INGEGNERIA DEL SOFTWARE

Alessandro Zappatore

Università del Piemonte Orientale
Anno accademico 2024/2025, 1° semestre

Indice

Parte I
Teoria

1 Introduzione

1.1 Cos'è l'Ingegneria del software

L'ingegneria del software è l'applicazione del processo ingegneristico allo sviluppo di prodotti software.

1.2 Sistemi software

Un *sistema software* è un insieme di componenti software che funzionano in modo coordinato allo scopo di informatizzare una certa attività. La realizzazione di un sistema software richiede l'impiego di un gruppo di lavoro, nel quale ogni persona ricopre un ruolo ben preciso e le attività dei vari gruppi vanno coordinate, e tempo da dedicare alle varie fasi di sviluppo.

Esistono due categorie di sistemi software:

- i **sistemi generici**, realizzati per essere utilizzati da tanti clienti, venduti sul mercato. Requisiti dettati dalle tendenze di mercato;
- **sistemi customizzati**, richiesti da uno specifico cliente (il committente).

1.3 Il processo software

Con *ingegneria del software* si intende l'applicazione del processo dell'Ingegneria alla produzione di sistemi software. Il processo è suddiviso in:

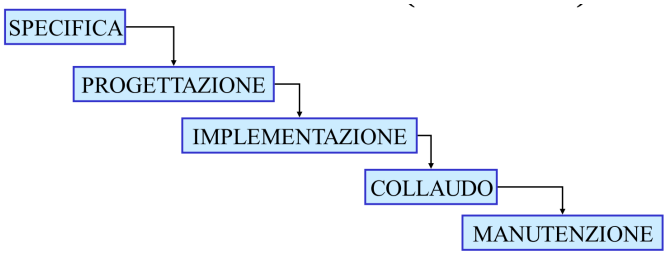
- **specifica**: definizione dei requisiti funzionali e non funzionali
- **progettazione**: si definiscono architettura, controllo, comportamento dei componenti, strutture dati, algoritmi, struttura del codice, interfaccia utente
- **implementazione**: scrittura del codice e integrazione dei moduli
- **collaudo**: si controlla se il sistema ha difetti di funzionamento e se soddisfa i requisiti
- **manutenzione**: modifiche del sistema dopo la consegna

1.4 Modelli di processo

1.4.1 Modello a cascata (waterfall)

Eseguo le 5 fasi in modo sequenziale (modello tradizionale):

$$specifica \rightarrow progettazione \rightarrow implementazione \rightarrow collaudo \rightarrow manutenzione$$



1.4.2 Modello iterativo

1. Si esegue un po' di specifica, un po' di progettazione, un po' di implementazione e un po' di collaudo;
2. Viene generato un prototipo che realizza un sottoinsieme dei servizi;
3. Si ripete dal punto 1 fino a realizzare tutti i servizi richiesti.

1.4.3 Altri modelli

- Modello agile;
- Modello a spirale;
- Tanti altri.

1.5 Gestione del processo

L'ingegneria del software si occupa anche della gestione del progetto, attraverso il **project manager**, che si svolge in parallelo al processo software.

Le principali attività di gestione sono l'**assegnazione** di risorse (umane, finanziarie...), la **stima del tempo** necessario per ogni attività, la **stima dei costi** e la **stima dei rischi**.

1.6 Strumenti CASE

CASE = **C**omputer **A**ide **S**oftware **E**ngineering

Sono tutti i tool informatici che supportano le fasi dell'ingegneria.

- **Specifica :** editor dei requisiti;
- **Progettazione :** editor dei diagrammi di progettazione;
- **Implementazione :** IDE, strumenti per condivisione dei file;
- **Collaudo :** strumenti per ispezione e testing automatico del codice;
- **Manutenzione :** strumenti per la gestione delle versioni del codice e della documentazione.

2 Specifica

La **specifica** è l'insieme di attività necessarie per generare il documento dei requisiti che descrive i *requisiti funzionali* e i *requisiti non funzionali*: descrive il "cosa" il sistema deve fare, non il "come". I requisiti servono per una proposta di contratto e modellare fasi successive del processo software.

Nella specifica non viene considerata l'architettura del sistema (interfaccia utente, database, $\cdots$).

2.1 Requisiti funzionali

I requisiti funzionali sono i servizi che il cliente richiede al sistema. Per ogni servizio si descrive:

- cosa accade nell'interazione tra utente e sistema;
- cosa accade in seguito ad un certo input o stimolo;
- cosa accade in particolari situazioni, ad esempio in caso di eccezioni.

Il sistema viene considerato come un'unica entità, non viene descritto come funziona internamente il sistema, in quanto è oggetto della successiva fase di progettazione.

2.2 Requisiti non funzionali

2.2.1 Requisiti di prodotto

Descrivono la qualità del prodotto. Una **proprietà emergente** è una proprietà che "emerge" dal funzionamento del sistema, dopo che è stato implementato.

Esempi:

- **Usabilità** (requisiti che riguardano la facilità d'uso del sistema);
- **Mantenibilità** (facilità di manutenzione del sistema eg. stabilire quali tipi di commenti aggiungere nel codice);
- **Portabilità** (facilità con cui posso far migrare il sistema da una piattaforma ad un altra);
- **Efficienza** (eg. tempo di risposta del sistema, quantità di memoria occupata);
- **Affidabilità** (grado di fiducia con cui si ritiene che il sistema funzioni correttamente);
  - *Reliability* (il sistema deve funzionare per un certo periodo di tempo senza avere dei malfunzionamenti, eg. 90% di reliability dopo un anno che non abbia problemi, MTTF);
  - *Availability* (probabilità con cui il sistema fornisce il sistema nel momento in cui è richiesto);
  - *Safety* (capacità del sistema di non causare danni alle persone, ambiente di natura economica);
  - *Security* (sicurezza dagli attacchi informatici).
- **Recoverability** (capacità del sistema di tornare disponibile dopo un crash, recuperare dati persi).

2.2.2 Requisiti organizzativi

Caratteristiche riguardanti le fasi del processo software o la gestione del progetto.

- **Requisiti di sviluppo** Sono tutti i vincoli relativi alle tecniche di sviluppo del software.
  - Metodi e tecniche (eg. no ricorsione);
  - Linguaggi;
  - Tool.
- **Requisiti gestionali**
  - Tempo di consegna;
  - Risorse da usare (eg. persone, hardware, licenze);
  - Costi di sviluppo.

2.2.3 Requisiti esterni

Tutti i requisiti che provengono dall'esterno.

- **Requisiti giuridici**
  - Safety;
  - Standard;
  - Privacy;
- **Requisiti di compatibilità** (eg. il sistema si interfaccia con un altro sistema)

2.3 Sistemi critici

Un sistema viene detto critico quando un suo malfunzionamento può causare danni:

- **Safety critical system**
  - Persone;
  - Ambiente;
  - Strutture.
- **Business critical system**
  - Economico.

2.4 Costo dell'affidabilità

Il costo cresce in modo esponenziale rispetto al grado di affidabilità richiesto.

- Tecniche di tolleranze ai guasti (fault-tolerance);
- Tecniche per proteggersi da attacchi.

2.5 Processo di specifica

Il *processo di specifica* è il processo per generare il documento dei requisiti, ed è diviso in più fasi. Lo stesso requisito viene definito con due gradi di dettaglio diversi. Il *requisito utente* è descritto ad alto livello, in linguaggio naturale, ed è il risultato della deduzione dei requisiti. Il *requisito di sistema* è descritto dettagliatamente, fornendo tutti i dettagli necessari per la fase di progettazione, ed è il risultato dell'analisi dei requisiti.

2.5.1 Studio di fattibilità

Lo *studio di fattibilità* è la valutazione della possibilità di sviluppare il sistema e dei suoi vantaggi per il committente. Si decide se la costruzione del sistema è fattibile date le risorse disponibili e se il sistema è effettivamente utile al cliente. Per svolgere lo studio si raccolgono informazioni e si prepara un rapporto di fattibilità, che contiene la valutazione della possibilità di costruire un sistema e dei vantaggi che possono derivare dalla sua introduzione.

2.5.2 Deduzione dei requisiti

La *deduzione dei requisiti* è la raccolta di informazioni da cui dedurre quali sono i requisiti. Le informazioni si possono raccogliere mediante uno studio del dominio applicativo del sistema richiesto, il dialogo con stakeholder, studio di sistemi simili già realizzati e studio di sistemi con cui dovrà interagire quello da sviluppare.

Dominio applicativo è l'insieme di entità reali su cui il sistema software ha effetto.

Stakeholder è, in ambito economico, il soggetto che può influenzare il successo di un'impresa o che ha interessi nelle decisioni dell'impresa; in ambito del processo software sono persone che possono influenzare il processo o che hanno interesse nelle decisioni assunte in esso.

È possibile dialogare con gli stakeholder tramite *interviste*, nelle quali viene chiesto di raccontare attraverso degli esempi reali come l'attività lavorativa funziona realmente, e tramite *etnografia*, l'osservazione dei potenziali utenti nello svolgimento delle loro mansioni.

Il dialogo con gli stakeholder presenta vari problemi, in quanto non sono in grado di indicare chiaramente cosa vogliono dal sistema, omettendo informazioni ritenute ovvie ed utilizzando terminologia non adatta. Inoltre, lo stesso requisito può essere espresso da più stakeholder in maniera differente, ed addirittura essere in conflitto.

Molti problemi scaturiscono dal *linguaggio naturale*, in quanto una descrizione ad alto livello di un requisito può generare confusione. La soluzione è quella di utilizzare il linguaggio in modo coerente, evitando gergo tecnico ed illustrando i requisiti tramite semplici diagrammi.

2.5.3 Analisi dei requisiti

L'*analisi dei requisiti* è l'organizzazione, negoziazione e modellazione dei requisiti.

Comprende:

- **classificazione e organizzazione dei requisiti**
- **assegnazione di priorità ai requisiti**: si stabilisce il grado di rilevanza di ogni requisito
- **negoziazione dei requisiti**
- **modellazione analitica dei requisiti**: produzione di modelli che rappresentano o descrivono nel dettaglio i requisiti

Requisiti di sistema sono l'espansione dei requisiti utente, e formano la base per la progettazione. Il linguaggio naturale non è adatto alla definizione di un requisito di sistema, quindi è necessario usare template, modelli grafici o notazione matematica.

Modello data-flow detto anche pipe & filter, permette di modellare il flusso e l'elaborazione dei dati, ma non prevede la gestione degli errori. L'elaborazione è di tipo batch: input $\rightarrow$ elaborazione $\rightarrow$ output.

I requisiti non funzionali si possono specificare definendone delle misure quantitative:

- *efficienza*: tempo di elaborazione delle richieste, occupazione di memoria
- *affidabilità*: probabilità di malfunzionamento, disponibilità
- *usabilità*: tempo di addestramento, aiuto contestuale

Documento dei requisiti contiene il risultato della deduzione e dell'analisi, ed è la dichiarazione ufficiale di ciò che si deve sviluppare. Il documento contiene una breve introduzione che descrive le funzionalità del sistema, un glossario contenente le definizione di termini tecnici, i requisiti utente e i requisiti di sistema, correlati con modelli UML. Il documento è letto da tutte le figure coinvolte nella realizzazione del progetto.

2.5.4 Validazione dei requisiti

La *validazione dei requisiti* è la verifica del rispetto di alcune proprietà da parte del documento dei requisiti, serve ad evitare la scoperta di *errori di specifica* durante le fasi successive del processo software. Sono da verificare le seguenti proprietà:

- **completezza**: tutti i requisiti richiesti dal committente devono essere documentati;
- **coerenza**: la specifica dei requisiti non deve contenere definizioni tra loro contraddittorie;
- **precisione**: l'interpretazione di una definizione di requisito deve essere unica;

- **realismo**: i requisiti devono essere implementati date le risorse disponibili;
- **tracciabilità**: quando si modifica un requisito bisogna valutarne l'impatto sul resto della specifica: è quindi necessario tracciarlo. Vari tipi:
  - *tracciabilità della sorgente*: reperire la fonte d'informazione relativa al requisito
  - *tracciabilità dei requisiti*: individuare i requisiti dipendenti;
  - *tracciabilità del progetto*: individuare i componenti del sistema che realizzano il requisito;
  - *tracciabilità dei test*: individuare i test-case usati per collaudare il requisito.

Per validare i requisiti si può impiegare un gruppo di *revisori* che ricontrolli i requisiti e *costruire dei prototipi*.

3 Progettazione

3.1 Attività di progettazione

Durante la fase di *progettazione architetturale* viene definita la struttura del sistema, come questo verrà distribuito e come il sistema si dovrà comportare. Sono inoltre progettate le strutture dati, gli algoritmi e la GUI.

3.1.1 Progettazione architetturale

- **Strutturazione del sistema:** definire quali sono i sottosistemi e i moduli interni ad ogni sottosistema;
- **Metodo di controllo:** stabilire quali sono i componenti che invocano le operazioni e quali le eseguono;
- **Modellazione del comportamento:** per ogni servizio stabilire come i componenti interagiscono tra di loro.

Posso utilizzare diagrammi UML.

3.1.2 Progettazione delle strutture dati

Se è presente un database dovrò progettare le strutture dati. (Eg. stabilire le tabelle, chiavi, $\cdots$).
Modello entità-relazione, modello relazionale, UML (diagramma delle classi).

3.1.3 Progettazione degli algoritmi

Gli algoritmi da implementare per ogni metodo.
Diagrammi di flusso, UML (diagrammi delle attività).

3.1.4 Progettazione dell'interfaccia utente (grafica)

Se necessaria.
Disegnare su carta/tool per l'aspetto dell'interfaccia grafica.

3.2 Progettazione architetturale

3.2.1 Strutturazione

Stile stratificato Ogni strato (solitamente 3) interagisce con gli strati adiacenti

- **Presentazione:** interfaccia utente;
  - raccolta degli input dell'utente;
  - visualizzazione output all'utente.
- **Elaborazione:** applicazione;
  - elabora dati in input e produce dati in output.
- **Gestione dei dati:** database.
  - aggiunta, modifica, rimozione di dati;
  - ricerca di dati.

Sottosistema Parte del sistema dedicata a svolgere una certa attività.

Modulo Parte di un sottosistema dedicata a svolgere particolari funzioni legate alle attività del sottosistema.

- Presentazione: UI di login, UI del menu, $\cdots$;
- Elaborazione: gestore accessi, gestore ricerche, $\cdots$;
- Gestione dei dati: DB utenti, DB pagamenti, $\cdots$.

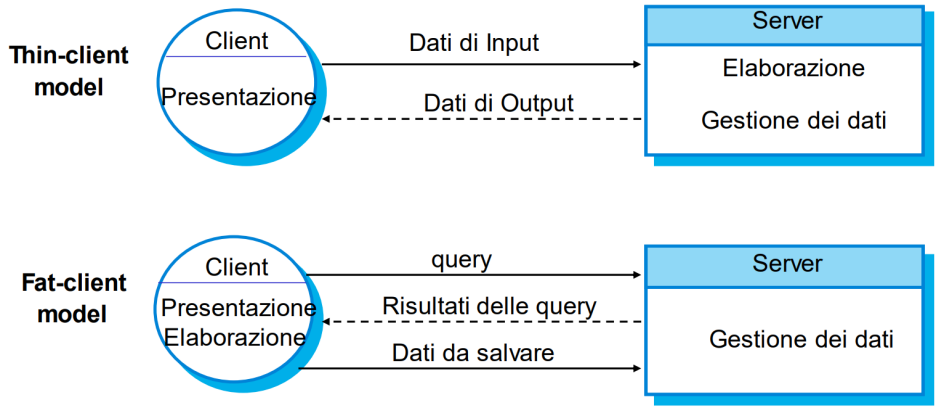
3.2.2 Deployment

Con *deployment* si intende la distribuzione dei componenti in vari dispositivi hardware.

Nel **deployment a 1-tier** i tre stati del sistema sono concentrati su un dispositivo, in quello a **2-tiers** su due e in quello a **3-tiers** su tre.

Deplyment 2-tiers

- **Thin client:** il server si occupa dell'elaborazione e della gestione dei dati, mentre il client si occupa della presentazione.
  - carico spostato sulla macchina server;
  - macchina client sprecata dal punto di vista computazionale.
- **Fat client:** il server si occupa della gestione dei dati, mentre il client si occupa della presentazione e dell'elaborazione.
  - minore carico sulla macchina server;
  - il software di elaborazione deve essere aggiornato su tutte le macchine client.



3.2.3 Metodo di controllo

Un componente fornisce servizi ad altri componenti. Un'**interfaccia** è un insieme di operazioni che il componente mette a disposizione di altri componenti ed è condivisa con i componenti che lo invocano. Un **corpo** è la parte interna del componente e non è conosciuto agli altri componenti. La separazione tra interfaccia (pubblica) e corpo (privato) è detta **information hiding**.

Esistono diversi stili di controllo (attivazione) tra componenti:

- controllo **centralizzato**: è presente un componente detto **controllore**, che controlla l'attivazione e il coordinamento degli altri componenti detti passivi, che eseguono i comandi e restituiscono l'output al controllore;

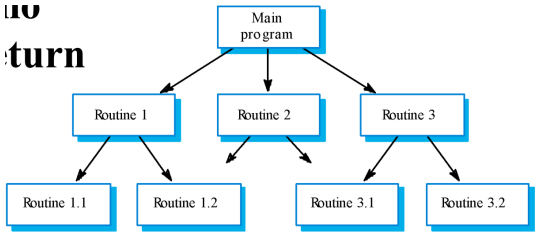
  Il controllore periodicamente esegue un **control-loop**:

  - controlla se dei componenti hanno prodotto dati da elaborare;
  - attiva o disattiva dei componenti a seconda dei dati raccolti;
  - passa i dati ai componenti per l'elaborazione;
  - raccoglie i risultati.

- controllo **basato su eventi**: è basato su eventi esterni (ad esempio un segnale) ed ogni componente si occupa di gestire determinati eventi; il gestore degli eventi è detto **broker**, che rileva l'evento e lo notifica tramite broadcast:

  - Broadcast **selettivo:** invia una notifica ai componenti interessati.
    Il gestore degli eventi mantiene un registro con gli eventi e i corrispondenti componenti di interesse. I componenti possono comunicare al gestore quali sono gli eventi di loro interesse.
  - Broadcast **non selettivo:** invia una notifica a tutti i componenti.

- controllo **call-return**: il controllo passa dall'alto verso il basso (**Top-down**); Ogni operazione ritorna un risultato al livello superiore, al componente che lo aveva invocato.



- controllo **client-server**: un componente client (controllore) chiede un servizio ad un componente server (passivo) attraverso una chiamata di procedura e il componente server risponde.

3.2.4 Modellazione del comportamento ad oggetti

I componenti del sistema sono considerati come oggetti che interagiscono. Un *oggetto* è definito da:

- **attributi:** definiscono lo stato dell'oggetto;
- **operazioni:** operazioni che l'oggetto può compiere.

Gli oggetti comunicano tra di loro attraverso lo scambio di messaggi.

4 Collaudo

La fase di collaudo avviene alla fine dell'implementazione (scrittura del codice) del sistema. Si ricercano e correggono difetti, si controlla che il prodotto realizzi ogni servizio senza malfunzionamenti o bug (**fase di verifica**) e che soddisfi i requisiti del committente (**fase di validazione)**.

4.1 Verifica

Si va alla ricerca di eventuali bug.

$$\text{Error} \rightarrow \text{Fault} \rightarrow \text{Failure}$$

- **Error** = causa di un baco, errore umano (errore di scrittura, errore logico);
- **Fault** (baco) = difetto del programma dovuto a un errore (eg. simbolo sbagliato all'interno di un operazione algebrica);
- **Failure** (malfunzionamento) = manifestazione visibile della presenza di un baco (eg. divisione per zero causa il crash) - comportamento imprevisto.

Are we building the system right?

4.2 Validazione

Are we building the right system?

4.3 Ispezione

L'ispezione è una **tecnica statica** di analisi del codice, basata sulla lettura di quest'ultimo e della documentazione. L'ispezione è meno costosa del testing e può essere eseguita si una versione incompleta del sistema, ma alcuni requisiti non funzionali non si possono collaudare solo con l'ispezione (efficienza, affidabilità, usabilità, $\cdots$).

Un team analizza il codice e segnala i possibili difetti. Viene seguita una checklist che indica i possibili difetti da investigare:

Classe di errore	Controllo di ispezione
Errori nei dati	Tutte le variabili del programma sono inizializzate prima che il loro valore sia utilizzato? Tutte le costanti hanno avuto un nome? Il limite superiore degli array è uguale alla loro dimensione o alla loro dimensione -1? Se si utilizzano stringhe di caratteri, viene assegnato esplicitamente un delimitatore? Ci sono possibilità di buffer overflow?
Errori di controllo	Per ogni istruzione condizionale la condizione è corretta? E' certo che ogni ciclo sarà ultimato? Le istruzioni composte sono correttamente messe fra parentesi? Se è necessario un break dopo ogni caso nelle istruzioni case, è stato inserito?
Errori di Input/Output	Sono utilizzate tutte le variabili di input? A tutte le variabili di output viene assegnato un valore prima che siano restituite? Input imprevisti possono causare corruzione?
Errori di interfaccia	Tutte le chiamate a funzione e a metodo hanno il giusto numero di parametri? Il tipo di parametri formali e reali corrisponde? I parametri sono nel giusto ordine? Se i componenti accedono alla memoria condivisa, hanno lo stesso modello di struttura per questa?
Errori nella gestione della memoria	Se una struttura collegata viene modificata, tutti i collegamenti sono correttamente riassegnati? Se si utilizza la memoria dinamica, lo spazio è assegnato correttamente? Lo spazio viene esplicitamente deallocato quando non è più richiesto?
Errori di gestione delle eccezioni	Sono state prese in considerazione tutte le possibili condizioni d'errore?

4.3.1 Ruoli nel team di ispezione

Il team di ispezione è composto da:

- **Autori** = programmatori del codice; correggono i difetti rilevati durante l'ispezione;
- **Ispettori** = trovano i difetti;
- **Moderatore** = gestisce il processo di ispezione.

4.3.2 Processo di ispezione

Fase di pianificazione il moderatore seleziona gli ispettori e controlla che il materiale (codice, documentazione) sia completo.

Fase di introduzione il moderatore organizza una riunione preliminare con autori e ispettori nella quale è discusso lo scopo del codice e la checklist da seguire.

Fase di preparazione individuale gli ispettori studiano il materiale e cercano difetti nel codice in base alla checklist ed all'esperienza personale.

Fase di riunione di ispezione gli ispettori indicano i difetti individuati.

Fase di rielaborazione il programma è modificato dall'autore per correggere i difetti.

Fase di prosecuzione il moderatore decide se è necessario un ulteriore processo di ispezione.

4.3.3 Analisi statica del codice

Gli strumenti *CASE* (eg. gcc -wall) supportano l'ispezione del codice eseguendo su di esso:

- **Analisi del flusso di controllo** per ricercare:
  - cicli con uscite multiple;
  - salti incondizionati all'interno di un ciclo;
  - codice non raggiungibile:
    - delimitato da istruzioni di salto incondizionato;
    - vincolato ad una condizione sempre falsa.
- **Analisi dell'uso dei dati** per ricercare:
  - variabili non inizializzate prima di essere usate;
  - variabili scritte, ma non lette;
  - variabili dichiarate ma non usate;
  - violazione dei limiti di un array;
  - condizioni che danno sempre lo stesso valore booleano.
- **Analisi delle interfacce**:
  - consistenza delle dichiarazioni di metodi, funzioni, procedure;
    - errore nel tipo di parametri, nel numero di parametri;
  - metodi, funzioni, procedure dichiarate, ma mai invocate;
  - risultati di funzioni non usati.
- **Analisi della gestione della memoria**:
  - puntatori non assegnati;
  - errori nell'aritmetica dei puntatori.

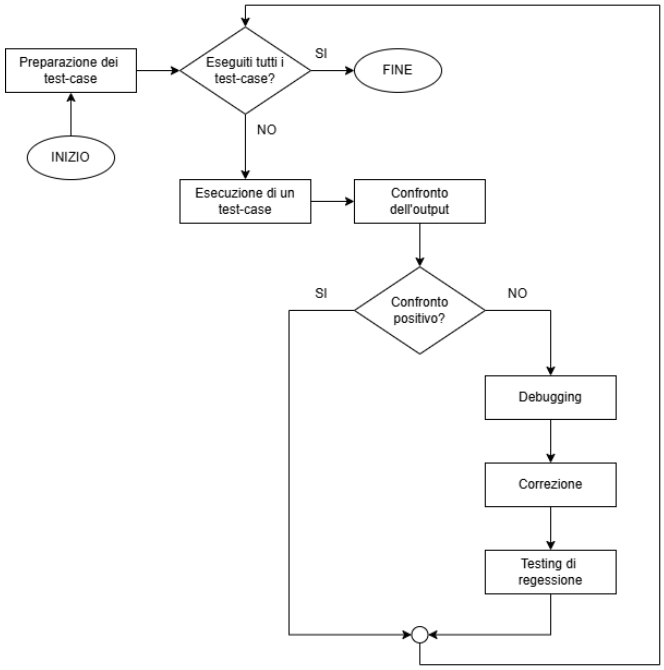
L'analisi statica precede l'ispezione del codice fornendo informazioni utili all'individuazione dei difetti.
L'analisi statica non è sufficiente, ad esempio:

- individua inizializzazioni mancanti, ma non inizializzazioni errate;
- individua il passaggio di parametri di tipo errato, ma non il passaggio del parametro errato.

4.4 Testing

Il testing è una **tecnica dinamica** dove il sistema viene messo in esecuzione e si osserva come questo si comporta quando elabora dei test-case. Se il testing non rileva difetti, questo non significa che il sistema non ha difetti, ma che il testing non ha riguardato parti del sistema soggette a difetti.

4.4.1 Processo di testing



Preparazione dei test-case prodotte le "black box" o "white box" e si forniscono i dati di input del test e i dati di output attesi.

Esecuzione di un test-case fornendo al sistema i dati di input del test-case.

Confronto dei risultati di output ritornati dal sistema con i dati di output attesi. Se i dati di output attesi e quelli ritornati non corrispondono, allora è presente un difetto e si eseguono le fasi successive.

Debugging si controlla l'esecuzione del programma, istruzione per istruzione. Dopo l'esecuzione di ogni istruzione si controllano i valori delle variabili. Alla fine si scopre la posizione dell'errore.

Correzione dell'errore, si modifica il codice per eliminare il difetto.

Testing di regressione

- Si ripete l'ultimo test-case, per verificare che le modifiche al codice abbiano effettivamente corretto l'errore;
- Si ripetono i test-case precedenti, per verificare che le modifiche non abbiano introdotto altri difetti (effetto collaterale).

4.4.2 Test-case

Un test-case è composto da dati in input e dati in output attesi. I dati di input possibili potrebbero essere infiniti e di conseguenza i test-case, però il sistema può essere testato un numero finito di volte. Serve un criterio per scegliere un numero finito di test-case che permetta un testing efficace del sistema:

Black box testing il sistema è visto come una scatola **non** trasparente, la scelta dei test-case è basata sulla conoscenza di quali sono i dati di input e i dati di output.

- Si considerano solo gli input e gli output della funzione da testare;
- non si considera il modo in cui i dati di input vengono trasformati in dati di output.

Dato l'insieme dei dati di input e l'insieme dei dati di output, una **partizione di equivalenza** è un sottoinsieme dei dati di input per cui:

- il sistema produce sempre lo stesso dato di output;
- oppure il sistema produce dati di output che appartengono sempre ad un determinato sottoinsieme.

Normalmente, il sistema si comporta nello stesso modo per i dati di input che appartengono alla stessa partizione.

Si può fare black box testing usando delle partizioni di equivalenza: si suppone che se un test è negativo/positivo per alcuni elementi della partizione, allora il test è negativo/positivo per tutti gli elementi della partizione.

Per individuare le partizioni:

- si identificano i possibili dati di output;
- se numerosi, si distribuiscono in sottoinsiemi;
- per ogni dato di output o sottoinsieme di dati di output si individuano una o più partizioni di equivalenza tra i dati di input;
- da ogni partizione si scelgono un numero finito di test-case:
  - **BVA (Boundary Value Analisys)**: si scelgono test-case con dati di I/O al **confine** delle partizioni di equivalenza.
- maggiore è il partizionamento dei dati, maggiore sarà il numero di test-case, e maggiore sarà l'efficacia del testi.

White box (o glass-box) testing il sistema è visto come una scatola trasparente, la scelta dei test-case è basata sulla struttura del codice.

- É visibile il modo in cui i dati vengono elaborati.

Il collaudatore può vedere il codice e i test-case sono ricavati dall'analisi del codice.

L'obiettivo è testare ogni parte del codice:

- ogni istruzione nel programma deve essere eseguita almeno una volta durante il testing;
- ogni condizione nel programma deve essere eseguita sia nel caso sia vera, sia nel caso sia falsa.

Il codice viene rappresentato con un **flow graph** (grafo di flusso) che rappresenta i possibili cammini nel codice. Un nodo può rappresentare un'istruzione, un gruppo di istruzioni in sequenza o una condizione, gli archi rappresentano il flusso del controllo. Il flow graph può essere ricavato in modo manuale o automatico (strumento CASE).

Nel codice, un **cammino** si dice **indipendente** se introduce almeno una nuova sequenza di istruzioni o una nuova condizione. Nel flow graph corrispondente, un cammino è indipendente se attraversa almeno un arco non ancora percorso nei cammini precedenti.

4.5 Complessità ciclomatica

Il numero di cammini indipendenti equivale alla **CC (Complessità Ciclomatica)** del flow graph. CC è il numero di test-case richiesti per eseguire almeno una volta ogni parte del codice. I **dynamic program analyser** (EclEmma) sono strumenti CASE che, dato un test-case, indicano quali parti del codice sono state interessate da un test-case e quali sono ancora da testare.

4.6 Tipi di componenti da integrare nel sistema

- Componenti sviluppati ad-hoc per il sistema, sono tipicamente implementati in parallelo;
- Componenti commerciali (COTS - Commercial Off-The-Shelf) già disponibili, il costo è inferiore rispetto ad un componente ad-hoc;
- Componenti realizzati per sistemi precedenti, costo quasi nullo.

4.6.1 Approcci di integrazione dei componenti

- **Integrazione** = composizione dei componenti in un unico sistema;
- **Approccio incrementale** = i componenti sono integrati uno alla volta, la consegna dei componenti deve essere coordinata, ma non contemporanea;
- **Approccio "big bang"** = i componenti sono tutti integrati contemporaneamente, i componenti devono essere completati entro lo stesso termine temporale.

4.6.2 Testing con integrazione incrementale

Il testing riguarda ciascun componente e la sua integrazione nel sistema:

- **Testing dei componenti** = testing di ogni componente (ad-hoc) considerato in modo isolato;
- **Testing di integrazione** = si testa il sistema costituito dai componenti testati ed integrati finora, viene ripetuto ogni volta che un componente viene integrato;
- **Release testing** = testing del sistema costituito da tutti i componenti:
  - Prodotti customizzati: test di accettazione;
  - Prodotti generici: alpha testing, beta testing.

Testing dei componenti Riguarda i singoli componenti. Il test di un componente:

- può essere preceduto dall'**ispezione** del codice, il codice di un componente non è di solito troppo lungo;
- può essere di tipo **white box**, il codice di un componente non è di solito troppo lungo;
- ha lo scopo di **verifica** (ricerca dei bachi), di solito non si può fare la validazione sulla base di un singolo componente;
- viene svolto di solito dal programmatore del componente.

Testing di integrazione Viene svolto quando si integrano due o più componenti. Lo scopo del test di integrazione è verificare l'interfacciamento corretto dei componenti:

- le funzioni di un componente devono essere invocate nel modo corretto;
- le funzioni di un componente devono ritornare i risultati attesi.

Si applicano i test-case applicati nei test di integrazione precedenti (test di regressione).

- L'aggiunta di nuovi componenti può creare problemi nel funzionamento di quelli integrati e testati precedentemente.

In fine si applicano nuovi test-case.

Quando si presenta un difetto, è più probabile che esso riguardi l'ultimo componente integrato.

Il test di integrazione riguarda la verifica, può riguardare la validazione se sono già stati integrati i componenti necessari per un certo requisito.

Tipo di testi:

- White box se pochi componenti sono stati integrati finora;
- Black box se molti componenti sono stati integrati finora.

Top-down integration testing Si costruiscono prima i moduli primari, poi quelli secondari. I moduli secondari non ancora disponibili sono sostituiti da **STUB** (moduli ausiliari "dummy" che forniscono servizi preimpostati con dati costruiti appositamente per verificare i test-cases.

Bottom-up integration testing Si costruiscono prima i moduli secondari, poi quelli primari. I moduli primari non ancora disponibili sono sostituiti da **DRIVER** (test program che invoca il modulo secondo i dati selezionati per i test-case).

Release testing Riguarda il sistema completo. Riguarda la **validazione** ed è di tipo **black-box**.

Sistemi customizzati: test di accettazione:

- il committente controlla il soddisfacimento dei requisiti;
- si effettuano eventuali correzioni;
- queste fasi si ripetono finché il committente non è soddisfatto; quindi il sistema viene effettivamente consegnato.

Sistemi generici:

1. **Alpha testing**:
   - La versione "Alpha" del sistema viene resa disponibile ad un gruppo di sviluppatori (tester) o utenti esperti, per il collaudo;
   - La versione "Alpha" viene collaudata ed eventualmente corretta;
   - Si ripete l'Alpha testing oppure si genera la versione "Beta".
2. **Beta testing**:
   - La versione "Beta" del sistema viene resa disponibile (via web) ad un gruppo di potenziali clienti per il collaudo;
   - La versione "Beta" viene collaudata dai clienti ed eventualmente corretta;
   - Si ripete il Beta testing oppure il prodotto viene messo sul mercato.

Testing con integrazione "big-bang" I componenti sono integrati tutti insieme in una sola volta. Si effettuano:

- Test dei componenti (prima dell'integrazione);
- Release testing (dopo l'integrazione).

Non si effettuano testing di integrazione. Questo tipo di integrazione rende molto difficile localizzare un eventuale difetto dopo l'integrazione.

Altri tipi di testing

- **Recovery testing**, valuta requisiti come:
  - Fault tolerance;
  - Recoverability;
  - Mean Time to Repair.
- **Security testing** = simulazione di attacchi dall'esterno;
- **Deployment testing** = verifica dell'installazione sui dispositivi.
- $\cdots$

4.7 Test di usabilità

Vari tipi di utente provano ad utilizzare il sistema per la prima volta (utenti non esperti, utenti che hanno già usato sistemi simili, utenti con competenze tecniche, $\cdots$). Si valuta la facilità con cui riescono ad usare il sistema, si raccolgono commenti, critiche e suggerimenti dagli utenti.

4.8 Stress testing

Serve per verificare:

- **efficienza** = (tempo di risposta, memoria occupata, numero di job processati, ecc.), il sistema deve elaborare il carico di lavoro previsto;
- **affidabilità** = il sistema non deve fallire (andare in crash).

Viene svolto quando il sistema è completamente integrato (efficienza e affidabilità sono proprietà complessive ed emergenti), riguarda software e hardware.

Si eseguono test in cui il carico di lavoro è molto superiore a quello previsto normalmente

- se il sistema "resiste" a carichi di lavoro superiori al previsto, dovrebbe resistere a carichi di lavoro normali;
- se il sistema non "resiste" è possibile individuare difetti non emersi dagli altri test, corruzione dei dati, perdita dei servizi, $\cdots$

4.9 Back-to-back testing

Si usa quando varie versioni del sistema sono disponibili (prototipi, versioni del sistema per sistemi operativi diversi, versioni del sistema per tipi di hardware diversi, nuove versioni con funzioni in comune con le precedenti).

Si effettua lo stesso test su tutte le versioni e si confrontano gli output delle varie versioni

- se l'output non è sempre lo stesso, c'è la presenza di difetti in qualche versione;
- se l'output è sempre lo stesso, è probabile che sia corretto (oppure tutte le versioni potrebbero avere lo stesso difetto).

5 Manutenzione

La manutenzione riguarda tutte le modifiche fatte al sistema **dopo la consegna**.

La manutenzione è un **processo ciclico**, dura dalla consegna alla dismissione del sistema, permette al sistema di "sopravvivere", un sistema va aggiornato nel tempo oppure perde progressivamente di qualità, utilità e valore economico, tutto ciò permette al sistema di "evolversi".

5.1 Tipi di manutenzione

5.1.1 Manutenzione correttiva

Serve a correggere i difetti non emersi in fase di collaudo:

- Difetti di **implementazione** = sono i meno costosi da correggere;
- Difetti di **progettazione** = sono più costosi perché richiedono di modificare vari componenti del sistema;
- Difetti di **specifica** = sono i più costosi da correggere, potrebbe essere necessario riprogettare il sistema.

5.1.2 Manutenzione adattiva

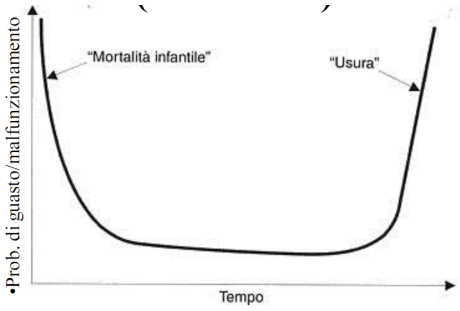
Adattamento del sistema a cambiamenti di piattaforma (nuovo hardware o nuovo sistema operativo).

5.1.3 Manutenzione migliorativa

Aggiunta, cambiamento o miglioramento di requisiti funzionali e non funzionali, secondo le richieste del committente/cliente/utente o in base alle tendenze di mercato.

5.2 Curva dei guasti

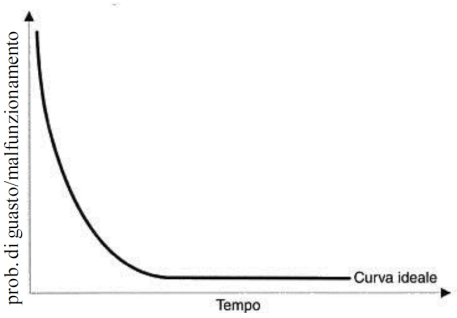
5.2.1 Guasti dell'hardware



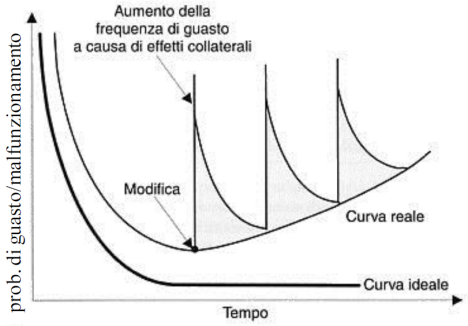
"Mortalità infantile" = periodo in cui si possono rappresentare difetti non scoperti nel collaudo.
Causa dell'**usura** = deterioramento dei materiali, polvere, vibrazioni, fattori ambientali, temperatura, *cdots*

5.2.2 "Guasti" del software

Curva ideale nel caso si richieda solo manutenzione correttiva. Non c'è usura per il software.



Curva reale causata da modifiche di manutenzione migliorativa. Ogni modifica può introdurre difetti (effetti collaterali) e quindi successive richieste di manutenzione correttiva.



5.3 Fattori che influenzano il costo di manutenzione

Dipendenza dei componenti la modifica di un componente potrebbe avere ripercussioni sugli altri componenti.

Linguaggio di programmazione i programmi scritti con linguaggi ad alto livello sono più facili da capire e quindi da mantenere.

Struttura del codice il codice ben strutturato e documentato rende più facile la manutenzione.

Collaudo una fase di collaudo approfondita riduce il numero di difetti scoperti successivamente alla consegna.

Qualità della documentazione una documentazione chiara e completa facilita la comprensione del sistema da mantenere.

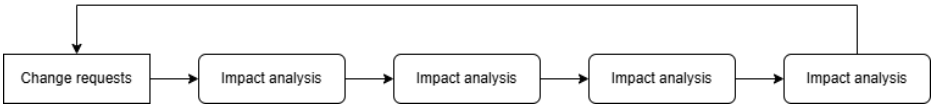
Stabilità dello staff i costi si riducono se lo staff che ha sviluppato il prodotto è lo stesso che lo mantiene.

Età del sistema il costo di manutenzione tende a crescere con l'età del sistema.

Stabilità del dominio dell'applicazione se il dominio subisce variazioni, il sistema deve essere aggiornato (eg. programmi gestionali).

Stabilità della piattaforma il cambiamento della piattaforma (hardware o sistema operativo) può richiedere la manutenzione del software; è raro che la piattaforma non cambi durante la vita del prodotto software.

5.4 Processo di manutenzione



Identificazione dell'intervento arrivo di una richiesta di modifica (nuovi requisiti, correzione di difetti, adattamenti, $\cdots$).

Analisi dell'impatto

- Individuare requisiti, componenti, operazioni e test-case influenzati dalla modifica;
- Individuare le possibili soluzioni e e valutare tempo, costi e benefici di ognuna.

Pianificazione della release Selezionare la soluzione migliore, accettare o rifiutare la modifica.

Realizzazione della modifica se la modifica è accettata:

- Se necessario, aggiornamento della **specifica** dei requisiti;
- Se necessario, aggiornamento della **progettazione**;
- Sicuramente, aggiornamento dell'**implementazione** (coding);
- **Collaudo**: test-case relativi alla modifica effettuata + test di regressione (ripetizione dei test-case originali)

Rilascio della nuova versione del codice.

5.5 CRF (Change Request Form)

Il CRF è un documento formale che descrive una modifica. Viene compilato da:

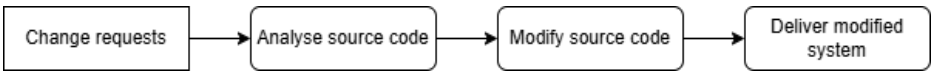
- **Proponente della modifica** (committente, utente finale, sviluppatori) che avanzano una proposta di modica e priorità;
- **Sviluppatori**
  - requisiti, componenti, test-case influenzati dalla modifica;
  - valutazione della modifica, benefici, costi, tempo;
  - modalità di implementazione.
- **CCB (Change Control Board)** committente, manager, alcuni sviluppatori che approvano o rifiutano la proposta della modifica.

5.6 Manutenzione di emergenza (patch)

Se si presentano problemi che devono essere risolti in fretta:

- errore grave di sistema che non permette la continuazione delle normali operazioni;
- malfunzionamento del sistema che causa danno economici;
- cambiamenti imprevisti nell'azienda (eg. nuove norme giuridiche)

L'urgenza richiede una manutenzione di emergenza, non si passa attraverso tutte le fasi del processo di manutenzione ma si modifica direttamente il codice, applicando la soluzione più immediata, e si rilascia una versione aggiornata (**path**).



Più tardi si fa una manutenzione ordinaria per la stessa richiesta di modifica e si rilascia un'ulteriore versione.

5.7 Versioni e Release

5.7.1 Versione

Istanza del sistema che differisce per qualche aspetto dalle altre istanza.

- Variazione dei requisiti;
- Correzione di difetti;
- Piattaforma;
- $\cdots$

5.7.2 Release

Una particolare versione che viene distributiva a committente/clienti. Ci sono più versioni che release (versioni intermedie possono essere create durante lo sviluppo).

Una release compre:

- **Programma eseguibile**;
- **File di configurazione** = indicano come la release deve essere configurata per particolari installazioni;
- **File di dati** = necessari per il funzionamento del sistema;
- **Programma di installazione** = per installare il sistema su una certa piattaforma;
- **Documentazione** cartacea, elettronica, on line.

5.7.3 Identificare numericamente le versioni

$$A.B.C$$

- A = major version
- B = minor version
- C = patches

5.8 Identificare le versioni tramite gli attributi

Gli attributi di una versione possono essere:

- Committente;
- Linguaggio di programmazione;
- Stato dello sviluppo;
- Piattaforma (hardware, sistema operativo);
- Data di creazione.

Un numero di versione identifica una versione, ma non fornisce informazioni su di essa, quindi si può usare una combinazione di alcuni suoi attributi (eg. 2000/Linux/Java1.2), gli attributi devono essere scelti in modo che ogni versione possa essere identificata in modo univoco (**Unicità**).

5.9 Configurazione software

Insieme di informazioni prodotte da un processo software.

- Documentazione (requisiti, modelli di progettazione, test-case, $\cdots$);
- Codice;
- Dati.

5.9.1 Database delle configurazioni

Contiene le varie configurazioni software corrispondenti alle release di un sistema. Fornisce informazioni utili quando si deve fare la manutenzione, può essere integrato con il VMS (Git, SVN) e può essere integrato con altri strumenti CASE (Editor dei requisiti, UML editor, IDE, Tool per testing automatico).

5.10 Sistemi ereditati (Legacy system)

Un sistema ereditato è un vecchio sistema che deve essere mantenuto nel tempo che presenta tecnologia obsoleta e sono costosi da mantenere, ma la loro dismissione e sostituzione può essere rischiosa (sistemi critici, contengono informazioni che devono essere conservate, servizi molto perfezionati durante la vita del sistema).

Problemi con i sistemi ereditati

- Poco strutturati;
- Mancanza di documentazione;
- Difficile reperire gli sviluppatori originali;
- Sviluppati per vecchie piattaforme (eg. main-frame).

5.10.1 Strategie per i sistemi ereditati

Manutenzione ordinaria del sistema.

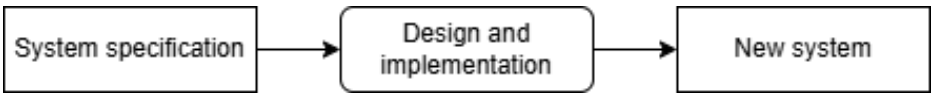
Software re-engineering ristrutturazione dei sistemi ereditati per renderli più mantenibili.

Reimplementazione del sistema mantenendo gli stessi requisiti funzionali.

Dismissione del sistema, cambiare la propria organizzazione in modo che l'uso del sistema non sia più necessario. La scelta della strategia dipende dalla qualità del codice del sistema e dalla sua utilità (business value). Il costo di manutenzione è inversamente proporzionale alla qualità del codice.

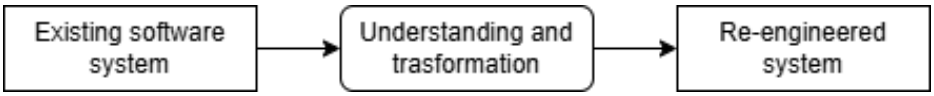
5.11 Forward engineering vs re-engineering

5.11.1 Forward engineering (ingegneria diretta)



É il processo software (specifica, progettazione, $\cdots$), si crea un nuovo prodotto partendo dalla specifica dei requisiti.

5.11.2 Re-engineering



Il nuovo sistema è dato da qualche trasformazione del vecchio, allo scopo di rinnovarlo e aumentare la mantenibilità. I requisiti funzionali non cambiano e non si altera il comportamento del sistema.

Le attività di re-engineering sono: traduzione del codice, ristrutturazione del codice (refactoring), ristrutturazione dei dati, $\cdots$.

Traduzione del codice sorgente Traduzione del codice dal linguaggio di programmazione originale a:

- una versione più recente dello stesso linguaggio;
- un linguaggio differente.

Le ragioni della traduzione sono:

- aggiornamento dell'hardware, i compilatori del linguaggio originale potrebbero non essere disponibili per il nuovo hardware;
- mancanza di conoscenza del linguaggio. lo staff potrebbe non conoscere il linguaggio originale perché non più in uso (eg. Cobol);
- politiche organizzative, decisione di avere un certo linguaggio come standard.

Ristrutturazione del codice (refactoring) Le modifiche fatte al sistema nel corso del tempo, tendono a rendere il codice sempre meno ordinato. Il codice diventa sempre più difficile da leggere e comprendere.

La ristrutturazione cerca di rendere il codice più semplice, leggibile, comprensibile e modulare:

- raggruppamento delle parti correlate del programma, il codice relativo allo stesso componente o alla stessa funzionalità, non deve essere sparso ma coeso (eg. nello stesso file o nella stessa directory);

- eliminazione delle ridondanze;
- aggiunta di commenti nel codice.
- eliminare codice duplicato (tramite funzioni/metodi);
- ridurre la lunghezza di una funzione/metodo distribuendo il codice in varie funzioni/metodi;
- dare nomi significativi a variabili, funzioni, classi, metodi, attributi, per far capire il loro scopo;
- fondere classi simili in un'unica classe;
- uniformare l'indentazione e la posizione dei commenti.

Codice semplice e uniforme → più leggibile. La ristrutturazione non modifica il comportamento del codice, ma ne migliora la struttura interna.

La ristrutturazione avviene ispezionando ed editando il codice.

La ristrutturazione è un processo costoso. Deve essere limitata alle parti del codice che effettivamente lo richiedono:

- parti in cui si verificano fallimenti, devono essere comprensibili per poter essere corrette;
- parti che hanno subito molte modifiche, la strutturazione del codice ha subito un degradamento;
- parti in cui il codice è particolarmente complesso, difficile da leggere.

Ristrutturazione dei dati Processo di riorganizzazione delle strutture dati, serve per:

- cambiamento delle strutture dati;
- adattare un sistema ad usare un DBMS anziché usare file di dati di formato proprio;
- concentrare i dati sparsi in vari file in un unico DB;
- migrazione dei dati da un tipo di DBMS ad un altro;
- migrazione dei dati in un DBMS più moderno.

5.12 Reverse engineering

Per sistemi ereditati, la documentazione può essere scarsa o assente, complicando le attività di re-engineering. Il re-engineering può essere supportato dal reverse engineering.

Il reverse engineering serve per generare documentazione partendo da codice e dati.

- Input:
  - codice sorgente;
  - dati.
- Output:
  - diagrammi di progettazione (UML);
  - diagrammi dei dati (E-R);
  - ...

5.13 Reimplementazione del sistema

Quando il vecchio sistema è troppo mal strutturato per il re-engineering.

La reimplementazione del sistema è inevitabile quando c'è un cambiamento radicale del linguaggio di programmazione (eg. funzionale → object-oriented), modifiche architetturali sostanziali (eg. centralizzato → distribuito).

5.13.1 Creazione di un nuovo sistema partendo dal vecchio sistema

Il vecchio sistema fornisce la specifica dei requisiti funzionali, tutto il resto (req. non funzionali, progettazione, implementazione e collaudo) viene rifatto usando tecnologia aggiornata. I maggiori rischi sono l'introduzione di errori di specifica, progettazione e implementazione. Questo ha un costo maggiore rispetto al re-engineering. Il sistema diventa più mantenibile.

6 Gestione del progetto

6.1 Personaggi chiave

- **Committente**
  - Suggerisce i requisiti del prodotto;
  - Valuta il costo del prodotto;
  - Verifica che il prodotto soddisfi i requisiti.
- **Manager**
  - Negozia un contratto (tempo e costi) con il committente;
  - É il collegamento tra committente e sviluppatori;
  - Si occupa delle attività di gestione.
- **Sviluppatore**
  - Colui che effettivamente sviluppa il sistema (Analista, progettista, programmatore, collaudatore, manutentore).

6.2 Attività di gestione

- **Scrittura proposte** = per ottenere un contratto o vincere una gara d'appalto (obiettivi, metodi, stima preliminare di tempi e costi, $\cdots$);
- **Pianificazione** = task, milestone, risorse, tempi;
- **Stima dei costi**;
- **Gestione dei rischi** = previsione di problemi e soluzioni;
- **Monitoraggio del processo** = si valutano i progressi del processo, e si confrontano con la pianificazione;
- **Scrittura e presentazione di rapporti** = si informa il committente sull'avanzamento del progetto;
- **Revisione** del progetto = aggiornamento della pianificazione, della stima dei costi e della gestione dei rischi.

6.3 Processo sw e Gestione

Il processo software e la gestione si svolgono in parallelo.

Progetto = processo software + attività di gestione

Gli scopi sono:

- **Processo sw** = creare un sistema che soddisfi i requisiti;
- **Gestione** = il sistema deve essere consegnato nei tempi previsti e deve avere un costo finale non superiore a quello previsto.

Un progetto **fallisce** quando:

- Il prodotto è consegnato in ritardo (gestione);
- Costa più di quanto stimato (gestione);
- Non soddisfa i requisiti (processo software).

La buona gestione non garantisce il successo di un progetto, ma la cattiva gestione di solito lo fa fallire.

6.3.1 Difficoltà di gestione

- **Risorse limitate** = umane, tecnologiche, finanziarie;
  - Risorsa = ciò che viene utilizzato durante un'attività.
- **Tempo limitato** = stabilito con il committente nel contratto;
- **Costo limitato** = stabilito con il committente nel contratto;
- **Irreversibilità** = tempo e risorse già impiegate non sono recuperabili;
- **Incertezza** = le conseguenze delle decisioni non sono certe; si possono verificare problemi (previsti e non previsti);
- **Complessità** = bisogna coordinare molte attività distribuite tra vari gruppi di persone;
- **Il software è intangibile** = non si può vedere materialmente il progresso di un progetto; è difficile stabilire lo stato di avanzamento;
  - Per questo è necessaria la produzione di documentazione.
- **I progetti sono "unici"** = ognuno è diverso dall'altro;
  - Non sempre si può fare riferimento all'esperienza fatta in progetti precedenti.

6.4 Le attività di Pianificazione

Il processo software è scomposto in **task**. Tra task vi sono delle dipendenze (eg. un task può iniziare solo quando un insieme di altri task è stato ultimato, task paralleli o sequenziali a seconda delle dipendenze);
Ad ogni task è necessario assegnare:

- delle **milestone** e **deliverable**;
- **delle risorse**;

- persone (con eventuale ricerca di nuovo personale);
- hardware o software (con eventuale acquisto di macchine o licenze);
- budget (denaro spendibile per quel task);

- **tempistiche** (scheduling)
  - durata di un task = tempo di svolgimento (+ ricerca di risorse);
  - durata del progetto = tempo complessivo.

6.4.1 Milestone e deliverable

Per valutare l'avanzamento del processo, si stabiliscono dei momenti precisi in cui determinati risultati intermedi devono essere raggiunti.

Milestone terminazione di un certo insieme di task previsto per una certa data,

- punto di controllo per l'avanzamento del lavoro;
- un **rapporto** sullo stato di avanzamento del progetto viene preparato in corrispondenza di una milestone, questo fornisce le informazioni per verificare il rispetto della pianificazione.

Frequenza delle milestone: se troppo frequenti, causano uno spreco di tempo nella preparazione di rapporti; se poco frequenti, non permettono di valutare l'avanzamento del processo.

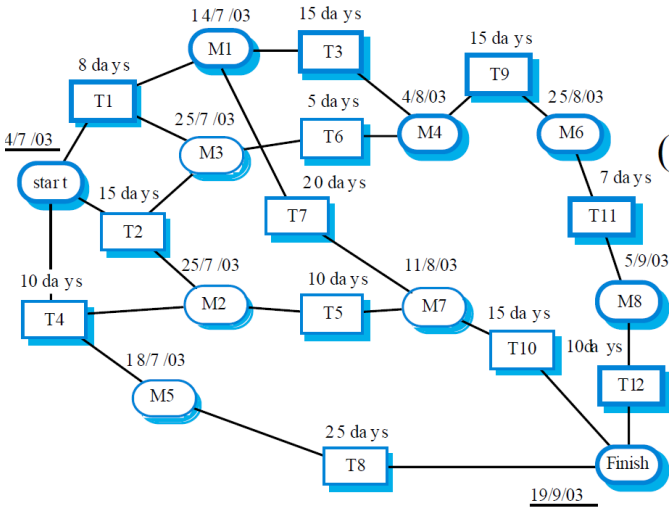
Deliverable particolare milestone i cui risultati devono essere notificati al committente.

6.4.2 Tempistica (scheduling)

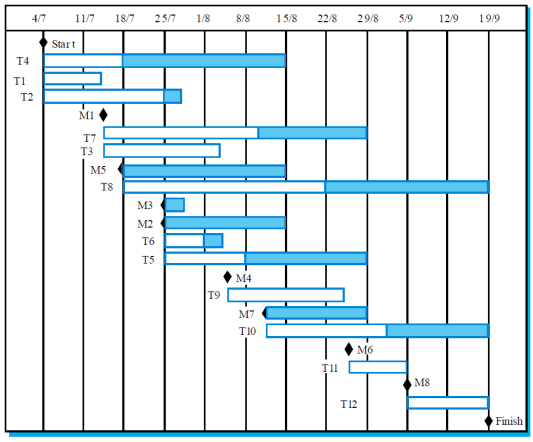
Una **dipendenza tra task** indica che un task non può iniziare prima che un insieme di task sia stato ultimato.

Cammino critico Cammino più lungo in termini di tempo, dal nodo "inizio" al nodo "fine". Indica il tempo per finire l'intero progetto.

Ritardi di task nel cammino critico determinano ritardo nel completamento del progetto. Ritardi di task fuori dal cammino critico non determinano un ritardo finale, a meno che tali ritardi non determinino un nuovo cammino critico.



Bar chart indica la durata prevista di un task, e il suo ritardo consentito, senza modificare il cammino critico.



Si può utilizzare anche un bar chart per visualizzare l'**allocazione delle persone nel tempo**.

6.5 Rischi

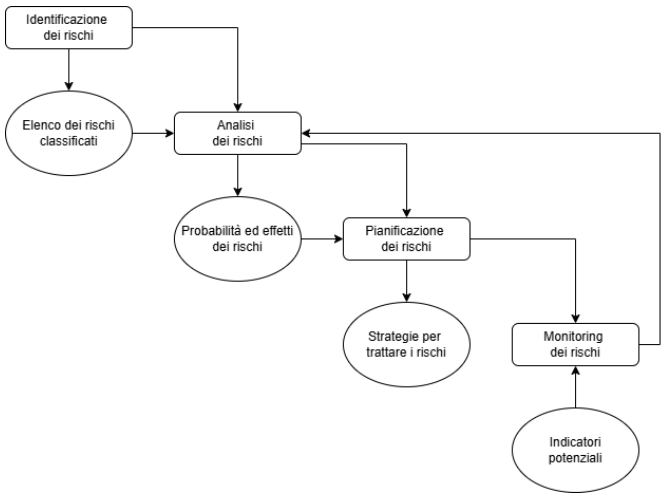
Un rischio è una circostanza avversa che si preferirebbe che non accadesse, questo ha due caratteristiche:

- **Incertezza** = il rischio ha una probabilità di verificarsi;
- **Perdita** = se il rischio si verifica, comporta delle conseguenze indesiderate. I rischi influenzano negativamente la pianificazione e la qualità del sistema.

I rischi si possono classificare in base:

- alla loro fonte (causa);
- a ciò che colpiscono (effetto).

6.5.1 Processo di gestione dei rischi



La **gestione dei rischi** consiste in: identificare i rischi, valutarne l'impatto, definire strategie preventive o reattive.

Il **processo** della gestione dei rischi si suddivide in:

- **Identificazione dei rischi** = identificare i rischi potenziali e la loro natura (classificazione);
- **Analisi dei rischi** = determinare la probabilità di ogni rischio e la gravità delle sue conseguenze;
- **Pianificazione dei rischi** = piani per affrontare i rischi;
- **Monitoring dei rischi** = si controlla se la probabilità o la gravità degli effetti ogni rischio sono cambiate.

6.5.2 Classificazione dei rischi in base alla causa

- Rischi **tecnologici** = derivano dall'hardware o dal software utilizzati;
- Rischi riguardanti il **personale** = derivano dalle persone del team;
- Rischi **organizzativi** = derivano dall'organizzazione aziendale;
- Rischi **strumentali** = derivano dagli strumenti CASE;
- Rischi dei **requisiti** = derivano dal cambiamento dei requisiti;
- Rischi di **stima** = derivano dalle valutazioni relative a tempi, costi, risorse, o caratteristiche del sistema.

6.5.3 Classificazione dei rischi in base all'effetto

- Rischi di **progetto** = colpiscono la pianificazione (tempi e risorse);
- Rischi di **prodotto** (rischi tecnici) = colpiscono la qualità del sistema;
- Rischi di **business** = colpiscono il committente/cliente o lo sviluppatore sul piano economico.

6.5.4 Analisi dei rischi

L'analisi dei rischi consiste nello stimare la probabilità del rischio e stimare la gravità dei suoi effetti. La valutazione si base su:

- la quantità di informazioni sul progetto, sul processo, sul team, sull'organizzazione, $\cdots$;
- giudizio e esperienza del manager.

Probabilità

- Molto bassa (<10%);
- Bassa (10-25%);
- Moderata (25-50%);
- Alta (50-75%);
- Molto alta (>75%).

Effetti possono essere: insignificanti, tollerabili, gravi e catastrofici.

6.5.5 Pianificazione dei rischi

La pianificazione dei rischi consiste nello sviluppo di strategie per limitare i rischi:

- **Strategie preventive** = piani per ridurre la probabilità che il rischio si verifichi;
- **Strategie reattive** = piani per ridurre gli effetti nel caso il rischio si verifichi.

6.5.6 Monitoring dei rischi

Si verifica per ogni rischio identificato, se la sua probabilità può crescere o decrescere, e se la gravità dei suoi effetti può cambiare. Alcuni fattori (indicatori) permettono di monitorare i rischi.

6.5.7 Monitoraggio e revisione del progetto

La gestione del progetto si basa sulle **informazioni**.

Inizialmente le informazioni sono ridotte, si fanno una pianificazione, una stima dei costi ed una gestione dei rischi in modo preliminare.

Monitoraggio durante il progetto si raccolgono maggiori informazioni, si controlla il rispetto delle milestone e si possono verificare problemi.

Revisione periodicamente si aggiornano pianificazione, stima dei costi e gestione dei rischi. Per i sistemi customizzate, gli aggiornamenti su tempi e costi devono essere concordati con il committente.

6.6 Piano di progetto

Il piano di progetto è un **documento** che raccoglie le informazioni sulle attività di gestione di un progetto, questo si suddivide in:

- **Introduzione** = obiettivi del progetto;
- **Organizzazione** = persone nel team e ruolo di ciascuno;
- **Pianificazione** =
  - **Divisione del lavoro** = divisione del progetto in task, dipendenze, definizione di milestone e deliverable;
  - **Risorse** = assegnamento delle risorse (persone, hardware, software, budget, $\cdots$) ad ogni task; acquisti di hw e sw, personale assunto;
  - **Tempistica (schedule)** = tempi per ogni task, per ogni milestone, per la fine del progetto; tempo per la ricerca di nuove risorse.
- **Gestione dei rischi** = possibili rischi, probabilità, effetti e strategie;
- **Rapporti** = sull'avanzamento del progetto.

Questo documento viene scritto all'inizio del progetto ma viene aggiornato durante lo svolgimento.

7 Modelli di processo

7.1 Attributi del processo software

Visibilità capacità di stabilire ("vedere") lo stato di avanzamento del processo;

- In quale fase ci troviamo?
- Quante fasi mancano alla fine del processo?

Affidabilità i difetti devono essere scoperti durante il processo software, anziché durante l'uso del prodotto.

Robustezza il processo deve essere in grado di continuare nonostante si presentino problemi inaspettati, in particolare il cambiamento dei requisiti.

Rapidità tempo necessario per consegnare il sistema.

Non è possibile ottimizzare tutti gli attributi, visibilità e rapidità sono in contrasto, la produzione di documenti con i risultati di ogni fase rallenta il processo.

Gli attributi dipendono dal modello di processo.

Modello di processo organizzazione delle fasi del processo software;

- Organizzazione di riferimento = si può applicare alla lettera oppure ispirare altre soluzioni.

7.2 Modello a cascata

Il modello a cascata è caratterizzato da **fasi sequenziali**, prima di passare alla fase successiva, la fase corrente deve essere stata completata. Il committente è coinvolto solo nella specifica e può visionare il sistema solo alla consegna.
Il modello a cascata è caratterizzato da:

- **Visibilità** alta = ogni fase produce della documentazione ce ne descrive i risultati. Le fasi sono ben distinte (senza sovrapposizione).
- **Affidabilità** bassa =
  - *Verifica* = si collauda il sistema in blocco, è più difficile trovare difetti in una grande quantità di codice;
  - *Validazione* = il committente può controllare il soddisfacimento dei requisiti solo alla consegna. (É tardi per accorgersi di difetti nei requisiti, bisogna correggere specifica, progettazione e implementazione).
- **Robustezza** bassa = i requisiti non devono cambiare (difficoltà ad effettuare cambiamenti ai requisiti dopo la specifica, il sistema può essere già interamente progettato e magari già implementato).
- **Rapidità** bassa = il committente può disporre del sistema solo alla consegna, dopo una lunghissima attesa dovuta a:
  - Lunga fase di specifica, i requisiti devono essere tutti definiti in dettaglio;
  - Lunga fase di progettazione, la progettazione deve essere dettagliata e completa;
  - Lunga fase di implementazione, bisogna implementare l'intero sistema;

  - Lunga fase di collaudo, bisogna collaudare tutto il sistema.

- Alcuni sviluppatori potrebbero andare in "stato bloccante", cioè attendere la fine dell'attività di un collega;
  - I progettisti devono aspettare la fine della specifica;
  - I programmatori devono aspettare la fine della progettazione;
  - I collaudatori devono aspettare la fine dell'implementazione.

7.2.1 Commenti

I requisiti sono definiti una volta per tutte, all'inizio. Quindi devono essere ben compresi, definiti in modo completo e dettagliato e stabili ("congelati").

Nella realtà raramente è così si possono avere:

1. Difficoltà da parte del committente nell'esprimere i requisiti;
2. Il committente ha un'idea generale dei requisiti del sistema;
3. Capita che il committente cambi idea dopo la specifica.

Nella realtà le fasi non sono così sequenziali:

- Durante la progettazione si scoprono problemi nei requisiti e quindi si ritorna alla specifica;
- Durante l'implementazione si scoprono problemi nella progettazione e quindi si ritorna alla fase di progettazione.

7.3 Prototipo "usa e getta"

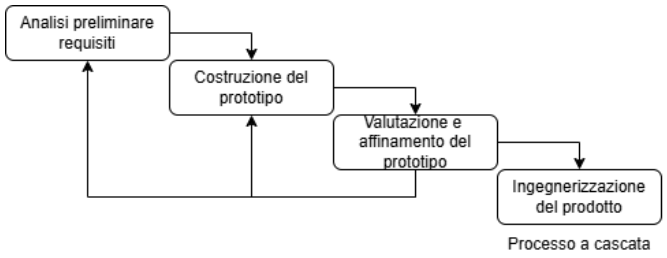
Un **prototipo** è una versione preliminare del sistema che realizza un sottoinsieme dei requisiti.

Per i prototipi "usa e getta":

- Lo scopo del prototipo è comprendere un insieme di requisiti poco chiari; il prototipo realizza tale insieme di requisiti;
- Il prototipo non ha la qualità del prodotto finale (si trascurano i requisiti non funzionali);
- Il prototipo viene mostrato al committente e valutato;
- Il prototipo non evolve nel sistema finale, ma viene "gettato via".

Utile se combinato con il modello a cascata:

- Nella fase di specifica (convalida dei requisiti) si producono e si valutano uno o più prototipi;
- Si chiariscono i requisiti;
- Si sviluppa il vero sistema con il modello a cascata.



7.3.1 Altri tipi

Sperimentazione soluzioni software

- Architetture;
- Linguaggi di programmazione.

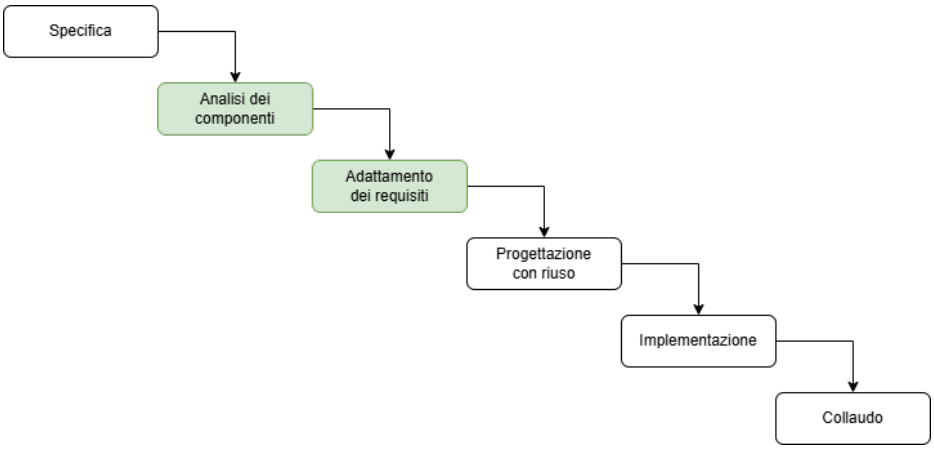
Back-to-back testing il sistema da consegnare e un prototipi alternativo sono sottoposti agli stessi testi. I due sistemi devono dare gli stessi risultati, altrimenti c'è un errore.

Training degli utenti in attesa del vero sistema.

Un prototipo "usa e getta" ha un costo, si può recuperare il costo attraverso un risparmio durante lo sviluppo del sistema, grazie ai requisiti più chiari.

7.3.2 Sviluppo basato sul riuso

CBSE (Component Based Software Engineering).
Finora si è fatta l'assunzione che tutti i componenti del sistema siano sviluppati ad-hoc. CBSE si basa sul riuso di componenti esistenti realizzati per altri sistemi o componenti commerciali COTS (Commercial Off The Shelf). I componenti sviluppati ad-hoc possono essere integrati con i componenti esistenti. L'uso di questo approccio è sempre più frequente a causa dell'affermarsi di componenti standard.



Fasi

- **Specifica**;
- **Analisi dei componenti** = si valutano quali dei componenti esistenti possono essere riutilizzati;
- **Adattamento dei requisiti** = si modificano i requisiti in modo che possano essere realizzati attraverso i componenti esistenti;
- **Progettazione con riuso** = definizione dell'architettura del sistema composta da componenti esistenti e componenti ad-hoc da implementare;
- **Implementazione** = codifica dei componenti ad-hoc ed integrazione con quelli esistenti;
- **Collaudo**.

Attributi Confronto con modello a cascata:

- Stessa **visibilità** (alta);
- **Affidabilità** migliore (media) = i componenti pre-esistenti sono già stati collaudati in precedenza (*verifica*);
- **Robustezza** migliore (media) =
  - necessità di compromessi sui requisiti, i componenti esistenti potrebbero non essere perfettamente adatti a soddisfare i requisiti;
  - Riduzione del costo, un componente esistente costa meno che un componente sviluppato ad-hoc;
  - riduzione dei rischi, limitati ai nuovi componenti.
- **Rapidità** migliore (media) = una sola consegna, ma più veloce, dato che alcuni componenti sono già pronti all'inizio del processo.

7.4 Modelli iterativi

I modelli iterativi prevedono la ripetizione di alcune fasi del processo e la partecipazione del committente durante tutto il processo per verificare il soddisfacimento dei requisiti. Il sistema è un **prototipo evoluto**, il prototipo evolve nel sistema finale (non viene gettato via) aggiungendo gradualmente i requisiti mancanti.

7.4.1 Sviluppo evolutivo

Fasi

1. Definizione ad alto livello dei requisiti;
2. Si seleziona un sottoinsieme dei requisiti;
3. Si aggiorna il prototipo corrente facendo la specifica, la progettazione, l'implementazione e il collaudo dei nuovi requisiti. (In ogni fase è possibile ritornare a quella precedente per risolvere problemi);
4. Si genera una release del sistema che realizza i requisiti considerati finora (prototipo). Si mostra il prototipo al committente che fornisce un feedback;
5. Se necessario, si corregge il prototipo in base al feedback (aggiornando specifica, progettazione ed implementazione);
6. Se ci sono altri requisiti da trattare, si ritorna al passo 2.

Il sistema è realizzato tramite un **prototipo evolutivo**:

- Dopo il primo ciclo, il prototipo realizza il primo sottoinsieme di requisiti;
- Ad ogni ciclo successivo, il prototipo mantiene i requisiti precedenti e viene esteso con altri requisiti;
- Si itera il processo fino a quando tutti i requisiti sono stati considerati.

I requisiti da trattare vengono decisi all'inizio di ogni ciclo (passo 2).

Requisiti Si comincia con i requisiti funzionali più chiari e con maggiore priorità. I requisiti non funzionali si trattano successivamente (efficienza, affidabilità, $\cdots$). Il committente valuta ogni versione del prodotto.
Feedback da parte del committente durante il processo:

- Indica se i requisiti implementati non sono soddisfatti;
- Chiarisce i requisiti poco chiari (ancora da implementare);
- Introduce nuovi requisiti.

Attributi

- **Visibilità** bassa = la documentazione non viene sempre aggiornata, sarebbe troppo costoso se le versioni intermedie fossero numerose, non si sa quante versioni mancano alla fine del processo;
- **Affidabilità** media =
  - *Verifica* = il collaudo riguarda un sottoinsieme di requisiti, è più facile trovare difetti se il codice è limitato, i bachi sono scoperti anticipatamente;
  - *Validazione* = controllo del soddisfacimento dei requisiti per ogni versione, grazie al coinvolgimento del committente, scoperta anticipata di equivoci tra committente e sviluppatori e di requisiti incompleti o mancanti.
- **Robustezza** media =
  - Il processo supporta il cambiamento di requisiti:
    * Specifica e progettazione non devono essere definite in modo dettagliato all'inizio del processo. Una parte dei dettagli si definiscono ad ogni ciclo;
    * Il committente può cambiare idea sui requisiti durante il processo. É possibile che il requisito da cambiare non sia stato ancora analizzato, progettato e implementato.
  - Il codice rischia di diventare poco strutturato:
    * I continui cambiamenti all'implementazione posso corrompere gradualmente la struttura del codice, i cambiamenti diventano sempre più faticosi.
- **Rapidità** media = disponibilità anticipata di un prototipo funzionante (non bisogna aspettare la fine del processo), training anticipato all'uso (parziale) del sistema, scoperta anticipata di difficoltà d'uso da parte dell'utente.
  La consegna di una versione richiede comunque di aspettare il tempo necessario per l'analisi, la progettazione, l'implementazione e il collaudo dei relativi requisiti. Il committente non deve aspettare troppo a lungo le release del sistema ma deve essere disponibile a fornire un feedback. Non tutti i committenti sono disponibili a partecipare al processo.

7.4.2 Sviluppo incrementale

Sviluppo evolutivo non è prevedibile il numero di versioni intermedie necessarie per arrivare alla versione completa del sistema (visibilità bassa).

Sviluppo incrementale mantiene i vantaggi dello sviluppo evolutivo (sviluppo del sistema tramite un prototipo evolutivo) ed ha maggiore visibilità (definizione a propri del numero di versioni del prototipo per arrivare al sistema finale).

Fasi

- **Fasi iniziali**:
  - Specifica ad alto livello;
  - Si stabilisce quale sarà il numero di **incrementi** (versioni) del sistema durante lo sviluppo. Ogni incremento realizza un sottoinsieme dei requisiti;
  - Si stabilisce l'ordine degli incrementi. Ogni incremento ha una priorità in base all'importanza del requisito;
  - Viene progettato il sistema.
- **Fasi cicliche**
  - Specifica in dettaglio dei requisiti dell'incremento;
  - Implementazione dell'incremento, codice ed integrazione;
  - Collaudo dell'incremento e di integrazione;
  - Generazione della release corrente del sistema, il committente la esamina e fornisce un feedback. Se necessario si fanno correzioni aggiornando specifica ed implementazione.

Attributi

- **Visibilità** alta = il processo è meno soggetto all'incertezza; all'inizio del processo si definiscono:
  - L'architettura del sistema;
  - Il numero di incrementi (versioni) e i requisiti realizzati in ogni incremento, si conosce sempre il numero di versioni che servono per arrivare alla fine. É più facile pianificare la documentazione sapendo il numero di versione;
  - L'ordine degli incrementi.
- **Affidabilità** e **Rapidità** = come per sviluppo evolutivo;
- **Robustezza** media = supporta il cambiamento dei requisiti:
  - I requisiti non devono essere definiti in modo dettagliato all'inizio del processo. Un sottoinsieme dei requisiti è analizzato ad ogni ciclo.
  - Il committente può cambiare idea sui requisiti durante il processo, purché non comporti un cambiamento dell'architettura (già definita all'inizio);
  - Conoscendo in anticipo l'architettura e il numero di incrementi, il codice può essere strutturato più ordinatamente.

7.5 Riepilogo

	CASCATA	RIUSO	EVOLUTIVO	INCREMENTALE
Visibilità	Alta	Alta	Bassa	Alta
Affidabilità	Bassa	Media	Media	Media
Robustezza	Bassa	Media	Media	Media
Rapidità	Bassa	Media	Media	Media

8 Modelli agili

Per rispondere alla pressione concorrenziale, oggi il software deve essere consegnato in modo rapido. Il modello a cascata è poco adatto alla rapidità, invece lo sviluppo evolutivo ed incrementale sono migliori sotto questo punto di vista.

Per team composti da poche persone, e per progetti limitati, seguire un modello di processo "prescrittivo" può generare overhead. I modelli precedenti sono troppo vincolanti per un piccolo team. Rispettare le fasi del processo può rubare tempo utile all'implementazione e posticipare la consegna. In un gruppo piccolo, non ci sono ruoli precisi (ognuno fa un po' di tutto).

Negli anni '90 viene ideato il **modello "agile"** che risulta essere più **flessibile** perché le persone non si adattano ad un modello di processo "prescrittivo", ma è il processo che si adatta alle esigenze delle persone. Lo scopo è la **consegna rapida** del prodotto. Riguarda progetti limitati, realizzati da team di sviluppo composti da poche persone, non riguarda progetti industriali.

8.1 XP (eXtreme Programming)

É il più noto modello di processo tra quelli "agili". XP è una forma "estrema" di **sviluppo evolutivo**:

- La **rapidità** è l'obiettivo fondamentale, i tempi di consegna sono ristretti, l'**implementazione** è la fase su cui ci si concentra. Le altre fasi si eseguono solo ad alto livello, per guadagnare tempo. Servono accorgimenti che garantiscano la **qualità** del prodotto nonostante la rapidità dello sviluppo;
- Numerose release del sistema;
- Prevede il cambiamento dei requisit;
- Il committente è particolarmente coinvolto nel processo.

8.1.1 Fasi

- **Specifica a (molto) alto livello** =
  - i requisiti sono definiti soltanto ad alto livello, sotto forma di scenari non strutturali;
  - gli scenari sono ordinati per priorità;
  - ogni scenario viene suddiviso in incrementi;
  - una specifica dettagliata richiederebbe più tempo riducendo così la rapidità del processo.
- **Pianificazione** = si stabiliscono insieme al committente
  - quali incrementi saranno realizzati dalla prossima release, di solito una release realizza uno o pochi incrementi;
  - i tempi di consegna della prossima release;
  - i test da eseguire in fase di collaudo.
- **Progettazione semplice** =
  - si definisce o si aggiorna l'architettura a (molto) alto livello, la progettazione definisce solo ciò che è strettamente necessario;
  - i dettagli saranno stabiliti mentre si implementa;
  - una progettazione dettagliata richiederebbe più tempo riducendo così la rapidità del processo.
- **Implementazione** =

  - **Programmazione a coppie** =
    * ogni linea di codice è scritta e visionata da due persone, ciò aumenta la probabilità di "vedere" un difetto (prima modalità di collaudo);
    * il *driver* scrive il codice, il *navigatore* può controllarlo e fare refactoring.
      · Codice più pulito, uniforme e capibile: stile di indentazione, posizione delle parentesi graffe, stile dei nomi, posizione dei commenti, ecc.
      · Codice più efficiente: evitare ridondanze, variabili e cicli inutili;
      · Non si altera il comportamento, ma diventa più mantenibile.
    * Condivisione della conoscenza del codice.
      · Le coppie cambiano dinamicamente a seconda degli incrementi;
      · Quando uno sviluppatore cambia compagno può spiegargli la parte di codice che aveva scritto;
      · Gradualmente, cambiando le coppie varie volte, ogni parte di codice è conosciuta da un numero maggiore di sviluppatori;
      · Alla fine ogni sviluppatore conosce tutto il codice.
  - **Sviluppo con test iniziale** = i test-case sono definiti durante la fase di pianificazione, prima dell'implementazione dell'incremento (TDD (Test-Driven Development));
  - **Possesso collettivo** = un programmatore può modificare qualunque parte del codice.

- **Collaudo** =
  - I test-case sono scelti prima dell'implementazione;
  - 3 tipi di testing ad ogni incremento:
    1. **Testing dell'incremento (unit test)** = fatto sull'incremento, se l'incremento supera il test, viene integrato. I test-case sono stabiliti dagli sviluppatori.
    2. **Test di integrazione** = tutti i test di unità precedenti sono ripetuti quando si integra un incremento;
    3. **Test di accettazione** = il committente controlla il soddisfacimento dei requisiti, i test-case sono stabiliti da sviluppatori e committente;
  - Uso di strumenti CASE per il testing (riduce i tempi del collaudo, quindi migliora la rapidità).

8.1.2 Ruolo del committente

Il committente è molto più coinvolto nel processo, rispetto ai modelli precedenti:

- assegna le priorità ai requisiti;
- decide cosa deve contenere ogni release;
- decide le scadenze per le release;
- partecipa alla definizione dei requisiti (scenari);
- partecipa alla definizione dei test-case;
- partecipa all'esecuzione dei test-case;
- si trova nella stessa sede degli sviluppatori.

8.1.3 Commenti

Committente on-site = presenza a temo pieno del committente (o di un suo rappresentante) nel team di sviluppo. Il committente deve essere una singola entità e deve essere costantemente a disposizione del team di sviluppo.

I programmatori lavorano su ogni parte del sistema (**possesso collettivo**) = devono avere varie conoscenze, devono essere flessibili, non ci sono ruoli prestabiliti, devono lavorare nella stessa sede, chiunque può modificare qualunque parte del sistema e chiunque trova un problema lo risolve.

8.1.4 Attributi

- **Visibilità** bassa =
  - non c'è la documentazione nella sua forma classica (documentazione in XP = scenari, codice e test-case);
  - non si conosce il numero di versioni per arrivare alla fine, i requisiti di ogni versione sono scelti di volta in volta).
- **Rapidità** alta = è ottenuta attraverso:
  - specifica a (molto) alto livello;
  - progettazione a (molto) alto livello;
  - uso di strumenti CASE durante il collaudo;
  - release molto frequenti (ogni 1-2 settimane, time boxed), ogni release realizza pochi incrementi quindi lo sviluppo richiede poco tempo.
- **Affidabilità** alta = la qualità del prodotto è mantenuta nonostante la rapidità alta;
  - *Verifica*
    - sviluppo con test iniziale (TDD);
    - release frequenti, tante release, tanti collaudi;
    - ogni collaudo riguarda piccoli incrementi, è più facile trovare un difetto se il codice è limitato e scritto recentemente.
  - *Validazione* = coinvolgimento costante del committente:
    - il committente valuta frequentemente il prodotto fornendo feedback sul soddisfacimento dei requisiti
- **Robustezza** alta =
  - supporta il cambiamento dei requisiti (prototipo evolutivo, committente on-site);
  - il codice rimane ben strutturato e ordinato (programmazione a coppie con refactoring).

8.1.5 Refactoring

Il codice è soggetto a frequenti modifiche, attraverso le seguenti fasi:

1. modifica per aggiornare il codice;
2. testing;
3. refactoring per mantenerlo "pulito";
4. rifare testing (test di regressione).

Il refactoring non modifica il comportamento del codice, ma ne migliora la struttura interna.

Il refactoring serve per:

- Eliminare codice duplicato (tramite funzioni/metodi);
- Ridurre la lunghezza di una funzione/metodo distribuendo il codice in varie funzioni/metodi;
- Dare nomi significativi a variabili, funzioni, classi, metodi, attributi per far capire il loro scopo;
- Fondere classi simili in un'unica classe;
- Uniformare l'indentazione e la posizione dei commenti.

Codice semplice e uniforme → più leggibile

8.1.6 TDD (Test Driven Developmente)

Invece di scrivere prima il codice e poi i test-case, si scrivono prima i test-case e poi il codice, in modo da superarli.

1. Scrivi test che fallisce;
2. Scrivi codice che supera il test;
3. Refactoring + Ripeti test;
4. Go to 1.

8.2 Scrum

Scrum = "pacchetto di mischia" (rugby) = squadra coesa e compatta.

Ruoli

- Product owner = committente;
- Scrum team = 6-10 sviluppatori;
- Scrum master = manager del team.

Fasi del processo

- Definizione del *product backlog* = insieme dei requisiti;
- Da 3 a 8 *sprint*. Ogni sprint:
  - contiene micro-waterfall (progettazione, implementazione e collaudo);
  - genera un prototipo evolutivo da mostrare al product owner;
  - dura 1 mese (time boxed).

8.2.1 Backlog

Product backlog è un elenco di tutti i requisiti e relative priorità. Viene definito all'inizio del processo dal product owner. Può essere aggiornato dopo ogni sprint dallo scrum master. I requisiti con priorità più alta saranno sviluppati prima.

Sprint backlog è un sottoinsieme del product backlog da sviluppare durante uno specifico sprint.

8.2.2 Meeting

Sprint planning meeting è una riunione all'inizio di ogni sprint.
Partecipanti : product owner, scrum master e scrum team.

- Aggiornamento del product backlog (se necessario);
- Definizione dello sprint backlog dello sprint corrente;
- Definizione dello sprint goal (obiettivo delo sprint corrente).

Scrum meeting (daily scrum) è una riunione giornaliera.
Partecipanti : scrum master, scrum team (15-30 minuti).

- Ogni sviluppatore descrive cosa ha fatto il giorno prima, gli impedimenti riscontrati, cosa farà nel giorno corrente;
- Scrum master controlla l'avanzamento del lavoro, aggiorna lo sprint backlog (se necessario), aiuta a superare gli impedimenti e ad impostare il lavoro del giorno corrente.

Sprint review meeting è una riunione alla fine di ogni sprint.
Partecipanti : product owner, scrum maste e scrum team.

- Si mostra il prototipo corrente;
- Il product owner indica se lo sprint goal è stato raggiunto o meno.

8.2.3 Commenti

Le riunioni giornaliere promuovono la comunicazione tra gli sviluppatori, la conoscenza collettiva e la responsabilità collettiva del progetto (gli sviluppatori formano uno scrum). Il coinvolgimento del committente limita i fraintendimenti sui requisiti.

Scrum è più un modello gestionale che di processo perché non fornisce indicazioni sulle fasi fondamentali (specifica, progettazione, implementazione e collaudo), per questo può essere integrato con altri modelli.

8.2.4 Attributi

- **Visibilità** media =
  - controllo giornaliero dell'avanzamento del lavoro;
  - non si conosce a priori il numero di sprint.
- **Affidabilità** media =
  - coinvolgimento mensile del committente;
  - il collaudo è effettuato per ogni prototipo.
- **Robustezza** alta =
  - gli impedimenti sono notificati e risolti giornalmente;
  - il backlog può essere modificato dopo ogni sprint.
- **Rapidità** media =
  - il prototipo viene rilasciato ogni mese.

8.3 DevOps

Perché Passaggio da "System of record" a "Systems of engagement".

- **System of record** = sistemi software "tradizionali"
  - registrano (record) grandi quantità di dati e transazioni;
  - stabili e affidabili : una o due release all'anno.
- **Systems of engagement** = applicazioni mobili, cloud, big data, social media, $\cdots$
  - maggiore frequenza di utilizzo da parte dell'utente (engagement);
  - richiesta di alte prestazioni;
  - tempi rapidi di sviluppo e manutenzione (i requisiti cambiano spesso in base alle richieste degli utenti e al mercato).

Soggetti coinvolti

- Esperti del dominio (Dev);
- Sviluppatori (Dev);
- Amministratori di sistema (Ops);
- Addetti alla qualità del software (Ops);
- Utenti finali (Ops).

Scopi

- Creare un "ponte" tra sviluppatori, amministratori e utenti;
- Rilasciare software in modo continuo e diretto (cloud);
- Avere frequenti feedback sull'usabilità (User eXperience, UX).

8.3.1 Processi interni

Dev sviluppare e testare come in un ambiente di produzione (catena di montaggio).

Release distribuire con processi automatici.

Ops monitorare la qualità operativa.

Ritorno di feedback da parte di utenti, addetti alla qualità, amministratori.

8.3.2 Processi

CI (Continuous Integration and testing) Gli sviluppatori integrano il codice diverse volte al giorno tramite repository condiviso (SVN, git, ecc.).
Ad ogni check-in:

- il codice viene assemblato da uno strumento di build automatico per rilevare eventuali problemi;
- si eseguono in modo automatico test funzionali e non funzionali (prestazioni, sicurezza, ecc.).

CD (Continuous Delivery e deployment) L'obiettivo è quello di costruire il software in modo da rilasciarlo in qualsiasi momento (production-ready). Ogni singola modifica che passa con successo i test viene automaticamente distribuita (cloud).

CO (Continuous Operations) Utilizzo da parte degli utenti finali, le modifiche software o hardware non devono interrompere l'uso del software da parte degli utenti. Monitoring del sistema da parte degli amministratori di sistema e addetti alla qualità.

CA (Continuous Assessment)

- **Feedback loops** = continuo monitoraggio dello stato e delle prestazioni, registrando l'esperienza degli utenti e informando amministratori e sviluppatori. (Ottimizzazione continua del software, aumento della soddisfazione d'uso (UX) degli utenti finali)
- **Planning prioritization** = ai feedback (nuove funzioni o bug-fixing) viene data una priorità in base alle esigenze di business e alle richieste degli utenti finali.
- **Portfolio investment** = quando si ricevono i feedback il gruppo di pianificazione assegna una priorità anche in termini di investimenti necessari.

8.3.3 Attributi

- **Visibilità** media =
  - non si conoscono il numero e frequenza delle release;
  - sviluppo organizzato come una catena di montaggio.
- **Affidabilità** alta =
  - integrazione e collaudo automatico di ogni commit.
- **Robustezza** alta =
  - feedback numerosi e frequenti da parte degli utenti;
  - monitoring di amministratori e addetti alla qualità.
- **Rapidità** alta =
  - software rilasciato in modo continuo e automatico.

8.3.4 Riepilogo

	CASCATA	RIUSO	EVOLUTIVO	INCREMENTALE	XP	SCRUM	DEV-OPS
Visibilità	Alta	Alta	Bassa	Alta	Bassa	Media	Media
Affidabilità	Bassa	Media	Media	Media	Alta	Media	Alta
Robustezza	Bassa	Media	Media	Media	Alta	Alta	Alta
Rapidità	Bassa	Media	Media	Media	Alta	Media	Alta

Parte II
UML

9 Modelli di specifica

9.1 Deduzione dei requisiti

UML = Unified Modeling Language.

Deduzione = si compie un'indagine per "dedurre" requisiti. Si stabiliscono i requisiti raccogliendo informazioni da:

- Studio del dominio applicativo del sistema (in UML);
- Dialogo con stakeholder;
- Studio di sistemi già realizzati e con lo stesso dominio applicativo o con un dominio simile;
- Studio dei sistemi con cui dovrà interagire quello da sviluppare.

Dominio applicativo = insieme di oggetti reali a cui il sistema software viene applicato. Il dominio definisce il contesto in cui opera il sistema. Ogni oggetto del dominio è caratterizzato da **attributi**. Alcuni oggetti hanno gli stessi attributi e quindi possono essere raggruppati in sottoinsiemi del dominio detti **classi**.

Un dominio può essere visto come un insieme di classi.

9.1.1 Diagramma delle classi

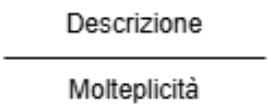
Le classi del dominio corrispondono alle classi del diagramma. Si possono rappresentare gli attributi delle classi, le relazioni tra classi e si possono generalizzare classi con attributi in comune.

Nodo classe Composto da 3 compartimenti.

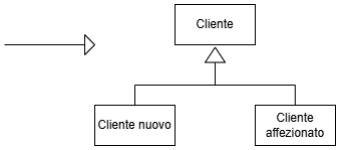
Nome della classe
Attributi
Operazioni

Archi Rappresentano le relazioni tra classi.
Molteplicità dell'associazione = numero di oggetti delle classi coinvolte. ($* = \infty$).

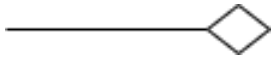
- **Associazioni** = relazione logica tra classi.



- **Generalizzazione** = una classe eredita attributi e metodi da un'altra.



- **Composizione** = un oggetto di una classe contiene oggetti di altre classi.



  - Rombo pieno = **forte**, gli oggetti possono esistere solo all'interno del contenitore;
  - Rombo vuoto = **debole**, gli oggetti possono esistere anche all'esterno del contenitore.

  Se non scrivo niente la molteplicità sottintesa è: (1 0...$*$)

Attributi Notazione = *visibilità nome_attributo : tipo_attributo[molteplicità] = default*
Tipi di visibilità di un attributo:

- Public (+) = visibile da tutte le classi;
- Private (-) = visibile solo dalla classe in cui è definito.
- Protected (#)

Tipi di attributo:

- Tipi primitivi = integer, float, string, Boolean, $\cdots$
- Classi = si utilizza un **arco di dipendenza** (orientato e tratteggiato);
  - classe dove il tipo è usato $- - - >$ classe dove il tipo è definito;
  - A $- - - >$ B significa che A dipende da B perché A usa B come tipo di attributo.

Molteplicità di un attributo serve per dichiarare che un attributo è un array. Si esprime come la molteplicità delle relazioni tra classi.

Note (commenti) In ogni tipo di diagramma UML è possibile scrivere delle note (commenti) e collegarle ai nodi o agli archi a cui si riferiscono. Un arco tratteggiato collega la nota al nodo/arco.

9.1.2 Diagramma dei casi d'uso

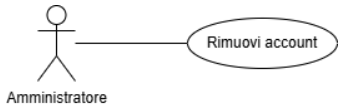
Tipi di nodo

- **Caso d'uso** = indica un requisito funzionale (servizio) del sistema.

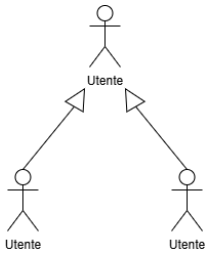
- **Attore** = entità esterna che interagisce con il sistema (utente o un altro sistema).

Tipo di arco

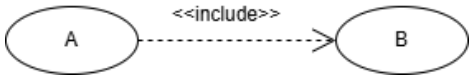
- **Associazione tra attore e caso d'uso** = indica che l'attore interagisce con il sistema durante l'esecuzione del caso d'uso.



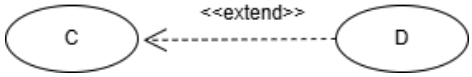
- **Generalizzazione** = l'utente specifico eredita i casi d'uso dell'utente più generico.



- **Dipendenze** = un caso d'uso determina l'esecuzione di un altro caso d'uso ($- - - >$)
  - **Include** = l'esecuzione di un caso d'uso include necessariamente un altro caso d'uso.



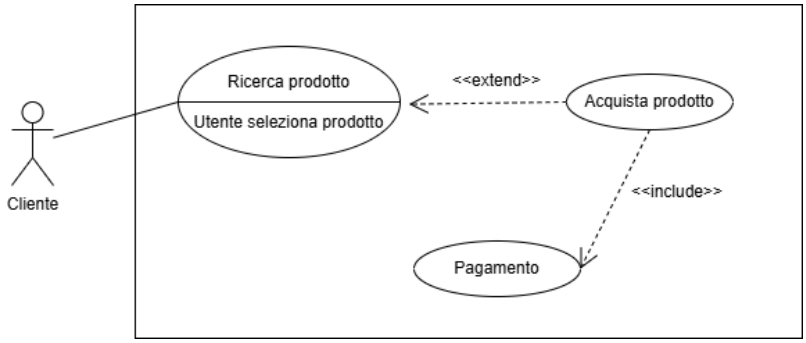
    Eseguo prima A e poi, obbligatoriamente, B.
  - **Extend** = un caso d'uso può estendere un altro caso d'uso se vale una certa condizione o si verifica un certo evento (*extension point*, condizione che determina l'extend).



    Eseguo C, se vale una certa condizione, eseguo D.

Altro

- **Boundary** = riquadro che separa i casi d'uso dagli attori.



9.2 Analisi dei requisiti

9.2.1 Diagramma degli stati

Ogni singolo caso d'uso (requisito funzionale) si può modellare in termini di stati e transizioni tra stati che si verificano nel sistema durante l'esecuzione del caso d'uso.

Stato un intervallo di tempo durante il quale il sistema:

- si trova in una particolare condizione;
- sta svolgendo una o più attività;
- è in attesa di un evento,

La durata dello stato è delimitata da due eventi:

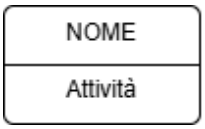
- ingresso nello stato;
- uscita dallo stato.

Il passaggio da uno stato ad un altro è detto **transizione di stato**.

9.2.2 Elementi

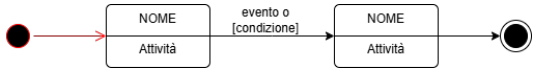
Stato nodo con 2 compartimenti:

- Nome;
- Attività interne dello stato.



Archi di transizione rappresentano le transizioni di stato (passaggio da uno stato ad un altro). Un'etichetta sull'arco indica la causa della transizione:

- [Condizione] (vero/falso);
- Evento;
- Periodo di tempo.



Stati particolari C'è 1 stato iniziale e ci sono $n \geq 0$ stati finali.

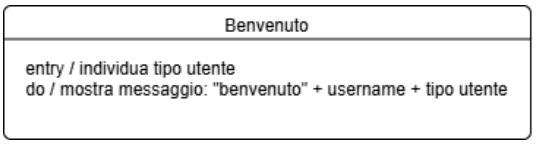
- **Stato iniziale** = non ha archi entranti provenienti da altri stati (**obbligatorio**).

- **Stato finale** = non ha archi uscenti verso altri stati (**opzionale** se presente un ciclo).

9.2.3 Attività all'interno di uno stato

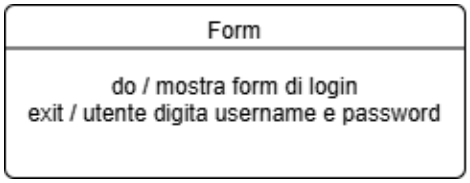
Attività in entrata = attività svolta nel momento in cui si entra nello stato.

- Notazione = *entry / attività*;
- Non è l'evento o la condizione di ingresso nello stato.



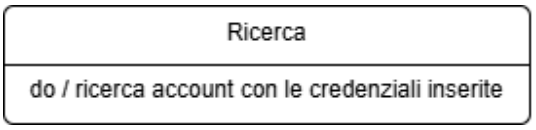
Attività di uscita = attività svolta nel momento in cui si esce dallo stato.

- Notazione = *exit / attività*;
- Non è l'evento o la condizione di uscita dallo stato.



Do-activity = attività svolta per tutta la durata dello stato.

- Notazione = *Do / attività*;



9.2.4 Macrostato

Un macrostato è uno stato che contiene al suo interno degli stati e delle transizioni. Gli stati e le transizioni interne possono essere rappresentate in un diagramma degli stati separati. Il macrostato può essere usato come riferimento ad un altro diagramma degli stati.

Si applica quando un caso d'uso invoca (include/extend) un altro caso d'uso.

9.3 Corrispondenza tra diag. dei casi d'uso e diag. degli stati

Se nel diagramma dei casi d'uso, A << *include* >> B oppure B << *extend* >> A, allora il diagramma degli stati di A deve contenere:

- Un macrostato B;
- Un arco di transizione da uno stato di A verso B;
- Se necessario, un arco da B verso uno stato di A.

Se A << *include* >> B allora la transizione da uno stato di A al macrostato B deve avvenire prima o poi.

Se A << *extend* >> B la transizione da uno stato di A al macrostato B avviene solo in situazioni particolari.

10 Modelli di progettazione

10.1 Attività di progettazione

Progettazione architetturale

- **Strutturazione del sistema** = definire sottosistemi e moduli;
- **Deployment** = distribuzione dei componenti sui dispositivi;
- **Metodo di controllo** = definire quali componenti invocano le operazioni e quali le eseguono;
- **Modellazione del comportamento** = definire le interazioni tra componenti allo scopo di svolgere una certa funzione.

Progettazione delle strutture dati

- Strutture dati che mantengono i dati del sistema.

Progettazione degli algoritmi

- Algoritmi per la funzione del sistema (in UML).

Progettazione della GUI (Graphical User Interface)

10.2 Diagramma dei componenti

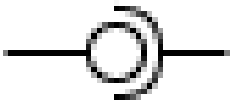
Rappresenta i sottosistemi e i moduli interni che compongono un'architettura.

10.2.1 Elementi

Componente nodo che rappresenta un sottosistema o un modulo.

Interfaccia rappresenta il fatto che un componente richiede l'esecuzione di operazioni da parte di un altro componente. Appare come la coppia di archi "ball & socket".

- Ball = indica che un componente mette a disposizione ed esegue delle operazioni;
- Socket = indica che un altro componente richiede (invoca) tali operazioni.

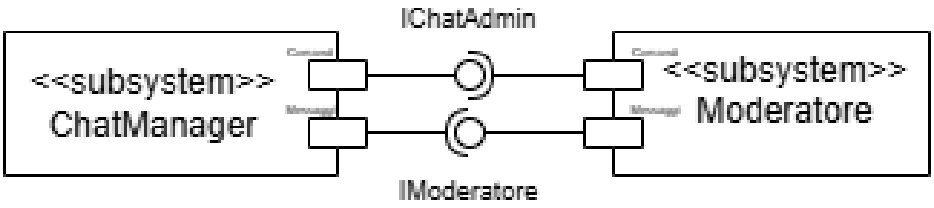


10.2.2 Porte

Una coppia di componenti potrebbe avere più di una interfaccia, queste potrebbero avere lo stesso verso o versi opposti. Le interfacce possono essere distinte:

- assegnando un'etichetta ad ogni interfaccia;
- indicando i dati o le operazioni che le riguardano.

Le porte sono identificate da un simbolo di forma quadrata collocato sul componente e toccato da un arco di tipo "ball" o "socket". Il nome della porta indica il dato o l'operazione associata all'interfaccia. Le porte relative alla stessa interfaccia hanno lo stesso nome.



10.3 Diagramma di deployment

Necessario per la distribuzione fisica del sistema.

10.3.1 Elementi

Nodo elemento in grado di eseguire software. Esistono 2 tipi di nodo:

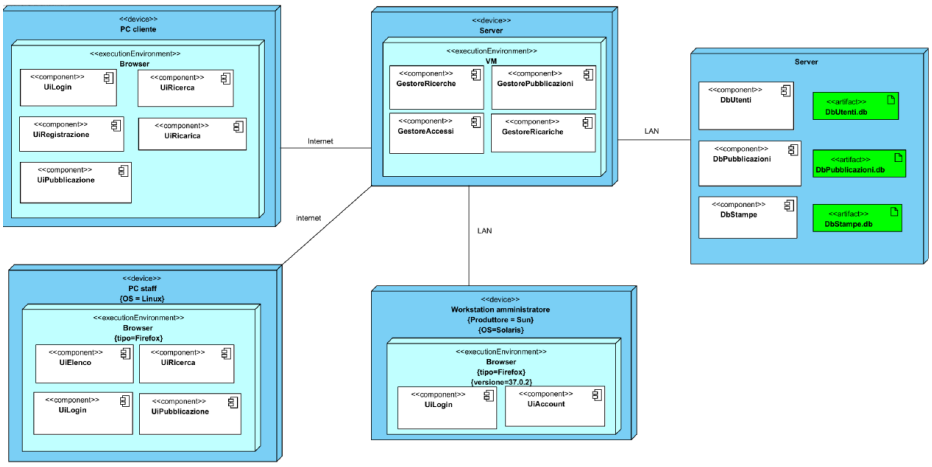
- **Dispositivo** = dispositivo hardware in grado di eseguire software (computer, cellulare, ecc.);
- **Ambiente di esecuzione** = software in grado di eseguire altro software (browser, macchina virtuale, ecc.).

I nodi contengono:

- **Componenti**;
- **Elaborati (artifact)** = file eseguibili, file di dati, file di configurazione, pagine html, ecc.
- **Etichette** = indicano caratteristiche del nodo.

I nodi possono esserei annidati, un dispositivo può contenere un ambiente di esecuzione.

Path di comunicazione arco tra due nodi che indica la comunicazione tra due nodi, il protocollo, la tecnica o il mezzo di comunicazione (http, TCP/IP, RMI, socket, ethernet, LAN, internet).



10.4 Diagramma delle classi

Il diagramma dei componenti è poco descrittivo, allora si può "tradurre" in un diagramma delle classi per definire più dettagli.

- Ogni modulo diventa una classe;
- Ogni interfaccia diventa un arco di dipendenza A $- - - >$ B
  - A è il componente con "socket";
  - B è il componente con "ball".
- Si aggiungono le classi per rappresentare le strutture dati di un componente (archi di composizione);
- Si aggiungono le operazioni di ogni classe (dopo aver fatto i diagrammi di sequenza).

10.4.1 Elementi

Nodo classe rappresenta una classe. É composto da 3 compartimenti:

- Nome della classe;
- Attributi;
- Operazioni (metodi).

10.4.2 Tipi di relazioni tra classi

Composizione un componente (classe) contiene una struttura dati (classe).

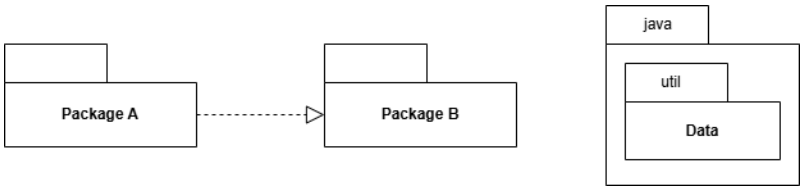
Dipendenza una classe invoca operazioni di un'altra classe, A $- - - >$ B (A dipende da B, cioè A invoca le operazioni di B).

10.5 Diagramma di package

Serve per organizzare le classi in package in vista dell'implementazione.

Package contenitore di classi, i package possono essere annidati.

Archi di dipendenza stesso ruolo nel diagramma delle classi. Le dipendenze tra package derivano da dipendenze tra classi presenti in package diversi (package A $- - - >$ package B).



10.6 Diagramma di sequenza

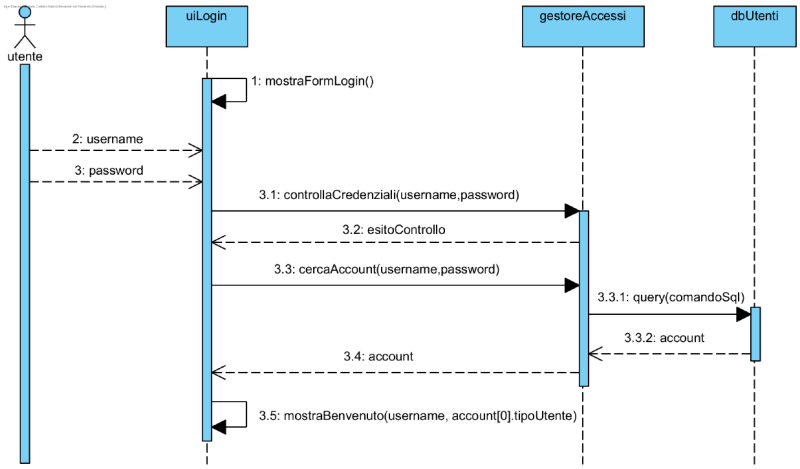
Mostra una sequenza temporale di interazioni tra oggetti (e attori) allo scopo di svolgere un determinato caso d'uso. L'interazione avviene attraverso lo scambio di messaggi.

Partecipanti

- **Oggetti** = rappresentano componenti e sono istanze delle classi;
- **Attori** = rappresentano tipi di utente o sistemi esterni.

Il diagramma è sviluppato in 2 dimensioni:

- **Verticale** = rappresenta il tempo;
- **Orizzontale** = contiene gli oggetti e gli attori.



10.6.1 Messaggi

Invocazione di operazioni un oggetto richiede ad un altro oggetto di eseguire una determinata operazione.

- Notazione grafica = arco orientato e continuo, tracciato dalla linea di vita dell'oggetto invocante verso la linea di vita dell'oggetto invocato;
- Un'etichetta sull'arco indica l'operazione invocata e gli eventuali parametri.

Ritorno di valore valore prodotto dall'esecuzione dell'operazione.

- Notazione grafica = arco orientato e tratteggiato, tracciato dalla linea di vita dell'oggetto invocato verso la linea di vita dell'oggetto invocante;
- Un'etichetta sull'arco indica il valore ritorno che ha un tipo (tipo primitivo, classe), le operazioni di tipo void non hanno un valore di ritorno.

Autoinvocazione un oggetto invoca un'operazione su se stesso.

- Notazione grafica = arco orientato e continuo, tracciato dalla linea di vita dell'oggetto verso la stessa linea;
- Si può rappresentare l'eventuale valore di ritorno in modo analogo (arco tratteggiato).

Input/Output con l'attore archi orientati e tratteggiati.

- Input = da attore (utente) verso oggetto (componente);
- Output = da oggetto verso attore.

Messaggio con ritardo se trascorre del tempo tra l'invio del messaggio e la sua ricezione, allora l'arco corrispondente (continuo o tratteggiato) può essere tracciato in modo obliquo (se non c'è ritardo l'arco è orizzontale).

10.6.2 Vita e attivazione degli oggetti

Linea di vita indica il periodo di esistenza dell'oggetto.

- Notazione grafica = linea tratteggiata verticale che parte dall'oggetto e prosegue verso il basso.

Creazione/distruzione di oggetti

- Gli oggetti già esistenti inizialmente sono indicati nella parte alta del diagramma;
- Un oggetto può essere creato tramite un'operazione in un momento successivo. Viene indicato più in basso, a livello del tempo di creazione.
- Un oggetto può essere distrutto tramite un'operazione in un momento successivo. La linea di vita si interrompe a livello del tempo di distruzione.

Periodo di attivazione periodo di tempo in cui l'oggetto è attivo

- esegue un'operazione;
- attende un valore di ritorno;
- Notazione grafica = rettangolo di altezza pari al tempo di attivazione, posto sopra la linea di vita.

10.6.3 Frame

Un insieme di messaggi può essere racchiudo in un frame. I frame possono essere annidati ed esistono 4 tipi:

- **Loop** = i messaggi all'interno del frame sono ripetuti finché vale una certa condizione (o guardia);
- **Opt** = se la condizione è vera vengono eseguiti i messaggi nella sezione. Può avere una seconda sezione che viene eseguita se la condizione è falsa.
- **Alt** = il frame è diviso in varie sezioni; ogni sezione è eseguita se vale la condizione dichiarata nella sezione stessa;
- **Ref** = il frame è un riferimento ad un altro diagramma di sequenza.

10.7 Dal diagramma degli stati al diagramma di sequenze

In fase di **specifica**, ogni caso d'uso è stato modellato da un diagramma degli stati.

In fase di **progettazione**, ogni caso d'uso deve essere rappresentato da un diagramma di sequenza.

- Mostra le interazioni tra i componenti (oggetti) del sistema durante l'esecuzione del caso d'uso;
- I componenti devono essere quelli previsti nei diagrammi dei componenti e delle classi.

Il diagramma di sequenza deve essere coerente con il diagramma degli stati corrispondente.

- Le transizioni di stato e le attività rappresentate nel diagramma degli stati devono corrispondere alle interazioni rappresentate nel diagramma di sequenza;
- **Bisogna "tradurre" il diagramma degli stati in un diagramma di sequenza**.

10.7.1 Corrispondenze tra macrostati e ref

Se il diagramma degli stati di A contiene il macrostato B, allora il diagramma di sequenza di A deve contenere un riferimento (**frame Ref**) a B.

All'interno del diagramma degli stati di A, viene invocato il diagramma degli stati di B. All'interno del diagramma di sequenza di A, viene invocato il diagramma di sequenza di B.

10.7.2 Metodo di avvio

Ogni servizio (caso d'uso) avrà il proprio metodo di avvio, questo metodo può avere dei parametri in ingresso se necessari (eg. Visualizza Pubblicazione << *extend* >> Ricerca Pubblicazione, il metodo è invocato da UI_Ricerca su UI_Pubblicazione indicando la pubblicazione da visualizzare) o avere un valore di ritorno, se necessario (eg. Stampa Pubblicazione << *include* >> Pagamento, il metodo è invocato da UI_Pubblicazione su UI_Pagamento e ritorna true se il pagamento è riuscito, false altrimenti).

In ogni caso bisogna **indicare il metodo di avvio nel diagramma di sequenza**.

10.8 Corrispondenza tra il diag. di sequenza (SD) e il diag. delle classi (CD)

Ad ogni oggetto nel SD corrisponde una classe nel CD. Ogni operazione invocata su un oggetto nel SD, deve essere dichiarata nella classe corrispondente nel CD. L'invocazione di un'operazione nel SD deve rispettare la sua dichiarazione nel CD : numero di parametri, tipi dei parametri, tipo del valore di ritorno.

Se l'oggetto di classe A invoca operazioni sull'oggetto di classe B nel SD, allora nel CD un arco di dipendenza è tracciato dalla classe A verso la classe B.

10.9 Dichiarazione di operazioni nel diagramma delle classi

Dichiarazione di un'operazione

visibilità `nome_operazione`(*param1* : *tipo*[*molteplicità*], *param2* : *tipo*[*molteplicità*], . . .) :
tipo_valore_ritorno[*molteplicità*]

Tipi di visibilità di un'operazione

- Public (+) = visibile da tutte le classi;
- Private (-) = visibile solo dalla classe in cui è definito.

Tipi di parametro o valore di ritorno

- Tipi primitivi = integer, float, string, boolean, ecc.
- Classi.

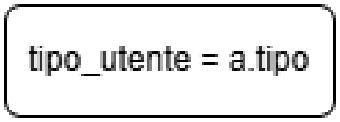
10.10 Diagramma di attività

Il diagramma di attività è una versione evoluta dei digrammi di flusso. Viene utilizzato per la definizione dell'algoritmo relativo ad un'operazione presente nel diagramma di sequenza; mostra i passi necessari per generare il valore di ritorno a partire dai parametri in ingresso.

10.10.1 Elementi

Attività insieme strutturato di azioni (algoritmo).

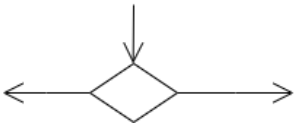
Azioni passi elementari dell'algoritmo



- Azione iniziale;

- Azione finale.

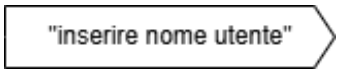
Decisione se vale una certa condizione si esegue una certa azione, altrimenti ne eseguo un'altra.



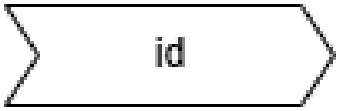
CF (Control Flow) archi che collegano azioni, decisioni, segnali, barre di sincronizzazione per indicare l'ordine di esecuzione.

Segnali

- **Inviato** = produzione di output verso un attore;

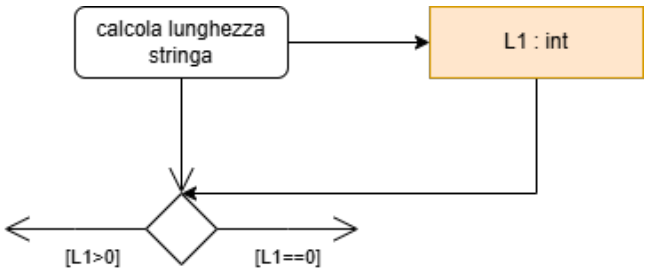


- **Ricevuto** = produzione di input da parte di un attore.

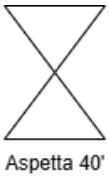


Nodi oggetto rappresentano parametri in ingresso, valori di ritorno, variabili/oggetti locali.

- **OF (Object Flow)** = archi che collegano i nodi oggetto alle azioni/decisioni/degnali che ne impostano, utilizzano o ritornano il valore.
  - azione → oggetto = assegnamento o return del valore;
  - oggetto → azione = utilizzo del valore.
- **CF** e **OF** hanno la stessa notazione grafica.



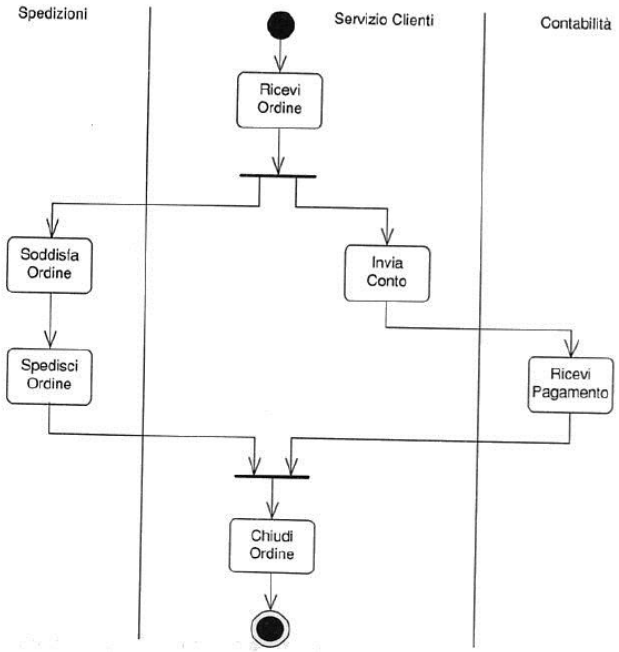
Attesa sospensione dell'attività per un certo tempo.



Barre di sincronizzazione

- **Fork** = il completamento di un'azione avvia l'esecuzione di più azioni in parallelo;
- **Join** = il completamento di più azioni avvia l'esecuzione di un'azione.

Partizioni (swimlanes) indicano il soggetto (componente, attore, ecc.) che esegue l'azione.

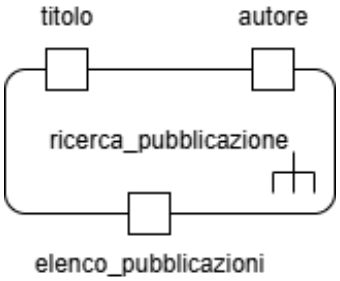


10.10.2 Attività esterna

Prima di ritornare un valore, il componente potrebbe invocare un'ulteriore operazione su un altro componente.

L'attività esterna viene richiamata tramite il simbolo detto "rastrello" che fa riferimento ad un altro diagramma di attività.

L'uso di pin (quadrati) consente di rappresentare i parametri passati dall'attività esterna, o restituiti da essa.



10.11 Sketch di UI

Per progettare ciò che la UI dovrà visualizzare si possono fare degli sketch (bozze) delle varie finestre che compongono la UI.

In Visual Paradigm si può usare il diagramma di tipo Desktop Wireframe (non fa parte di UML).

11 Codice

Dopo aver definito strutturazione, controllo e comportamento in UML, abbiamo le informazioni necessarie per implementare un requisito funzionale (caso d'uso):

- Il **diagramma delle classi** mostra le classi da implementare (nome, attributi, metodi) e i dati da memorizzare (DB).
- I **diagrammi di attività** mostrano gli algoritmi di elaborazione da implementare, ognuno all'interno di un metodo, indicando parametri in ingresso, valore di ritorno, variabili/oggetti locali e i loro tipi.
- Il **diagramma di sequenza** mostra per ogni servizio (caso d'uso) quali oggetti sono coinvolti e l'ordine di invocazione dei metodi (detti $<<$ *operazioni* $>>$ in UML).

11.1 Ordine di implementazione

Ci sono 3 livelli, presentazione (UI), elaborazione (Gestore) e gestione dati (DB).
UI invoca Gestore che a sua volta invoca DB:

$$UI -- > Gestore -- > DB$$

Conviene procedere in questo ordine:

- Implementare una classe "prototipo" dove si creano i componenti e si invoca l'avvio del primo caso d'uso (login);
- Implementare i metodi del DB (**diagramma di attività**);
- Implementare e testare i metodi del Gestore (**diagramma di attività**);
- Implementare i metodi della UI (**sketch** della UI);
- Implementare nella UI un metodo di avvio che invoca gli altri metodi secondo l'ordine e i frame indicati nel **diagramma di sequenza**.

11.2 Diagramma delle classi: interfacce

- Arco di realizzazione:
  - collega l'interfaccia con la classe che la implementa;
  - è tratteggiato e termina con un triangolo chiuso;
  - è orientato verso l'interfaccia.
- Arco di dipendenza:
  - collega l'interfaccia con la classe che la invoca;
  - è tratteggiato e termina con un triangolo aperto;
  - è orientato verso l'interfaccia invocata.

Parte III
Pattern

Un modo per ottenere **rapidità** consiste nel riutilizzare soluzioni di progettazione utilizzate in passato per risolvere problemi analoghi.
Design pattern = soluzione progettuale generale ad un problema ricorrente.
Esistono 23 design patterns suddivisi in baseal loro scopo:

- **Creational** = soluzioni per creare oggetti
  - Abstract Factory, Builder, **Factory Method**, Prototype, **Singleton**.
- **Structural** = soluzioni per la composizione strutturale di classi e oggetti
  - Adapter, Bridge, **Composite**, **Decorator**, Facade, Flyweight, Proxy.
- **Behavioral** = soluzioni per l'interazione e la suddivisione delle responsabilità tra classi e oggetti
  - Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, **Observer**, **State**, Strategy, Template Method, Visitor.

Diagramma delle classi: abstract e interface

- $<<$ *abstract* $>>$ è una classe che non si può istanziare. Le classi figlie ereditano attributi e operazioni. L'arco di generalizzazione va dalla classe figlia alla classe *abstract*;
- $<<$ *interface* $>>$ è una classe che definisce i prototipi di un insieme di operazioni. L'arco di realizzazione va dalla classe che implementa le operazioni alla classe *interface*.

Navigabilità delle associazioni Se l'arco di associazione è orientato l'associazione è navigabile solo in una direzione.

- A $\rightarrow$ B significa che date le informazioni in A posso ottenere le informazioni in B, ma non viceversa.

Se l'arco non è orientato l'associazione è navigabile in entrambe le direzioni.

12 Pattern creazionali

12.1 Singleton

Consente di definire classi per cui può esistere una e una sola istanza. Viene utilizzato quando l'esistenza di istanze multiple può portare a problemi:

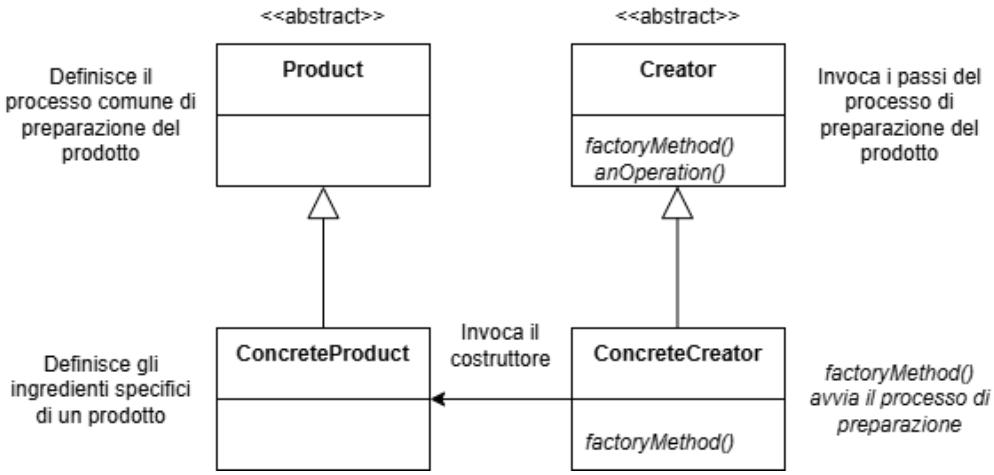
- comportamento scorretto del programma, utilizzo eccessivo delle risorse, risultati non consistenti, ecc. (eg. thred pool, cache, impostazioni, device driver).

```
1  // Singleton class implementation
2  public class MyClass {
3      private static MyClass uniqueInstance;
4
5      private MyClass() {
6          // Constructor logic
7      }
8
```

```
9       public static MyClass getInstance() {
10          if (uniqueInstance == null)
11              uniqueInstance = new MyClass();
12          return uniqueInstance;
13      }
14  }
```

12.2 Factory method

Supponiamo di dover gestire gli ordini di pizze presso varie pizzerie in varie località. Il processo di preparazione della pizza è lo steso ovunque (impasto, ingredienti, cottura, taglio e confezione). I tipi di pizza sono gli stessi ovunque, gli ingredienti potrebbero variare parzialmente in base alla località.



Factory consente di definire un'interfaccia per la creazione di famiglie di oggetti correlati tra loro senza specificare la loro concreta implementazione.

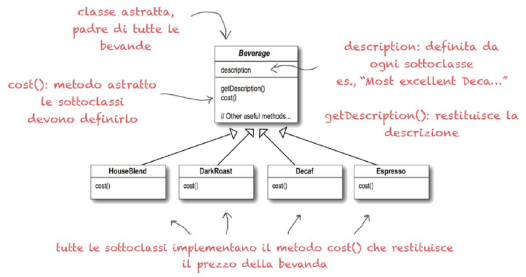
13 Pattern strutturali

13.1 Decorator

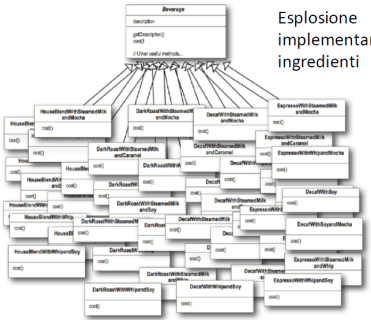
Aggiunge nuove funzionalità (responsabilità) ad un oggetto dinamicamente. Fornisce un'alternativa più flessibile del meccanismo dell'ereditarietà per estendere le funzionalità.

13.1.1 Caffetteria

Caffè in rapida espansione → necessità di avere un sistema automatizzato per effettuare gli ordini di tutte le bevande offerte.
All'inizio le bevande sono rappresentate come singole sottoclassi.



Successivamente viene aggiunta la possibilità di chiedere diversi condimenti aggiuntivi, per ogni aggiunta il prezzo della bevanda cambia, inizialmente viene proposta la seguente architettura per il sistema software delle ordinazioni.



Esplosione nel numero delle classi da implementare. Oneroso in presenza di svariati ingredienti

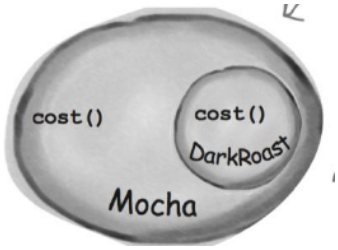
ogni sottoclasse definisce il metodo cost() in modo opportuno!

Mantenibilità difficoltosa: cosa succede se cambia il prezzo del latte? Se aggiungo un nuovo ingrediente? Se rimuovo un ingrediente?

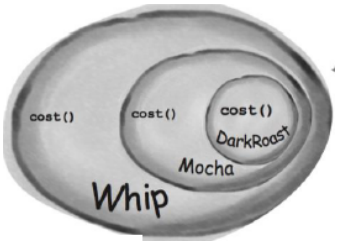
Decorator: decorazione della bevanda

1. Bevanda base. DarkRoast eredita da Beverage, quindi implementa il metodo *cost()* per calcolare il costo.

2. Aggiunta di cioccolato. L'oggetto Mocha è un DECORATOR. Incapsula un oggetto di tipo Beverage ed è esso stesso di tipo Beverage.



3. Aggiunta di panna. L'oggetto Whip è un DECORATOR. Incapsula un oggetto di tipo Beverage ed è esso stesso di tipo Beverage.

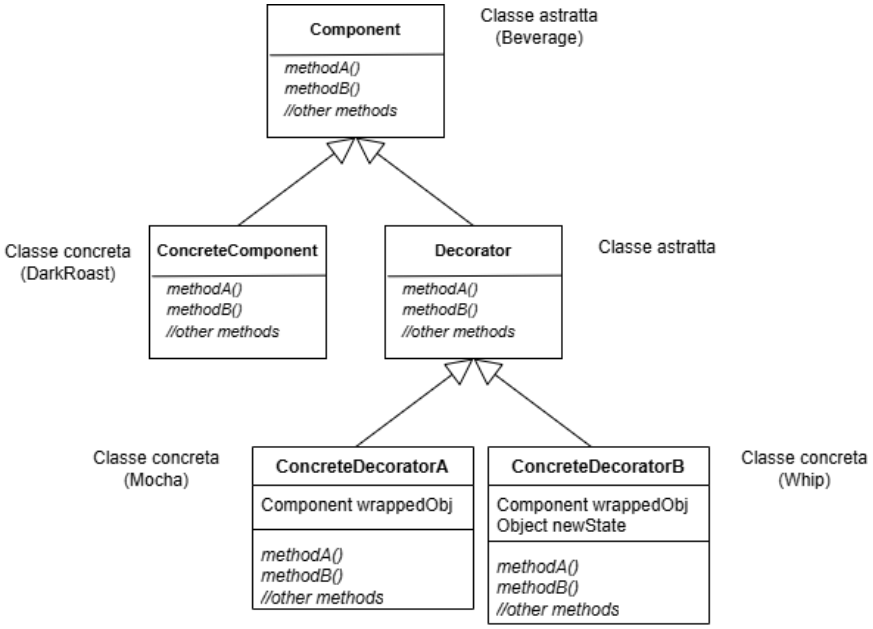


Per calcolare il costo:

1. chiamata di *cost()* su Whip;

2. chiamata di *cost()* su Mocha;
3. chiamata di *cost()* su DarkRoast;
4. DarkRoast ritorna $0,99$;
5. Mocha ritorna $0,99 + 0,20$;
6. Whip ritorna $1,19 + 0,10$.

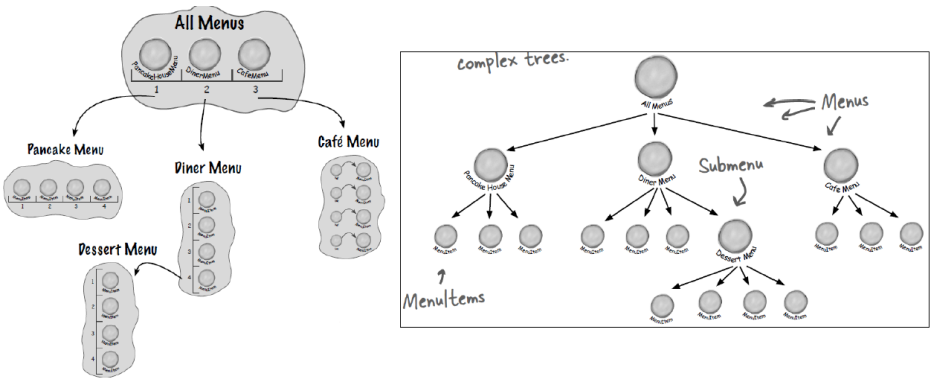
13.1.2 Pattern



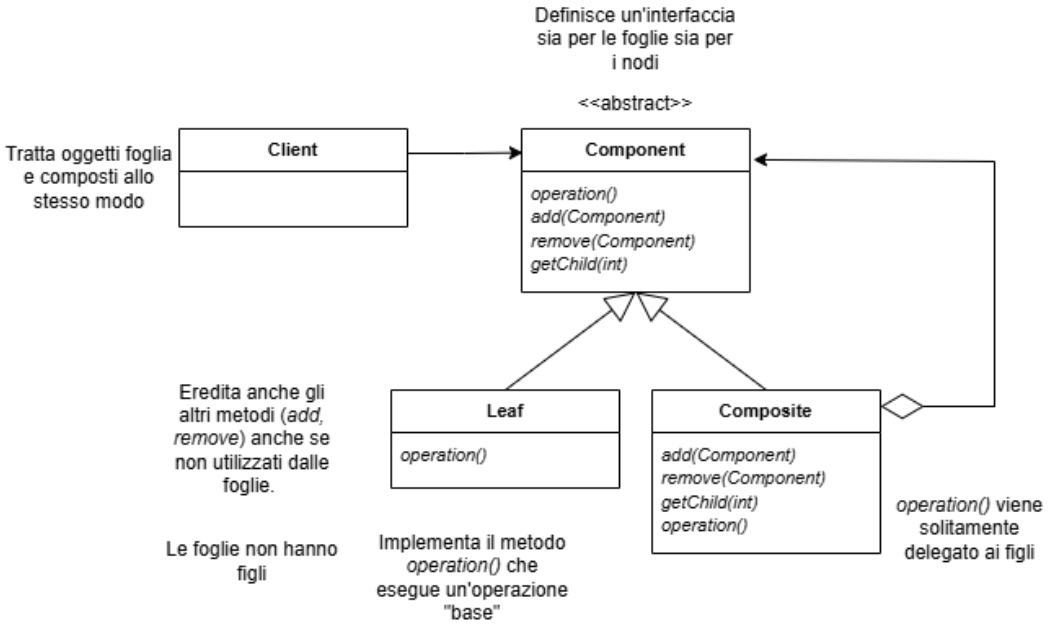
13.2 Composite

Consente di comporre oggetti in strutture ad **albero** per rappresentare delle gerarchie. Permette, inoltre, al codice client di trattare in modo uniforme oggetti individuali e oggetto composti.

13.2.1 Menù e sotto menù



13.2.2 Pattern



14 Pattern comportamentali

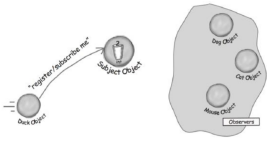
14.1 Observer

Realizza una relazione di **dipendenza** tra oggetti di tipo **uno** (Observable) a **molti** (Observer) in modo che quando l'oggetto Observable cambia il proprio stato, tutti gli oggetti dipendenti vengano **notificati** e **aggiornati**.

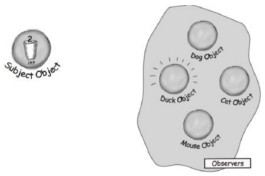
Consente di avere **dinamicità** nell'insieme degli **oggetti dipendenti**.

14.1.1 Funzionamento

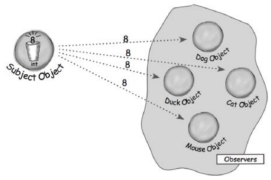
- **register/subscribe** = l'oggetto *Duck* chiede all'oggetto *Subject* di diventare un **Observer**.



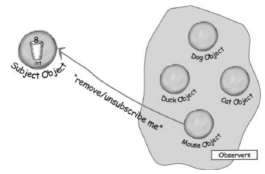
- **registered** = l'oggetto *Duck* diventa ufficialmente un oggetto **Observer**.



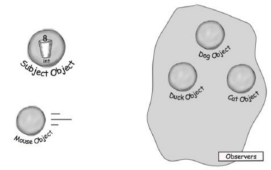
- **update** = l'oggetto *Subject* cambia il proprio **stato** e lo comunica a tutti gli **Observer**.



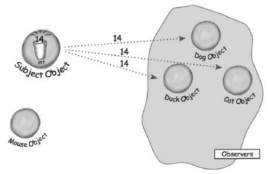
- **remove/unsubscribe** = l'oggetto *Mouse* richiede all'oggeto *Subject* di essere **rimosso** dall'insieme degli **Observer**.



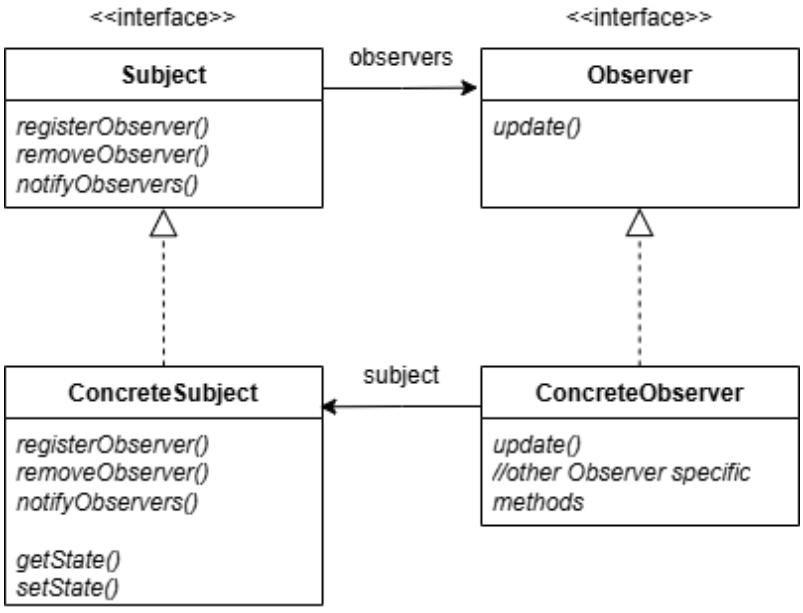
- **removed** = l'oggetto *Mouse* viene ufficialmente **rimosso** dall'insieme degli **Observer**.



- **update** = l'oggetto *Subject* cambia nuovamente stato e lo **comunica** a tutti gli oggetti **Observer**.



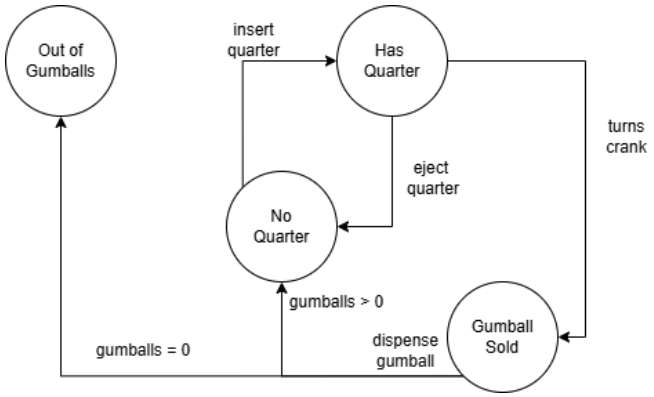
14.1.2 Pattern



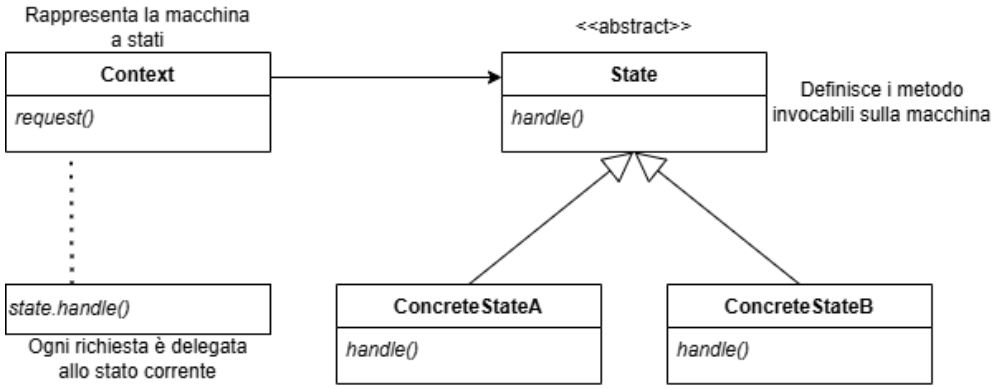
14.2 State

Il pattern State consente ad un **oggetto** di **cambiare il proprio comportamento** quando il proprio **stato interno cambia**.

14.2.1 Gumball machine



14.2.2 Pattern



Parte IV
Strumenti

15 Git

15.1 VMS (Version Management System)

Strumenti CASE per gestire le versioni: CVS, RCS, SVN, Git, ecc.
Una **repository** contiene le versioni di un progetto:

- codice sorgente;
- documentazione;
- file dati;
- file di configurazione;
- ...

Per modificare il progetto si estrae una sua versione dal repository (**check-out**) e se ne salva una copia (**working copy**) in una directory di lavoro.
Quando il lavoro è completo, i file modificati vengono rimessi nel repository (**check-in** o **commit**) creando automaticamente una nuova versione del progetto.
Ogni modifica genera una nuova versione e conserva la versione precedente.

15.1.1 Funzioni

Version Control possibilità di creare una nuova versione o recuperare una qualunque versione precedente. La nuova versione non sovrascrive quella precedente. Le versioni sono numerate automaticamente.

Condivisione consentire la condivisione e l'accesso contemporaneo a più sviluppatori aiutando a gestire i conflitti.

Tracciabilità tracciare i contributi degli sviluppatori (informazioni sugli autori di ogni versione).

Registro storico delle modifiche tutte le modifiche al progetto sono registrate e possono essere elencate.

Gestione della memorizzazione le varie versioni occupano spazio e si differenziano solo per le modifiche. Per ridurre lo spazio occupato le versioni sono descritte dalle loro differenze rispetto ad una versione master. I file possono essere compressi.

15.2 RCS: Δ based VMS

RCS = Revision COntrol System, primo tool di VMS in Unix.
Quando una nuova versione viene creata, quella precedente viene cancellata e sostituita da un Δ. Δ indica le differenze tra una versione e quella precedente tramite comandi di editing.

RCS memorizza l'ultima versione (master) del codice e una sequenza di Δ, riducendo lo spazio occupato.

RCS applica i Δ all'ultima versione per ottenere la versione richiesta.

15.3 Git e repository

Git è uno strumento per la condivisione dei file e per la gestione delle loro versioni.

Git richiede l'uso di un repository accessibile dagli sviluppatori, dove memorizzare i file e tener traccia delle modifiche.

- **Repository locale e privato:** GitLab;
- **Repository commerciale:** BitBucket;
- **Repository pubblico:** GitHub.

15.3.1 Configurazione

Per impostare username, email, editor:

```
git config --global user.name "John Smith"
git config --global user.email matricola@studenti.uniupo.it
git config --global core.editor vim
```

File di configurazione di Git : **/.gitconfig**.
Per vedere le impostazioni:

```
    git config --list
```

Per vedere una particolare impostazione:

```
    git config user.name
    git config user.email
    git config core.editor
```

15.3.2 Help

Tre modi per avere informazioni su un comando:

```
    git help <verb>
    git <verb> --help
    man git-<verb>
```

15.3.3 Workflow

Stati di un file:

- **Modified** = file nella working directory, modificato ma non contrassegnato per il commit;
- **Staged** = file contrassegnato per il commit (incluso nella "staging area" o "stage").

```
        git add nome_file
```

  - Staging area = zona dove si indicano i file da includere nel prossimo commit (versione).

- **Committed** = file nel commit aggiunto alla history del repository locale. Viene creato un commit (versione) con i file nello stage (il messaggio descrive il commit). Svuota lo stage.

```
        git commit -m "messaggio"
```

- History = serie dei commit effettuati.

- **Pushed** = file nel commit aggiunto alla history del repository locale e remoto.

```
1        git push origin main (master)
```

  - origin è il riferimento al repository remoto;
  - main (master) è il riferimento al branch principale.

Rimuovere file dallo stage rimuove il file specificato dallo stage lasciando inalterata la working directory.

```
1    git reset <file>
```

Rimuove tutti i file dallo stage lasciando inalterata la working directory.

```
1    git reset
```

Rimuovere file nel repository remoto Per cancellare file dal repository remoto.

```
1    git rm <file>
```

Rinominare file nel repository remoto Per rinominare file nel repository remoto.

```
1    git mv <name><new_name>
```

Scaricare dal repository remoto i commit degli altri sviluppatori Scarica nella working directory tutti i file modificati da altri utenti nel frattempo. Aggiorna il repository locale al commit più recente.

In caso di **conflitti**, git indica su quali file bisogna intervenire. Un **conflitto** avviene quando due sviluppatori effettuano modifiche allo stesso file e git non riesce a "fondere" le modifiche.

```
1    git pull
```

Conoscere lo stato Mostra le informazioni sui file nella working directory e nello stage (file fuori e dentro lo stage, file fuori e dentro il commit, ecc.).

```
1    git status
```

Mostra l'elenco dei commit nella history.

```
1    git log
```

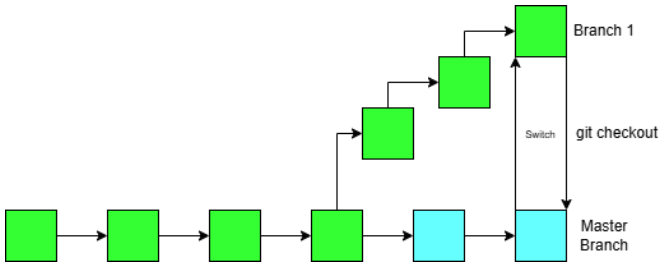
git blame Per ogni riga di un file specifica:

- chi ha introdotto la riga;
- quando è stata introdotta la versione attuale.

```
1    git blame <nomefile>
```

15.3.4 Branch

Linea parallela alla linea main (master) della history.



Su un branch secondario è possibile eseguire le stesse operazioni che si fanno sul main branch.

Creare un branch

```
1    git branch nome_branch
```

Spostarsi su un branch

```
1    git checkout nome_branch
```

Differenze tra due branch

```
1    git diff <branch1><branch2>
```

Fondere un branch secondario al main (master) branch
Per spostarsi sul main (master branch).

```
1    git checkout main
```

Fonde le modifiche fatte sul branch secondario con le modifiche fatte sul main branch. Possono verificarsi **conflitti**, così come quando si esegue **push** di un commit.

```
1    git merge nome_branch
```

Cancella il branch secondario localmente.

```
1    git branch -d nome_branch
```

Mostra che la history remota non è aggiornata rispetto a quella locale.

```
1    git status
```

Aggiorna la history remota.

```
1    git push origin main
```

Altri comandi

Mostra i branch locali.

```
git branch
```

Mostra i branch remoti.

```
git branch -r
```

Mostra i branch locali e remoti.

```
git branch -a
```

Cancella un branch dal repository remoto.

```
git push origin --delete nome_branch
```

15.3.5 Altri comandi

Ripristina il contenuto del file allo stato dell'ultimo commit.

```
git restore <file>
```

Consente di portare l'intera working directory alla situazione del commit specificato.

```
git checkout <commit>
```

Crea un nuovo commit rimuovendo i cambiamenti fatti nel commit indicato nel comando.

```
git revert <commit>
```

Non rimuove commit dalla history.

```
git revert HEAD
```

15.4 Algoritmo diff3

Individua differenze e conflitti tra due versione dello stesso file rispetto alla versione originale.

```
diff3 versione1 originale versione2
```

Opzioni

```
-X //mostra i conflitti
-m //mostra il merge risultante
```

16 Swing

Classi di Java dedicate alla costruzione di interfacce utente grafiche (GUI). Swing eredita o estende gli elementi di AWT (Abstract Window Toolkit).

16.1 Elementi

16.1.1 Elementi di base

- JLabel(testo e/o immagine);
- JTextField(casella di testo editabile);
- JComboBox (menu a tendina);
- JCheckBox (casella di spunta);
- JRadioButton (opzione selezionabile);
- JList (elenco di voci);
- JTable (tabella composta da righe e colonne).

16.1.2 Pannelli

Composti da elementi di base (e pannelli annidati).

- JPanel (non scrollabile);
- JScrollPane (scrollabile).

16.1.3 Contenitori

Finestre che contengono e visualizzano un pannello.

- JFrame: generica finestra;
- JDialog: generica finestra di dialogo;
- JOptionPane: finestra di dialogo con pulsanti predefiniti o specifici.

16.2 JOptionPane

É una finestra di dialogo (contenitore) che contiene un pannello composto da vari elementi di base.
I pulsanti (e le icone) vengono creati automaticamente in base al tipo di finestra:

- *showMessageDialog()* = pulsante OK;
- *showConfirmDialog()* = pulsanti OK-Cancel, Yes-No, Yes-No-Cancel;
- *showInputDialog()* = menu a tendina con le scelte e pulsanti Ok-Cancel;
- *showOptionDialog()* = pulsanti specifici.

16.3 Costruire una GUI

1. Definire alcuni elementi di base;
2. Definire un pannello con un certo layout;
3. Inserire gli elementi di base nel pannello;
4. Inserire e visualizzare il pannello in una finestra (contenitore);
5. Prelevare l'input dall'utente;
6. Gestire i click sui pulsanti.

16.4 JLabel

Mostra un testo non modificabile dall'utente e/o un'immagine.

Costruttori

```
label = new JLabel(stringa);
label = new JLabel(new ImangeIcon(nomeFile));
label = new JLabel(stringa, new ImageIcon(nomeFile));
```

Metodi

```
    setText(stringa);
    getText();
    setIcon(new ImageIcon(nomeFile));
    getIcon();
```

16.5 JTextField

Mostra un casella di testo dove l'utente può scrivere.

Costruttori

```
    field = new JTextField(dimensione);
    field = new JTextField(testoPredefinito);
    field = new JTextField(testPredefinito, dimensione);
```

Metodi

```
    getText();
    setText(stringa);
    setBackground(Color.COLORE);
```

Per la scrittura delle password si può usare **JPasswordField**. Il metodo *setEchoChar(carattere)* imposta il simbolo da usare per nascondere i caratteri della password.

16.6 JOptionPane.showMessageDialog()

Parametri

- parent = null se l'OptionPane non è collegato ad un JFrame;
- pannello (oppure singolo elemento o stringa) da visualizzare nelle finestre;
- titolo della finestra (stringa);
- tipo di icona predefinita (intero):
  - errore, informazione, avvertimento, domanda, messaggio.
- icona specifica.

Valore di ritorno

- nessuno (void).

16.7 JOptionPane.showConfirmDialog()

Parametri

- parent = null se l'OptionPane non è collegato ad un JFrame;
- pannello (oppure singolo elemento o stringa) da visualizzare nella finestra;
- titolo della finestra (stringa);
- tipo di pulsanti (intero):
  - Yes, No
  - Yes, No, Cancel
  - Ok, Cancel
- tipo di icona predefinita (intero):
  - errore, informazione, avvertimento, domanda, messaggio.
- icona specifica.

Valore di ritorno

- il pulsante cliccato dall'utente (intero).

16.8 JOptionPane.showInputDialog()

Parametri

- parent = null se l'OptionPane non è collegato ad un JFrame;
- pannello (oppure singolo elemento o stringa) da visualizzare nella finestra;
- titolo della finestra (stringa);
- tipo di icona predefinita (intero):
  - errore, informazione, avvertimento, domanda, messaggio.
- icona specifica;
- possibili scelte (array);
- scelta di default.

Valore di ritorno

- Valore inserito dall'utente.

16.9 JOptionPane.showOptionDialog()

Parametri

- parent = null se l'OptionPane non è collegato ad un JFrame;
- pannello (oppure singolo elemento o stringa) da visualizzare nella finestra;
- titolo della finestra (stringa);
- tipo di pulsanti predefiniti (intero):
  - Default
  - Yes, No
  - Yes, No, Cancel
  - Ok, Cancel
- tipo di icona predefinita (intero):
  - errore, informazione, avvertimento, domanda, messaggio.
- icona specifica;
- pulsanti specifici (array);
- scelta di default.

Valore di ritorno

- pulsante cliccato dall'utente (intero).

16.10 JOptionPane

16.10.1 Vantaggi

Aspetti gestiti in modo automatico:

- creazione e posizione dei pulsanti;
- gestione dei click sui pulsanti;
- posizionamento della finestra al centro dello schermo;
- adattamento delle dimensioni della finestra in base al contenuto;
- apertura e chiusura delle finestre.

16.10.2 Svantaggi

- non si può abbassare o allargare la finestra;
- non si possono avere più finestre aperte allo stesso tempo;
- l'aspetto grafico dei pulsanti è quello standard;
- i pulsanti sono posizionati solo in basso.

16.11 Tipi di layout (disposizione) per i pannelli

FlowLayout Dispone gli elementi in orizzontale, da sinistra verso destra. Layout di default se non viene definito.

BoxLayout Dispone gli elementi in verticale, dall'alto verso il basso.

BorderLayout Dispone gli elementi in posizioni predefinite (alto, centro, basso, sinistra, destra).

GridLayout Dispone gli elementi secondo una matrice (grid) riempendo una riga per volta. Un elemento per casella.

GridBagLayout Dispone gli elementi secondo una matrice. Un elemento può occupare una o più caselle.

16.12 JList

Mostra una lista di elementi grafici o stringhe. Se la lista è lunga si può mettere all'interno di un JScrollPane (pannello scorrevole), altrimenti in un JPanel.

Costruttori

```
1     list = new JList<tipo>();
2     list = new JList(vettore);
```

Metodi

```
1     setSelectionMode(DefaultListSelectionModel.SINGLE_SELCTION) //rende
         selezionabile un solo elemento della lista
2     setListData(vettore) //assegna il contenuto della lista
3     getSelectedIndex() //ritorna l'elemtno selezionato
4     setSelectedIndex() //imposta l'elemento selezionato
```

16.13 JCheckBox

JCheckBox mostra una singola voce spuntabile. Vari JCheckBox possono essere raggruppati in un ButtonGroup (in modo che un solo JCheckBox sia spuntabile).

Costruttori

```
1     check = new JCheckBox(testo);
2     check = new JCheckBox(testo, stato);
3     group = new ButtonGroup();
```

Metodi

```
1     add(check) //aggiunge una JCheckBox ad un ButtonGroup
2     clearSelection() //rimuove la spunta dal ButtonGroup
3     isSelected() //indica se la JCheckBox e' spuntata o no (boolean)
4     setSelected(stato) //imposta lo stato (spuntata o no) della JCheckBox
```

16.14 JRadioButton

JRadioButton mostra una singola voce selezionabile. Vari JRadioButton possono essere raggruppati in un ButtonGroup (in questo modo un solo JRadioButton può essere selezionato).

Costruttori

```
    check = new JRadioButton(testo);
    check = new JRadioButton(testo, stato);
    group = new ButtonGroup();
```

Metodi

```
    add(check) //aggiunge un JRadioButton ad un ButtonGroup
    clearSelection() //rimuove la selezione dal ButtonGroup
    isSelected() //indica se il JRadioButton e' selezionato o no (boolean)
    setSelected(stato) //imposta lo stato (selezionato o no) del JRadioButton
```

16.15 JComboBox

Mostra un menù a tendina composto da varie voci.

Costruttori

```
    combo = new JComboBox();
    combo = new JComboBox(vettore);
```

Metodi

```
    addItem(testo) //aggiunge una voce al menu
    getSelectedIndex() //indica l'indice della voce selezionata
    setSelectedIndex(indice) //imposta la voce selezionata traimite l'indice
       (-1 per non selezionare nessuna voce)
    getSelectedItem() //indica la voce selezionata
    setSelectedItem() //imposta la voce selezionata
```

16.16 JTable

Mostra una tabella (matrice) organizzata in header, righe e colonne. Se la tabella è grande si può mettere all'interno di un JScrollPane (pannello scorrevole), altrimenti in un JPanel. Con JScrollPane gli header (tiotoli delle colonne) sono visualizzati automaticamente.

Costruttori

```
    table = new JTable();
    table = new JTabel(righe, colonne);
    table = new JTabel(matrice, headers);
```

Metodi

```
getColumnModel().getColumn(j).setHeaderValue(testo) //imposta l'header della colonna con indice j
getColumnModel().getColumn(j).setPreferredWidth(larghezza) //imposta la larghezza della colonna con indice j
setValueAt(valore, i, j) //assegna un valore alla casella con indici i e j
setAutoResizeMode(JTable.AUTO_RESIZE_OFF) //per far apparire la scrollbar orizzontale
setSelectionMode(DefaultListSelectionModel.SINGLE_SELECTION) //rende selezionabile un sola riga della tabella
setPreferredSize(new Dimension(larghezza, altezza)) //imposta la dimensione della tabella
getSelectedRow() //indica la riga selezionata
setRowSelectionInterval(inizio, fine) //imposta le righe selezionate tramite i loro indici
```

16.17 ToolTip

É possibile associare ad ogni elemento grafico un suggerimento (ToolTip) da visualizzare quando il puntatore del mouse è sopra l'elemento.

```
elemento.SetToolTipText(testo)
```

17 SQLite

SQLite è un DBMS **relazionale**, **serverless**, perché non richiede un processo server in esecuzione, ma è integrato nell'applicazione che accede al database.

17.1 Classi Java per SQLite

Connection, DriverManager per aprire la connessione con il database. Si basano su JDBC (Java DataBase Connectivity).

Statement per eseguire dei comandi SQL:

- *executeUpdate()* per modificare i record nel database:
  - *INSERT* (aggiunta)
  - *UPDATE* (modifica)
  - *DELETE* (cancellazione)
- *executeQuery()* per recuperare record dal database.

ResultSet per contenere il risultato di una query (insieme di record).

17.2 ResultSet → ArrayList

Quando si chiude la connessione con il database, il contenuto del ResultSet viene perso. Si consiglia di non passare oggetti di classe ResultSet a metodi o altri oggetti.

Si consiglia di convertire ResultSet in un altro formato. Per esempio, un ArrayList di Hashmap:

- HashMap = struttura composta da vari campi, ciascuno identificato da una certa etichetta (stringa). Utile per contenere un singolo record. Ogni campo è identificato dal nome dell'attributo.
- ArrayList = contiene una lista di record, ciascuno identificato da un certo indice (int).

ResultMetaData classe per contenere gli attributi e i relativi tipi della tabella interrogata.
Per ogni record nel ResultSet:

- si crea una HashMap;
- si copia nella HashMap ogni attributo del record (nome attributo e valore) in base a ResultMetaData;
- si aggiunge la HashMap all'ArrayList.

17.3 Accesso ad un ArrayList di HashMap

ArrayList di Hashmap:

```
1    ArrayList<HashMap<String, Object>> list;
```

Accesso all'attributo di un elemento:

```
1    (Tipo)list.get(indice).get(nomeAttributo);
```

18 Gradle

Gradle permette di:

- Compilare;
- Assembleare;
- Testare;
- Eseguire.

18.1 Init

Creare una cartella di lavoro.
Inizializzare la cartella con il comando

```
1    gradle init --type java-application
2    //oppure
3    gradle init
4    //scegliendo il tipo di progetto (applciazione), il linguaggio (Java),
        strumento di test (JUnit), nome del progetto
```

Init genera:

- lo script (wrapper) chiamato *gradlew* (o *gradlew.bat*) che servirà per le operazioni successive;
- i file di configurazione *setting.gradle* e *build.gradle*;
- le cartelle per i sorgenti e per i test-case;
- esempi nelle due cartelle.

18.2 File di configurazione

setting.gradle contiene il nome del progetto. *rootProject.name = '...'*

build.gradle contiene il nome della classe principale. *MainClassName = '...'*

Per usare standard input nel codice, bisogna aggiungere:

```
run{standardInput = System.in}
```

Per usare SQLite bisogna aggiungere:

```
dependencies {implementation 'org.xerial:sqlite-jdbc:3.30.1'}
```

18.3 Comandi

Mostra i possibili comandi ed opzioni.

```
gradlew tasks
```

Compila (e assembla) il codice sorgente (i file eseguibili si trovano nella cartella *build*).

```
gradlew assemble
```

Compila ed esegue i test-case. L'esito dei test è mostrato nel file *build/reports/tests/test/index.html*.

```
gradlew build
```

gradlew run esegue

```
gradlew run --console plain //esegue senza messaggi di controllo sul
    terminale
gradlew run --args="..." //riceve dei parametri da riga di comando (
    assegnati al vettore args del metodo main)
```

18.4 Wrapper

gradlew (o *gradlew.bat*) è lo script necessario per compilare (assemble), testare (build), eseguire (run), ecc. Si trova nella cartella di lavoro.
Può essere generato in due modi:

- Creando un nuovo progetto (*gradle init*). In questo caso si creano anche i file di configurazione *build.gradle* e *setting.gradle*.
- Accedendo ad un progetto esistente. In questo caso *build.gradle* e *setting.gradle* esistono già. Il wrapper deve essere creato con il comando *gradle wrapper*.

19 RMI (Remote Method Invocation)

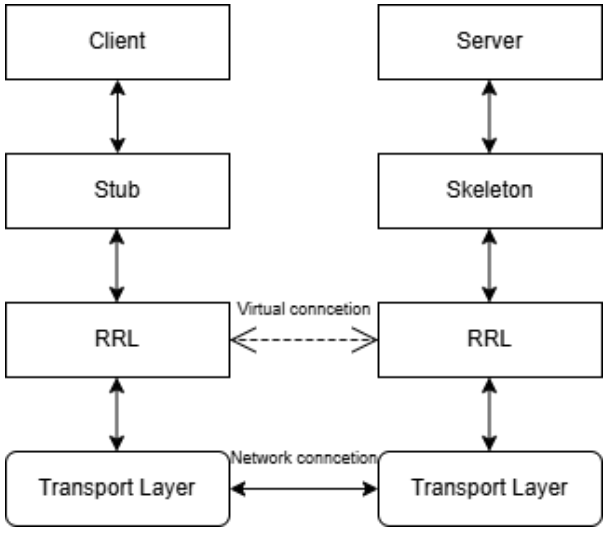
Protocollo che consente ad un oggetto che si trova in una JVM di invocare i metodi di un oggetto che si trova in un'altra JVM.

Le JVM possono trovarsi sulla stessa macchina fisica o su macchine fisiche diverse. Nel secondo caso, le macchine fisiche possono trovarsi nella stessa LAN o essere tra loro remote.

RMI prevede un **oggetto-server** e un **oggetto-client**. Ogni JVM ha un registro con i riferimento agli oggetti-server in esecuzione su di essa. Un oggetto-client può individuare l'oggetto server tramite tale registro.

A livello di codice le invocazioni di metodi remoti avvengono come se gli oggetti fossero locali.

19.1 Protocollo



Livello di trasporto gestisce la connessione tra le macchine fisiche.

RRL (Remote Reference Layer) gestisce la connessione virtuale tra le JVM.

Stub rappresentazione dell'oggetto-server usata dall'oggetto-client per invocarlo. L'oggetto-client invoca l'oggetto-server come se fosse locale.

Skeleton gestisce le invocazione all'oggetto-server che arrivano da remoto. L'oggetto-server risponde alle invocazioni come se l'oggetto-client fosse locale.

1. Il client invoca un metodo dell'oggetto remoto;
2. Marshalling:
   - l'invocazione è ricevuta dallo stub che lo passa a RRL-client;
   - RRL-client crea un messaggio contenente il riferimento all'oggetto-server, il nome del metodo e i valori dei parametri;
   - RRL-client invia il messaggio a RRL-server tramite il Livello di trasporto.
3. Unmarshalling:
   - RRL-server estrae metodo e parametri dal messaggio e li passa allo skeleton destinatario che invoca l'oggetto-server a cui fa riferimento;
   - l'oggetto-server esegue il metodo.
4. Il valore di ritorno è passato all'oggetto-client facendo il percorso inverso.

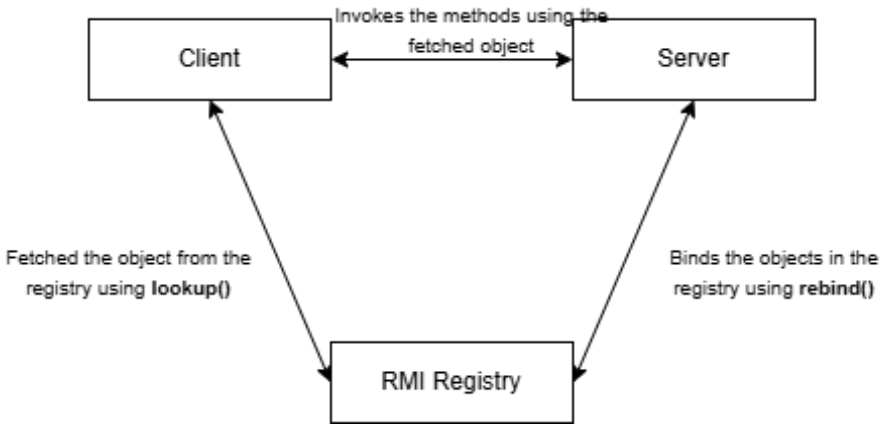
19.2 Registro RMI

Su ogni JVM si può creare un registro con i riferimenti agli oggetti-server presenti.

Quando si avvia un oggetto-server, questo è da aggiungere al registro (bind) assegnandogli un certo nome (bind name).

L'oggetto-client ha bisogno di un riferimento all'oggetto-server:

- si connette al registro sulla JVM-server;
- ottiene il riferimento all'oggetto-server (lookup) tramite il corrispondente bind name;
- invoca i metodi dell'oggetto-server.



19.3 Esempio

19.3.1 Costruzione dell'oggetto-server

1. interfaccia dell'oggetto;
2. classe che implementa l'interfaccia;
3. l'oggetto è istanziato, registrato (bind) e pronto a ricevere le invocazioni. Il registro è avviato su una porta di default, si può indicare una porta specifica. L'oggetto-client individua il registro remoto e l'oggetto-server (lookup). Quindi invoca *printMsg()* sull'oggetto-server.

19.3.2 Esecuzione

Compilare tutti i sorgenti.

Lato server

- Aprire un terminale;
- Avviare il registro con il comando rmiregistry (si può avviare anche nel codice);
- Avviare l'oggetto-server: *java Server*.

Lato client

- Aprire un terminale;
- Avviare l'oggetto-server: *java Client*.

L'oggetto-client invoca *printMsg()* sull'oggetto-server. L'effetto è la stampa del messaggio "This is an example RMI program" sul terminale lato server.