Introduzione:

Per **Ingegneria del Software** si intende l'applicazione di un metodo di sviluppo sistematico, rigoroso e ripetibile allo sviluppo e al mantenimento del software.
E' una disciplina che si occupa di tutti gli aspetti della produzione software, dalle prime fasi di specifica alla manutenzione, usando teorie e metodi tipici dell'approccio ingegneristico applicate allo sviluppo software

Origini:

Le origini della disciplina risalgono alla fine degli anni '60 (ufficialmente 1968) quando si è resa necessaria una risposta alla cosiddetta *crisi del software*; si assisteva in quegli anni, infatti, a progetti informatici che spesso superavano il budget prefissato, il tempo limite imposto, non rispettavano i prerequisiti e soprattutto erano difficilmente mantenibili.
Per questo motivo si diffonde l'idea di approcciare il software **non** come arte, ma come disciplina ingegneristica al pari della costruzione di ponti o di edifici

Perchè è necessario?

La necessità di un approccio ingegneristico alla progettazione software e allo sviluppo di progetti da esso derivante si nota analizzando casi in cui un errore più o meno banale ha portato a disastri (economici e non solo)

- *Ariane 5* → Razzo dell'Agenzia spaziale Europea, lanciato nel 1996 ed esploso durante il volo inaugurale a causa di un overflow di un modulo che ne gestiva l'inclinazione
- *Therac25* → Macchinario per la radioterapia che a causa di una rece condition in particolari condizioni irraggiava i pazienti con una quantità di radiazioni x100, causando così tra l'85 e l'87 6 feriti di cui 3 morti a causa dell'eccesso di onde.

Il problema principale è che la crescita di difficoltà di un un progetto software **non** è lineare come può esserlo in altre discipline; in caso di progettazione informatica il peggioramento è di $O(n^2)$ o $O(2^n)$ a seconda di come lavora il calcolatore, se in parallelo o serialmente.
Inoltre grandi dimensioni fanno crescere in modo esponenziale anche *gli stati del programma* e *i cammini esecutivi* e grandi progetti richiedono molti sviluppatori.
Legge di Brooks: l'aggiunta di forza lavoro ad un progetto in ritardo causerà un ritardo ulteriore;

Perchè è difficile?

I grandi progetti sono difficili da visualizzare e modellizzare, poichè nessuno ne ha mai una visione complessiva ma conosce alla perfezione soltanto il proprio modulo e talvolta delle modifiche improvvise o cambi di strategie necessitano di comunicazione tra più parti differenti.
E' inoltre necessario rispettare i requisiti, le specifiche di progettazione e di sviluppo, eseguire test che validino il codice per ogni possibile scenario e soprattutto deve essere garantita una manutenzione non eccessivamente impegnativa ed onerosa in termini di impegno di persone ed economico.

Una delle gestioni più difficoltose dell'ingegneria del software è quella relativa al cambiamento e al miglioramento del software e delle sue specifiche. In questo caso i *modelli astratti* possono aiutare notevolmente ad astrarre e comprendere meglio le specifiche e la loro esecuzione;
Si devono tenere conto la progettazione, la modellazione, la qualità e la verifica, gli aspetti gestionali, gli aspetti legali e l'etica professionale

Processi SW:

Un processo software è un insieme di attività che porta alla craezione di un prodotto software; ne esistono di vari tipi ma la maggior parte si accomuna perchè include la *definizione dei requisiti* per il rispetto dei vincoli, *la progettazione e implementazione* del prodotto che rispetta i requisiti, *la convalida e l'evoluzione* in funzione dei cambiamenti. Il tipo di processo software che si deve preferire dipende soprattutto dall'applicazione che si sta andando a progettare e dal campo di operatività di quest'ultima:
Se un sistema critico necessiterà di un processo molto strutturato, sistemi azienddali meno stringenti necessiteranno di processi agili e flessibili.

Modello: un modello di processo software è una rappresentazione astratta del processo che include la definizione delle **attività**, dei loro **prodotti**, i **ruoli** delle persone coinvolte...
Ne esistono di varie tipologie:

Modello sequenziale:

Ogni fase deve essere teerminata prima di pasasre alla successiva; un esempio di questo tipo di sviluppo è quello a cascata.

Modello a cascata:

Le attività fondamentali sono fasi del processo separate affrontate sequenzialmente e la fase successiva non deve essere iniziata prima di aver terminato la precedente

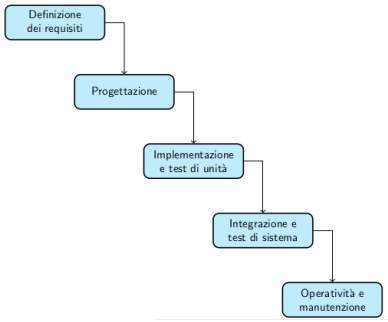

Figura 1: Modello a cascata, con i compiti eseguiti sequenzialmente

- **Analisi e definizione dei requisiti:**si stabiliscono le funzionalità, i vincoli e gli obiettivi del sistema → *specifiche di sistema*
- **Progettaizione:** identificazione delle *astrazioni* fondamentali, definizione dell'*architettura*, modellazione del sistema
- **Test unità:** realizzazione del programma e verifica del funzionamento delle sue parti
- **Integrazione e test:** il progetto complessivo viene cpompletato unendo le sue componenti e testato con un test finale che ne verifichi i requisiti
- **Operatività e manutenzione:** il sistema viene installato, messo in opera, con la correzione di eventuali errori scoperti in seguito e aggiunta di funzionalità richieste
- **Documentazione:** il risultato di ogni attività è un insieme di documenti approvati.

Vantaggi:

Il risultato di ogni attività è documentato e il processo utilizzato è quanto di più simile ci sia ad altri processi ingegneristici (ottimo epr progetti ibridi con altre branche)

Svantaggi:

Necessita di stabilire tutti i requisiti in anticipo e nessun prodotto tangibile esiste fino al completamento del progetto in toto. Non vi è inoltre serializzazione delle attività e la reazione ai cambiamenti nei requisiti o nella tecnologia è difficoltosa

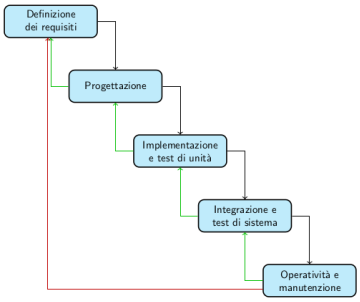

Figura 2: Modello a cascata modificato, con i feedback tra le fasi

Modello a cascata modificato:

E' possibile modificare il modello a cascata introducendo un feedback tra le fasi successive, mitigando la suddivisione netta tra le fasi e ripetendo (*iterazioni*) l'intero processo, andando però a snaturare il modello.

Sviluppo evolutivo:

Lo sviluppo **evolutivo** (o iterativo) viene eseguito con costruzione fin dall'inizio di versioni parziali del sistema digitale, attività intrecciate tra di loro con feedback rapido e giudizio degli utenti sulle verioni parziali ed è tipicamente di due tipologie:

- **Esplorativo:** lavoro a contatto con il cliente, iniziando già con requisiti più chiari e integrando nuove funzionalità proposte
- **Usa e getta:** prototipi vengono usati per capire i requisiti meno chiari partendo dai più chiari e per sperimentare approcci alla progettazione, con l'idea ad ogni iterazione di riscrivere il progetto

Vantaggi:

la soddisfazione del cliente è immediata e i requisiti raffinati mentre il cliente li comprende, con una maggior flessibilità ai cambiamenti

Svantaggi:

il processo non è visibile, la documentazione può essere incompleta e i continui cambiamenti rischiano di deteriorare la struttura del progetto, con specifiche incomplete fino alla fine

Modello a Spirale:

Lo sviluppo del sistema segue una spirale verso l'esterno, da una descrizione sommaria dei requisiti fino al sistema finale completamente sviluppato. A

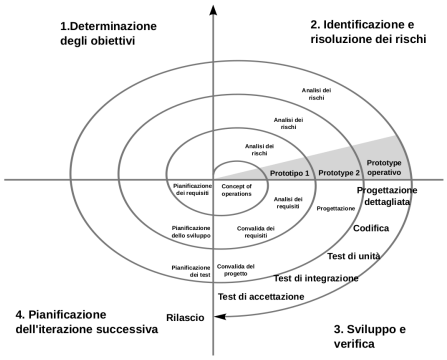

Figura 3: Modello a spirale, esempio di modello evolutivo

differenza di altri modelli viene riconosciuto il concetto di **rischio**, ovvero il fatto che qualcosa possa adare male, sia a livello sw che hw. Ogni giro è diviso in 4 settori:

- *Determinazione degli obiettivi:* definizione degli obiettivi dell'iterazione
- *Valutazione e riduzione del rischio:* sviluppo di un prototipo per analizzare i rischi e ridurli
- *Sviluppo e verifica:* seconda modalità coerenti con i rischi individuati
- *Pianificazione:* revisione del progetto e piani per l'iterazione

Sviluppo incrementale:

Soluzione intermedia tra il modello a cascata e quello evoluzionistico: lo sviluppo del sistema viene diviso in **incrementi**, a volte chiamati sprint, dove ciascuno di essi viene completato e consegnato al cliente prima di procedere con il successivo. Partendo dagli incrementi con le funzoinalità più importanti. Durante lo sviluppo si procede con l'analisi dei requisiti dell'incrememnto successivo, ma senza modificare quelli dell'incremento in via di sviluppo → *ogni sistema termina con una versione usabile*

Vantaggi:

I clienti non devono aspettare il sistema completo e i primi incrementi fungono anche da prototipi per determinare i requisiti dei successivi; le funzoini più importanti vengono sconsegnate per prime e verificate più a lungo.

Svantaggi:

Difficoltà nell'individuar incrementi "sufficientemente piccoli" e alcune funzionalità di base sono comuni a tutti gli incrementi ma verranno progettate pensando solo al primo incremento

Processi *pesanti* e *leggeri*:

In un processo **pesante** il lavoro è svolto lentamente e sistematicamente con l'obiettivo di consegnare un prodotto completo che richieda controlli e modifiche minimi
In un processo **leggero** la pianificazione è adattiva e si adottano pratiche che incoraggiano una risposta rapida e flassibile ai cambiamenti

Metodi *agili*:

Metodi basati su processi leggeri, solitamente incrementali, riassunti nel manifesto per lo sviluppo agile. In questo tipo di sviluppo contano maggiormente gli individui che le interazioni tra i processi e la collaborazione con il cliente più che la negoziazione dei contratti

Unified Process:

E' un esempio di modello ibrido che si adatta ad approcci evolutivi o incrementali, leggeri o pesanti, profondamente basato sull'UML e orientato all'analisi e progettazione per programmazione orientata ad oggetti. Spesso viene adottato nella versione RUP (*Rational Unified Process*)

Fasi:

Le fasi di sviluppo con l'unified process sono quattro, eseguite in sequenza

- **Ideazione (inception):** visione approssimativa, studio economico, stime approssimative di costi e tempi
- **Elaborazione:** identificazione della maggior parte dei requisiti, risoluzoine di rischi maggiori, implementazione iterativa del nucleo dell'architettura, stime realistiche dei costi
- **Costruzione:** implementazione iterativa degli elementi rimanenti e preparazione al rilascio
- **Transizione:** completamento del progetto, verifiche finali e rilascio effettivo agli utenti

N.B. le fasi non corrispondono ad attività come nel modello a cascata—

- Ideazione:

Elaborazione del concetto di studio di fattibilità, stima iniziale dei costi del budget e analisi di mercato e del contesto commerciale. A questi si aggiungono analisi dei requisiti principali e analisi preliminare dei *casi d'uso*, con individuazione dei rischi principali con successiva pianificazione approssimativa del progetto. Al termine di questa fase si deceide se proseguire o abbandonare il progetto

- Elaborazione:

In questa fase lo scopo principale è la **mitigazione dei rischi maggiori**. Il progetto inizia a prendere forma attraverso un'analisi del domimio e della maggior parte dei casi d'utilizzo, descrivendo l'architettura software che caratterizzarà il progetto. Viene inoltre realizzato un prototipo funionante dell'architettura e un piano complessivo di progetto. Il risultato è una collezione di modelli di dominio, casi d'uso, architetturale... con almeno un prototipo preliminare di qualche caso d'uso significativo

- Costruzione:

L'obiettivo principale è la costruzione del sistema e la sua realizzazione, con lo sviluppo dei componenti, l'implementazione delle funzoinalità, anche attraaverso numerose **iterazioni**. Al termine di questa fase si ha una prima release del sistema

- Transizione:

L'obiettivo è il passaggio dallo sviluppo alla produzione, con il sistema che deve essere reso disponibile per l'utente finale (con quest'ultimo che viene istruito) e il tutto finisce con una convalida terminale/beta testing

Iterazioni:

Ciascuna fase è composta da iterazione; la fase di ideazione include una sola iterazione, le altre fasi includono più iterazioni.
Le iterazioni hanno una dirata fissa (**timeboxing**) a seconda del tipo di progetto; in caso di ritardo si riducono gli obiettivi dell'iterazione pur di rispettarne la scadenza

Discipline:

Le attività lavorative sono collocate in **discipline**, di cui sei *ingegneristiche:*

- Business modelling
- Analisi requisiti
- Progettazione
- Implementazione
- Test
- Installazione

E tre di tipo *supporto*:

- Gestione delle configurazioni e delle modifiche
- gestione del progetto
- infrastruttura

Ciascuna iterazione coinvolge tutte o quasi le discipline in modo diverso, con le iterazioni iniziali che si concentrano su requisiti e progettazione e quelle finali su test e installazione Affrontare le questioni fronte di rischi maggiori

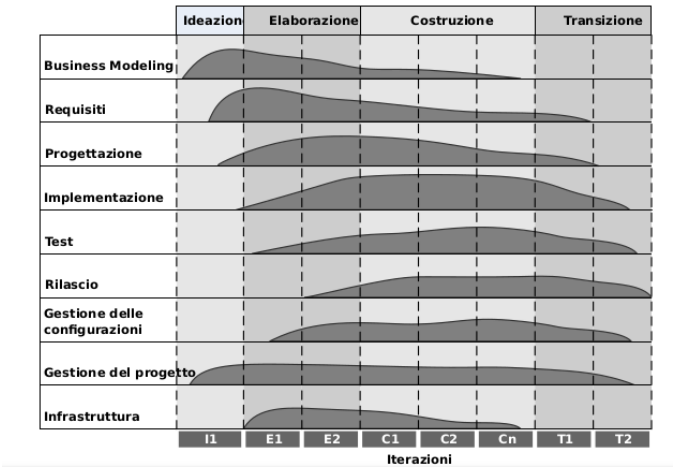

Figura 4: Andamento delle discipline durante lo sviluppo

e di maggiore valore nelle iterazioni iniziali, impegnando continuamente gli utenti sui requisiti e sulla convalida. Verificare inoltre di continuo la qualità tramite numerosi test e usare strumenti per la modellaizone visuale UML.

Controllo versione:

Un sistema di controllo versione (**VCS, Version Vontrol System**) è un sistema software che gestisce i cambiamenti nei progetti (a volte sono integrati nei software, altre colte sono strumenti autonomi

Sistemi locali:

Erano i primi VCS, e si limitavano a mantenere un databae delle versioni di ciascun file del progetto, chiamato *repository*; i più diffusi erano SCCS (source code control system) e RCC (revision control system) e i comandi principali erano il chechin e il checkout dei file

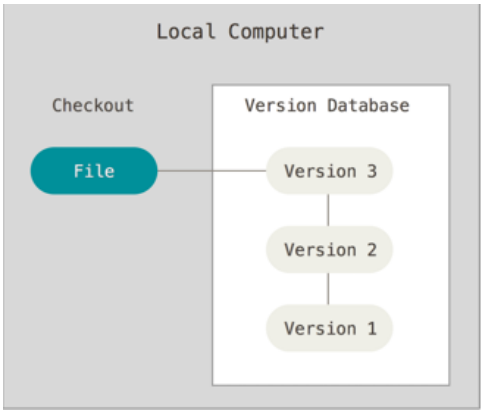

Figura 5: Struttura dei primi vcs locali

Sistemi centralizzati:

I vcs centralizzati consentono a più utenti di collaborare ad un progetto (CVS (concurrent version system) e Subversion) attraverso comandi di update e commit. era tuttavia necessario gestire il problema dei *conflitti*, in particolare per quel che riguarda le modifiche parallele al progetto: esse richiedono dei meccanismi di risoluzione dei conflitti per evitare che un utente modifichi un file in corso di modifica da parte di un altro utente. Due le possibili soluzioni:

- meccanismo di **lock** per bloccare i file in corso di modifica
- **merge** delle modifiche con possibile risoluzione manuale dei conflitti

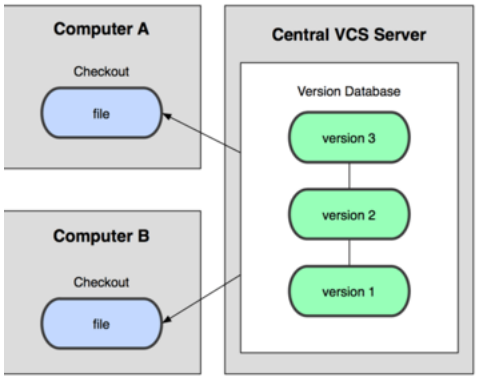

Figura 6: Struttura generica dei sistemi centralizzati

Modifiche ingenti che riguardano trasversalmente molti file sono affrontate attraverso l'utilizzo del meccanismo di **branch**

Sistemi distribuiti:

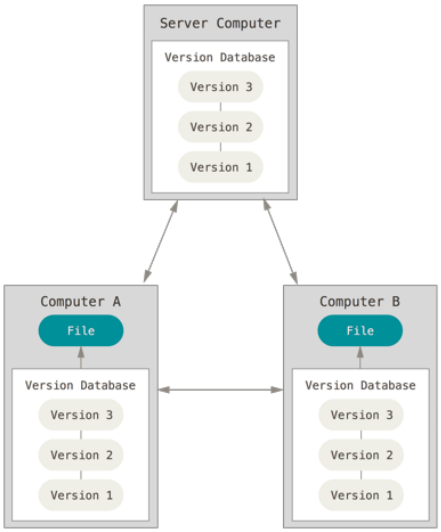

Figura 7: Struttura dei distribuiti

In un VCS distribuito ci sono più copie del progetto ospitate da repository multipli (ogni repo locale ha una copia dei repo distribuiti, più o meno

sincronizzata. I più famosi sono Git, Mercurial, Bazaar. I vantaggi principali sono la separazione tra l'archiviazione e la condivisione, la possibilità di meccanismi di collaborazione complessi (e.g. repository gerarchiche) e affidabilità garantita dalle copie distribuite.

GIT:

Creato da Torvalds nel 2005 implementa un design semplice, efficiente e completamente distribuito, in grado di gestire progetti di grandi dimensioni e supporto allo svilupop non lineare. In git la maggior parte delle operazioni

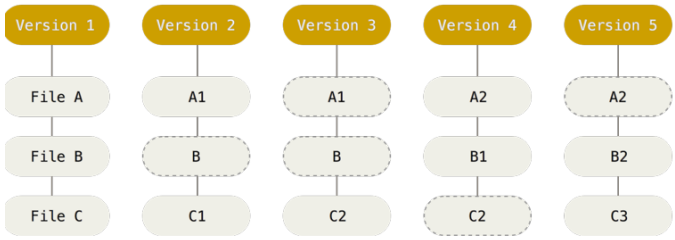

Figura 8: Struttura di git: un repo è visto come una sequenza di snapshot del progetto

avviene localmente e le più comuni sono:

- modific dei file nella working copy
- aggiunta delle modifiche alla **staging area (o index)**
- **commit** delle modifiche che vengono aggiunte al repository locale

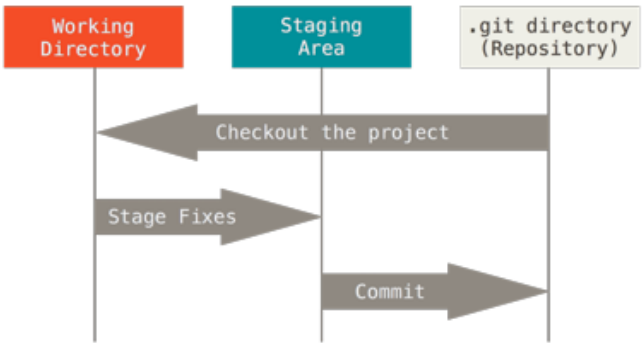

Figura 9: Gestione locale di git

Esistono anche comandi di sincronizzazione remota per i repository globali che aggiornano il locale sincronizzandolo con il globale (*git pull*) e che pubblicano la versione locale sul globale (*git push*)

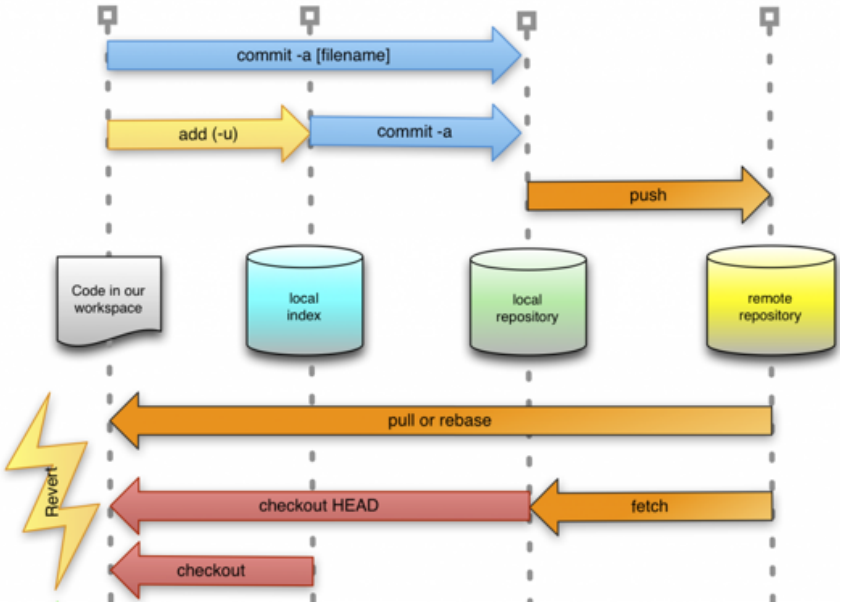

Figura 10: Struttura dei comandi remoti

Analisi dei requisiti:

Una componente fondamentale dello sviluppo software è l'analisi dei **requisiti**, ovvero:

- descrizione dei servizi forniti
- descrizione dei vincoli operativi

Essi riflettono l'esigenza dell'utente di avere un sistema che aiuti a risolvere determinati problemi.

Requisiti:

Può essere utilizzato come termine per indicare due aspetti principali:

- **Requisiti utente**: sono espressi in linguaggio naturale usando termini informali
- **Requisiti di sistema:** sono più dettagliati e talvolta definiti *specifiche di sistema*

La necessità di utilizzare due livelli nasce dal fatto che talvolta ci si rivolge ad utenze completamente differenti per specificare i requisiti, e parlare con sviluppatori o con committenti fa la differenza nel linguaggio da utilizzare. I requisiti si possono dividere anche in categorie:

- *Requisiti funznoiali*: descrivono i servizi che deve offrire il sistema (specifica dell'output in corrispondenza di determinati input e specifica dei comportamenti del sistema in determinate situazioni)
- *Requisiti NON funzionali*: descrivono vincoli sui servizi offerti (vincolo di tempo, di usabilità etc etc)
- *Requisiti di dominio*: derivano dal dominio applicativo del sistema e possono essere funzionali o non

Requisiti funzionali:

Sono **requisiti utente** che descrivono informalmente cosa deve fare il sistema e **requisiti di sistema** che descrivono precisamente e dettagliatamente i servizi specificando ad esempio l'esatta relazione tra input e output. E' importante che le specifiche siano **complete** e **coerenti** tra loro; nello scrivere le speficihe inoltre bisogna tener conto di tutte le persone potenzialmente influenzate dal sistema (**stakeholder**).

Requisiti NON funzionali:

Si riferiscono a proprietà del sistema come tempi di risposta, affidabilità, occupazione di memoria etc etc Raramente sono legati a speciffiche funzionalità ma riguardano l'intero sistema e talvolta sono più critici dei requisiti funzionali. Possono essere ulteriormente divisi in categorie:

- **Requisiti del prodotto:** ne specificano alcuni aspetti del suo comportamento come usabilità, efficienza, portabilità
- **Requisiti organizzativi:** derivano da procedure politiche e pratiche delle organizzazioni del cliente e dello sviluppatore, come consegna, standard, implementazione
- **Requisiti esterni:** non derivano dal sistema o dal suo processo di sviluppo e sono legati all'etica o alla legislazione

Esempi di requisiti non funznionali sono come l'interfaccia utente debba essere realizzata con una semplice pagina html senza applet, come il processo di sviluppo e la consegna dei documenti debbano seguire standard XYZ o come il sistema non debba rivelare agli operatori nessun dato dell'utente oltre al nome ed al numero identificativo. Se possibile andrebbero indicati in modo da risultare *verificabili*, se la verifica non richiede uno sforzo eccessivo (e.g. il sistema deve essere semplice da utilizzare per operatori inesperti → NON verificabile; gli operatori esperti dovrebbero essere in grado di utilizzare tutte le funzioni dopo due ore di addestramento → verificabile). Quando è possibile andrebbero specificati quantitativamente:

- velocità (tempi di risposta)
- spazio (quantità di memoria)
- usabilità (tempo di addestramento)
- affidabilità
- portabilità (numero di architetture destinazione)

Requisiti di dominio:

Derivano dal dominio applicativo e sono espressi con termini del dominio facendo riferimenti a concetti propri del dominio. Sono spesso requisiti specialistici e per essere compresi potrebbero richiedere una formazione specifica (e c'è il rischio che per gli esperti di dominio alcuni requisiti appaiano talmente ovvi che non vengano esplicitati) E.g. tutte le basi di dati devono potersi interfacciare con lo standard XYZ o a causa di restrizioni alcuni documenti devono essere cancellati dopo che sono stati inviati etc etc

Processi di ingegneria e requsiti:

Lo scopo del processo di ingegneria dei requisiti è il mantenimento di un documento dei requisiti di sistema e include 4 sottoprocessi:

- **Studio di fattibilità**: l'obiettivo è rispondere alle domande sul sistema e il risultato dello studio porta ad una decisione sul proseguimento o meno del progetto
- **Deduzione e analisi dei requisiti**: si raccolgono informazioni sul dominio applicativo, sulle funzionalità richieste dal sistema e sui vincoli operativi In questa fase si individuano gli **stakeholder**, ovvero tutti coloro che sono influenzati dal sistema e le loro necessità; la raccolta dei requisiti può avvenire con interviste, individuazione di scenari operativi, studi etnografici...
- **Specifica dei requisiti:** i requisiti devono essere raccolti in documenti appositamente strutturati ed esistono standard appositi come l'IEEE/ANSI 830-1998 dove viene indicata la struttura (divisa in introduzione, divisione generale, requisiti, appendici, indice). La descrizione dei requisiti utente dovrebbe essere chiaramente comprensibile per l'utente finale (linguaggio naturale, descrizione solo dei comportamenti visibili, nessuna confusione tra requisiti distinti, motivare i requisiti...) Essa inoltre dovrebbe far riferimento solamente al comportamento esterno del sistema e ai suoi vincoli operativi, *senza indicare scelte progettuali e usando una notazione chiara e precisa* Se inoltre il sistema deve poter interagire con altri sistemi occorre specificare le interfacce da utilizzare per il collegamento (*interfacce procedurali, strutture dati, rappresentazione dei dati, protocolli di comunicazione...*
- **Convalida dei requisiti:** errori nell'analisi dei requisiti possono determinare conseguenze gravi nel seguito del progetto e andrebbero convalidati per verificarne: **la validità** (il fatto che espreimano in modo appropriato le necessità degli stakeholder), **consistenza** (non contradditorietà), **completezza** (la copertura di tutte le funzionalità

e di tutti i vincoli), **il realismo** (il fatto che possano essere tecnicamente soddisfatti considerando anche tempo e budget) e **la verificabilità** (la possibilità di verificare oggettivamente il soffisfacimento) Le tecniche di convalidà includono *revisione dei requisiti* (da parte di un team di revisors), *prototipizzazione* (in cui un modello eseguibile viene preparato e mostrato agli utenti per averne un feedback) e *generazione di casi di test* (che mettono in luce la verificabilità dei requisiti spesso rivelando anche molti altri difetti nei requisiti stessi)

Gestione dei requisiti:

I requisiti sono in continua trasformazione e il problema può essere troppo complesso per poterlo descrivere in anticipo. L'ambiente tecnico aziendale del sistema possono subire mutamenti che portano alle variazioni, durante lo sviluppo cambia ruolo degli stakeholder etc ect Esistono pertanto:

- Requisiti Duraturi: tendono a rimanere invariati e spesso sono requisiti del dominio applicativo
- Requisiti Volatili: requisiti di tipo legale o vincoli tecnologici, mutabili

Use case:

I casi d'uso sono gli strumenti principali per l'analisi dei requisiti in molti modelli di processo, tra cui lo *Unified Process* Secondo il modello UP l'analisi dei requisiti si svolge durante tutta la durata del progetto, concentrandosi però nelle prime iterazioni. I casi d'uso sono storie scritte, testuali, ampiamente utilizzati per scoprire ed analizzare i requisit (raccontano di qualche **attore** che usa il sistema per **raggiungere un obiettivo**).
e.g. *elabora vendite: un cliente arriva alla cassa con alcuni articoli da acquistare. Il cassiere utilizza il sistema POS per registrare ogni articolo acquistato. Il sistema mostra il totale e i dettagli per ogni articolo. Il cliente inserisce informazioni sul pagamento, che il sistema convalida e registra. Il sistema aggiorna l'inventario. Il cliente riceve dal sistema una ricevuta e poi se ve va con gli articoli acquistati.* **N.B.** i casi d'uso non sono diagrammi, bensì testo; UML tuttavia definisce anche *i diagrammi dei casi d'uso* che vengono usati per rappresentarli in forma sommaria (benchè siano secondari al testo)

- **Attore:** qualcosa o qualcuno dotato di un comportamento
- **Scenario:** (o **istanza di caso d'uso**) è una sequenza di azioni e interazioni tra il sistema e alcuni attori
- **Caso d'uso:** è una collezione di scenari correlati che descrivono un attore che usa il sistema per raggiungere l'obiettivo.

Casi d'uso:

ESEMPIO:***Scenario di successo principale****: un cliente arriva alla cassa con alcuni articoli da restituire. Il cassiere utilizza il sistema POS per registrare ciascun articolo restituito. . .* E' tuttavia necessario per ogni caso d'uso analizzare possibili *scenari alternativi*

Perchè i casi d'uso?

Perchè rispetto ad altri modelli offrono vantaggi in termini di semplicità tale che gli esperti di dominio e i fornitori possano partecipare alla loro stesura, mettono in risalto gli obiettivi e il punto di vista dell'utente e si prestano a

diversi livelli di dettaglio
I casi d'uso **sono** requisiti (benchè non tutti i requisiti siano rappresentabili da casi d'uso); i casi d'uso si prestano a sappresentare i requisiti funzionali.

Modello dei casi d'uso:

In UP il **modello dei casi d'uso** comprende l'insieme di tutti i casi d'uso scritti e opzionalmente un diagramma in notazione UML che mostra i nomi di attori e casi d'uso coinvolti **Caasi d'uso e metodi alternativi:** la scrittura e il raffinamento dei casi d'uso procede per tutta la durata del progetto iniziando da quelli più significativi, usando un livello di detatglio dipendente dall'importanza del caso d'uso e rivedendo i modelli man mano che si raccolgono feedback. Si utilizzano diversi **formati:**

- *Formato breve:* un paragrado per ciascun caso d'uso, descrivendo normalmente solo il caso di successo e utile durante un'analisi iniziale
- *Formato informale:* un paragrafo per ciascun scenario, raccomandato per la maggior parte dei casi d'uso
- *Fromato dettagliato:* descrizione di tutti i passi e di tutte le possibili varaizioni, includendo anche sezioni aggiuntive

Struttura dei casi:

Un caso d'uso dovrebbe rappresentare una situazione di cui uno specifico attore ottiene un risultato **osservabile di valore**
Si deve porre enfasi sugli utenti e sui loro obiettivi, comprendendo ciò che gli utenti considerano di valore. Il tutto deve essere scritto con modalità **black box**, specificando cosa deve fare il sistema e senza indicare come. Andrebbe inoltre mantenuto uno stile conciso ed essenziale concentrandosi sullo scopo. La procedura di base è:

1. Determinare i confini del sistema
2. Identificare gli attori primari
3. Identificare gli obiettivi di ciascun attore primario
4. Definire i casi d'uso che soddisfano gli obiettivi

Requisiti ed elaborati:

UP prevede diversi elaborati che riguardano i requisti:

- **Modello dei casi d'uso**
- **Specifiche supplementari**

- **Glossario**
- **Documento di visione**
- **Regole di business**

Modellazione del dominio:

Modelli di sistema:

Non sempre il linguaggio naturale permette di rappresentare i requisiti di sistema in modo sufficentemente dettagliato, pertanto si utilizzano rappresentazioni grafiche (**modelli di sistema**) che possono descrivere il sistema in modo dettagliato; possono inoltre aiutare a definire i problemi che si stanno affrontando, i processi aziendali e alcuni aspetti del sistema che si sta sviluppando.
Un modello di sistema **non** è una sua rappresentazione, ma una sua ***astrazione*** che include soltanto gli aspetti significativi ignorando i dettagli irrilevanti. Esistono diversi tipi di modelli, basati su diversi tipi di astrazione:

- **Modelli del flusso dei dati**, detti anche dataflow, mettono in luce le trasformazioni dei dati
- **Modelli di machcina a stati**, descrivono le risposte a eventi interni o esterni del sistema
- **ER**, ovvero Entity Relationship, descrivono le relazioni semantiche tra i dati
- **Modelli a oggetti** descrivono le entità di interesse del dominio applicativo

Modello di dominio:

E' una rappresentazione visuale di classi concettuali o di oggetti del mondo reale di un dominio;

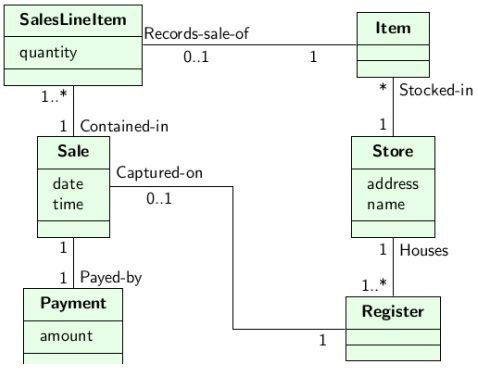

Figura 11: Esempio del modello di dominio di un pos

È lo strumento principale dell'analisi orientata agli oggetti e fornisce ispirazione per la progettazione orientata agli oggetti. ornisce inoltre un dizionario visuale degli elementi del dominio e come notazione consiste in uno o più dei *diagrammi delle classi* UML.
N.B.: in uesto caso le classi corrispondono a **concetti**, non ad artefatti software. L'idea alla base del modello di dominio è quella di offrire un salto rappresentazionale basso verso il modello software secondo un concetto fondamentale del paradigma ObjOri.
Vengono rappresentati elementi reali del dominio secondo alcune regole:

- Vengono riportati solo gli elementi più rilevanti
- Nesun elemento software a meno che non sia l'elemento stesso del dominio
- Nessuna responsabilità o metodo
- Non coincide con il modello dei dati

Creazione di un modello di dominio:

In UP il modello di dominio comprende i concetti relativi agli scenari dei casi d'uso attualmente considerati, partendo con pochi scenari dei casi d'uso più significativi, facendo crescere il modello nel corso delle iterazioni e svolgendo la maggior parte del lavoro nella fase di elaborazione.
I passi sono:

1. Identificate le classi concettuali
2. Disegno delle classi in un diagramma UML
3. aggiunta di associazioni e attributi

Classi concettuali:

Una classe concettuale è caratterizzata da tre elementi: **simbolo, intensione, estensione** (e.g. Vendita, evento di una transazione, insieme delle istanze di vendita). I metodi per individuare le classi concettuali sono tre, principalmente: - Riusare dei modelli esistenti - Usare un elenco di categorie - Analisi linguistica
Quando possibile, il primo modello è preferibile
Analisi lingustica: si tratta dell'identificazione di nomi e locuzioni nominali nelle descrizioni testuali del dominio che vengono considerati come candidati per classi concettuali e attributi di classi. I casi d'uso in formato dettagliato sono un ottimo punto di partenza per questo tipo di analisi; non tutti i nomi devono essere classi concettuali, poichè alcuni lo diventeranno con iterazioni successive mentre alcuni non rosultano rilevanti.

UC1: Elabora vendita
Scenario principale

1. Il cliente arriva alla cassa POS con gli articoli e/o i servizi da acquistare
2. Il cassiere inizia una nuova vendita
3. Il cassiere inserisce il codice identificativo dell'articolo
4. Il Sistema registra la riga di vendita per l'articolo e mostra la descrizione dell'articolo, il suo prezzo, il totale parziale

   Il cassiere ripete i passi 2–3 finché non indica che ha terminato
5. Il Sistema mostra il totale con le imposte calcolate
6. Il cassiere riferisce il totale al cliente e chiede il pagamento
7. Il cliente paga e il Sistema gestisce il pagamento
8. Il Sistema registra vendita completata e invia informazioni sulla vendita e sul pagamento ai sistemi esterni di contabilità e di inventario
9. Il Sistema genera la ricevuta
10. Il cliente va via con la ricevuta e gli articoli acquistati

Attributi:

Un attributo è un valore logico (un dato) di un oggetto (un'istanza di una classe). Il modello di dominio deve includere gli attributi suggeriti dai casi

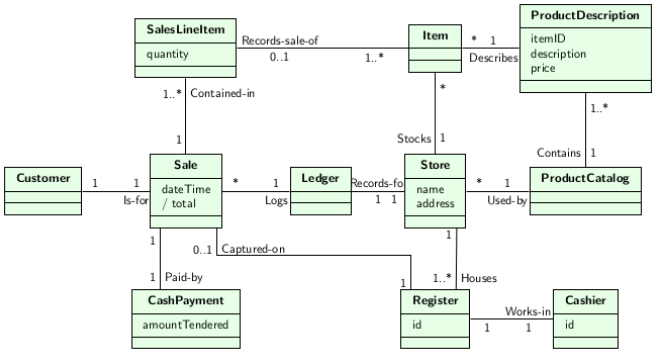

Figura 12: Esempio del modello di dominio di un pos

d'uso, ove vi sia la necessità di ricordare informazioni. La notazione UML consente di specificare numerosi dettagli riguardo gli attributi (tipo, molteplicità, visibilità, atttributi derivati...) benchè questi dettagli possano essere omessi nel modello di dominio e andrebbero riportati nel glossario.
Gli attributi devono essere di tipo semplice (senza corrispondere necessariamente a tipi di dato elementari per un linguaggio di programmazione)

Tipi di attributi: