

# Risposte-Domande orali

- [Design pattern](#)
  - [Classificazione](#)
  - [Descrizione dei DS](#)
  - [Lista DS](#)
    - [Abstract Factory](#)
    - [Factory Method](#)
    - [Adapter](#)
    - [Composite](#)
    - [Decorator](#)
    - [Observer](#)
    - [Template Method](#)
    - [Strategy](#)
    - [Singleton](#)
  - [Domanda 1 - Significato Include/Exclude negli Use Case](#)
  - [Domanda 2](#)
  - [Domanda 3 - Differenze tra requisiti funzionali/non funzionali](#)
  - [Domanda 4 - Caratteristiche modello a spirale](#)
  - [Domanda 4.1 - Cosa significa fare Risk analysis](#)
    - [Fasi principali](#)
  - [Domanda 6 - Architettura Oggetti Distribuiti](#)
  - [Domanda 7 - ORB](#)
    - [Funzionamento](#)
  - [Domanda 8 - Attivazione Use Case Progetto](#)
  - [Domanda 9 - Che relazione c'è tra prodotto e costo di produzione?](#)
  - [Domanda 11 - Implementazione SOA](#)
  - [Domanda 12 - Secondo quali attività va declinata la fase di testing?](#)
  - [Domanda 13 - Come si chiama il testing di validazione per i software a contratto?](#)
  - [Domanda 14 - Adapter](#)
    - [Differenze tra Adapter su classi e oggetti](#)
      - [Adapter su Classi](#)
      - [Adapter su Oggetti](#)
      - [Quando usare Class Adapter o Object Adapter?](#)
  - [Domanda 16 - Ciclo di vita del SW](#)
    - [Dettaglio delle fasi](#)
      - [1. Requisiti](#)
      - [2. Specifiche \(Analisi dei requisiti\)](#)

- [3. Pianificazione](#)
- [4. Progetto](#)
- [5. Codifica](#)
- [6. Integrazione](#)
- [Domanda 17 - Qual è la manutenzione più frequentemente utilizzata?](#)
- [Domanda 18 - Regola 10-90](#)
- [Domanda 19 - Stima durata progetto](#)
- [Domanda 20 - Organizzazione modello di qualità del SW](#)
  - [SQA \(Software Quality Assurance\)](#)
- [Domanda 21 - Factory Method a cosa serve](#)
- [Domanda 22 - Motivazioni dietro Abstract Factory](#)
- [Domanda 23 - Diagramma UML per interazione tra oggetti](#)
- [Domanda 24 - COCOMO](#)
  - [Formula di COCOMO](#)
  - [Esempio](#)
- [Domanda 25 - Caratteristiche del modello Microsoft](#)
- [Domanda 26 - Differenze tra testing black/white box](#)
  - [Black-Box testing](#)
    - [Equivalence Partitioning](#)
  - [White-Box testing](#)
    - [Path testing](#)
  - [Differenze tra white/black box testing](#)
- [Domanda 27 - Affidabilità SW](#)
  - [Affidabilità](#)
  - [Caratteristiche dell'affidabilità SW](#)
- [Domanda 28 - Specifica requisito di affidabilità](#)
  - [Come specificare un requisito di affidabilità](#)
- [Domanda 29 - Impatto sui costi](#)
- [Domanda 30 - Stima della dimensione del prodotto SW](#)
  - [Tecniche di scomposizione](#)
    - [LOC - Lines of Codes](#)
    - [FP - Function Points](#)
      - [Componenti FP](#)
    - [Calcolo TCF](#)
- [Domanda 31 - Cos'è un software critico?](#)
  - [Caratteristiche:](#)
  - [Esempi:](#)
- [Domanda 32 - PetriNet](#)
  - [Esecuzione delle PetriNet](#)

- [Domanda 33/33.1 - Component Framework](#)
  - [Funzionamento](#)
- [Domanda 34 - Qual è il metodo per certificare chi produce il software?](#)
- [Domanda 35 - Modello McCall](#)
- [Domanda 36 - Cosa si intende per durata di un prodotto SW?](#)
- [Domanda 37 - CMM](#)
- [Gestione progetti software](#)
  - [Le quattro "P"](#)
- [Metriche di Struttura](#)
  - [Tree Impurity](#)
  - [Riuso Interno \(Misura di Yin e Winchester\)](#)
  - [Information Flow](#)
    - [Fan-In e Fan-Out](#)
    - [Misura di IF \(Henry & Kafura\)](#)
  - [Misure Strutturali](#)
    - [Flowgraph](#)
      - [Sequencing](#)
      - [Nesting](#)
      - [Flowgraph primi](#)
    - [Misure Gerarchiche](#)
      - [Depth of Nesting](#)
      - [D-Structuredness](#)
    - [Complessità Ciclomatica](#)
    - [Complessità essenziale di McCabe](#)
- [Testing](#)
  - [Defect testing](#)

## Design pattern

**Caratteristiche :**

- Rappresentano soluzioni a problematiche ricorrenti che si incontrano durante le varie fasi di sviluppo del software.
- Organizzano l'esperienza di OOD favorendone il riuso
- Permettono di definire un linguaggio comune che semplifica la comunicazione tra gli addetti ai lavori.
- Portano di norma ad una buona progettazione

## Classificazione

2 criteri, *scopo* (purpose) e *raggio di azione* (scope)

Criterio scopo :

- **Creazionali** : I pattern di questo tipo sono relativi alle operazioni di creazione di oggetti
- **Strutturali** : Sono utilizzati per definire la struttura del sistema in termini della composizione di classi ed oggetti
- **Comportamentali** : Permettono di modellare il comportamento del sistema definendo le responsabilità delle sue componenti e definendo le modalità di interazione.

Criterio raggio di azione :

- **Classi** : Pattern che definiscono le relazioni fra classi e sottoclassi. Le relazioni sono **statiche**
- **Oggetti** : Pattern che definiscono relazioni tra oggetti. Le relazioni sono **dinamiche**

		Scopo		
		Creazionale	Strutturale	Comportamentale
Raggio d'azione	Classi	<b>Factory Method</b>	<b>Adapter</b> (class)	Interpreter <b>Template Method</b>
	Oggetti	<b>Abstract Factory</b> Builder Prototype Singleton	<b>Adapter</b> (object) Bridge <b>Composite</b> <b>Decorator</b> Facade Flyweight Proxy	Chain of responsibility Iterator Mediator Memento <b>Observer</b> State <b>Strategy</b> Visitor

## Descrizione dei DS

- **Nome e Classificazione** : Il nome illustra l'essenza di un pattern definendo un vocabolario; la classificazione lo identifica in termini di scopo e raggio
- **Motivazione** : Scenario che descrive in modo astratto il **problema al quale applicare il pattern**
- **Applicabilità** : Descrive le situazioni in cui il pattern può essere applicato
- **Struttura** : Descrive graficamente la configurazione
- **Partecipanti** : Classi ed oggetti che fanno parte del pattern con le relative responsabilità
- **Conseguenze** . Risultati che si ottengono applicando il pattern
- **Implementazioni** : Tecniche e suggerimenti utili all'implementazione
- **Codice di esempio**
- **Usi conosciuti** : Esempi di applicazione in sistemi reali
- **Pattern Correlati** : Altri pattern correlati

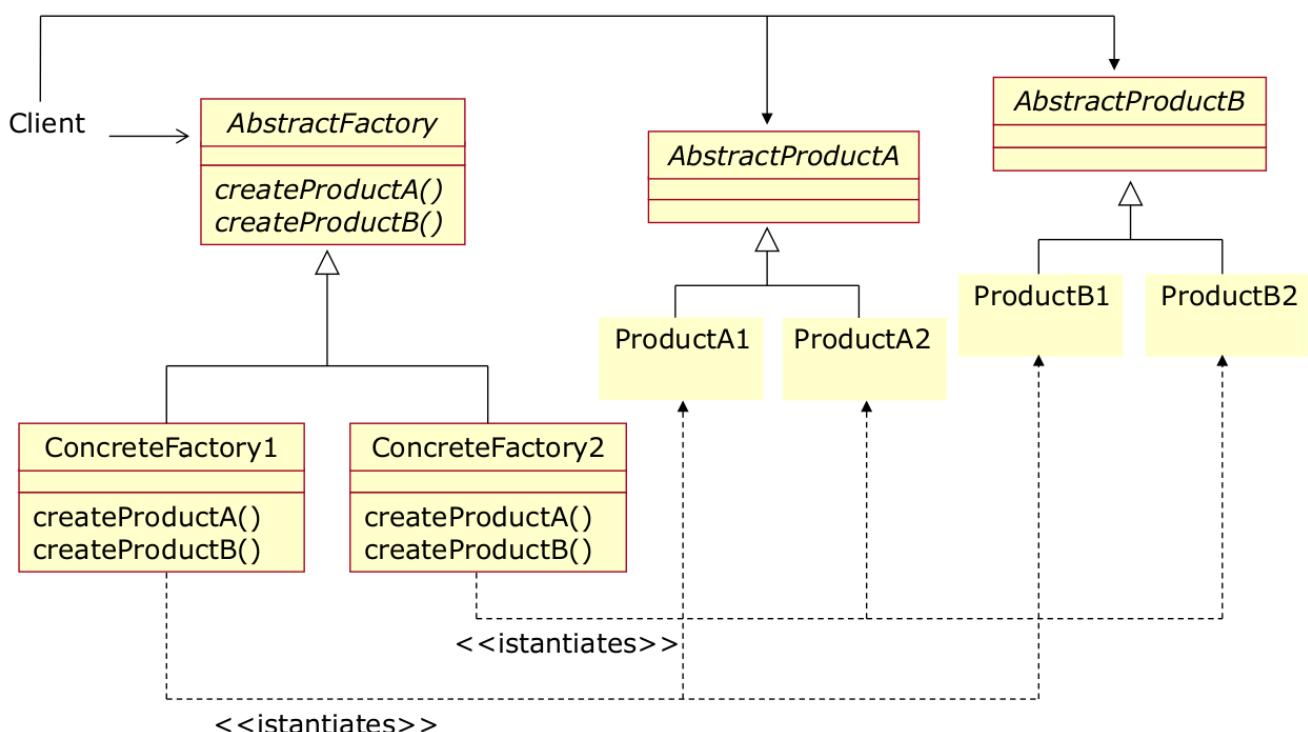
# Lista DS

## Abstract Factory

- **Scopo** : Fornire una interfaccia per la creazione di famiglie di oggetti tra loro correlati
- **Motivazione** : Realizzazione di uno strumento per lo sviluppo di *user interface* (UI) in grado di sopportare diversi tipi di *look & fell*. Per garantire la portabilità di una applicazione tra look & feel diversi, gli oggetti non devono essere cablati nel codice.

Classificazione : Creazionale basato su oggetti

### Struttura



### Applicabilità :

- A sistema che deve essere **indipendente** dalle modalità di creazione dei prodotti con cui opera
- A sistema che deve poter essere configurato per usare **famiglie di prodotti diversi**
- Il client **non deve essere legato** ad una specifica famiglia

### Partecipanti

- **AbstractFactory** e **ConcreteFactory**
- **AbstractProduct** e **ConcreteProduct**
- Applicazione Client

### Conseguenze :

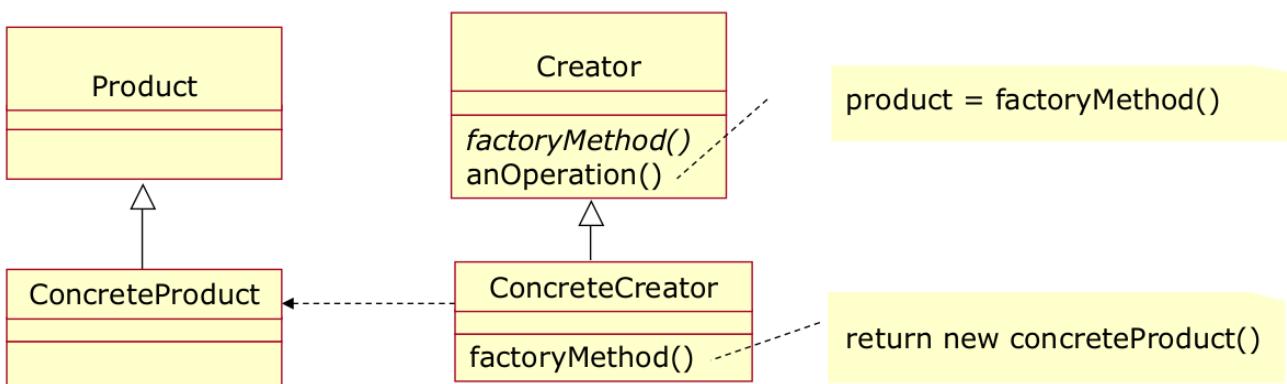
- Le classi concrete sono isolate e sotto controllo

- La famiglia di prodotti può essere cambiata rapidamente perché la factory completa compare in un unico punto del codice
- Aggiungere nuove famiglie di prodotti richiede ricompilazione perché l'insieme di prodotti gestiti è legato all'interfaccia della factory

## Factory Method

- **Scopo** : Definire una interfaccia per la creazione di un oggetto, che consenta di decidere a tempo di esecuzione quale specifico oggetto istanziare
- **Motivazione** : È un pattern ampiamente usato nei framework, dove le classi astratte definiscono le relazioni tra gli elementi del dominio, e sono responsabili per la creazione degli oggetti concreti.

Classificazione : Creazionale basato su classi



### Affidabilità :

- Una classe non è in grado di sapere in anticipo le classi di oggetti che deve creare.
- Una classe vuole che le sue sottoclassi scelgano gli oggetti da creare.
- Le classi delegano la responsabilità di creazione.

### Partecipanti

- **Product** e **ConcreteProduct**
- **Creator** e **ConcreteCreator**

### Conseguenze

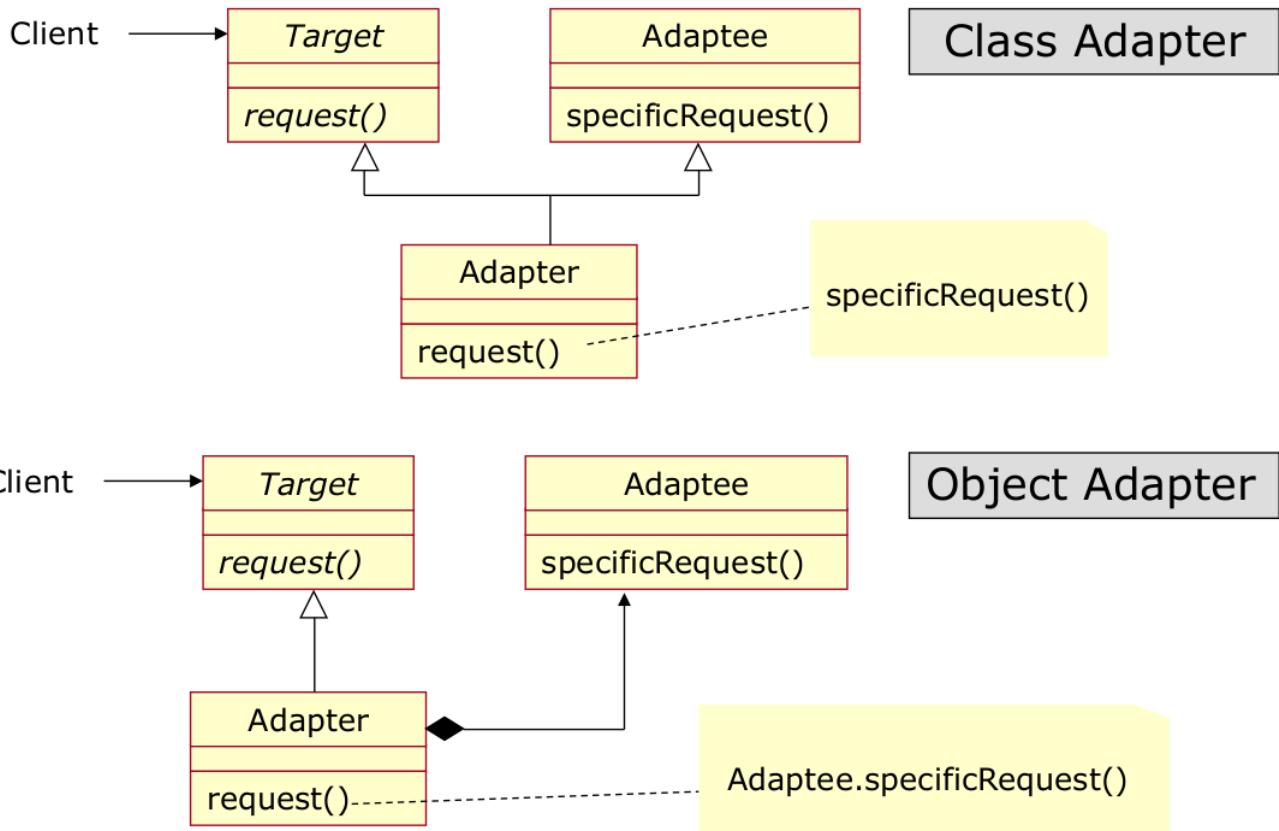
- Elimina la necessità di riferisi a classi dipendenti dall'applicazione all'interno del codice.

## Adapter

- **Scopo** : Convertire l'interfaccia di una classe esistente **incompatibile** con un client, in una **compatibile**
- **Motivazione** : Consideriamo un editor che consente di disegnare e comporre elementi grafici. L'astrazione chiave è un singolo oggetto grafico. Supponiamo di voler integrare

un nuovo componente, ma che questo non abbia una interfaccia compatibile con l'editor

Classificazione : Strutturale basato su classi/oggetti



- **Applicabilità**

- Si usa quando si vuole riusare una classe esistente, ma con interfaccia incompatibile con quella desiderata.

- **Partecipanti**

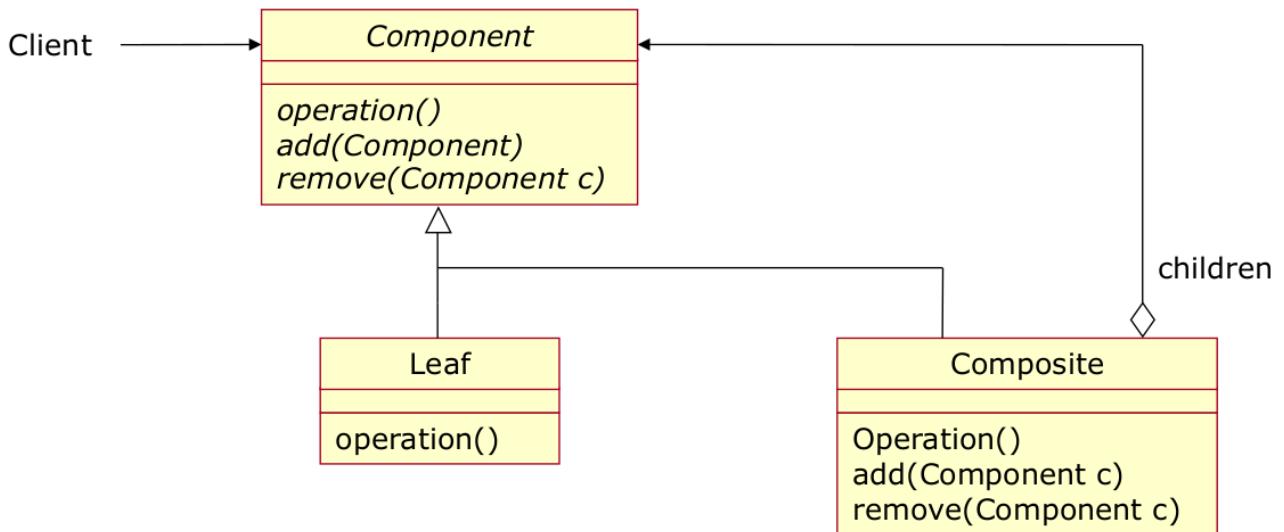
- Client
- Target
- Adapter ed Adaptee

- **Conseguenze:** E' necessario prendere in considerazione l'effort necessario all'adattamento

## Composite

- **Scopo:** Comporre oggetti in strutture che consentano di trattare i singoli elementi e la composizione in modo uniforme.
- **Motivazione:** Le applicazioni grafiche consentono di trattare in modo uniforme sia le forme geometriche di base (linee, cerchi,...) sia gli oggetti complessi che si creano a partire da questi elementi semplici. Molti editor grafici ad esempio hanno la funzione raggruppa

Classificazione : Strutturale basato su oggetti



- **Applicabilità**

- Si usa quando si vogliono rappresentare gerarchie di oggetti in modo che oggetti semplici e oggetti compositi siano trattati in modo uniforme.

- **Partecipanti**

- Component e Composite
- Leaf
- Client

- **Conseguenze**

- I client sono semplificati perché gli oggetti semplici e quelli composti sono trattati allo stesso modo.
- L'aggiunta di nuovi oggetti Leaf o Composite è semplice, e questi potranno sfruttare il codice dell'applicazione Client già esistente.
- Può rendere il sistema troppo generico. Non è possibile fare in modo che un oggetto composito contenga solo un certo tipo di oggetti

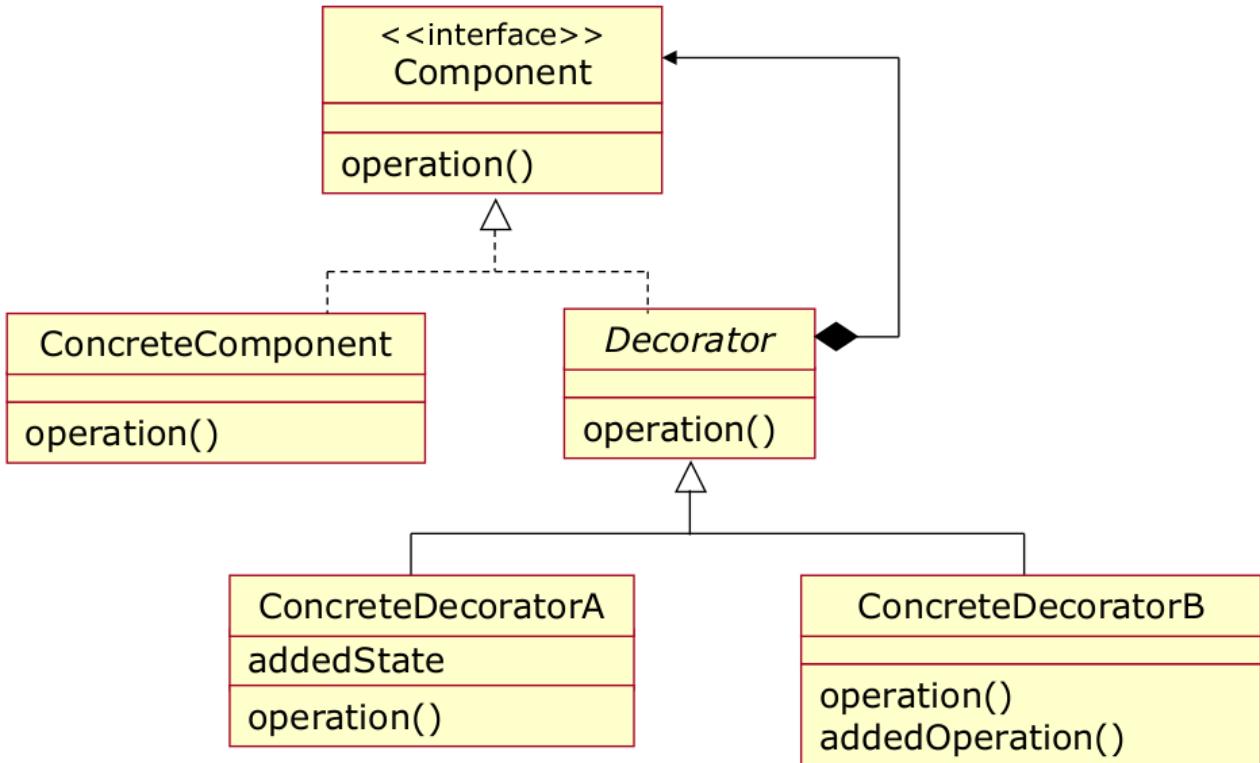
## Decorator

- **Scopo:** Aggiungere dinamicamente funzionalità (responsabilità) ad un oggetto.

Il subclassing è una alternativa statica e il cui scope è a livello di classe e non di singolo oggetto.

- **Motivazione:** Uno scenario classico di applicabilità per questo pattern è la realizzazione di interfacce utente. Responsabilità quali il testo scorrevole o un particolare bordo devono poter essere aggiunti a livello di singolo oggetto

Classificazione : Strutturale basato su oggetti



- **Applicabilità**

- Si applica quando è necessario aggiungere responsabilità agli oggetti in modo trasparente e dinamico.
- Si applica quando il subclassing non è adatto.

- **Partecipanti**

- Component e ConcreteComponent
- Decorator e ConcreteDecorator(s)

- **Conseguenze:**

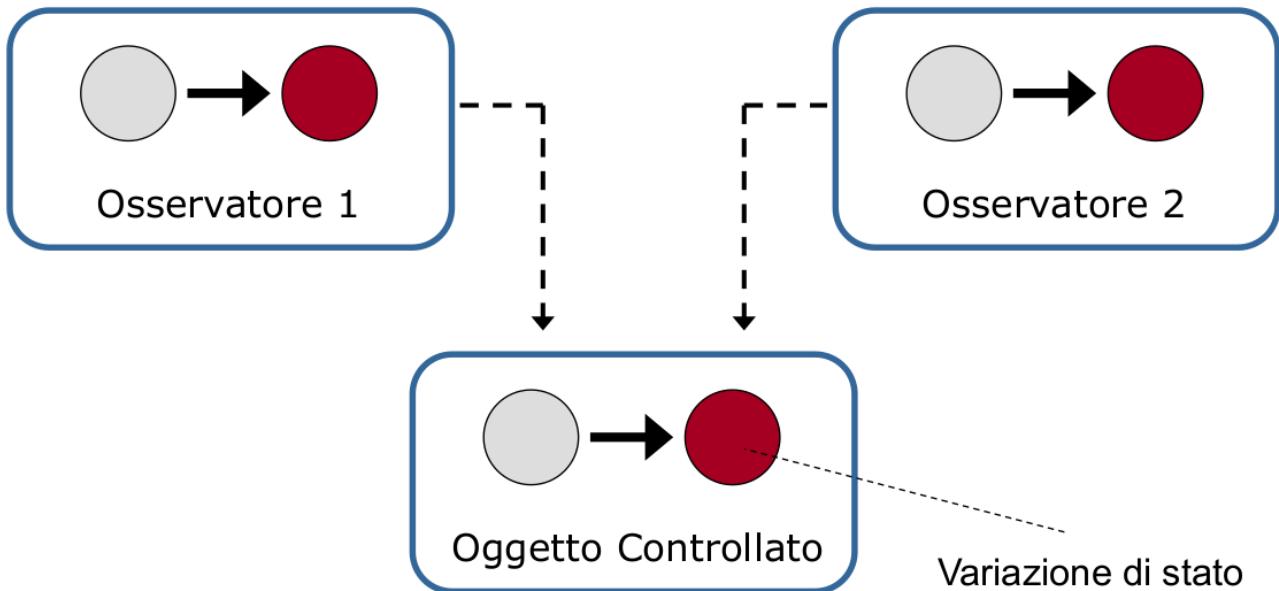
- Maggiore flessibilità rispetto all'approccio statico
- Evita di definire strutture gerarchiche complesse

## Observer

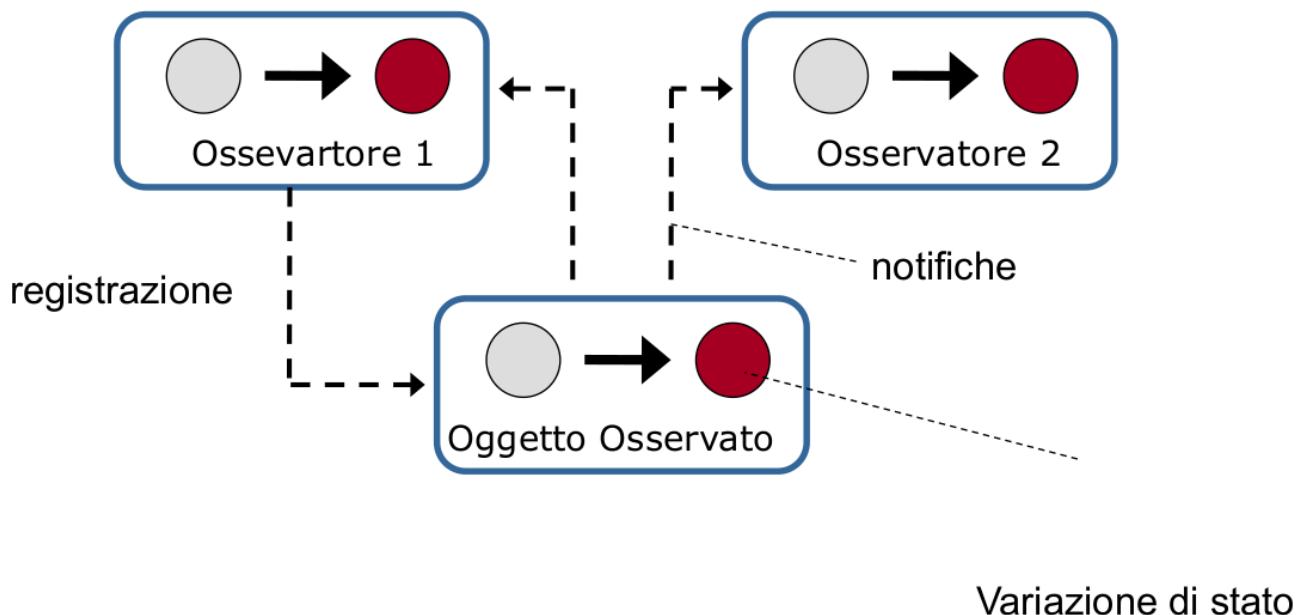
- **Scopo:** Definire una *dipendenza uno a molti tra oggetti*, mantenendo basso il grado di coupling. In altre parole la variazione dello stato di un oggetto deve essere osservata da altri oggetti, in modo che possano aggiornarsi automaticamente.
- **Motivazione:** Lo scenario classico è quello di applicazioni con GUI, realizzate secondo il paradigma **Model-View-Control**. Quando il Model cambia, gli oggetti che implementano la View devono aggiornarsi

Classificazione : Comportamentale basato su oggetti

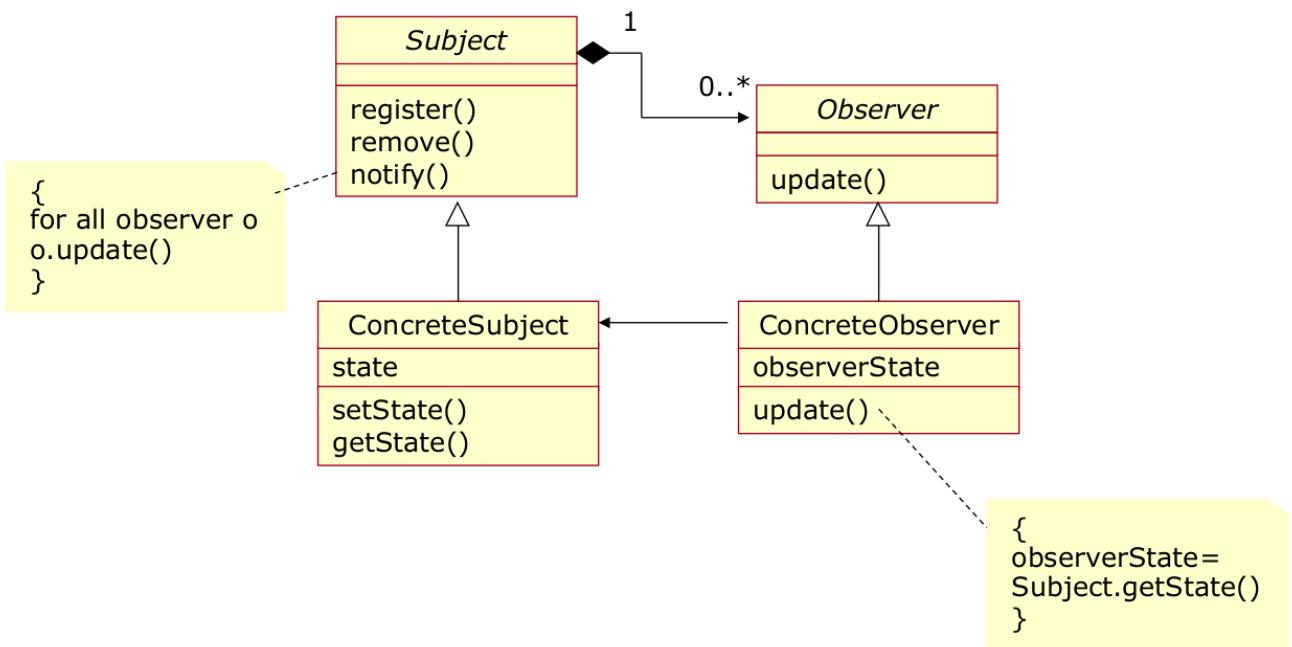
### Idea di fondo



### Approccio corretto



### Struttura



- **Applicabilità**

- Si applica quando una azione può essere scomposta in due ambiti, ciascuno dei quali encapsulato in oggetti separati per mantenere basso il livello di coupling.
- Gestire le modifiche di oggetti conseguenti alla variazione dello stato di un oggetto.

- **Partecipanti**

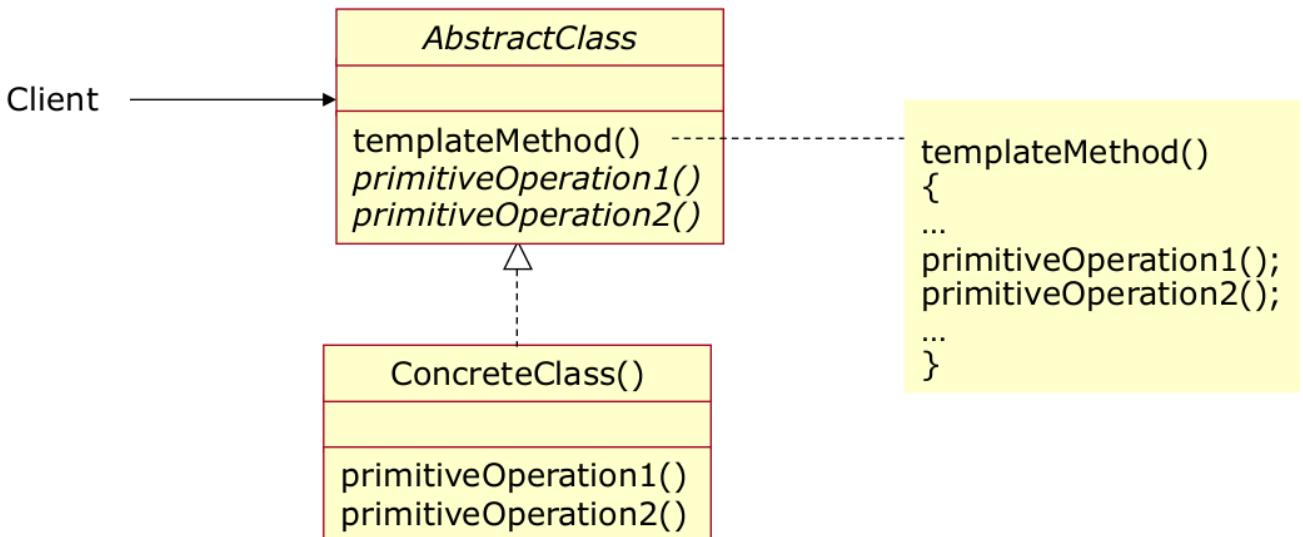
- **Subject** e **ConcreteSubject**
- **Observer** e **ConcreteObserver**

- **Conseguenze**

- L'accoppiamento tra **Subject** ed **Observer** è astratto
  - il **Subject** conosce solo la lista degli osservatori
- La notifica è una comunicazione di tipo broadcast
  - il **Subject** non si occupa di quanti sono gli **Observer** registrati
- Attenzione perché una modifica al **Subject** scatena una serie di modifiche su tutti gli osservatori e su tutti gli oggetti da questi dipendenti

## Template Method

- **Scopo:** Definire la struttura di un algoritmo all'interno di un metodo, delegando alcuni passi alle sottoclassi
  - | Le sottoclassi ridefiniscono solo alcuni passi dell'algoritmo ma non la sua struttura
- **Motivazione:** consideriamo un framework per costruire applicazioni in grado di gestire documenti diversi. Il Template Method definisce un algoritmo in base ad operazioni astratte che saranno definite nelle sottoclassi specifiche.



- **Applicabilità**

- E' utilizzato per implementare la parte invariante di un algoritmo, lasciando alle sottoclassi la definizione degli step variabili
- E' utile quando ci sono comportamenti comuni che possono essere inseriti nel template

- **Partecipanti**

- AbstractClass e ConcreteClass
- Client

- **Conseguenze**

- I metodi template permettono il riuso del codice
- Creano una struttura di controllo invertito dove è la classe padre che chiama le operazioni ridefinite nei figli e non viceversa
- Per controllare l'estendibilità delle sottoclassi, i metodi richiamati dal template sono chiamati metodi **gancio** (hook)
- I metodi hook possono essere implementati, offrendo un comportamento standard, che la sottoclasse può volendo ridefinire

Il Templat Method è simile al Factory Method

Indirizzo però problemi diversi

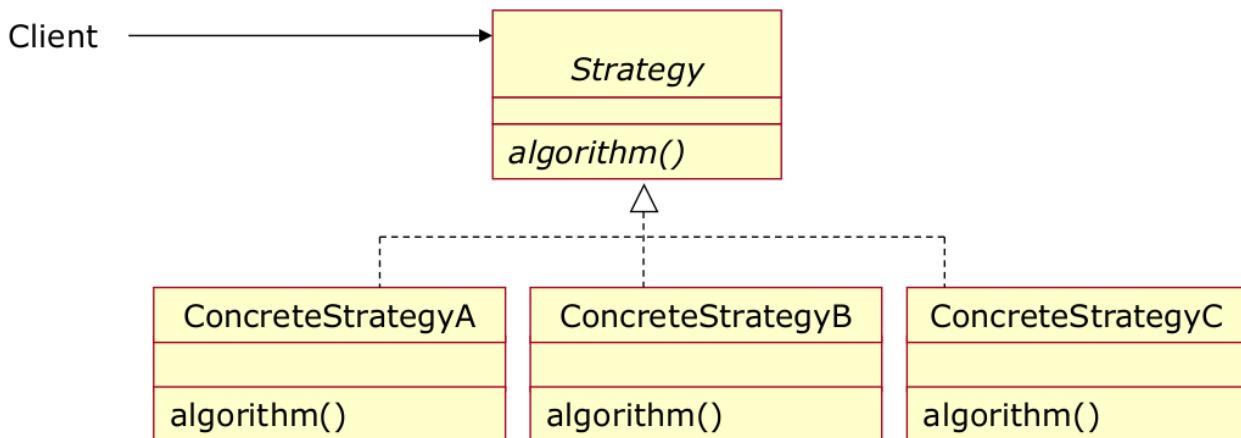
- Il Template Method è il metodo che invoca i metodi astratti, al fine di **generalizzare un algoritmo**
- Il Factory Method è un metodo astratto che deve creare e restituire l'istanza di classe concreta, al fine di **sganciare il cliente dalla scelta del tipo specifico**

## Strategy

- **Scopo:** Definire ed encapsulare una famiglia di algoritmi in modo da renderli intercambiabili indipendentemente dal client che li usa.

- **Motivazione:** Consideriamo la famiglia degli algoritmi di ordinamento. Ne esistono diversi (QuickSort, BubbleSort, MergeSort, etc). Costruiamo una applicazione che li supporti tutti, che possa essere facilmente estendibile, e che permetta una scelta rapida del tipo di algoritmo

Classificazione : Comportamentale basato su oggetti



- **Applicabilità**

- Molte classi correlate differiscono solo per il comportamento
  - Il pattern fornisce un modo per avere una interfaccia comune.
- Sono necessarie più varianti di uno stesso algoritmo, a seconda dei tipi di dato in ingresso o delle condizioni operative.

- **Partecipanti**

- Strategy e ConcreteStrategy
- Client

- **Conseguenze:**

- Il pattern separa l'implementazione degli algoritmi dal contesto dell'applicazione
  - usare il subclassing della classe Client per aggiungere un algoritmo non sarebbe stata una buona scelta.
- Le diverse strategie eliminano i blocchi condizionali che sarebbero necessari inserendo tutti i diversi comportamenti in una unica classe
- Lo svantaggio principale è che i client devono conoscere le diverse strategie

## Singleton

- **Scopo :** Gestire la creazione di un'unica istanza di una classe
- **Motivazione :** Il pattern Singleton risolve il problema di garantire che una classe abbia una sola istanza globale e di fornire un punto di accesso controllato a essa. Un esempio concreto è un registro di configurazione di sistema: deve esserci una sola copia delle configurazioni condivisa tra tutte le parti del sistema per evitare incongruenze o comportamenti imprevedibili.

## Classificazione : Strutturale basato su classi

La struttura è esattamente quella che sta nel progetto

### • **Applicabilità**

- Quando è necessario garantire che ci sia una sola istanza di una classe, ad esempio:
  - Un logger.
  - Un registro di configurazione.
  - Un pool di connessioni.
- Quando è necessaria una gestione centralizzata di una risorsa condivisa.
- Quando l'istanza unica deve essere facilmente accessibile da più punti nel sistema

### • **Partecipanti** : Classe Singleton

### • **Conseguenze** :

- **Vantaggi:**
  - **Controllo:** Assicura l'esistenza di una singola istanza.
  - **Accesso globale:** Fornisce un punto di accesso centralizzato.
  - **Riduzione del carico:** Riduce la necessità di istanziare più oggetti quando una sola istanza è sufficiente.
- **Svantaggi:**
  - **Testing:** Difficile da testare poiché l'istanza è globale e non facilmente sostituibile.
  - **Dipendenze implicite:** Può introdurre accoppiamento eccessivo.
  - **Problemi di concorrenza:** Deve essere progettato per essere thread-safe nei sistemi concorrenti.

## Domanda 1 - Significato Include/Exclude negli Use Case

Il diagramma degli Use Case permette di rappresentare graficamente le interazioni tra attori e sistema, specificando quali funzionalità (**casi d'uso**) gli attori possono attivare.

Esaminiamo i concetti di include ed extend nel contesto dell'immagine fornita (esempio use case Arbitro, vedi sotto).

### Significato di "include" ed "extend"

#### • **Include**

- Indica che un caso d'uso **obbligatoriamente include** un altro caso d'uso per completare la sua funzionalità.
- È utilizzato quando una parte del comportamento di un caso d'uso è riutilizzabile da altri casi d'uso.

- **Relazione:** Linea tratteggiata con freccia verso il caso d'uso incluso, etichettata come `<<include>>`.
- **Esempio dall'immagine:**
  - Nel caso d'uso "Inserisci risultato incontro", possiamo immaginare che ci sia un `<<include>>` per un caso come "Verifica autenticazione", poiché il login è una precondizione obbligatoria per eseguire l'operazione.-

- **Extend**

- Indica che un caso d'uso può opzionalmente estendere un altro caso d'uso, aggiungendo funzionalità opzionali o alternative.
- È utilizzato per rappresentare comportamenti che si verificano solo in determinate condizioni.
- **Relazione:** Linea tratteggiata con freccia verso il caso d'uso principale, etichettata come `<<extend>>`.
- **Esempio dall'immagine:**
  - Il caso d'uso "Inserisci risultato incontro" potrebbe essere esteso da un caso come "Correggi risultato", che si attiva solo se l'arbitro commette un errore durante l'inserimento del risultato.

## Chi attiva chi?

- **Include:**

- È **obbligatorio**: il caso d'uso principale richiede l'esecuzione del caso d'uso incluso.
- **Esempio:** "Inserisci risultato incontro" include il caso "Verifica autenticazione". L'attivazione di "Inserisci risultato incontro" attiva necessariamente "Verifica autenticazione"

- **Extend:**

- È **opzionale**: l'attivazione del caso d'uso esteso dipende da una specifica condizione.
- **Esempio:** "Inserisci risultato incontro" può essere esteso da "Correggi risultato". Questo caso d'uso si attiva solo se l'arbitro commette un errore durante l'inserimento del risultato.

## Domanda 2

## Domanda 3 - Differenze tra requisiti funzionali/non funzionali

 **Requisiti SW (SW Requirements) >**

Descrizione dei servizi che un sistema SW deve fornire, insieme ai vincoli da rispettare sia in fase di sviluppo che durante la fase di operatività del SW.

Ci sono 2 tipi di requisiti SW, e sono :

- **Requisiti utente** : Descrizione in linguaggio 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 che il sistema SW deve fornire

I requisiti SW di sistema si dividono in 3 categorie principali, che sono :

- **Requisiti Funzionali**

- Descrivono le funzionalità del sistema SW, in termini di *servizi* che il sistema SW deve fornire, di come il sistema SW *reagisce* a specifici tipi di input e di come si *comporta* in situazioni particolari

- **Requisiti Non Funzionali**

- Descrivono le *proprietà* del sistema SW in relazione a determinati servizi o funzioni e possono anche essere relativi al processo
  - Caratteristiche di **efficienza, affidabilità, safety, etc...**
  - Caratteristiche del **processo di sviluppo**
  - Caratteristiche **esterne**

- **Requisiti di dominio**

- Requisiti derivati dal dominio applicativo del sistema SW piuttosto che da necessità dettate dagli utenti

Le principali differenze tra req. funzionali/non funzionali sono quindi che :

- I primi identificano tutte quelle operazioni che il sistema deve eseguire, ad esempio **"Arriva la richiesta di cancellazione di un account, il sistema deve essere in grado di eseguire tale operazione senza problemi"**
- Gli altri identificano tutte quelle funzionalità che il sistema deve avere e rispettare, ad esempio **"il sistema deve essere affidabile, sicuro e deve garantire la crittografia dei dati"**, oppure **"Il sistema deve garantire performance adeguate al numero di query che arrivano"**, etc..

Altri esempi possono essere :

### Req. Funzionali

- Il sistema Sw deve fornire un appropriato visualizzatore per i documenti memorizzati
- L'utente deve essere in grado di effettuare ricerche

### Req. Non Funzionali

- Il tempo di risposta del sistema all'inserimento della password deve essere inferiore ai 10 sec.
- I documenti di progetto devono essere conformi allo standard XYZ-ABC-12345

## Domanda 4 - Caratteristiche modello a spirale

### Modello del ciclo di vita del SW >

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

La scelta del modello dipende dalla natura dell'organizzazione, dalla maturità dell'organizzazione, da metodi e tecnologie usate e da eventuali vincoli dettati dal cliente.

Ci sono vari modelli, tra cui i più importanti sono :

- **Build & Fix** : Non è un vero modello, infatti si usa in assenza di uno degli altri modelli.
  - Qui il prodotto SW viene prima sviluppato e poi rilavorato fino a soddisfare le esigenze del cliente
- **Modello Waterfall** : Modello dove le operazioni vengono effettuate a "cascata", con approccio **sequenziale e lineare**.
  - Ogni fase deve essere completata prima di passare alla successiva
  - Ad ogni fine fase è associata la verifica, per poter passare alla successiva
  - Ideale per progetti i cui requisiti sono chiari e ben definiti dall'inizio.
- **Modello a Spirale** : Modello con approccio **iterativo** che combina elementi del modello Waterfall con attenzione particolare per l'analisi dei rischi
  - Il SW viene sviluppato attraverso iterazioni, ognuna delle quali affronta una parte del progetto.
  - Ideale per progetti complessi e ad alto rischio, in cui i requisiti possono cambiare durante il ciclo di vita

Le caratteristiche principali del modello a spirale sono :

- **Struttura iterativa**
  - Il progetto è suddiviso in cicli (spirali), ognuno dei quali comporta pianificazione, progettazione, implementazione e valutazione.
  - Ogni iterazione aggiunge valore al prodotto.
- **Fasi principali in ogni ciclo**
  - **Customer communication** (Comunicazione con il cliente): Interazione costante per raccogliere e aggiornare i requisiti.

- **Planning** (Pianificazione): Definizione degli obiettivi, stime di costo, tempi e gestione dei rischi.
- **Risk analysis** (Analisi dei rischi): Identificazione e gestione dei rischi associati al progetto.
- **Engineering** (Ingegnerizzazione): Progettazione e implementazione tecnica.
- **Construction & Release** (Costruzione e rilascio): Realizzazione e rilascio dei deliverable.
- **Customer evaluation** (Valutazione del cliente): Feedback e valutazione del prodotto per prepararsi al ciclo successivo.

#### • Gestione dei rischi

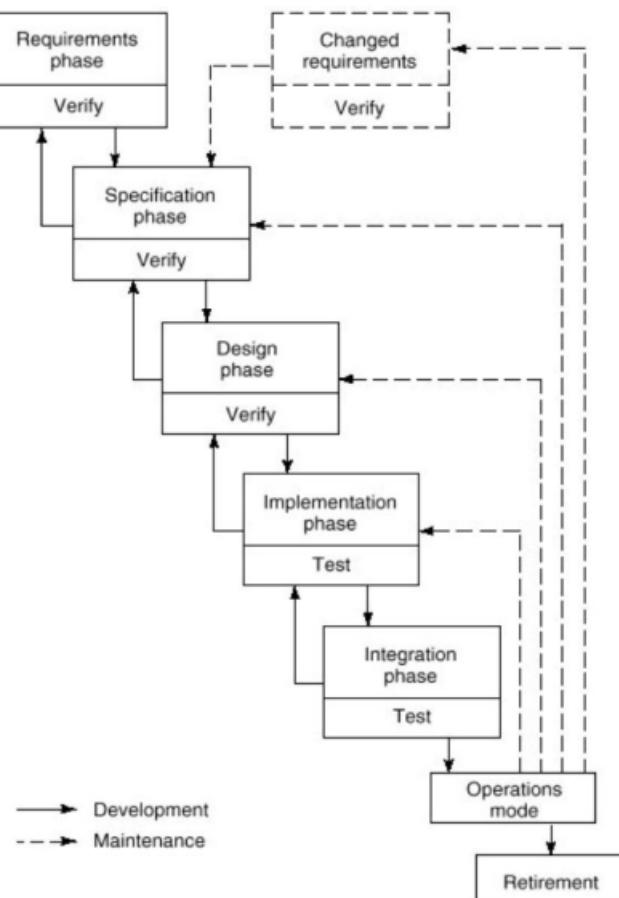
- L'identificazione e la mitigazione dei rischi sono centrali in questo modello.

#### • Incrementalità

- Ogni ciclo produce una versione incrementale del software, migliorando il prodotto fino alla versione finale.

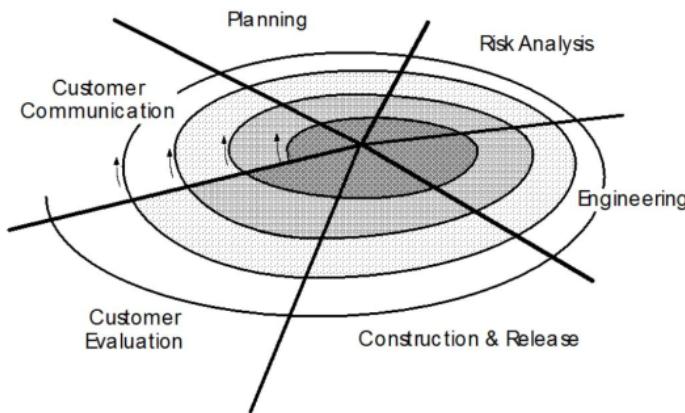
## Modello Waterfall

### • MODELLO WATERFALL:



## Modello a Spirale

### Modello a spirale:



## Domanda 4.1 - Cosa significa fare 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. Nel contesto dello sviluppo software (e in particolare nel modello a spirale), l'analisi dei rischi aiuta a prevenire problemi futuri, ottimizzare i costi e garantire che il progetto rimanga nei limiti previsti di tempo e qualità.

## Fasi principali

- **Identificazione dei rischi**

- Obiettivo: Trovare tutte le possibili minacce che potrebbero compromettere il progetto.
- Tipologie di rischi da considerare:
  - Tecnici: Incertezza nella fattibilità tecnica, utilizzo di nuove tecnologie o complessità del sistema.
  - Gestionali: Budget insufficiente, risorse non adeguate o scadenze non realistiche.
  - Operativi: Cambiamenti nei requisiti del cliente, modifiche alle priorità del progetto.
  - Esterni: Fattori esterni come problemi legali, economici o di mercato.
  - Di qualità: Il software potrebbe non soddisfare i requisiti di qualità stabiliti.
- Strumenti utilizzati: Brainstorming, checklist, interviste con esperti, studio di progetti simili.

- **Valutazione dei rischi**

- Obiettivo: Stimare l'impatto e la probabilità di ogni rischio.
- Due parametri principali:
  - **Probabilità (Likelihood)**: Quanto è probabile che il rischio si verifichi? (es. alta, media, bassa).

- **Impatto (Impact)**: Qual è l'effetto sul progetto se il rischio si verifica? (es. critico, moderato, lieve).
- Matrici di rischio: Tabelle che classificano i rischi combinando probabilità e impatto. Ad esempio :

Probabilità / Impatto	Basso	Medio	Alto
Basso	Verde	Verde	Giallo
Medio	Verde	Giallo	Rosso
Alto	Giallo	Rosso	Rosso

- **Prioritizzazione dei rischi**

- Obiettivo: Stabilire quali rischi devono essere affrontati immediatamente.
- Tecniche comuni:
  - Analisi Pareto (80/20): Concentrarsi sui pochi rischi che causano la maggior parte dei problemi.
  - Diagramma di Ishikawa: Visualizzare le cause principali dei rischi.

- **Pianificazione delle azioni di mitigazione**

- Obiettivo: Definire le azioni per ridurre la probabilità o l'impatto dei rischi.
- Azioni comuni:
  - **Evitare**: Cambiare il piano per eliminare il rischio (es. evitare l'uso di tecnologie inaffidabili).
  - **Ridurre**: Prendere misure per diminuire la probabilità o l'impatto (es. formazione del personale, aggiungere test approfonditi).
  - **Trasferire**: Delegare il rischio a terzi (es. contratti di outsourcing o assicurazioni).
  - **Accettare**: Decidere di non agire e gestire le conseguenze se il rischio si verifica (spesso per rischi minori).

- **Monitoraggio e revisione continua**

- Obiettivo: Assicurarsi che i rischi siano costantemente gestiti durante tutto il ciclo di vita del progetto.
- Strumenti utilizzati:
  - Revisioni periodiche dei rischi.
  - Aggiornamenti alle matrici di rischio.
  - Verifica delle azioni di mitigazione intraprese.

## Domanda 6 - Architettura Oggetti Distribuiti

Architetture software - Che cos'è e per cosa si usa l'architettura ad oggetti distribuiti?

L'architettura di sistema **definisce** la struttura dei **componenti** del sistema software, insieme alle **relazioni tra questi** componenti.

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.

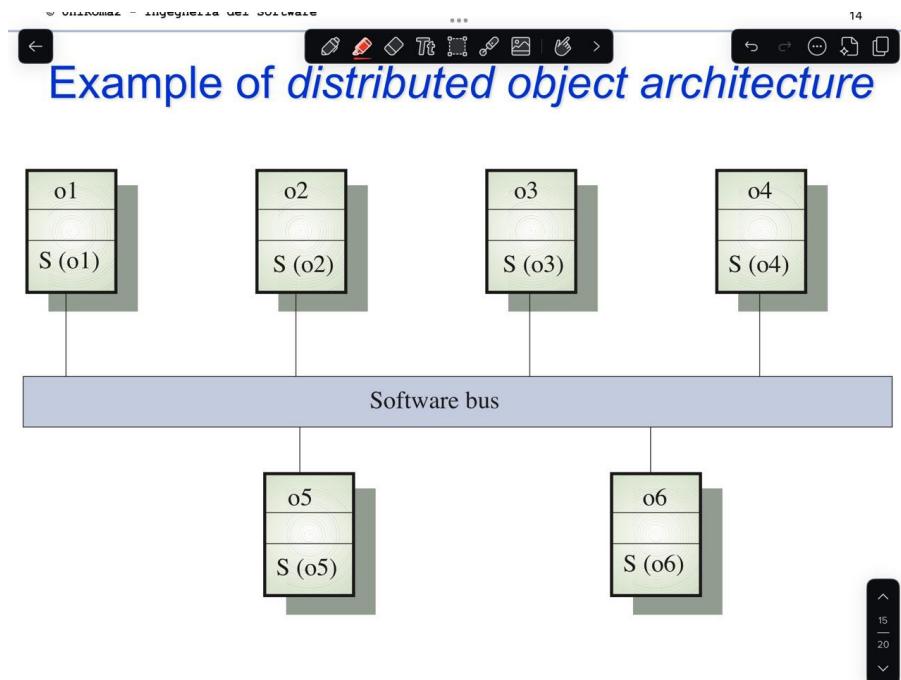
La comunicazione remota fra gli oggetti è resa **trasparente** usando *middleware* basati sul concetto di **software bus**.

Le applicazioni basate su questa architettura consistono in un **insieme di oggetti** che sono eseguiti su piattaforme **distribuite ed eterogenee**, e comunicano tramite invocazioni remote dei metodi

### Esempio 1



### Esempio 2



## Domanda 7 - ORB

[🔗 ORB \(Object Request Broker\) >](#)

Componente SW (*Middleware*) che permette la comunicazione tra oggetti distribuiti su una rete, indipendentemente dalla loro posizione fisica, dal linguaggio di programmazione utilizzato o dal S.O. su cui sono in esecuzione.

Funge quindi da **intermediario** per facilitare l'interazione tra oggetti in [un'architettura distribuita](#).

## Funzionamento

- **Intermediazione delle richieste:**

- L'ORB consente a un oggetto client (richiedente) di invocare metodi su un oggetto server (fornitore di servizi) che può risiedere su una macchina remota o locale.
- Il client non deve conoscere i dettagli sull'implementazione o sulla posizione fisica dell'oggetto server.

- **Astrazione:**

- Fornisce un'interfaccia standard che permette agli sviluppatori di concentrarsi sulla logica applicativa, senza preoccuparsi della complessità della comunicazione a basso livello.

- **Gestione della comunicazione:**

- L'ORB si occupa della serializzazione (marshalling) e deserializzazione (unmarshalling) dei dati trasmessi tra il client e il server.
- Gestisce i protocolli di rete necessari per il trasporto dei dati.

- **Trasparenza:**

- Fornisce trasparenza sia nella posizione (**location transparency**) che nell'implementazione (**implementation transparency**), garantendo che gli oggetti possano interagire senza preoccuparsi dei dettagli di dove o come sono implementati

Ci sono varie implementazioni per l'ORB, ma la più famosa è **CORBA** (Common Object Request Broker Architecture)

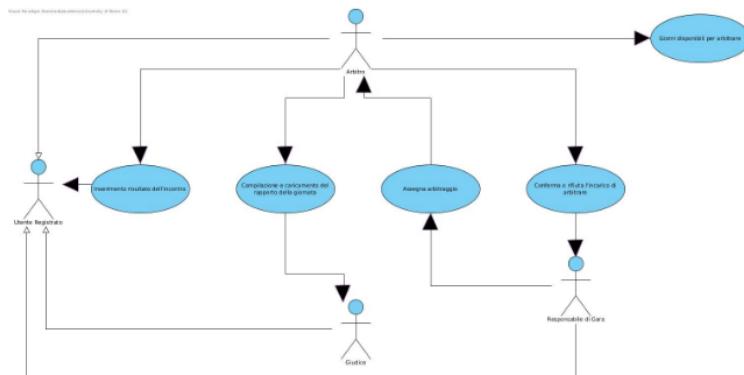
## Domanda 8 - Attivazione Use Case Progetto

Esempio Use Case

### 3 User Requirements Definition

#### 3.1 Use Case Arbitro

##### 3.1.1 Diagramma



##### 3.1.2 Documentazione

Use case	Inserisci risultato incontro	
Descrizione	Passo	Azione
	1	L'arbitro, dalla pagina dedicata, seleziona l'incontro di cui vuole inserire il risultato.
	2	Il sistema aggiorna risultato e classifica.
	3	Il sistema notifica il risultato agli utenti registrati che hanno scaricato l'applicazione e a quanti hanno accettato la ricezione di aggiornamenti tramite email e/o SMS dei partecipanti (pescatori) da loro selezionati nelle preferenze.
Attori	Arbitro, Utente Registrato	
Precondizioni	L'arbitro ha effettuato il login ed ha completato l'arbitraggio della partita della quale vuole inserire il risultato.	
Scenario principale	L'arbitro inserisce il risultato della partita.	
Scenari alternativi	Se l'arbitro sbaglia l'inserimento del risultato, cancella il risultato errato appena inserito, e lo reinserirà nello stesso modo.	
Post-condizioni	L'arbitro può continuare la compilazione del rapporto.	

L'attivazione di un caso d'uso avviene tramite l'interazione dell'utente con il sistema, generalmente attraverso un'interfaccia grafica.

Nel caso in questione:

- **Attore:** L'arbitro seleziona un'opzione specifica (es. "Inserisci risultato incontro") dalla pagina dedicata nel software.
- **Precondizioni:**
  - L'utente deve aver effettuato il login.
  - Deve aver completato l'arbitraggio della partita di cui vuole inserire il risultato.

#### Cosa succede quando un Use Case viene attivato?

Quando l'arbitro attiva un caso d'uso (es. "Inserisce risultato incontro"), il software esegue i seguenti passi:

1. **Input dell'utente:** L'arbitro seleziona il match dall'elenco delle partite disponibili.
2. **Elaborazione del sistema:**
  1. Il software aggiorna il risultato della partita e ricalcola la classifica.
  2. Aggiorna i dati interni relativi al match (es. stato, punteggi).
3. **Notifica agli utenti:**

1. Il sistema invia notifiche ai partecipanti e agli utenti registrati (tramite app, email, o SMS) per comunicare l'aggiornamento.

## Comportamento del SW

Il software, una volta attivato il caso d'uso:

- **Controlla le precondizioni:** Verifica che l'arbitro abbia effettuato il login e completato le fasi preliminari (es. arbitraggio).
- **Gestisce scenari alternativi:**
  - Se l'arbitro commette un errore nell'inserimento del risultato, il sistema permette di cancellare e reinserire i dati.
- **Post-condizioni:**
  - Dopo l'aggiornamento, il sistema rende disponibili i dati aggiornati per ulteriori elaborazioni o visualizzazioni.
  - L'arbitro può proseguire con altre operazioni (es. compilazione del rapporto di gara)

## Domanda 9 - Che relazione c'è tra prodotto e costo di produzione?

La relazione tra prodotto e costo di produzione nel contesto del software è complessa e si basa su diversi fattori:

- **Dimensione del prodotto:**
  - Maggiore è la complessità e la dimensione del software, maggiore sarà il costo di produzione.
- **Qualità del prodotto:**
  - Un software di alta qualità richiede più tempo e risorse per essere progettato, sviluppato e testato, aumentando i costi iniziali.
  - Tuttavia, un software di bassa qualità può comportare costi aggiuntivi per manutenzione e correzioni.
- **Efficienza dei processi di sviluppo:**
  - L'uso di metodologie di sviluppo efficienti (e.g., Agile, DevOps) può ridurre i costi mantenendo alta la qualità.
- **Automazione e strumenti:**
  - L'uso di strumenti di automazione può ridurre il costo di produzione migliorando l'efficienza.
- **Scalabilità del prodotto:**
  - Un software scalabile e modulare può ridurre i costi a lungo termine, facilitando aggiornamenti e manutenzione.

In sintesi, esiste un equilibrio tra il costo iniziale di produzione e i costi successivi legati alla manutenzione, supporto e aggiornamenti.

## Domanda 11 - Implementazione SOA

### ☒ SOA (Service Oriented Architecture) >

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

All'interno di SOA troviamo [l'ORB](#), che gestisce la comunicazione tra i client e i servizi offerti nel SOA.

Anche se le SOA sono concettualmente platform-indipendenti, attualmente vengono fornite con grande successo su piattaforme tecnologiche di **Web Services**.

Da un punto di vista SW, i Web Services sono le API (Application Programming Interface) che forniscono i metodi standard di comunicazione

Da un punto di vista del business, i Web Services sono funzionalità di business fornite da una compagnia nella forma di servizio esplicito in Internet.

I Web Services sono quindi il fulcro dell'implementazione 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 e HTML 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 le operazioni **CRUD** (Create Read Update Delete)

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 utilizzato per descrivere le interfacce dei servizi in un'architettura SOA
- Specifica cosa il servizio fa, come può essere invocato e dove si trova

## Domanda 12 - Secondo quali attività va declinata la fase di testing?

La fase di testing, nel ciclo di vita del software, è articolata in una serie di attività che assicurano che il prodotto soddisfi i requisiti definiti. Queste attività includono:

### 1. Pianificazione del testing:

- Definizione degli obiettivi del testing.
- Identificazione delle risorse necessarie (strumenti, personale).
- Pianificazione temporale e stima dei costi.

### 2. Progettazione dei casi di test:

- Creazione di scenari di test basati sui requisiti funzionali e non funzionali.
- Identificazione delle condizioni iniziali, dei dati di input e dei risultati attesi.

### 3. Preparazione dell'ambiente di test:

- Configurazione degli ambienti (hardware, software, reti).
- Installazione delle versioni del software da testare.

### 4. Esecuzione dei test:

- Esecuzione dei casi di test progettati.
- Registrazione dei risultati (successo o fallimento) e documentazione delle anomalie.

### 5. Analisi dei risultati:

- Confronto tra i risultati ottenuti e quelli attesi.
- Identificazione e classificazione dei difetti.

### 6. Risoluzione dei problemi:

- Collaborazione con il team di sviluppo per risolvere i difetti identificati.
- Verifica delle correzioni mediante retesting.

### 7. Test di regressione:

- Assicurarsi che le modifiche al codice non abbiano introdotto nuovi errori.

### 8. Valutazione finale e report:

- Verifica se il prodotto soddisfa i criteri di accettazione.
- Redazione di un report con le conclusioni del testing.

## Domanda 13 - Come si chiama il testing di validazione per i software a contratto?

Vedi pdf testi9ng pagina 5

Il testing di validazione per i software sviluppati a contratto è chiamato **Acceptance Testing** (o **User Acceptance Testing**, UAT).

- **Definizione:** È l'ultima fase del ciclo di testing, in cui il cliente o l'utente finale verifica che il software soddisfi i requisiti contrattuali e sia pronto per il rilascio.
- **Scopo:**
  - Validare che il software sia conforme alle specifiche del contratto.
  - Garantire che il prodotto sia utilizzabile nel contesto operativo del cliente.

## Domanda 14 - Adapter

L'**Adapter** è un design pattern strutturale che permette a classi o oggetti con interfacce incompatibili di lavorare insieme. L'obiettivo è quello di creare una classe che faccia da "adattatore", traducendo le richieste di una classe in un formato che l'altra classe possa comprendere.

L'Adapter si utilizza principalmente per:

1. **Integrare codice esistente** con nuove librerie o framework senza modificare il codice originale.
2. **Rendere compatibili** interfacce diverse, fungendo da "ponte" tra due classi o oggetti.
3. **Facilitare il riutilizzo del codice**, evitando di riscrivere componenti per farli funzionare insieme.

## Differenze tra Adapter su classi e oggetti

### Adapter su Classi

- Si realizza tramite **ereditarietà**.
- La classe Adapter eredita sia dall'interfaccia che dalla classe da adattare, implementando i metodi richiesti dall'interfaccia target.
- Questo approccio è possibile solo nei linguaggi che supportano **ereditarietà multipla** (come C++). In linguaggi come Java, si può simulare l'approccio con l'uso delle interfacce.

### Caratteristiche:

1. È **statico**, poiché utilizza la struttura delle classi a tempo di compilazione.
2. Ha una relazione forte con la classe adattata, in quanto l'ereditarietà vincola a quella classe.
3. Meno flessibile: non può essere utilizzato per adattare classi che non sono correlate tramite ereditarietà.

## Adapter su Oggetti

- Si realizza tramite **composizione**.
- L'Adapter contiene un riferimento a un'istanza della classe da adattare (Adaptee) e delega le chiamate ai metodi dell'Adaptee.
- È più flessibile rispetto all'Adapter su classi perché non dipende dall'ereditarietà.

### Caratteristiche:

1. È **dinamico**, poiché la relazione tra le classi viene stabilita a runtime.
2. Più flessibile: può essere utilizzato per adattare più classi, anche non correlate tra loro.
3. Può adattare più di una classe contemporaneamente, grazie alla composizione

## Quando usare Class Adapter o Object Adapter?

- Usa **Class Adapter** quando:
  - Hai bisogno di un'implementazione semplice e diretta.
  - Puoi usare l'ereditarietà multipla (e.g., in C++).
  - Sei sicuro che il tuo Adapter non debba adattare più classi o cambiare dinamicamente.
- Usa **Object Adapter** quando:
  - Vuoi maggiore flessibilità e indipendenza dall'Adaptee.
  - Devi adattare più classi diverse.
  - Stai lavorando con linguaggi che non supportano l'ereditarietà multipla (e.g., Java, Python).

## Domanda 16 - Ciclo di vita del SW

Definizione formale del ciclo di vita :

### ☞ Definizione ciclo di vita >

- Intervallo i tempo che intercorre tra l'istante in cui nasce l'esigenza di costruire un prodotto Sw e l'istante in cui il prodotto viene dismesso
- Include le fasi di definizione dei requisiti, specifica, pianificazione, progetto preliminare, progetto dettagliato, codifica, itergrazione, testing, uso, manutenzione e dismissione

Definizione informale :

### ☞ Ciclo di vita del SW >

Il ciclo di vita del SW è un processo strutturato utilizzato per progettare, sviluppare, testare, distribuire e mantenere un sistema software.

Garantisce che il software sia sviluppato in modo efficiente, rispettando i requisiti del cliente e mantenendo alta la qualità.

Il ciclo di vita del SW è suddiviso in 3 Stadi e 6 Fasi

### **Sviluppo (Stadio 1) = 6 fasi**

- Requisiti
- Specifiche (analisi dei requisiti)
- Pianificazione
- Progetto
- Codifica
- Integrazione

### **Manutenzione (Stadio 2)**

- Copre circa il 60% dei costi del ciclo di vita

### **Dismissione (Stadio 3)**

## **Dettaglio delle fasi**

### **1. Requisiti**

**Obiettivo:** Identificare e raccogliere le esigenze degli stakeholder.

**Descrizione:**

- In questa fase, vengono raccolte tutte le informazioni necessarie per capire cosa si aspetta il cliente dal software. Gli stakeholder (utenti, manager, tecnici) discutono le funzionalità richieste e i vincoli.

**Attività principali:**

- Interviste con gli utenti e i clienti.
- Analisi delle esigenze funzionali (cosa il sistema deve fare).
- Identificazione dei requisiti non funzionali (prestazioni, sicurezza, scalabilità).
- Creazione di una lista iniziale di requisiti.

### **2. Specifiche (Analisi dei requisiti)**

**Obiettivo:** Documentare e dettagliare i requisiti raccolti.

**Descrizione:**

- Questa fase traduce i requisiti generali in specifiche chiare e dettagliate. I requisiti sono formalizzati in documenti che serviranno come base per le fasi successive.

**Attività principali:**

- Analisi di fattibilità tecnica ed economica.
- Stesura di un documento di specifiche dei requisiti software (**Software Requirements Specification, SRS**).
- Validazione dei requisiti con il cliente per assicurarsi che siano completi e corretti.

## 3. Pianificazione

**Obiettivo:** Organizzare il lavoro necessario per completare il progetto.

**Descrizione:**

- In questa fase si definiscono le tempistiche, le risorse, il budget e i rischi. Si crea un piano per guidare l'intero processo di sviluppo.

**Attività principali:**

- Creazione di una roadmap del progetto.
- Definizione delle milestone e dei deliverable.
- Assegnazione dei ruoli ai membri del team.
- Identificazione dei potenziali rischi e pianificazione delle strategie di mitigazione.

## 4. Progetto

**Obiettivo:** Progettare l'architettura e i dettagli tecnici del software.

**Descrizione:**

- Qui si definisce la struttura del sistema e le sue componenti, fornendo un modello chiaro da seguire nella fase di sviluppo.

**Attività principali:**

- Progettazione dell'architettura (es.: modelli client-server o microservizi).
- Creazione di diagrammi UML, diagrammi di flusso e design delle interfacce.
- Progettazione di database e definizione delle entità.
- Redazione del piano di test iniziale per assicurarsi che il progetto sia verificabile.

## 5. Codifica

**Obiettivo:** Tradurre il design in un software funzionante.

**Descrizione:**

- I programmatore implementano il sistema scrivendo il codice sorgente basato sul progetto. Il lavoro è suddiviso in moduli o componenti.

**Attività principali:**

- Scrittura del codice seguendo standard e best practice.
- Utilizzo di strumenti di controllo versione per gestire il codice (es.: Git).
- Implementazione di test unitari per verificare il funzionamento dei singoli moduli.

## 6. Integrazione

**Obiettivo:** Collegare i diversi moduli e testare il sistema completo.

**Descrizione:**

- In questa fase, i singoli moduli sviluppati vengono combinati per creare il sistema completo. Si eseguono test approfonditi per verificare il corretto funzionamento dell'intero software.

**Attività principali:**

- Integrazione di moduli e componenti.
- Esecuzione di test di integrazione per garantire che i moduli lavorino insieme senza errori.
- Correzione dei bug emersi durante i test.
- Preparazione per il rilascio o la distribuzione finale.

## Domanda 17 - Qual è la manutenzione più frequentemente utilizzata?

La manutenzione più frequentemente utilizzata è la **manutenzione correttiva**.

- **Definizione:** Si tratta dell'attività necessaria per correggere errori o difetti scoperti nel software dopo il rilascio. Questi difetti possono riguardare bug nel codice o problemi legati a funzionalità non implementate correttamente.
- **Altri tipi di manutenzione:**
  - **Manutenzione adattiva:** Modifica il software per adattarlo a cambiamenti nell'ambiente operativo (e.g., nuove versioni di sistemi operativi o hardware).
  - **Manutenzione perfettiva:** Migliora le prestazioni o aggiunge nuove funzionalità per soddisfare le richieste degli utenti.
  - **Manutenzione preventiva:** Effettua modifiche per prevenire futuri problemi o ridurre il rischio di guasti.

## Domanda 18 - Regola 10-90

Esperimenti condotti su programmi di notevoli dimensione mostrano che :

### **Regola 10-90** >

Il 90% del tempo di esecuzione totale è speso eseguendo solo il 10% delle istruzioni

Tale 10% viene chiamato **core** (nucleo) del programma

## Domanda 19 - Stima durata progetto

Per stimare la durata di un progetto software partendo dal documento di specifica, è necessario utilizzare tecniche di pianificazione e stima che si basano su diversi fattori come la dimensione stimata del prodotto, le risorse disponibili e i processi di sviluppo.

La prima fase della stima è quella di stimare la **dimensione** del prodotto SW, usando le tecniche [viste qua](#)

Dopo aver stimato la dimensione, possiamo usare sostanzialmente due approcci per stimare la durata :

1. Usando il modello algoritmico [COCOMO](#)
  1. Stimando prima Effort e poi Tempo, vedi sopra
2. Usando metodi empirici (Bottom-UP)
  1. Suddividendo il progetto in attività specifiche e stimare la durata di ogni attività
  2. Sommare tutte le durate per ottenere una stima complessiva

## Domanda 20 - Organizzazione modello di qualità del SW

**Com'è organizzato il modello di qualità del software a livello di standard utilizzati?**

Il modello di qualità del software si basa principalmente sullo **standard 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.

La definizione dello **standard IEEE 1061** è questo :

### **IEEE 1061 - Software Quality Metrics Methodology** >

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

## 1. Struttura del modello secondo lo standard IEEE 1061

Lo standard IEEE 1061 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

Un concetto centrale dello standard è l'utilizzo di **indici di qualità** che aggregano metriche per fornire un quadro d'insieme delle prestazioni del software rispetto a determinati obiettivi di qualità.

Gli indici sono suddivisi in 3 gruppi principali, che rispettano le attività nel modello [McCall](#)

- **Attività di revisione :**

- *Manutenibilità* :
  - effort necessario per individuare e correggere un errore in un programma operativo.
- *Flessibilità* :
  - effort necessario per modificare un prodotto operativo
- *Testabilità* :
  - effort necessario per testare un prodotto al fine di garantire che svolga la funzione prevista

- **Attività di Transizione :**

- *Portabilità* :
  - effort necessario per trasferire un prodotto da un ambiente hardware e/o software ad un altro
- *Riusabilità* :
  - misura in cui un prodotto (o parti di esso) può essere riutilizzato in altre applicazioni
- *Interoperabilità* :
  - effort necessario per accoppiare un prodotto con un altro
- *Evolutibilità* :
  - effort richiesto per aggiornare il prodotto al fine di soddisfare nuovi requisiti

- **Attività di Operazioni :**

- *Correttezza* :
  - misura in cui un prodotto soddisfa le specifiche e risponde agli obiettivi degli utenti
- *Affidabilità* :
  - in che misura ci si può aspettare che un prodotto svolga la funzione prevista con la precisione richiesta

- *Efficienza* :
  - quantità di risorse di calcolo e di codice necessarie a un prodotto per eseguire una funzione
- *Integrità* :
  - misura in cui l'accesso al software o ai dati da parte di persone non autorizzate persone non autorizzate
- *Usabilità* :
  - sforzo necessario per apprendere, utilizzare, preparare l'input e interpretare l'output di un prodotto.

### **3. Attributi di qualità**

Gli attributi di qualità definiti nello standard sono categorie specifiche che descrivono le caratteristiche essenziali per valutare la qualità di un sistema software. Gli attributi comuni includono:

- **Complessità** : livello di comprensibilità e verificabilità degli elementi del software e della loro software e delle loro interazioni.
- **Precisione** : precisione dei calcoli e dei risultati
- **Completezza** : completa implementazione delle funzionalità richieste
- **Coerenza** : utilizzo di tecniche di progettazione e tecniche di implementazione e notazioni uniformi
- **Tolleranza agli errori** : continuità di funzionamento garantita in condizioni avverse condizioni avverse
- **Tracciabilità** : grado in cui una relazione può essere relazione tra due o più prodotti prodotti del processo di sviluppo processo di sviluppo
- **Espandibilità** : le memorie o le funzioni possono essere espanso
- **Generalità** : ampiezza delle potenziali applicazioni
- **Modularità** : disposizioni di moduli altamente indipendenti
- **Autodocumentazione** : documenti in linea

Questi attributi sono correlati alle metriche che li misurano e agli obiettivi di qualità del progetto.

### **4. Processo di valutazione della qualità**

Lo standard IEEE 1061 descrive un processo chiaro e iterativo per valutare la qualità del software:

1. **Definizione degli obiettivi di qualità:** Gli obiettivi dipendono dal contesto applicativo del software (es. sicurezza per applicazioni critiche, usabilità per software consumer, ecc.).
2. **Identificazione degli attributi:** In base agli obiettivi di qualità, vengono scelti gli attributi da misurare.

3. **Selezione delle metriche:** Per ogni attributo vengono identificate metriche che lo rappresentano in modo quantitativo.
4. **Misurazione e valutazione:** Le metriche vengono calcolate e confrontate con valori attesi o accettabili.
5. **Feedback e miglioramento:** I risultati dell'analisi guidano il miglioramento 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 monitorando 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 IEEE per la preparazione di un piano SQA sono :

- Gestione
- Documentazione
- Standard, pratiche e convenzioni
- Revisioni e audit
- Gestione della configurazione del software
- Segnalazione dei problemi e azioni correttive
- Strumenti, tecniche e metodologie
- Controllo del codice
- Controllo dei supporti
- Controllo dei fornitori
- Raccolta, manutenzione e conservazione dei documenti

## Domanda 21 - Factory Method a cosa serve

Lo scopo dell'utilizzo del Desing Pattern **Factory Method** è il seguente :

Definire una interfaccia per la creazione di un oggetto, che consenta di decidere a tempo di esecuzione quale specifico oggetto istanziare.

Quali sono le situazioni in cui si applica il Desing Pattern Factory Method?

Ce ne sono varie, per esempio :

- Una classe vuole che le sue sottoclassi scelgano gli oggetti da creare

- Quando una classe non è in grado di sapere in anticipo le classi di oggetti che deve creare
- etc...

Quali sono le conseguenze dell'applicazione di questo DS?

- L'utilizzo di questo DS **elimina** la necessità di riferirsi a classi dipendenti dall'applicazione all'interno del codice.

## Domanda 22 - Motivazioni dietro Abstract Factory

Quali sono le motivazioni dietro l'utilizzo del DS Abstract Factory?

l'Abstract Factory è un pattern potente per mantenere il codice scalabile, configurabile e indipendente, offrendo una soluzione elegante per gestire famiglie di prodotti correlati in modo coerente e flessibile.

Per esempio, lo usiamo quando vogliamo realizzare strumenti per lo sviluppo di User Interface in grado di supportare diversi tipi di *look & feel*. Per garantire la portabilità di una applicazione tra look & feel diversi, gli oggetti non devono essere cablati nel codice.

Il design pattern **Abstract Factory** trova la sua applicabilità in contesti in cui è necessario garantire l'indipendenza del sistema dalle modalità di creazione dei prodotti con cui opera. Questo pattern si rivela particolarmente utile quando il sistema deve poter essere configurato per utilizzare **famiglie di prodotti diversi**, mantenendo una coerenza nell'interfaccia e senza richiedere modifiche al codice client.

Un aspetto fondamentale dell'Abstract Factory è che il **client non deve essere legato** a una specifica famiglia di prodotti. Questo significa che il codice del client può funzionare con qualsiasi famiglia di oggetti fornita, a patto che essa aderisca all'interfaccia definita dalla factory astratta. In questo modo, si ottiene un sistema flessibile, in grado di evolversi nel tempo con nuove famiglie di prodotti senza richiedere cambiamenti strutturali nel codice esistente.

## Domanda 23 - Diagramma UML per interazione tra oggetti

Per specificare l'interazione tra oggetti, usando la struttura dei diagrammi UML, possiamo usare il così detto **sequence diagram**.

L'altro diagramma UML che permette di fare questa cosa si chiama **Collaboration Diagram**. Questo diagramma è una rappresentazione *equivalente* del sequence diagram, infatti a partire da uno possiamo generare l'altro e viceversa.

Il funzionamento del Collaboration diagram è identico a quello del sequence diagram, le uniche cose che cambiano sono che :

- Il sequence descrive lo scambio di messaggi tra oggetti in ordine temporale, mentre **il collaboration descrive lo scambio di messaggi tra oggetti mediante relazioni**
- I sequence vengono usati in fase di **OOA**, mentre i collaboration in fase di **OOD**

## Domanda 24 - COCOMO

### Ξ COCOMO (COnstructive COst MOdel) >

Modello algoritmico introdotto da Boehm per determinare il valore dell'effort in base alla dimensione del progetto.

Il valore dell'**effort** viene successivamente utilizzato per determinare **durata e costi** di sviluppo

COCOMO comprende 3 **modelli** :

- **Basic** : Per stime iniziali
- **Intermediate** : Usato dopo aver suddiviso il sistema in sottosistemi
- **Advanced** Usato dopo aver suddiviso in moduli ciascun sottosistema

La stima dell'effort viene effettuata a partire da :

- Stima delle dimensioni del progetto in **KLOC** (*Kilo Lines of Code*)
- Stima del modo di sviluppo del prodotto, che misura il livello intrinseco di difficoltà nello sviluppo, tra (anche detti modi/categorie):
  - **Organic** : per prodotti di piccole dimensioni
  - **Semi-detached** : per prodotti di dimensioni intermedie
  - **Embedded** : per prodotti complessi

## Formula di COCOMO

Formula base di COCOMO

**Stima dell'effort nominale :**

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

$a, b$  sono coefficienti specifici per ciascuna categoria di progetto

**Stima effort :**

$$Effort = EffortNom \cdot C$$

Dove  $C$  è il costo driver multiplier, vedi sotto

## Stima del tempo

$$Time = c \cdot (Effort)^d$$

$c, d$  come  $a, b$

## Esempio

**MM** = Uomo/Mesi, quante persone per mese devono lavorare al progetto

Modello *Intermediate*, modo *organic*

- Passo 1 : Determinare l'**effort nominale** con la formula :

$$3.2 \cdot (KLOC)^{1.05} MM$$

Esempio :

$$3.2 \cdot (33)^{1.05} = 126 MM$$

- Passo 2 : Ottenere la stima dell'effort applicando un fattore moltiplicativo C basato su **15 cost drivers** (spolier non li metterò) :

$$effort = effort nominale \cdot C$$

Esempio

$$126 \cdot 1.15 = 145 MM$$

- Passo 3 : Stima del tempo alla consegna, usando la formula

$$T = 2.5 \cdot E^{0.38} M$$

Esempio

$$T = 2.5 \cdot 145^{0.38} \sim 16.56 M$$

### ⇒ **C (Cost driver multiplier)** >

Si ottiene come la *produttoria* dei cost driver  $c_i$ .

Ogni  $c_i$  determina la complessità del fattore  $i$  che influenza il progetto e può assumere uno tra più valori assegnati con variazioni intorno al valore unitario (VEDITE STE CAZZO DE SLIDE CHE NSE CAPISCE NA SEGA)

In generale, il time schedule di COCOMO è questo :

- Modo *organic* :  $T = 2.5 \cdot E^{0.38}$  (mesi M)
- Modo *semi-detached* :  $T = 2.5 \cdot E^{0.35}$  (mesi M)
- Modo *embedded* :  $T = 2.5 \cdot E^{0.32}$  (mesi M)

# Domanda 25 - Caratteristiche del modello Microsoft

La **Microsoft** ha dovuto affrontare problemi di :

- **incremento della qualità** dei prodotti SW
- **riduzione di tempi e costi** di sviluppo

Per risolvere questi problemi si è adottato un processo che allo stesso tempo è **iterativo, incrementale** e **concorrente**, e che permette di esaltare le doti di creatività delle persone coinvolte nello sviluppo di prodotti SW.

L'approccio usato attualmente da Microsoft è noto come **synchronize-and-stabilize**

Con questo modello di Microsoft, otteniamo un ciclo di sviluppo a 3 fasi

## Planning

- Definisce la visione del prodotto, le specifiche e la pianificazione

## Development

- Sviluppo di funzionalità in 3 – 4 sottoprogetti sequenziali, ognuno dei quali si traduce in un rilascio milestone.

## Stabilizzazione

- Test interni ed esterno completi, prodotto finale, stabilizzazione e spedizione.

Le **caratteristiche principali** di questo modello sono :

- Sviluppo Sw e testing eseguiti in parallelo
- Features prioritizzate e integrate in 3 – 4 sottoprogetti cardine
- Sincronizzazioni frequenti e stabilizzazioni intermedie
- Continuo feedback dei customer durante il processo di sviluppo
- Progettazione di prodotto e processi in modo che teams grandi possano lavorare come teams piccoli

# Domanda 26 - Differenze tra testing black/white box

Il **testing black-box** e il **testing white-box** sono due approcci fondamentali per testare software, ma si differenziano per il loro focus e il livello di conoscenza del sistema da parte del tester

## Black-Box testing

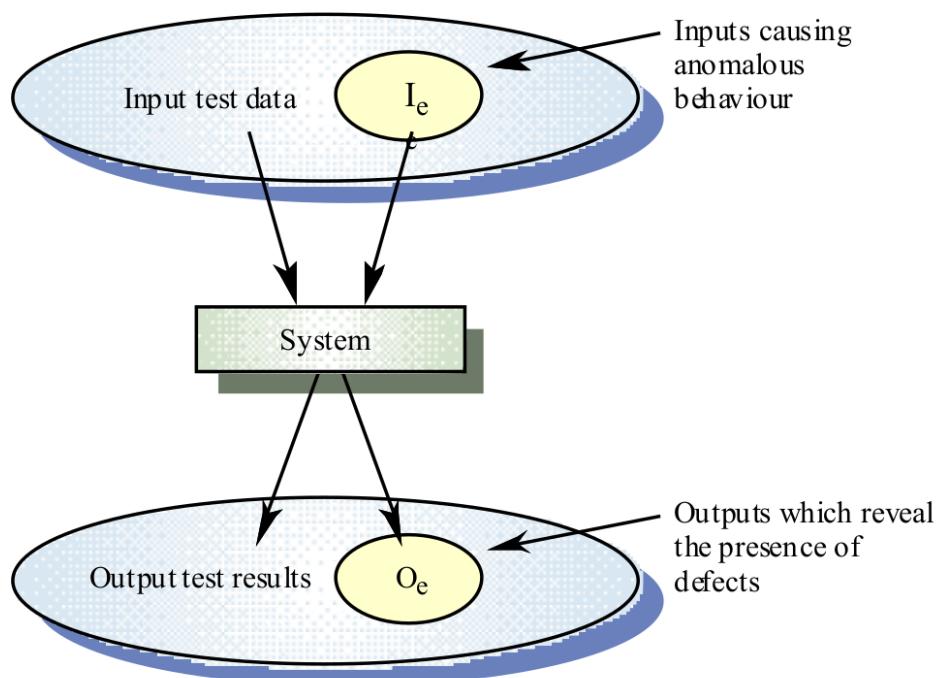
Approccio al testing dove il programma viene considerato come una **scatola nera (black-box)**

I test case vengono derivati dalle specifiche del software.

Il tester presenta gli input al componente o al sistema ed esamina l'output corrispondente.

Se gli output non sono quelli specificati, allora il test ha rilevato *con successo* un problema con il software.

Questo tipo di testing viene spesso chiamato **testing funzionale**, perchè il tester si concentra solo sulle funzionalità del SW, e non considera per niente l'implementazione interna del SW stesso.



## Equivalence Partitioning

Tecnica di testing black-box.

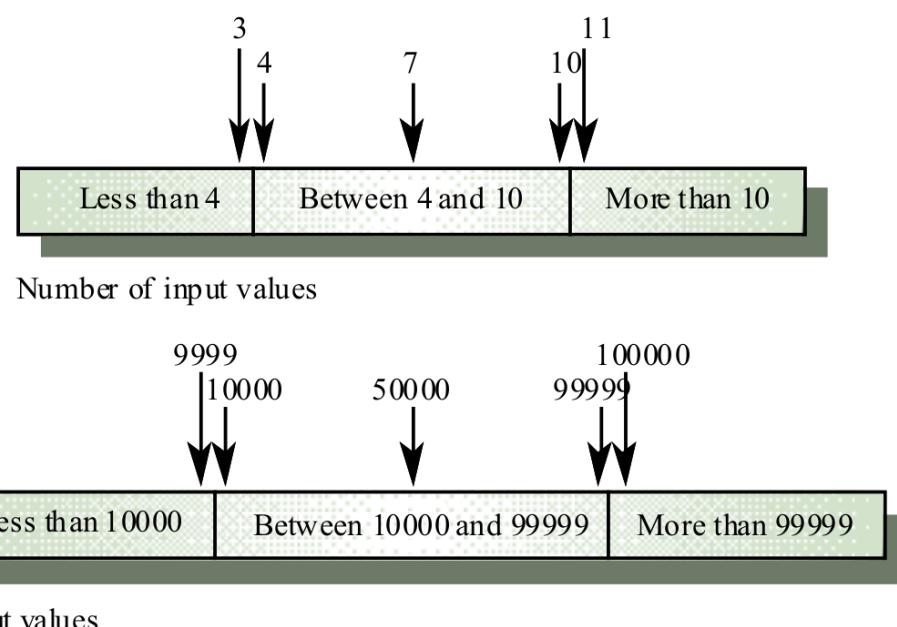
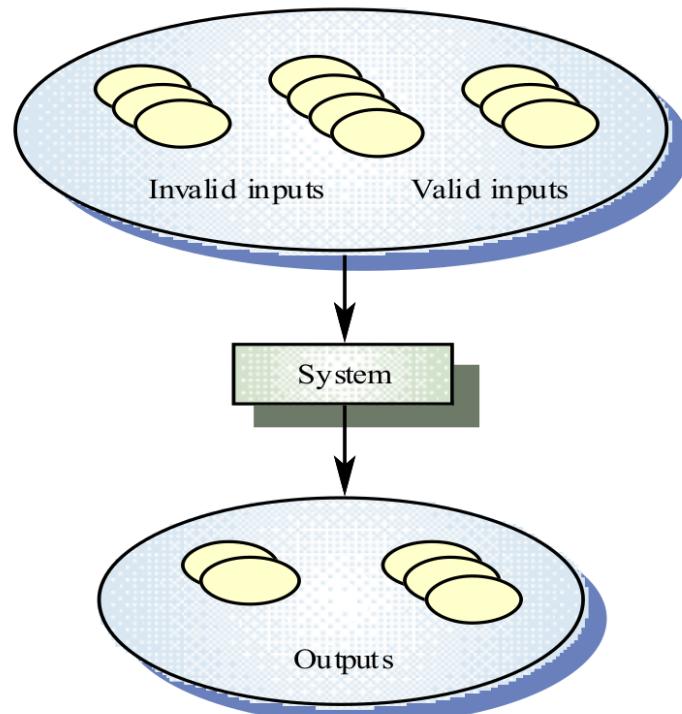
I dati di input e i risultati di output spesso rientrano in diverse classi, dove tutti i membri di una classe sono correlati

Ognuna di queste classi è una *partizione equivalente*, in cui il programma si comporta in modo equivalente per ogni membro della classe.

I test cases dovrebbero essere scelti da ogni partizione

Come funziona?

- Si partizionano gli input/output del sistema in *partizioni equivalenti*
- Si scelgono i test cases al confine di queste partizioni più i casi vicini al punto medio delle partizioni con input validi.



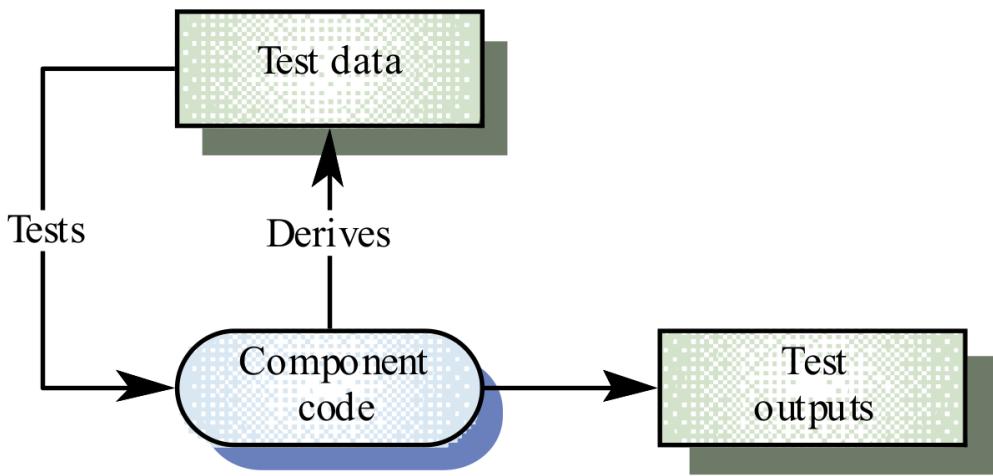
## White-Box testing

Viene spesso chiamato **testing strutturale**

I test case sono derivati dalla *struttura del programma*

La conoscenza del programma viene usata per identificare ulteriori test case.

Obiettivo di questo tipo di testing è quello di far lavorare ogni istruzione di codice all'interno del programma



## Path testing

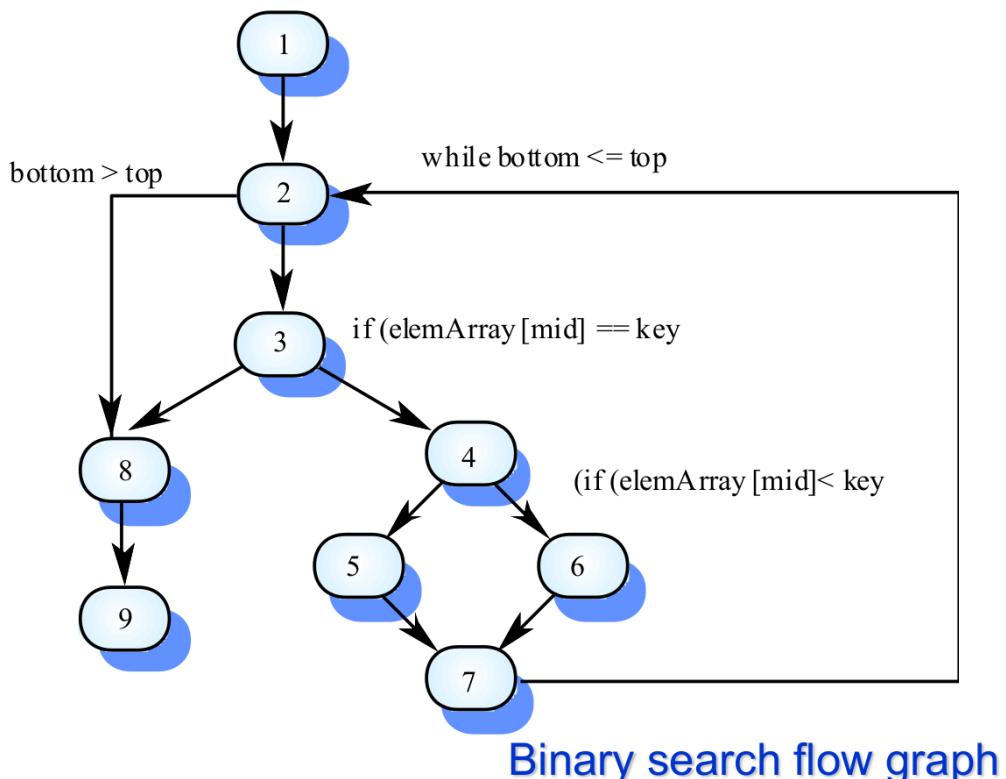
Tecnica di testing white-box

L'obiettivo del path testing è quello di assicurarsi che l'insieme dei test case sia tale che ogni path attraverso il programma sia eseguito almeno una volta.

Il numero di path attraverso il programma è di solito proporzionale alla sua dimensione.

Man mano che i moduli vengono integrati nei sistemi, diventa impraticabile l'utilizzo di tecniche di path testing che vengono, di conseguenza, utilizzate per lo più nelle fasi di test delle unità o dei moduli.

Il punto di partenza per il path testing è il flowgraph del programma che mostra nodi che rappresentano le decisioni del programma e archi che rappresentano il flusso di controllo.



## Differenze tra white/black box testing

Caratteristica	Black-Box	White-Box
<b>Conoscenza del codice</b>	Non necessaria	Necessaria
<b>Focus</b>	Funzionalità e output	Implementazione e logica interna
<b>Tipi di test</b>	Test funzionali, test di sistema	Test unitari, analisi del flusso di codice
<b>Tester</b>	Non deve conoscere la struttura del codice	Deve conoscere la struttura del codice
<b>Obiettivo principale</b>	Verificare i requisiti	Verificare la correttezza interna

In sintesi, il **testing black-box** valuta il "cosa" il software fa, mentre il **testing white-box** si concentra sul "come" lo fa.

## Domanda 27 - Affidabilità SW

Parlami dell'affidabilità del software

L'affidabilità del software in modo informale la possiamo definire come la **credibilità** del software, mentre in modo formale come la

### Ξ Affidabilità >

Probabilità che il prodotto software lavori "correttamente" in un determinato intervallo temporale.

Prima di continuare, diamo le definizioni di

#### ⓘ Info

- Difetto (defect) : Anomalia presente in un prodotto SW
- Guasto (failure) : Comportamento anomalo del prodotto SW dovuto alla presenza di un Difetto
- Errore : Azione errata di chi introduce un difetto nel prodotto SW

## Affidabilità

Intuitivamente, un prodotto SW con molti difetti è poco affidabile (GRAZIE AR CAZZO)

È chiaro che l'affidabilità del prodotto **migliora** via via che si riduce il numero di difetti

# Caratteristiche dell'affidabilità SW

È una relazione non semplice tra :

- affidabilità osservata
- numero di difetti latenti

(1) Eliminare difetti dalle parti del prodotto raramente usate ha piccoli effetti sull'affidabilità osservata.

(2) Il miglioramento dell'affidabilità per l'eliminazione di un difetto dipende dalla localizzazione del difetto (ovvero se appartiene o meno al nucleo del programma [vedi regola 10-90](#)

(3) Quindi, l'affidabilità osservata dipende da come è usato il prodotto, ovvero in termini tecnici, dal suo profilo operativo (**operational profile**).

(4) Dunque, dato che utenti differenti usano il software secondo profili operativi diversi, si ha che i difetti che si manifestano per un utente potrebbero non manifestarsi per l'altro (ALTRO GRAZIE AR CAZZO)

Quindi, possiamo affermare che l'affidabilità di un prodotto SW **dipende dall'utente**

## Domanda 28 - Specifica requisito di affidabilità

I **requisiti di affidabilità** fanno parte dei **requisiti non funzionali**, e di conseguenza rientrano anche nei **requisiti di sistema**. I requisiti non funzionali definiscono caratteristiche qualitative del sistema, e l'affidabilità è una di queste.

## Come specificare un requisito di affidabilità

Per definire un requisito di affidabilità in modo chiaro e verificabile, è fondamentale utilizzare metriche misurabili e standard riconosciuti.

### Utilizzare metriche comuni di affidabilità

Esempi di metriche per esprimere un requisito di affidabilità includono:

- **MTBF (Mean Time Between Failures):**
  - Indica il tempo medio tra un guasto e l'altro.
  - Esempio: "Il sistema deve avere un MTBF di almeno 1.000 ore."
- **MTTR (Mean Time To Repair):**
  - Indica il tempo medio necessario per ripristinare il sistema dopo un guasto.
  - Esempio: "Il sistema deve essere riparabile entro un MTTR di massimo 2 ore."
- **Tasso di disponibilità (Availability):**
  - Indica la percentuale di tempo in cui il sistema è operativo rispetto al tempo totale.

- Formula: Availability =  $\frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$
- Esempio: "Il sistema deve garantire un tasso di disponibilità del 99,95%."
- **Frequenza dei guasti:**
  - Specifica il numero massimo di guasti accettabili in un dato periodo.
  - Esempio: "Non più di 2 guasti critici per anno di utilizzo."

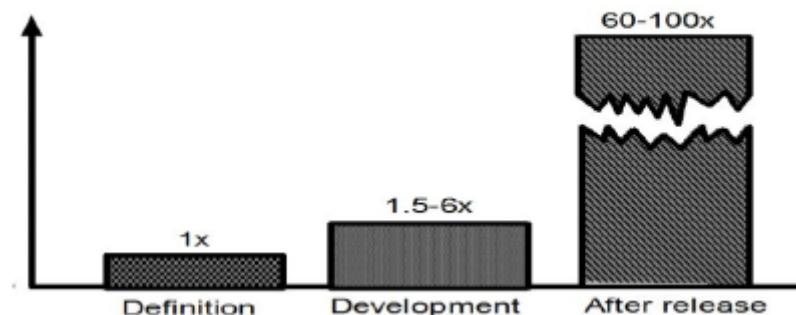
## Domanda 29 - Impatto sui costi

L'effetto delle modifiche varia molto a seconda della fase in cui vengono introdotte.

In fasi avanzate, una modifica può comportare rivolgimenti che richiedono nuove risorse o correzioni importanti al progetto, cioè costi supplementari.

Man mano che si avanza nello sviluppo, l'impatto delle modifiche aumenta significativamente

Come possiamo notare dall'immagine sottostante, l'impatto delle modifiche sui costi aumenta vertiginosamente, fino ad arrivare a creare interruzioni e blocchi del prodotto post rilascio.



I modi per mitigare l'impatto delle modifiche sono :

- **Investire in un'analisi dettagliata dei requisiti**
- Integrare strumenti di **version control**
- Effettuare un'adeguata pianificazione del rischio
- etc...

## Domanda 30 - Stima della dimensione del prodotto SW

Per stimare la dimensione del prodotto software a partire dal documento di specifica, vengono utilizzate diverse tecniche che analizzano i requisiti del sistema e cercano di quantificare la dimensione in termini di linee di codice (LOC), funzionalità, o altri parametri. Tra i metodi principali troviamo:

- **Analisi Bottom-Up (tecniche di scomposizione)** : Di cui fanno parte

- FPA (Function Points Analysis)
- LOC (Lines of Code)
- Modelli algoritmici empirici
- Stime su progetti simili già completati

Vediamo nel dettaglio le tecniche di scomposizione **LOC** e **FPA**

## Tecniche di scomposizione

Le tecniche di scomposizione utilizzano una strategia "*divide et impera*", e sono basate su :

- Stime dimensionali, come **LOC** o **FP**
- Suddivisione dei task e/o delle funzioni con relativa stima di allocazione dell'effort

## LOC - Lines of Codes

Come dice la parola stessa, **LOC** *stima il numero complessivo di linee di codice all'interno dell'intero SW*

## Esempio di stima di LOC

Functions	estimated LOC	LOC/pm	\$/LOC	Cost	Effort (MM)
UICF	2340	315	14	32,000	7.4
2DGA	5380	220	20	107,000	24.4
3DGA	6800	220	20	136,000	30.9
DBM	3350	240	18	60,000	13.9
CGDF	4950	200	22	109,000	24.7
PCF	2140	140	28	60,000	15.2
DAM	8400	300	18	151,000	28.0
<b>Totals</b>	<b>33,360</b>			<b>655,000</b>	<b>144.5</b>

## FP - Function Points

FP è una misura pesata delle funzionalità del SW.

FP misura l'ammontare di funzionalità all'interno del sistema basandosi sulle specifiche dello stesso.

FP viene calcolato in due step :

1. Calcolare **Unadjusted Function point Count (UFC)**
2. Moltiplicare UFC per il **Technical Complexity Factor (TCF)**

Il FP finale sarà quindi

$$FP = UFC \cdot TCF$$

## Componenti FP

FP è diviso in componenti, a loro volta divisi in due categorie :

- Componenti per il conteggio delle **categorie di dati** :
  - **Number of Internal Logical Files (ILF)** : Un gruppo di dati o informazioni di controllo generati, utilizzati o mantenuti dal sistema SW.
  - **Number of External Interface Files (EIF)** : Un gruppo di dati o informazioni di controllo che passano o vengono condivisi tra le applicazioni, ovvero interfacce leggibili dalla macchina verso altri sistemi e/o l'utente.
- Componenti per il conteggio delle **categorie di transazioni** :
  - **Number of External Inputs (EI)** : Gli elementi forniti dall'utente che descrivono dati orientati all'applicazione, informazioni di controllo (come i nomi di nomi di file e selezioni di menu), o output di altri sistemi che entrano in un'applicazione e modificano lo stato dei suoi file logici interni.
  - **Number of External Outputs (EO)** : Tutti i dati unici o le informazioni di controllo prodotte dai sistemi software, ad esempio report e messaggi.
  - **Number of External Inquiries (EQ)** : Tutte le combinazioni uniche di input/output, in cui un input causa e genera un'uscita immediata senza modificare lo stato dei file logici interni.

## Calcolo TCF

Esistono 15 fattori chiamati **fattori di degree of influence**.

Ad ogni fattore viene associato un valore intero compreso fra :

- 0 (influenza **irrilevante**)
- 5 (influenza **essenziale**)

A questo punto, il TCF viene calcolato come :

$$TCF = 0.65 + 0.01 \sum_{j=1}^{14} F_j$$

con  $F_j$  il j-esimo fattore.

Il TCF varia da 0.65 (se tutti i fattori  $F_j$  sono a 0) a 1.35 (se tutti i fattori  $F_j$  sono a 5), quindi abbiamo un *aggiustamento* di  $\pm 35\%$

### Esempio

$$UFC = 55$$

$$TCF = 0.65 + 0.01(18 + 10) = 0.93$$

Quindi

$$FP = 55 \cdot 0.93 \sim 51$$

## Domanda 31 - Cos'è un software critico?

### 💡 Software Critico >

Un software critico è un sistema software la cui malfunzione può avere gravi conseguenze, come la perdita di vite umane, danni significativi a infrastrutture o elevati costi economici.

### Caratteristiche:

- **Alta affidabilità:** Deve funzionare correttamente in tutte le condizioni previste.
- **Sicurezza:** Deve essere protetto da guasti, attacchi e accessi non autorizzati.
- **Performance rigorose:** Spesso deve operare in tempo reale.

### Esempi:

- Software per dispositivi medici (e.g., pacemaker).
- Sistemi di controllo del traffico aereo.
- Sistemi finanziari critici.
- Sistemi di controllo industriale o infrastrutture energetiche.

## Domanda 32 - PetriNet

Le **Reti di Petri** (**PetriNet**) sono tecniche **formali** di **analisi** (e **specifica**) dei **requisiti**, ovvero servono a descrivere la struttura di un sistema distribuito.

Una PetriNet è formata da un grafo diretto, costituito da :

### Posti (Places)

- Sono rappresentati come i **nodi** del grafo
- Simbolizzano stati, condizioni o risorse
- Possono contenere **token** (marcature) che indicano la disponibilità di risorse o il verificarsi di una condizione

- Possono essere di due tipi :
  - *input* : se da loro escono archi (equivalente a nodo sorgente)
  - *output* : se da loro entrano archi (equivalente a nodo pozzo)

## Transizioni (Transitions)

- Rappresentate graficamente con **rettangoli** o **barre**
- Simbolizzano eventi o attività che possono avvenire nel sistema
- Una transizione può essere abilitata (vedi sotto) ed eseguire un'azione se soddisfa certe condizioni basate sui token.

## Archi (Arcs)

- Collegano posti e transizioni (o viceversa).
- Definiscono la relazione di input e output tra posti e transizioni.
- Possono avere un peso, che indica il numero di token richiesti o prodotti.

## Token

- Rappresentati come puntini all'interno dei posti.
- Indicano lo stato corrente del sistema e vengono spostati dai posti alle transizioni e viceversa.

Si parla di **Marked PetriNet** quando vi è una distribuzione di token all'interno dei posti.

## Esecuzione delle PetriNet

L'esecuzione delle PetriNet è ***non deterministica*** (!!)

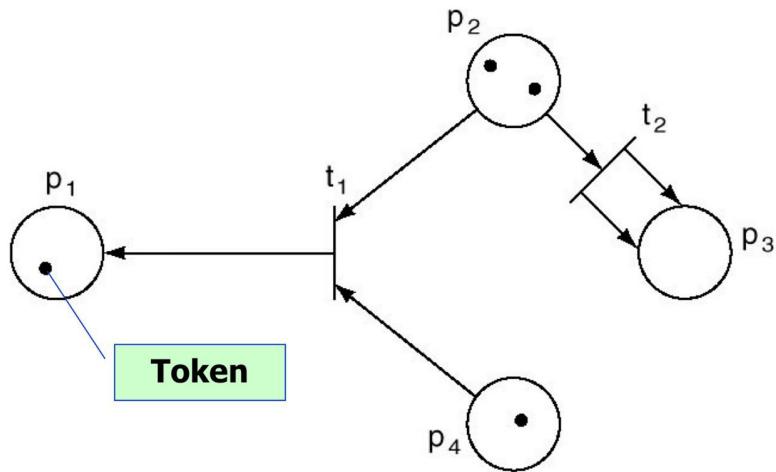
Una transizione si dice **abilitata** se e solo se  $\exists$  token  $\forall$  posto di input connesso alla transizione stessa.

Una trasizione **agisce** sui token secondo la così detta **regola di scatto**

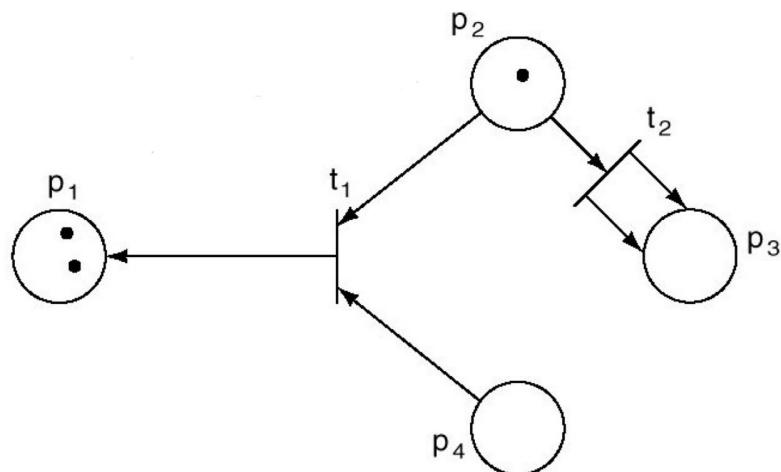
- Lo scatto consuma token dai posti di input, e assegna ai posti di output tanti token quanti sono gli archi di output sulla transizione

## Esempio

Vediamo un esempio di marked petrinet

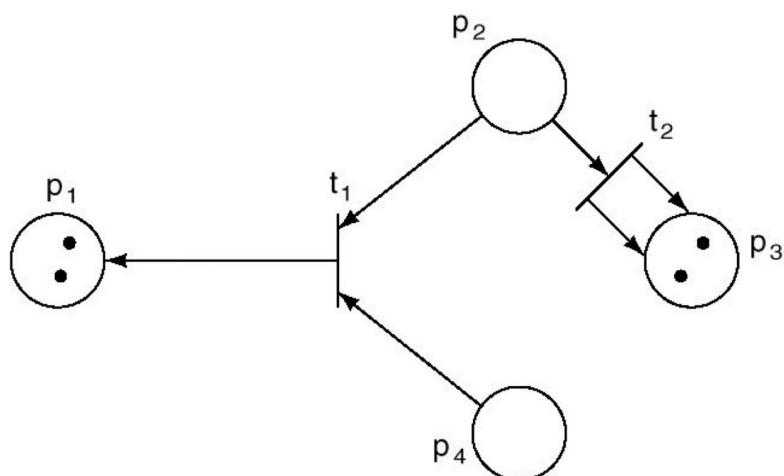


Dopo aver "lanciato" la transizione  $t_1$  la situazione è la seguente :



Si può vedere come la transizione  $t_1$  abbia "consumato" i token di  $p_2$  e  $p_4$

Dopo aver "lanciato" la transizione  $t_2$  la situazione sarà la seguente :



## Domanda 33/33.1 - Component Framework

[Component Framework >](#)

Un **Component Framework** è una struttura software generica che formalizza classi specifiche di applicazioni, fornendo un modo per assemblare sistemi software

utilizzando componenti riutilizzabili.

Questi componenti sono progettati per risolvere problemi comuni di coordinamento e sintesi, garantendo efficienza e affidabilità.

Le caratteristiche principali del Component Framework sono :

- **Componenti riutilizzabili:**

- I componenti sono astrazioni che incapsulano strutture software.
- Possono essere configurati per soddisfare requisiti specifici tramite parametri o legami con altri componenti.

- **Black-box reuse:**

- I componenti nascondono i dettagli della loro implementazione, offrendo un'interfaccia chiara per l'integrazione.
- La composizione di sistemi avviene collegando i "plugs" dei componenti secondo regole predefinite.

- **Encapsulation e Composition:**

- **(Variability)** L'encapsulation consente di rappresentare componenti con una variabilità definita.
- **(Adaptability)** La composition permette di legare parametri o altri componenti per creare sistemi complessi.

- **Differenze tra oggetti e componenti:**

- Gli oggetti incapsulano servizi e sono entità runtime, mentre i componenti sono solitamente statici e servono a costruire sistemi al momento del build.
- I componenti possono essere più granulari degli oggetti e possiedono un'interfaccia di composizione esplicita e verificabile.

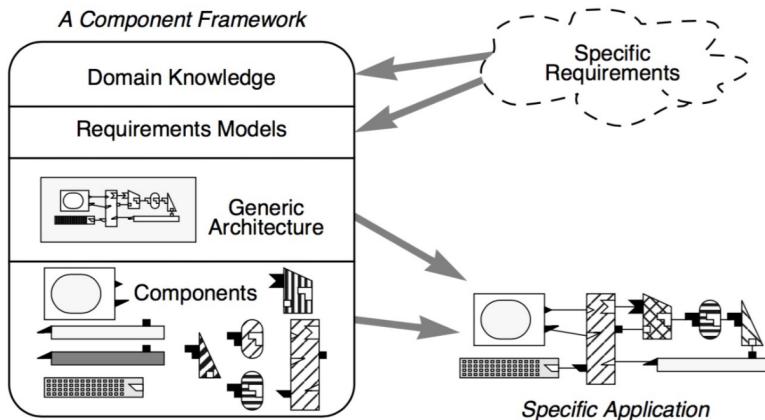
## Funzionamento

I componenti si collegano tramite interfacce definite che specificano le modalità di interazione.

Ogni componente è progettato per essere utilizzato in molteplici applicazioni, fornendo flessibilità e modularità.

I framework offrono una base strutturata su cui sviluppatori possono creare software, utilizzando i componenti come blocchi costitutivi.

## Component framework



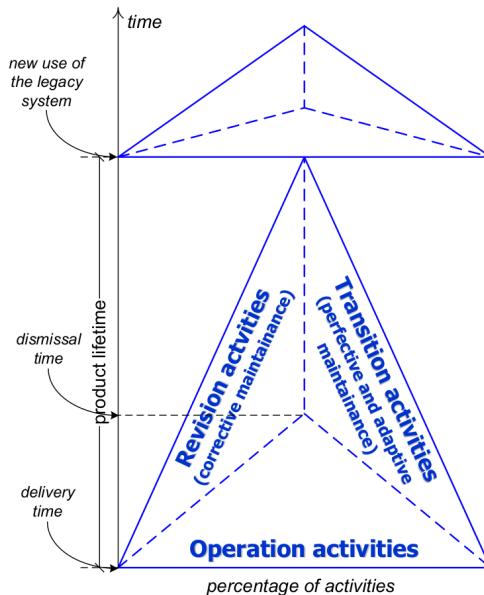
## Domanda 34 - Qual è il metodo per certificare chi produce il software?

Il metodo più utilizzato per certificare chi produce il software è l'adozione di **standard internazionali e certificazioni di qualità**. Alcuni esempi includono:

- Standard di processo
  - **ISO/IEC 9001**: Certificazione di gestione della qualità applicata allo sviluppo software.
  - **ISO/IEC 12207**: Standard che definisce i processi per l'ingegneria del software.
  - **CMMI (Capability Maturity Model Integration)** [vedi qua](#):
    - Misura la maturità dei processi di sviluppo software su una scala da 1 a 5.
    - Livelli più alti indicano processi più ottimizzati e prevedibili.
- Certificazioni individuali
  - **Certified Scrum Master (CSM)** o **Professional Scrum Master (PSM)**: Per chi adotta metodologie Agile.
  - **ISTQB (International Software Testing Qualifications Board)**: Certificazioni per tester.
  - **Certified Software Development Professional (CSDP)**: Certificazione per sviluppatori professionisti.
- Valutazioni settoriali
  - In settori critici (e.g., aerospaziale, medico), ci sono standard specifici, come:
    - **DO-178C** (aerospaziale).
    - **IEC 62304** (dispositivi medici).

## Domanda 35 - Modello McCall

## The Quality Triangle (McCall quality model)



L'immagine mostra il **Triangolo della Qualità** nel contesto del **modello di McCall**, una rappresentazione grafica che evidenzia come le attività legate alla qualità del software si distribuiscano nel tempo e contribuiscano alla longevità del prodotto.

### Interpretazione dell'immagine

Il triangolo collega tre principali tipi di attività di qualità:

#### 1. Operation activities (attività operative):

- Sono legate all'uso effettivo del software nel suo ciclo di vita.
- Si concentrano su aspetti come **affidabilità, efficienza e correttezza**.
- Rappresentano le attività che garantiscono il funzionamento quotidiano del software.

#### 2. Revision activities (attività di revisione):

- Includono la manutenzione correttiva, necessaria per risolvere i bug o correggere errori.
- Si riferiscono alla capacità del software di essere aggiornato per mantenere la sua funzionalità.

#### 3. Transition activities (attività di transizione):

- Includono la manutenzione **adattiva e perfettiva**.
- Queste attività permettono al software di adattarsi a nuovi ambienti o requisiti e migliorano le funzionalità esistenti.

### Elementi chiave rappresentati

#### 1. Asse verticale: Tempo

- Rappresenta la durata del ciclo di vita del prodotto.
- Parte dal **delivery time** (tempo di consegna del software) e termina al **dismissal time** (tempo di dismissione del software).
- Include momenti importanti, come il riutilizzo del software legacy o il suo ritiro.

## 2. Asse orizzontale: Percentuale di attività

- Indica la proporzione relativa delle diverse attività di qualità che vengono svolte durante il ciclo di vita del prodotto.

## 3. Forma triangolare: Equilibrio delle attività

- La distribuzione delle attività deve essere bilanciata per massimizzare la qualità del software durante il suo ciclo di vita.
- L'apice del triangolo (new use of the legacy system) rappresenta la capacità del software di essere riutilizzato o adattato a nuovi scenari.

### Significato del triangolo nel ciclo di vita del software

#### • Fase iniziale (operation activities predominant):

- Nella fase iniziale del ciclo di vita, l'attenzione è principalmente sul rilascio e sul corretto funzionamento del software.
- Le attività operative occupano la maggior parte dello sforzo.

#### • Fase intermedia (revision activities):

- Man mano che il software viene utilizzato, emergono errori o cambiamenti nei requisiti, e il focus si sposta sulla manutenzione correttiva.

#### • Fase avanzata (transition activities):

- Quando il software si avvicina alla fine del ciclo di vita, diventa cruciale adattarlo a nuovi ambienti o sostituirlo con sistemi più moderni.

### Collegamenti al modello di McCall

L'immagine riflette l'idea centrale di McCall: la qualità del software è il risultato di un equilibrio tra le attività operative, di revisione e di transizione. Ogni fase richiede un'attenzione diversa a fattori come **manutenibilità, portabilità e testabilità** per garantire la longevità del prodotto.

## Domanda 36 - Cosa si intende per durata di un prodotto SW?

La **durata di un prodotto software** si riferisce al periodo di tempo durante il quale il software è considerato utile, operativo e rilevante per soddisfare i bisogni degli utenti o degli stakeholder.

Quindi, la durata del prodotto altro non è che il ciclo di vita stesso del prodotto

Le varie fasi e stadi sono spiegate [qua](#)

La durata può essere misurata in termini di anni o mesi e viene spesso analizzata attraverso indicatori come:

- Il tempo trascorso dalla messa in produzione fino al ritiro.
- La frequenza e il costo degli interventi di manutenzione.

- Il livello di soddisfazione degli utenti durante l'intero ciclo di vita.

In sintesi, la durata di un prodotto software non è solo un parametro temporale, ma un indicatore della capacità del software di evolvere e continuare a fornire valore nel tempo.

## Domanda 37 - CMM

### Parlami del CMM

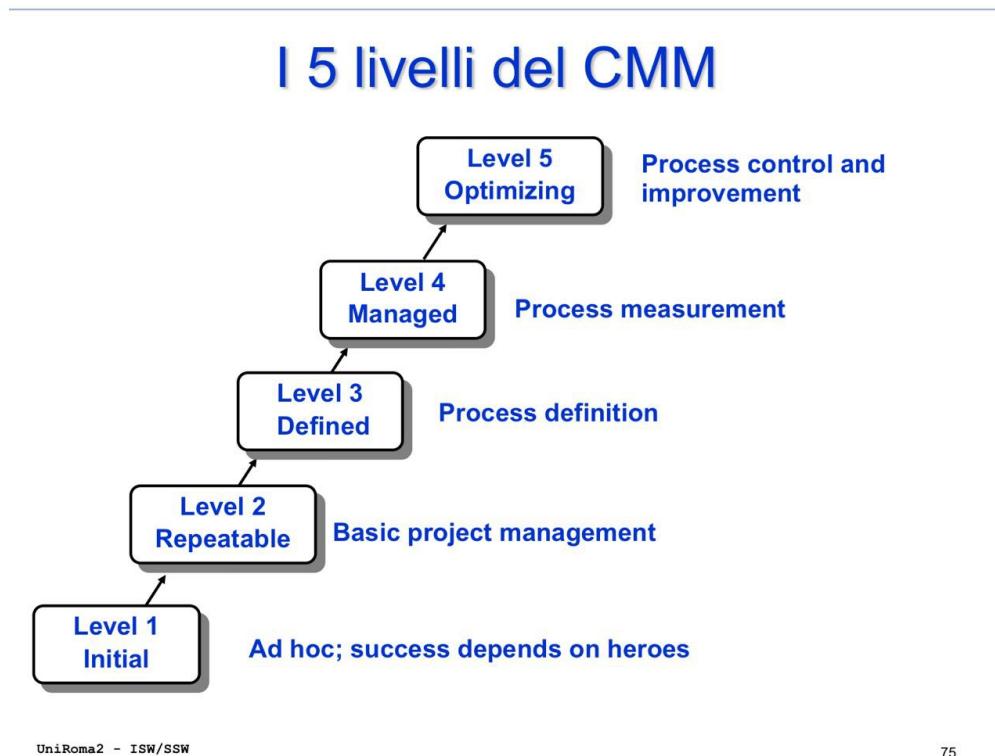
Il **CMM** è un modello atto a determinare il livello di maturità del processo software di una organizzazione.

In altre parole, il CMM è una misura dell'efficacia globale dell'applicazione di tecniche di ingegneria del software.

Predisposto dal **SEI (Software Engineering Institute)**, il CMM è un modello basato su un questionario ed uno schema valutativo a **cinque livelli**.

Ogni livello comprende le caratteristiche definite per il livello precedente

I 5 livelli del CMM sono :



Il CMM associa a ogni livello di maturità alcune **KPA (Key Process Area)**, tra le 18 definite, che descrivono le funzioni che devono essere presenti per garantire l'appartenenza ad un certo livello.

Ogni KPA è descritta rispetto a :

- obiettivi
- impegni e responsabilità da assumere

- capacità e risorse necessarie per la realizzazione
- attività da realizzare
- metodi di "monitoring" della realizzazione
- metodi di verifica della realizzazione

## Gestione progetti software

Lo sviluppo di un prodotto software è una operazione complessa che richiede una specifica attività di gestione

La gestione di un progetto SW implica la **pianificazione**, il **monitoraggio** e **controllo** di persone

## Le quattro "P"

La gestione efficace di un progetto software si fonda sulle "**quattro P**":

- **Personne**, che rappresentano l'*elemento più importante di un progetto software* di successo (il SEI ha elaborato il People Management - Capability Maturity Model)
- **Prodotto**, che identifica le caratteristiche del software che deve essere sviluppato (obiettivi, dati, funzioni, comportamenti principali, alternative, vincoli)
- **Processo**, che definisce il quadro di riferimento entro cui si stabilisce il piano complessivo di sviluppo del prodotto software
- **Progetto**, che definisce l'insieme delle attività da svolgere, identificando persone, compiti, tempi e costi

## Metriche di Struttura

### ☒ Modulo >

Un modulo è una sequenza contigua di statements del programma

### ☒ Metriche di struttura >

Misure nate per determinare quanto un software sia "buono", ovvero misurano la qualità stessa del software

Ci troviamo nella situazione in cui il SW viene diviso in moduli, di conseguenza abbiamo la così detta **architettura dei moduli (structured chart)**

Si rappresenta l'architettura dei moduli come un albero diretto  $S = \{N, R\}$ , dove :

- ogni nodo  $n \in N$  corrisponde a un modulo

- ogni arco  $r \in R$  indica le relazioni

Partendo da questa rappresentazione, vediamo alcune metriche di struttura atte a determinare la qualità del SW

## 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**

La Tree Impurity può essere definita così :

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

## Riuso Interno (Misura di Yin e Winchester)

È una misura che indica il grado di riutilizzo dei moduli all'interno dello stesso prodotto

**Più piccolo è il valore  $r(G)$  meno è il riutilizzo**

**Criticità** : Non può tener conto delle chiamate ripetitive e non può tener conto delle dimensioni del modulo riutilizzato.

Il valore del Riuso si calcola così :

$$r(G) = e - n + 1$$

## 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**

Il livello totale di Information Flow tra un modulo individuale e il resto del sistema è un'**attributo intra-modulare**

Le misure di Information Flow sono contate basandosi su l'interconnessione che un modulo ha con altri moduli nel sistema (es. il *Fan-In* e *Fan-Out* del modulo)

Le misure di Information Flow si basano su :

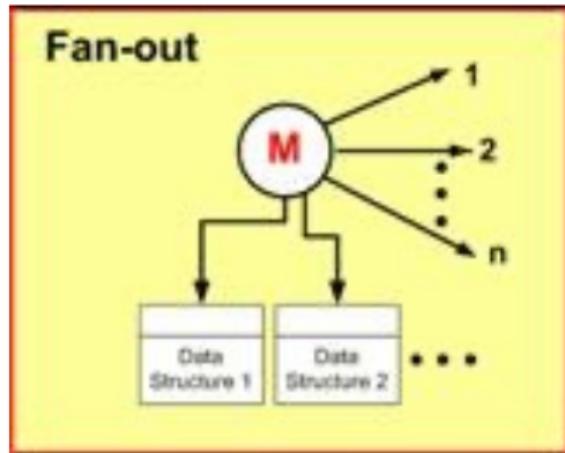
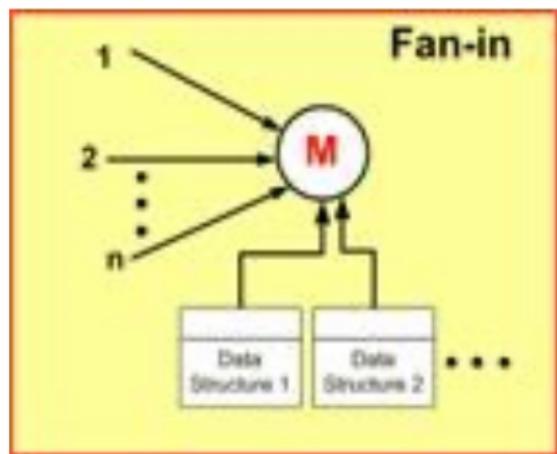
- flusso di informazioni **locale**
  - Diretto : Un modulo chiama un'altro modulo
  - Indiretto : Flusso derivante dai valori di ritorno

- flusso di informazioni **globale**
  - Informazioni che vengono passate tra i moduli secondo una struttura dati globale

## Fan-In e Fan-Out

### ☒ Fan-In e Fan-Out >

- **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  $M$
- **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** di Fan-Out di un modulo indica che lui **influenza/controlla** molti altri moduli

Un valore **basso** di Fan-In di un modulo indica che lui è **influenzato/controllato** da molti altri moduli

## Misura di IF (Henry & Kafura)

L'Information Flow (IF) per un modulo  $M_i$  è definito così :

$$IF(M_i) = [\text{fan-in}(M_i) \cdot \text{fan-out}(M_i)]^2$$

Il valore di IF per un sistema con  $n$  moduli è calcolato così

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

## Misure Strutturali

Le strutture hanno 3 componenti :

- Strutture Control-Flow : Sequenza di esecuzione delle istruzioni
- Data Flow : Tenere traccia dei dati creati o gestiti dal programma
- Struttura Dati : L'organizzazione dei dati indipendentemente dal programma

Le misure strutturali sono usate in tools SW

Come rappresentiamo la "struttura" del programma? Usando i **Control Flowgraphs** (diagrammi di flusso)

Come definiamo la "complessità" in termini della struttura? Usando la **Complessità Ciclomatica**

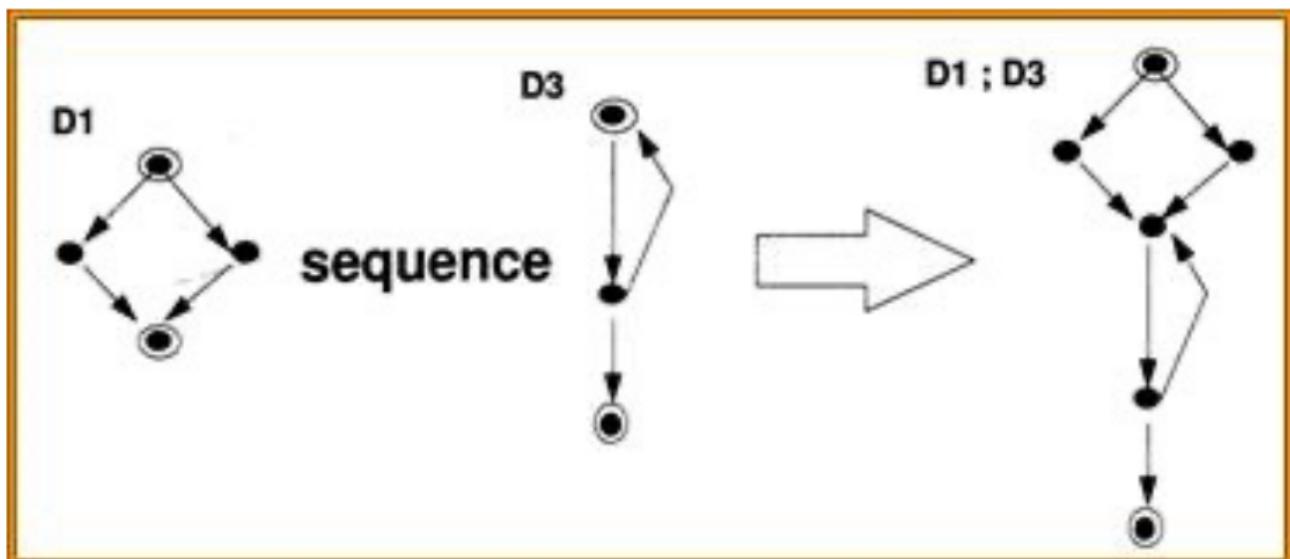
## Flowgraph

La struttura Control Flow è modellata da un flowgraph  $FG = \{N, E\}$

- ogni nodo  $n \in N$  rappresenta uno statement del programma
  - **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 terminale** : nodi con grado di uscita 0
- ogni arco diretto  $e \in E$  indica il flusso di controllo da uno statement ad un altro statement

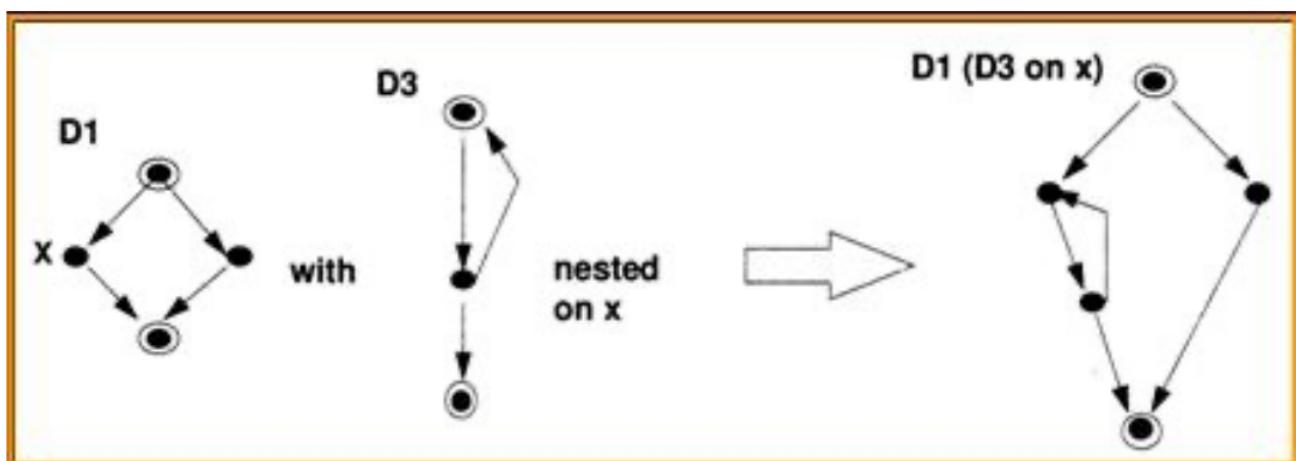
## Sequencing

Siano  $F_1, F_2$  due flowgraphs. Allora la sequenza di  $F_1, F_2$  (scritta come  $F_1; F_2$ ) è un'altro flowgraph formato unendo il nodo terminale di  $F_1$  con il nodo iniziale di  $F_2$



## Nesting

Siano  $F_1, F_2$  due flowgraph. Allora, l'annidamento di  $F_2$  in  $F_1$  sul nodo  $x$ , descritto come  $F_1(F_2)$  è un'altro flowgraph formato partendo da  $F_1$  sostituendo l'arco da  $x$  con l'intero  $F_2$



## Flowgraph primi

I flowgraphs primi sono flowgraph che non possono essere decomposti in modo non banale tramite sequenze e nidificazioni.

### Teorema decomposizione prima >

Ogni flowgraph ha una decomposizione *unica* in una gerarchia di primi, chiamata **Albero di Decomposizione**

## 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 così :

- 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

Esempi di misure per i flowgraph che possono dare informazioni sulla complessità della struttura del codice sono :

- Depth of Nesting
- D-Structuredness

## Depth of Nesting

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

- **Primes** :

$$n(P_1) = 0; n(P_2) = n(P_3) = \dots = n(P_k) = 1 \\ n(D_0) = n(D_1) = n(D_2) = n(D_3) = 1$$

- **Sequencing** :

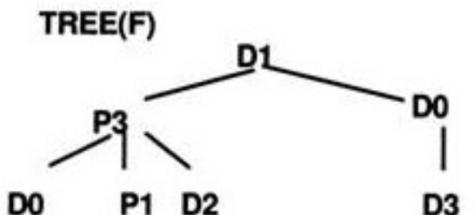
$$n(F_1; F_2; \dots; F_k) = \max\{n(F_1), \dots, n(F_k)\}$$

- **Nesting** :

$$n(F(F_1, F_2, \dots, F_k)) = 1 + \max\{n(F_1), n(F_2), \dots, n(F_k)\}$$

- **Example:**

$$F = D_1((D_0; P_1; D_2), D_0(D_3))$$



$$F = D_1((D_0; P_1; D_2), D_0(D_3))$$

$$n(F) = n(D_1((D_0; P_1; D_2), D_0(D_3))) =$$

$$= 1 + \max\{n(D_0; P_1; D_2), n(D_0(D_3))\} =$$

$$= 1 + \max\{\max\{n(D_0), n(P_1), n(D_2)\}, 1+n(D_3)\} =$$

$$= 1 + \max\{\max\{1, 0, 1\}, 2\} = 1 + \max\{1, 2\} = 3$$

## D-Structuredness

Molte definizioni popolari di programmazione strutturata asseriscono che il programma è strutturato se può essere composto usando solo uno numero piccolo di costrutti ammissibili.

La definizione informale di programmazione strutturata può essere espressa formalmente asserendo che un programma è strutturato  $\iff$  è **D-strutturato**

La D-Structuredness  $d(F)$  di un flowgraph può essere misurata in termini di

- **Primes :**

$$d(P_1) = 0; d(D_0) = d(D_1) = d(D_2) = d(D_3) = 1$$

0 altrimenti

- **Sequencing :**

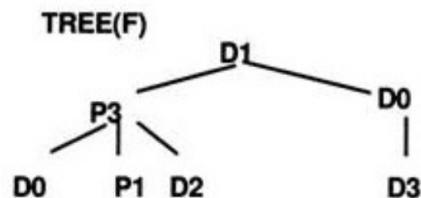
$$d(F_1; F_2; \dots; F_k) = \min\{d(F_1), \dots, d(F_k)\}$$

- **Nesting :**

$$d(F(F_1, F_2, \dots, F_k)) = \min\{d(F_1), d(F_2), \dots, d(F_k)\}$$

- **Example:**

$$F = D_1((D_0; P_1; D_2), D_0(D_3))$$



$$\begin{aligned} d(F) &= d(D_1((D_0; P_1; D_2), D_0(D_3))) = & F = D_1((D_0; P_1; D_2), D_0(D_3)) \\ &= \min\{d(D_1), d(D_0; P_1; D_2), d(D_0(D_3))\} = \\ &= \min\{d(D_1), \min\{d(D_0), d(P_1), d(D_2)\}, \\ &\quad \min\{d(D_0), d(D_3)\}\} = \\ &= \min\{1, \min\{1, 1, 1\}, \min\{1, 1\}\} = \min\{1, 1, 1\} = 1 \end{aligned}$$

→ **F is D-structured** (F is built up of common primes, i.e. simple structures allowable in structured programming)

## Complessità Ciclomatica

La complessità di un programma può essere misurata dal numero **ciclomatico** del flowgraph del programma.

Si può calcolare in due modi diversi :

- Flowgraph-based
- Code-based

### Flowgraph-based

La complessità ciclomatica  $v(F)$  è misurata come :

$$v(F) = e - n + 2$$

Questo valore misura il numero di percorsi lineari indipendenti in  $F$

Oppure, può essere calcolato come

$$v(F) = 1 + d$$

con  $d$  = numero dei nodi predicato, ovvero il numero di punti di decisione nel programma

La complessità del priming è  $v(F) = 1 + d$

La complessità del sequencing è  $v(F_1; \dots; F_n) = \sum_{i=1}^n v(F_i) - n + 1$

La complessità di annidare dentro un dato prime è  $v(F(F_1; \dots; F_n)) = v(F) + \sum_{i=1}^n v(F_i) - n$

$$v(F) = e - n + 2$$

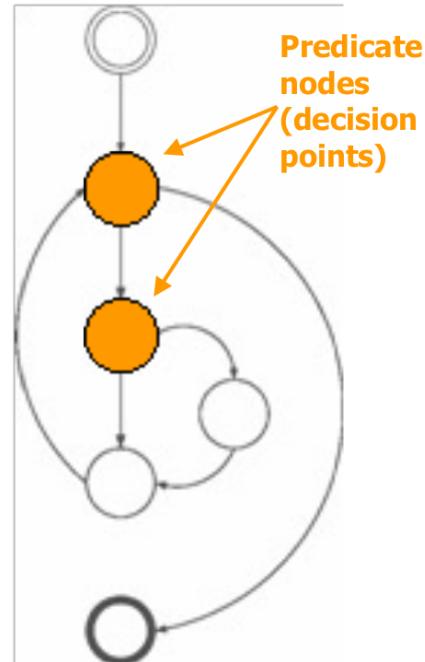
$$v(F) = 7 - 6 + 2$$

$$v(F) = 3$$

or

$$v(F) = 1 + d$$

$$v(F) = 1 + 2 = 3$$



## Complessità essenziale di McCabe

La **complessità essenziale** di un programma con flowgraph  $F$  è data da

$$ev(F) = v(F) - m$$

con  $m$  = numero di sub-flowgraphs  $D_0, D_1, D_2, D_3$

## Testing

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

### Verification & Validation >

- Verification : Risponde a domande tipo "stiamo costruendo il prodotto correttamente?"
  - Ovvero, il software deve essere conforme alle sue specifiche
- Validation : Risponde a domande tipo "stiamo costruendo il giusto prodotto?"
  - Ovvero, il software deve essere quello che l'utente veramente ha richiesto.

- *Ispezioni Software* : Si occupa dell'analisi della rappresentazione statica del sistema per scoprire i problemi (**tecniche statiche**)
- *Testing Software* : Si occupa dell'esercizio e dell'osservazione del comportamento del prodotto (**tecniche dinamiche**)

Ci sono 3 tipi di testing

- Validation testing
- Defect testing
- Statistical testing

## Defect testing

Il goal del **defect testing** è quello di scoprire **difetti** nei programmi (😊)

Questo va in contrasto con il *validation testing*

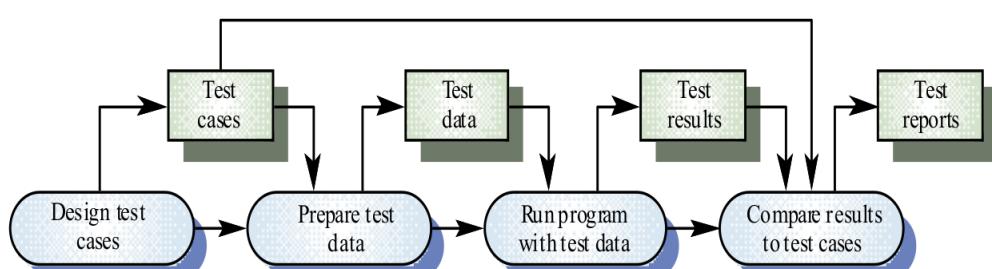
Un defect test *riuscito* è un test che causa un comportamento anomalo all'interno del programma.

Il defect testing ha due fasi :

- *Component testing*
  - Si testano le componenti del programma in modo individuale
  - In generale la responsabilità è del developer del componente
  - I test sono derivati dall'esperienza del developer
- *Integration testing*
  - Si testano gruppi di componenti integrati per creare un sistema o sottosistema
  - La responsabilità è del team di testing indipendente
  - I test si basano sulle specifiche del software

**Politiche del testing :**

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



Nel testing ci sono due approcci fondamentali, che sono [Black/White box testing](#)

Abbiamo vari tipi di testing, che si categorizzano in base all'approccio usato

- **Integration testing** : Si testano sistemi/sottosistemi completi composti da componenti integrati
  - Categoria : Black-Box
- **Interface testing** : Trovare difetti dovuti ad errori dell'interfaccia o assunzioni invalide sulle interfacce
  - Categoria : Black-Box
- **Stress testing** : Testa il sistema oltre il suo massimo carico. Stressare il sistema spesso fa emergere i difetti.
  - Categoria : Black-Box
- **Object-oriented testing** : Vengono testati i componenti che sono classi di oggetti, istanziati come oggetti
  - Clasificazione : White-Box
- **Object class testing** : Test che copre l'intera classe
  - Testando tutte le operazioni associate ad un oggetto
  - Fanno lavorare l'oggetto in tutti i possibili modi
  - Categoria : White-Box
- **Scenario-based testing** : Identificare gli scenari dai casi d'uso e integrarli con diagrammi di interazione che mostrino gli oggetti coinvolti nello scenario.
  - Categoria : White-Box