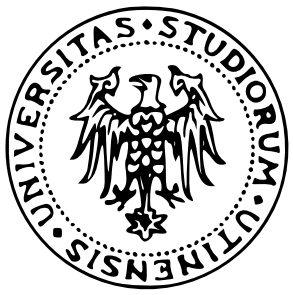
Università degli Studi di Udine

Dipartimento di Scienze Matematiche,
Informatiche e Fisiche

IBML - Internet of Things, Big Data, Machine Learning



Ingegneria del Software

Indice

1 Concetti fondamentali

1.1 Il software

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).

Definizione SW IEEE
IEEE (Institute of Electrical and Electronic Engineers) definisce il software come:

> Insieme di programmi, procedure, regole, e ogni altra documentazione relativa al funzionamento di un sistema di elaborazione dati

1.1.1 Tipologie di SW professionale

Software generico: Software prodotto autonomamente da un'organizzazione, per incontrare necessità di vari clienti. Il produttore ha controllo sulle specifiche del software.
Software su richiesta: Software sviluppato da un'organizzazione su commissione di uno specifico cliente, il produttore deve attenersi alle specifiche indicate dal cliente.

1.1.2 Complessità e costi del software

Il software può diventare complesso, difficile da capire e costoso da modificare.
Le nuove tecniche e tecnologie permettono lo sviluppo di software sempre più grandi e complessi e lo sviluppo richiede meno tempo.
Il mercato in continua evoluzione richiede rilasci rapidi dei sistemi.
Il successo o fallimento di un software è legata o meno all'applicazione dell'ingegneria del software.

1.1.3 Sviluppo professionale di Software di qualità

Sviluppo personale: Software difficilmente riusato in futuro da altri utenti, non necessario scrivere guida o documenti di progetto, non necessario raggiungere alti livelli di qualità.
Sviluppo professionale: Software usato da altre persone diverse dai propri sviluppatori, sviluppo in team, deve avere caratteristiche di qualità.

1.1.4 Qualità del Software

Un Software di qualità deve fornire le funzionalità e le prestazioni richieste, le caratteristiche qualitative di un Software professionale sono:

- **Accettabilità**: il Software dev'essere accettato dai suoi utenti, dev'essere comprensibile, usabile e compatibile con i sistemi utilizzati dagli utenti.
- **Fidatezza e Protezione**: il Software non deve causare danni in caso di malfunzionamento e utenti malintenzionati non devono accedere.
- **Efficienza**: il Software non deve sprecare le risorse di sistema.
- **Mantenibilità**: il Software deve poter evolvere per soddisfare le nuove richieste dei clienti.

L'insieme specifico dei caratteri qualitativi di un sistema Software dipende dalla sua applicazione.

1.2 Ingegneria del Software

Le principali cause del fallimento di un Software sono decretate dalla crescente complessità dei sistemi, dallo sviluppo senza approccio ingegneristico (come testing) e richiesta elevata con tempi di rilascio estremamente brevi.

1.2.1 Nascita

L'Ingegneria del Software nasce negli anni '70 dove i costi del Software iniziano a dominare sui costi dell'hardware.
Nasce dal **problema** di sviluppare Software facilmente manitentibile in maniera economicamente vantaggiosa.
Nel '69 prima Crisi del Software: i sistemi Software erano sempre più complessi, difficili da manutenere, inaffidabili, più costosi del previsto e rilasciati in ritardo.
Soluzione della crisi: adozione di processi ingegneristici alla produzione del Software.

1.2.2 Definizioni

Definizione Ingegneria del Software IEEE
IEEE (Institute of Electrical and Electronic Engineers) definisce l'ingegneria del Software come:

> L'applicazione di un approccio sistematico, disciplinato e quantificabile allo sviluppo, all'operatività e alla manutenzione del software.

Definizione Ingegneria del Software

> L'Ingegneria del Software è una disciplina ingegneristica che riguarda tutti gli aspetti della produzione del software"

1.3 Processi Software

Il **processo Software** è l'insieme di attività che porta alla creazione (o evoluzione) di un prodotto software.

- **Acquisizione, analisi e specifica dei requisiti**: clienti e ingegneri definiscono le funzionalità e i vincoli operativi del Software da produrre.
- **Progettazione e Sviluppo**: progettazione e programmazione.
- **Verifica e Validazione**: verifica che il Software sia ciò che il cliente richiede.
- **Evoluzione**: il Software viene modificato per soddisfare cambiamenti dei requisiti del cliente e del mercato.

NO FREE LUNCH: non esiste una soluzione che risolve tutti i problemi

1.3.1 Metodi e Strumenti

- **Metodi**: approcci strutturati per sviluppare Software di qualità, con costi contenuti ed entro i tempi di consegna
- **Strumenti**: Software utilizzati per aiutare le attività dei processi Software (come ChatGPT).

1.3.2 Sfide

- **Diversità**: definire i metodi per produrre Software eseguito su dispositivi eterogenei
- **Consegna**: consentire la consegna del Software in tempi rapidi, rispondendo ai cambiamenti circostanti
- **Fiducia**: sviluppare tecniche che dimostrano all'utente che può fidarsi del Software, garantendo la protezione delle informazioni
- **Scala**: il Software dev'essere distribuito su molti sistemi diversi

1.3.3 Principi Fondamentali dell'IS

Metodi specifici, tecniche e strumenti utilizzati dipendenti dall'organizzazione che sviluppa il Software, dal tipo e dalle persone coinvolte nel processo di sviluppo. Non esistono metodi universali applicabili a tutti i sistemi e a tutte le aziende.
I concetti fondamentali dell'ingegneria del Software sono **indipendenti dal linguaggio di programmazione** utilizzato per sviluppare il Software.

1.3.4 Processo chiaro

Il processo specifico da utilizzare dipende dal tipo di Software che dev'essere sviluppato, indipendentemente da questo, devono esserci chiarezza di comunicazione su ciò che viene eseguito e fatto, bisogna definire gli obiettivi per avere le ideee chiare sulle responsabilità.

1.3.5 Fidatezza e prestazioni

Il Software deve comportarsi come vogliono i stakeholder e dev'essere utilizzabile quando richiesto (caso della manutenzione).

1.3.6 Requisiti

Il requisito non è solo quello che fa ma anche i vincoli operativi, durante l'analisi bisogna capire cosa si aspettano i clienti e chi usa il Software.
Fase importante che può determinare il fallimento di un progetto.

1.3.7 Riuso

Riutilizzo del Software esistente già funzionante in modo appropriato per un nuovo sistema Software.

1.3.8 Modello di processo Software

Ogni organizzazione utilizza il proprio processo.
Il **modello di processo Software** è una rappresentazione astratta che descrive l'intero ciclo di vita del Software.

2 Processi Software

2.0.1 Code and Fix

Dal problema scrivo poi il codice (o vibe coding), si crea il problema di aggiustare il codice per gli errori e l'aggiunta di funzionalità.
Risulta un approccio con limitazioni che lo rendono inadeguato per lo sviluppo di Software professionale.

2.0.2 Definizione

Processo Software: insieme strutturato di attività tecniche, collaborative e manageriali che porta alla creazione (o evoluzione) di un prodotto Software.
Un elevata qualità di tale processo consente di migliorare:

- qualità del prodotto finale
- tempi per portare il prodotto sul mercato
- costi affrontati dall'organizzazione

2.1 Attività dei processi Software

2.1.1 Attività fondamentali

Ogni processo è composto da attività fondamentali, condivise da tutti i processi, queste attività possono essere organizzate e realizzate in modi diversi e in processi diversi.
Ci sono 5 (4+1) attività fondamentali:

1. **Acquisizione, analisi e specifica dei requisiti**:
2. **Progettazione e Sviluppo**:
3. **Verifica e Validazione**:
4. **Evoluzione**
5. **Studio di fattibilità**

Stabilisce se lo sviluppo debba essere avviato: se esiste un mercato per il Software e se il Software sia tecnicamente ed economicamente realistico.
Definisce quali sono le alternative possibili e le scelte più ragionevoli, stimando le risorse necessarie per ciascuna alternativa.
Fornisce un report di fattibilità e dovrebbe essere relativamente rapido e poco costoso.

2.2 Acquisizione, analisi e specifica dei requisiti

Stabilisce cosa il Software dovrà fare, non bisogna vincolare progettazione e implementazione, è necessaria l'interazione con il committente. Rappresenta un'attività critica: un errore può costare molto.
Sviluppa metodi per raccogliere, documentare, classificare e analizzare i requisiti:

1. **Deduzione e analisi dei requisiti**: comprensione di come e cosa sono questi requisiti dai stakeholder.
2. **Specifica dei requisiti**: traduzione dettagliata dei requisiti in specifiche.
3. **Convalida dei requisiti**: controllo che i requisiti siano realistici, coerenti e completi. Permette di correggere eventuali errori.

Queste attività sono intrecciate tra loro, come mostrato nel diagramma:

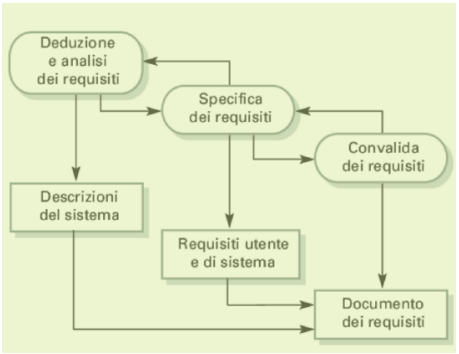


Figura 1: Diagramma "Acquisizione, analisi e specifica dei requisiti.

2.2.1 Deduzione e Analisi

La **deduzione** richiede spirito critico e può coinvolegere: osservazioni di sistemi già esistenti e discussione con possibili utenti.

Durante l'**analisi** può avvenire lo sviluppo di uno o più modelli prototipi per capire meglio cosa si attende l'utente finale.

2.2.2 Specifica

La **specifica** traduce le informazioni dedotte in un insieme di requisiti, possono essere di due tipi:

- **sistema**: descrizione dettagliata delle funzionalità e caratteristiche che devono essere fornite (utile per gli sviluppatori)
- **utente**: proposizioni astratte dei requisiti del sistema per i clienti e gli utenti finali

2.2.3 Convalida

La **convalida** controlla che i requisiti siano realistici, coerenti e completi. Nel mentre possono essere rilevati errori nel documento dei requisiti, documento che dovrà essere modificato in presenza di errori, in modo da correggerli.

2.2.4 Documento dei requisiti

Al termine della convalida si ha un documento che definisce l'insieme dei requisiti, questo documento può essere più dettagliato e formale a seconda del processo. Dev'essere comprensibile, completo, coerente, non ambiguo, modificabile. In questa fase è predisposta una **fase di test** del sistema.

2.3 Progettazione e sviluppo

Ha l'obiettivo di trasformare le specifiche in un sistema eseguibile

- **Progettazione**: progetto la struttura del Software che realizzi le specifiche.
- **Sviluppo**: implementazione dei componenti della fase di progetto.

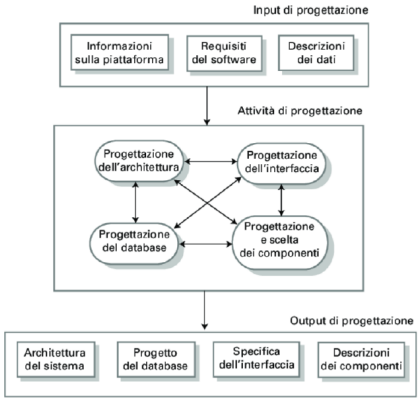


Figura 2: Diagramma "Progettazione e Sviluppo".

2.3.1 Progettazione

Input e Output di progettazione .
Attività della fase di progettazione sono intrecciate e interdipendenti.

2.3.2 Sviluppo

Attività dipendente dagli sviluppatori e dalle loro caratteristiche, l'attività può essere soggetta a standard aziendale.

2.4 Verifica e Validazione

Termini simili (V&V), entrambi devono valutare e misurare il corretto funzionamento del Software.
Verifica: mostra che un sistema è conforme alle sue specifiche, non possiamo essere soddisfatti dopo la fase di verifica perchè non sappiamo effettivamente se al cliente va bene.
Validazione: dimostrazione che un sistema soddisfa le aspettative di un cliente.
L'ispezione e la revisione può essere effetuata in qualsiasi momento della fase del processo Software e la tecnica più utilizzata è quella del **testing**.
Testing: stresso il Software con degli Input per vedere se mi ritorna gli Output attesi, in piccoli progetti la fase di test è eseguita durante lo sviluppo.

Figura 3: Diagramma "Fase di Testing".

2.4.1 Test dei componenti

Test dei singoli componenti del sistema (linee di codice, file, classi, ...), un componente rappresenta l'unità fondamentale del Software.
Il componente è testato singolarmente **in isolamento** simulando il resto del Software che non è momentaneamente disponibile.

2.4.2 Test del sistema

Testing del sistema completo, apre ad una criticità: passaggio dal test delle singole componenti al componente totale.
Fase di **integrazione**: aggiunta delle singole componenti in sottoinsiemi del programma per portare al Software intero.
Maggiore è la complessità finale, più integrazioni verranno eseguite durante il processo Software.

2.4.3 Test del cliente

Test del sistema finale con i dati reali del cliente e test effettuati con il cliente.

2.5 Evoluzione

Il Software continua ad essere di competenza anche una volta uscito, l'adattamente dipende dalle esigenze di mercato, dalla modifica dei vincoli ed esigenza di correzione

errori, viene evoluto errori della versione finale.
Esiste sempre meno Software totalmente nuovo, c'è sempre una base esistente o librerie (o componenti Software riusabili).
La modifica avviene in diverse fasi del processo e può avvenire dopo il rilascio per:

- correzione difetti
- migliorare qualità del Software
- adattare il sistema ai mutamenti dei sistemi operativi

Queste attività fanno evolvere il sistema in nuovi sistemi.

3 Modelli di processi Software

3.1 Definizioni

3.1.1 Definizione W. Scacchi

Definizione **modello del ciclo di vita del software** (CVS) di W. Scacchi
"Un modello del ciclo di vita del software è una caratterizzazione descrittiva o prescrittiva di come un sistema software viene o dovrebbe essere sviluppato".

3.1.2 Definizione IEEE

Definizione standard IEEE
"framework che contiene processi, attività e task che fanno parte dello sviluppo di un prodotto Software che va dalla definizione dei requisiti alla terminazione del suo uso".

3.2 Informazioni processo - modello

3.2.1 Processo VS. Modello

Non esiste un processo universale, spiegazione di un modello ad alto livello e istanziato poi nel concreto in maniera differente.
Una volta conosciuti si possono estendere e adattarli ad un'esigenza reale.

3.2.2 Punti di vista del processo

Ogni processo ha diverse possibili descrizioni:

- **architetturali**: descrizione della sequenza delle attività senza fornire dettagli sulle specifiche attività
- **data-flow**: come avviene la trasformazione dei dati durante il processo
- **role/action**: focalizzazione sulle persone, definisce ruoli e responsabilità delle persone coinvolte nel processo

3.2.3 Descrizione processo Software

La descrizione del processo Software include:

- **Attività** da svolgere e l'ordine
- **Prodotti** risultanti da ciascuna attività
- **Ruoli** persone coinvolte nel processo e le loro responsabilità

- **Pre e Post-condizioni** per iniziare un'attività devo avere delle pre-condizioni e al termine dell'attività voglio che si siano realizzate delle post-condizioni

3.2.4 Scelta del modello

Differenti tipi di modelli adatti a diversi tipi di prodotto, per il corso due tipologie:
- **plan-driven** dove ogni fase è ben definita
- **agili**

3.3 Modello a cascata

Primo modello, per le sue caratteristiche percepito come vetusto, applicato allo sviluppo Software di tutti i giorni.
Processo plan-driven: ciascuna fase segue quella precedente, va tutto deciso in anticipo prima di iniziare lo sviluppo del Software.
Gli Output di una fase sono gli Input della fase successiva.
Le fasi riflettono le attività di sviluppo fondamentali del Software.
È un processo centrato sui documenti: è fondato sulla produzione di documentazione che rende tracciabile l'intero andamento del processo, al termine di ogni fase produco un documento (il dettaglio lo decide il processo), che venga approvato (se non approvato, rifaccio la fase) e la fase successiva è congelata finché non riceve il documento approvato della fase precedente.
È un processo rigido, i prodotti di una fase validata non sono più modificabili, se non attraverso un processo formale e sistematico di modifica. La fine di ogni fase è un punto rilevante del processo (milestone). Definizione di milestone è importante per valutare l'avanzamento del processo.
È un processo monolitico, il cliente interagisce con il Software solo al termine della cascata (scopro solo al termine se il processo ha degli errori).

3.3.1 Vantaggi e Svantaggi

Vantaggi
- fasi ben definite
- Output di ogni fase precisamente individuati

Svantaggi
- Richiede una conoscenza completa, immediata e stabile di tutti i requisiti (nel mondo reale i requisiti possono variare durante lo sviluppo).
- Time to Market: sviluppo di eccessiva documentazione, spesso non richiesta, genera perdita di tempo.
- Poco flessibile: se durante il ciclo di sviluppo del processo bisogna gestire modifiche (problemi o requisiti) bisogna aprire un processo formale di modifica.

Il modello a cascata è adatto a Software che richiedono documentazione, a sistemi integrati dove il Software deve interfacciarsi con sistemi hardware non flessibili.
Non è adatto se i requisiti cambiano rapidamente o se si lavora in piccoli team.

3.3.2 Modello a cascata con retroazione

In ciascuna fase viene introdotto il **Feedback** (retroazione), così facendo, se nella fase successiva si nota un problema, si può tornare alla fase precedente.

La retroazione mitiga la monoliticità del processo a cascata: non bisogna più attendere il termine del processo per modificare il prodotto, non lo rende comunque completamente flessibile.
Rappresenta un modello utile quando il processo avrà pochi cambiamenti nel percorso.

3.3.3 Modello a V

Estensione del modello a cascata. Team di progettazione e di testing possono lavorare in parallelo, nel ramo superiore è presente la fase di progettazione e nel ramo inferiore la fase di verifica e validazione.
Nella specifica dei requisiti e di sistema siamo già in grado di sviluppare un grado di test per il cliente, che può anticipare la validazione.

3.4 Modelli Evolutivi

Modelli chiari a contesti dove i requisiti non sono congelabili all'inizio del processo, due macromodelli:

- **Sviluppo/Consegna incrementale**: lavoro con il cliente per sviluppare i requisiti iniziali, evolvere con iterazioni durante il processo, raffinamenti successivi con il cliente per arrivare al prodotto finale
- **modello prototipale**: capire quali sono i requisiti del sistema, sviluppando il prototipo per capire quali sono i requisiti del sistema

3.4.1 Modello a Sviluppo Incrementale

Sviluppata un'implementazione iniziale, sviluppo i requisiti, avviene in contemporanea la fase di sviluppo e testing, dove rilascio degli incrementi (versioni successive) fino ad arrivare alla versione finale. Parte da pochi requisiti ben compresi o da una descrizione sommaria degli stessi. Attività di specifica, sviluppo e convalida sono intrecciate, con feedback veloci tra le attività. La versione iniziale implementa i requisiti della descrizione sommaria e viene esposta agli utenti (clienti o proxy di clienti), più semplice dare feedback su una versione iniziale (come demo). L'utente suggerisce ulteriori funzionalità e requisiti che sono implementati in versioni intermedie successive, solitamente non mostrate al cliente. La versione finale rappresenta l'ultimo incremento del sistema rilasciata al cliente e corrisponde ad un'evoluzione della versione iniziale.

3.4.2 Modello a Consegna Incrementale

Il sistema viene consegnato a incrementi, non vengono attese versioni finali da rilasciare al cliente (non per forza tutti), vengono consegnati al cliente e installati nel loro ambiente operativo gli incrementi.
Vantaggio: feedback più realistico, il cliente testa il sistema effettivamente.
Limitazione: il cliente non ha competenze-tempo per dare dei feedback corretti all'incremento.
Ogni incremento rilascia una parte definita delle funzionalità richieste, in aggiunta alle precedenti. Con il cliente vengono definiti i requisiti da rilasciare prima, asse-

gnando delle priorità, nel mentre gli altri requisiti possono evolvere.

Parto da una descrizione sommaria dei requisiti, andando a fare un planning: assegnare ciascun requisito a uno o più incrementi, pianificando le versioni da sviluppare.
La convalida viene fatta a ciascun incremento.
Svolgo **integration test** e **test di sistema complessivo**, ottengo una versione successiva.

3.4.3 Consegna VS Sviluppo Incrementale

Nello sviluppo incrementale la valutazione della prima versione è effettuata da un proxy del cliente, le versioni intermedie non sono tendenzialmente rilasciate al cliente.
La consegna incrementale permette una valutazione più realistica.

Vantaggi (entrambi):

-
-
-

Problemi

-
-
-

Applicabilità

-
-
-

3.4.4 Modello Prototipale

Prototipo: versione iniziale di un intero sistema o di parte di esso.
Viene sviluppato rapidamente per contenere i costi e poter sperimentare con il cliente prima della consegna, nelle fasi iniziali del processo.
Il prototipo è usa e getta, dev'essere scartato dopo la validazione perchè non rappresenta una buona base per sviluppare il sistema finale.
Anche se realizza le funzionalità richieste potrebbe non rispettare vincoli fondamentali e potrebbe non avere documentazione.
La rapidità di sviluppo e frequenti modifiche ne deteriora la qualità.

Bisogna stabilire gli obiettivi del prototipo all'inizio del processo, se gli obiettivi non sono chiari gli utenti finali possono fraintendere la funzione del prototipo e la prototipazione risulta inefficace.
Devo circoscrivere le funzionalità del prototipo, definisce lo schema architetturale del prototipo. Ci permette di avere un focus sulle attività critiche per svilupparlo nel minor tempo possibile.
Il valutatore dev'essere formato sull'utilizzo di ogni prototipo prima della valutazio-

ne.
Problemi della rappresentatività:

- prototipo non è parte del sistema reale
- i valutatori potrebbero essere non rappresentativi degli utenti finali
- il modo di usare il prototipo può differire dall'utilizzo del sistema reale

Lo sviluppo prototipale può essere integrato all'interno di altri cicli di vita classici (può essere usato per identificare, validare e raffinare i requisiti).

Uso i requisiti estratti dalla fase di prototipazione con più modelli di progettazione.

3.4.5 Back-To-Back Testing

Il prototipo può essere utilizzato nella fase di validazione per controllare che il sistema sviluppato si comporti come modellato ne prototipo nelle fasi iniziali del progetto. Comparo i risultati del prototipo del sistema e del sistema applicativo, per capire dov'è il problema nel codice.

3.5 Modelli Orientati al Riuso

Il riuso non avviene solo in maniera informale, può anche essere un approccio sistematico.
Approccio orientato al riuso:

- componenti software riutilizzabili
- interi sistemi

Sfrutta framework di integrazione, software che consentono di comporre i componenti per creare il sistema finale che si adattano ai requisiti del cliente.
Approccio diffuso grazie a standard per la specifica dei componenti.

Requisiti essenziali specificati in maniera non eccessivamente dettagliata.
Vengono ricercati componenti e sistemi che possono fornire le funzionalità dei requisiti, i candidati vengono valutati per vedere se soddisfano i requisiti essenziali e se sono disponibili per essere utilizzati.
I requisiti vengono perfezionati usando le informazioni sulle applicazioni e componenti riutilizzabili che sono stati trovati, la specifica viene aggiornata con i requisiti perfezionati.
Verifico se ci sono sistemi interamente disponibili o singole componenti, nel caso dove ho un sistema configurabile che mi realizza le funzionalità lo adatto, nel caso ci siano delle componenti le posso adattare e svilupparle da me, integrandole nel sistema finale.

Vantaggi

- riduce la quantità di codice da sviluppare da capo
- riduce costi e rischi
- maggiore velocità di consegna

Svantaggi

- compromessi nei requisiti –> sistema potrebbe non soddisfare tutte le reali necessità degli utenti
- evoluzione dei componenti non controllata direttamente (se la libreria cambia le interfacce potrei esser fregato)

3.6 Modelli Trasformazionali

Sono dei metodi formali, definiscono con i linguaggi formali:
- specifiche algebriche
- modelli di stato

Usa tecniche di model checking per provare la correttezza. Le specifiche formali sono trasformate in codice:
- la correttezza è preservata
- la verifica è ottenuta implicitamente

Un documento in linguaggio formale dei requisiti, presenta una comprensione chiara e non ambigua, le specifiche sono verificate automaticamente prima di essere trasformate da opportuni strumenti.

La descrizione formale viene astratta e dettagliata, fino a ottenere specifiche di basso livello eseguibili. Le trasformazioni, questo processo può trarre vantaggi da componenti riusabili, possono essere eseguite manualmente o supportate da appositi strumenti.

Problemi
- necessità di competenze specifiche in linguaggi formali
- difficile specificare formalmente alcune parti del sistema
- difficile per il cliente convalidare i requisiti

Applicabilità
- non adatti per sistemi di grandi dimensioni
- usati per parti critiche, dove la validità va dimostrata by construction

4 Modelli Agili

4.0.1 Sviluppo agile del Software

La necessità di essere agili nello sviluppo nasce da 3 segnali di un report del 2000:
- meno del 30% dei progetti software aveva successo
- progetti grandi fallivano più spesso
- solo metà delle feature richieste era effettivamente rilasciata

E dal problema di instabilità dei requisiti del software: molti progetti presentavano requisiti non chiari, instabili e variabili, il cliente si accorgeva di funzionalità da richiedere durante la fase di sviluppo.

4.0.2 Sviluppo rapido nel Software

Contesto ricco dove ognuno può diventare sviluppatore, dove le opportunità hanno rapido cambiamento e presentano prodotti concorrenti, spesso è importante prevedere la prossima funzionalità richiesta dal mercato.
La **rapidità di sviluppo e consegna** è spesso il requisito più critico per i sistemi software.
Al termine degli anni 90 sono emersi i primi metodi agili per lo sviluppo software che erano mirati alla produzione software in tempo ridotto.

4.1 Agilità

Significato di **agilità**:

- Efficace risposta ai cambiamenti
- Efficace comunicazione fra tutti gli stakeholder
- Portare il cliente nel team di lavoro per avere feedback rapido

L'agilità consente di avere una rapida ed incrementale consegna del Software.

4.2 Caratteristiche dei metodi agili

Documentazione minima

- focus sul codice, invece che sulla progettazione
- niente specifiche dettagliate, avremo dei requisiti che evolvono con la comunicazione con l'utente e l'osservazione del mercato
- overhead di documentazione limitati (non si cancella ma striminzita)

Consegna rapida ed incrementale

- sistema sviluppato in incrementi rilasciati frequentemente, ogni due-tre settimane
- stakeholder coinvolti nella specifica e nella valutazione di ogni incremento, definendo insieme gli incrementi successivi

Strumenti di supporto

- utilizzo di strumenti al supporto del processo di sviluppo (come automatizzazione dei test)

4.2.1 Processi Plan-Driven VS Agili

- **Plan-Driven** processi dove le attività sono pianificate in anticipo e il loro avanzamento è misurato rispetto a quanto previsto dal piano. Fasi del processo Software sono distinte tra loro e gli output di ciascuna fase sono necessari per la fase successiva.
- **Agili** pianificazione incrementale e continua durante lo sviluppo. Più facile modificare il processo per adeguarsi alle modifiche dei requisiti del cliente o del prodotto. Requisiti, progettazione e implementazione avvengono assieme.

4.2.2 Processi Plan-Driven e Agili

Pratiche plan-driven e agili possono coesistere nello stesso processo, ma in momenti diversi (o nello stesso momento):

- processi agili possono produrre documentazione di progettazione quando ritenuto necessario e possono includere attività pianificate. Lo scopo è di supporto per comunicazione e comprensione, sapendo che possono essere incompleti o imprecisi.
- processi plan-driven possono essere incrementali

Per grandi sistemi serve trovare un compromesso tra processi pianificati e processi agili.

DISCLAIMER: Gli incrementi esistono anche nei plan-driven.

4.2.3 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)

4.2.4 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.

Vantaggi

- consegne predicibili: impegno periodico
- rapida risposta ai cambiamenti dei requisiti
- rischi attenutati grazie ai cicli di consegna brevi
- sensazione di alta produttività
- clienti soddisfatti perché si sentono ascoltati e responsabilizzati
- il successo dei progetti evolve attraverso multiple brevi iterazioni

4.3 Tecniche Agili

4.3.1 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.

4.3.2 Extreme Programming

Metodo agile più conosciuto, spinge le pratiche di sviluppo a un livello estremo. Ha un approccio iterativo estremo ed ha piccoli e frequenti incrementi rilasciati al cliente.

Requisiti espressi come storie utente (scenario di utilizzo base del software per ottenere un risultato), per decidere quali funzionalità includere come incremento di sistema.
Scenari divisi in task (costituiscono le unità principali dell'implementazione) di sviluppo più semplici che vengono implementate direttamente.
La pianificazione è presente però è flessibile e non appesantita da documentazione eccessiva.
I programmatori lavorano a coppie (pair programming) e sviluppano test per ogni task prima di scrivere codice. Tutti i test devono essere eseguiti con successo quando il nuovo codice viene integrato nel sistema. Il cliente è coinvolto nello sviluppo, valida la release corrente, fornisce requisiti nuovi o modificati, partecipa alla selezione delle storie utente, definisce i test di accettabilità.

Coinvolgimento del cliente: rappresentante del cliente on-site, parte del team di sviluppo.
Accettare i cambiamenti: quando un task viene concluso, viene integrato nel sistema.
Sviluppo incrementale: piccoli e frequenti rilasci che aggiungono in modo incrementale nuove funzionalità.
Mantenere la semplcità: progetto più semplice possibile che soddisfi i requisiti correnti e costante attività di miglioramento del codice (refactoring).
Persone, non processi: programmazione in coppia, proprietà collettiva del codice, sviluppo non richiede orari di lavoro eccessivamente lunghi e non sono accettabili troppi straordinari.

4.3.3 Influenza dell'Extreme Programming

XP come proposta originaria non è quasi mai adottato.
Non è pratico perché richiede un cambio radicale nel modo di lavorare di un'organizzazione.
Le pratiche chiave dell'XP hanno ispirato tutti gli approcci agili e spesso incorporate nei processi di sviluppo:

- storie utente
- refactoring
- sviluppo preceduto dai test
- pair programming

4.4 Storie Utente

Chiamate anche scenari, descrivono i requisiti degli utenti come scenari d'uso dove l'utente dovrebbe trovarsi.

Descritte da cliente e team su schede (**story cards**) che il team di sviluppo suddivide in **task** da implementare.
I task rilevati sono la base per definire la pianificazione delle iterazioni e le stime dei costi.

In XP il cliente è parte del team ed il suo compito è prendere decisioni sui requisiti con il team.
Cliente e team definiscono una priorità e una stima dei costi per ciascuna storia, oltre ai criteri di accettazione.
Sulla base di queste informazioni, cliente e team scelgono le storie che saranno incluse nella prossima versione del prodotto.

PRO

- integrano la deduzione dei requisiti con lo sviluppo, invece di avere apposite attività di ingegneria dei requisiti, in modo da gestire i cambiamenti nei requisiti
- più semplice relazionarsi con storie utente, coinvolgendo così maggiormente l'utente
- ordinate in base a quelle che possono fornire un supporto utile all'azienda
- se i requisiti cambiano, vengono aggiunte story cards e le storie non ancora realizzate possono essere modificate o scartate

CONTRO

- stabilire se le storie utente coprono completamente i requisiti di sistema non è semplice
- descrizione incompleta del requisito: i clienti esperti potrebbero omettere scenari o task considerati ovvi

4.5 Refactoring

Progettare pensando al cambiamento, riduce i costi di manutenzione futura. XP rinuncia a gestire in anticipo i cambiamenti perché ritiene che le modifiche non si possano prevedere con certezza. Fowler propone il termine di **refactoring** per rendere più semplice l'implementazione delle eventuali modifiche future, rappresenta un processo di miglioramento del codice che viene riorganizzato e riscritto per renderlo più efficiente e comprensibile, senza cambiarne le funzionalità.

- il team di sviluppo cerca proattivamente aspetti del software da migliorare e implementa i miglioramenti immediatamente
- il miglioramento può riguardare anche situazioni dove non c'è necessità immediata
- un codice di più alta qualità riduce la necessità di documentazione e facilita le modifiche future

PRO

- sviluppo incrementale tende a portare al deterioramento del codice, il refactoring continuo mitiga il deterioramento, migliorando la struttura e la leggibilità del codice
- esistenza di tool per automatizzare (o proporre suggerimenti su) alcune operazioni di refactoring

CONTRO

- a volte il refactoring a livello di codice non basta per supportare un cambiamento, però è necessaria una modifica dell'intera architettura che è più costosa
- bisogna trovare un compromesso tra tempo dedicato allo sviluppo di nuove funzionalità e refactoring

4.6 Sviluppo Preceduto dai Test

In XP il testing è fondamentale: il Software viene testato dopo ogni cambiamento.
Caratteristiche fondamentali:

- **Sviluppo test-driven** - casi di test da soddisfare sono scritti prima del codice e guidano lo sviluppo
- **Automatizzazione dei test** - strumenti eseguono automaticamente i test ogni volta che viene rilasciata una nuova versione
- **Coinvolgimento del cliente** - il cliente è coinvolto nello sviluppo dei test di accettazione

4.6.1 Sviluppo Test-Driven (TDD)

Test scritti precedentemente al codice chiarisce i requisiti da implementare, aspetto cruciale in assenza di specifiche documentate accuratamente che guidino i test di sistema.
Test scritti come programmi affinchè vengano eseguiti automaticamente, ogni test simula invio di input e controlla l'output, è possibile eseguire i test durante lo sviluppo in modo da scoprire subito eventuali problemi.
Sia i test pre-esistent, che i nuovi test, vengono eseguiti ogni qualvolta una nuova funzionalità viene aggiunta al sistema, questo permette di verificare le nuove funzionalità per vedere che non abbiano errori, lo sviluppo non può procedere finchè tutti i test non vengono superati.

4.6.2 Automazione dei test

I test sono scritti come programmi eseguibili, il programma simula l'immissione di input e valuta che l'output sia quello atteso. L'automatizzazione rende più facile e rapida la fase di verifica e validazione.

4.6.3 Pro e Contro

PRO

- La scrittura del test implica la definizione di un'interfaccia e di una specifica comportamentale delle funzionalità da sviluppare, riducendo incomprensioni, ambiguità e omissioni.

CONTRO

- Pratica onerosa per il cliente
- Sforzo per tenere aggiornati i test interessati dalle modifiche del codice
- I test potrebbero essere incompleti, per varie ragioni non verificano tutti i possibili scenari che potrebbero verificarsi

4.7 Pair Programming

Programmatori lavorano in coppia sulla stessa postazione, viene incoraggiato il refactoring poichè è più facile che l'intero team ne benefici.
Egoless programming: tutti quelli che ci lavorano sono proprietari del codice, quindi anche responsabili dei problemi.

4.7.1 Pro e Contro

PRO

- Aiuta a sviluppare il senso di proprietà del codice nel team e a diffondere la conoscenza nel team
- Permette un processo di revisione informale: gli sviluppatori verificano reciprocamente il proprio lavoro e ogni linea di codice è controllata da più di una persona
- Riduce i rischi di fallimento dovuti a turn-over

CONTRO

- La programmazione in coppie può essere meno efficiente della programmazione individuale, soprattutto quando riguarda programmatori esperti

4.8 SCRUM - Gestione Agile della Progettazione

Il metodo agile **Scrum** offre un framework per organizzare agilmente progetti e fornire una visibilità esterna su ciò che sta accadendo, in maniera puntuale, all'interno del team di sviluppo del software.

Metodo iterativo, hanno creato una loro identità del framework con la loro terminologia.
Si parte dai requisiti, storie utente o documento dei requisiti, queste necessità di sviluppo sono trasferite nel **Product backlog** (To-Do list per la realizzazione del prodotto).
Sprint rappresenta un ciclo di sviluppo del modello. Il product owner definisce i requisiti del prodotto, stabilisce le priorità e rivede continuamente il product backlog per evitare criticità.
All'inizio di ogni ciclo vengono stabilite le priorità e si stabilisce quali sono gli elementi a priorità più alta e a quali team verranno assegnati. Viene fatta una stima della velocità, sulla velocità di realizzazione delle passate iterazioni, per capire come si lavora nel team e per perfezionare come vengono implementati i diversi sprint. Viene creato lo **sprint backlog**, il lavoro da svolgere nello sprint, il team decide dchi dovrà lavorare su determinati elementi e avvia lo sprint.
Giornalmente viene fatta una riunione per esaminare l'avanzamento del lavoro e stabilire le priorità del lavoro da svolgere in quel giorno, dovrebbe essere un breve incontro di tutti i membri del team. Lo ScrumMaster ha la responsabilità di garantire che il processo Scrum venga eseguito con efficienza, è un moderatore.
Al termine di ogni sprint ci dev'essere un software potenziamente rilasciabile presso il cliente, deve trovarsi in uno stato finito però nella pratica non è sempre realizzabile.
Al termine dello sprint viene eseguita una riunione di verifica per rivedere il processo e valutare il prodotto.

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

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.

Figura 4: Diagramma "Deduzione e Analisi dei Requisiti: Processo"

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».