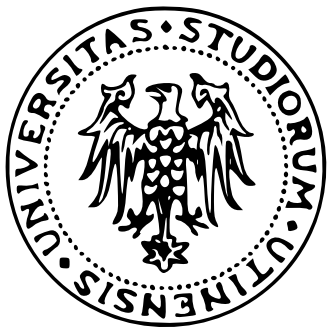
Università degli Studi di Udine

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

IBML - Internet of Things, Big Data, Machine Learning



Riassunto di Ingegneria del Software

Corso e materiale del docente: Vincenzo Riccio

Realizzato da: Alessio Nicodemo

A.A. 2025/2026

Indice

1 Concetti fondamentali

1.1 Cos'è il software?

Il **software** è un termine che indica l'insieme dei programmi per computer e la relativa documentazione (modelli di progetto, manuali utente, siti web di supporto, ...).

Le economie di tutte le nazioni industrializzate dipendono dal software e sempre più sistemi sono controllati dal software (es. sistema sanitario).

1.1.1 Tipologie di software

Software generico

- Software prodotto autonomamente da un'organizzazione per soddisfare le necessità di vari clienti (ad esempio, applicazioni per smartphone disponibili negli store ufficiali).
- Il produttore ha il controllo completo sulle specifiche del software.

Software su richiesta

- Software sviluppato da un'organizzazione su commissione di un cliente specifico (ad esempio, sistemi di controllo per dispositivi elettronici).
- Il produttore deve attenersi alle specifiche fornite dal cliente.

1.1.2 Complessità e costi del software

Il software può diventare estremamente **complesso**, **difficile** da capire e **costoso** da modificare (es. in presenza di difetti). La crescita della complessità richiede sempre **meno tempo** per lo sviluppo software. Il mercato in continua evoluzione richiede un rilascio rapido dei sistemi. Questo miglioramento è legato all'applicazione di approcci dell'**Ingegneria del Software**.

1.1.3 Sviluppo Professionale di software di qualità

Software personale

- Software prodotto per se stessi, difficilmente riusato in futuro da altri utenti.
- Non necessario scrivere una documentazione.
- Non necessario raggiungere alti livelli di qualità basta che funzioni.

Software professionale

- Software sarà usato da altre persone diverse dai propri sviluppatori.
- Sviluppo in team e probabilmente il software sarà utilizzato anche da altri sviluppatori.
- Il software deve possedere caratteristiche di qualità.

1.1.4 Qualità del software

Un **software di qualità** non deve solo fornire le funzionalità richieste, ma anche garantire prestazioni adeguate e conformi alle aspettative dell'utente.

Caratteristiche qualitative principali

- **Accettabilità**: deve essere accettato dagli utenti, risultando comprensibile, usabile e compatibile con gli altri sistemi utilizzati.
- **Fidatezza e protezione**: non deve causare danni fisici o economici in caso di malfunzionamento e deve essere sicuro contro accessi o modifiche non autorizzate.
- **Efficienza**: ottimizzazione delle risorse del sistema, evitando sprechi.
- **Mantenibilità**: deve poter evolvere nel tempo per rispondere alle nuove esigenze dei clienti o del mercato.

La qualità è relativa: le sue metriche dipendono dall'**applicazione/contesto**.

1.2 Ingegneria del software

La **Crisi del Software** venne discussa per la prima volta nel **1969** in Germania.

Motivazioni della crisi

- I software risultavano troppo **complessi**, **costosi**, **inaffidabili** e **rilasciati in ritardo**.
- I **costi del software** iniziarono a superare quelli dell'**hardware**.
- I **costi di manutenzione** divennero maggiori rispetto a quelli di sviluppo.

Obbiettivo: sviluppare software **facilmente manutenibile** e **vantaggioso nei costi**.
Soluzione: adottare approcci ingegneristici alla produzione del software.

L'**ingegneria del software** è una disciplina ingegneristica che riguarda tutti gli aspetti della produzione del software.

1.2.1 Ingegneria del Software: tra Ingegneria e Informatica

L'**Ingegneria del Software (IS)** è una disciplina ingegneristica che integra le competenze informatiche per affrontare i problemi pratici della produzione di software. Supporta lo sviluppo di software professionale, occupandosi di tutte le fasi della produzione, dalla **specifica** e **progettazione** fino a **evoluzione** e **manutenzione** dei programmi in uso.

A differenza della semplice programmazione, l'IS richiede il rispetto di vincoli organizzativi e di budget, combinando un approccio **sistematico e organizzato** (processo software) con **capacità creative** nella scelta dei metodi più adatti tra le possibili alternative. Spesso sono necessari compromessi per conciliare vincoli tecnici, temporali ed economici.

1.3 Processi Software

I **processi software** sono un insieme di attività che porta alla creazione o all'evoluzione di un prodotto software. Ciascun processo è composto da attività fondamentali e comprendono:

- **Acquisizione, analisi e specifica dei requisiti**: definizione di funzionalità e vincoli operativi del SW.
- **Progettazione e sviluppo**: progettazione e programmazione del SW.
- **Verifica e validazione**: controllo che il software soddisfi le richieste del cliente.
- **Evoluzione**: adattamento del software ai cambiamenti dei requisiti o del mercato.

Lo **studio di fattibilità** può essere un altra attività fondamentale, preliminare al processo ed è rapida e poco costosa. Valuta la **fattibilità tecnica**, **economica** e **di mercato** del progetto.

No Free Lunch : Non esiste una soluzione universale per tutti i tipi di software; ogni progetto richiede un processo specifico, pur condividendo le attività fondamentali.

1.3.1 Metodi e strumenti

- **Metodi**: approcci strutturati per sviluppare software di qualità rispettando tempi e costi.
- **Strumenti**: software di supporto alle attività del processo (analisi, modellazione, debugging, testing, ecc.).

1.3.2 Sfide dei processi software

- **Diversità**: adattamento a dispositivi e piattaforme eterogenee.
- **Consegna**: rilascio rapido in risposta ai cambiamenti tecnologici.
- **Fiducia**: garantire affidabilità e protezione dei dati.
- **Scala**: distribuire il software su sistemi diversi mantenendo qualità e prestazioni.

Principi fondamentali

- Metodi, tecniche e strumenti variano in base a software, all'organizzazione e al team.
- Non esistono metodi universali applicabili a tutti i software o contesti.
- I concetti fondamentali sono indipendenti dal linguaggio di programmazione.

Il processo deve essere **chiaro**, ossia ben definito e compreso da tutti i membri del team. Deve risultare **affidabile**, sicuro e usare le risorse in modo efficiente.
Per sviluppare sistemi utili, rispettando tempi e costi, è essenziale comprendere i **requisiti** e **riutilizzare** i componenti esistenti.

Modello di processo software : Rappresentazione semplificata e astratta che descrive l'intero ciclo di vita del software.

2 Processi software: attività fondamentali

2.1 Code and fix

Approccio alla produzione di software che consiste nello **scrivere codice e aggiustarlo** per correggere errori, migliorare e/o raggiungere funzionalità. **Limitato** in sistemi di grandi dimensioni a cui lavorano team numerosi, rendendo necessari metodi di sviluppo più organizzati.

2.2 Acquisizione, analisi e specifica dei requisiti

Questa attività stabilisce **cosa** il software dovrà fare, senza definire **come** dovrà farlo. Serve a identificare e descrivere le funzionalità e le qualità richieste, tramite interazione con il committente. Le decisioni prese non devono vincolare le fasi successive di progettazione e implementazione. Errori in questa fase comportano alti costi di correzione nelle fasi successive.

Ingegneria dei requisiti Disciplina che sviluppa metodi per raccogliere, documentare, classificare e analizzare i requisiti. Comprende tre sotto-attività principali:

1. **Deduzione e analisi dei requisiti:** comprensione delle necessità e delle aspettative degli *stakeholder*, tramite osservazione di sistemi esistenti, discussioni e sviluppo di modelli o prototipi per chiarire i bisogni.
2. **Specifica dei requisiti:** traduzione delle informazioni raccolte in un insieme strutturato di requisiti. Si distinguono:
   - **Requisiti utente**: proposizioni astratte rivolte a clienti e utenti finali;
   - **Requisiti di sistema**: descrizioni dettagliate di funzionalità e caratteristiche per gli sviluppatori.
3. **Convalida dei requisiti:** verifica che i requisiti siano realistici, coerenti e completi. Eventuali errori portano a revisioni del documento dei requisiti.

Documento dei requisiti Al termine della convalida si produce un documento che:

- definisce l'insieme dei requisiti del sistema;
- deve essere chiaro, preciso, coerente, non ambiguo e modificabile;
- può variare in dettaglio e formalità a seconda del processo adottato;
- include un piano di test preliminare del sistema.

Le attività di deduzione, specifica e convalida sono **intrecciate**: l'analisi dei requisiti può continuare anche durante lo sviluppo, specialmente nei processi **agili**, evolvendo nel tempo.

2.3 Progettazione e sviluppo

Questa attività converte le specifiche dei requisiti in un **sistema eseguibile** da consegnare al cliente. Si compone di due fasi principali:

- **Progettazione**: definizione della struttura del software che realizzi le specifiche, in modo formale o informale.
- **Sviluppo**: implementazione dei componenti definiti nel progetto.

Nei **processi agili**, progettazione e sviluppo sono spesso intrecciati, mentre nei **processi plan-driven** (ad es. per software critici) lo sviluppo segue rigidamente la progettazione.

2.3.1 Fase di progettazione

Le attività di progettazione sono **interdipendenti** e si sviluppano per iterazioni successive, aggiungendo dettagli o correggendo difetti. Il progetto evolve nel tempo: nuove informazioni possono modificare scelte progettuali precedenti.

Progettazione della piattaforma Il software deve integrarsi con la **piattaforma software** in cui verrà eseguito (es. sistema operativo, database, altre applicazioni). Le informazioni sulla piattaforma sono essenziali per definire come integrare il prodotto nell'ambiente operativo.

Progettazione dell'architettura Identifica la **struttura complessiva del sistema**, i componenti principali e le loro relazioni. Rappresenta il **progetto di alto livello** del software.

Progettazione delle interfacce Definisce come i componenti comunicano tra loro, senza esporre i dettagli interni di implementazione. Le specifiche delle interfacce devono essere **chiare e non ambigue**. Consente lo sviluppo separato dei componenti rispettando la compatibilità.

Progettazione dei componenti Si riusano componenti esistenti quando possibile, altrimenti vengono progettati nuovi componenti. La descrizione progettuale lascia spesso ai programmatori libertà nei dettagli dell'implementazione. Rappresenta il **progetto di dettaglio**.

Risultato della progettazione Il risultato di questa fase è un **progetto del software**, più o meno formale, che descrive:

- la struttura del sistema da implementare;
- i modelli e le strutture di dati utilizzati;
- le interfacce tra i componenti.

2.3.2 Fase di sviluppo

Consiste nell'implementazione del progetto attraverso la programmazione dei componenti definiti. È un'attività fortemente dipendente dalle competenze e dallo stile dei singoli programmatori, quindi non segue processi rigidi o standard universali. Tuttavia, nelle organizzazioni professionali, può essere regolata da **standard aziendali**, convenzioni di codifica e pratiche condivise per garantire coerenza, leggibilità e qualità del codice.

2.4 Verifica e validazione (V&V)

La fase di **verifica e validazione** ha l'obiettivo di dimostrare che il software:

- **Verifica:** rispetta le specifiche stabilite;
- **Validazione:** soddisfa le aspettative e le necessità del cliente.

Le attività di V&V possono includere ispezioni, revisioni e test, eseguite in diversi stadi del processo, dalla definizione dei requisiti fino all'implementazione. La tecnica più comune è il **testing**, che consiste nell'eseguire il sistema utilizzando dati di prova derivati dalle specifiche.

Test dei componenti Il **componente** è l'unità fondamentale del software (funzione, classe, modulo o gruppo di essi). Ogni componente viene testato **isolatamente**, per verificare il corretto funzionamento indipendentemente dagli altri.

Test del sistema Mira a testare il **sistema completo**, ma nei progetti complessi può essere necessario integrare progressivamente i componenti in sottosistemi. Serve a verificare la **conformità ai requisiti funzionali e non funzionali**. Può rilevare malfunzionamenti dovuti a interazioni impreviste tra componenti corretti.

Test del cliente Il sistema è testato con **dati reali del cliente**, non simulati. Permette di individuare eventuali problemi nei requisiti o prestazioni insoddisfacenti. Determina se il software **soddisfa effettivamente le esigenze del cliente**.

Testing iterativo E' un processo **iterativo**: implica la ripetizione delle fasi di test quando si scoprono difetti, poiché alcuni errori emergono solo durante l'integrazione o l'uso reale.

2.5 Evoluzione

L'attività di **evoluzione** riguarda la modifica e l'aggiornamento continuo del software dopo il suo rilascio. Oggi, sempre meno sistemi software sono sviluppati da zero: la maggior parte dei prodotti è soggetta a riuso, manutenzione ed estensione nel tempo.

- Il software viene modificato per adattarsi ai **cambiamenti dei requisiti** o dell'ambiente operativo.
- È la **fase più lunga** del ciclo di vita del software.
- Le attività di evoluzione includono:
  - **Correzione** di difetti non rilevati in precedenza;
  - **Miglioramento** della qualità o delle prestazioni;
  - **Adattamento** del sistema a nuovi contesti, tecnologie o esigenze del mercato.

Questa fase garantisce la **sopravvivenza e la rilevanza** del software nel tempo, mantenendolo allineato ai bisogni degli utenti e alle evoluzioni tecnologiche.

3 Modelli di processi software

Il **modello di processo SW** o **modello del ciclo di vita del SW**, è una caratterizzazione descrittiva o prescrittiva di come un sistema SW viene o dovrebbe essere sviluppato.

Si tratta di **descrizioni astratte di alto livello** che possono essere utilizzate per spiegare i diversi approcci allo sviluppo e che forniscono strutture di processo da estendere e adattare per creare processi concreti e specifici adatti a un particolare progetto o contesto organizzativo.

3.1 Punti di vista del modello

Un processo software può essere descritto da diversi **punti di vista**:

- **Architetturale**: mostra sequenza e organizzazione delle attività (approccio del corso);
- **Data-flow**: evidenzia le trasformazioni dei dati effettuate dalle diverse attività;
- **Role/action**: descrive i ruoli coinvolti nel processo e le rispettive responsabilità.

3.2 Descrizione del processo software

La descrizione di un processo include:

- le **attività** da svolgere e il loro ordine;
- i **prodotti** generati da ciascuna attività;
- i **ruoli** e le **responsabilità** delle persone coinvolte;
- le **pre- e post-condizioni**, ovvero i criteri che devono essere soddisfatti prima e dopo ogni attività per consentire l'avanzamento del processo.

3.3 Scelta del modello di processo

Esistono diversi modelli di processo, ognuno adatto a differenti tipologie di software e contesti organizzativi. In generale, si distinguono due grandi categorie:

- **Modelli plan-driven**: caratterizzati da pianificazione e documentazione rigorosa;
- **Modelli agili**: basati su iterazioni brevi, adattabilità e collaborazione.

Nella pratica, per lo sviluppo di sistemi complessi, è spesso necessario un **compromesso** tra approcci plan-driven e agili.

3.4 Modello a cascata (Waterfall Model)

Il **modello a cascata** è un processo **plan-driven** e **document-centric**, in cui le fasi di sviluppo sono eseguite in sequenza e ogni fase produce documenti che costituiscono gli **input** della successiva. Tutte le attività devono essere pianificate prima dell'avvio del progetto, e la fase successiva inizia solo dopo il completamento e l'approvazione della precedente.

Caratteristiche principali

- Ogni fase riflette direttamente una delle **attività fondamentali** di sviluppo (requisiti, progettazione, implementazione, verifica, manutenzione).
- Gli output di una fase vengono "**congelati**": non sono modificabili se non attraverso un processo formale di revisione.
- La fine di ogni fase rappresenta una **milestone**, utile a monitorare l'avanzamento del progetto.

- È un processo **monolitico**: il cliente può vedere il prodotto al termine di tutte le fasi.
- Nella pratica, sono necessari feedback tra fasi, anche se il modello è concettualmente lineare.

3.4.1 Vantaggi e svantaggi in generale

Vantaggi:
- Fasi e obiettivi chiaramente definiti.
- Output ben identificati e verificabili.

Svantaggi:
- Richiede **requisiti stabili e completi** fin dall'inizio, condizione spesso irrealistica nello sviluppo software moderno.
- Genera eccessiva documentazione rispetto alle reali necessità.
- **Poco flessibile**: difficile gestire modifiche o nuove richieste durante lo sviluppo.
- Gli errori nei requisiti emergono solo alla fine, con costi di correzione elevati.
- Non adatto a progetti con requisiti variabili, piccoli team o contesti agili in cui la comunicazione è informale.

Applicabilità:
- Adatto a progetti che richiedono documentazione formale e controllo rigoroso.
- Adatto a progetti con requisiti NON variabili, grandi team o contesti lenti in cui la comunicazione è formale.

3.4.2 Modello a cascata con retroazione

Il **modello a cascata con retroazione** è una variante del modello a cascata tradizionale, che introduce **meccanismi di feedback** tra le fasi del processo. Nella pratica, infatti, le attività non sono perfettamente sequenziali, ma si **sovrappongono** e si scambiano informazioni.

Ogni fase include una verifica dei risultati prodotti, con la possibilità di ritornare alla fase precedente per correggere eventuali errori. I feedback permettono di **rilevare problemi in anticipo**, evitando di scoprirli solo al termine del processo. La correzione comporta la revisione dei documenti o degli output della fase precedente.

Questo modello **rende meno rigido** il modello a cascata classico, consentendo modifiche durante lo sviluppo, ma non è flessibile rispetto a cambiamenti significativi in qualunque momento del progetto. Viene usato quando si prevede una **limitata variabilità dei requisiti**.

3.4.3 Estensione del modello a cascata: Modello a V

Il **modello a V** estende il modello a cascata collegando ogni fase di **progettazione** con una corrispondente fase di **verifica e validazione (V&V)**.
- Le attività del ramo superiore (progettazione) sono associate a quelle del ramo inferiore (test e validazione).
- Per ogni fase di progetto, viene definito un **piano di test**, che guida le attività di V&V.
- La V&V sono eseguite da un **team indipendente** rispetto a quello di sviluppo.
- In caso di errore rilevato durante la V&V, si rieseguono le fasi di progetto collegate.
- Consente una **validazione anticipata dei requisiti**, riducendo il rischio di errori tardivi.

Il modello a V è particolarmente usato in ambiti che richiedono elevata **affidabilità e tracciabilità**, come ad esempio nello sviluppo di sistemi **automotive** o **aerospaziali**.

3.5 Modelli evolutivi

I **modelli evolutivi** sono adatti a contesti in cui i **requisiti non sono completamente chiari** fin dall'inizio del processo di sviluppo. Permettono di far evolvere il sistema progressivamente attraverso cicli successivi di sviluppo e validazione. I due principali modelli sono:

- **Modello a sviluppo/consegna incrementale** (due varianti) : sviluppo ciclico con il cliente, che evolve i requisiti iniziali fino al sistema finale;
- **Modello prototipale** : esplorazione dei requisiti attraverso prototipi, ideale quando questi sono inizialmente poco chiari

3.5.1 Modello a sviluppo incrementale

Il **modello a sviluppo incrementale** si basa sull'idea di creare una **versione iniziale** del software che implementa i requisiti fondamentali, esporla agli utenti o ai loro rappresentanti, e **perfezionarla attraverso incrementi successivi** fino al sistema finale.

- Le attività di **specifica**, **sviluppo** e **convalida** sono intrecciate, con rapidi feedback tra le varie fasi.
- Ogni incremento aggiunge nuove funzionalità e migliora le versioni precedenti.
- Solo la versione iniziale è in genere mostrata ai clienti, mentre le versioni intermedie vengono utilizzate per la verifica interna.
- L'**ultimo incremento** corrisponde alla versione finale rilasciata al cliente.

3.5.2 Modello a consegna incrementale

Nel **modello a consegna incrementale**, non è consegnato direttamente nella sua forma finale alla fine del progetto, alcuni incrementi vengono effettivamente **rilasciati e installati** presso i clienti durante lo sviluppo. Fornisce un **feedback realistico** sull'uso in ambiente operativo, ma richiede agli utenti **tempo sufficiente** per testare ogni incremento.

- Ogni incremento fornisce una parte delle funzionalità richieste.
- I requisiti utente sono classificati per **priorità**: quelli più importanti vengono implementati per primi.
- I requisiti relativi a un incremento sono **congelati** dopo la sua consegna, mentre quelli successivi possono evolvere.
- Le funzionalità comuni a più requisiti dovrebbero essere individuate e implementate precocemente.

3.5.3 Differenze principali tra consegna e sviluppo

- Nello **sviluppo incrementale**, solo la prima versione è valutata da rappresentanti del cliente in un ambiente di test.
- Nella **consegna incrementale**, invece, ciascun incremento può essere usato dagli utenti finali nel proprio ambiente operativo, fornendo un feedback più concreto e affidabile.

3.5.4 Vantaggi e svantaggi in generale

Vantaggi:

- Rapido feedback del cliente su versioni preliminari del software, rispetto a documenti su progetto.
- Possibilità di **adattare i requisiti** prima della consegna finale, riducendo i costi di modifica.
- Consegna anticipata di versioni funzionanti con le funzionalità fondamentali.
- I primi incrementi aiutano a chiarire e migliorare i requisiti successivi.
- Le funzionalità prioritarie vengono testate più approfonditamente.

Svantaggi:
- Scarsa **visibilità e documentazione** del processo, poiché è oneroso documentare ogni versione.
- Rischio di **architettura mal strutturata** a causa dei continui cambiamenti.

Applicabilità:
- Componenti di **piccole o medie dimensioni** (es. interfacce utente).
- Sistemi con **vita breve** o con requisiti soggetti a frequenti variazioni.

3.5.5 Modello Prototipale

Il **modello prototipale** prevede la realizzazione di una versione iniziale (*prototipo*) del sistema, o di una sua parte, sviluppata rapidamente per risparamiare il costo ed esplorare meglio i requisiti del cliente nelle fasi iniziali del processo di sviluppo.
- È un sistema **usa e getta**: dopo la validazione, deve essere scartato poiché non rappresenta una base adeguata per lo sviluppo finale.
- Realizzando le funzionalità richieste, potrebbe **non rispettare** aspetti fondamentali come le prestazioni o essere documentato correttamente.
- Sviluppato in tempi brevi e con costi contenuti, privilegia la rapidità rispetto alla qualità.
- È fondamentale definire **obiettivi chiari** per la prototipazione: in assenza di essi, utenti o management possono fraintendere la funzione del prototipo, compromettendone l'efficacia.
- Non tutte le funzionalità del sistema finale devono essere incluse nel prototipo; è consigliabile concentrarsi sulle aree con **requisiti incerti**, per ridurre costi e tempi.

Inoltre, per una valutazione corretta:
- Gli utenti devono essere opportunamente formati sull'uso del prototipo.
- Esiste un problema di **rappresentatività**: i valutatori possono non coincidere con gli utenti finali, e l'interazione con il prototipo può differire da quella con il sistema reale.

La prototipazione può essere integrata con altri modelli di ciclo di vita:
- Nel modello a cascata, può essere impiegata durante la fase di progettazione per valutare **opzioni alternative**.
- Può essere utilizzata per **identificare, validare e raffinare i requisiti** prima dello sviluppo effettivo.

Infine, nella fase di **validazione**, si può applicare la tecnica di **Back-to-Back Testing**, confrontando il comportamento del sistema sviluppato con quello previsto dal prototipo iniziale, per verificare la coerenza tra le due versioni.

3.6 Modello Orientato al Riuso

Il **modello orientato al riuso** si basa sull'idea di sviluppare nuovi sistemi sfruttando componenti software o sistemi già esistenti, piuttosto che realizzare tutto da zero. Questo approccio è supportato da framework che facilitano l'integrazione dei componenti riutilizzabili.
Il riuso può riguardare:

- **Componenti software riutilizzabili**, progettati per essere facilmente integrati in più applicazioni.
- **Sistemi COTS** (*Commercial Off-The-Shelf*), ovvero prodotti software pronti all'uso e commercializzati da terzi.

Il processo si articola generalmente nelle seguenti fasi:

1. **Definizione dei requisiti essenziali**, descritti in modo sintetico e non dettagliato.
2. **Ricerca e valutazione** dei componenti o sistemi già esistenti per soddisfare tali requisiti.
3. **Perfezionamento dei requisiti** sulla base dei componenti trovati e aggiornamento della specifica del sistema.
4. **Composizione del sistema**:
    - Se esiste un'applicazione pronta all'uso, viene configurata per creare il nuovo sistema.
    - Se non esiste, si integrano **componenti riutilizzabili e nuovi moduli** sviluppati appositamente.

3.6.1 Vantaggi e Svantaggi

Vantaggi

- Riduzione della quantità di software da sviluppare ex novo.
- Diminuzione dei costi e dei rischi di progetto.
- Maggiore velocità nella consegna del prodotto finale.

Svantaggi

- Possibili **compromessi sui requisiti**: il sistema finale potrebbe non soddisfare completamente le esigenze degli utenti.
- **Limitato controllo sull'evoluzione dei componenti riutilizzati**, poiché aggiornamenti o nuove versioni dipendono da terze parti.

3.7 Modello Trasformazionale

Il **modello trasformazionale** si basa sull'uso di **specifiche formali** per descrivere in modo preciso e non ambiguo i requisiti del sistema. Queste specifiche, espresse tramite linguaggi formali (es. specifiche algebriche o modelli di stato), vengono successivamente trasformate in codice eseguibile, mantenendo la correttezza e la verifica.

- Le specifiche sono definite formalmente durante la fase di analisi, garantendo una **comprensione chiara e priva di ambiguità** dei requisiti.
- È possibile applicare tecniche di **model checking** per verificare automaticamente la correttezza delle specifiche prima della loro trasformazione in codice.
- Le specifiche vengono progressivamente trasformate in descrizioni meno astratte e più dettagliate, fino ad ottenere specifiche di basso livello direttamente eseguibili.
- Le trasformazioni possono essere eseguite manualmente o con il supporto di appositi strumenti, eventualmente riutilizzando componenti già esistenti.

3.7.1 Vantaggi e svantaggi

Vantaggi:
- Alta affidabilità del software grazie alla verifica formale e alla trasformazione controllata.
- Verifica implicita della correttezza, riducendo la necessità di test successivi.

Svantaggi:
- Richiede **competenze avanzate** in linguaggi e metodi formali.
- Difficile specificare formalmente tutte le parti di un sistema complesso.
- Il cliente può avere difficoltà a comprendere e convalidare le specifiche formali.

Applicabilità:
- Non adatto per sistemi di grandi dimensioni o con requisiti poco strutturati.
- Utilizzato soprattutto per **componenti critiche**, in cui è necessario garantire la correttezza *by construction*.

4 Sviluppo agile del software

Molti progetti software hanno requisiti **incerti** o in **continua evoluzione**. I modelli plan-driven, basati su pianificazione rigorosa e molta documentazione, risultano spesso troppo **rigidi e pesanti**, aumentando il rischio di **ritardi o fallimenti**. Sono però indispensabili per lo sviluppo di sistemi critici (come quelli aeronautici), dove servono elevato coordinamento, processi di lunga durata e manutenzione estesa nel tempo. Tuttavia, questi modelli si adattano poco a contesti dinamici e alla necessità di rilasci rapidi, poiché introducono un eccessivo **overhead e poca flessibilità** ai cambiamenti dei requisiti.

Nel contesto competitivo attuale, la **velocità di sviluppo e consegna** del software è diventata il requisito più critico per:

- Adattarsi rapidamente alle esigenze di committenti e utenti;
- Mantenere la competitività rispetto ai prodotti concorrenti.

4.1 Metodi agili

Alla fine degli anni '90 emergono i **metodi agili**, nati con l'obiettivo di **ridurre radicalmente i tempi di consegna** dei prodotti software, introducendo:

- Risposta in modo **rapido e flessibile** ai cambiamenti;
- Favorire una **comunicazione efficace** tra tutti gli stakeholder;
- Coinvolgere il **cliente all'interno del team di lavoro** per ottenere feedback immediato.
- Realizzare una **consegna incrementale e continua** del software.

Le caratteristiche fondamentali dei metodi agili:

- **Documentazione minima:** il focus è sul codice più che sulla progettazione; non vengono prodotte specifiche dettagliate e l'overhead documentale è limitato.
- **Consegna rapida e incrementale:** il sistema viene sviluppato e rilasciato in *incrementi frequenti*. Gli stakeholder partecipano alla definizione e valutazione di ogni incremento, contribuendo alla pianificazione dei successivi.
- **Strumenti di supporto:** vengono impiegati strumenti per automatizzare parti del processo, come ad esempio i test.

4.1.1 Processi plan-driven e agili

Differenze tra i processi

Plan-driven: Tutte le attività sono pianificate in anticipo; l'avanzamento del progetto è misurato rispetto al piano. Le fasi del processo software sono ben distinte e gli output di ciascuna fase costituiscono input per quella successiva.

Agili: La pianificazione è **incrementale e continua**. È quindi più semplice modificare il processo in risposta a cambiamenti dei requisiti del cliente o del prodotto. Requisiti, progettazione e implementazione avvengono **in parallelo**.

Coesistenza tra i processi

Le pratiche *plan-driven* e *agili* possono coesistere nello stesso processo di sviluppo:

- I processi agili possono includere documentazione di progettazione o attività pianificate quando necessario, con l'obiettivo di migliorare la comunicazione e la comprensione, pur accettando che i documenti siano incompleti.

- I processi plan-driven possono essere organizzati in modo **incrementale**.

Per lo sviluppo di **grandi sistemi**, è necessario trovare un **compromesso** tra pianificazione rigorosa e flessibilità agile.

4.1.2 Principi Agili

Coinvolgimento del cliente: i clienti sono coinvolti in tutto il processo di sviluppo. Intervengono a ciascuna iterazione del prodotto:
- valutano e validano l'iterazione
- forniscono nuovi requisiti del sistema o propongono modifiche alle iterazioni proposte
- assegnano priorità a ciascun requisito richiesto

Accettare cambiamenti
- prodotto non pianificato rigidamente
- prevedere che i requisiti possono cambiare

Mantenere semplicità
- prodotto e processo di sviluppo devono essere il più semplice possibile
- quando possibile, lavorare attivamente per eliminare le complessità dal sistema

Sviluppo incrementale
- software sviluppato incrementalmente
- cliente specifica i requisiti da includere in ciascun incremento

Persone, non processi
- processo di sviluppo non dev'essere fortemente prescrittivo
- membri del team devono essere liberi di sviluppare il software secondo i loro metodi
- membri del team devono poter sviluppare il software secondo i loro metodi (evitare standard inutili)

4.1.3 Applicabilità dei metodi agili

- Per prodotti di piccoli o medie dimensioni, prodotti personalizzati per cui c'è un chiaro impegno del cliente nell'essere coinvolto nel processo di sviluppo.
- Prodotti dove ci sono pochi stakeholder e non bisogna rispettare rigidi regolamenti.
- Team fisicamente vicini: le comunicazioni informali e facilitate.

4.2 Tecniche Agili

4.2.1 Metodi Agili

Diverse proposte di metodi agili:
- Extreme Programming
- SCRUM
- Feature Driven Development
- Crystal
- DSDM

Ognuno di questi metodi propone un diverso processo, condividono gli stessi principi.

4.2.2 Extreme Programming (XP)

Extreme Programming (XP) è il metodo agile più conosciuto. Spinge le pratiche di sviluppo a un livello estremo e adotta un approccio fortemente iterativo, con piccoli e frequenti incrementi rilasciati al cliente.

Caratteristiche principali

- **Requisiti come storie utente**: ogni requisito è espresso come una *user story*, cioè uno scenario di utilizzo del software per ottenere un risultato specifico.
- **Task di sviluppo**: le storie vengono suddivise in *task* più semplici, che rappresentano le unità principali dell'implementazione.
- **Pianificazione flessibile**: è presente una pianificazione, ma non appesantita da documentazione eccessiva.
- **Pair programming**: i programmatori lavorano in coppia e scrivono test per ogni task prima di implementare il codice.
- **Testing continuo**: tutti i test devono essere superati prima dell'integrazione del nuovo codice nel sistema.
- **Coinvolgimento del cliente**: il cliente partecipa attivamente allo sviluppo, valida le release, fornisce nuovi requisiti e definisce i test di accettazione.

Principi di XP

- **Coinvolgimento del cliente**: un rappresentante del cliente è presente nel team (*on-site customer*).
- **Accettare i cambiamenti**: ogni task completato viene immediatamente integrato nel sistema.
- **Sviluppo incrementale**: frequenti rilasci che aggiungono nuove funzionalità in modo graduale.
- **Mantenere la semplicità**: il progetto deve essere il più semplice possibile e il codice migliorato costantemente tramite *refactoring*.
- **Persone, non processi**: enfasi sulla collaborazione, proprietà collettiva del codice, orari di lavoro sostenibili e rifiuto degli straordinari eccessivi.

4.2.3 Influenza dell'Extreme Programming

Nella sua forma originaria, è raramente adottato integralmente perché richiede un cambiamento radicale. Tuttavia, le sue pratiche chiave sono state integrate in altri metodi agili:

- Storie utente
- Refactoring
- Sviluppo preceduto dai test
- Pair programming

4.2.4 Storie Utente

Le **storie utente** (o *user stories*) descrivono i requisiti come scenari d'uso. Sono redatte da cliente e team su *story cards*, poi suddivise in *task* implementativi. Questi task guidano la pianificazione delle iterazioni e la stima dei costi.

Processo

- Cliente e team definiscono priorità, costi e criteri di accettazione per ciascuna storia.
- Le storie prioritarie vengono selezionate per la prossima versione del prodotto.

Pro

- Integrano requisiti e sviluppo, facilitando la gestione dei cambiamenti.
- Maggior coinvolgimento dell'utente grazie a storie comprensibili.

- Ordinabili per valore aziendale.
- Facile aggiungere, modificare o scartare storie in base a nuove esigenze.

Contro
- Copertura incompleta dei requisiti difficilmente verificabile.
- Possibile omissione di scenari da parte di clienti esperti.

4.2.5 Refactoring

Il **refactoring** consiste nel migliorare il codice senza modificarne le funzionalità. XP adotta il refactoring continuo per ridurre i costi di manutenzione futura e semplificare le modifiche.

Caratteristiche
- Il team cerca proattivamente aspetti migliorabili e li ottimizza immediatamente.
- Anche modifiche non urgenti possono essere effettuate per aumentare la qualità complessiva.
- Un codice più chiaro riduce la necessità di documentazione e semplifica gli aggiornamenti.

Pro
- Contrasta il deterioramento del codice tipico dello sviluppo incrementale.
- Esistono tool che automatizzano alcune operazioni di refactoring.

Contro
- A volte serve una revisione architetturale, più costosa del semplice refactoring.
- Occorre bilanciare il tempo tra sviluppo di nuove funzionalità e miglioramento del codice.

4.2.6 Sviluppo preceduto dai test

Nel metodo XP, il testing è centrale: il software viene testato dopo ogni modifica.

Caratteristiche fondamentali
- **Test-driven development (TDD)**: i test vengono scritti prima del codice e ne guidano l'implementazione.
- **Automazione**: i test sono automatizzati ed eseguiti a ogni rilascio.
- **Coinvolgimento del cliente**: partecipa alla definizione dei test di accettazione.

Test-Driven Development (TDD) I test, scritti come programmi eseguibili, simulano input e verificano output. Ogni nuova funzionalità è aggiunta solo se tutti i test (nuovi e preesistenti) vengono superati, garantendo stabilità e coerenza del sistema.

Automazione dei test L'automatizzazione semplifica la verifica continua e riduce i tempi di validazione, aumentando l'affidabilità del software.

Pro
- La scrittura dei test chiarisce i requisiti e riduce ambiguità.

Contro

- Richiede tempo e impegno costante da parte del cliente.
- I test devono essere mantenuti aggiornati.
- Copertura dei test non sempre completa.

4.2.7 Pair Programming

Nel **pair programming**, due sviluppatori lavorano insieme sulla stessa postazione. Ciò favorisce la revisione continua e la diffusione della conoscenza all'interno del team.

Pro

- Aumenta il senso di proprietà collettiva del codice.
- Ogni linea di codice è verificata da più persone.
- Riduce i rischi dovuti al turn-over del personale.

Contro

- Può risultare meno efficiente per sviluppatori esperti che lavorano individualmente.

4.2.8 SCRUM: Gestione Agile della Progettazione

Scrum è un framework per la gestione agile dei progetti, che offre una visibilità costante sull'avanzamento del lavoro all'interno del team.

Elementi principali

- **Product backlog**: lista delle funzionalità da realizzare, derivate dai requisiti o dalle storie utente.
- **Sprint**: ciclo di sviluppo con obiettivi e durata definiti; al termine di ogni sprint deve esserci un prodotto potenzialmente rilasciabile.
- **Product owner**: definisce le priorità e aggiorna continuamente il backlog.
- **Sprint backlog**: insieme dei task da realizzare durante lo sprint, assegnati dal team.
- **Daily meeting**: breve riunione giornaliera per analizzare i progressi e pianificare la giornata.
- **ScrumMaster**: garante del processo Scrum, modera gli incontri e rimuove gli ostacoli.

Ciclo di sviluppo

All'inizio di ogni sprint vengono stabilite le priorità e assegnati i compiti. Il team stima la propria velocità basandosi sulle iterazioni precedenti, per ottimizzare i tempi di consegna. Alla fine di ogni sprint si tengono due riunioni:

- **Sprint review**: per valutare il prodotto realizzato.
- **Retrospective**: per analizzare il processo e introdurre miglioramenti.

5 Tipologie di Requisiti

5.1 Definizione dei requisiti

Definire:

- quali esigenze del cliente il sistema deve fornire
- entro quali vincoli operativi

Se il sistema funziona correttamente ma non rispetta i vincoli, lo stesso non sarà di interesse per il cliente.

La definizione di requisito è ampia.

Descrizione di qualcosa che il sistema dovrà fare o di una proprietà o vinclo operativo che si desidera per il sistema.

Tale termine può indicare diverse tipologie di descrizione:

- sus

5.1.1 Ingegneria dei Requisiti

Necessaria un'ampia definizione poichè un requisito può avere vari scopi nella pratica.

L'**Ingegneria dei Requisiti** è il processo di ricerca, analisi, documentazione e verifica dei requisiti.

Questo processo, fatto dagli ingegneri dei requisiti, si occupa di stabilire le funzionalità del software, i vincoli operativi e i vincoli per lo sviluppo, tutti insieme cristallizzati in un'appropriata documentazione.

5.2 Tipi di Requisiti

5.2.1 Requisiti Utente

Requisiti Utente

- frasi in linguaggio naturale relative alle funzionalità che il sistema deve fornire e i suoi vincoli operativi
- generalmente sono di alto livello
- descritti usando linguaggio naturale e diagrammi, comprensibili a tutti gli utenti

5.2.2 Requisiti Sistema

Requisiti di Sistema

Requisiti dal punto di vista di chi il sistema lo deve realizzare.

- documento strutturato che fornisce una descrizione dettagliata delle funzionalità del sistema e dei vincoli operativi
- definisce cosa dovrà essere sviluppato
- può far parte del contratto tra cliente e sviluppatore

5.2.3 Lettori delle specifiche dei requisiti

I lettori dei requisiti utente non si occpuano del modo in cui il sistema sarà implementato e i lettori dei requisiti di sistema hanno bisogno di sapere precisamente cosa dovrà fare il sistema.

5.3 Requisiti funzionali e non funzionali

Requisiti Funzionali: descrivono cosa il sistema dovrebbe fare, come reagirà agli input in vari scenari di utilizzo.
Requisiti Non Funzionali: sono vincoli sulle funzionalità del sistema o vincoli sul processo di sviluppo, includono anche gli standard che devono essere rispettati.

5.4 Requisiti Funzionali

Descrivno le funzionalità che dovranno essere offerte dal sistema, possono essere espressi a due livelli di astrazione:

- **Requisiti funzionali utente**: descrizione ad lato livello su ciò che il sistema farà
- **Requisiti funzionali di sistema**: descrizione dettagliata delle funzionalità, compresi input, output ed eccezioni

5.4.1 Imprecisioni nei requisiti

Requisiti imprecisi, o ambigui, possono essere interpretati in modi diversi da diversi stakeholder.

5.4.2 Completezza e Consistenza dei requisiti

Le specifiche dei requisiti devono essere complete e consistenti:

- **Completezza**: tutti i requisiti richiesti dai clienti devono essere presenti
- **Consistenza**: i requisiti non devono avere definizioni contraddittorie o essere in conflitto

Facile commettere errori o omissioni, soprattutto per sistemi complessi e di grandi dimensioni.

5.5 Requisiti Non Funzionali

Non riguardano direttamente le funzionalità offerte dal sistema, definiscono le proprietà e i vincoli del sistema e i vincoli del processo di sviluppo.

- **Proprietà del sistema** ()
- **Vincoli del sistema** ()
- **Vincoli del processo di sviluppo** ()

Possono essere anche più critici dei requisiti funzionali, se non sono soddisfatti il sistema potrebbe rivelarsi inutilizzabile.
Risulta difficile identificare quali componenti di sistema implementano specifici requisiti non funzionali.
Possono influire sull'intera architettura del sistema e non sui singoli componenti.
Un singolo requisito non funzionale può generare numerosi requisiti funzionali.

5.5.1 Tipi di requisiti non funzionali

Requisiti del prodotto

- derivano dalle caratteristiche richieste al software
- specificano il comportamento del prodotto (usabilità, efficienza, prestazioni)

Requisiti organizzativi

- derivano da politiche e procedure dell'organizzazione che sviluppa il software e del cliente

Requisiti esterni

- tutti i requisiti derivano da fattori esterni al sistema e al suo processo di sviluppo

5.6 Verificabilità dei Requisiti

I requisiti non funzionali possono essere difficili da definire precisamente, quindi difficili da verificare.
Il cliente li specifica come **obiettivi** generici/vaghi.
I requisiti devono essere **verificabili**:

- bisognerebbe descrivere i requisiti non funzionali quantitativamente, in modo che possano essere verificati in maniera oggettiva
- il requisito deve contenere qualche misura oggettivamente verificabile

I requisiti non funzionali possono essere in contraddizione tra loro o con requisiti funzionali, specialmente in sistemi complessi, in questi casi è necessario trovare un compromesso (trade-off).

5.7 Requisiti di Dominio

Arrivano dal dominio applicativo specifico di utilizzo, provenienti dagli esperti del dominio (navale, aereo, ...), possono essere funzionali o non funzionali.
Problema: l'esperto di dominio potrebbe tralasciare delle ovvietà che per l'ingegnere, però, non lo sono. Ci dev'essere un passaggio di conoscenza adeguato.

5.7.1 Problemi dei Requisiti di Dominio

Comprensibilità

- tendenzialmente espressi in linguaggio specializzato del dominio
- possono far riferimento a concetti specifici del dominio
- potrebbe non essere immediatamente comprensibile agli ingegneri software

Esplicitazione

- gli specialisti del dominio conoscono così bene il dominio stesso, da lasciare fuori dai requisiti che a loro sembrano ovvie

5.8 Ingegneria dei Requisiti

L'ingegneria dei requisiti è formata da tre attività chiave:

1. **Deduzione e analisi dei requisiti**: comprensione dei requisiti tramite l'interazione con gli stakeholder
2. **Specifica dei requisiti**: traduzione dei requisiti in specifiche in un formato coerente
3. **Convalida dei requisiti**: controllo che i requisiti corrispondano alle richieste del cliente

5.8.1 Modello Sequenziale

Le attività non sono per forza sequenziali come mostrato nel diagramma.

5.8.2 Modello a Spirale

Sono a spirale perchè cresce la conoscenza, si abbassa il rischio e cresce il valore del software.
Possibile prima definire i requisiti in maniera generale, poi i requisiti utente e infine i requisiti di sistema.

6 Analisi, Specifica e Convalida dei Requisiti

6.1 Deduzione e Analisi dei Requisiti

Stakeholder: sono gli individui che hanno interesse nel progetto di sviluppo software, possono impattare sul successo o l'insuccesso del progetto.

Gli ingegneri devono interagire con gli stakeholder per scoprire informazioni sul dominio applicativo, sulle funzionalità che dovrebbe avere il sistema, sulle prestazioni ed altri vincoli operativi.

6.1.1 Difficoltà

Tendenzialmente nè fornitore nè committente sono in grado, da soli, di estrarre efficaciemente i requisiti di sistema:

- il committente non ha la necessaria conoscenza dei processi software per definire in maniera efficace i requisiti, può non avere un'idea chiara sui requisiti, può usare termini propri del dominio di appartenenza
- diversi stakeholder potrebbero avere requisiti contrastanti
- il fornitore non ha conoscenza perfetta del dominio applicativo, e non può esprimere le effettive necessità

6.1.2 Processo

Figura 1: Diagramma "Deduzione e Analisi dei Requisiti: Processo"

Processo iterativo che termina quando il documento dei requisiti è completo, la comprensione dei requisiti da parte dell'ingegnere migliora ad ogni iterazione.

1. **Scoperta e Comprensione**: gli analisti interagiscono con gli stakeholder per scoprire i loro requisiti.
2. **Classificazione e Organizzazione**: i requisiti scoperti sono una raccolta non strutturata, quelli tra loro correlati vanno raggruppati in gruppi coerenti, eliminando i duplicati.
3. **Negoziazione e priorità**: dare una priorità ai requisiti, trovare e risolvere i conflitti attraverso la negoziazione.
4. **Documentazione**: i requisiti vengono documentati e diventano l'input della successiva iterazione. Diversi livelli di documentazione a seconda del processo: bozze di documenti dei requisiti software, o informalmente su lavagne, wiki o spazi condivisi.

6.1.3 Tecniche per estrarre i requisiti

- **Interviste**: sia formali che informali, a risposta chiusa o aperta
- **Etnografia**: osservare e analizzare le persone nell'ambiente operativo
- **Storie Utente e Scenari**: testi narrativi che descrivono scenari pratici di utilizzo del software

6.1.4 Interviste

I team di ingegneria fanno domande (chiuse o aperte) agli stakeholder sul sistema che utilizzano e su ciò che dev'essere sviluppato al fine di comprendere le loro necessità e dalle loro risposte si ottengono i requisiti.

Sueggerimenti per le interviste per renderle più efficaci
Un prototipo può aiutare ad avere requisiti dettagliati.
Gli specialisti di dominio potrebbero usare termini specifici o omettere dettagli che considerano ovvi.
Dettagli organizzativi o politici potrebbero essere non rilevati a degli estranei, come dettagli aziendali.
L'intervistatore dev'essere open-minded, evitando di avere preconcetti durante l'intervista.
Evitare domande aperte generiche.

6.1.5 Etnografia

Un analista osserva l'ambiente operativo immergendosi in esso, osserva il lavoro quotidiano, prendendo nota dei compiti in cui i partecipanti sono coinvolti.
Scopre requisiti implitici che riflettono processi reali.
Datco che presenta un focus sugli utenti finali, andrebbe arricchita con altri metodi per requisiti ad alto livello.

Motivazioni
I sistemi non sono mai isolati, vengono usati in un contesto operativo assieme ad altri software e in un contesto sociale dove ogni persona lo utilizza in maniera differente.
Ci sono persone che possono avere problemi-inefficienze, o un basso livello di comunicatività, dove non riescono a spiegare bene cosa gli serve.

6.1.6 Storie e Scenari

Descrfivono come il sistema può essere utilizzato per svolgere particolari compiti, descrivono cosa fanno le persone, quali informazioni utilizzano e producono, e quali sistemi possono utilizzare questo processo.

- **Storie**: testi narrativi che presentano una descrizione di alto livello di come viene usato il sistema
- **Scenari**: informazioni specifiche, spesso strutturate e raccolte come input, output e flusso di eventi durante un'interazione con il sistema

Più persone possono mettersi in relazione con storie e scenari, permettendo la raccolta di informazioni da un pubblico più vasto.
Le persone si trovano più a loro agio riferendosi a esempi di vita reale.
Più semplice per una persona raccontare come vorrebbe che il software funzionasse.

6.2 Specifica dei Requisiti

Processo di descrizione dei requisiti utente e di sistema in un documento.

6.2.1 Specifica dei Requisiti Utente

I requisiti utente devono essere comprensibili a tutti gli utilizzatori del sistema, anche a quelli privi di conoscenze tecniche specifiche. Devono descrivere esclusivamente il comportamento del sistema visto dall'esterno e i suoi vincoli operativi.
La specifica dei requisiti utente non deve (o non dovrebbe) in alcun modo includere dettagli architetturali o di progettazione interna del sistema.
Sono tendenzialmente scritti in linguaggio naturale.

6.2.2 Specifica dei Requisiti Sistema

6.2.3 Specifica dei Requisiti VS Progetto

In teoria
- la specifica dei requisiti non dovrebbe contenere informazioni su progettazione o implementazione del sistema.
- il progetto deve descrivere come i requisiti sono realizzati.

In pratica
- requisiti e progetto sono inseparabili: non è possibile, né auspicabile, escludere tutte le informazioni sulla progettazione.

6.2.4 Linguaggi per la specifica

Possono essere espressi in linguaggio naturale (NL), ci sono però delle alternative:
- linguaggio naturale strutturato o semi-strutturato
- modelli grafici
- specifiche formali

6.2.5 Linguaggio Naturale (NL)

PRO
- espressivo, intuitivo e universale: può essere compreso da utenti e clienti.

CONTRO
- mancanza di chiarezza: difficile usare il linguaggio in modo conciso, e allo stesso tempo, preciso e non ambiguo.
- confusione: difficile distinguere le varie tipologie di requisiti. Inoltre, diversi requisiti potrebbero essere espressi in una singola frase.

LINEE GUIDA
- formato standard coerente e conciso, riduce il rischio di omissioni e semplifica il controllo dei requisiti.
- utilizzo coerente del linguaggio: "deve" per obbligatori, "dovrebbe" se desiderabili.
- formattazione coerente del testo per evidenziare i punti chiave di un requisito.
- evitare l'utilizzo del linguaggio tecnico.
- spiegare perchè un requisito è necessario e chi lo ha proposto, in modo da sapere chi consultare se il requisito dovrebbe essere modificato.

6.2.6 Specifiche Strutturate

L'utilizzo del linguaggio naturale con una struttura predefinita standard per tutti i requisiti, garantisce maggiore uniformità. Ogni elemento della struttura fornisce informazioni su un aspetto del requisito. Limita la libertà di chi scrive i requisiti, permette la stesura in maniera guidata. Possibilità di utilizzo di costrutti del linguaggio di programmazione (IF, FOR). Si può usare una formattazione per evidenziare i punti chiave di un requisito.

Specifiche Tabellari
Se bisogna specificare calcoli complessi, è difficile non introdurre ambiguità.
Possibile aggiungere informazioni supplementari al linguaggio naturale, come tabelle o modelli grafici del sistema.
Utili per la descrizione di situazioni alternative e azioni da intraprendere in ogni situazione.

VANTAGGI E SVANTAGGI
Vantaggi
- conserva l'espressività del linguaggio naturale.
- impone uniformità per descrivere le specifiche, riducendo variabilità.
- organizza i requisiti in modo efficace.

Svantaggi
- troppo rigido per descrivere alcuni tipi di requisiti che richiedono descrizioni più libere ed espressive.
- difficile scrivere i requisiti in modo non ambiguo quando sono molto complessi.

7 Introduzione UML

7.1 Motivazione

I sistemi software moderni tendono a crescere e a diventare sempre più complessi.
- è impensabile comprendere sistemi complessi direttamente dal codice.
- il codice spesso è incomprensibile.
- necessità di forme di rappresentazione più astratte per discutere le scelte di progetto.
- la modellazione è un modo per gestire la complessità del software.

7.2 Modellazione di un sistema

Processo che sviluppa modelli astratti di un sistema, non una rappresentazione alternativa:
- la rappresentazione di un sistema mantiene tutte le informazioni sull'entità che rappresenta.
- un'astrazione semplifica deliberatamente un sistema evidenziandone le caratteristiche salienti, eventualmente tralasciando altre caratteristiche.

Ad un sistema possono corrispondere più modelli:
- ogni modello rappresenta una differente vista o prospettiva del sistema.

Si possono utilizzare notazioni grafiche o notazioni matematiche

7.3 Terminologia

Modello: astrazione che descrive un sistema o sottosistema. **Vista (o prospettiva)**: descrizione di aspetti specifici di un sistema da una certa prospettiva, in cui si omettono dettagli non rilevanti per tale prospettiva **Notazione**: insieme di elementi grafici o testuali e regole per rappresentare le viste. Diverse viste/modelli possono sovrapporsi.

7.4 Linguaggi di Modellazione

Per descrivere i modelli si utilizzano linguaggi di modellazione, in passato, diversi linguaggi di modellazione erano usati da supporto delle metodologie che si applicavano nelle varie fasi del processo di sviluppo software. Negli ultimi anni il linguaggio UML si sta affermando come linguaggio unificato che possa essere utilizzato in tutte le attività di modellazione. Esistono anche altri linguaggi come SysML (variante per sistemi complessi, es. IoT)

7.4.1 Storia UML

Skippabile?

7.5 UML: Notazioni + Meta-Modello

UML è una famiglia di notazioni grafiche che si basano su un singolo meta-modello (modello che definisce i concetti stessi del linguaggio di modellazione, indica secondo quali regole sia possibile costruite modelli UML). UML non è una metodologia: ha l'obiettivo di fornire un supporto al processo di sviluppo software. Può essere usato all'interno dei processi di sviluppo che adottano le proprie metodologie.

7.6 Regole Prescrittive e Descrittive

Le regole UML possono essere considerate sia prescrittive che descrittive:

- regole **prescrittive**: regole stabilite da organismi standardizzati che definiscono precisamente lessico, sintassi e semantica del linguaggio. Fondamentali quando UML è usato come linguaggio di programmazione.
- regole **descrittive**: stabilite per convenzione comune, possono essere meglio comprese guardando come UML viene usato nella pratica da un'organizzazione.

Bisogna conoscere le convenzioni particolari della specifica organizzazione e del singolo progetto, anche se al di fuori dello standard.

7.7 UML

Secondo Fowler e Mellor, UML può essere usato:

1. **Come bozza (sketch)**: per tracciare un modello informale di sistema da realizzare o per descrivere un sistema esistente.
2. **Come progetto dettagliato (blueprint)**: per realizzare un modello completo della soluzione architetturale del sistema.
3. **Come linguaggio di programmazione**: in grado di modellare in maniera completa e precisa il sistema software (spesso associato a Model-Driven Architecture - MDA).

7.8 Bozze: Espressività > Completezza

Quando si realizza una bozza, lo scopo è aiutare la comunicazione e la discussione delle idee, esplorando le soluzioni alternative, i diagrammi non devono essere esaustivi e definire tutti gli aspetti del codice.
Le bozze sono disegnate in poco tempo e in modo collaborativo, non rispettando tutte le regole formali dello standard.

7.9 Progetto Dettagliato: Espressività + Completezza

Quando si realizza o si deve capire un progetto, lo scopo è aiutare la comprensione e la completezza.
Il progettista sviluppa un modello di progetto che lo sviluppatore dovrà realizzare, salvandolo in file condivisi.
La completezza e non ambiguità del modello aiutano il programmatore, guidato dal modello e non dovrà avere aspetti ambigui da interpretare.

7.10 UML come linguaggio di programmazione

L'approccio MDA (Model Driven Architecture) esplora la possibilità di usare UML come linguaggio di programmazione, si vorrebbe stabilire una sintassi e semantica precisa per UML che portino alla generazione di codice eseguibile rappresentativo del modello.
La sfida risiede nel modellare precisamente anche la logica del progetto, il vantaggio consiste nel generare codice per diverse piattaforme target da un modello indipendente dalla piattaforma.

7.11 Informazioni soppresse

L'assenza di qualche informazione in un diagramma UML non significa che tale informazione non esista, alcuni aspetti del problema potrebbero essere assenti da un diagramma perché non ancora trattati nella fase in cui è stato tracciato il diagramma.

7.12 Tipi di diagramma UML

UML 2 possiede 13 diversi tipi di diagrammi ufficiali, appartenenti a due categorie:

- diagrammi strutturali: modellano l'organizzazione del sistema.
- diagrammi comportamentali: modellano il comportamento e le interazioni tra le entità del sistema.

Questi diagrammi rappresentano i deliverables di diversi fasi del ciclo di vita del software, tra cui attività di analisi dei requisiti e attività di progettazione, sia di alto che di basso livello.

7.13 Prospettive UML

Prospettiva esterna: modellati in contesto operativo del sistema.
Prospettiva delle interazioni: sono modellate le interazioni tra il contesto e il sistema o tra diverse componenti del sistema.
Prospettiva strutturale: sono modellate l'organizzazione del sistema e/o la struttura dei dati.
Prospettiva comportamentale: sono modellati il comportamento dinamico del sistema e come esso risponde agli eventi.

7.14 Integrare UML nel processo di sviluppo

UML può essere usato in diverse fasi del processo di sviluppo

- analisi dei requisiti: facilita la deduzione dei requisiti, la notazione non dev'essere troppo complessa per favorire la comunicazione con il cliente.
- progettazione: modelli più tecnici e dettagliati per descrivere il sistema agli ingegneri che lo devono implementare.
- documentazione (dopo implementazione): modelli rendono più semplice la descrizione di parti complesse o convogliano messaggi in maniera intuitiva immediata.
- comprensione di software pre-esistente: evoluzione o reverse-engineering.

Nei processi iterativi ogni iterazione arricchisse i diagrammi delle iterazioni precedenti.

8 Diagrammi dei Casi d'Uso

8.0.1 Def. Diagramma Comportamentale

DEF. **Diagramma comportamentale** modella il comportamento esterno del sistema, senza specificare nel dettaglio come questo comportamento viene realizzato, nella fattispecie modella l'interazione tra il sistema e gli agenti esterni.

8.0.2 Diagramma dei casi d'uso nel processo software

Utilizzato nella raccolta dei requisiti, diagramma con tutti i requisiti funzionali del sistema, descrive le tipiche interazioni tra utenti e sistema. Viene adottato il punto di vista degli utenti, so solo cosa do e cosa mi torna il software.

8.0.3 Scenari e Casi d'Uso

- **Scenario**: sequenza di passi che caratterizzano una particolare interazione tra utente e sistema.
- **Caso d'uso**: insieme di scenari che hanno uno scopo finale dell'utente in comune.

Gli utenti sono **attori** nello scenario. Il caso d'uso è una tipica interazione tra attore e sistema per svolgere un'unità di lavoro utile, non rivela l'organizzazione interna del sistema. L'insieme dei casi d'uso rappresenta le funzionalità che il sistema offre agli attori. Il diagramma UML dei casi d'uso è comprensibile anche ai non addetti ai lavori. La descrizione di un caso d'uso specifica cosa fa il sistema in seguito ad uno stimolo. Lo stimolo può partire da un attore o anche dal sistema. Un caso d'uso corrisponde ad un compito: - che l'attore chiede al sistema di eseguire - che il sistema esegue autonomamente

8.1 Elementi dei Casi d'Uso

8.1.1 Subject (Confini del sistema)

Rappresenta il confine tra ciò che è all'interno e ciò che è all'esterno del sistema, quello che si trova all'interno andrà progettato, realizzato, verificato e validato.

8.1.2 Attore

Rappresenta un ruolo che l'utente del caso d'uso svolge nell'interagire con il sistema. Gli attorni sono sempre esterni al sistema. Un attore di un caso d'uso può essere:

- una classe di persone fisiche.
- altro sistema software.
- dispositivo hardware esterno.

Un attore di caso d'uso può essere:
- attore primario, se persegue lo scopo che il caso d'uso cerca di soddisfare.
- attore secondario, altri attori con cui il sistema interagisce per svolgere con successo il caso d'uso.

Un attore primario può fornire lo stimolo che avvia il caso d'uso o interagisce dopo che il caso d'uso è stato avviato.

8.1.3 Caso d'Uso

Rappresenta una sequenza di azioni che un sistema può eseguire interagendo con attori esterni, è un'unità di lavoro utile (elaborazione) che il sistema esegue dopo l'evento di innesco del caso d'uso:
- stimolato dall'attore primario per eseguire un compito che l'attore deve eseguire.
- il sistema può iniziare il caso d'uso e interagire con uno o più attori esterni per eseguire un compito.

8.2 Descrizione degli scenari

8.2.1 Scenari

Un caso d'uso è descritto tramite un insieme di scenari di interazione tra gli attori ed il sistema. Uno scenario è una sequenza di azioni/interazioni fra sistema e attori. Il focus è rivolto all'interazione, non alle attività interne del sistema. Uno scenario definisce cosa accade nel sistema in seguito all'evento di innesco:
- come e quando il caso d'uso inizia.
- chi inizia il caso d'uso.
- interazione tra attore e caso d'uso e cosa viene scambiato.
- come e quando c'è bisogno di dati memorizzati o di memorizzare i dati.
- come e quando il sistema d'uso termina.

Per ogni caso d'uso sono previsti scenari normali e scenari alternativi.

8.2.2 Stili di descrizione degli scenari

Non esiste un modo standard per descrivere il contenuto di un caso d'uso. Stile di descrizione:
- **testuali**: con flusso chiaro di eventi da seguire.
- **diagrammatici**: diagrammi UML di stato, sequenza, interazione.

8.2.3 Scenari come sequenze di passi

Ogni scenario può essere espresso come sequenza di passi numerati:
- ciascun passo corrisponde a un'interazione tra attore e sistema.
- il passo dev'essere espresso con una frase semplice che indichi chi lo sta eseguendo e qual è il suo intento, senza riportare dettagli tecnici sulle azioni.

8.2.4 Scenari principali e alternativi

Un caso d'uso è una collezione di scenari correlati in cui gli attori interagiscono con il sistema per raggiungere l'obiettivo:

- scenario principale di successo: descrive il flusso principale.
- percorsi alternativi, possono essere sia di successo che di insuccesso.

Il percorso alternativo riporta:

- il numero del passo in cui si discosta dallo scenario principale.
- la condizione che deve essere soddisfatta per scatenare tale percorso invece dello scenario principale.
- al suo termine in quale punto (numero di passo) rientra nel flusso principale.

8.2.5 Pre-Condizioni e Post-Condizioni

Oltre ai passi che compongono gli scenari, un caso d'uso può riportare le condizioni che si devono verificare prima e dopo il caso d'uso

- Pre-Condizioni: ciò di cui il sistema deve assicurarsi prima di eseguire il caso d'uso.
- Post-Condizioni: ciò che il sistema deve garantire al termine del caso d'uso.

8.2.6 Descrizione di un Caso d'Uso

Per ogni caso d'uso è opportuno documentare gli scenari, non esiste un formato standard della descrizione, ogni organizzazione definisce il proprio formato.

8.2.7 Uso di IF, WHILE e FOR negli scenari

Cicli WHILE o FOR possono essere usati per racchiudere gruppi di passi che devono essere ripetuti più volte (o in italiano per ogni e finchè). Le alternative possono essere descritte attraverso costrutti di selezione IF/ELSE (o in italiano, se e altrimenti).

8.2.8 Linee guida per la descrizione degli scenari

- Scrivere in stile essenziale, senza riferimenti all'implementazione.
- Descrivere casi d'uso concisi e completi.
- Descrivere casi d'uso a scatola nera.

Nella descrizione di un caso d'uso non devono essere indicati dettagli che rivelino le scelte di progetto del software:

- quando si pensa ai casi d'uso, non si è ancora affrontato alcun aspetto della progettazione.
- non possono esserci riferimenti a specifici file, a meno che essi non rappresentino dei vincoli.

8.3 Relazioni tra Attori e tra Casi d'Uso

8.3.1 Generalizzazione di Attori

L'attore specializzato conserva le proprietà del generale oltre a possedere sue caratteristiche particolari. La freccia parte dall'attore specializzato e punta all'attore generale. La generalizzazione permette di astrarre ruoli comuni a più attori e semplificare i diagrammi.

8.4 Relazioni tra Casi d'Uso

Tra i casi d'uso possono esistere relazioni di tipo:

- generalizzazione.
- inclusione.
- estensione.

Usate per strutturare ulteriormente un diagramma dei casi d'uso:

- generalizzando/specializzando un caso d'uso.
- estraendo comportamenti comuni tra casi d'uso e inclusi in più casi d'uso.
- distinguendo comportamenti alternativi rispetto al caso d'uso base, estendendo il caso d'uso base con un caso d'uso alternativo.

8.4.1 Generalizzazione tra Casi d'Uso

Il caso d'uso generale rappresenta diversi casi d'uso simili. Un caso d'uso specializzato eredita comportamento e significato del caso d'uso generale, fornendo i dettagli specifici dei casi d'uso simili Un caso d'uso specializzato può aggiungere passi o modificare il comportamento del generale

8.4.2 Inclusione tra Casi d'Uso

La relazione d'inclusione formalizza i casi in cui più casi d'uso includono una serie di azioni comuni. Il comportamento comune a più casi d'uso diventa un caso d'uso che è incluso nei casi d'uso di partenza. Il caso d'uso base è incompleto senza il caso incluso Rappresentato graficamente come una dipendenza stereotipata «include» che parte dal caso base e arriva al caso incluso. L'inclusione non contiene informazioni sull'ordine dei casi d'uso, il caso incluso è una sequenza di azioni che è eseguita una o più volte dai casi d'uso includenti. L'inclusione è simile a una chiamata a procedura: Include(TrovaDatiImpiegato)

8.4.3 Inclusione e Generalizzazione

Se un caso d'uso generale include un altro caso d'uso, tutte le sue specializzazioni "ereditano" tale inclusione.

8.4.4 Estensione dei Casi d'Uso

Modella una sequenza opzionale di eventi o casi eccezionali, ogni estensione definisce un nuovo caso d'uso che estende il caso di partenza e ne varia il comportamento normale. Nel caso d'uso esteso (base) si agganciano ad uno o più punti d'estensione (XP: eXtension Points), le condizioni che fanno scattare l'estensione. Rappresentato graficamente come una dipendenza stereotipata «extend» che parte dall'estensione e arriva al caso base.

L'estensione non contiene informazioni sull'ordine dei casi d'uso. Le estensioni potrebbero anche essere accessibili direttamente da un attore. In questo caso nel diagramma dei casi d'uso ci sarà un segmento di comunicazione tra l'attore e il caso d'uso esteso.

I casi d'uso di estensione aggiungono un comportamento in corrispondenza dei punti di estensione. Il caso d'uso base si può svolgere anche senza i casi d'uso d'estensione. Creando estensioni separate, la descrizione del caso base rimane semplice.

8.4.5 Estensioni VS Scenari alternativi: alternative modellistiche

L'esempio "EffettuaOrdine" si poteva anche risolvere usando un unico caso d'uso:

- la soluzione con tre casi d'uso è più utile nel caso in cui i casi d'uso estesi abbiano ulteriori legami e/o siano direttamente richiamabili dall'utente.
- la soluzione con un solo caso d'uso e più scenari fornisce una vista più compatta del sistema, e potrebbe essere preferibile se si vuole realizzare un modello dei casi d'uso meno dettagliato e più leggibile.

8.4.6 Errori tipici con i Casi d'Uso

Diagrammi troppo complessi con molti casi d'uso: i casi d'uso rappresentano sequenze di azioni, non una singola azione. Ripetere il nome dello stesso caso d'uso più volte nello stesso diagramma. Le frecce delle relazioni di estensione o inclusione non sono tratteggiate, etichettate con «extend» o «include», oppure nel verso sbagliato.

8.4.7 Requisiti Funzionali e Casi d'Uso

La modellazione dei casi d'uso è una tecnica di ingegneria dei requisiti

- Requisito funzionale: funzionalità richiesta dal committente.
- Caso d'uso: modalità di utilizzo del sistema da parte di un utente (attore).

Tracciabilità tra requisiti e casi d'uso è importante incrociare requisiti funzionali e casi d'uso per verificare la reciproca copertura: ogni requisito dev'essere coperto da almeno un caso d'uso e viceversa, questa informazione può essere riportata nella **matrice di traccibilità**.

8.4.8 Prodotto Finale

L'analisi dei casi d'uso produce:

- un diagramma dei casi d'uso.
- le descrizioni di tutti gli scenari di tutti i casi d'uso.

Nel diagramma è contenuto solo un piccolo sottoinsieme delle informazioni contenute nelle descrizioni degli scenari, tutte le informazioni contenute nel diagramma sono contenute anche nelle descrizioni degli scenari. Il diagramma dei casi d'uso non dovrebbe mai essere considerato separatamente dalle descrizioni degli scenari.

8.5 Suggerimenti per la costruzione del Diagramma dei Casi d'Uso

1. Definisci i confini del sistema.
2. Identifica gli attori.
3. Identifica i casi d'uso.
4. Definisci il diagramma.
5. Descrivi i casi d'uso.
6. Struttura i casi d'uso.

8.5.1 Definisci i Confini

Quali responsabilità rientrano nei confini del sistema che stiamo modellando? Esempio: "Pagamento alla cassa automatica"

8.5.2 Identifica Attori

Identifica gli attori che interagiscono con il sistema per eseguire qualche compito
- identifica gli attori che necessitano del sistema per svolgere qualche compito.
- identifica gli attori cui il sistema si rivolge per svolgere qualche compito.

Raggruppa le persone identificate secondo i loro ruoli rispetto al sistema Identifica altri sistemi software e dispositivi esterni che interagiscono con il sistema per svolgere qualche compito: essi potrebbero essere altri attori.

8.5.3 Identifica i Casi d'Uso

Per ogni attore:
1. Identifica compiti e funzioni
   - identifica i compiti o funzioni di più basso livello che l'attore dev'essere in grado di eseguire attraerso il sistema.
   - identifica i compiti che il sistema richiede che l'attore esegua.
2. Raggruppa compiti e funzioni in casi d'uso
   - i casi d'uso devono corrispondere ad un obiettivo specifico per l'attore o per il sistema.
   - raggruppa funzioni che sono eseguite in sequenza o che sono innescate dallo stesso evento.
   - il caso d'uso dev'essere nè troppo grande nè troppo piccolo.
3. Dai un nome al caso d'uso sintetizzando la funzionalità svolta

8.5.4 Definisci il diagramma dei casi d'uso

Il diagramma contiene le relazioni tra attori e casi d'uso, ogni attore deve partecipare ad almeno un caso d'uso, ogni caso d'uso deve avere almeno un attore con cui comunica. Se due attori partecipano agli stessi casi d'uso considerare la possibilità di unirli in un unico attore.

8.5.5 Descrivi i casi d'uso

Considerare sia lo scenario principale che scenari altrernativi ed eccezionali
Per ogni scenario specificare:
- quale evento scatena il caso d'uso (trigger).
- chi inizia il caso d'uso.
- quali precondizioni sono da ritenersi verificate nel momento in cui il caso d'uso inizia.
- quali sono le interazioni tra il/gli attore/i e il sistema e quali dati/comandi vengono scambiati.
- come e quando c'è bisogno di memorizzare i dati.
- come e quando il caso d'uso termina.
- quali post-condizioni sono da ritenersi verificate nel momento in cui il caso d'uso termina.

Se due casi d'uso hanno comportamenti leggermentie diversi e gli stessi attori, considerare la possibilità di avere un unico caso d'uso con scenari alternativi.

8.5.6 Struttura i casi d'uso

Identifica le relazioni di estensione:
- specializza i casi d'uso che hanno molti scenari alternativi.
- collega i nuovi casi d'uso a quelli di partenza mediante relazione «extend».

Identificare le relazioni di inclusione:

- estrarre parti comuni in casi d'uso differenti.
- collegare i casi d'uso che condividono una parte comune al nuovo caso d'uso rappresentante il comportamento condiviso mediante l'associazione «include».

9 Diagramma delle classi

9.1 Definizione

9.1.1 Diagramma UML delle classi

Diagramma strutturale, mostra una vista statica del modello, l'organizzazione del progetto del sistema ma non mostra informazioni temporali. I suoi elementi corrispondono a concetti fondamentali dell'OO.
Rappresenta:

- tipi di oggetti (classi) del sistema, corrispondono alle entità esistenti nel sistema.
- relazioni statiche tra classi ed i vincoli che si applicano a tali relazioni.
- caratteristiche delle classi: proprietà (attributi e responsabilità dei metodi) e responsabilità.

9.1.2 Utilizzo del diagramma

- Fase di definizione dei requisiti: vede le **entità** del dominio e cattura i suoi concetti principali, fissando anche i confini del sistema. Modello architetturale rivolto alla comprensione di cosa fa il sistema, non di come lo fa.
- Fase di progettazione: identifica le **identità** del sistema e le relazioni tra esse, prescinde dall'implementazione.
- Fase di implementazione: identifica **classi** del software e le relazioni tra esse, corrisponde alla struttura reale del software poiché ne modella l'implementazione OO.

9.1.3 Diagramma delle classi nel progetto di software

Si concentra sulla struttura e trascura i comportamenti del sistema, affianca i diagrammi delle classi con diagrammi comportamentali per rappresentare come il sistema si comporta in scenari reali. Un buon approccio prevede un costante passaggio tra diagrammi di struttura e di comportamento, migliorando la comprensione globale del sistema.

9.2 Sintassi

9.2.1 Classe

Descrive un insieme di oggetti con le stesse proprietà (attributi), comportamento (operazioni) e relazioni con altri oggetti.
Ogni oggetto è istanza di una sola classe.
In un dato istante un oggetto ha uno specifico stato.
In base al proprio stato, due oggetti rispondono diversamente alla stessa operazione.
Una classe, con nome singolare e tendenzialmente scritto in UpperCamelCase, è rappresentata da un rettangolo con tre sottosezioni: nome, attributi, operazioni.

9.2.2 Attributi

```
visibilità nome molteplicità:  tipo = valoreDefault {proprietà}
```

Tre livelli di **visibilità** degli attributi:

- + **pubblico**: accesso esteso a tutte le classi.
- # **protetto**: accesso consentito solo alle classi che derivano dalla classe originale.
- **- privato**: solo la classe originale può accedere a tali attributi.

Il **nome** dell'attributo è l'unico parametro obbligatorio ed il **tipo** può essere sia un tipo primitivo che una classe definita nello stesso diagramma.
La **molteplicità** indica il quantitativo degli attributi di un certo tipo: permette di indicare attributi come array o matrici, il valore default è 1. Il numero minimo e massimo possono essere racchiusi tra parentesi quadre quando il tipo è semplice.
Gli elementi di una molteplicità a più valori sono considerati come un insieme.
Se sono dotati di ordine è usata la notazione {ordered}
Se sono possibili valori duplicati è usata la notazione {nonunique}
valoreDefault è il valore assegnato all'attributo di default, se nessun valore è specificato durante la creazione.
{proprietà} indica caratteristiche aggiuntive dell'attributo (es: readOnly).

9.2.3 Operazioni

Rappresentano i metodi di una classe: operazioni invocabili sugli oggetti istanza della classe. Sono le azioni che possono essere svolte da una classe di oggetti, tendenzialmente non sono riportate le operazioni che si occupano solo di modificare gli attributi, perché facilmente deducibili.

```
visibilità nome (listaParametri) :  tipoRestituito {proprietà}
```

Visibilità e **nome** sono analoghi agli attributi, mentre, **tipoRestituito** è il tipo di valore di ritorno.
listaParametri contiene nome e tipo dei parametri della formula.

```
direzione nome parametro :  tipo = valoreDefault
```

direzione: input (in, rappresenta anche il valore default), output (out) o entrambi (inout).
nome, **tipo** e **valoreDefault** sono analoghi agli attributi.

9.2.4 Responsabilità

All'interno dei diagrammi concettuali che descrivono il dominio non si inseriscono metodi perché rappresenterebbero un elemento del dominio della soluzione, è possibile usare le **responsabilità**: insiemi di funzioni principali che la classe dovrebbe garantire, utili per controllare la completezza del modello di dominio.
Sono riportate come stringe di commento (–) nel riquadro delle operazioni.

9.2.5 Note e testo descrittivo

Commenti aggiuntivi in linguaggio naturale, simili ai commenti in un linguaggio di programmazione, possono essere collegati ll'elemento di riferimento tramite una linea tratteggiata, oppure, fluttuare senza collegamenti.

9.2.6 Relazioni tra classi

Generalmente le classi di un diagramma delle classi sono in relazione con altre entità con legami di varia natura, le principai relazioni tra classi sono:

- relazione di **associazione**.
- relazione di **generalizzazione-specializzazione**.
- relazione di **contenimento**.

9.3 Associazioni

L'associazione tra due classi esprime una relazione tra istanze delle classi, è caratterizzata da:

- **nome**: esprime il legame semantico tra le classi associate.
- eventuale **ruolo** giocato da ciascuna delle parti associate.
- **molteplicità** dell'associazione.
- **verso di navigazione** dell'associazione.

9.3.1 Nome

Il **nome** è un'etichetta dell'associazione, solitamente un verbo, permette di formare frasi di senso compiu to e per evitare ambiguità è possibile specificare la direzione dell'associazione (con simboli $<$ o $>$).

9.3.2 Ruolo

Possibilità di specificare il **ruolo** che gli oggetti di una classe rivestono quando collegati da istanze dell'associazione, il ruolo è specificato sull'estremità dell'associazione in prossimità della classe e spesso è usato in alternativa al nome.

9.3.3 Ruoli e Nomi

Non tutte le associazioni necessitano un nome: assegnarlo se migliora la chiarezza e comprensione del modello, ed evitare nomi generici che non aggiungono info utili.
Quando due classi sono coinvolte in associazioni differenti tra le stesse classi, per distinguerle è opportuno riportare un nome dell'associazione o il ruolo delle classi nelle diverse associazioni.

9.3.4 Molteplicità

La molteplicità vincola il numero di oggetti di una classe che possono partecipare ad un'associazione in ogni istante, sono riportate sull'estremità dell'associazione prossima alla classe nella forma: **minimo...massimo**

9.3.5 Verso di navigazione

Da un punto di vista concettuale, un'associazione non ha un verso di percorrenza (se A è legato a B, B è legato ad A).
Se specificata con una **freccia** la direzionalità attribuisce alla classe origine del verso di percorrenza la responsabilità di tenere traccia dell'associazione. Indica in quale direzione è possibile reperire le informazioni.

9.3.6 Associazioni VS Attributi

Le caratteristiche strutturali di una classe possono essere rappresentate sia come associazioni che come attributi di una classe.
Le associazioni, a differenza degli attributi, possono riportare anche le molteplicità di entrambe le classi ma sono meno compatte, il progettista deve scegliere le entità che hanno maggiore importanza e rappresentarle come classi per dar loro più enfasi.
Meglio non appesantire il diagramma con troppe associazioni 1 a 1, possibile usare **attributi** per concetti secondari e **associazioni** per classi significative, la scelta dipende da cosa si vuole enfatizzare nel programma.

9.4 Associazioni Riflessive e Classi Associative

9.4.1 Associazioni Riflessive

Associazione che collega una classe con sé stessa: oggetti di una classe possono essere associati ad oggetti appartenenti alla classe stessa.

9.4.2 Classi Associative

In certi casi, un attributo caratterizza un'associazione ed è difficile stabilire a quale delle classi coinvolte appartiene, può essere opportuno definire una classe associativa che può avere attributi, metodi e altre associazioni, usate in genere per associazioni molti a molti.
La classe associativa connette due classi e definisce un insieme di caratteristiche proprie dell'associazione stessa, ogni istanza di classe associativa, rappresenta un'associazione tra due istanze delle classi e contiene specifici valori degli attributi. Dato che caratterizza un'associazione, prima di creare un'istanza della classe associativa, devono esistere le istanze delle classi collegate. Ci può essere solo un'istanza di una specifica classe associativa tra ogni coppia di oggetti associati (istanze delle classi).
Per rendere reale una classe associativa sarò opportuno trasformarla in classe.

9.5 Generalizzazione

9.5.1 Generalizzazione-Specializzazione (Gen-Spec)

I concetti di generalizzazione e specializzazione in UML sono analoghi a quelli OO, indica legami del tipo is-a ("è un") tra le sottoclassi e una superclasse (concetto generale, sottoclassi la loro specializzazione).
Tipo di associazione rappresentato da una freccia che va dalla classe specializzata alla classe generale (info comuni mantenute in un posto solo, facilitano le modifiche del progetto).
Le sottoclassi hanno tutti gli attributi e operazioni delle classi generali ed aggiungono operazioni e attributi più specifici.

A livello **concettuale** (analisi dei requisiti), gen-spec esprime una relazione is-a tra un concetto generale e le sue specializzazioni.
A livello di **dettaglio** (progetto e implementazione), gen-spec può essere interpretato:

- come relazione di ereditarietà tra due classi concrete: le classi derivate ereditano attributi e metodi public e protected della classe base.
- come relazione di **realizzazione** tra una classe astratta e una classe concreta: la classe base ha soltanto prototipi di metodi, la classe specializzata implementa i metodi sfruttando l'overriding.

Quando Gen-Spec descrive una gerarchia di **ereditarietà**:

- la superclasse raccoglie caratteristiche e comportamenti comuni agli elementi delle sottoclassi.
- la classe derivata (sottoclasse) eredita metodi e attributi della classe base (superclasse) e può:
  1. aggiungere altri attributi e operazioni.
  2. ridefinire i metodi della classe base (overriding).

9.6 Contenimento

La relazione di contenimento rappresenta il legame tra un insieme e le sue parti (tra il contenitore e il suo contenuto).

9.6.1 Aggregazione e Composizione

Il contenimento può essere:

- debole (lasco), detto anche **aggregazione**: la parte mantiene la sua identità quando entra a far parte del tutto, è modellata da un rombo vuoto.
- stretto, detto anche **composizione**: la parte perde la sua identità quando entra a far parte del tutto, è modellata da un rombo pieno.

9.6.2 Aggregazione

Il lato del "tutto" (rombo) è chiamato **aggregato**, le parti possono esistere indipendentemente da quell'aggregato ed è possibile che più aggregati condividano la stessa parte. La molteplicità del lato dell'aggregato, è sottintesa, vale 0..1.
Il ciclo di vita dell'oggetto contenuto è indipendente da quello dell'oggetto contenitore e può esistere indipendentemente dal contenitore.
L'oggetto contenitore non è necessariamente responsabile della costruzione e distribuzione dell'oggetto contenuto.

9.6.3 Composizione

Forma forte di contenimento, in un problema, potrebbe non aver senso parlare di stanze se non sono legate alla casa in cui si trovano.
Le parti non esistono senza il tutto, se il composito viene distrutto anche le sue parti saranno distrutte (o responsabilità ceduta a un altro oggetto), il composito è responsabile della costruzione e distribuzione degli oggetti contenuti, ogni parte può appartenere ad un solo composito per volta.

Principio di non condivisione una classe può essere componente di più classi, un'istanza di classe componente può essere componente di un solo oggetto composito, anche se la classe componente può essere componente di molteplici classi composito diverse.
La molteplicità del lato del "tutto" (composito), quando sottintesa vale 1 nel caso in cui la classe componente ha solo un'altra classe composito. L'altro caso è quando la classe componente può essere contenuta in due o più classi composito diverse, in questo caso la molteplicità è 0..1.