Process Perspective

Un processo di sviluppo e è costituito da un insieme di attività che conducono alla realizzazione di un prodotto software e può essere tipicamente analizzato secondo diverse prospettive:

- **Activities Perspective**: definisce di quali attività si compone un processo;
- **Workflow Perspective**: definisce le relazioni temporali tra le attività che devono essere svolte e le possibili condizioni che è necessario verificare per avviare un'attività;
- **Data-flow Perspective**: definisce quali sono gli artefatti che devono essere prodotti dalle varie attività e dunque gli artefatti che saranno prodotti al termine del progetto;
- **Role/Actions Perspective**: definisce i ruoli necessari nell'organizzazione al fine di portare avanti un progetto in accordo al processo nonché le responsabilità di tali ruoli in relazione alle attività da compiere.

Modelli di processi

Un modello è una descrizione astratta di un sistema, che aiuta a studiarlo focalizzandosi sulle caratteristiche necessarie al raggiungimento di un determinato scopo.
Esistono diverse tipologie di modelli che differiscono per aspetti mostrati e anche per il diverso livello di dettaglio.

Modello di processo a Cascata "Waterfall"

Il modello a cascata si basa su uno **svolgimento sequenziale** delle varie attività di sviluppo del software. Come gli altri modelli, anche questo presenta una **suddivisione in fasi,** dove alla fine di ogni fase viene prodotto un **documento relativo alla fase** appena conclusa che verrà poi rielaborato nelle fasi successive.
La peculiarità del processo consiste nel presupposto che **ciascuna fase sia finita prima dell'inizio della fase successiva**, quindi si passa dalla definizione dei requisiti alla modellazione che poi lascia il passo all'effettiva programmazione seguita dalla verifica e dal rilascio del software.
Dal momento che il modello **implica che in ogni fase si operi sul prodotto nella sua interezza**, lo rende un modello adatto a progetti i cui requisiti sono chiari fin dall'inizio e che hanno uno sviluppo prevedibile (eventuali errori durante la "cascata" comporta la revisione di tutti gli step precedenti).
Se da una parte si ottiene il vantaggio di un'alta visibilità ed idea generale della fase in cui si trova il progetto, questo modello si associa ad **un'alta percentuale di fallimenti** ed è quindi **poco efficace**.

Modello di processo evolutivo

Questo modello cerca di ovviare alle problematiche del modello a cascata **basandosi sulla prototipazione.** Viene quindi creato un **prototipo**, ovvero una versione approssimata, parziale, ma funzionante dell'applicazione che viene sviluppata.
Questo prototipo si andrà ad **evolvere ad ogni fase**, andando ad aggiungere man mano delle funzionalità fino ad ottenere un prodotto finale funzionante e che risponde a tutte le esigenze del committente. Tutto ciò viene fatto a costo di un'**organizzazione poco strutturata** e **difficoltà in fase di mantenimento**.
Nel momento in cui si giunge all'analisi dei requisiti bisogna scegliere cosa fare del prototipo:

- lo si butta via e si procede ad un progetto ex novo del sistema, quindi un **throw-away**
- lo si usa come base (ampliandolo e perfezionandolo) per il prodotto finale (**prototipo evolutivo** o sistema pilota)

Con un prototipo throw-away si può realizzare il prototipo senza vincoli di efficienza, mentre con quello evolutivo si rischia di andare a mantenere scelte fatte nelle fasi precedenti nonostante possano essere inadeguate al prodotto finale.

Modelli di processo Iterativo

I modelli iterativi vanno a **fondere il modello evolutivo e quello a cascata** andando a conciliare gli aspetti positivi di entrambi ed eliminando gli aspetti negativi.
Si va ad organizzare lo sviluppo totale in più mini-progetti detti **iterazioni** che ha le proprie attività

di requisiti, progettazione ed implementazione.
Ci possono essere **due diversi approcci** a questo tipo di modello:

- **Rilascio incrementale:** il processo prosegue con *tanti mini waterfall in sequenza*, andando a rilasciare ad ogni iterazione un **sistema funzionante** pianificando nel mentre le iterazioni successive con nuove funzionalità.
  Il ciclo può e dev'essere **ripetuto diverse volte** andando sempre a ridurre il rischio di fallimento fino ad arrivare ad una **validazione del prodotto** soddisfacente.
  I vantaggi sono l'efficienza per team di sviluppo piccoli, riduzione del rischio di fallimento, funzionalità più importanti maggiormente testate e l'uso sul prototipo (che comporta scoperta, definizione e modifica dei requisiti).
- **Sviluppo a spirale:** introduce la **gestione del rischio** nella pianificazione delle varie iterazioni. I capisaldi sono: pianificazione, analisi dei rischi, sviluppo, verifica.

Nel modello iterativo, quindi, il lavoro procede con una serie di cicli strutturati di **costruzione-feedback-adattamento.** L'approccio iterativo consente ai team di sviluppo di adattarsi ai cambiamenti nei requisiti, di risolvere i problemi in modo incrementale e di consegnare parti del software funzionanti più rapidamente rispetto ai modelli di sviluppo tradizionali.

Il processo unificato UP (unified process)

Lo unified process è un **processo iterativo standardizzato** usato per lo sviluppo del software ed è divenentato il processo di riferimento associato all'UML.
Le caratteristiche fondamentali sono le seguenti:

- **Guidato dal rischio**: nelle attività di pianificazione si mira a fare scelte che *riducano il rischio di fallimento*, dapprima individuando i rischi e poi attuando delle contromisure.
- **Guidato dai casi d'uso**: vengono utilizzati i casi d'uso come strumento principale per la specifica dei requisiti, progettazione e validazione del sistema.
- **Incentrato sull'architettura**: Questo approccio è fondamentale per creare sistemi software di alta qualità, scalabili, robusti e manutenibili. L'architettura è al centro degli sforzi del team di progetto per modellare il sistema.
- **Processo incrementale**: presenta tutti i vantaggi del modello iterativo incrementale.

L'arco temporale del processo di sviluppo è s**uddiviso in quattro fasi successive**. Ogni fase ha un obiettivo e produce un insieme di semilavorati chiamato **milestone** (pietra miliare). Ogni fase è suddivisa in un numero variabile di iterazioni ed ogni iterazione produce una versione provvisoria detta **baseline**, ovvero un'insieme di artefatti rivisti e approvati che forniscono una base condivisa con documentazione associata.

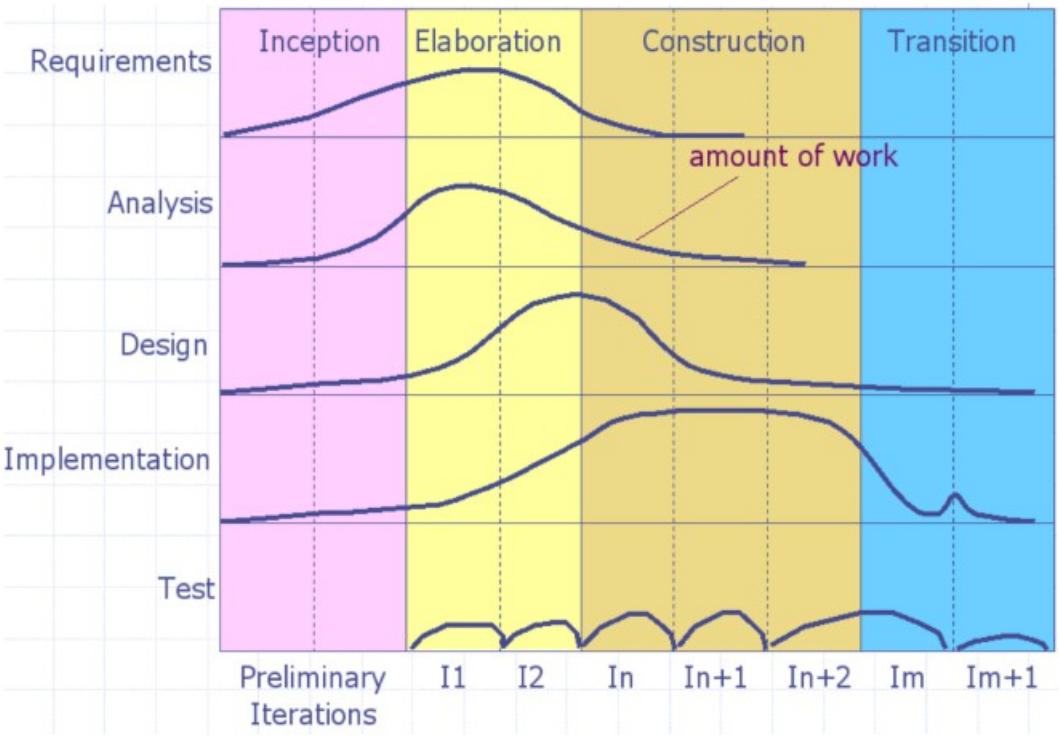
Le iterazioni prevedono le seguenti attività: *pianificazione, requisiti ed analisi, progettazione, implementazione, integrazione e test, validazione, rilascio* (interno o esterno).
Generalmente i metodi iterativi raccomandano una durata dell'iterazione dalle **2 alle 6 settimane**.
Un'iterazione con una durata prefissata viene definita **timeboxed.**

Le **fasi** invece sono periodi di tempo che hanno come obiettivo il raggiungimento di un macro-obiettivo per il progetto. Il **passaggio alla fase successiva avviene dopo il raggiungimento dello scopo** della fase attuale.
Le 4 fasi sono:

1. **Avvio (inception)**: possiede gli *stessi obiettivi dello studio di fattibilità* ai quali viene aggiunta l'analisi del rischio. Principalmente si cerca di **individuare il business case**, ovvero il mercato al cui il progetto afferisce. Si effettua anche l'ideazione di un prototipo.
   Se il progetto non supera questa milestone, detta **Lifecycle Objective Milestone**, esso dovrà essere abbandonato o ridefinito.
   Si tratta della fase più piccola del progetto, dalla durata di una o alcune settimane.
2. **Elaborazione (elaboration)**: ci si concentra sul perfezionare le conoscenze della fase

precedente allo scopo di **produrre una baseline** architetturale eseguibile. Si ottiene quindi un primo prototipo eseguibile andando a coprire circa l'80% dei casi d'uso, si esegue una *revisione dei rischi e del business case* e si completa una **pianificazione del progetto complessivo**. Questa fase deve concludersi con il superamento di una milestone detta **Lifecycle Architecture Milestone**.

3. **Costruzione (construction)**: ha come obiettivo la **produzione del sistema finale** che abbia una qualità sufficiente ad essere rilasciato (e completo di modelli e test). Viene prodotta la prima release (interna) del sistema. La milestone di questa fase si chiama **Initial Operational Capability**. Per il rilascio viene effettuato un *accordo tra le parti.*
4. **Transizione (transition)**: ci si concentra sulla correzione degli errori individuati in fase di beta-test nella fase precedente e si **concorda sul rilascio con successo** del prodotto.
   Si deve in particolare verificare che il prodotto sia conforme alle aspettative descritte nella fase di avvio. Si raggiunge così la milestone **Product Release**.



Ingegneria dei Requisiti

È il processo di scoperta, analisi, documentazione e mantenimento dei requisiti nei processi di sviluppo software. Si occupa di definire **cosa un sistema debba fare**, le sue proprietà essenziali ed i vincoli a cui deve rispondere.
I **Software Intensive Systems (SIS)** sono sistemi complessi in cui il software gioca un ruolo cruciale nella realizzazione delle funzionalità principali. Questi sistemi sono caratterizzati da un'elevata dipendenza dal software per il loro funzionamento, controllo e gestione.

I Requisiti

Un requisito è una condizione o una capacità specifica che stabilisce ciò che un sistema, un prodotto o un servizio deve fare o raggiungere per soddisfare determinate esigenze o aspettative dell'utente.
I requisiti vengono **classificati in base al focus** che si persegue nell'analisi, distinguendoli in: ***carattere** dei requisiti, **destinatari*** ed ***origine dei requisiti***.
Ci sono *diverse tecniche* possibili per specificare i requisiti:

- **informali**: utilizzano linguaggi naturali e una struttura meno rigorosa per descrivere i requisiti. Queste tecniche sono facili da comprendere per tutti gli stakeholder, ma possono essere ambigue e difficili da mantenere;
- **semi-formali**: utilizzano linguaggi e notazioni standardizzati che riducono l'ambiguità e migliorano la precisione rispetto alle tecniche informali (diagrammi e modelli).
- **formali**: utilizzano linguaggi matematici e logici rigorosi per descrivere i requisiti, richiedendo per tanto competenze avanzate.

Destinatari dei Requisiti

- **Requisiti Utente**: come dice il nome sono **destinati agli utenti.** Si occupano del **comportamento osservabile** del sistema dal punto di vista dell'utente e senza interessare le scelte di design.
- **Requisiti di Sistema**: rivolti ai **progettisti/sviluppatori** e vanno a precisare le necessità a livello di sistema utili a soddisfare gli obiettivi specificati nei requisiti utente. Hanno quindi un **alto livello di dettaglio e precisione**. Anche questi *non* dovrebbero influenzare le scelte di design, tuttavia a volte si hanno le mani legate (ad esempio quando bisogna interagire con applicazioni già esistenti di terze parti).

Carattere dei Requisiti

- **Requisiti funzionali**: definisce i servizi che il **sistema dovrebbe fornire** (si focalizza sulle funzionalità)**,** il **comportamento** di quest'ultimo nelle varie situazioni ed in alcuni casi anche cosa *non* dovrebbe fare.
- **Requisiti qualitativi**: descrivono le caratteristiche del sistema che influenzano l'esperienza dell'utente (estetica, facilità d'uso ecc.), ovvero le **proprietà qualitative**. Sono essenziali per garantire un'esperienza utente soddisfacente e per creare sistemi software che siano apprezzati dagli utenti.
- **Vincoli**: Un vincolo è un requisito organizzativo o tecnologico che restringe il modo in cui il sistema deve essere sviluppato.

Origine dei Requisiti

- **Requisiti di Dominio**: riguardano quei requisiti che derivano direttamente dallo specifico dominio e/o contesto applicativo. Sono difficili da identificare perché ovvi al committente e spesso ignorati e non riportati.

Attività dell'ingegneria dei requisiti

Nel ciclo di ingegneria dei requisiti, ci sono diverse attività: *studio di fattibilità, elicitazione ed analisi dei requisiti, validazione* e *gestione*.

Formato definizione Requisiti
Le informazioni minime da riportare in un formato strutturato per la specifica di requisiti sono:

- **ID**: identificativo unico in un formato utile agli scopi;
- **Nome**: nome mnemonico, tipicamente il nome dell'azione;
- **Descrizione**: fornisce ulteriori indicazioni che servono a migliorare comprensione del requisito;
- **Sorgente**: l'origine o la fonte del requisito, che può essere derivata da interviste con gli utenti, analisi del dominio, normative, requisiti legali, ecc.

Formato VOLERE, modello standardizzato per la definizione dei requisiti

ID	*Identificativo unico*
Tipo	*Tipologia del requisito*
Evento/CU correlato	*A quale CU o evento si riferisce*
Descrizione	*Una frase che caratterizzi il sistema*
Motivazione	*Breve descrizione a contestualizzare il requisito*
Sorgente	*Chi ha rischiesto inserimento requisito*
Criterio di Valutazione	*Come valutare il soddisfacimento del requisito*
Soddisfazione Cliente	*Grado di soddisf. se requisito sarà implementato*
Insoddisfazione Cliente	*Grado di insoddisf. se requisito non sarà implementato*
Conflitti	*conflitti con altri requisiti*
Priorità	*importanza per il cliente*
Materiale di supporto	*documentazione che può migliorare comprensione*
Storia	*Creazione, modifiche*

Studio di fattibilità

Si tratta di uno **studio preliminare** sulle implicazioni che il sistema avrà allo scopo di fornire una **raccomandazione sul continuare o meno lo sviluppo**.
Nel fare ciò si risponde anche a questioni sulla natura del sistema, se sarà utile al raggiungimento dello scopo, se si dispone della tecnologia e del tempo necessario. Per la **raccolta d'informazioni** è importante l'interazione con il "cliente".

Elicitazione ed analisi dei requisiti

L'elicitazione e analisi dei requisiti è il processo che permette di **comprendere, raccogliere, analizzare e documentare** le esigenze, le aspettative e i vincoli degli stakeholder nei confronti di un sistema software.
L'obiettivo principale di questa fase è descrivere in modo **chiaro, completo e non ambiguo** ciò che il committente desidera dal prodotto software, riducendo il rischio di incomprensioni nelle fasi successive di sviluppo.
Questo processo è complesso perché dipende fortemente dall'interazione umana: gli stakeholder possono avere **punti di vista differenti**, usare linguaggi di dominio diversi e non essere sempre in grado di esprimere in modo esplicito i propri bisogni.

1) Individuazione degli Stakeholder (Attori)

Gli stakeholder sono **tutte le persone o i gruppi che hanno un interesse nel sistema** o che ne sono influenzati, direttamente o indirettamente.
Una stessa entità può ricoprire più ruoli all'interno del progetto. L'individuazione degli stakeholder è un passo fondamentale, poiché requisiti incompleti o errati spesso derivano proprio da una **mancata o errata identificazione degli attori coinvolti**. Gli attori si suddividono in:

- **Attori primari**: interagiscono direttamente con il sistema per svolgere azioni o ottenere benefici.
- **Attori finali**: sottoinsieme degli attori primari, sono gli utenti che traggono un beneficio

diretto dall'uso del sistema.

- **Attori di supporto**: forniscono servizi o informazioni al sistema oppure assistono gli attori primari, senza ottenere benefici diretti.
- **Attori fuori scena**: non interagiscono direttamente con il sistema, ma ne influenzano requisiti e funzionamento (es. enti regolatori, policy aziendali).

Per ottenere una visione completa dei requisiti è necessario considerare diversi **punti di vista**:

- **Punto di vista diretto**: riguarda i requisiti raccolti dagli stakeholder primari che interagiscono direttamente con il sistema. Si analizzano flussi normali, eccezioni, attività parallele e stato finale del sistema.
- **Punto di vista indiretto**: coinvolge stakeholder che non usano direttamente il sistema, ma che lo influenzano o ne sono influenzati.
- **Punto di vista di dominio**: si basa sulla conoscenza del dominio applicativo in cui il sistema opera, includendo norme, pratiche e vincoli specifici.

L'integrazione di questi punti di vista consente di ottenere requisiti **bilanciati, completi e coerenti** con il contesto di utilizzo.

2) Scoperta dei Requisiti

La scoperta dei requisiti è la fase in cui gli analisti collaborano con gli stakeholder per identificare **cosa il sistema deve fare** e **quali vincoli deve rispettare**.
Per svolgere questa attività esistono diverse tecniche, ciascuna con vantaggi e limiti.

2.1.1) Interviste

Le interviste sono incontri diretti tra analista e stakeholder, con l'obiettivo di raccogliere informazioni in un contesto di **fiducia e comunicazione efficace**. Ci sono più **tipi di intervista:**

- **Interviste strutturate**: prevedono domande predefinite e una sequenza fissa. Sono utili per raccogliere informazioni precise e confrontabili.
- **Interviste semi-strutturate**: combinano domande preparate con la possibilità di approfondire aspetti emersi durante la conversazione.
- **Interviste non strutturate**: assumono la forma di una conversazione libera e sono utili per esplorare idee e bisogni non ancora formalizzati.

Inoltre ogni intervista è **divisibile in diverse fasi:**

1. **Preparazione**: definizione degli obiettivi, selezione dei partecipanti e preparazione delle domande.
2. **Esecuzione**: presentazione degli obiettivi, conduzione dell'intervista e sintesi finale.
3. **Follow-up**: rielaborazione delle informazioni raccolte, individuazione dei gap e comunicazione dei risultati.

Le interviste sono efficaci per comprendere i bisogni principali, ma **non sono ideali per individuare requisiti fortemente innovativi**.

2.1.2) Workshop

I workshop coinvolgono **più stakeholder contemporaneamente** e favoriscono la collaborazione e il confronto diretto.

- Sono particolarmente utili per individuare **requisiti complessi e innovativi**.
- Richiedono un moderatore e un segretario (minute-taker).
- Comportano uno sforzo organizzativo elevato, ma possono produrre risultati di alta qualità.

I workshop inoltre si compongono di diverse fasi che sono:

- **preparazione**, dove si definiscono gli obiettivi, le tecniche da applicare, partecipanti, luogo ed un organizzatore con un minute-taker (che prende nota dei passaggi più importanti)
- **esecuzione**, che si divide in *apertura* (esposizione di obiettivi e tecniche), *conduzione* (dove il moderatore gestisce ed il minute-taker prende nota) e la *chiusura* (raccolta dei risultati ed illustrazione sommaria di questi)
- **Follow-up**, vengono riorganizzati i risultati e fatti girare ai partecipanti che *possono chiedere modifiche*

2.1.3) Altre tecniche

- **Focus group**: coinvolge un gruppo selezionato di stakeholder che discutono, sotto la guida di un moderatore, vari aspetti del sistema da sviluppare. Al termine un documento potrà essere prodotto e utilizzato per altre attività;
- **Osservazione (etnografia)**: l'analista osserva gli utenti nel loro ambiente naturale di lavoro per comprendere meglio le loro attività, processi e contesto d'uso del sistema;
- **Questionari**: strumenti per raccogliere informazioni dai stakeholder attraverso domande scritte, che possono essere distribuite e raccolte in forma cartacea o elettronica;
- **Perspective-Based Reading (PBR)**: revisione dei requisiti da diversi punti di vista (utente, sviluppatore, tester).

2.1.4) Tecniche Ausiliarie

Queste tecniche supportano e integrano l'elicitazione:

- **Brainstorming**: stimola la generazione di idee creative.
- **KJ Method**: forma di brainstorming particolarmente utile in contesti di partecipanti eterogenei. Permette di far emergere requisiti a partire da un gruppo di persone allo stesso tempo. È una tecnica utilizzata per organizzare grandi quantità di dati o idee in gruppi tematici per trovare strutture o pattern.
  Consta di tre fasi principali:
  1) **riflessione individuale** ogni partecipante singolarmente riflette e identifica caratteristiche ritenute rilevanti;
  2) **presentazione e discussione**: le carte vengono prese dal moderatore e lette ad alta voce. I partecipanti possono porre domande e richiedere chiarimenti;
  3) **raggruppamento e sintesi**: le carte ritenute rilevanti vengono raggruppate considerando la possibilità che si riferiscano a tematiche omogenee del sistema;
- **Prototyping**: consiste nella creazione di versioni preliminari del sistema (prototipi) per esplorare idee, raccogliere feedback e validare i requisiti con gli stakeholder.
- **Mind mapping**: rappresentazione visiva dei requisiti e delle relazioni tra essi.
- **Elicitation checklist**: lista di controllo per garantire la copertura di tutte le aree rilevanti.

3) Documentazione dei Requisiti

I requisiti raccolti vengono formalizzati in un **documento dei requisiti software**, che rappresenta il riferimento principale per gli sviluppatori.

La documentazione include sia **requisiti utente** che **requisiti di sistema** e segue generalmente un formato standard, dipendente dal processo adottato.

Un criterio molto utilizzato è il **principio MoSCoW**, che classifica i requisiti in base alla loro priorità.

Validazione

Verificare se i requisiti scritti sono necessari a soddisfare esigenze del progetto e cerca di **rimuovere possibili problemi** nella specifica dei requisiti.
Possibili verifiche sono:

- Controllo di **validità**: ciò che è stato specificato=ciò che è necessario all'utente
- Controllo di **consistenza**: requisiti non contradditori
- Controllo di **completezza**: i requisiti coprono tutte le possibili funzionalità
- Controllo di **concretezza**: ciò che è stato specificato è realizzabile
- **Verificabilità**: requisiti scritti in modo che possa essere verificata la loro soddisfazione

Alcune **attività comuni** di questo processo sono: **revisione dei requisiti** (processo manuale che coinvolge team e stakeholders nell'esaminare i requisit), **prototipazione** (uso di prototipi dimostrativi) e **generazione di test-case** (si fanno sessioni di test dedicate).

Gestione

L'attività di gestione dei requisiti si occupa di far **emergere, permettere e gestire modifiche** ai requisiti, dal momento che questi una volta creati non sono fissi.
Per gestire le modifiche si usano tecniche come: Identificazione dei requisiti tramite ID, definizione di un processo di modifica, meccanismi di tracciabilità ed uso e supporto da parte di tool.

Requisiti Qualitativi

Efficienza (interna/esterna): l'efficienza si riferisce alla necessità del software di utilizzare risorse al fine di svolgere i propri compiti. Quando la risorsa usata è il tempo si parla di prestazioni. Si riferiscono alla velocità con cui il sistema risponde agli stimoli, oppure al numero di stimoli che il sistema riesce a gestire nell'unità di tempo. L'efficienza può essere misurata tramite: misurazioni a runtime, analisi, simulazione.

Robustezza (esterna): misura di quanto il software si comporta in maniera ragionevole in circostanze non prevista dalla specifica. Non si può misurare quanto un prodotto software sia robusto ma lo si può paragonare ad altri software dello stesso tipo o simili.

UML

È un linguaggio di modellazione specifica che si basa sul paradigma object-oriented. Si tratta di un **linguaggio grafico** semi-formale ed applicabile dalla fase di analisi e specifica dei requisiti alla fase di codifica.
Può essere utilizzato come **progetto, linguaggio di programmazione** o **abbozzo** e la *modellazione agile* ne risalta *l'uso da abbozzo* andando a rappresentare **due caratteristiche** del sistema:

- **Struttura:**
  Per applicare l'UML in modo efficace bisogna avere diversi punti di vista e prospettive come un punto di **vista concettuale** (interpretazione dei diagrammi come descrizione del mondo reale) ed un **punto di vista software** (con diagrammi che descrivono componenti sw come implementazioni).
  La struttura di UML è composta di 3 elementi fondamentali:
  Costituenti fondamentali: entità, relazioni e diagrammi (delle classi, dei casi d'uso, delle attività...);
  Meccanismi comuni: tecniche comuni per raggiungere specifici obiettivi, come *specifiche, ornamenti, meccanismi di estendibilità* ;
  Architettura: il modo in cui UML esprime l'architettura di un sistema. Il modello contiene diverse visualizzazioni come vista *logica, del processo, sviluppo* e *distrubuzione,* più gli scenari.
- **Comportamento:** ciclo di vita degli oggetti e specifica delle collaborazioni che e intercorrono tra gli oggetti per fornire le funzionalità richieste.

Diagramma dei casi d'uso (Use Case Diagram)

Il **diagramma dei casi d'uso** è un diagramma UML che rappresenta **le funzionalità offerte da un sistema** dal punto di vista degli **attori esterni** che interagiscono con esso.
È uno strumento di **analisi dei requisiti funzionali** e viene utilizzato principalmente nelle **fasi iniziali del processo di sviluppo**, in particolare nel **Processo Unificato**.

Il diagramma non descrive *come* il sistema è realizzato internamente, ma *cosa* il sistema deve fare per soddisfare gli obiettivi degli utenti.
Un **caso d'uso** è la **specifica di una sequenza di azioni** che un sistema, sottosistema o classe può eseguire **interagendo con uno o più attori esterni**, includendo:

- lo **scenario principale di successo**;
- eventuali **sequenze alternative**;
- eventuali **sequenze di errore**.

I casi d'uso sono **testi strutturati**, non diagrammi: il diagramma dei casi d'uso fornisce solo una **vista sintetica e grafica** delle interazioni.

Un caso d'uso rappresenta una **storia** in cui un attore utilizza il sistema per **raggiungere un obiettivo concreto**.

Le componenti principali sono:

1. **Confine del sistema**: si rappresenta come un rettangolo dove all'esterno ci sono gli attori e all'interno tutti i casi d'uso del sistema. Possono anche esserci casi d'uso non appartenenti al sistema e quindi fuori dal rettangolo.
2. **Attori**: Un **attore** rappresenta un **ruolo** che un'entità esterna assume quando interagisce con il sistema. Non rappresenta una persona specifica, ma un **comportamento o funzione**. Gli attori si dividono nelle seguenti tipologie: attori ***primari*** (uso diretto del sistema), ***finali*** (interessato al risultato del caso d'uso), ***di supporto*** (fornisce un servizio al sistema) e ***fuori scena*** (nessuna interazione diretta).

3. **Casi d'uso**: rappresentano le funzionalità espresse dal sistema e rappresentate con un'ellisse ed espressi come verbi all'infinito.
4. **Relazioni** nel diagramma: relazioni come **associazione** (tra casi d'uso e attore), **generalizzazione**, **inclusione** ed **estensione** sono ampiamente presenti nel diagramma.
5. **Descrizione testuale**: i vari elementi nel diagramma devono essere accompagnati da una descrizione testuale (anche informale) contenente nome ed ID, descrizione, attori coinvolti, pre e post condizioni e sequenze alternative.

Diagramma delle Classi

Rappresentano la struttura statica del sistema mostrando le classi, i loro attributi, metodi e le relazioni tra le classi. Una classe ed un oggetto rispondono ad una relazione di dipendenza stereotipata (istanza). UML cerca di essere il più possibile generico rispetto a possibili linguaggi di implementazione.

Esistono due modelli principali:

- **Modello di dominio,** rappresentante la struttura e le relazioni tra le entità
- **Modello di progetto**, che rappresenta la struttura ed il funzionamento del sistema.

Esistono anche due tipi di modello principali che sono il **diagramma delle classi di analisi** ed il **diagramma delle classi di progetto**, rispettivamente parte dei modelli di dominio e di progetto.

- Diagramma delle classi di **analisi**: si concentra sulla modellazione dei concetti e relazioni del problema nelle prime fasi dello sviluppo. Sono utili a **comprendere e documentare i requisiti** per poi passare alla progettazione dettagliata. Si va a specificare il **cosa deve fare**.
- Diagramma delle classi di **progetto**: fornisce una panoramica visiva delle classi, attributi, metodi, nonché le relazioni tra di esse. Specifica quindi il **come deve farlo.**

Nel modello di dominio si descrivono i **concetti del dominio del problema**, mettendo in evidenza relazioni e responsabilità delle classi, senza dettagli di implementazione, concentrandosi sul cosa deve fare il sistema.
Il modello di progetto, invece, **deriva dal modello di dominio** e lo traduce in una soluzione tecnica, specificando in modo dettagliato attributi e metodi delle classi, concentrandosi sul come il sistema realizza le funzionalità.

Classi di associazione

Rappresenta una classe che **incapsula l'associazione tra due o più classi**. Tornano utili nel momento in cui l'associazione tra classi diventa troppo complicata da gestire direttamente.

Ad esempio, una classe denominata Student rappresenta uno studente e dispone di un'associazione con una classe denominata Course, che rappresenta un corso didattico. La classe Student può essere registrata in un corso. Una classe di associazione denominata Enrollment definisce ulteriormente la relazione tra le classi Student e Course fornendo informazioni su sezione, grado e semestre relative alla relazione di associazione

Diagramma di Sequenza

Servono a mostrare la **sequenza temporale** degli avvenimenti per ogni entità di programma. Viene utilizzato quindi per **descrivere uno scenario,** ovvero una determinata sequenza di azioni dove tutte le scelte sono già state effettuate.
Il diagramma di sequenza descrive le relazioni che intercorrono, in termini di messaggi, tra Attori, Oggetti di business, Oggetti o Entità del sistema che si sta rappresentando.

Gli elementi principali sono:

- **Lifeline**: servono a rappresentare un elemento di una classe all'interno di un'interazione. (Oggetti, Attori, Entità, **Boundary,** Controllo e Focus di Attivazione).
- **Messaggi**: rappresentano il tipo di interazione tra due linee di vita. Ci sono diversi tipi di messaggi come **Sincroni** (il mittente aspetta una risposta), **Asincroni** (invia altri messaggi senza aspettare risposta dal destinatario), di **ritorno**, di **creazione**, di **distruzione.**
- **I frammenti combinati** nei diagrammi di sequenza servono a rappresentare in modo compatto le strutture di controllo del flusso dei messaggi. Permettono di modellare condizioni, alternative e cicli all'interno di un unico diagramma, evitando di frammentare la descrizione del comportamento.

Diagramma di Attività

Sono utilizzati per modellare i flussi di controllo ed il comportamento sequenziale di un sistema. Vanno quindi a rappresentare il **comportamento dinamico del sistema**. Modellano un processo come **un'attività** costituita da un insieme di nodi connessi da archi, ovvero descrivono *il livello più elevato di funzionalità.*
Un'azione rappresenta un'entità discreta di funzionalità in un'attività. Esistono tre tipi di nodo: Nodi azione, Nodi di controllo e Nodi di oggetto.
Nodo Azione: esegue un'azione quando arrivano tutti i token richiesti e le condizioni sono soddisfatte; produce token in uscita al termine.

Nodo di controllo: gestisce il flusso di esecuzione e l'ordine delle azioni.

Nodo di oggetto: rappresenta dati o oggetti usati o prodotti dalle azioni, indicando il loro stato.

Gli **archi** possono essere o di **flusso di controllo** (linea intera da un'azione all'altra) oppure flusso **di oggetti** (segmentata, utilizzata per gli oggetti) .

I cinque principi SOLID

1. S — Single Responsibility Principle (SRP)

Una classe dovrebbe avere **una sola ragione per cambiare**.
In altre parole, deve avere **una singola responsabilità** o un singolo asse di variazione.

2. O — Open/Closed Principle (OCP)

Le entità software dovrebbero essere **aperte all'estensione** ma **chiuse alla modifica**.
Si estende il comportamento senza toccare il codice esistente.

3. L — Liskov Substitution Principle (LSP)

Le sottoclassi devono poter **sostituire** le loro superclassi **senza alterare la correttezza del programma**.
È un principio di sostituibilità comportamentale.

4. I — Interface Segregation Principle (ISP)

È meglio avere **molte interfacce specifiche** piuttosto che una singola interfaccia generale e pesante.
I client non dovrebbero essere costretti a dipendere da metodi che non usano.

5. D — Dependency Inversion Principle (DIP)

I moduli di alto livello non dovrebbero dipendere da quelli di basso livello: **entrambi devono dipendere da astrazioni**.
Le astrazioni non devono dipendere dai dettagli; i dettagli devono dipendere dalle astrazioni.

APPUNTI NON RIASSUNTI PER L'ESAME

- **classi di analisi (modello di dominio)**: è una rappresentazione UML che si concentra sulla modellazione dei concetti e delle relazioni nel dominio del problema durante le fasi iniziali dello sviluppo del software. Questi diagrammi sono utilizzati per comprendere e documentare i requisiti e il contesto del sistema, prima di passare alla progettazione dettagliata. Si occupa di specificare il cosa deve fare. Le classi di dominio rappresentano concetti reali o astratti presenti nel dominio del problema. Queste classi non includono dettagli di implementazione ma si concentrano su attributi e relazioni che sono rilevanti per comprendere il dominio.
- **Classi di progetto (modello di progetto)**: è una rappresentazione UML che rappresenta la struttura e le relazioni delle classi all'interno di un sistema software. Questo diagramma fornisce una panoramica visiva delle classi, dei loro attributi e dei metodi, nonché delle relazioni tra di esse. Si occupa di specificare il come deve farlo.
  - ***Le classi di progetto*** rappresentano le entità principali del sistema. Ogni classe è rappresentata da un rettangolo diviso in tre sezioni: il nome della classe nella parte superiore, gli attributi nella sezione centrale e i metodi nella parte inferiore.
  - Gli ***attributi*** sono le proprietà o i dati associati a una classe. Sono elencati nella sezione centrale del rettangolo della classe e possono includere il tipo di dato e la visibilità (pubblico, privato, protetto).
  - ***I metodi*** sono le azioni che una classe può eseguire. Sono elencati nella parte inferiore del rettangolo della classe e includono il nome del metodo, i parametri di input e il tipo di ritorno.
  - ***Le relazioni*** tra le classi sono rappresentate da linee che collegano le classi coinvolte. Le relazioni possono essere di vario tipo, come associazioni, aggregazioni, composizioni o generalizzazioni.

Nel modello di dominio, quindi, si rappresentano le relazioni e le responsabilità delle classi, mentre nel modello di progetto si rappresentano più dettagliatamente i metodi relativi alle responsabilità della classe. Il modello di dominio fornisce una base concettuale per il modello di progetto. Il modello di progetto, a sua volta, prende il modello di dominio come punto di partenza e lo traduce in una soluzione tecnica specifica. In altre parole, il modello di dominio informa il modello di progetto, guidando le decisioni di progettazione e implementazione del sistema.

I **GRASP (General Responsibility Assignment Software Patterns)** sono un insieme di principi che guidano l'assegnazione delle **responsabilità agli oggetti** durante la progettazione orientata agli oggetti. Essi si basano sull'approccio della **Responsibility Driven Development (RDD)**, che consiste nel pensare il sistema in termini di responsabilità, ruoli e collaborazioni tra oggetti.

Nella RDD, le responsabilità di un oggetto si dividono in due categorie principali:

- **Fare**, cioè eseguire azioni, delegare attività ad altri oggetti o coordinarne il comportamento;
- **Conoscere**, cioè conoscere i propri dati, gli oggetti correlati e le informazioni che può derivare o calcolare.

Le responsabilità possono essere realizzate da un singolo oggetto oppure tramite la **collaborazione** tra più oggetti. L'approccio RDD è iterativo: si individuano le responsabilità, si assegna ciascuna a una classe adeguata e si valutano eventuali collaborazioni necessarie.

Creator

Il pattern **Creator** risponde al problema di stabilire **chi debba creare un oggetto**.
La responsabilità di creare un'istanza della classe A viene assegnata a una classe B se B contiene, aggrega, registra o utilizza strettamente oggetti di tipo A, oppure se possiede i dati necessari alla loro inizializzazione.
Questo pattern favorisce **basso accoppiamento**, poiché la classe creatrice è già naturalmente legata all'oggetto creato.

Information Expert

Il pattern **Information Expert** fornisce un principio generale per l'assegnazione delle responsabilità: una responsabilità va assegnata alla classe che possiede le **informazioni necessarie** per svolgerla.
Questo favorisce **incapsulamento** e **alta coesione**, anche se in alcuni casi le informazioni possono essere distribuite su più classi, rendendo necessaria la collaborazione tra oggetti. È importante non abusarne per funzionalità trasversali.

Controller

Il pattern **Controller** risponde alla domanda su **quale sia il primo oggetto, oltre alla UI, a ricevere e coordinare un'operazione di sistema**.
La responsabilità viene assegnata a una classe che rappresenta il sistema nel suo complesso (controller di tipo *facade*) oppure a una classe che gestisce uno specifico caso d'uso (ad esempio *<UseCaseName>Handler*).
Il controller coordina le attività e delega il lavoro agli altri oggetti, favorendo il **disaccoppiamento dalla UI**. È importante evitare controller troppo "gonfi".