

Appunti per l'Esame di Stato
Ingegnere Junior - Settore Informazione

Paolo Pietrelli

19 luglio 2025

Indice

| | | |
|----------|---|-----------|
| 1 | Sistemi Operativi | 1 |
| 1.1 | Struttura e Organizzazione del Sistema Operativo | 1 |
| 1.1.1 | Componenti Principali | 1 |
| 1.1.2 | Modelli di Sistemi Operativi | 2 |
| 1.2 | Scheduling della CPU | 2 |
| 1.2.1 | Principali Problematiche dello Scheduling | 2 |
| 1.2.2 | Esempi di Algoritmi di Scheduling | 3 |
| 2 | Basi di Dati | 5 |
| 2.1 | Modello Concettuale Entità-Relazione (ER) | 5 |
| 2.1.1 | Componenti Principali del Modello ER | 5 |
| 2.2 | Progettazione Logica: Normalizzazione e Forme Normali | 6 |
| 2.2.1 | Obiettivi della Normalizzazione | 6 |
| 2.2.2 | Principali Forme Normali | 7 |
| 2.2.3 | Linguaggio SQL | 7 |
| 3 | Reti di Calcolatori | 11 |
| 3.1 | Modello di Comunicazione Client/Server | 11 |
| 3.1.1 | Funzionamento del Modello Client/Server | 11 |
| 3.1.2 | Vantaggi e Svantaggi del Modello Client/Server | 11 |
| 3.2 | Protocollo HTTP | 12 |
| 3.2.1 | Funzionamento e Applicazione in HTTP | 12 |
| 3.2.2 | Connessioni Stateless e Stateful | 12 |
| 3.2.3 | Metodi per Gestire la Persistenza dello Stato in HTTP | 13 |
| 3.3 | Standard ISO/OSI | 13 |
| 3.3.1 | Struttura a Sette Strati del Modello OSI | 13 |
| 3.4 | Livello di Trasporto (TCP vs UDP) | 14 |
| 3.4.1 | TCP (Transmission Control Protocol) | 14 |
| 3.4.2 | UDP (User Datagram Protocol) | 15 |
| 4 | Programmazione Orientata agli Oggetti (e Fondamenti) | 17 |
| 4.1 | Concetti Base della Programmazione Orientata agli Oggetti (POO) | 17 |
| 4.2 | Strutture Dati Astratte (ADT) e Fondamentali di Programmazione | 18 |
| 4.2.1 | Abstract Data Type (ADT) | 18 |
| 4.2.2 | Passaggio di Parametri nelle Chiamate di Funzione | 18 |
| 4.2.3 | Overloading di Funzioni (o Metodi) | 19 |
| 4.3 | Linguaggi Compilati e Linguaggi Interpretati | 19 |
| 4.3.1 | Linguaggi Compilati | 20 |
| 4.3.2 | Linguaggi Interpretati | 20 |
| 4.4 | Algoritmi e Complessità Computazionale | 21 |
| 4.4.1 | Complessità Temporale | 21 |

| | | |
|----------|---|-----------|
| 4.4.2 | Complessità Spaziale | 21 |
| 5 | Progettazione del Software | 23 |
| 5.1 | Analisi dei Requisiti | 23 |
| 5.1.1 | Tipi di Requisiti | 23 |
| 5.1.2 | Diagrammi di Casi d'Uso UML (Use Case Diagrams) | 24 |
| 5.2 | Progettazione dell'Architettura del Sistema Software | 24 |
| 5.2.1 | Design Patterns (Pattern Architetturali e di Progettazione) | 24 |
| 5.2.2 | Diagrammi UML di Classi e di Deployment | 25 |
| 5.3 | Implementazione e Diagrammi di Sequenza UML | 26 |

Elenco delle figure

Listings

4.1 Esempio di Function Overloading 19

Capitolo 1

Sistemi Operativi

Un **sistema operativo (SO)** è un software di sistema che gestisce le risorse hardware e software di un computer e fornisce servizi comuni per i programmi del computer e per l'utente. È l'interfaccia tra l'hardware e l'utente/applicazioni. La sua importanza risiede nell'astrazione dell'hardware, nella gestione efficiente delle risorse e nell'esecuzione controllata dei programmi.

1.1 Struttura e Organizzazione del Sistema Operativo

Un sistema operativo è un'entità complessa, ma può essere scomposto in componenti modulari che cooperano per fornire un ambiente funzionale per l'esecuzione dei programmi.

1.1.1 Componenti Principali

I principali componenti di un sistema operativo includono:

- **Kernel:** Il cuore del SO, responsabile della gestione dei processi (creazione, scheduling, terminazione, comunicazione interprocesso), della memoria (allocazione, protezione, gestione della memoria virtuale), dei file system (gestione dei file e delle directory, allocazione dello spazio su disco) e dell'I/O (gestione dei dispositivi di input/output, driver).
- **Gestore dei Processi (Process Management):** Si occupa della creazione, terminazione, sospensione e ripristino dei processi, e della gestione dei loro stati (pronto, in esecuzione, in attesa).
- **Gestore della Memoria (Memory Management):** Responsabile dell'allocazione e deallocazione della memoria ai processi, della gestione della memoria virtuale (paginazione, segmentazione) e della protezione della memoria per evitare interferenze tra i processi.
- **File System Management:** Organizza e gestisce i dati su dispositivi di archiviazione, controllando l'accesso e la protezione dei file e delle directory.
- **Gestore I/O (I/O Management):** Fornisce un'interfaccia standardizzata per interagire con i dispositivi hardware (stampanti, tastiere, dischi) tramite driver specifici.
- **Network Management:** Gestisce le comunicazioni di rete e i protocolli di comunicazione.
- **Security and Protection:** Implementa meccanismi per proteggere le risorse del sistema e i dati degli utenti da accessi non autorizzati o malfunzionamenti.

- **Interfaccia Utente (User Interface):** Può essere una GUI (Graphical User Interface) con elementi visivi o una CLI (Command Line Interface) basata su testo, permettendo all'utente di interagire con il sistema.

1.1.2 Modelli di Sistemi Operativi

I sistemi operativi possono essere strutturati secondo diversi modelli architetturali:

- **Monolitici:** Tutti i servizi del SO risiedono nello stesso spazio di indirizzamento (kernel space).
 - **Vantaggi:** Alta performance grazie al minimo overhead di comunicazione.
 - **Svantaggi:** Difficili da debuggare, poco flessibili, un crash di un componente può bloccare l'intero sistema.
 - **Esempio:** Linux, Unix (tradizionali).
- **Layered (a Strati):** Il SO è diviso in strati, ognuno dei quali offre servizi allo strato superiore e utilizza servizi dallo strato inferiore.
 - **Vantaggi:** Modularità, facilità di debug e manutenzione.
 - **Svantaggi:** Performance ridotte a causa dell'overhead di comunicazione tra strati.
 - **Esempio:** THE (Dijkstra).
- **Microkernel:** Solo i servizi essenziali (gestione processi, gestione memoria base, comunicazione interprocesso) risiedono nel kernel (microkernel). Altri servizi (file system, driver, network) sono implementati come processi utente (server).
 - **Vantaggi:** Modularità, robustezza (un crash di un server non blocca il sistema), flessibilità.
 - **Svantaggi:** Performance potenzialmente più basse a causa di più cambi di contesto.
 - **Esempio:** Mach (base per macOS), QNX.
- **Modulari (o Ibridi):** Un approccio intermedio che combina le migliori caratteristiche dei modelli monolitici e microkernel. Permettono il caricamento dinamico dei moduli kernel (es. driver) senza richiedere un riavvio completo del sistema.
 - **Esempio:** Versioni moderne di Linux, Windows.

1.2 Scheduling della CPU

Lo **scheduling della CPU** è l'attività di selezionare quale processo, tra quelli pronti per l'esecuzione, deve essere assegnato alla CPU in un dato momento. Ha un impatto cruciale sulle performance complessive del sistema.

1.2.1 Principali Problematiche dello Scheduling

Lo scheduling deve affrontare diverse sfide e problematiche per bilanciare l'efficienza e l'equità:

- **Ottimizzazione degli obiettivi:** Bilanciare metriche contrastanti come massimizzare il throughput (numero di processi completati per unità di tempo), minimizzare il tempo di risposta (tempo tra richiesta e prima risposta), minimizzare il tempo di attesa e garantire l'equità tra i processi.

- **Contesto Switching (Cambio di Contesto):** L'overhead di tempo necessario per salvare lo stato di un processo in esecuzione e caricare lo stato del prossimo processo da eseguire. Questo tempo è "sprecato" e non contribuisce all'esecuzione del lavoro utile.
- **Starvation (Inedia):** Un processo a bassa priorità potrebbe non essere mai eseguito se processi a priorità più alta arrivano continuamente e monopolizzano la CPU.
- **Deadlock:** Sebbene sia una problematica più ampia della gestione della concorrenza, situazioni di deadlock possono emergere in sistemi con scheduling se le risorse non sono gestite correttamente, bloccando indefinitamente i processi.
- **Dipendenza dall'I/O:** Processi che trascorrono molto tempo in attesa di operazioni di I/O (I/O-bound) possono rendere inefficiente lo scheduling se la CPU rimane inattiva mentre attende il completamento di tali operazioni.

1.2.2 Esempi di Algoritmi di Scheduling

Diversi algoritmi sono stati sviluppati per affrontare le problematiche dello scheduling, ognuno con i propri compromessi:

- **First-Come, First-Served (FCFS):**
 - **Descrizione:** Non preemptive. I processi vengono eseguiti nell'ordine in cui arrivano.
 - **Vantaggi:** Semplice da implementare.
 - **Svantaggi:** "Effetto convoglio", dove un processo lungo blocca tutti gli altri, aumentando il tempo medio di attesa.
- **Shortest-Job-First (SJF):**
 - **Descrizione:** Può essere preemptive (Shortest-Remaining-Time-First, SRTF) o non preemptive. Il processo con il tempo di esecuzione stimato più breve viene eseguito per primo.
 - **Vantaggi:** Ottimale per minimizzare il tempo medio di attesa.
 - **Svantaggi:** Difficile conoscere a priori la durata esatta di un job; può portare a starvation per processi lunghi.
- **Priority Scheduling:**
 - **Descrizione:** Può essere preemptive o non preemptive. Ai processi viene assegnata una priorità numerica o concettuale, e viene eseguito quello con la priorità più alta.
 - **Vantaggi:** Prioritizza lavori critici o importanti.
 - **Svantaggi:** Può portare a starvation per processi a bassa priorità; una soluzione è l'aging (aumentare la priorità di un processo che aspetta da troppo tempo).
- **Round Robin (RR):**
 - **Descrizione:** Preemptive. Ogni processo ottiene una piccola porzione di tempo di CPU (quantum). Se non finisce entro il quantum, viene preempted e messo in coda per il prossimo turno.
 - **Vantaggi:** Equo, garantisce un buon tempo di risposta per processi interattivi.

- **Svantaggi:** L'overhead del context switching aumenta se il quantum è troppo piccolo; le performance degradano se il quantum è troppo grande (tende a FCFS).
- **Multilevel Queue Scheduling:**
 - **Descrizione:** I processi sono divisi in diverse code, ognuna con il proprio algoritmo di scheduling (es. una coda per processi foreground con RR, una per processi background con FCFS).
- **Multilevel Feedback Queue Scheduling:**
 - **Descrizione:** Permette ai processi di muoversi tra le code in base al loro comportamento (es. un processo che usa molto la CPU scende di priorità, un processo che aspetta molto sale). È uno degli schedulatori più generali e complessi.

Capitolo 2

Basi di Dati

Le **Basi di Dati (Database)** sono collezioni organizzate di dati che permettono un'efficiente memorizzazione, recupero e gestione delle informazioni. Sono fondamentali per la maggior parte delle applicazioni software moderne.

2.1 Modello Concettuale Entità-Relazione (ER)

Il **Modello Entità-Relazione (ER)** è uno strumento concettuale di alto livello utilizzato nella fase iniziale della progettazione di database. Permette di rappresentare il mondo reale in termini di "entità" (oggetti o concetti di interesse) e "relazioni" (associazioni tra le entità). L'obiettivo è fornire una rappresentazione intuitiva e facilmente comprensibile della struttura dei dati prima di tradurla in un modello logico.

2.1.1 Componenti Principali del Modello ER

- **Entità:** Rappresentano "cose" o "oggetti" del mondo reale su cui si vogliono memorizzare informazioni. Possono essere concrete (es. Persona, Prodotto) o astratte (es. Corso, Ordine). Nel diagramma ER, le entità sono generalmente rappresentate con un rettangolo.
- **Attributi:** Sono le proprietà o caratteristiche che descrivono un'entità o una relazione. Ad esempio, un'entità "Studente" può avere attributi come "Nome", "Cognome", "Matricola", "DataNascita". Nel diagramma ER, gli attributi sono spesso rappresentati con un ovale.
 - **Attributi Semplici/Composti:** Un attributo semplice non può essere scomposto (es. "Età"), mentre uno composto è formato da più attributi (es. "Indirizzo" composto da "Via", "Civico", "Città").
 - **Attributi Mono-valore/Multi-valore:** Mono-valore ha un singolo valore per istanza (es. "DataNascita"), multi-valore può avere più valori (es. "NumeriDiTelefono").
 - **Attributi Derivati:** Il loro valore può essere calcolato da altri attributi (es. "Età" derivata da "DataNascita").
 - **Chiave (Key Attribute):** Un attributo (o un insieme di attributi) che identifica in modo univoco ogni istanza di un'entità. Viene tipicamente sottolineato nel diagramma ER.
- **Relazioni:** Rappresentano associazioni logiche tra due o più entità. Ad esempio, un "Docente" "insegna" a un "Corso". Nel diagramma ER, le relazioni sono rappresentate con un rombo.

- **Cardinalità delle Relazioni:** Definisce il numero di istanze di un'entità che possono essere associate a un'istanza dell'altra entità nella relazione. Le cardinalità più comuni sono:
 - * **Uno a Uno (1:1):** Una istanza di entità A è associata a una e una sola istanza di entità B, e viceversa (es. "Persona" - "ha" - "Patente").
 - * **Uno a Molti (1:N):** Una istanza di entità A è associata a zero o molte istanze di entità B, ma un'istanza di B è associata a una e una sola istanza di A (es. "Dipartimento" - "comprende" - "Docente").
 - * **Molti a Molti (N:M):** Una istanza di entità A è associata a zero o molte istanze di entità B, e viceversa (es. "Studente" - "frequenta" - "Corso").
- **Partecipazione (o Dipendenza):** Indica se l'esistenza di un'istanza di un'entità dipende dalla sua partecipazione a una relazione.
 - * **Totale (o Obbligatoria):** Ogni istanza dell'entità deve partecipare alla relazione (indicata da una doppia linea).
 - * **Parziale (o Opzionale):** Un'istanza dell'entità può partecipare o meno alla relazione (indicata da una singola linea).

2.2 Progettazione Logica: Normalizzazione e Forme Normali

La **normalizzazione** è un processo sistematico di organizzazione dei dati in un database relazionale. Il suo scopo è ridurre la ridondanza dei dati, eliminare le anomalie di aggiornamento (inserimento, cancellazione, modifica) e migliorare l'integrità e la coerenza dei dati. La normalizzazione si basa su una serie di regole chiamate "forme normali".

2.2.1 Obiettivi della Normalizzazione

- **Riduzione della Ridondanza:** Evitare la duplicazione inutile dei dati, che spreca spazio e può portare a incongruenze.
- **Miglioramento dell'Integrità dei Dati:** Assicurare che i dati siano accurati e consistenti.
- **Prevenzione delle Anomalie:**
 - **Anomalia di Inserimento:** Impossibilità di inserire un'informazione a meno che non si inseriscano anche altre informazioni non correlate.
 - **Anomalia di Cancellazione:** La cancellazione di un dato comporta la perdita accidentale di altre informazioni non desiderate.
 - **Anomalia di Aggiornamento:** La modifica di un dato ripetuto richiede l'aggiornamento di più occorrenze, con rischio di inconsistenza se non tutte vengono aggiornate.
- **Flessibilità e Manutenibilità:** Rendere il database più facile da modificare ed estendere.

2.2.2 Principali Forme Normali

Le forme normali sono una serie di regole progressive; per essere in una forma normale N, una relazione deve soddisfare i requisiti della forma normale N-1. Le più comuni e rilevanti per la maggior parte delle applicazioni sono la Prima, Seconda e Terza Forma Normale.

Prima Forma Normale (1NF)

Una relazione è in 1NF se e solo se:

- Tutti gli attributi sono **atomici** (indivisibili). Non ci sono attributi con valori multipli o attributi composti che non sono stati scomposti.
- Ogni record (riga) nella relazione è **unico**. Questo implica che deve esistere una chiave primaria.

Esempio di Violazione: Una colonna "NumeriDiTelefono" che contiene più numeri per un'unica riga.

Seconda Forma Normale (2NF)

Una relazione è in 2NF se e solo se:

- È in 1NF.
- Tutti gli attributi non-chiave dipendono **completamente** dalla chiave primaria. Non ci sono dipendenze parziali, il che significa che nessun attributo non-chiave dipende solo da una parte di una chiave primaria composta.

Esempio di Violazione: In una tabella '(IDCorso, IDStudente, NomeCorso, Voto)', se '(IDCorso, IDStudente)' è la chiave primaria, e 'NomeCorso' dipende solo da 'IDCorso' (e non da 'IDStudente'), allora 'NomeCorso' è parzialmente dipendente e viola la 2NF.

Terza Forma Normale (3NF)

Una relazione è in 3NF se e solo se:

- È in 2NF.
- Non contiene **dipendenze transitive**. Ovvero, nessun attributo non-chiave dipende da un altro attributo non-chiave (anziché dipendere direttamente dalla chiave primaria).

Esempio di Violazione: In una tabella '(IDImpiegato, NomeImpiegato, Dipartimento, CapoDipartimento)', se 'IDImpiegato' è la chiave primaria e 'CapoDipartimento' dipende da 'Dipartimento' (che a sua volta dipende da 'IDImpiegato'), si ha una dipendenza transitiva.

2.2.3 Linguaggio SQL

Il **Structured Query Language (SQL)** è il linguaggio standard per la gestione dei sistemi di gestione di database relazionali (RDBMS). Permette di definire, manipolare e controllare i dati.

Categorie di Comandi SQL

- **Data Definition Language (DDL)**: Utilizzato per definire e modificare la struttura del database.
 - **CREATE**: Crea database, tabelle, viste, indici, ecc. (es. 'CREATE TABLE Studenti (...)').
 - **ALTER**: Modifica la struttura di oggetti esistenti (es. 'ALTER TABLE Studenti ADD COLUMN Età INT').
 - **DROP**: Cancella oggetti dal database (es. 'DROP TABLE Studenti').
- **Data Manipulation Language (DML)**: Utilizzato per manipolare i dati all'interno delle tabelle.
 - **SELECT**: Recupera dati da una o più tabelle. È la query più usata.
 - **INSERT**: Aggiunge nuove righe a una tabella.
 - **UPDATE**: Modifica righe esistenti in una tabella.
 - **DELETE**: Rimuove righe da una tabella.
- **Data Control Language (DCL)**: Utilizzato per gestire i permessi di accesso ai dati.
 - **GRANT**: Concede privilegi agli utenti.
 - **REVOKE**: Rimuove privilegi dagli utenti.
- **Transaction Control Language (TCL)**: Utilizzato per gestire le transazioni (gruppi di operazioni che devono essere eseguite atomicamente).
 - **COMMIT**: Salva le modifiche di una transazione.
 - **ROLLBACK**: Annulla le modifiche di una transazione.

Elementi Comuni delle Query SQL (SELECT)

La query **SELECT** è la più potente e versatile, permettendo di interrogare il database.

- **SELECT**: Specifica le colonne da recuperare.
 - **SELECT** colonna1, colonna2
 - **SELECT** * (tutte le colonne)
 - **SELECT** DISTINCT colonna (solo valori unici)
- **FROM**: Specifica la tabella (o le tabelle) da cui recuperare i dati.
- **WHERE**: Filtra le righe in base a una condizione specificata.
 - **WHERE** condizione (es. 'WHERE Età > 18')
 - Operatori: '=', '>', '<', '>=', '<=', '<>', 'LIKE' (per pattern matching), 'IN', 'BETWEEN', 'IS NULL'.
- **JOIN**: Combina righe da due o più tabelle basandosi su una colonna correlata.
 - **INNER JOIN**: Restituisce solo le righe che hanno corrispondenze in entrambe le tabelle.

- **LEFT (OUTER) JOIN**: Restituisce tutte le righe dalla tabella sinistra e le righe corrispondenti dalla tabella destra (con NULL se non ci sono corrispondenze).
- **RIGHT (OUTER) JOIN**: Simile al LEFT JOIN, ma per la tabella destra.
- **FULL (OUTER) JOIN**: Restituisce tutte le righe quando c'è una corrispondenza in una delle due tabelle.
- **GROUP BY**: Raggruppa le righe che hanno gli stessi valori in una o più colonne, spesso usato con funzioni di aggregazione.
- **HAVING**: Filtra i gruppi creati da 'GROUP BY' in base a una condizione. Si usa con le funzioni di aggregazione.
- **ORDER BY**: Ordina il set di risultati in base a una o più colonne (ASC per ascendente, DESC per discendente).
- **Funzioni di Aggregazione**: Calcolano un singolo valore da un insieme di valori (es. 'COUNT()', 'SUM()', 'AVG()', 'MAX()', 'MIN()').

Capitolo 3

Reti di Calcolatori

Le **reti di calcolatori** sono sistemi che permettono a dispositivi interconnessi di scambiare dati e condividere risorse. Sono la base di quasi ogni infrastruttura informatica moderna, dal World Wide Web alle reti aziendali locali.

3.1 Modello di Comunicazione Client/Server

Il **modello Client/Server** è un'architettura di rete distribuita in cui i client (richiedenti servizi) e i server (fornitori di servizi) sono entità separate che comunicano su una rete. È il modello dominante per la maggior parte delle applicazioni web e molte applicazioni enterprise.

3.1.1 Funzionamento del Modello Client/Server

- **Client:** Invia richieste di servizi al server, riceve le risposte e le presenta all'utente. Tipicamente è l'applicazione utente (es. browser web, app mobile).
- **Server:** Ascolta le richieste dei client, le elabora (es. recupera dati, esegue calcoli), e invia le risposte al client. Gestisce le risorse condivise (database, file, stampanti).
- **Comunicazione:** Avviene tramite protocolli di rete (es. TCP/IP) su porte specifiche. Il client avvia la connessione e la richiesta.

3.1.2 Vantaggi e Svantaggi del Modello Client/Server

- **Vantaggi:**
 - **Centralizzazione:** Gestione centralizzata di dati e risorse, facilitando la sicurezza e la manutenzione.
 - **Scalabilità:** Possibilità di scalare il server per gestire più richieste o aggiungere più client alla rete.
 - **Sicurezza:** Controllo più agevole degli accessi e dei permessi sui dati centralizzati.
 - **Manutenzione Facilitata:** Aggiornamenti e backup possono essere eseguiti sul server senza influenzare i client.
- **Svantaggi:**
 - **Single Point of Failure (Punto Singolo di Fallimento):** Se il server si blocca, tutti i client perdono l'accesso ai servizi.

- **Collo di Bottiglia del Server:** Un server sovraccarico può rallentare l'intera rete.
- **Costo:** I server e la loro manutenzione possono essere costosi.

3.2 Protocollo HTTP

L'**Hypertext Transfer Protocol (HTTP)** è il protocollo applicativo fondamentale per il World Wide Web. Opera nel modello client/server ed è stateless.

3.2.1 Funzionamento e Applicazione in HTTP

- Un **client** (tipicamente un browser web) invia una **richiesta HTTP** (es. GET, POST, PUT, DELETE) a un **server web**. La richiesta include URL, metodo, header e, opzionalmente, un body.
- Il **server** riceve la richiesta, la elabora (es. recupera una pagina web, esegue uno script), e invia una **risposta HTTP**. La risposta include uno stato (es. 200 OK, 404 Not Found), header e il body (es. il contenuto HTML della pagina richiesta).
- La comunicazione avviene tipicamente su TCP/IP (porta 80 per HTTP, 443 per HTTPS).

Esempio: Quando un utente digita un URL nel browser, il browser è il client che invia una richiesta HTTP al server. Il server risponde con la pagina HTML e le risorse associate, che il browser poi renderizza.

3.2.2 Connessioni Stateless e Stateful

Stateless Connection (Connessione Senza Stato)

- **Definizione:** Ogni richiesta inviata dal client al server è completamente indipendente e autocontenuta. Il server non mantiene alcuna informazione (stato) sulle richieste precedenti del client. Ogni richiesta include tutte le informazioni necessarie per essere elaborata.
- **Vantaggi:** Scalabilità elevata (il server non deve allocare memoria per lo stato di ogni client), resilienza (se un server si blocca, un altro può prendere il suo posto senza perdere lo stato), semplicità di progettazione lato server.
- **Svantaggi:** Richiede che ogni richiesta contenga potenzialmente informazioni ridondanti (es. credenziali di autenticazione), e può essere meno efficiente per operazioni che richiedono una sequenza di passaggi.
- **HTTP come Stateless:** HTTP è intrinsecamente stateless. Ogni richiesta HTTP è trattata come se fosse la prima e unica richiesta tra il client e il server.

Stateful Connection (Connessione Con Stato)

- **Definizione:** Il server mantiene e ricorda lo stato delle interazioni passate con un client per un certo periodo di tempo. Le richieste successive possono fare riferimento a questo stato. **Vantaggi:** Minore ridondanza di informazioni nelle richieste successive, può semplificare la logica client per sequenze di operazioni complesse.

- **Svantaggi:** Minore scalabilità (il server deve mantenere lo stato per ogni client attivo, consumando risorse), minore resilienza (se il server che detiene lo stato si blocca, la sessione del client viene persa), complessità maggiore.
- **Esempi:** Una connessione TCP (a un livello più basso) è stateful; una sessione di login a un database.

3.2.3 Metodi per Gestire la Persistenza dello Stato in HTTP

Dato che HTTP è stateless, per costruire applicazioni web interattive che richiedono il mantenimento dello stato (es. carrelli della spesa, sessioni utente), sono stati sviluppati diversi meccanismi:

- **Cookies:** Piccoli frammenti di dati che il server invia al browser del client e che il browser memorizza. Ad ogni richiesta successiva verso lo stesso server, il browser invia nuovamente i cookie al server. Usati per ID di sessione, preferenze utente, tracciamento.
- **Session IDs (ID di Sessione):** Il server crea un ID univoco per ogni sessione utente e lo invia al client (spesso tramite cookie). Il server memorizza i dati della sessione sul proprio lato (nel database o in memoria cache) associati a quell'ID. Usati per mantenere lo stato di login, carrelli della spesa.
- **URL Rewriting (Parametri URL):** Lo stato viene incorporato direttamente nell'URL come parametri. Usato quando i cookie non sono disponibili.
- **Hidden Form Fields:** Dati nascosti all'interno di moduli HTML che vengono inviati con ogni richiesta POST. Usati per mantenere lo stato tra le pagine di un modulo multi-step.
- **Web Storage (Local Storage, Session Storage):** API JavaScript che permettono alle applicazioni web di memorizzare dati nel browser del client (Local Storage persistente, Session Storage per la durata della sessione del browser). Usati per memorizzare dati client-side, cache di dati.

3.3 Standard ISO/OSI

Il **modello ISO/OSI (Open Systems Interconnection)** è un modello concettuale che descrive come i sistemi di comunicazione in una rete interagiscono e cooperano. È diviso in sette strati (layer), ciascuno con responsabilità specifiche, che operano sopra lo strato precedente e forniscono servizi a quello successivo.

3.3.1 Struttura a Sette Strati del Modello OSI

1. **Strato Fisico (Physical Layer):** Gestisce la trasmissione e ricezione di flussi di bit non strutturati e grezzi su un mezzo fisico. Definisce le specifiche elettriche, meccaniche, procedurali e funzionali. (Es. cavi Ethernet, Wi-Fi, connettori).
2. **Strato di Collegamento Dati (Data Link Layer):** Fornisce la trasmissione di dati da nodo a nodo, rilevando e correggendo potenzialmente gli errori che possono verificarsi a livello fisico. Gestisce l'indirizzamento MAC e il controllo del flusso. (Es. Ethernet, PPP).

3. **Strato di Rete (Network Layer):** Gestisce l'instradamento dei pacchetti attraverso la rete (routing). È responsabile dell'indirizzamento logico (IP) e della selezione del percorso migliore. (Es. IP, ICMP).
4. **Strato di Trasporto (Transport Layer):** Fornisce la comunicazione end-to-end tra processi su host diversi. Assicura la consegna affidabile dei dati, il controllo di flusso e il controllo della congestione. (Es. TCP, UDP).
5. **Strato di Sessione (Session Layer):** Stabilisce, gestisce e termina le sessioni di comunicazione tra applicazioni. Gestisce la sincronizzazione e il dialogo.
6. **Strato di Presentazione (Presentation Layer):** Si occupa della sintassi e della semantica dei dati scambiati. Traduce i dati tra il formato dell'applicazione e il formato di rete, e gestisce la crittografia/decriptografia e la compressione.
7. **Strato di Applicazione (Application Layer):** Fornisce servizi di rete direttamente alle applicazioni dell'utente finale. (Es. HTTP, FTP, SMTP, DNS).

3.4 Livello di Trasporto (TCP vs UDP)

Il **Livello di Trasporto** è il quarto strato del modello OSI e fornisce servizi di comunicazione end-to-end tra applicazioni in esecuzione su host diversi. I due protocolli principali a questo livello sono TCP e UDP.

3.4.1 TCP (Transmission Control Protocol)

TCP è un protocollo orientato alla connessione, affidabile e con controllo di flusso e congestione.

- **Orientato alla Connessione:** Stabilisce una connessione (handshake a tre vie) prima di iniziare la trasmissione dei dati e la termina esplicitamente.
- **Affidabile:** Garantisce la consegna dei dati, senza perdite o duplicazioni, e nell'ordine corretto.
 - **Numerazione e Riconoscimenti (ACK):** Ogni segmento inviato è numerato e il mittente si aspetta un riconoscimento (ACK) dal destinatario. Se un ACK non arriva entro un certo tempo, il segmento viene ritrasmesso.
 - **Checksum:** Utilizza un checksum per rilevare errori nei dati.
- **Controllo di Flusso:** Impedisce a un mittente veloce di sovraccaricare un destinatario lento. Il destinatario comunica al mittente quanto spazio buffer è disponibile (finestra di ricezione).
- **Controllo di Congestione:** Evita che un mittente invii troppi dati in una rete congestionata, riducendo la velocità di trasmissione se rileva congestione (es. tramite perdite di pacchetti o ritardi).
- **Segmentazione e Riassemblaggio:** Spezza i dati dell'applicazione in segmenti più piccoli per la trasmissione e li riassembla alla destinazione.
- **Applicazioni Tipiche:** Web Browse (HTTP), trasferimento file (FTP), email (SMTP), connessioni sicure (SSH).

3.4.2 UDP (User Datagram Protocol)

UDP è un protocollo senza connessione, inaffidabile e che non implementa controllo di flusso o congestione.

- **Senza Connessione:** Non stabilisce una connessione preliminare. Ogni datagramma viene inviato indipendentemente.
- **Inaffidabile (Best-Effort):** Non garantisce la consegna dei dati, l'ordine di arrivo, né che non ci siano duplicazioni. Non ci sono ACK né ritrasmissioni automatiche.
- **Nessun Controllo di Flusso/Congestione:** Trasmette i dati alla massima velocità possibile senza preoccuparsi della capacità del destinatario o della rete.
- **Overhead Minimo:** Ha un header molto piccolo, il che lo rende molto efficiente in termini di overhead.
- **Applicazioni Tipiche:** Streaming multimediale (audio/video), VoIP, DNS, giochi online, dove la velocità è più importante dell'affidabilità perfetta (piccole perdite possono essere accettabili).

Capitolo 4

Programmazione Orientata agli Oggetti (e Fondamenti)

La **Programmazione Orientata agli Oggetti (OOP)** è un paradigma di programmazione basato sul concetto di "oggetti", che possono contenere dati e codice. È uno dei paradigmi più diffusi per lo sviluppo di software moderno.

4.1 Concetti Base della Programmazione Orientata agli Oggetti (POO)

La POO si fonda su alcuni pilastri fondamentali che ne definiscono la struttura e il funzionamento:

- **Classe:** Una blueprint o un modello per creare oggetti. Definisce le proprietà (attributi/campi) e i comportamenti (metodi/funzioni) che gli oggetti di quel tipo avranno. Non è un'entità fisica, ma una definizione logica.
- **Oggetto:** Un'istanza di una classe. È un'entità concreta che ha uno stato (valori specifici degli attributi) e un comportamento (i metodi che può eseguire).
- **Incapsulamento (Encapsulation):** Il principio di raggruppare i dati (attributi) e le funzioni (metodi) che operano su quei dati all'interno di un'unica unità (la classe). Protegge i dati interni dall'accesso diretto esterno, permettendone la manipolazione solo tramite metodi pubblici della classe. Questo migliora la sicurezza e la manutenibilità del codice.
- **Ereditarietà (Inheritance):** Un meccanismo che permette a una classe (sottoclasse o classe derivata) di ereditare proprietà e comportamenti da un'altra classe (superclasse o classe base). Promuove il riutilizzo del codice e la creazione di gerarchie di classi che riflettono relazioni "è un tipo di".
- **Polimorfismo (Polymorphism):** Il concetto che un oggetto possa assumere molte forme. In OOP, si riferisce alla capacità di oggetti di classi diverse di rispondere allo stesso messaggio (chiamata di metodo) in modi diversi, o alla capacità di un'interfaccia di riferirsi a oggetti di diverse classi che la implementano. Si manifesta tramite:
 - **Overriding:** Una sottoclasse fornisce un'implementazione specifica di un metodo già definito nella sua superclasse.
 - **Overloading:** Definire più metodi con lo stesso nome all'interno della stessa classe, ma con liste di parametri diverse (numero, tipo o ordine).

4.2 Strutture Dati Astratte (ADT) e Fondamentali di Programmazione

4.2.1 Abstract Data Type (ADT)

Un **Abstract Data Type (ADT)** è una definizione matematica di una struttura dati, che specifica un insieme di dati e un insieme di operazioni che possono essere eseguite su quei dati. È "astratto" perché si concentra sul "cosa" la struttura dati fa (il suo comportamento) piuttosto che sul "come" lo fa (la sua implementazione interna). L'ADT separa l'interfaccia (pubblica) dall'implementazione (privata).

- **Caratteristiche:**

- **Astrazione dei Dati:** Nasconde i dettagli di rappresentazione interna dei dati.
- **Astrazione Funzionale:** Nasconde i dettagli di implementazione delle operazioni.
- **Insieme di Operazioni:** Definisce un insieme ben preciso di funzioni o metodi che possono essere applicati ai dati.

- **Esempi Comuni di ADT:**

- **ADT List (Lista):** Un insieme ordinato di elementi. Operazioni tipiche: inserimento (add), rimozione (remove), accesso a un elemento per indice (get), dimensione (size), verifica se vuota (isEmpty). L'implementazione può essere con array, liste concatenate, ecc.
- **ADT Stack (Pila):** Una collezione di elementi che segue il principio LIFO (Last-In, First-Out). Operazioni tipiche: push (aggiungere un elemento in cima), pop (rimuovere l'elemento in cima), peek (vedere l'elemento in cima senza rimuoverlo), isEmpty.
- **ADT Queue (Coda):** Una collezione di elementi che segue il principio FIFO (First-In, First-Out). Operazioni tipiche: enqueue (aggiungere un elemento in coda), dequeue (rimuovere l'elemento in testa), peek, isEmpty.

4.2.2 Passaggio di Parametri nelle Chiamate di Funzione

Quando si chiama una funzione (o routine, o metodo), i valori o i riferimenti alle variabili vengono passati come argomenti. Esistono due meccanismi principali per il passaggio dei parametri:

Passaggio per Valore (Call by Value)

- **Descrizione:** Viene passata una **copia del valore** dell'argomento alla funzione. La funzione opera su questa copia locale.
- **Effetto:** Qualsiasi modifica apportata alla copia del parametro all'interno della funzione **non influisce** sulla variabile originale passata dal chiamante.
- **Quando usato:** Per tipi di dati primitivi (interi, booleani, caratteri) nella maggior parte dei linguaggi, e per oggetti complessi quando si desidera che la funzione non alteri l'originale.

Passaggio per Riferimento (Call by Reference / Call by Address)

- **Descrizione:** Viene passato l'**indirizzo di memoria** della variabile originale alla funzione. La funzione accede e opera direttamente sulla variabile originale tramite questo indirizzo.
- **Effetto:** Qualsiasi modifica apportata al parametro all'interno della funzione **influenza direttamente** la variabile originale passata dal chiamante.
- **Quando usato:** Spesso per oggetti complessi (in linguaggi come Java, Python, C# gli oggetti sono tipicamente passati per riferimento implicito, anche se tecnicamente è un "passaggio per valore del riferimento"), o esplicitamente in linguaggi come C++ (usando '&' o puntatori).
- **Differenze Chiave:** La differenza fondamentale è se la funzione lavora su una copia (per valore) o direttamente sull'originale (per riferimento). Il passaggio per riferimento è più efficiente per oggetti grandi in quanto evita la copia, ma richiede maggiore attenzione per evitare effetti collaterali indesiderati.

4.2.3 Overloading di Funzioni (o Metodi)

L'**Overloading di funzioni** (o metodi, nel contesto OOP) è la capacità di definire più funzioni o metodi con lo **stesso nome** all'interno dello stesso scope (solitamente la stessa classe), ma che si distinguono per avere **liste di parametri diverse**. La lista dei parametri può differire per:

- Il **numero** di parametri.
- Il **tipo** dei parametri.
- L'**ordine** dei parametri.

Il compilatore (o l'interprete) determina quale versione del metodo chiamare basandosi sul numero e tipo degli argomenti forniti durante la chiamata. **Esempio:**

```
1 FUNCTION Add(a: Integer, b: Integer):  
2     RETURN a + b  
3 END FUNCTION  
4  
5 FUNCTION Add(a: Double, b: Double):  
6     RETURN a + b  
7 END FUNCTION  
8  
9 FUNCTION Add(a: Integer, b: Integer, c: Integer):  
10    RETURN a + b + c  
11 END FUNCTION
```

Listing 4.1: Esempio di Function Overloading

4.3 Linguaggi Compilati e Linguaggi Interpretati

La distinzione tra linguaggi compilati e interpretati riguarda il modo in cui il codice sorgente viene eseguito dal computer.

4.3.1 Linguaggi Compilati

- **Descrizione:** Il codice sorgente viene tradotto (compilato) in codice macchina (o bytecode per la JVM) una sola volta da un programma chiamato "compilatore" prima dell'esecuzione. Il file eseguibile risultante può essere eseguito direttamente dal sistema operativo.
- **Processo:** Codice Sorgente → Compilatore → Codice Macchina Eseguitibile.
- **Vantaggi:**
 - **Prestazioni Elevate:** Il codice macchina è ottimizzato per l'hardware specifico, risultando in esecuzioni molto veloci.
 - **Esecuzione Indipendente:** Una volta compilato, l'eseguibile non richiede il compilatore per essere eseguito.
 - **Rilevamento Errori Precoci:** La maggior parte degli errori di sintassi e alcuni errori logici vengono rilevati in fase di compilazione.
- **Svantaggi:**
 - **Tempo di Compilazione:** Richiede un passaggio aggiuntivo di compilazione prima dell'esecuzione.
 - **Dipendenza dalla Piattaforma:** L'eseguibile compilato è specifico per una particolare architettura hardware e sistema operativo (a meno di VM come Java).
- **Esempi:** C, C++, Java (compila in bytecode che viene interpretato/JIT compilato dalla JVM), Go, Rust.

4.3.2 Linguaggi Interpretati

- **Descrizione:** Il codice sorgente viene eseguito riga per riga da un programma chiamato "interprete" al momento dell'esecuzione, senza una fase di compilazione preventiva in codice macchina.
- **Processo:** Codice Sorgente → Interprete → Esecuzione.
- **Vantaggi:**
 - **Flessibilità e Rapidità di Sviluppo:** Non c'è un passo di compilazione, quindi le modifiche possono essere testate immediatamente.
 - **Indipendenza dalla Piattaforma:** Lo stesso codice sorgente può essere eseguito su qualsiasi piattaforma che abbia un interprete installato.
- **Svantaggi:**
 - **Prestazioni Generalmente Inferiori:** L'interprete analizza e traduce il codice in tempo reale, il che è più lento dell'esecuzione di codice macchina nativo.
 - **Rilevamento Errori Tardo:** Gli errori (anche di sintassi) vengono spesso rilevati solo a runtime, quando l'interprete tenta di eseguire la riga problematica.
- **Esempi:** Python, JavaScript, Ruby, PHP.

4.4 Algoritmi e Complessità Computazionale

L'analisi della **complessità computazionale** è lo studio delle risorse richieste da un algoritmo per risolvere un problema. Le risorse principali sono il tempo di esecuzione e lo spazio di memoria.

4.4.1 Complessità Temporale

Misura il tempo che un algoritmo impiega per completare la sua esecuzione, in funzione della dimensione dell'input. Viene espressa utilizzando la **notazione O-grande (Big O notation)** ($O(n)$), che descrive il tasso di crescita superiore del tempo di esecuzione dell'algoritmo al crescere della dimensione dell'input.

- **$O(1)$ - Tempo Costante:** Il tempo di esecuzione non dipende dalla dimensione dell'input.
- **$O(\log n)$ - Tempo Logaritmico:** Il tempo di esecuzione cresce logaritmicamente con la dimensione dell'input (es. ricerca binaria).
- **$O(n)$ - Tempo Lineare:** Il tempo di esecuzione cresce linearmente con la dimensione dell'input (es. scansione di un array).
- **$O(n \log n)$ - Tempo Lineare-Logaritmico:** Tempo di esecuzione per algoritmi di ordinamento efficienti (es. Merge Sort, Quick Sort).
- **$O(n^2)$ - Tempo Quadratico:** Il tempo di esecuzione è proporzionale al quadrato della dimensione dell'input (es. algoritmi di ordinamento semplici come Bubble Sort, nested loops).
- **$O(2^n)$ - Tempo Esponenziale:** Il tempo di esecuzione cresce esponenzialmente con la dimensione dell'input (es. problemi NP-completi non ottimizzati, ricerca esaustiva).

4.4.2 Complessità Spaziale

Misura la quantità di memoria ausiliaria (oltre all'input stesso) che un algoritmo richiede per completare la sua esecuzione, in funzione della dimensione dell'input. Anche questa è espressa con la notazione O-grande.

- **$O(1)$ - Spazio Costante:** L'algoritmo utilizza una quantità fissa di memoria, indipendentemente dalla dimensione dell'input.
- **$O(n)$ - Spazio Lineare:** La memoria richiesta cresce linearmente con la dimensione dell'input (es. memorizzare una copia dell'input).
- **$O(n^2)$ - Spazio Quadratico:** La memoria richiesta cresce quadraticamente con la dimensione dell'input (es. memorizzare una matrice $N \times N$).

Capitolo 5

Progettazione del Software

La **Progettazione del Software** è il processo di definizione dell'architettura, dei componenti, delle interfacce e di altri attributi di un sistema o di un componente. È una fase cruciale nel ciclo di vita dello sviluppo del software, che traduce i requisiti in un piano dettagliato per la costruzione del sistema.

5.1 Analisi dei Requisiti

L'**analisi dei requisiti** è il processo di definizione, documentazione e mantenimento dei requisiti software. È la fase iniziale di qualsiasi progetto software e mira a comprendere le esigenze degli stakeholder per il sistema da costruire.

5.1.1 Tipi di Requisiti

I requisiti possono essere classificati in due categorie principali:

- **Requisiti Funzionali:** Descrivono ciò che il sistema *deve fare*. Definiscono le funzioni, i servizi e i comportamenti specifici che il sistema deve fornire agli utenti.
 - **Esempi:** "Il sistema deve consentire la registrazione di nuovi utenti.", "Il sistema deve calcolare la somma delle vendite giornaliere.", "Il sistema deve permettere la visualizzazione del calendario delle partite."
- **Requisiti Non Funzionali (o di Qualità):** Descrivono come il sistema *deve funzionare*. Definiscono le caratteristiche di qualità, i vincoli e gli attributi del sistema, piuttosto che le sue funzionalità specifiche. Spesso influenzano l'architettura e l'implementazione.
 - **Esempi:**
 - * **Prestazioni:** "Il sistema deve rispondere a una query entro 2 secondi."
 - * **Scalabilità:** "Il sistema deve supportare 1000 utenti concorrenti."
 - * **Sicurezza:** "Il sistema deve autenticare gli utenti tramite username e password con crittazione."
 - * **Usabilità:** "L'interfaccia utente deve essere intuitiva e di facile apprendimento."
 - * **Affidabilità:** "Il sistema deve essere disponibile il 99.9% del tempo."
 - * **Manutenibilità:** "Il codice deve essere modulare e ben documentato."
 - * **Portabilità:** "Il sistema deve funzionare su Windows e Linux."

5.1.2 Diagrammi di Casi d'Uso UML (Use Case Diagrams)

I **Diagrammi di Casi d'Uso** sono uno strumento UML (Unified Modeling Language) utilizzato nell'analisi dei requisiti funzionali. Descrivono le interazioni tra gli utenti (attori) e il sistema, rappresentando le diverse funzionalità che il sistema offre dal punto di vista dell'utente.

- **Componenti Principali:**

- **Attore (Actor):** Rappresenta un ruolo esterno che interagisce con il sistema (persona, altro sistema, dispositivo). Disegnato come un omino stilizzato.
- **Caso d'Uso (Use Case):** Rappresenta una funzionalità specifica del sistema, un servizio che il sistema fornisce all'attore. Disegnato come un ovale.
- **Confine del Sistema (System Boundary):** Un rettangolo che racchiude i casi d'uso, distinguendo ciò che è all'interno del sistema da ciò che è esterno.
- **Relazioni:**
 - * **Associazione:** L'interazione tra un attore e un caso d'uso (linea semplice).
 - * **Include:** Un caso d'uso include la funzionalità di un altro caso d'uso (freccia tratteggiata da caso d'uso includente a caso d'uso incluso, con etichetta '«include»').
 - * **Extend:** Un caso d'uso estende il comportamento di un altro caso d'uso in circostanze specifiche (freccia tratteggiata da caso d'uso estendente a caso d'uso esteso, con etichetta '«extend»').
 - * **Generalizzazione:** Una relazione di ereditarietà tra attori o casi d'uso (freccia con punta triangolare).

- **Scopo:** Fornire una visione ad alto livello dei requisiti funzionali, facilitare la comunicazione tra stakeholder e sviluppatori, e servire da base per la progettazione successiva.

5.2 Progettazione dell'Architettura del Sistema Software

La **progettazione dell'architettura del software** definisce la struttura di alto livello di un sistema software, delineando come i suoi componenti interagiscono e sono organizzati. Include la scelta di pattern architetturali e di design.

5.2.1 Design Patterns (Pattern Architetturali e di Progettazione)

I **Design Patterns** sono soluzioni riutilizzabili a problemi comuni che si presentano nella progettazione del software. Non sono soluzioni pronte all'uso, ma modelli da adattare al contesto specifico.

- **Vantaggi:**

- **Riutilizzo di Soluzioni Comprovate:** Utilizzano approcci che hanno dimostrato di funzionare.
- **Vocabolario Condiviso:** Facilitano la comunicazione tra gli sviluppatori.
- **Miglioramento della Qualità del Codice:** Portano a codice più manutenibile, scalabile e flessibile.

- **Esempi Comuni:**

- **MVC (Model-View-Controller):** Un pattern architetturale che separa l'applicazione in tre componenti interconnessi per gestire meglio l'interfaccia utente.
 - * **Model:** Gestisce i dati e la logica di business.
 - * **View:** Si occupa della presentazione dei dati all'utente.
 - * **Controller:** Gestisce l'input dell'utente e coordina Model e View.
- **Singleton:** Garantisce che una classe abbia una sola istanza e fornisce un punto di accesso globale a essa. Utile per risorse uniche come gestori di configurazione o log.
- **Factory Method:** Definisce un'interfaccia per creare un oggetto, ma lascia alle sottoclassi la decisione di quale classe istanziare. Permette di creare oggetti senza specificare la classe esatta che verrà creata.
- **Observer:** Definisce una dipendenza uno-a-molti tra oggetti in modo che quando un oggetto cambia stato, tutti i suoi dipendenti vengono notificati e aggiornati automaticamente. Utile per eventi e notifiche.
- **Strategy:** Definisce una famiglia di algoritmi, incapsula ciascuno di essi e li rende intercambiabili. Permette all'algoritmo di variare indipendentemente dai client che lo usano.

5.2.2 Diagrammi UML di Classi e di Deployment

Diagrammi delle Classi (Class Diagrams)

I **Diagrammi delle Classi** sono diagrammi strutturali UML che mostrano la struttura statica di un sistema, le classi, i loro attributi, i loro metodi e le relazioni tra le classi.

- **Componenti Principali:**

- **Classi:** Rappresentate da un rettangolo diviso in tre sezioni: nome della classe, attributi, e metodi.
- **Relazioni:**
 - * **Associazione:** Connessione tra istanze di classi (linea semplice, può avere molteplicità).
 - * **Aggregazione:** Una forma debole di relazione "parte di", dove un oggetto può esistere indipendentemente dal contenitore (diamante vuoto).
 - * **Composizione:** Una forma forte di relazione "parte di", dove il componente non può esistere senza il contenitore (diamante pieno).
 - * **Generalizzazione (Ereditarietà):** Una classe eredita da un'altra (freccia con punta triangolare non piena).
 - * **Realizzazione (Implementazione di Interfaccia):** Una classe implementa un'interfaccia (linea tratteggiata con punta triangolare non piena).
 - * **Dipendenza:** Un cambiamento in una classe può influenzare un'altra (linea tratteggiata con freccia).

- **Scopo:** Modellare il design logico del database, la struttura del codice e le relazioni tra le classi.

Diagrammi di Deployment (Deployment Diagrams)

I **Diagrammi di Deployment** sono diagrammi strutturali UML che mostrano la configurazione fisica dei nodi hardware (computer, server, dispositivi) e come i componenti software sono distribuiti su questi nodi.

- **Componenti Principali:**

- **Nodo (Node):** Una risorsa computazionale fisica o logica (hardware, ambiente di esecuzione software). Rappresentato da un cubo 3D.
- **Artefatto (Artifact):** Un prodotto fisico risultante dal processo di sviluppo (file eseguibile, libreria, file di configurazione). Rappresentato da un documento con l'icona di un artefatto.
- **Comunicazione (Communication Path):** Una linea che connette i nodi, indicando un percorso di comunicazione.

- **Scopo:** Visualizzare la topologia di sistema, la distribuzione dei componenti software sull'hardware e le interazioni fisiche. Utile per architetture distribuite.

5.3 Implementazione e Diagrammi di Sequenza UML

I **Diagrammi di Sequenza** sono diagrammi di interazione UML che mostrano l'ordine cronologico dei messaggi scambiati tra oggetti in un'interazione. Sono usati per modellare la logica di una funzionalità o di un algoritmo.

- **Componenti Principali:**

- **Lifeline (Linea di Vita):** Rappresenta la partecipazione di un oggetto o attore all'interazione (linea verticale tratteggiata).
- **Attore:** Utente o sistema esterno che avvia l'interazione.
- **Messaggio:** Chiamata di metodo o comunicazione tra lifeline (freccia orizzontale).
- **Barra di Attivazione (Activation Bar):** Indica il periodo di tempo durante il quale un oggetto è attivo e sta eseguendo un'operazione.

- **Scopo:** Dettagliare il flusso di controllo e di dati per funzionalità specifiche, spesso come implementazione di casi d'uso. Sono utili per visualizzare come diversi oggetti collaborano per raggiungere un obiettivo.