richiede un servizio. Prevede un interfaccia utente per prelevare le richieste e le invia al server mediante software middleware. Infine

visualizza le risposte del server mediante l'interfaccia utente.
- Il server è l’entità che offre il servizio richiesto. Risponde alle richieste del client e nasconde tutta l'architettura C/S che c'è dietro.

L'architettura Two-Tier C/S è suddivisa in due livelli principali:

- **Client**: l'interfaccia utente, che gestisce la logica di presentazione e talvolta parte della logica applicativa.
- **Server**: il gestore dei dati, solitamente un DBMS, che conserva e restituisce le informazioni.

In questo tipo di architettura abbiamo due tipologie di client:

1. **Thin-Client**: Tutto il processo applicativo e la gestione dei dati avviene nel server. Il client si occupa solo della presentazione.
2. **Fat-Client**: Il client è responsabile della logica applicativa e della presentazione. Il server si occupa solo della gestione dei dati (database).

L'architettura a Three-Tier C/S migliora la scalabilità, la manutenibilità e la riusabilità del sistema rispetto ai modelli a due livelli. È Strutturato su 3 livelli:

- Presentation Tier, ovvero il client
- Application Tier, contiene la logica di business (server)
- Data Tier, contiene i database e i sistemi di persistenze (server)

Livello	Funzione principale	Esempi
Presentazione	Interfaccia utente	Browser web, app mobile
Logica	Esecuzione della logica di business	Server Java, .NET, Node.js
Dati	Gestione della persistenza e delle query	MySQL, PostgreSQL, MongoDB

2. Distributed Objects Architectures

L'architettura ad oggetti distribuiti è un'architettura di sistema che non fa

distinzioni tra client/server, infatti ogni oggetto distribuito può fungere sia da client che da server. La comunicazione remota fra gli oggetti è resa trasparente usando middlewar basati sul concetto di software bus. Le applicazioni basate sul questa architettura consistono in un insieme di oggetti che sono eseguiti su piattaforme distribuite ed eterogenee, e comunicano tramite invocazioni remote
dei metodi.

3. Component-Based Architectures

È un approccio in cui l'intero sistema viene costruito assemblando componenti riutilizzabili, ciascuno dei quali incapsula una parte ben definita della funzionalità. Un componente software è un entità software che incapsula struttura e comportamento, ha interfacce ben definite e può essere configurata e riutilizzata in diversi contesti applicativi.

Oggetti	Componenti
Sono **entità a run-time**, con identità, stato e comportamento.	Sono **astrazioni statiche** usate nella fase di costruzione del sistema.
Incapsulano **servizi**.	Incapsulano **soluzioni strutturali e comportamentali** riutilizzabili.
Granularità più fine (classe, oggetto)	Possono avere granularità più fine o più grossa (moduli, mixin, template).

In breve: gli oggetti vivono quando il programma è in esecuzione, mentre le componenti sono usate durante la costruzione del programma.

L’approccio component-based permette di costruire sistemi complessi in modo efficiente, affidandosi a componenti predefinite, modulari e configurabili. La chiave è pensare in termini di assemblaggio, più che di scrittura da zero.

4. Service Oriented Architecture (SOA)

La SOA è un'architettura software distribuita che consiste in molteplici servizi. I servizi sono distribuiti in modo tale da poter essere eseguiti su nodi differenti con differenti service provider. L'obiettivo di SOA è quello di sviluppare applicazioni SW che sono composte da servizi distribuiti, in modo tale che i singoli servizi possano

essere eseguiti su più piattaforme differenti e implementati con differenti linguaggi di programmazione.

Un servizio SOA è una funzionalità software autonoma, ben definita, accessibile via rete che:

- puo essere invocata da altri componenti,
- ha un'interfaccia pubblica,
- nasconde i dettagli interni di implementazione,
- è riutilizzabile in vari contesti applicativi.

Ci sono diversi pattern architetturali tra cui:

1. Broker Patterns
   In un ambiente SOA, i servizi possono essere registrati e scoperti dinamicamente. Il pattern Service Broker si occupa di gestire la comunicazione tra client e servizi in modo disaccopiato e flessibile. In pratica fa da intermediario, smistando richieste e risposte senza che il client debba conoscere i dettagli di dove e come il servizio è effettivamente eseguito.
   Ogni servizio, prima di diventare utilizzabile, si registra presso il broker.
   Quando un client ha bisogno di un servizio, invece di cercalo in modo diretto, si rivolge al broker, che gli fornisce i dettagli necessari per stabilire la connessione.
   Un punto cruciale è la trasparenza che il broker garantisce, cioè il client non deve sapere dove il servizio è fisicamente localizzato (location transparency) né su quale piattaforma tecnologica gira (platform transparency).
   La comunicazione mediata dal broker (brokered communication) permette di ridurre la complessità e aumentare la flessibilità. Un requisito fondamentale è che ogni servizio si registri presso il broker (come previsto dal pattern Service Registration). Solo così il broker può fare da "centralino" per la scoperta e la gestione dei servizi.
   Uno dei più usati broker è `CORBA`.
   Anche se le SOA sono concettualmente platform-indipendent, attualmente vengono fornite con grande successo su piattaforme tecnologiche di **Web Services**. Da un punto di vista di SW, i Web Services sono le API che forniscono i metodi standard di comunicazione. Da un punto di vista business, i Web Services sono funzionalità di business fornite da una compagnia nella

forma di servizio esplicito di Internet.
I Web Services sono quindi il fulcro dell'implmentazione SOA, e sfruttano vari protocolli per far comunicare i servizi fra di loro. Tra questi troviamo:

- **SOAP (Simple Object Access Protocol)**: protocollo basato su linguaggio XML che permette lo scambio di informazioni in un sistema distribuito. Utilizza protocolli di trasporto come HTTP, SMTP e altri.
- **REST (Representational State Transfer)**: architettura che utilizza i metodi HTTP per performare operazioni CRUD.

Oltre ai Web Services, le architetture SOA utilizzano altre tecnologie, tra cui:

- **UDDI (Universal Description Discovery and Integration)**: registro per pubblicare e trovare servizi web all'interno di un architettura SOA. Aiuta a localizzare e identificare i servizi disponibili su una rete.
- **WSDL (Web Services Description Language)**: Linguaggio basato su XML per descrivere le interfacce dei servizi in un architettura SOA. Specifica cosa fa il servizio, come può essere invocato e dove si trova.

2. Trascaction Patterns
   Una transazione è una richiesta composta da due o più operazioni che insieme realizzano una singola funzione logica, stiamo parlando di un’unità di lavoro che deve essere trattata come indivisibile: o viene completata tutta, oppure non viene completata affatto. Questo principio è fondamentale per garantire la coerenza e l’affidabilità del sistema.
   Le transazioni devono rispettare le proprietà ACID:

Proprietà	Descrizione	Perché è importante
Atomicità	La transazione è un’unità indivisibile: o si completa tutta o non si esegue nulla.	Evita stati parziali e incoerenti
Coerenza	La transazione mantiene le regole di integrità e porta il sistema da uno stato valido a un altro valido.	Garantisce dati affidabili e corretti
Isolamento	Le transazioni concorrenti non si interferiscono tra loro, come se fossero eseguite in sequenza.	Previene effetti collaterali dovuti a esecuzioni

Proprietà	Descrizione	Perché è importante
		parallele
Durabilità	Gli effetti di una transazione completata sono permanenti, anche in caso di crash o guasti.	Assicura che i dati non vadano persi

- Il **Two-Phase Commit Protocol** è un meccanismo che serve a garantire l'atomicità delle transazioni distribuite, cioè a sincronizzare correttamente le modifiche su nodi diversi.
- Il **Compound Transaction** pattern entra in gioco quando una richiesta complessa da parte del client può essere suddivisa in più transazioni atomiche più semplici, ciascuna indipendente e gestibile singolarmente.
- Il **Long-Living Transaction** pattern è pensato per gestire transazioni che durano molto a lungo, spesso perché richiedono interventi o decisioni umane tra le varie fasi.

3. Negotiation Patters
   In questo pattern, un client agent agisce per conto dell'utente e fa una proposta a un service agent, che rappresenta il servizio. Il service agent valuta la proposta, eventualmente interagendo con altri servizi, e risponde con una o più opzioni che si avvicinano alla richiesta iniziale.
   Il client agent può accettare una delle opzioni, proporre modifiche o rifiutare l'offerta. Se il service agent può soddisfare la richiesta finale, l'accetta; altrimenti, la rifiuta.
   In pratica, il pattern consente uno scambio dinamico e flessibile di proposte e controproposte tra client e servizio, permettendo negoziazioni sia per servizi negoziabili (con possibilità di controfferte) sia non negoziabili (richieste ferme). Questo approccio è utile quando i requisiti del client non sono rigidi e serve un processo interattivo per trovare un accordo soddisfacente.

Design Patterns

I design pattern sono soluzioni generiche, riutilizzabili e collaudate a problemi comuni che si incontrano nello sviluppo del software orientato agli oggetti. Non

sono blocchi di codice pronti da incollare, ma piuttosto schemi concettuali, strutture astratte che aiutano a organizzare il codice in modo chiaro, flessibile e manutenibile.

I design pattern sono importanti perchè risolvono problemi che si incontrano spesso nello sviluppo di software. I design pattern offrono standard per risolvere tutte queste problematiche.

Si classificano secondo due criteri, uno riguardo lo scopo (purpose) mentre l'altro riguardo il raggio di azione (scope).

- **Purpose**
  i. **Creazionali**: gestiscono l'istanziazione degli oggetti. Offrono meccanismi per creare oggetti in modo flessibile e indipendente dal contesto concreto.
  ii. **Strutturali**: si concentrano sul modo in cui le classi e gli oggetti si compongono.
  iii. **Comportamentali**: descrivono come gli oggetti interagiscono e comunicano tra loro, mettendo l'accento sul flusso di controllo e la responsabilità.
- **Scope**
  i. **Classi**: Definiscono le relazioni fra classi e sottoclassi. Le relazioni sono basate sul concetto di ereditarietà e sono quindi statiche (compile time).
  ii. **Oggetti**: Definiscono relazioni tra oggetti, che possono cambiare durante l'esecuzione e sono quindi dinamiche.

1. Abstract Factory

Fa parte della classe di DP `creazionali` ed ha lo scopo di fornire un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete. Si definisce un'interfaccia comune che permette di creare oggetti appartenenti alla stessa famiglia, poi si implementano diverse `fabbriche concrete` per ogni variante.

```
interface Button {
    void paint();
}

class WindowsButton implements Button {
    public void paint() {
        System.out.println("Rendering a Windows button");
    }
}

class MacButton implements Button {
    public void paint() {
        System.out.println("Rendering a Mac button");
    }
}

interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}

class WindowsFactory implements GUIFactory {
    public Button createButton() {
        return new WindowsButton();
    }
}

class MacFactory implements GUIFactory {
    public Button createButton() {
        return new MacButton();
    }
}

public class Application {
    private Button button;
    private Checkbox checkbox;

    public Application(GUIFactory factory) {
        button = factory.createButton();
```

```
        checkbox = factory.createCheckbox();
    }

    public void render() {
        button.paint();
        checkbox.render();
    }
}
```

In questo modo, il client può creare una GUI per Windows o Mac semplicemente cambiando l’istanza della factory passata al costruttore, senza toccare il resto del codice.

Categoria	Contenuto
Applicabilità	- Quando il sistema deve essere **indipendente** dalla modalità di creazione degli oggetti - Quando si devono gestire **famiglie di oggetti correlati** (es. GUI per Windows/Mac) - Quando il **client non deve essere legato** a una classe concreta o famiglia specifica
Partecipanti	- `AbstractFactory`: interfaccia per creare prodotti astratti - `ConcreteFactory`: implementazioni che creano prodotti concreti - `AbstractProduct`: interfaccia dei prodotti da creare - `ConcreteProduct`: implementazioni specifiche dei prodotti - `Client`: utilizza la factory e lavora solo con le interfacce
Conseguenze	- Le **classi concrete sono isolate** e facilmente gestibili - Cambiare famiglia di prodotti è facile: basta sostituire la factory - Aggiungere una nuova **famiglia** richiede nuove classi e **ricompilazione** - Il codice è **più modulare** e flessibile, ma può diventare più complesso se usato in contesti semplici

2. Factory Method

L’idea di fondo è che la creazione di un oggetto non avviene direttamente nel codice, ma viene delegata a un metodo specializzato, chiamato appunto `factory method`. Questo metodo può essere ridefinito dalle sottoclassi per istanziare oggetti specifici. Nel codice orientato agli oggetti, spesso si vuole evitare di usare direttamente `new` per creare un’istanza di una classe concreta, perché così si rende il codice rigido e poco estensibile. Il Factory Method serve proprio a risolvere questo problema: consente alle sottoclassi di decidere quale classe concreta instanziare, mantenendo il codice cliente indipendente.

Il Factory Method è utile quando un codice deve lavorare con un’interfaccia o una superclasse, ma non deve sapere quale sottoclasse concreta utilizzare. La responsabilità della creazione viene così demandata a chi conosce i dettagli.

Vantaggi	Svantaggi
Il codice è **indipendente dalle classi concrete** dei prodotti da creare	Aggiunge **complessità strutturale** (serve creare nuove sottoclassi per ogni tipo di prodotto)
Favorisce il **polimorfismo** e l’estensibilità del codice	Può generare una **gerarchia di classi più ampia** e difficile da gestire
È **facile aggiungere nuovi tipi di prodotto**: basta una nuova sottoclasse con override	Ogni modifica alla logica di creazione richiede la modifica o l’estensione delle classi
Incapsula la logica di istanziazione e semplifica i test	Può risultare **ridondante** in casi semplici dove basta una `new`

- Il Factory Method crea un solo oggetto per volta e si basa sul override del metodo da parte delle sottoclassi.
- L’Abstract Factory fornisce un’intera famiglia di oggetti e lavora con più prodotti tra loro correlati.

3. Adapter

Fa parte dei design pattern strutturali, serve a far collaborare classi con interfacce incompatibili tra loro. È come un "traduttore" che si mette in mezzo tra due componenti per farle comunicare, anche se originariamente non erano pensate per lavorare insieme. È strutturato mediante 3 componenti principali:

- **Target** è l'interfaccia che il codice client si aspetta.
- **Adaptee** è la classe esistente che ha l'interfaccia incompatibile.
- **Adapter** è la classe che implementa l'interfaccia `Target`, ma al suo interno usa un oggetto `Adaptee`, adattando i metodi.

```
// Interfaccia attesa dal client
interface MediaPlayer {
    void play(String filename);
}

// Classe esistente incompatibile
class AdvancedPlayer {
    void playFile(String filePath) {
        System.out.println("Riproduzione: " + filePath);
    }
}

// Adapter
class MediaAdapter implements MediaPlayer {
    private AdvancedPlayer advancedPlayer;

    public MediaAdapter(AdvancedPlayer player) {
        this.advancedPlayer = player;
    }

    public void play(String filename) {
        // adatta il metodo play al metodo playFile
        advancedPlayer.playFile(filename);
    }
}
```

Categoria	Contenuto
Quando usarlo	- Quando hai classi esistenti (o librerie esterne) con **interfacce incompatibili** rispetto a quelle del tuo sistema - Quando vuoi **riutilizzare codice legacy** senza modificarlo - Quando il tuo sistema ha un'interfaccia standard e devi **integrare componenti esterni**
Vantaggi	- **Riutilizzo del codice esistente** senza modificarlo - Permette l’integrazione con librerie o sistemi legacy - Favorisce la **separazione delle responsabilità**: il codice client rimane pulito - Può essere usato per adattare classi con interfacce complesse o sbilanciate
Svantaggi	- Aggiunge un **livello di astrazione** in più - Può portare a **una proliferazione di classi adapter**, se ne servono molti - Rischio di **mascherare incompatibilità semantiche** (metodi che sembrano simili ma fanno cose diverse) - Può diventare un punto fragile se il codice da adattare cambia spesso

4. Composite

Il Composite è un design pattern strutturale molto elegante, che permette di rappresentare strutture ad albero, in cui oggetti singoli e gruppi di oggetti possono essere trattati in modo uniforme. È particolarmente utile quando si vuole costruire una gerarchia dove ogni nodo può essere una “foglia” (cioè un oggetto semplice) oppure un “composito” (cioè un contenitore di altri oggetti).

Prendiamo per esempio un file system: ci sono file (elementi semplici) e cartelle (che possono contenere altri file o cartelle). Vogliamo poter dire a un file "dammi la dimensione" e ottenere, per esempio, 100KB. Ma vogliamo poter dire la stessa cosa a una cartella, e ricevere la somma delle dimensioni di tutto ciò che contiene, in maniera trasparente.
Ecco dove entra in gioco il Composite: unifica il trattamento di oggetti singoli e

composti.

È strutturato mediante 3 principali componenti:

1. **Component**: Un'interfaccia o classe astratta che definisce le operazioni comuni.
2. **Leaf**: Una classe concreta che rappresenta un oggetto "atomico" e implementa direttamente le operazioni.
3. **Composite**: Una classe che rappresenta un contenitore, che contiene una lista di altri `Component` (figli o composite). Implementa le operazioni in modo ricorsivo.

```
interface Graphic {
    void draw();
}

// Foglia
class Circle implements Graphic {
    public void draw() {
        System.out.println("Disegno un cerchio");
    }
}

// Composito
class CompositeGraphic implements Graphic {
    private List<Graphic> children = new ArrayList<>();

    public void add(Graphic g) {
        children.add(g);
    }

    public void draw() {
        for (Graphic g : children) {
            g.draw(); // chiamata ricorsiva
        }
    }
}
```

Categoria	Contenuto
Quando usarlo	- Quando vuoi rappresentare **strutture gerarchiche ad albero** (es. file system, menu, componenti grafici) - Quando hai bisogno di **trattare allo stesso modo oggetti singoli e contenitori** - Quando le operazioni devono essere **applicate ricorsivamente** a una struttura composta
Vantaggi	- Permette un **trattamento uniforme** di oggetti e composizioni - Facilita l'uso della **ricorsione** su strutture complesse - Il codice client è **semplificato**: lavora sempre con l'interfaccia `Component` - Migliora la **flessibilità** e la **componibilità** degli oggetti
Svantaggi	- Può essere difficile **limitare le operazioni solo a certi tipi** (es. operazioni che hanno senso solo per i Leaf) - Rischio di **compromettere la trasparenza**: il client potrebbe dover distinguere fra Leaf e Composite - Aggiunge **complessità inutile** se usato in strutture semplici - La **gestione dei figli** (aggiunta, rimozione) può creare ambiguità se lasciata libera

5. Decorator

È uno dei più raffinati tra i design pattern strutturali. Il suo scopo è aggiungere dinamicamente nuovi comportamenti a un oggetto, senza dover modificare il codice della classe originale, e senza creare sottoclassi multiple per ogni possibile combinazione di funzionalità.

Supponiamo di avere un oggetto di tipo `TextView` che mostra del testo. A volte vuoi che il testo abbia il bordo, altre volte scrollabile, altre ancora colorato. Si potrebbe pensare di creare tante sottoclassi (es. BordedTextView, ScrollTextView, ScrollAndBordedTextView...) ma diventa presto ingestibile.

Con il Decorator, invece, possiamo comporre dinamicamente questi comportamenti, creando oggetti che avvolgono altri oggetti, aggiungendo logica

prima o dopo la chiamata originale.

Il decorator si struttura con 4 componenti:

1. **Component**: Un interfaccia comune che definisce l'operazione base.
2. **ConcreteComponent**: La classe reale che implementa `Component`, quella da decorare.
3. **Decorator**: Una classe astratta che implementa `Component` e contiene un riferimento a un altro component (quello da decorare).
4. **ConcreteDecorator**: Estende `Decorator`, aggiungendo comportamenti prima o dopo aver delegato la chiamata all'oggetto decorato.

```java
interface VisualComponent {
    void draw();
}

// Componente base
class TextView implements VisualComponent {
    public void draw() {
        System.out.println("Disegno del testo");
    }
}

// Decorator astratto
abstract class ComponentDecorator implements VisualComponent {
    protected VisualComponent component;

    public ComponentDecorator(VisualComponent component) {
        this.component = component;
    }

    public void draw() {
        component.draw();  // delega al componente decorato
    }
}

// Concrete Decorator
class BorderDecorator extends ComponentDecorator {
    public BorderDecorator(VisualComponent component) {
        super(component);
    }

    public void draw() {
        super.draw();
        System.out.println("...e disegno un bordo");
    }
}

class ScrollDecorator extends ComponentDecorator {
    public ScrollDecorator(VisualComponent component) {
        super(component);
```

```
    }

    public void draw() {
        super.draw();
        System.out.println("...e aggiungo lo scrolling");
    }
}

VisualComponent text = new TextView();
VisualComponent decorated = new BorderDecorator(new ScrollDecorator(text));
decorated.draw();
```

Categoria	Contenuto
Quando usarlo	- Quando vuoi **aggiungere funzionalità a oggetti esistenti** senza modificare il loro codice - Quando non vuoi o non puoi usare l'**ereditarietà** per estendere il comportamento - Quando hai bisogno di **combinare dinamicamente** diversi comportamenti opzionali - Quando il sistema deve essere **estendibile e configurabile a runtime**
Vantaggi	- Permette l'**aggiunta flessibile e modulare** di responsabilità - **Evita la proliferazione di sottoclassi** per ogni combinazione possibile - Favorisce la **composizione** invece dell'ereditarietà - Funziona anche con **classi già esistenti o chiuse alla modifica**
Svantaggi	- L'**annidamento di più decoratori** può rendere il codice difficile da seguire - La **complessità aumenta** con troppi livelli di decorazione - È importante **gestire con attenzione l'ordine** con cui i decoratori vengono applicati - Difficile da debuggare: ogni comportamento può essere "nascosto" in un decoratore diverso

6. Observer

L’Observer serve per creare una relazione uno-a-molti tra oggetti: quando un oggetto cambia stato, tutti gli oggetti che lo osservano vengono automaticamente notificati e aggiornati.

È strutturato mediante 4 principali componenti:

1. **Subject (Observable)**: Espone metodi per aggiungere/rimuovere osservatori e per notificare gli aggiornamenti.
2. **ConcreteSubject**: Implementa il `Subject` e contiene lo stato da osservare. Quando cambia, notific tutti gli observer.
3. **Observer**: Interfaccia che dichiara il metodo `update()`.
4. **ConcreteObserver**: Implementa `Observer` e reagisce agli aggiornamenti del soggetto.

```java
// Observer
interface Observer {
    void update(String news);
}

// Subject
interface NewsAgency {
    void registerObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers(String news);
}

// ConcreteSubject
class RealNewsAgency implements NewsAgency {
    private List<Observer> observers = new ArrayList<>();

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        observers.remove(o);
    }

    public void notifyObservers(String news) {
        for (Observer o : observers) {
            o.update(news);
        }
    }
}

// ConcreteObserver
class EmailSubscriber implements Observer {
    public void update(String news) {
        System.out.println("Email ricevuta: " + news);
    }
}

RealNewsAgency agency = new RealNewsAgency();
```

```
Observer user1 = new EmailSubscriber();

agency.registerObserver(user1);
agency.notifyObservers("È uscito un nuovo articolo!");
```

Categoria	Contenuto
Quando usarlo	- Quando un cambiamento in un oggetto richiede **l’aggiornamento automatico di altri oggetti** - Quando vuoi mantenere una **dipendenza loosely coupled** tra soggetto e osservatori - Quando vuoi implementare **sistemi di notifica, eventi, broadcast**
Vantaggi	- Favorisce il **disaccoppiamento** tra soggetto e osservatori - Gli oggetti possono **reagire ai cambiamenti senza conoscere in dettaglio la logica del soggetto** - Estendibile: puoi aggiungere o rimuovere osservatori **dinamicamente** - È alla base dei sistemi **event-driven**
Svantaggi	- In sistemi grandi, può diventare difficile **tracciare l’ordine e le conseguenze delle notifiche** - Se non gestito bene, può portare a **problemi di performance** (troppe notifiche) - Rischio di **dipendenze nascoste** e comportamenti imprevisti - Il soggetto notifica tutti gli osservatori **senza sapere se sono ancora interessati o validi**

7. Template Method

È un pattern comportamentale che si basa su un’idea molto semplice ma estremamente potente: definire lo scheletro di un algoritmo in una classe astratta, lasciando alcuni passi da implementare alle sottoclassi.

La struttura del pattern consiste di:

1. **AbstractClass**: Contiene il template method che definisce i passi dell'algoritmo. Implementa alcuni passi, lasciando altri come metodi astratti o "hook" da sovrascrivre.
2. **ConcreteClass**: Implementa i metodi astratti, completando così il comportamento.

```
// Classe astratta
abstract class DataProcessor {
    // Template method
    public final void process() {
        readData();
        processData();
        saveData();
    }

    protected abstract void readData();
    protected abstract void processData();

    // Metodo "hook": opzionale da sovrascrivere
    protected void saveData() {
        System.out.println("Salvataggio su file di default");
    }
}

// Classe concreta
class CSVDataProcessor extends DataProcessor {
    protected void readData() {
        System.out.println("Lettura dati da CSV");
    }

    protected void processData() {
        System.out.println("Elaborazione dati CSV");
    }
}

DataProcessor processor = new CSVDataProcessor();
processor.process();  // Esegue lo schema definito dalla superclasse
```

Categoria	Contenuto
Quando usarlo	- Quando hai un algoritmo con una **struttura fissa ma passi variabili** - Quando vuoi **centralizzare il controllo del flusso** lasciando personalizzazione ai figli - Quando vuoi **evitare la duplicazione del codice comune** in più sottoclassi
Vantaggi	- Favorisce il **riuso del codice**: la logica comune è definita una sola volta - Permette di **variare solo ciò che serve**, lasciando intatta la struttura - Riduce l'**effetto domino** delle modifiche (le regole base restano concentrate nella superclasse) - Promuove il principio **"Hollywood"**: "Don't call us, we'll call you"
Svantaggi	- Introduce una **forte dipendenza tra superclasse e sottoclassi** - Più difficile da **comprendere e mantenere** se la gerarchia diventa complessa - L'eccessivo uso di metodi astratti può **forzare le sottoclassi** a scrivere codice non sempre necessario - Poco flessibile in confronto alla **composizione** (che può essere preferibile in certi contesti)

8. Strategy

Permette di definire una famiglia di algoritmi (o comportamenti), incapsularli ciascuno in una classe diversa, e renderli intercambiabili. In questo modo, l'oggetto che li usa non ha bisogno di sapere quale strategia specifica sta usando: sa solo che tutte seguono un'interfaccia comune.
È un'alternativa alla programmazione ad `if` annidati o alla logica dispersa: ogni comportamento è separato e testabile da solo.

È strutturato nel seguente modo:

1. **Strategy**: Definisce il metodo comune che tutte le strategie devono implementare.
2. **ConcreteStrategy**: Una o più classi che implementano `Strategy` e forniscono versioni diverse dell'algoritmo.
3. **Context**: Contiene un riferimento a una `Strategy` e la usa per eseguire l'operazione desiderata.

```java
// Strategy
interface PaymentStrategy {
    void pay(double amount);
}

// Concrete Strategies
class CreditCardPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Pagato " + amount + " con carta di credito");
    }
}

class PayPalPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Pagato " + amount + " con PayPal");
    }
}

// Context
class ShoppingCart {
    private PaymentStrategy strategy;

    public ShoppingCart(PaymentStrategy strategy) {
        this.strategy = strategy;
    }

    public void checkout(double amount) {
        strategy.pay(amount);
    }
}

ShoppingCart cart = new ShoppingCart(new PayPalPayment());
cart.checkout(50.0);

cart = new ShoppingCart(new CreditCardPayment());
cart.checkout(120.0);
```

Categoria	Contenuto
Quando usarlo	- Quando hai **comportamenti diversi** che possono essere **separati e riutilizzati** - Quando vuoi **evitare logiche condizionali complesse** - Quando hai bisogno di **cambiare algoritmo o logica a runtime** - Quando vuoi **testare o estendere comportamenti** in modo modulare
Vantaggi	- **Aggiunta o modifica facile** delle strategie senza toccare il codice del contesto - Favorisce la **composizione** rispetto all'ereditarietà - Ogni strategia è **incapsulata**, testabile e riutilizzabile - Promuove il **principio aperto/chiuso**: puoi aggiungere nuove strategie senza modificare quelle esistenti
Svantaggi	- Può introdurre **troppe classi** se le strategie sono molte - Il contesto deve **conoscere la strategia esternamente**, o delegare la scelta - Se le strategie condividono troppo codice, può esserci **duplicazione o necessità di un'astrazione intermedia** - Meno intuitivo da usare se i comportamenti sono semplici e statici

9. Singleton

Il Singleton è un pattern che garantisce che una classe abbia una sola istanza e fornisce un punto di accesso globale a quella istanza.

È utile quando è necessario un solo oggetto condiviso da tutto il sistema, ad esempio per:

- la configurazione di un'applicazione
- un logger
- una connessione a un database
- un gestore di risorse

L’idea è semplice:

1. Rendere il costruttore privato, così nessuno può creare istanze dall’esterno.
2. Fornire un metodo statico che restituisce l’unica istanza (creata alla prima chiamata).

```
public class Logger {
    private static Logger instance;

    // Costruttore privato
    private Logger() {
        // inizializzazione
    }

    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();  // lazy initialization
        }
        return instance;
    }

    public void log(String message) {
        System.out.println("LOG: " + message);
    }
}
```

Categoria	Contenuto
Quando usarlo	- Quando serve **una sola istanza condivisa** nel sistema (es. logger, configurazione) - Quando serve **controllare l'accesso globale** a una risorsa o a un componente centrale - Quando vuoi **centralizzare lo stato o il coordinamento** di un sistema
Vantaggi	- Garantisce **una sola istanza** (controllo centralizzato) - È facile da implementare e usare - Fornisce un **accesso globale e coerente** a risorse comuni

Categoria	Contenuto
	- Può essere migliorato per essere **lazy o thread-safe**
Svantaggi	- Può violare il **principio di singola responsabilità**: gestisce anche il proprio ciclo di vita - Rende difficile **testare o sostituire** l'oggetto in test unitari - È spesso considerato una forma **"controllata" di global state** - In ambienti multi-thread, può causare **problemi di concorrenza** se non gestito correttamente

Stime nei progetti software

Le attività di stima di `tempi`, `costi` ed `effort` nei progetti software sono effettuate con gli obiettivi di:

- ridurre al minimo il grado di incertezza
- limitare i rischi comportati da una stima

Risulta quindi necessario usare tecniche per incrementare l'affidabilità e l'accuratezza di una stima. Le tecniche di stima possono basarsi su:

- Stime su progetti simili già completati.
- Tecniche di scomposizione
- Modelli algoritmici empirici

Tecniche di scomposizione

Queste tecniche si rifanno alla logica del `divide et impera`. L'idea è quella di tentare di stimare un intero progetto scomponendolo in parti più piccole, più comprensibili e più facilmente valutabili. Parliamo dunque di **stime dimensionali**, che fanno riferimento a misure quantitative della dimensione del software, come:

- **LOC (Line of Code)**: Si cerca di stimare quante righe di codice saranno necessarie per sviluppare ogni componente. Sulla base della produttività media si ricava lo sforzo.
- **Function Point (FP)**: I function points misurano la quantità di funzionalità

offerte dal software all'utente finale, secondo criteri come: input, output, query etc... . Ogni elemento viene pesato in base alla sua complessità. Permettono di stimare l'effort in modo più stabile e sono utili nei contesti in cui si vuole fare una valutazione anticipata del progetto. La formula per calcolare **FP** è: $FP = UFC \cdot TFC$ dove:

- **L'UFC (Unadjusted Function Count)** si ottiene conteggiando e pesando cinque tipi di componenti funzionali che l'utente percepisce nel sistema: External Inputs, External Outputs, External Inquiries, Internal Logical Files (strutture dati del sistema), External Interface Files (strutture dati non mantenute dal sistema). $UFC = \Sigma \left(N_{comp} \cdot weight\right)$
- **TFC (Technical Complexity Factor)** è un fattore di aggiustamento è si calcola partendo da 14 fattori di complessità tecnica. $TFC = 0.65 + 0.01 \cdot \Sigma F_i$, dove F_i è il punteggio di ciascun dei 14 fattori

Modelli algoritmici empirici

Il termine **empirico** indica che le relazioni matematiche non sono derivate da principi teorici assoluti, ma sono basate sull'osservazione e sull'analisi statistica di progetti passati.

Si tratta di un approccio strutturato, che cerca di stimare variabili come l'effort, il costo o la durata del progetto applicando delle formule matematiche, costruite a partire da dati storici reali raccolti su progetti precedenti.

Un esempio di modello è **COCOMO**, sviluppato da Boehm per determinare il valore dell'effort, che viene poi utilizzato per determinare durata e costi di sviluppo. Il modello esprime l'effort in `persone-mese`

$$Effort_N = a \cdot (KLOC)^b$$

- **KLOC** è il numero di migliaia di linee di codice previste,
- **a e b** sono coefficienti empirici, determinati analizzando molto progetti reali.

Calcolato l'effort nominale, ovvero una stima preliminare, possiamo calcolare la quantità di effort come:

$$Effort = C \cdot Effort_N$$

Dove C è un fattore moltiplicativo, chiamato cost driver multiplier, ed è basato su 15 cost drivers.

Il tempo di consegna si deriva poi dalla formula: $T = a \cdot Effort^b$, dove a e b sono due costanti.

Metriche di struttura

Un modulo è una sequenza contigua di statements del programma. Le metriche di struttura sono per determinare quanto un software sia "buono", ovvero misurano la qualità stessa del software.

Ci troviamo nella situazione in cui il software viene diviso in moduli, di conseguenza abbiamo la cosi detta **architettura a moduli**. Si rappresenta l'architettura a moduli cone un albero diretto $S = N, R$ dove:

- ogni nodo $n \in N$ corrisponde a un modulo.
- ogni arco $r \in R$ indica le relazioni tra moduli.

Partendo da questa rappresentazione, vediamo alcune metrice di struttura atte a determinare la qualità del software.

Tree Impurity

La Tree Impurity $m(G)$ misura quanto il grafo G è differente da essere un albero. **Più piccolo è questo valore, migliore è il design**. È definita nel seguente modo:

$$m(G) = \frac{2(e - n + 1)}{(n - 1)(n - 2)}$$

Riuso Interno

È una misura che indica il grado di riutilizzo dei moduli all'interno dello stesso prodotto. **Più piccolo è il valore, meno è il riutilizzo**. Si calcola nel seguente modo: $r(G) = e - n + 1$.

Information Flow

Le misure di Information Flow assumono che la complessità di un modulo dipende da 2 fattori:

- La complessità del codice del modulo
- La complessità delle interfacce del modulo

Il livello totale di Information Flow attraverso un sistema è un'**attributo inter-modulare**, mentre se considerato il livello di Information Flow attraverso un singolo modulo è un'**attributo intra-modulare**. Le misure di Information Flow si basano su:

- Flusso **locale**, che può essere diretto (un modulo chiama un altro modulo) o indiretto (flusso derivante dai valori di ritorno).
- Flusso **globale**, informazioni che vengono passate tra i moduli secondo una struttura dati globale.

Queste misure dipendono dal livello di interconnessione tra i moduli, si definiscono:

- **Fan-In**: di un modulo M è il numero di flussi locali (diretti + indiretti) che terminano su M più il numero di flussi globali le cui informazioni sono prelevate da M. Un valore basso indica che lui è **influenzato/controllato** da molti altri moduli.
- **Fan-Out**: di un modulo M è il numero di flussi locali (diretti + indiretti) che iniziano da M più il numero di flussi globali aggiornati da M. Un valore alto indica che lui **influenza/controlla** molti altri moduli.

L'Information Flow (IF) di un modulo M è dunque definito come segue (Henry & Kafura):

$$IF(M) = [fan - in(M) \cdot fan - out(M)]^2$$

Mentre L'IF di tutto il sistema con n moduli:

$$IF = \sum_{i=1}^{n} IF(M_i)$$

Misure Strutturali

Le strutture hanno 3 componenti: Strutture Control-Flow (sequenza di esecuzioni di istruzioni), Data Flow (tenere traccia dei dati creati o gestiti dal programma), Strutture Dati (organizzazione dei dati).

Flowgraph

Le strutture Control-Flow sono modellate come un flowgraph, ovvero un grafo diretto $FG = \{N, E\}$.

- Ogni nodo $n \in N$ rappresenta uno statement del programma. Abbiamo diverse tipologie di nodi:
  - **nodi procedurali**: nodi con grado di uscita 1,
  - **nodi predicati**: nodi con grado di uscita > 1,
  - **nodo iniziale**: nodi con grado di entrata 0,
  - **nodo finale**: nodi con grado di uscita 0.
- Ogni arco diretto $e \in E$ indica il flusso di controllo di uno statement ad un'altro statement.

1. **Sequencing**: Siano F_1, F_2 due flowgraphs. Allora la sequenza di $F_1, F_2 (F_1; F_2)$ è un'altro flowgraph formato unendo il nodo terminale di F_1 con il nodo iniziale di F_2.
2. **Nesting**: Siano F_1, F_2 due flowgraphs. Allora l'annidamento di F_1, F_2 sul nodo x descitto come $(F_1; F_2)$ è un'altro flowgraph formato partendo da F_1 sostituendo l'arco da x con l'intero F_2.
3. **Flowgraph Primi**: I flowgraph primi sono flowgraph che non possono essere decomposti in modo non banale tramite sequenze e nidificazioni. Ogni flowgraph ha una decomposizione `unica` in una gerarchia di primi, chiamata **Albero di Decomposizione**.

Di seguito alcuni flowgraph primi comuni.

Le D-structures rappresentano le strutture di controllo fondamentali promosse da Dijkstra:

- D_0: Sequenza lineare (es: istruzione 1 -> istruzione 2 -> istruzione 3)
- D_1: Selezione (es: if/else)

- D_2: Iterazione con controllo in testa (es: while)
- D_3: Iterazione con controllo in coda (es: do/while)

Misure Gerarchiche

È un modo di definire le misure dei flowgraph usando l'albero di decomposizione. È basato sull'idea che afferma che noi possiamo misurare un attributo del flowgraph nel seguente modo:

- definendo la misura per il flowgraph prime
- descrivendo come l'operazione di sequencing affetti l'attributo
- descrivendo come l'operazione di nesting affetti l'attributo

Depth of Nesting

La depth di un flowgragh $n(F)$ può essere misurata in termini di:

- **Primes**
  $n(P_1) = 0; n(P_2) = n(P_3) = \cdots = n(P_k) = 1$
  $n(D_0) = 0; n(D_1) = n(D_2) = n(D_3) = 1$
- **Sequencing**
  $n(F_1; F_2; \ldots F_k) = \max\{n(F_1) \ldots n(F_k)\}$
- **Nesting**
  $n(F(F_1, F_2, \ldots F_k)) = 1 + \max\{n(F_1) \ldots n(F_k)\}$

D-Structuredness

Molte definizioni popolari di programmazione strutturata asseriscono che il programma è strutturato se può essere composto usando solo un numero piccolo di costrutti ammissibili. La definizione informale di programmazione strutturale può essere espressa formalmente asserendo che un programma è strutturato $\Leftrightarrow$ è **D-strutturato**.

- **Primes**
  $n(P_1) = 0; n(D_0) = 0; n(D_1) = n(D_2) = n(D_3) = 1$
  0 altrimenti.
- **Sequencing**
  $n(F_1; F_2; \ldots F_k) = \min\{n(F_1) \ldots n(F_k)\}$

- **Nesting**
  $n(F(F_1, F_2, \dots F_k)) = \min\{n(F_1) \dots n(F_k)\}$

Un programma (o meglio, il suo flowgraph, cioè il grafo che rappresenta il flusso del controllo) è considerato D-strutturato quando ha un valore di D-Structuredness basso, idealmente 0 o 1.

Complessità Ciclomatica

La complessità di un programma può essere misurata dal numero **ciclomatico** del flowgraph del programma. Si può calcolare in due modi diversi:

1. **Flowgraph Based**: La complessità ciclomatica $v(F) = e - n + 2$, misura il numero di percorsi lineari indipendenti in F.
2. **Code Based**: $v(F) = 1 + d$ dove d è il numero di nodi predicato, ovvero il numero di nodi decisionali.

- Primes: $v(F) = 1 + d$
- Sequencing: $v(F_1; F_2; \dots F_n) = \sum_{i=1}^{n} v(F_1) - n + 1$
- Nesting: $v(F(F_1, F_2, \dots F_n)) = v(F) + \sum_{i=1}^{n} v(F_1) - n + 1$

Complesità essenziale di McCabe

La complessità essenziale di un programma con flowgraph F è data da $ev(F) = v(F) - m$ con m il numero di sub-flowgraphs D_0, D_1, D_2, D_3

Software Quality

Il modello di qualità del software si basa principalmente sullo **standarda IEEE 1061**, che fornisce una guida strutturata per la definizione e valutazione della qualità del software attraverso l'identificazione e l'utilizzo di metriche specifiche.

Si definisce **qualità del software** il livello con cui un software presenta una combinazione di attributi desiderati.

1. **Struttura**
   Lo standard organizza la qualità del software seguendo i principi chiave che

permettono di:

- Identificare gli **obiettivi di qualità** rilevanti per il progetto o sistema in questione
- Determinare gli **attributi di qualità** che rappresentano aspetti misurabili del software
- Sviluppare e applicare **metriche** che consentano di valutare tali attributi.

2. **Indici di qualità**
   Gli indici di qualità aggregano metriche per fornire un quadro d'insieme delle prestazioni del software rispetto a determinati obiettivi di qualità. Sono suddivisi in 3 gruppi principali, che rispettano le attività nel modello McCall, Il quality triangle. Sulle ascisse troviamo la percentuale di attività svolte dai vari attori sul prodotto mentre sulle ordinate il tempo, che parte dal momento di rilascio. Da questo periodo in poi il software è usato dagli utenti e nel mentre gli sviluppatori svolgono attività di monitoraggio e manutenzione. Quando gli utenti terminano di usare il software, esso passa nello stadio di dismissione, che sappiano non è istantaneo. Ciò che fa McCall è definire degli attributi per ciascuna attività di Revision, Transition o Operation, detti indici di qualità.

ID	Categoria	Caratteristica	Descrizione
i1	Operation	Correttezza	Indica quanto bene il software soddisfa le specifiche e le reali esigenze degli utenti.
i2	Operation	Affidabilità	Rappresenta la capacità del software di eseguire le sue funzioni con precisione e continuità.
i3	Operation	Efficienza	Misura il consumo di risorse computazionali, come CPU, memoria o tempo di esecuzione.
i4	Operation	Integrità	Valuta la sicurezza del sistema rispetto a accessi indesiderati o non autorizzati.
i5	Operation	Usabilità	Riguarda la facilità con cui l’utente può imparare e usare efficacemente

ID	Categoria	Caratteristica	Descrizione
			il software.
i6	Revision	Manutenibilità	Misura la facilità con cui si possono rilevare e correggere errori nel software.
i7	Revision	Testabilità	Indica quanto sia semplice verificare che il software funzioni correttamente attraverso i test.
i8	Revision	Flessibilità	Descrive la facilità di modificare il software per adattarlo a nuovi requisiti.
i9	Transition	Portabilità	Esprime lo sforzo necessario per trasferire il software su ambienti diversi.
i10	Transition	Riusabilità	Indica il grado in cui parti del software possono essere riutilizzate in altri contesti.
i11	Transition	Interoperabilità	Valuta la capacità del software di interagire con altri sistemi o componenti.
i12	Transition	Evolubilità	Misura la facilità di aggiornare il software per rispondere a nuovi requisiti o cambiamenti.

3. **Attributi di qualità**
   Avendo definito gli indici di qualità, ora bisogna trovare un modo per misurarli, e per fare ciò si fa uso degli **attributi di qualità**.

Caratteristica	Descrizione
Complessità	Rappresenta quanto il software è comprensibile nel suo

Caratteristica	Descrizione
	funzionamento, nelle strutture e nei flussi logici.
Accuratezza	Indica la precisione con cui il software esegue i calcoli o fornisce i risultati attesi.
Completezza	Misura quanto il software implementa completamente tutte le funzionalità richieste.
Consistenza	Si riferisce all'uso coerente di design, terminologia e convenzioni all'interno del sistema.
Error Tolerance	Valuta la capacità del software di gestire errori o anomalie senza interrompere il funzionamento.
Tracciabilità	Rappresenta il livello di connessione tra elementi del progetto (ad esempio tra codice e requisiti).
Espandibilità	Indica quanto facilmente il software può essere esteso, ad esempio in termini di nuove funzionalità o risorse.
Generalità	Misura la capacità del software di essere utilizzato in contesti diversi da quelli originari.
Modularità	Descrive il grado di indipendenza tra i vari componenti o moduli del sistema.
Auto-documentation	Riguarda la presenza di strumenti o funzionalità integrate (come help in linea) che assistono l'utente.

4. **Processo di valutazione della qualità**
   Lo standard IEEE 1061 descrive un processo chiaro e iterativo per valutare la qualità del software.
   - Si definiscono gli obiettivi di qualità, che dipendono dal contesto applicativo del software.
   - Si identificano gli attributi in base agli obiettivi di qualità.
   - Si selezionano le metriche di rappresentazione per ogni attributo.
   - Le metriche vengono poi calcolate e confrontate con valori attesi o accettabili.

- Infine, in base ai risultati si procede con il miglioramente continuo del software.

SQA (Software Quality Assurance)

La garanzia della qualità del software (SQA) è un approccio pianificato e sistematico per garantire che sia il processo che il prodotto software siano conformi agli standard, processi e procedure stabiliti. L'obiettivo della SQA è quello di migliorare la qualità del software monitorarando sia il software che il processo di sviluppo per garantire la piena conformità con gli standard e le procedure stabiliti. Il ruolo della SQA è quello di fornire al management la garanzia che il processo ufficialmente stabilito sia effettivamente implementato.

Gli standard in questo contesto rappresentano delle linee guida alle quali il progetto software dovrebbe esser comparato per assicurarsi di procedere bene (gli standard definiscono **QUELLO** che in generale dovrebbe essere fatto).
Il minimo standard indispendabile da seguire include:

- **Documentation Standard** (es. che si utilizzi un template per il documento di specifica dei requisiti piuttosto che inventarsene la struttura di sana pianta)
- **Design Standard** (standard di progettazione, si vuole cioè trasformare i requisiti software in progettazione software utilizzando un’adeguata documentazione progettuale come linguaggi di modellazione, specifiche di definizione degli algoritmi…)
- **Code Standard** (standard di codifica per ottenere codice che sia quanto più possibile uniforme)

Quando si utilizza il termine "Procedure" invece si fa riferimento alle linee guida che suggeriscono **COME** procedere effettivamente nello sviluppo (nella specifica, progettazione etc…) anche suggerendo chi deve farla, quando e cosa fare con i risultati.

Testing

La fase di testing si assicura che un sistema software sia conforme alle sue specifiche e che soddisfi le esigenze dell'utente. In particolare andremo a parlare di

Defect Testing.

Il goal del **Defect Testing** è quello di scoprire **difetti** nei programmi, e va in contrasto con il validation testing, che richiede che il sistema funzioni corretamente e che abbia le funzionalità richieste dal cliente. Un defect test riuscito è un test che causa un comportamento anomalo all'interno del programma. Ci sono due fasi nel defect testing:

- **Component testing**: Si testano le componenti individuali del programma.
- **Integration testing**: Si testano gruppi di componenti integrati per creare un sistema o sottosistema.
- **User testing**: Anche noto come validation testing o acceptance testing, in questa fase non si cercano difetti nel codice, ma si verifica se il software soddisfa i requisiti del cliente. In genere viene eseguito dall'utente finale o il cliente, infatti questa fase di testing viene fatta nei software a contratto, quando cliente e sviluppatore non coincidono.

Per la fase di testing si seguono delle politiche, ovvero:

- Solo test esaustivi possono mostrare che il programma sia libero di difetti
- I test devono essere basati su un sottoinsieme di possibili test case, in accordo con le politche de devono essere visionate dal team di V&V
- Testare situazioni tipiche è più importante che testare casi limite.

Parliamo adesso dei due approcci che esistono nel testing software:

- **Black-box testing**: si concentra sul comportamento esterno del software. Il tester non ha visibilità del codice sorgente: ossera input e output, cercando di capire se il software fa quello che deve fare, come da specifiche. Se l'output non è quello atteso allora il test defect ha avuto successo.
- **White-box testing**: si concentra sull'interno del software. Il tester analizza il codice, la logica, i flussi interni, e crea test basati sulla struttura del programma. Si parla quindi di path testing, ovvero di creare test case per assicure che ciascun statement venga eseguito almeno una volta.

In base a questa classificazione, abbiamo un insieme di approcci al testing:

- **Integration Testing**: si testano gruppi di componenti integrati per formare un

sistema o un sottosistema. L'obiettivo è verificare che le componenti collaborino corretamente. È un tipo di testing `black box`, perché si testa l'integrazione a livello funzionale, senza entrare nel dettaglio del codice.

- **Interface Testing**: ha l'obiettivo di trovare difetti nelle interfacce dei componenti, come dati malformati, assunzioni errate su formati e protocolli. È anch'esso `black box` perché si osserca l'integrazione esterna tra moduli, non il loro funzionamento interno.
- **Stress Testing**: si testa al massimo il comportamento del software in codizioni estreme, ad esempio con un carico superiore al massimo previsto. Serve a rivelare problemi di stabilità, gestione della memoria, prestazioni degradate. Fa parte della categoria `black box` perché interessa come il sistema si comporta globalmente sotto pressione.
- **Object-Oriented Testing**: si testano le classi, si analizzano gli oggetti da queste classi in vari contesti. Fa parte della categoria `white box`, perché richiede di comprendere la struttura della classe.
- **Object Class Testing**: è un sottoinsieme dell'object-oriented testing, si testa ogni operazione della classe in tutti i modi possibili, cercando di capire ogni combinazione significativa di stato/azione. È anch'esso ovviamente `white box`.
- **Scenario Based Testing**: È un tipo di testing che deriva dai casi d'uso. Si identificano scenari realistici (l'utente esegue una ricerca, salva un file, etc...) e si osservano le interazioni tra oggetti. È anch'esso `white box`.