

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

Paolo Pietrelli

28 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
1.1.3 Processi e Thread	2
1.2 Record di Attivazione (Activation Record / Stack Frame)	3
1.2.1 Contenuto di un Record di Attivazione	4
1.2.2 Ciclo di Vita di un Record di Attivazione	4
1.3 Gestione della Memoria	5
1.3.1 Memoria Virtuale	5
1.3.2 Algoritmi di Allocazione della Memoria (per Blocchi Contigui)	6
1.4 Gestione dei File System	7
1.4.1 Organizzazione Fisica dei Dati su Disco	7
1.4.2 Metodi di Allocazione dei File	7
1.5 Sincronizzazione e Deadlock	9
1.5.1 Problemi di Sincronizzazione	9
1.5.2 Meccanismi di Sincronizzazione	9
1.5.3 Deadlock (Interblocco)	11
1.6 Scheduling della CPU	12
1.6.1 Principali Problematiche dello Scheduling	12
1.6.2 Esempi di Algoritmi di Scheduling	12
Domande e Esercizi	16
Altre Possibili Domande	21
2 Basi di Dati	25
2.1 Modello Concettuale Entità-Relazione (ER)	25
2.1.1 Componenti Base e Rappresentazione Grafica	25
2.1.2 Dettagli degli Attributi e Identificatori	25
2.1.3 Dettagli delle Relazioni e Cardinalità	26
2.1.4 Generalizzazione nel Modello ER	27
2.1.5 Esempi Complessi di Schemi ER	27
2.1.6 Esempi Complessi di Schemi ER	27
2.2 Progettazione Logica: Normalizzazione e Forme Normali	32
2.2.1 Perché la Normalizzazione è Importante	32
2.2.2 Forme Normali: Livelli di Normalizzazione	33
2.2.3 Linguaggio SQL	36
3 Reti di Calcolatori	41
3.1 Modello di Comunicazione Client/Server	41
3.1.1 Funzionamento del Modello Client/Server	41

3.1.2	Vantaggi e Svantaggi del Modello Client/Server	41
3.2	Protocollo HTTP	42
3.2.1	Funzionamento e Applicazione in HTTP	42
3.2.2	Connessioni Stateless e Stateful	42
3.2.3	Metodi per Gestire la Persistenza dello Stato in HTTP	43
3.3	Standard ISO/OSI	43
3.3.1	Struttura a Sette Strati del Modello OSI	43
3.4	Livello di Trasporto (TCP vs UDP)	44
3.4.1	TCP (Transmission Control Protocol)	44
3.4.2	UDP (User Datagram Protocol)	45
3.5	Subnetting (Suddivisione di Sottoreti)	45
3.5.1	Obiettivi del Subnetting	45
3.5.2	Indirizzi IP e Maschere di Sottorete	45
3.5.3	Calcolo del Subnetting (Esempio Pratico)	46
4	Programmazione Orientata agli Oggetti (e Fondamenti)	49
4.1	Concetti Base della Programmazione Orientata agli Oggetti (POO)	49
4.2	Strutture Dati Astratte (ADT) e Fondamentali di Programmazione	50
4.2.1	Abstract Data Type (ADT)	50
4.2.2	Passaggio di Parametri nelle Chiamate di Funzione	50
4.2.3	Overloading di Funzioni (o Metodi)	51
4.3	Linguaggi Compilati e Linguaggi Interpretati	52
4.3.1	Linguaggi Compilati	52
4.3.2	Linguaggi Interpretati	53
4.4	Algoritmi e Complessità Computazionale	53
4.4.1	Complessità Temporale	53
4.4.2	Complessità Spaziale	54
5	Progettazione del Software	55
5.1	Analisi dei Requisiti	55
5.1.1	Tipi di Requisiti	55
5.1.2	Diagrammi di Casi d'Uso UML (Use Case Diagrams)	56
5.2	Progettazione dell'Architettura del Sistema Software	56
5.2.1	Design Patterns (Pattern Architetturali e di Progettazione)	56
5.2.2	Diagrammi UML (Unified Modeling Language)	57
6	Elettronica	67
6.1	Transistore MOS (Metal-Oxide-Semiconductor Field-Effect Transistor)	67
6.1.1	Struttura del Transistore MOS (nMOS)	67
6.1.2	Principio di Funzionamento (nMOS)	68
6.1.3	Regimi di Funzionamento (per nMOS con $V_{GS} > V_{TH}$)	68
6.1.4	Vantaggi del MOS rispetto al Transistore Bipolare (BJT)	69
6.2	Circuiti CMOS (Complementary MOS)	69
6.2.1	Invertitore CMOS	69
6.2.2	Vantaggi dell'Invertitore CMOS rispetto agli Invertitori con Carico Resistivo	70
6.2.3	Applicazioni Notevoli dell'Invertitore CMOS	70

7 Sistemi Numerici	73
7.1 Rappresentazione per Numeri Interi	73
7.1.1 Complemento a Due	73
7.2 Operazioni Aritmetiche con Complemento a Due	74
7.2.1 Addizione	74
7.2.2 Sottrazione	75
7.2.3 Overflow	75

Elenco delle figure

2.1	Esempi di rappresentazione di attributi semplici (es. Codice Fiscale) e attributi composti (es. Data di Nascita) in un Diagramma ER.	26
2.2	Esempi di utilizzo di identificatori esterni, inclusi scenari con entità deboli e concatenazione di identificazione.	27
2.3	Esempio di Generalizzazione nel Diagramma ER: l'entità Persona si generalizza nelle entità Uomo e Donna, mostrando la distinzione e l'ereditarietà delle proprietà.	28
2.4	Diagramma ER che modella le relazioni tra Studenti, Corsi ed Esami, inclusi attributi e cardinalità.	28
2.5	Diagramma ER che modella le relazioni tra Studenti, Corsi ed Esami, inclusi attributi e cardinalità.	29
2.6	Schema ER per l'esercizio 2: Sistema Cinema/Film.	29
2.7	Schema ER per l'esercizio 7: Sistema Campionato di Calcio.	30
2.8	Schema ER per l'esercizio 11: Sistema Gare Ciclistiche.	31
2.9	Diagramma di Venn delle Forme Normali (1NF, 2NF, 3NF, BCNF, 4NF, 5NF), che mostra la loro relazione gerarchica.	33
2.10	Esempio di una tabella non in 1NF e la sua normalizzazione a 1NF, eliminando l'attributo multi-valore.	34
2.11	Esempio di una tabella non in 2NF e la sua normalizzazione a 2NF tramite scomposizione in due relazioni.	34
2.12	Esempio di una tabella "Computer" non in 3NF a causa di dipendenze transitive.	35
2.13	Normalizzazione della tabella "Computer" a 3NF, scomponendo le relazioni per eliminare le dipendenze transitive.	35
2.14	Esempio di una tabella "Impiegato" non in 3NF per dipendenza transitiva.	35
2.15	Esempio di una query SQL che utilizza operatori e funzioni avanzate per filtrare e aggregare i dati.	39
5.1	Rappresentazione grafica delle relazioni comuni tra Casi d'Uso in UML (Inclusione, Estensione e Generalizzazione).	58
5.2	Esempio completo di Diagramma dei Casi d'Uso per un sistema di Ristorazione, che illustra le interazioni tra attori (Cameriere, Cliente, Cuoco, Cassiere) e le funzionalità del sistema.	59
5.3	Esempi di rappresentazione base di una classe UML, con attributi e metodi, e come si relaziona al codice.	60
5.4	Esempio di Diagramma delle Classi che illustra l'uso dei modificatori di visibilità (public, private, protected) per attributi e metodi.	60
5.5	Esempio di Diagramma delle Classi che mostra la relazione di ereditarietà tra le classi Persona, Cliente e Potenziale Cliente.	61
5.6	Esempio di Diagramma delle Classi che illustra come le interfacce consentono di simulare l'ereditarietà multipla in linguaggi che non la supportano nativamente.	61

5.7	Confronto visivo tra Aggregazione e Composizione nei Diagrammi delle Classi UML.	62
5.8	Diagramma delle Classi per un sistema di gestione prodotti/catalogo/carrello, che illustra varie classi e le loro relazioni (es. Prodotto, Carrello, DVD, Libro). .	62
5.9	Diagramma delle Classi per un sistema di Chat, che mostra le interazioni tra classi come Utente, Chat e Messaggio, e le loro molteplicità.	63
5.10	Diagramma di Sequenza UML che illustra il flusso di aggiornamento dei dati in un'architettura Model-View-Controller (MVC).	64
5.11	Diagramma di Sequenza UML che illustra il flusso di prestito di un libro in un sistema di gestione bibliotecaria.	65

Listings

1.1	Funzione pow2(x)	20
2.1	Esempio Operatori Logici	37
2.2	Esempio Operatori di Confronto	38
2.3	Esempio Funzioni Stringa	38
2.4	Esempio Funzioni Numeriche	38
2.5	Esempio Funzioni Data/Ora	39
4.1	Esempio di Passaggio per Valore	51
4.2	Esempio di Passaggio per Riferimento	51
4.3	Esempio di Function Overloading	52

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.

1.1.3 Processi e Thread

Nei sistemi operativi moderni, l'esecuzione dei programmi è gestita attraverso due concetti principali: i processi e i thread.

Processo (Processo "Pesante")

Un **processo** è un'istanza di un programma in esecuzione. È un'unità di allocazione delle risorse del sistema operativo e include:

- Il codice del programma.
- I dati (variabili globali, heap).
- Lo stack (per le chiamate di funzione e le variabili locali).

- Il Program Counter (PC) e i registri della CPU.
- Risorse del sistema operativo allocate (file aperti, segnali, memoria, ecc.).

Ogni processo ha il proprio spazio di indirizzamento virtuale separato, il che fornisce isolamento e protezione tra i processi. Se un processo crasha, di solito non influisce sugli altri processi. I processi sono considerati "pesanti" a causa dell'overhead associato alla loro creazione, alla distruzione e al cambio di contesto.

Thread (Processo "Leggero")

Un **thread** (o thread di esecuzione) è un'unità di esecuzione all'interno di un processo. Un singolo processo può contenere più thread. I thread all'interno dello stesso processo condividono lo stesso spazio di indirizzamento virtuale, il codice del programma, i dati globali e le risorse del sistema operativo (file aperti, ecc.). Ogni thread ha il proprio:

- Program Counter (PC).
- Set di registri della CPU.
- Stack separato.

I thread sono considerati "leggeri" perché la loro creazione, distruzione e il cambio di contesto sono molto più veloci ed efficienti rispetto ai processi, dato che non richiedono la creazione di un nuovo spazio di indirizzamento e la copia di risorse.

Punti in Comune e Differenze

- **Punti in Comune:** Sia processi che thread rappresentano unità di esecuzione che possono essere schedulate dalla CPU. Entrambi hanno un Program Counter, un set di registri e uno stack.
- **Differenze Principali:**
 - **Isolamento delle Risorse:** I processi hanno spazi di indirizzamento separati e risorse isolate. I thread all'interno dello stesso processo condividono lo spazio di indirizzamento e le risorse.
 - **Overhead:** I processi hanno un overhead maggiore per creazione, distruzione e cambio di contesto. I thread sono più "leggeri".
 - **Comunicazione:** La comunicazione tra processi (IPC - Inter-Process Communication) è più complessa (richiede meccanismi esplicativi come pipe, shared memory, message queues). La comunicazione tra thread (ITC - Inter-Thread Communication) è più semplice e diretta, poiché condividono la memoria.
 - **Robustezza:** Un crash di un thread può compromettere l'intero processo (e tutti i suoi thread). Un crash di un processo non influisce sugli altri processi.

1.2 Record di Attivazione (Activation Record / Stack Frame)

Il **Record di Attivazione** (o *Stack Frame*) è una struttura dati creata sullo stack del programma ogni volta che una funzione (o procedura, o subroutine) viene chiamata. Contiene tutte le informazioni necessarie per la gestione dell'esecuzione di quella specifica chiamata di

funzione. È una componente fondamentale del runtime di un linguaggio di programmazione e del supporto fornito dal sistema operativo per l'esecuzione di programmi che utilizzano stack per le chiamate a funzione.

1.2.1 Contenuto di un Record di Attivazione

Un tipico record di attivazione può contenere le seguenti informazioni, sebbene la loro esatta disposizione e i dettagli possano variare a seconda dell'architettura della CPU, del sistema operativo e del compilatore:

- **Parametri Attuali (Actual Parameters):** I valori degli argomenti passati alla funzione.
- **Indirizzo di Ritorno (Return Address):** L'indirizzo di memoria della istruzione nel codice chiamante a cui il controllo deve tornare una volta che la funzione corrente ha terminato l'esecuzione.
- **Valore di Ritorno (Return Value):** Uno spazio per memorizzare il valore restituito dalla funzione (se la funzione restituisce un valore).
- **Variabili Locali (Local Variables):** Le variabili dichiarate all'interno della funzione e che esistono solo per la durata di quella specifica attivazione.
- **Stato dei Registri Salvati (Saved Register State):** I valori dei registri della CPU che erano in uso dalla funzione chiamante e che devono essere ripristinati prima del ritorno.
- **Puntatore al Frame Precedente (Control Link / Dynamic Link):** Un puntatore (indirizzo) al record di attivazione della funzione chiamante, permettendo al sistema di "risalire" lo stack.
- **Puntatore al Contesto Statico (Access Link / Static Link):** (Solo per linguaggi con scope annidato statico, come Pascal o linguaggi funzionali) Un puntatore al record di attivazione della funzione che definisce lo scope lessicale della funzione corrente, permettendo l'accesso a variabili non locali.

1.2.2 Ciclo di Vita di un Record di Attivazione

Il ciclo di vita di un record di attivazione segue il flusso delle chiamate a funzione e la struttura dello stack:

1. **Creazione (Chiamata di Funzione):** Quando una funzione viene chiamata, il codice chiamante (o il runtime) crea un nuovo record di attivazione. Questo nuovo record viene "pushed" (inserito) in cima allo stack di esecuzione del processo. Il puntatore dello stack (Stack Pointer) viene aggiornato per puntare a questo nuovo frame.
2. **Esecuzione:** La CPU salta all'indirizzo di inizio del codice della funzione chiamata. La funzione utilizza le informazioni nel suo record di attivazione (parametri, variabili locali, registri) durante la sua esecuzione.
3. **Terminazione (Ritorno da Funzione):** Quando la funzione termina (raggiunge un 'return' o la fine del suo blocco di codice), il valore di ritorno (se presente) viene posizionato in una posizione designata (spesso un registro). Il record di attivazione corrente viene "popped" (rimosso) dallo stack, liberando lo spazio di memoria che occupava. Il puntatore dello stack viene ripristinato al record di attivazione precedente.

- 4. **Ripristino del Contesto:** Il controllo del programma viene trasferito all'indirizzo di ritorno memorizzato nel record di attivazione appena rimosso, e i registri salvati vengono ripristinati per continuare l'esecuzione della funzione chiamante.

Questo meccanismo a stack garantisce che le chiamate a funzione e i ritorni avvengano in modo LIFO (Last-In, First-Out), permettendo la corretta gestione delle funzioni annidate e ricorsive.

1.3 Gestione della Memoria

La **gestione della memoria** è una delle funzioni fondamentali del sistema operativo, responsabile di allocare e deallocare la memoria ai processi, di proteggere la memoria per evitare interferenze tra i processi e di fornire un'astrazione della memoria fisica per gli sviluppatori. L'obiettivo principale è massimizzare l'utilizzo della CPU mantenendo molti processi in memoria e fornendo un meccanismo efficiente per accedere ai dati.

1.3.1 Memoria Virtuale

La **memoria virtuale** è una tecnica che permette a un sistema operativo di compensare la carenza di memoria fisica (RAM) usando la memoria secondaria (spazio su disco) per simulare più RAM. Questo permette ai programmi di usare più memoria di quella fisicamente disponibile e facilita il multitasking, isolando lo spazio di indirizzamento di ogni processo. Le tecniche principali per implementare la memoria virtuale sono la paginazione e la segmentazione.

Paginazione (Paging)

La paginazione è una tecnica di gestione della memoria virtuale che suddivide lo spazio di indirizzamento logico di un processo in blocchi di dimensione fissa chiamati **pagine**. La memoria fisica è anch'essa divisa in blocchi della stessa dimensione, chiamati **frame**.

- **Funzionamento:** Quando un processo viene caricato, le sue pagine possono essere caricate in frame non contigui della memoria fisica. La traduzione dell'indirizzo logico (virtuale) in indirizzo fisico avviene tramite una **Page Table** (Tabella delle Pagine). Ogni processo ha la propria Page Table, che mappa le pagine logiche ai frame fisici.
- **Traduzione dell'Indirizzo:** L'indirizzo logico è diviso in due parti: il numero di pagina (page number, p) e l'offset all'interno della pagina (offset, d). La Page Table usa il numero di pagina per trovare il frame corrispondente, e l'offset viene aggiunto al frame address per ottenere l'indirizzo fisico.
- **TLB (Translation Lookaside Buffer):** Una cache hardware veloce che memorizza le mappature pagina-frame usate più frequentemente per accelerare il processo di traduzione degli indirizzi e compensare il fatto che ogni accesso alla memoria richiede due accessi (uno alla Page Table e uno al dato).
- **Vantaggi:**
 - Elimina la **frammentazione esterna** (non ci sono buchi inutilizzati tra i blocchi allocati).
 - Semplifica l'allocazione della memoria.
- **Svantaggi:**

- Introduce la **frammentazione interna** (il frame finale potrebbe non essere completamente riempito dalla pagina, lasciando spazio inutilizzato).
- Overhead della Page Table (può essere grande e richiede memoria).

Segmentazione (Segmentation)

La segmentazione è una tecnica di gestione della memoria che permette di visualizzare la memoria come una collezione di segmenti di dimensione variabile. Ogni segmento corrisponde a un'unità logica del programma (es. codice, dati, stack, subroutine).

- **Funzionamento:** L'indirizzo logico è composto da un numero di segmento e un offset. La **Segment Table** (Tabella dei Segmenti) memorizza l'indirizzo base e la lunghezza di ciascun segmento in memoria fisica. La traduzione avviene verificando che l'offset non superi la lunghezza del segmento e aggiungendo l'offset all'indirizzo base del segmento.

- **Vantaggi:**

- Riflette la visione logica del programma (moduli, funzioni).
- Facilita la protezione e la condivisione di segmenti tra processi.

- **Svantaggi:**

- Soffre di **frammentazione esterna** (spazi liberi tra i segmenti che possono essere troppo piccoli per nuove allocazioni).
- La gestione delle dimensioni variabili dei segmenti è più complessa.

1.3.2 Algoritmi di Allocazione della Memoria (per Blocchi Contigui)

Quando la memoria fisica viene allocata in blocchi contigui (come nella segmentazione o senza memoria virtuale), il sistema operativo deve decidere quale blocco libero assegnare a una richiesta di memoria.

- **First Fit:**

- **Descrizione:** Alloca il primo blocco libero che trova che sia abbastanza grande da soddisfare la richiesta.
- **Vantaggi:** Veloce da implementare, spesso porta a buoni risultati.
- **Svantaggi:** Può lasciare molti piccoli blocchi liberi all'inizio della lista, o frammentare blocchi grandi.

- **Best Fit:**

- **Descrizione:** Alloca il blocco libero più piccolo che sia comunque sufficientemente grande a soddisfare la richiesta.
- **Vantaggi:** Lascia i blocchi liberi più grandi intatti per richieste future, riducendo la frammentazione interna.
- **Svantaggi:** Richiede una scansione completa della lista dei blocchi liberi (o una ricerca in una lista ordinata), quindi è più lento. Genera molti blocchi liberi molto piccoli.

- **Worst Fit:**

- **Descrizione:** Alloca il blocco libero più grande disponibile.
- **Vantaggi:** Si cerca di lasciare un blocco residuo il più grande possibile dopo l'allocazione, teoricamente utile per richieste future.
- **Svantaggi:** Spesso è l'algoritmo peggiore in termini di frammentazione. Rende la ricerca più lenta e frammenta i blocchi più grandi.

1.4 Gestione dei File System

La **gestione dei file system** è un componente fondamentale del sistema operativo, responsabile di organizzare, archiviare e recuperare i dati su dispositivi di memoria secondaria (come dischi rigidi, SSD). Fornisce un'interfaccia logica per l'utente e le applicazioni per interagire con i dati, astraendo i dettagli di basso livello dell'hardware di archiviazione.

1.4.1 Organizzazione Fisica dei Dati su Disco

La memoria secondaria è tipicamente organizzata in unità di archiviazione fisiche che il file system gestisce.

- **Settori:** La più piccola unità fisica di memorizzazione su un disco.
- **Blocchi (o Cluster):** L'unità di trasferimento logica più piccola riconosciuta dal file system. Un blocco è composto da uno o più settori contigui. La dimensione del blocco è un compromesso: blocchi grandi riducono l'overhead di I/O ma aumentano la frammentazione interna; blocchi piccoli aumentano l'overhead di I/O ma riducono la frammentazione interna.
- **Cilindri/Tracce:** Concetti fisici legati ai dischi rotanti (hard disk), dove le tracce sono anelli concentrici e i cilindri sono l'insieme di tracce alla stessa distanza dal centro su tutti i piatti.

Struttura delle Directory

Le directory sono strutture che organizzano i file e altre directory in una gerarchia.

- **Struttura a Singolo Livello:** Tutti i file sono nella stessa directory. Semplice, ma difficile da gestire per molti utenti/file.
- **Struttura a Due Livelli:** Ogni utente ha la propria directory principale, separata dagli altri. Impedisce conflitti di nomi tra utenti, ma non offre organizzazione interna.
- **Struttura Ad Albero (Gerarchica):** La struttura più comune, con una directory radice e sottodirectory. Offre flessibilità e organizzazione.
- **Grafo Aciclico Diretto (DAG):** Permette la condivisione di file e directory tra diversi percorsi (tramite link hard o soft/symbolic).
- **Implementazione Interna:** Le directory possono essere implementate come liste di voci (nome file, puntatore a i-node/FAT entry) o tabelle hash per una ricerca più veloce.

1.4.2 Metodi di Allocazione dei File

Il metodo di allocazione determina come i blocchi di un file sono memorizzati e gestiti sul disco, influenzando l'efficienza di accesso e la gestione dello spazio.

Allocazione Contigua

- **Descrizione:** Ogni file è memorizzato come un blocco contiguo di blocchi sul disco. Il file system memorizza solo l'indirizzo del primo blocco e la lunghezza del file.

- **Vantaggi:**

- Semplice da implementare.
- Ottime prestazioni per l'accesso sequenziale e diretto (un solo seek per leggere un file intero, accesso diretto immediato a qualsiasi blocco).

- **Svantaggi:**

- **Frammentazione Esterna:** Con il tempo, lo spazio libero sul disco si frammenta in piccoli buchi non contigui, rendendo difficile trovare blocchi contigui grandi per nuovi file, anche se c'è molto spazio totale disponibile.
- Difficile prevedere la dimensione finale dei file.
- Richiede compattazione periodica per recuperare spazio contiguo (operazione costosa).

Allocazione Collegata (Linked Allocation)

- **Descrizione:** Ogni file è una lista collegata di blocchi su disco. Ogni blocco contiene un puntatore al blocco successivo. Il file system memorizza solo l'indirizzo del primo blocco.

- **Vantaggi:**

- Nessuna frammentazione esterna.
- Allocazione dinamica e facile espansione dei file.

- **Svantaggi:**

- Lento per l'accesso diretto/casuale (bisogna seguire la catena di puntatori).
- Spazio sprecato per i puntatori in ogni blocco (anche se questo è spesso mitigato da una File Allocation Table - FAT).
- Rischio di perdita dell'intera catena se un puntatore è corrotto.

Allocazione Indicizzata (Indexed Allocation)

- **Descrizione:** Ogni file ha un blocco indice dedicato, che è un array di puntatori ai blocchi di dati effettivi del file. Il file system memorizza solo l'indirizzo del blocco indice.

- **Vantaggi:**

- Combina i vantaggi dell'allocazione contigua e collegata: supporta sia l'accesso sequenziale che diretto.
- Nessuna frammentazione esterna.
- Facile espansione (aggiungendo puntatori al blocco indice o blocchi indice multipli/gerarchici).

- **Svantaggi:**

- Overhead dello spazio per i blocchi indice.

- La dimensione del file è limitata dalla dimensione del blocco indice.
- **Implementazione Comune: i-nodes (Unix/Linux):** Ogni file ha un i-node che contiene metadati del file e un array di puntatori ai blocchi di dati, inclusi puntatori a blocchi indiretti per file di grandi dimensioni. **FAT (File Allocation Table - MS-DOS/Windows):** Una tabella sul disco che memorizza le catene di blocchi per ogni file. Non è esattamente un blocco indice per file singoli, ma un array centrale di puntatori.

1.5 Sincronizzazione e Deadlock

Nei sistemi operativi multi-programmati o multi-thread, dove più processi o thread condividono risorse e dati, è fondamentale garantire la **sincronizzazione** per mantenere la coerenza dei dati e prevenire condizioni di errore come le race condition. La mancanza di sincronizzazione può portare anche a situazioni di **deadlock**.

1.5.1 Problemi di Sincronizzazione

- **Race Condition (Condizione di Corsa):** Si verifica quando più processi o thread accedono e manipolano dati condivisi concorrentemente, e il risultato finale dell'esecuzione dipende dall'ordine in cui le operazioni dei processi/thread si intersecano. Il risultato non è predicibile.
- **Sezione Critica (Critical Section):** Una sezione di codice in cui un processo o thread accede a risorse condivise (variabili, file, periferiche). L'obiettivo della sincronizzazione è garantire che, in un dato istante, al massimo un processo/thread sia nella sua sezione critica per quella risorsa.
- **Problema della Sezione Critica (CriticalSection Problem):** Progettare protocolli per garantire che i processi cooperanti accedano alle sezioni critiche in modo sicuro, rispettando le seguenti condizioni:
 - **Mutua Esclusione (Mutual Exclusion):** Se un processo è nella sua sezione critica, nessun altro processo può entrare nella propria sezione critica.
 - **Progresso (Progress):** Se nessun processo è nella sezione critica e alcuni processi vogliono entrare, solo quei processi che non sono nella loro sezione remainder possono partecipare alla decisione di quale processo entrerà nella sezione critica successiva, e questa decisione non deve essere ritardata indefiniteamente.
 - **Attesa Limitata (Bounded Waiting):** Esiste un limite al numero di volte in cui altri processi possono entrare nelle loro sezioni critiche dopo che un processo ha richiesto di entrare nella sua sezione critica e prima che la sua richiesta venga soddisfatta. Questo previene la starvation.

1.5.2 Meccanismi di Sincronizzazione

Diversi strumenti e meccanismi sono stati sviluppati per risolvere il problema della sezione critica:

- **Lock:** Un meccanismo semplice che permette di acquisire e rilasciare un blocco su una risorsa. Prima di entrare nella sezione critica, un processo acquisisce il lock; dopo essere uscito, lo rilascia. Se il lock è già acquisito, il processo deve attendere.

- **Semafori:** Un tipo di variabile intera (generalmente non negativa) a cui si può accedere solo tramite due operazioni atomiche:
 - **wait()** (o **P()**): Decrementa il valore del semaforo. Se il valore diventa negativo, il processo viene bloccato.
 - **signal()** (o **V()**): Incrementa il valore del semaforo. Se ci sono processi bloccati sul semaforo, uno di essi viene sbloccato.
 - **Semafori Binari (Mutex Lock):** Valori 0 o 1. Usati per la mutua esclusione.
 - **Semafori Contatori:** Valori interi. Usati per gestire l'accesso a un numero finito di risorse.

- **Esempio di Problema Produttore-Consumatore con Semafori:** Un problema classico di sincronizzazione dove un produttore genera dati e li inserisce in un buffer, mentre un consumatore preleva dati dal buffer.

```

// Semafori:
// empty: contatore (inizializzato a N, dimensione buffer) -> numero di
// slot vuoti
// full: contatore (inizializzato a 0) -> numero di slot pieni
// mutex: binario (inizializzato a 1) -> mutua esclusione per accesso al
// buffer

Producer:
LOOP
  produce item
  CALL wait(empty)
  CALL wait(mutex)
  add item to buffer
  CALL signal(mutex)
  CALL signal(full)
END LOOP

Consumer:
LOOP
  CALL wait(full)
  CALL wait(mutex)
  remove item from buffer
  CALL signal(mutex)
  CALL signal(empty)
  consume item
END LOOP
  
```

- **Monitor:** Un costrutto di sincronizzazione di alto livello che incapsula dati condivisi e le procedure che li manipolano. Garantisce che, in ogni momento, al massimo un processo possa essere attivo all'interno del monitor. Utilizza **variabili di condizione** con operazioni ‘wait()’ e ‘signal()’ per sospendere e riattivare i processi che devono attendere specifiche condizioni sui dati condivisi.

1.5.3 Deadlock (Interblocco)

Il **deadlock** è una situazione in cui due o più processi sono bloccati indefinitamente, in attesa di una risorsa detenuta da un altro processo bloccato.

Condizioni Necessarie per il Deadlock (Condizioni di Coffman)

Il deadlock può verificarsi se e solo se tutte e quattro le seguenti condizioni sono presenti contemporaneamente:

1. **Mutua Esclusione (Mutual Exclusion)**: Almeno una risorsa deve essere tenuta in modalità non-condivisibile, cioè al massimo un processo può usarla alla volta.
2. **Attesa e Mantenimento (Hold and Wait)**: Un processo deve detenere almeno una risorsa ed essere in attesa di acquisirne altre attualmente detenute da altri processi.
3. **Non-Preemptive (Nessuna Preemption)**: Le risorse non possono essere sottratte a un processo che le detiene; possono essere rilasciate solo volontariamente dal processo che le detiene.
4. **Attesa Circolare (Circular Wait)**: Deve esistere una catena circolare di due o più processi, in cui ogni processo in attesa nella catena sta aspettando una risorsa detenuta dal processo successivo nella catena.

Strategie di Gestione del Deadlock

Per affrontare il deadlock, i sistemi operativi possono adottare diverse strategie:

- **Prevenzione del Deadlock (Deadlock Prevention)**:

- Obiettivo: Garantire che almeno una delle quattro condizioni necessarie non si verifichi.
- Come: Negando una o più condizioni (es. non permettere l'attesa e mantenimento, assegnare tutte le risorse all'inizio).
- Svantaggi: Spesso porta a un basso utilizzo delle risorse e a un throughput ridotto.

- **Evitamento del Deadlock (Deadlock Avoidance)**:

- Obiettivo: Richiede che il sistema abbia informazioni a priori sulle risorse che un processo richiederà. Il sistema verifica se lo stato corrente è "sicuro" (se esiste una sequenza di esecuzione dei processi che eviterà il deadlock).
- Algoritmo Esempio: Algoritmo del banchiere.
- Vantaggi: Meno restrittivo della prevenzione, migliore utilizzo delle risorse.
- Svantaggi: Richiede conoscenza a priori, può essere computazionalmente costoso.

- **Rilevamento e Ripristino del Deadlock (Deadlock Detection and Recovery)**:

- Obiettivo: Permettere che il deadlock si verifichi, rilevarlo e poi ripristinare il sistema da esso.
- Rilevamento: Si usa un algoritmo di rilevamento del ciclo nel grafo di allocazione delle risorse.
- Ripristino:

- * Terminazione del processo: Terminare uno o più processi coinvolti nel deadlock.
- * Preemption della risorsa: Sottrarre risorse a un processo e assegnarle a un altro.
- Svantaggi: Comporta un overhead per il rilevamento e la perdita di lavoro per il ripristino.

- **Ignorare il Problema:**

- Questo è l'approccio più comune in molti sistemi operativi (es. Unix/Linux, Windows), assumendo che il deadlock sia un evento raro e che sia meno costoso lasciarlo gestire all'amministratore (riavvio del sistema) piuttosto che implementare algoritmi complessi.

1.6 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. È una funzione fondamentale del sistema operativo che ha un impatto cruciale sulle performance complessive del sistema, influenzando parametri come il throughput (numero di processi completati per unità di tempo), il tempo di risposta (tempo tra richiesta e prima risposta), e l'equità nella distribuzione delle risorse della CPU tra i processi.

1.6.1 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, minimizzare il tempo di 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.

1.6.2 Esempi di Algoritmi di Scheduling

Diversi algoritmi sono stati sviluppati per affrontare le problematiche dello scheduling, ognuno con i propri compromessi tra efficienza, equità e complessità di implementazione.

First-Come, First-Served (FCFS)

- **Descrizione:** Non preemptive. I processi vengono eseguiti nell'ordine in cui arrivano nella coda dei processi pronti. Una volta che un processo ottiene la CPU, la tiene fino al completamento o fino a quando non esegue un'operazione di I/O.
- **Vantaggi:** Semplice da implementare e comprendere.
- **Svantaggi:**
 - **"Effetto Convoglio":** Un processo con un tempo di esecuzione molto lungo può bloccare tutti i processi successivi più brevi, aumentando notevolmente il tempo medio di attesa.
 - Tempo di risposta e throughput possono essere scarsi in scenari sfavorevoli.
- **Esempio:** Consideriamo i processi P1 (burst time 24), P2 (burst time 3), P3 (burst time 3) che arrivano nell'ordine P1, P2, P3.

```
Gantt Chart (FCFS):
| P1 (24) | P2 (3) | P3 (3) |
0          24          27          30
Tempo di attesa:
P1 = 0
P2 = 24
P3 = 27
Tempo medio di attesa = (0 + 24 + 27) / 3 = 17
```

Shortest-Job-First (SJF)

- **Descrizione:** Può essere preemptive (chiamato Shortest-Remaining-Time-First, SRTF) o non preemptive. Assegna la CPU al processo che ha il tempo di esecuzione stimato più breve.
- **Vantaggi:** Ottimale per minimizzare il tempo medio di attesa per un dato insieme di processi.
- **Svantaggi:**
 - **Difficoltà di stima:** È difficile conoscere a priori la durata esatta del "burst time" di un processo (si usano stime basate sulla storia passata).
 - **Starvation:** Processi lunghi potrebbero non essere mai eseguiti se arrivano continuamente processi più brevi.
- **Esempio (Non Preemptive):** Processi P1 (burst 7), P2 (burst 4), P3 (burst 1), P4 (burst 4). Arrivano quasi contemporaneamente.

```
Gantt Chart (SJF Non-Preemptive):
| P3 (1) | P2 (4) | P4 (4) | P1 (7) |
0          1          5          9          16
Tempo di attesa:
P3 = 0
P2 = 1
P4 = 5
```

P1 = 9 Tempo medio di attesa = $(0 + 1 + 5 + 9) / 4 = 3.75$
--

Priority Scheduling

- **Descrizione:** Può essere preemptive o non preemptive. Ad ogni processo viene assegnata una priorità (un numero intero, dove un numero più basso può indicare una priorità più alta, o viceversa). La CPU viene assegnata al processo con la priorità più alta.
- **Vantaggi:** Permette di prioritizzare lavori critici o importanti del sistema.
- **Svantaggi:**
 - **Starvation:** Processi a bassa priorità potrebbero non essere mai eseguiti se processi a priorità più alta arrivano continuamente.
 - **Soluzione per Starvation (Aging):** La priorità di un processo che aspetta da troppo tempo viene gradualmente aumentata.
- **Esempio:** Processi P1 (burst 10, priority 3), P2 (burst 1, priority 1), P3 (burst 2, priority 4), P4 (burst 1, priority 5), P5 (burst 5, priority 2). (Priorità più basse = priorità più alte).

Gantt Chart (Priority Scheduling, Non-Preemptive): P2 (1) P5 (5) P1 (10) P3 (2) P4 (1) 0 1 6 16 18 19 Tempo di attesa (processi ordinati per arrivo, non priorita'): P1 = 6 P2 = 0 P3 = 16 P4 = 18 P5 = 1 Tempo medio di attesa = $(6 + 0 + 16 + 18 + 1) / 5 = 8.2$
--

Round Robin (RR)

- **Descrizione:** Preemptive. Progettato per sistemi time-sharing. Ogni processo ottiene una piccola porzione di tempo di CPU, chiamata "quantum" (o time slice), solitamente da 10 a 100 millisecondi. Se un processo non finisce entro il quantum, viene preempted e messo in coda alla fine della coda dei processi pronti.
- **Vantaggi:**
 - **Equità:** Garantisce che nessun processo aspetti per un tempo eccessivamente lungo.
 - **Buon Tempo di Risposta:** Adatto per processi interattivi, dando l'impressione che tutti i processi siano eseguiti contemporaneamente.
- **Svantaggi:**
 - L'overhead del context switching aumenta se il quantum è troppo piccolo (troppe cambi di contesto).

- Le performance degradano se il quantum è troppo grande (il RR tende a comportarsi come FCFS).
- Può essere meno efficiente se i processi hanno durate molto diverse.

- **Esempio:** Processi P1 (burst 24), P2 (burst 3), P3 (burst 3). Quantum = 4.

```

Gantt Chart (Round Robin, Quantum=4):
| P1 (4) | P2 (3) | P3 (3) | P1 (4) | P1 (4) | P1 (4) | P1 (4) |
0        4        7        10       14       18       22       26       30
Tempo di attesa:
P1 = (4+3+3+3+3) = 16 (somma dei tempi non consecutivi)
P2 = 4
P3 = 7
Tempo medio di attesa = (16 + 4 + 7) / 3 = 9

```

Multilevel Queue Scheduling

- **Descrizione:** I processi sono divisi in diverse code, ognuna con il proprio algoritmo di scheduling. Le code possono avere priorità fisse tra di loro, oppure essere gestite con time slices diversi per ogni coda.
- **Esempio:** Coda foreground (processi interattivi) con Round Robin; Coda background (processi batch) con FCFS. I processi non si muovono tra le code.

Multilevel Feedback Queue Scheduling

- **Descrizione:** Permette ai processi di muoversi tra le code in base al loro comportamento. Se un processo usa troppo la CPU, viene spostato in una coda con priorità inferiore o un quantum più grande. Se un processo aspetta molto, può essere spostato in una coda con priorità più alta.
- **Vantaggi:** Altamente configurabile, può approssimare SJF senza conoscere i burst time, può prevenire la starvation (tramite l'aging), e offrire una buona risposta per processi interattivi.
- **Svantaggi:** Molto complesso da implementare e ottimizzare, richiede molti parametri (numero di code, algoritmo per ogni coda, quando spostare i processi tra code).

Domande e Esercizi

Domanda: Struttura e Organizzazione del Sistema Operativo, e Scheduling della CPU

Domanda: Il candidato fornisca una panoramica sulla struttura e organizzazione di un sistema operativo, descrivendo i principali componenti e modelli di sistemi operativi. Si approfondisca, inoltre, il tema dello scheduling della CPU, evidenziando le principali problematiche che questo comporta e illustrando esempi di algoritmi.

Risposta :

Un **sistema operativo (SO)** è un software di sistema che gestisce le risorse hardware e software di un computer, fungendo da interfaccia tra l'hardware e l'utente/applicazioni.

I suoi componenti principali includono:

- **Kernel:** Il cuore del SO, che gestisce processi, memoria, file system e I/O.
- **Gestore dei Processi:** Si occupa della creazione, terminazione e scheduling dei processi.
- **Gestore della Memoria:** Responsabile dell'allocazione, protezione e gestione della memoria virtuale.
- **File System Management:** Organizzazione dei dati su storage secondario.
- **Gestore I/O:** Interazione con i dispositivi hardware.
- **Network Management:** Gestisce le comunicazioni di rete. **Security and Protection:** Protezione delle risorse.

I modelli architettonici dei SO possono essere:

- **Monolitici**
- **Layered** (a strati)
- **Microkernel**
- **Modulari** (Ibridi)

Ciascuno con vantaggi e svantaggi in termini di performance, flessibilità e robustezza.

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. Le principali problematiche dello scheduling includono l'ottimizzazione di obiettivi contrastanti (throughput, tempo di risposta, tempo di attesa, equità), l'overhead del contesto switching, la starvation (processi a bassa priorità che non vengono eseguiti) e la gestione implicita di situazioni di deadlock. Diversi algoritmi di scheduling sono utilizzati:

- **First-Come, First-Served (FCFS):** Non preemptive; esegue i processi in ordine di arrivo. Semplice, ma soffre l'effetto convoglio.
- **Shortest-Job-First (SJF):** Può essere preemptive o non preemptive; esegue il processo con il burst time stimato più breve. Ottimale per tempo medio di attesa, ma difficile stimare la durata.

- **Priority Scheduling:** Assegna la CPU al processo con priorità più alta. Permette di prioritizzare lavori critici, ma può causare starvation (risolvibile con l'aging).
- **Round Robin (RR):** Preemptive; ogni processo ottiene un quantum di tempo. Equo e con buon tempo di risposta, ma con overhead di contesto switching.
- **Multilevel Queue Scheduling:** I processi sono divisi in diverse code, ognuna con il proprio algoritmo.
- **Multilevel Feedback Queue Scheduling:** Permette ai processi di muoversi tra le code per adattarsi al loro comportamento, bilanciando efficienza e fairness.

Domanda: Processi e Thread, Sincronizzazione e Deadlock

Domanda: Il Candidato introduca i concetti di Processo "pesante" e "leggero" (Thread), illustrandone i punti in comune e le differenze. Si illustrino poi brevemente i possibili meccanismi di sincronizzazione tra processi al fine di evitare deadlock, quali ad esempio il meccanismo di lock su risorse, i "semafori", etc. Il Candidato illustri poi i concetti tramite un breve esempio a sua scelta.

Risposta :

Processi ("Pesanti") e Thread ("Leggeri") Nei sistemi operativi moderni, l'esecuzione dei programmi è gestita attraverso i processi e i thread, che rappresentano unità di esecuzione con diverse caratteristiche di gestione delle risorse.

Un **processo** è un'istanza di un programma in esecuzione, che possiede il proprio spazio di indirizzamento virtuale isolato e risorse dedicate come codice, dati, stack, Program Counter (PC) e registri CPU, oltre a file aperti e segnali. I processi sono considerati "pesanti" a causa dell'overhead significativo nella loro creazione, distruzione e nel cambio di contesto. Questo isolamento garantisce che il crash di un processo non influenzi direttamente gli altri.

Un **thread** (o thread di esecuzione) è un'unità di esecuzione più "leggera" all'interno di un processo. Più thread possono esistere all'interno dello stesso processo e condividono lo stesso spazio di indirizzamento virtuale, il codice del programma, i dati globali e le risorse del sistema operativo (es. file aperti). Ogni thread, tuttavia, ha il proprio Program Counter, il proprio set di registri della CPU e uno stack separato. La natura "leggera" dei thread si traduce in una creazione, distruzione e un cambio di contesto molto più rapidi ed efficienti rispetto ai processi.

Punti in Comune:

- Sia i processi che i thread sono unità di esecuzione che possono essere schedulate dalla CPU.
- Entrambi mantengono un Program Counter, un set di registri e uno stack (anche se lo stack è separato per ogni thread).

Differenze Principali:

- **Isolamento delle Risorse:** I processi operano in spazi di indirizzamento separati e con risorse isolate, fornendo una maggiore protezione. I thread all'interno dello stesso processo condividono lo spazio di indirizzamento e le risorse, rendendo la comunicazione più semplice ma con meno isolamento.
- **Overhead:** I processi comportano un overhead maggiore per la gestione del loro ciclo di vita e il cambio di contesto. I thread sono più efficienti in queste operazioni.

- **Comunicazione:** La comunicazione tra processi (IPC) è più complessa e richiede meccanismi specifici. La comunicazione tra thread (ITC) è più diretta grazie alla memoria condivisa.
- **Robustezza:** Un errore critico in un thread può causare il crash dell'intero processo e di tutti i suoi thread. Un processo in crash solitamente non influisce sugli altri processi del sistema.

Meccanismi di Sincronizzazione e Deadlock Nei sistemi multi-programmati o multi-thread, la **sincronizzazione** è cruciale per prevenire **Race Condition** (risultati imprevedibili dovuti all'accesso concorrente a dati condivisi) e garantire la coerenza dei dati. La **sezione critica** è la porzione di codice in cui si accede a risorse condivise, e la mutua esclusione è l'obiettivo primario per garantire che un solo processo/thread vi acceda per volta.

I **meccanismi di sincronizzazione** includono:

- **Lock:** Strumenti che permettono di acquisire un blocco su una risorsa prima di accedervi e rilasciarlo al termine, forzando l'attesa di altri processi.
- **Semafori:** Variabili intere gestite da operazioni atomiche ‘wait()’ (decrementa e blocca se negativo) e ‘signal()’ (incrementa e sblocca se ci sono processi in attesa). Esistono semafori binari (mutex) per la mutua esclusione e semafori contatori per risorse multiple. **Esempio (Produttore-Consumatore con Semafori):** Un problema classico in cui un produttore aggiunge elementi a un buffer e un consumatore li preleva, utilizzando semafori (‘empty’ per slot vuoti, ‘full’ per slot pieni, ‘mutex’ per l’accesso al buffer) per coordinare l’accesso e prevenire la sovrascrittura/lettura di dati non validi.

```
// Semafori: empty (N), full (0), mutex (1)

Producer:
LOOP
    produce item
    CALL wait(empty)
    CALL wait(mutex)
    add item to buffer
    CALL signal(mutex)
    CALL signal(full)
END LOOP

Consumer:
LOOP
    CALL wait(full)
    CALL wait(mutex)
    remove item from buffer
    CALL signal(mutex)
    CALL signal(empty)
    consume item
END LOOP
```

- **Monitor:** Costrutti di alto livello che encapsulano dati condivisi e le procedure che li manipolano, garantendo l’accesso esclusivo tramite variabili di condizione (‘wait()’ e ‘signal()’) per la sospensione/attivazione dei processi.

Il **deadlock** (interblocco) si verifica quando due o più processi sono bloccati indefinitamente, ciascuno in attesa di una risorsa detenuta da un altro processo bloccato. Le **quattro condizioni necessarie** per il deadlock sono:

1. **Mutua Esclusione:** Risorse non condivisibili.
2. **Attesa e Mantenimento:** Un processo detiene risorse e ne attende altre.
3. **Non-Preemptive:** Le risorse non possono essere sottratte forzatamente.
4. **Attesa Circolare:** Esiste una catena circolare di dipendenze tra processi.

Le **strategie di gestione del deadlock** includono:

- **Prevenzione:** Negare una o più delle condizioni necessarie (es. richiedere tutte le risorse all'inizio).
- **Evitamento:** Utilizzare informazioni a priori per decidere se uno stato è "sicuro" prima di allocare risorse (es. Algoritmo del Banchiere).
- **Rilevamento e Ripristino:** Permettere il deadlock, rilevarlo tramite algoritmi di ricerca di cicli e poi ripristinare il sistema (es. terminando processi o preemption di risorse).
- **Ignorare il Problema:** Assumere che il deadlock sia raro e gestirlo manualmente (es. riavvio del sistema).

Domanda: Record di Attivazione

Domanda: Il Candidato illustri il concetto di record di attivazione, descriva cosa esso contenga ed il suo ciclo di vita. Presenti inoltre un esempio di record di attivazione per il caso di una funzione "pow2" che riceve in input un solo parametro "x" e lo ritorna elevato al quadrato.

Risposta :

Concetto di Record di Attivazione Il **Record di Attivazione** (o *Stack Frame*) è una struttura dati fondamentale creata sullo stack di esecuzione di un processo ogni volta che una funzione (o procedura, o subroutine) viene chiamata. La sua funzione principale è contenere tutte le informazioni necessarie per la gestione dell'esecuzione di quella specifica chiamata di funzione, fornendo il contesto per il suo funzionamento e il suo corretto ritorno al punto di chiamata.

Contenuto di un Record di Attivazione Un record di attivazione è una struttura complessa la cui esatta composizione può variare leggermente a seconda dell'architettura della CPU, del sistema operativo e del compilatore, ma tipicamente include:

- **Parametri Attuali:** I valori degli argomenti che vengono passati alla funzione durante la sua chiamata.
- **Indirizzo di Ritorno:** L'indirizzo di memoria dell'istruzione nel codice della funzione chiamante a cui il controllo del programma deve essere trasferito una volta che la funzione corrente ha terminato la sua esecuzione.
- **Valore di Ritorno:** Uno spazio riservato per memorizzare il risultato (se la funzione restituisce un valore) che verrà poi recuperato dalla funzione chiamante.

- **Variabili Locali:** Le variabili dichiarate all'interno del corpo della funzione. Queste variabili hanno scope e durata limitati all'attivazione corrente della funzione.
- **Stato dei Registri Salvati:** I valori dei registri della CPU che erano in uso dalla funzione chiamante e che vengono salvati per essere ripristinati al ritorno, garantendo che lo stato della CPU non sia corrotto dalla funzione chiamata.
- **Puntatore al Frame Precedente (Control Link / Dynamic Link):** Un puntatore all'indirizzo del record di attivazione della funzione che ha effettuato la chiamata. Questo permette di "risalire" lo stack e ripristinare il contesto della funzione chiamante.
- **Puntatore al Contesto Statico (Access Link / Static Link):** Un puntatore al record di attivazione della funzione che definisce lo scope lessicale della funzione corrente (rilevante in linguaggi con scope annidato statico, come Pascal, per accedere a variabili non locali).

Ciclo di Vita di un Record di Attivazione Il ciclo di vita di un record di attivazione è strettamente legato alla dinamica delle chiamate e dei ritorni delle funzioni, seguendo una logica LIFO (Last-In, First-Out) tipica delle strutture a stack:

1. **Creazione (Chiamata di Funzione):** Quando un processo chiama una funzione, un nuovo record di attivazione viene creato (popolato con le informazioni rilevanti) e "pushed" (inserito) in cima allo stack di esecuzione del processo. Il puntatore dello stack viene aggiornato per puntare a questo nuovo frame. Il controllo viene poi passato all'inizio della funzione chiamata.
2. **Esecuzione:** La funzione utilizza i dati e le risorse definite all'interno del suo record di attivazione (parametri, variabili locali) per eseguire le sue operazioni. Durante l'esecuzione, può a sua volta chiamare altre funzioni, che a loro volta creeranno nuovi record di attivazione sul top dello stack.
3. **Terminazione (Ritorno da Funzione):** Una volta che la funzione completa la sua esecuzione (sia raggiungendo un'istruzione 'return' che la fine del suo blocco di codice), il valore di ritorno (se presente) viene posizionato in un registro designato. Il record di attivazione corrente viene quindi "popped" (rimosso) dalla cima dello stack, liberando lo spazio di memoria che occupava.
4. **Ripristino del Contesto:** Il sistema operativo (o il runtime) utilizza l'indirizzo di ritorno salvato nel record appena rimosso per trasferire il controllo alla funzione chiamante, e ripristina lo stato dei registri della CPU per consentire alla funzione chiamante di riprendere l'esecuzione dal punto in cui era stata interrotta.

Esempio di Record di Attivazione per la funzione "pow2(x)" Consideriamo una semplice funzione 'pow2(x)' che calcola il quadrato del suo input 'x'.

```
FUNCTION pow2(x):
    RETURN x * x
END FUNCTION
```

Listing 1.1: Funzione pow2(x)

Quando la funzione 'pow2(5)' viene chiamata, viene creato il seguente record di attivazione (semplificato) sullo stack:

- **Parametri Attuali:**

- ‘x’: 5

- **Indirizzo di Ritorno:** L’indirizzo nel codice della funzione chiamante da cui ‘pow2(5)’ è stata invocata (es. ‘0x00A0’ nel chiamante).
- **Valore di Ritorno:** Spazio per il risultato (es. 25).
- **Variabili Locali:** Nessuna in questo esempio specifico (o temporanee usate dal compilatore).
- **Stato dei Registri Salvati:** Valori dei registri che devono essere preservati per la funzione chiamante.
- **Puntatore al Frame Precedente:** Indirizzo del record di attivazione della funzione chiamante.

Ciclo di Vita Semplificato per ‘pow2(5)’:

1. **Chiamata:** La funzione chiamante spinge i parametri e l’indirizzo di ritorno sullo stack. Viene creato e spinto il record di attivazione per ‘pow2(5)’.
2. **Esecuzione:** ‘pow2’ prende il valore di ‘x’ (5), calcola ‘ $5 * 5 = 25$ ’.
3. **Ritorno:** Il valore 25 viene posizionato dove la funzione chiamante lo recupererà. Il record di attivazione di ‘pow2’ viene rimosso dallo stack.
4. **Ripristino:** Il controllo torna all’indirizzo di ritorno nella funzione chiamante, che continua la sua esecuzione.

Questo illustra come il record di attivazione gestisce il contesto di una singola invocazione di funzione, permettendo l’esecuzione modulare dei programmi.

Altre Possibili Domande

Domanda: Come la Memoria Virtuale Migliora la Gestione della Memoria nei Sistemi Operativi?

Domanda: Come la memoria virtuale migliora la gestione della memoria nei sistemi operativi?

Risposta:

La **memoria virtuale** è una tecnica del sistema operativo che permette di usare lo spazio su disco (memoria secondaria) per simulare una RAM maggiore, consentendo ai programmi di utilizzare più memoria di quella fisica disponibile. Questo facilita il multitasking isolando lo spazio di indirizzamento di ogni processo, aumentandone la protezione e la flessibilità. Le tecniche principali per implementare la memoria virtuale sono la paginazione e la segmentazione.

- **Paginazione:** Divide lo spazio logico in "pagine" di dimensione fissa e la memoria fisica in "frame", mappando le pagine ai frame tramite una Page Table. Questo elimina la frammentazione esterna ma introduce quella interna.
- **Segmentazione:** Vede la memoria come segmenti di dimensione variabile, corrispondenti a unità logiche del programma, facilitando protezione e condivisione ma soffrendo di frammentazione esterna.

Per l'allocazione della memoria (per blocchi contigui), il sistema operativo utilizza algoritmi come **First Fit** (alloca il primo blocco sufficiente), **Best Fit** (alloca il blocco più piccolo sufficiente) e **Worst Fit** (alloca il blocco più grande sufficiente), ognuno con diversi compromessi in termini di velocità e frammentazione.

Domanda: Quali sono i Problemi Principali Legati alla Sincronizzazione dei Processi e Come Vengono Gestiti i Deadlock?

Domanda: Quali sono i problemi principali legati alla sincronizzazione dei processi e come vengono gestiti i deadlock?

Risposta:

Nei sistemi multi-programmati o multi-thread, la **sincronizzazione** è cruciale per evitare "**Race Condition**", dove il risultato delle operazioni su dati condivisi dipende dall'ordine di esecuzione, portando a risultati imprevedibili. La "**Sezione Critica**" è una porzione di codice in cui un processo accede a risorse condivise, e l'obiettivo è garantire la "**Mutua Esclusione**" (solo un processo alla volta nella sezione critica), il "**Progresso**" (decisione non ritardata indefiniteamente) e l'"**Attesa Limitata**" (nessuna starvation). **Meccanismi di sincronizzazione** includono:

- **Lock:** Meccanismi semplici per acquisire e rilasciare un blocco su una risorsa.
- **Semafori:** Variabili intere gestite da operazioni atomiche 'wait()' e 'signal()', usate per mutua esclusione (semafori binari/mutex) o controllo di risorse (semafori contatori).
- **Monitor:** Costrutti di alto livello che encapsulano dati condivisi e procedure, garantendo l'accesso esclusivo tramite variabili di condizione.

Il "**Deadlock**" (interblocco) si verifica quando due o più processi sono bloccati indefiniteamente in attesa di risorse detenute da altri processi bloccati. Le **quattro condizioni necessarie** per il deadlock sono:

1. **Mutua Esclusione** (risorse non condivisibili).
2. **Attesa e Mantenimento** (un processo detiene risorse e ne attende altre).
3. **Non-Preemption** (le risorse non possono essere sottratte forzatamente).
4. **Attesa Circolare** (catena circolare di dipendenze).

Le **strategie per gestire il deadlock** includono:

- **Prevenzione:** Negare una o più delle condizioni necessarie (es. richiedere tutte le risorse all'inizio).
- **Evitamento:** Utilizzare informazioni a priori per decidere se uno stato è "sicuro" prima di allocare risorse (es. Algoritmo del Banchiere).
- **Rilevamento e Ripristino:** Permettere il deadlock, rilevarlo tramite algoritmi e poi ripristinare il sistema (es. terminando processi o sottraendo risorse).
- **Ignorare il Problema:** Assumere che il deadlock sia raro e gestirlo manualmente (es. riavvio del sistema).

Domanda: Come Funziona lo Scheduling della CPU e Quali Sono gli Algoritmi Più Comuni?

Domanda: Come funziona lo scheduling della CPU e quali sono gli algoritmi più comuni?

Risposta:

Lo **scheduling della CPU** è il processo di selezione del prossimo processo da assegnare alla CPU tra quelli pronti, con l'obiettivo di ottimizzare throughput, tempo di risposta ed equità. Le problematiche includono l'overhead del context switching, la starvation (processi a bassa priorità mai eseguiti) e la dipendenza dall'I/O. Esempi di algoritmi di scheduling includono:

- **First-Come, First-Served (FCFS):** Non-preemptive, i processi vengono eseguiti nell'ordine di arrivo. Semplice ma può soffrire dell'"Effetto Convoglio".
- **Shortest-Job-First (SJF):** Assegna la CPU al processo con il tempo di esecuzione stimato più breve (può essere preemptive come SRTF). Ottimale per minimizzare il tempo medio di attesa ma difficile da stimare e può causare starvation.
- **Priority Scheduling:** Assegna la CPU al processo con priorità più alta. Vantaggioso per lavori critici ma soggetto a starvation, mitigabile con l'Aging.
- **Round Robin (RR):** Preemptive, ogni processo ottiene un piccolo "quantum" di CPU. Garantisce equità e buon tempo di risposta per processi interattivi, ma l'overhead del context switching aumenta con quantum piccoli.
- **Multilevel Queue Scheduling e Multilevel Feedback Queue Scheduling:** Dividono i processi in code con algoritmi e priorità diverse, con l'ultimo che permette ai processi di spostarsi tra le code per prevenire starvation e ottimizzare la risposta.

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 Base e Rappresentazione Grafica

I principali componenti del Modello ER sono le entità, gli attributi e le relazioni, ognuno con una specifica rappresentazione grafica che ne facilita la visualizzazione.

- **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". Nel diagramma ER, gli attributi sono rappresentati con un ovale.
- **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.

2.1.2 Dettagli degli Attributi e Identificatori

Gli attributi possono presentare diverse caratteristiche e alcuni di essi assumono un ruolo speciale come identificatori.

- **Tipi di Attributi:**
 - **Attributi Semplici:** Non possono essere scomposti (es. "Età").
 - **Attributi Composti:** Formati da più attributi (es. "Indirizzo" composto da "Via", "Civico", "Città").

- **Attributi Mono-valore:** Hanno un singolo valore per istanza (es. "DataNascita").
- **Attributi Multi-valore:** Possono 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.

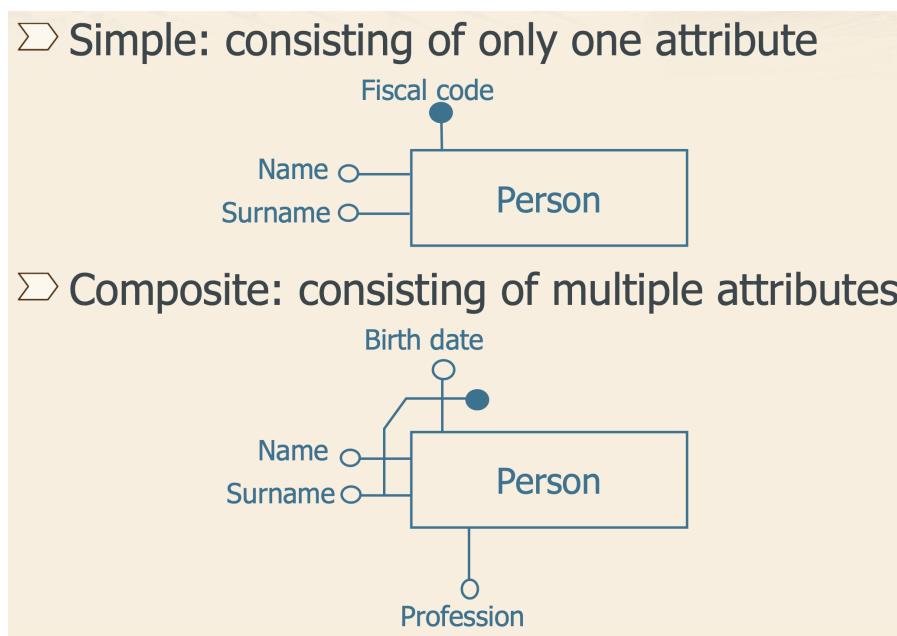


Figura 2.1: Esempi di rappresentazione di attributi semplici (es. Codice Fiscale) e attributi composti (es. Data di Nascita) in un Diagramma ER.

Identifieri

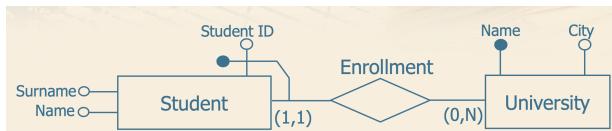
Gli identifieri permettono di distinguere univocamente le istanze di un'entità. Possono essere interni o esterni.

- **Identificatore Interno:** Costituito da uno o più attributi dell'entità stessa (es. Codice Fiscale per Persona).
- **Identificatore Esterno:** Coinvolge attributi dell'entità e la partecipazione a relazioni con altre entità, necessarie per la sua identificazione. È tipico delle entità deboli.

2.1.3 Dettagli delle Relazioni e Cardinalità

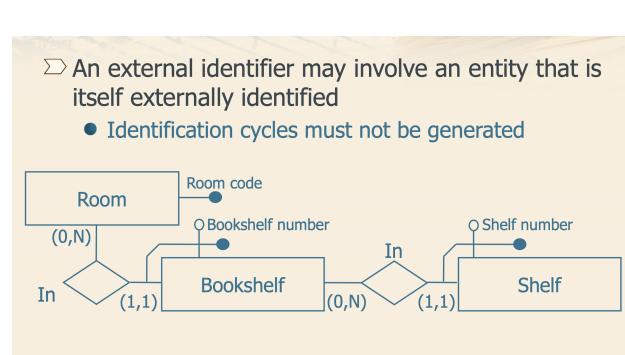
Le relazioni specificano le associazioni tra le entità, e le cardinalità definiscono il numero di istanze di un'entità che possono partecipare a una relazione.

- **Cardinalità delle Relazioni:** Definisce il numero minimo e massimo di istanze di un'entità che possono essere associate a un'istanza dell'altra entità nella relazione.
 - **Minima:** Indica se la partecipazione è obbligatoria (1) o opzionale (0).



- ▷ One entity that does not have sufficient internal attributes able to define an identifier is called **weak entity**.
- ▷ A weak entity must participate with cardinality (1,1) in each of the relationships that provide part of the identifier

(a) Identificatore Esterno per Entità Deboli (Studente, Immatricolazione, Università)



(b) Identificatore Esterno Complesso (Stanza, Scaffale, Ripiano)

Figura 2.2: Esempi di utilizzo di identificatori esterni, inclusi scenari con entità deboli e concatenazione di identificazione.

- **Massima:** Indica il numero massimo di partecipazioni (1 o N).
- **Notazione (min, max):** Es. (0, N) per zero a molti, (1, N) per uno a molti.

- **Partecipazione (o Dipendenza):**

- **Totale (o Obbligatoria):** Ogni istanza dell'entità deve partecipare alla relazione (indicata graficamente da una doppia linea di connessione).
- **Parziale (o Opzionale):** Un'istanza dell'entità può partecipare o meno alla relazione (indicata graficamente da una singola linea).

2.1.4 Generalizzazione nel Modello ER

La **Generalizzazione** è un costrutto del modello ER che permette di rappresentare una relazione "è un tipo di" (is-a) tra un'entità genitore (generale) e una o più entità figlie (specializzate).

- Ogni istanza di un'entità figlia è anche un'istanza dell'entità genitore.
- Le proprietà (attributi, identificatori, relazioni) dell'entità genitore sono ereditate da tutte le entità figlie.

2.1.5 Esempi Complessi di Schemi ER

Per consolidare la comprensione del Modello ER, è utile analizzare esempi complessi che integrano vari concetti come entità, attributi, relazioni con cardinalità diverse, identificatori, generalizzazioni e entità deboli. Questi schemi rappresentano contesti reali e mostrano come i costrutti ER vengono applicati per modellare domini complessi.

2.1.6 Esempi Complessi di Schemi ER

Per consolidare la comprensione del Modello ER, è utile analizzare esempi complessi che integrano vari concetti come entità, attributi, relazioni con cardinalità diverse, identificatori, generalizzazioni e entità deboli. Questi schemi rappresentano contesti reali e mostrano come i costrutti ER vengono applicati per modellare domini complessi.

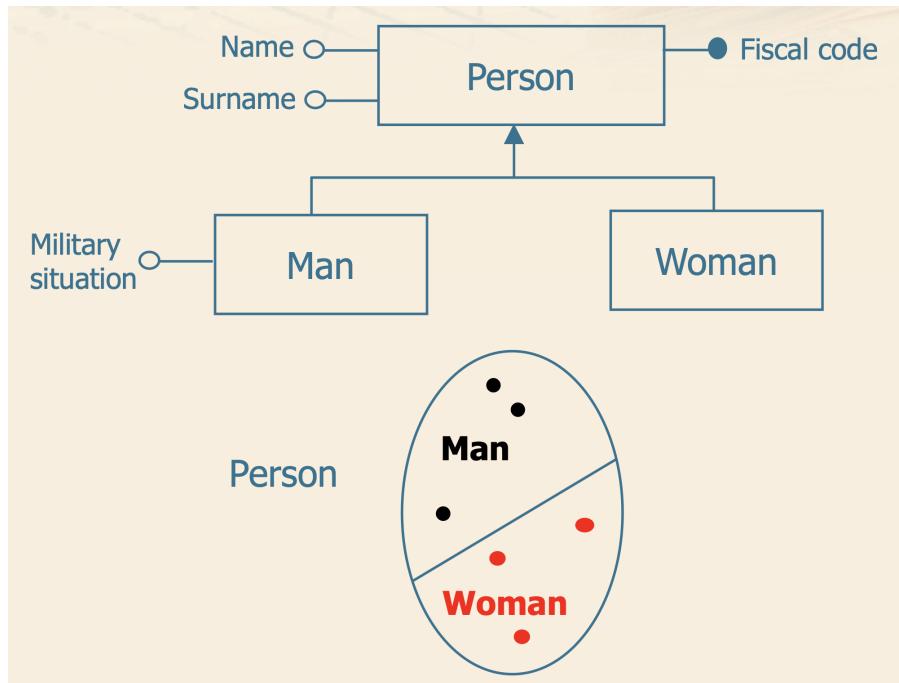


Figura 2.3: Esempio di Generalizzazione nel Diagramma ER: l'entità Persona si generalizza nelle entità Uomo e Donna, mostrando la distinzione e l'ereditarietà delle proprietà.

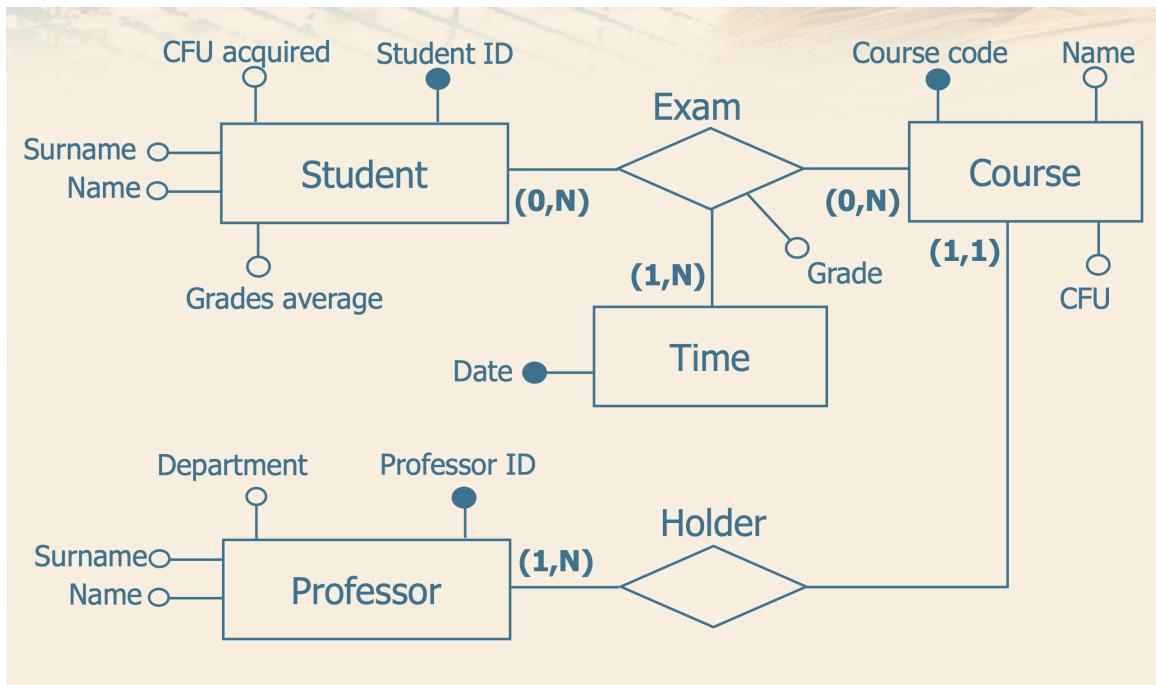


Figura 2.4: Diagramma ER che modella le relazioni tra Studenti, Corsi ed Esami, inclusi attributi e cardinalità.

Esercizio 2: Sistema Cinema/Film

Descrizione: Si consideri uno schema Entità-Relazione che rappresenta un sistema per la gestione di film, artisti, cinema e città. Il diagramma include entità come Film, Artista, Cinema e Città, con relazioni che specificano la regia, l'interpretazione e la proiezione dei film nei cinema locali.

Schema Relazionale corrispondente:

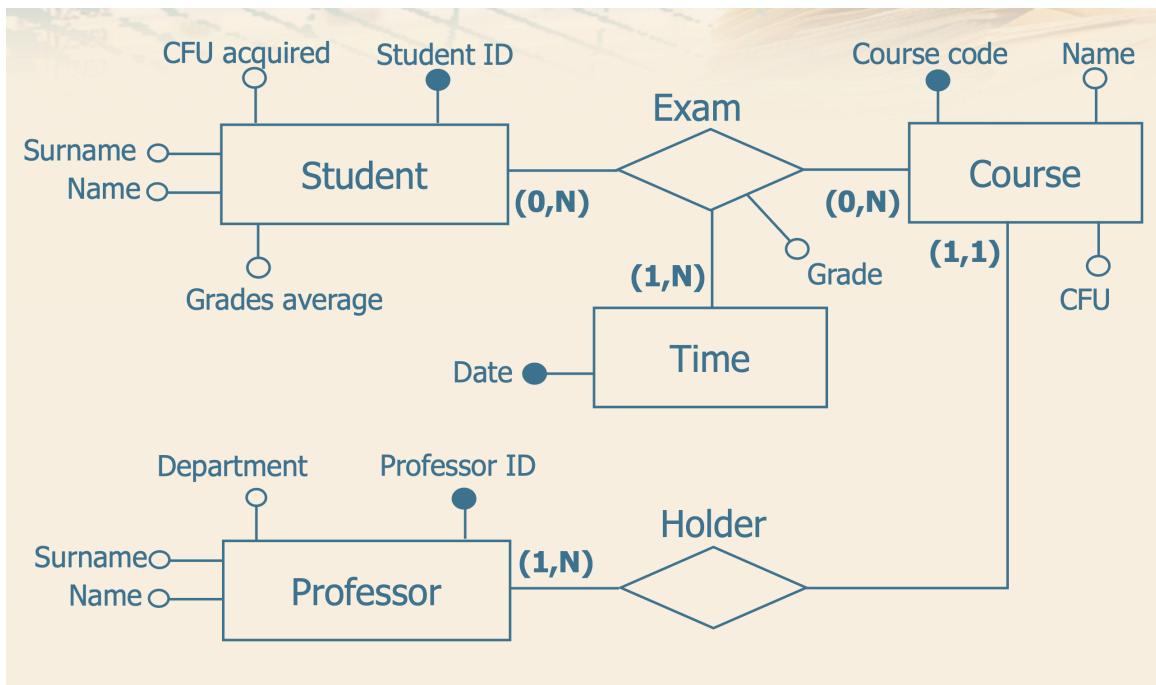


Figura 2.5: Diagramma ER che modella le relazioni tra Studenti, Corsi ed Esami, inclusi attributi e cardinalità.

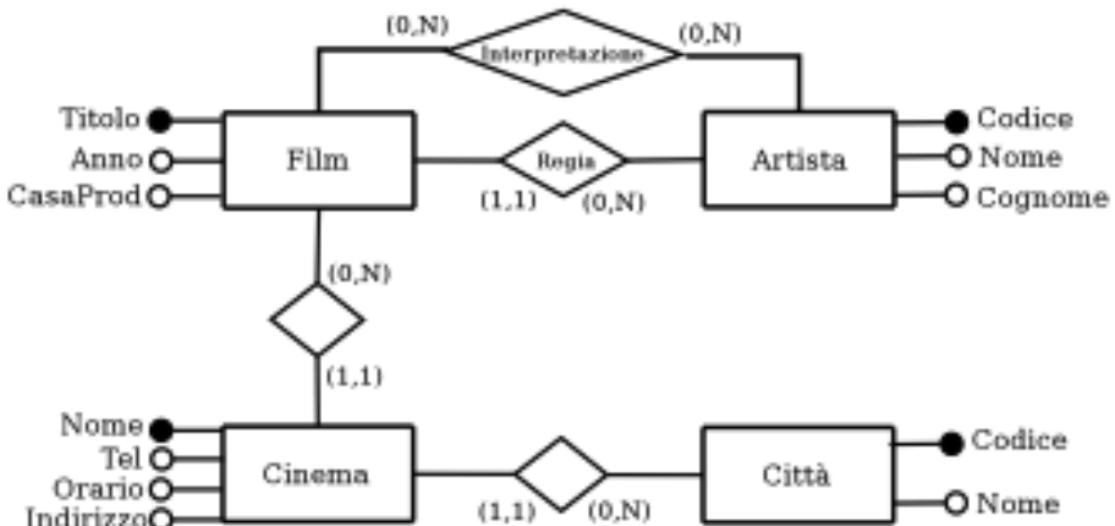


Figura 2.6: Schema ER per l'esercizio 2: Sistema Cinema/Film.

- 1 ARTISTI (Codice, Nome, Cognome)
- 2 FILM (Titolo, Anno, CasaProduttrice, RegistaArtista) -- *RegistaArtista FK a ARTISTI*
- 3 INTERPRETAZIONI (FilmTitolo, FilmAnno, ArtistaCodice) -- *FK a FILM, FK a ARTISTI*
- 4 CITTA (Codice, Nome)
- 5 CINEMA (Nome, Tel, Orario, Indirizzo, CittaCodice) -- *CittaCodice FK a CITTA*
- 6 PROIEZIONI (FilmTitolo, FilmAnno, CinemaNome, OrarioCinema) -- *FK a FILM, FK a CINEMA*

Esercizio 7: Sistema Campionato di Calcio

Descrizione: Modellare un sistema per la gestione di un campionato di calcio. Il sistema deve rappresentare squadre, giocatori (con i loro ruoli e dati personali), arbitri, giornate e partite, inclusi i risultati e le specificità come partite in campo neutro o rinviate.

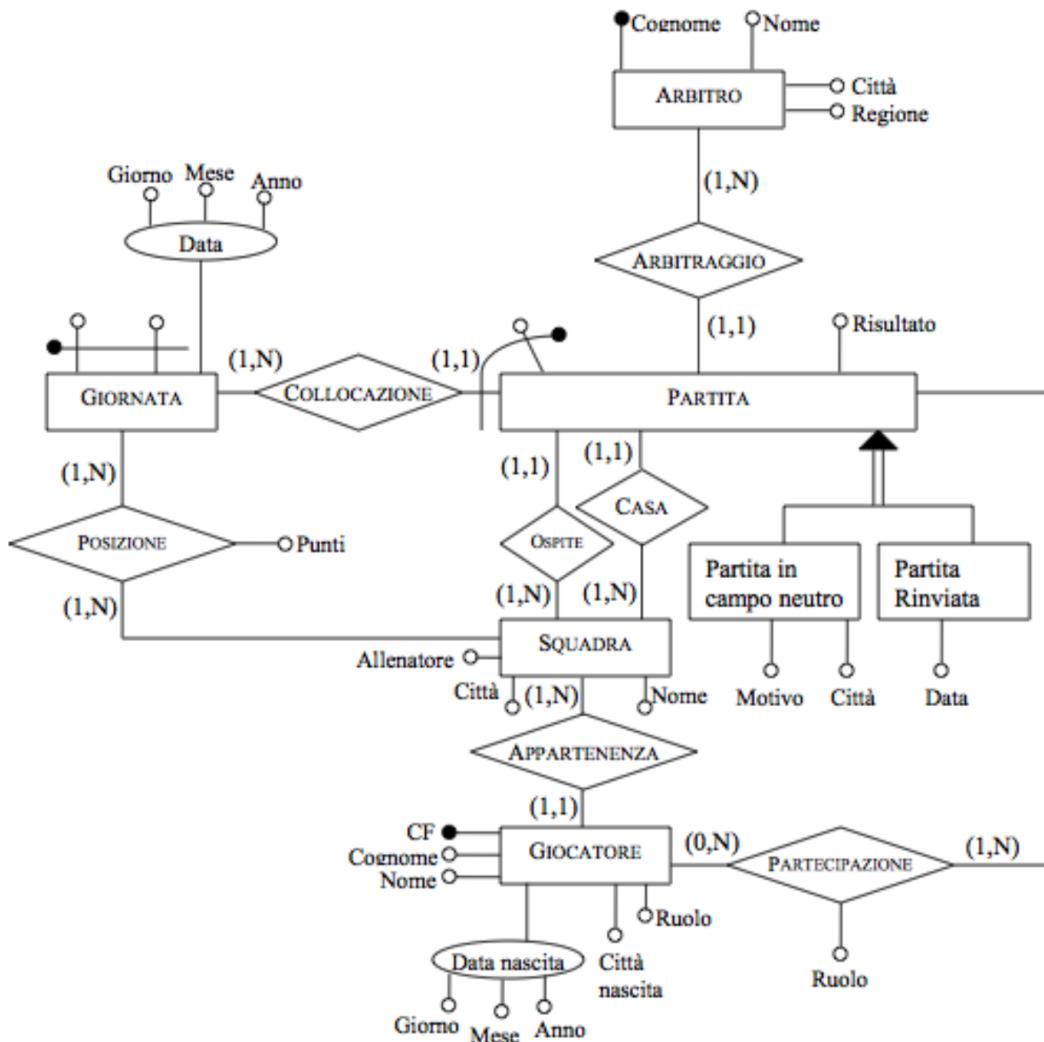


Figura 2.7: Schema ER per l'esercizio 7: Sistema Campionato di Calcio.

Schema Relazionale corrispondente:

- 1 ARBITRO (Cognome, Nome, Città, Regione) -- *Chiave primaria (Cognome, Nome) o ID_Arbitro generato*
- 2 GIORNATA (Numero, Serie, Giorno, Mese, Anno) -- *Chiave primaria (Numero, Serie)*
- 3 SQUADRA (Nome, Città, Allenatore) -- *Chiave primaria (Nome)*
- 4 GIOCATORE (CodiceFiscale, Cognome, Nome, Ruolo, CittaDiNascita) -- *Chiave primaria (CodiceFiscale)*
- 5 PARTITA (NumeroPartita, GiornoGiornata, MeseGiornata, AnnoGiornata, Risultato, ArbitroCognome, ArbitroNome, CasaSquadra, OspiteSquadra) -- *Chiave primaria (NumeroPartita), FK a GIORNATA, FK a ARBITRO, FK a SQUADRA (due volte)*
- 6 PARTITA_IN_CAMPO_NEUTRO (PartitaNumero, PartitaGiorno, PartitaMese, PartitaAnno, Motivo, CittaCampoNeutro) -- *FK a PARTITA*
- 7 PARTITA_RINVIATA (PartitaNumero, PartitaGiorno, PartitaMese, PartitaAnno, DataRinvio) -- *FK a PARTITA*

- 8 POSIZIONE (SquadraNome, GiornoGiornata, MeseGiornata, AnnoGiornata, Punteggio)
-- Chiave primaria (SquadraNome, GiornoGiornata, MeseGiornata, AnnoGiornata), FK a SQUADRA, FK a GIORNATA
- 9 APPARTENENZA (GiocatoreCodiceFiscale, SquadraNome, DataInizio, RuoloPrincipale, DataFine) -- Chiave primaria (GiocatoreCodiceFiscale, SquadraNome, DataInizio), FK a GIOCATORE, FK a SQUADRA

Esercizio 11: Sistema Gare Ciclistiche

Descrizione: Un sistema per la gestione di gare ciclistiche, includendo informazioni su ciclisti, squadre, località, competizioni, edizioni e tappe, con dettagli sulle classifiche e gli orari.

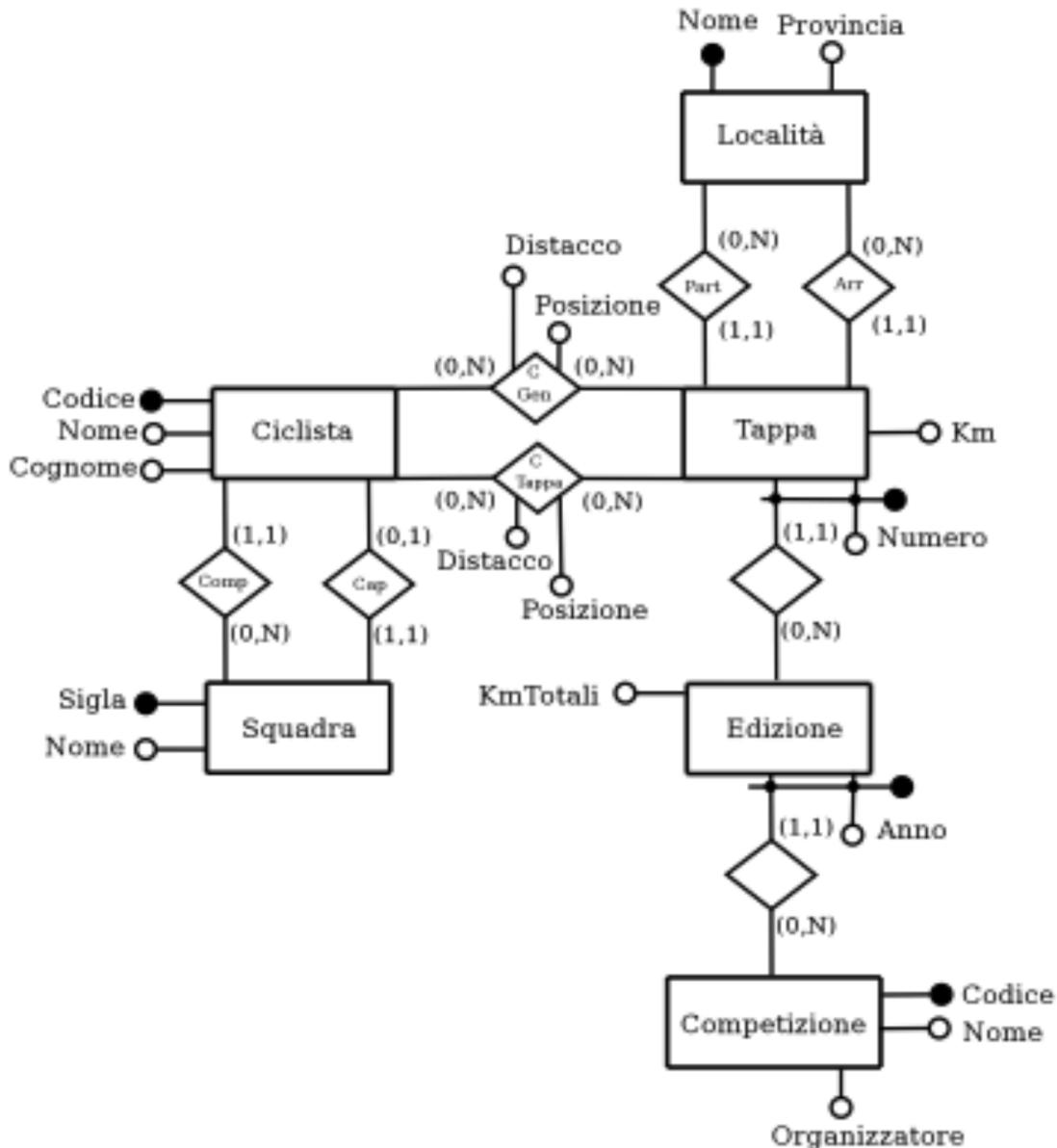


Figura 2.8: Schema ER per l'esercizio 11: Sistema Gare Ciclistiche.

Schema Relazionale corrispondente:

- 1 SQUADRE (Sigla, Nome, CapitanoCodice) -- CapitanoCodice FK a CICLISTI
- 2 CICLISTI (Codice, Cognome, Nome, SquadraSigla) -- SquadraSigla FK a SQUADRE

```

3 LOCALITA (Nome, Provincia) -- Chiave primaria (Nome, Provincia)
4 COMPETIZIONI (Codice, Nome, Organizzatore) -- Chiave primaria (Codice)
5 EDIZIONI (AnnoEdizione, CompetizioneCodice, KmTotali) -- Chiave primaria (
   AnnoEdizione, CompetizioneCodice), FK a COMPETIZIONI
6 TAPPE (NumeroTappa, AnnoEdizione, CompetizioneCodice, LocPartenzaNome,
   LocPartenzaProvincia, LocArrivoNome, LocArrivoProvincia) -- Chiave primaria
   (NumeroTappa, AnnoEdizione, CompetizioneCodice), FK a EDIZIONI, FK a
   LOCALITA (due volte)
7 CLASSIFICATAPPA (NumeroTappa, AnnoEdizione, CompetizioneCodice, CiclistaCodice,
   Posizione, Distacco) -- FK a TAPPE, FK a CICLISTI
8 CLASSIFICAGENERALE (NumeroTappa, AnnoEdizione, CompetizioneCodice,
   CiclistaCodice, Posizione, Distacco) -- FK a TAPPE, FK a CICLISTI

```

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 Perché la Normalizzazione è Importante

La normalizzazione è cruciale per diversi motivi:

- **Riduzione della Ridondanza:** Evitare la duplicazione inutile dei dati, che spreca spazio e può portare a incongruenze. La ridondanza può causare spreco di memoria a causa della memorizzazione multipla della stessa informazione.
- **Miglioramento dell'Integrità e Coerenza dei Dati:** Assicurare che i dati siano accurati e consistenti. Si ha incoerenza quando lo stesso campo ha valori diversi in tabelle diverse, fenomeno che può verificarsi quando le tabelle non sono aggiornate o l'aggiornamento non è stato effettuato correttamente.
- **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.
- **Efficienza del Database:** Una struttura di database normalizzata consente un accesso più rapido ed efficiente ai dati, migliorando le prestazioni di query e operazioni di recupero.
- **Flessibilità e Manutenibilità:** Rende il database più facile da comprendere, gestire e modificare.

2.2.2 Forme Normali: Livelli di Normalizzazione

Esistono vari livelli di normalizzazione, noti come "forme normali", che certificano la qualità dello schema del database. Ogni forma normale definisce un insieme di regole che lo schema del database deve rispettare per garantire un livello crescente di integrità e coerenza.

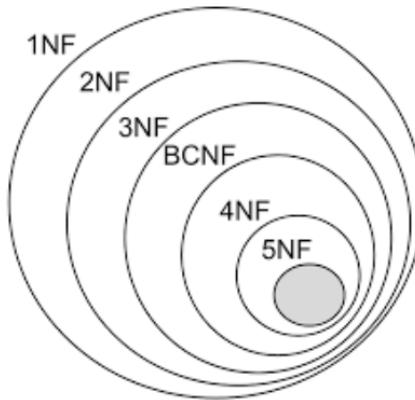


Figura 2.9: Diagramma di Venn delle Forme Normali (1NF, 2NF, 3NF, BCNF, 4NF, 5NF), che mostra la loro relazione gerarchica.

Prima Forma Normale (1NF)

Una relazione è in **Prima Forma Normale (1NF)** se e solo se:

- Non presenta attributi multipli (non ci sono gruppi ripetitivi di attributi all'interno di una singola tabella). Tutti gli attributi sono atomici (indivisibili).
- Esiste una chiave primaria (un insieme di attributi che identifica in modo univoco ogni tupla della relazione).

Esempio di Violazione e Normalizzazione a 1NF: Consideriamo una tabella "Voti" dove un singolo campo contiene più valori, come i voti di uno studente per semestri diversi in una stessa riga. Nella tabella non è in 1NF, il campo "Voto" contiene più voti e semestri. Per normalizzare, si scomponete la riga in più righe, una per ogni voto/semestre, assicurando che ogni attributo sia atomico.

Seconda Forma Normale (2NF)

Una relazione si dice in **Seconda Forma Normale (2NF)** quando è in 1NF e tutti i campi non-chiave dipendono funzionalmente dall'intera chiave primaria, e non solo da una sua parte. La 2NF elimina la dipendenza parziale degli attributi dalla chiave e riguarda relazioni con chiavi composte (formate da più attributi). **Esempio di Violazione e Normalizzazione a 2NF:** Consideriamo una relazione con chiave composta (es. Codice_atleta, Numero_gara) e un attributo non-chiave (Nome_atleta) che dipende solo da una parte della chiave (Codice_atleta). La soluzione è scomporre la relazione originale in due relazioni, una per la parte della chiave che determina l'attributo (es. Atleti(Codice_atleta, Nome_atleta)) e un'altra per la chiave completa con gli altri attributi (es. Risultati(Codice_atleta, Numero_gara, Tempo)).

non in 1NF***in 1NF***

Voti				
<u>Matricola</u>	<u>Studente</u>	<u>Materia</u>	<u>Voto</u>	
0000-000-01	Pietro	Basi di Dati	1 sem, B ; 2 sem, F	
0000-000-02	Pietro	Basi di Dati	1 sem, A ; 2 sem, A	
0000-000-03	Sara	Basi di Dati	1 sem, B ; 2 sem, A	

Voti				
<u>Matricola</u>	<u>Studente</u>	<u>Materia</u>	<u>Semestre</u>	<u>Voto</u>
0000-000-01	Pietro	Basi di Dati	1	B
0000-000-01	Pietro	Basi di Dati	2	F
0000-000-02	Pietro	Basi di Dati	1	A
0000-000-02	Pietro	Basi di Dati	2	A
0000-000-03	Sara	Basi di Dati	1	B
0000-000-03	Sara	Basi di Dati	2	A

Figura 2.10: Esempio di una tabella non in 1NF e la sua normalizzazione a 1NF, eliminando l'attributo multi-valore.

Codice_atleta	Numero_gara	Nome_atleta	tempo
3455	1	Maurizio	36
3455	2	Maurizio	32
3320	1	Luigi	38
3320	2	Luigi	39
3320	3	Luigi	33

la **chiave è composta**, in quanto il solo codice non basta per identificare il tempo che è realizzato in gare diverse.
Nome_atleta dipende solo dall'attributo Codice_atleta (**solo da una parte della chiave**)

Codice_atleta	Numero_gara	tempo
3455	1	36
3455	2	32
3320	1	38
3320	2	39
3320	3	33

Codice atleta	Nome atleta
3455	Maurizio
3320	Luigi

la relazione viene “spezzata” in due relazioni per eliminare l'anomalia

Figura 2.11: Esempio di una tabella non in 2NF e la sua normalizzazione a 2NF tramite scomposizione in due relazioni.

Terza Forma Normale (3NF)

Una relazione si dice in **Terza Forma Normale (3NF)** quando è in 2NF e tutti gli attributi non-chiave dipendono dalla chiave primaria soltanto, ossia non esistono attributi che dipendono da altri attributi non-chiave (non esistono dipendenze transitive). **Esempio di Violazione e Normalizzazione a 3NF (Esempio 1: Computer)**: Consideriamo una tabella "Computer" dove attributi come "Velocità" dipendono da "Codice Processore", che a sua volta dipende da "Codice". Questa è una dipendenza transitiva. La soluzione è scomporre la relazione originale in relazioni separate per eliminare le dipendenze transitive. **Esempio 2: Impiegato/Dipartimento**: In una relazione "Impiegato" con attributi (CodImpiegato, Nome, Reparto, CapoReparto), se "CodImpiegato" è la chiave primaria e "CapoReparto" dipende da "Reparto" (che è un attributo non-chiave), si ha una dipendenza transitiva. La soluzione è quella di scomporre la relazione "Impiegato" in due relazioni: una relazione Impiegati(CodImpiegato, Nome, Reparto) e una relazione Dipartimenti(Reparto, CapoReparto), eliminando così la dipendenza transitiva.

Computer

Codice	Descrizione	CodiceMemoria	Dimensione	CodiceProcessore	Velocità
1	Desktop base	20	512MB	P1	2GHz
2	Desktop medio	21	1GB	P2	3GHz
7	Desktop fascia alta	22	2GB	P3	4GHz
8	Portable wireless medio	25	1GB	P2	3GHz

- Codice → Descrizione;
- CodiceMemoria → Dimensione;
- CodiceProcessore → Velocità.

Figura 2.12: Esempio di una tabella "Computer" non in 3NF a causa di dipendenze transitive.

Computer

Codice	Descrizione	CodiceMemoria	CodiceProcessore
1	Desktop base	20	P1
2	Desktop medio	21	P2
7	Desktop fascia alta	22	P3
8	Portable wireless medio	25	P2

Processori

CodiceProcessore	Velocità
P1	2GHz
P2	3GHz
P1	2GHz
P3	1GHz

Memorie

CodiceMemoria	Dimensione
20	512MB
21	1GB
22	2GB
25	1GB

Figura 2.13: Normalizzazione della tabella "Computer" a 3NF, scomponendo le relazioni per eliminare le dipendenze transitive.

CodImpiegato	Nome	Reparto	CapoReparto
1	Verdi	Vendite	Rossi
2	Bianchi	Vendite	Rossi

Figura 2.14: Esempio di una tabella "Impiegato" non in 3NF per dipendenza transitiva.

Oltre la Terza Forma Normale: BCNF, 4NF, 5NF

Esistono forme normali ancora più elevate come la **Forma Normale di Boyce e Codd** (BCNF), la **Quarta Forma Normale** (4NF) e la **Quinta Forma Normale** (5NF). Queste

affrontano casi più specifici e complessi di dipendenze (es. dipendenze multi-valore). Tuttavia, in molti casi pratici, la normalizzazione fino alla 3NF è sufficiente per garantire un database efficiente e affidabile.

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()’).

Esempi di Operatori e Funzioni SQL Comuni

Oltre agli elementi base delle query SELECT, SQL offre un’ampia gamma di operatori e funzioni per manipolare e filtrare i dati in modo più complesso.

- **Operatori Logici:**

- AND: Combina due condizioni, entrambe devono essere vere.
- OR: Combina due condizioni, almeno una deve essere vera.
- NOT: Nega una condizione.

```

1  SELECT FirstName, LastName
2  FROM Students
3  WHERE Age > 20 AND City = 'Bologna';

```

Listing 2.1: Esempio Operatori Logici

- **Operatori di Confronto:**

- =: Uguale a.
- <> o !=: Diverso da.
- <, >, <=, >=: Minore, maggiore, minore o uguale, maggiore o uguale.

- BETWEEN min AND max: Valore compreso in un intervallo (inclusi gli estremi).
- LIKE pattern: Ricerca stringhe che corrispondono a un pattern (es. LIKE 'A%').
- IN (value1, value2, ...): Valore presente in una lista di valori.
- IS NULL / IS NOT NULL: Verifica se un valore è NULL.

```

1  SELECT ProductName, Price
2  FROM Products
3  WHERE Price BETWEEN 10.00 AND 50.00
4      AND ProductName LIKE 'Book%';
5
6  SELECT OrderID
7  FROM Orders
8  WHERE DeliveryDate IS NULL;

```

Listing 2.2: Esempio Operatori di Confronto

- **Funzioni Stringa:**

- CONCAT(s1, s2, ...): Concatena stringhe.
- SUBSTRING(string, start, length): Estrae una sottostringa.
- LENGTH(string): Restituisce la lunghezza di una stringa.
- UPPER(string) / LOWER(string): Converte in maiuscolo/minuscolo.

```

1  SELECT CONCAT(FirstName, ' ', LastName) AS FullName
2  FROM Users
3  WHERE LENGTH(FirstName) > 5;
4
5  SELECT UPPER(CategoryName)
6  FROM Categories;

```

Listing 2.3: Esempio Funzioni Stringa

- **Funzioni Numeriche:**

- ROUND(number, decimal_places): Arrotonda un numero.
- ABS(number): Valore assoluto.

```

1  SELECT ROUND(UnitPrice * Quantity, 2) AS RoundedTotal
2  FROM OrderDetails;
3
4  SELECT ABS(Balance)
5  FROM BankAccounts;

```

Listing 2.4: Esempio Funzioni Numeriche

- **Funzioni Data/Ora:**

- NOW(): Data e ora correnti.

- CURDATE(): Data corrente.
- DATE_ADD(date, INTERVAL value unit): Aggiunge un intervallo a una data.

```
1 SELECT EventName, EventDate
2 FROM Events
3 WHERE EventDate > CURDATE();
4
5 SELECT DATE_ADD(EventDate, INTERVAL 7 DAY) AS ExpectedEventDate
6 FROM Events;
```

Listing 2.5: Esempio Funzioni Data/Ora

Figura 2.15: Esempio di una query SQL che utilizza operatori e funzioni avanzate per filtrare e aggregare i dati.

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, optionalmente, 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/decrittografia 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 Riassembaggio**: 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).

3.5 Subnetting (Suddivisione di Sottoreti)

Il **subnetting** è il processo di divisione di una singola rete IP (Internet Protocol) più grande in sottoreti (subnets) più piccole e gestibili. Questa pratica è fondamentale per migliorare l'efficienza della rete, la sicurezza e la gestione degli indirizzi IP. Ogni sottorete è una rete IP indipendente e logicamente separata.

3.5.1 Obiettivi del Subnetting

- **Efficienza nell'uso degli indirizzi IP:** Invece di allocare un'intera classe di indirizzi a una singola organizzazione, il subnetting permette di suddividere lo spazio IP, utilizzando gli indirizzi in modo più parsimonioso.
- **Riduzione del traffico di rete:** Il traffico di broadcast è confinato alla propria sottorete, riducendo il carico sui router e migliorando le prestazioni complessive della rete.
- **Miglioramento della Sicurezza:** È più facile implementare politiche di sicurezza e isolare segmenti della rete.
- **Facilitazione della Gestione della Rete:** Le reti più piccole sono più facili da gestire, diagnosticare e risolvere i problemi.

3.5.2 Indirizzi IP e Maschere di Sottorete

Un indirizzo IP (IPv4) è un numero a 32 bit, diviso in quattro ottetti (separati da punti). È composto da due parti:

- **Indirizzo di Rete (Network Address):** Identifica la rete o sottorete a cui appartiene un dispositivo. Tutti i dispositivi sulla stessa sottorete hanno lo stesso indirizzo di rete.
- **Indirizzo Host (Host Address):** Identifica un dispositivo specifico all'interno di quella rete o sottorete.

La **maschera di sottorete (subnet mask)** è un numero a 32 bit che aiuta a distinguere la parte di rete dalla parte host di un indirizzo IP. È composta da una serie di ‘1’ (bit di rete) seguiti da una serie di ‘0’ (bit di host).

Notazione CIDR (Classless Inter-Domain Routing)

La notazione **CIDR** è un modo conciso per rappresentare indirizzi IP e subnet mask. Consiste nell’indirizzo IP seguito da uno slash (‘/’) e un numero intero che indica il numero di bit che compongono l’indirizzo di rete (la lunghezza della maschera di sottorete).

- **Esempio:** ‘192.168.1.0/24’ indica che i primi 24 bit sono dedicati alla rete, e i rimanenti 8 bit agli host.
- Il numero dopo lo slash è la **lunghezza del prefisso di rete**.

3.5.3 Calcolo del Subnetting (Esempio Pratico)

Consideriamo una rete di Classe C (es. ‘192.168.1.0/24’) e vogliamo suddividerla per ospitare diverse sottoreti per dipartimenti diversi, ognuno con un numero specifico di host.

Scenario: Una rete ‘192.168.1.0/24’ deve essere divisa in 4 sottoreti per ospitare almeno 50 host per sottorete.

Passo 1: Determinare il numero di bit necessari per le sottoreti. Abbiamo bisogno di 4 sottoreti. Il numero di bit (n) necessari per creare S sottoreti è dato da $2^n \geq S$. $2^n \geq 4 \Rightarrow n = 2$ bit.

Passo 2: Determinare il numero di bit per gli host. La maschera originale è /24, quindi abbiamo 8 bit dedicati agli host ($32 - 24 = 8$). Usando 2 bit per le sottoreti, rimangono $8 - 2 = 6$ bit per gli host in ogni sottorete. Il numero massimo di host per sottorete sarà $2^H - 2$, dove H è il numero di bit per gli host. $2^6 - 2 = 64 - 2 = 62$ host. Questo soddisfa il requisito di almeno 50 host per sottorete.

Passo 3: Calcolare la nuova maschera di sottorete. La nuova maschera avrà i bit di rete originali (24) più i bit presi in prestito per le sottoreti (2). Nuova lunghezza prefisso = $24 + 2 = 26$. La nuova maschera di sottorete sarà ‘/26’. In forma decimale: ‘255.255.255.192’ (perché i 2 bit accesi nell’ultimo ottetto sono $128 + 64 = 192$).

Passo 4: Elencare le sottoreti valide. Le sottoreti iniziano a intervalli del "blocco size" determinato dal numero di bit presi in prestito. La dimensione del blocco per l’ultimo ottetto è $256 - 192 = 64$.

- **Sottorete 1:**

- Indirizzo di Rete: ‘192.168.1.0/26’
- Primo Indirizzo Host Valido: ‘192.168.1.1’
- Ultimo Indirizzo Host Valido: ‘192.168.1.62’
- Indirizzo di Broadcast: ‘192.168.1.63’

- **Sottorete 2:**

- Indirizzo di Rete: ‘192.168.1.64/26’
- Primo Indirizzo Host Valido: ‘192.168.1.65’
- Ultimo Indirizzo Host Valido: ‘192.168.1.126’
- Indirizzo di Broadcast: ‘192.168.1.127’

- **Sottorete 3:**

- Indirizzo di Rete: ‘192.168.1.128/26’
- Primo Indirizzo Host Valido: ‘192.168.1.129’
- Ultimo Indirizzo Host Valido: ‘192.168.1.190’
- Indirizzo di Broadcast: ‘192.168.1.191’

- **Sottorete 4:**

- Indirizzo di Rete: ‘192.168.1.192/26’
- Primo Indirizzo Host Valido: ‘192.168.1.193’
- Ultimo Indirizzo Host Valido: ‘192.168.1.254’
- Indirizzo di Broadcast: ‘192.168.1.255’

Questo esempio mostra come una singola rete può essere efficientemente suddivisa per soddisfare esigenze specifiche di un’organizzazione.

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.

```

1 FUNCTION ModificaValore(numero: Integer):
2     numero = numero + 10
3     PRINT "Dentro la funzione: ", numero
4 END FUNCTION
5
6 DECLARE myNumber: Integer = 5
7 CALL ModificaValore(myNumber)
8 PRINT "Dopo la funzione: ", myNumber
9 // Output atteso: "Dopo la funzione: 5" (myNumber non e' cambiato)

```

Listing 4.1: Esempio di Passaggio per Valore

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.

```

1 FUNCTION ModificaArray(arrayRef: Array of Integer):
2     arrayRef[0] = arrayRef[0] + 10
3     PRINT "Dentro la funzione: ", arrayRef[0]
4 END FUNCTION
5
6 DECLARE myArray: Array of Integer = [1, 2, 3]
7 CALL ModificaArray(myArray)
8 PRINT "Dopo la funzione: ", myArray[0]
9 // Output atteso: "Dopo la funzione: 11" (myArray[0] e' cambiato)

```

Listing 4.2: Esempio di Passaggio per Riferimento

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 4.3: 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 Esegibile.
- **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 Precoce:** 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(n²) - 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 criptazione."
 - * **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 architettonale 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 (Unified Modeling Language)

Il **Unified Modeling Language (UML)** è un linguaggio di modellazione standardizzato, ampiamente utilizzato nell'ingegneria del software per specificare, visualizzare, costruire e documentare gli artefatti di un sistema software. Offre una notazione grafica per rappresentare vari aspetti di un sistema, dalla sua struttura statica al suo comportamento dinamico.

Diagrammi dei Casi d'Uso (Use Case Diagrams)

I **Diagrammi dei Casi d'Uso** sono diagrammi comportamentali UML utilizzati nella fase di analisi dei requisiti. Descrivono le interazioni tra gli utenti (attori) e il sistema, rappresentando le diverse funzionalità che il sistema offre dal punto di vista dell'utente. Si concentrano sul "cosa" il sistema fa per i suoi attori, piuttosto che sul "come" lo fa.

- **Scopo:** Identificare e documentare i requisiti funzionali del sistema. Forniscono una visione ad alto livello delle funzionalità, facilitano la comunicazione tra stakeholder e sviluppatori.

- **Componenti Principali:**

- **Attore (Actor):** Un ruolo esterno (persona, altro sistema, dispositivo hardware) che interagisce con il sistema. Rappresentato da un omino stilizzato.
- **Caso d'Uso (Use Case):** Una funzionalità specifica del sistema, un servizio che il sistema fornisce all'attore. Rappresentato da 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 ('«include»'):** Indica che un caso d'uso include la funzionalità di un altro caso d'uso. La freccia è tratteggiata, va dal caso d'uso includente a quello incluso. Usato per riutilizzare comportamenti comuni.

- * **Extend** ('«extend»'): Indica che un caso d'uso estende il comportamento di un altro caso d'uso in circostanze specifiche. La freccia è tratteggiata, va dal caso d'uso estendente a quello esteso. Usato per funzionalità opzionali o eccezionali.
- * **Generalizzazione**: Una relazione di ereditarietà tra attori o casi d'uso (freccia con punta triangolare non piena).

Esempio: Relazioni tra Casi d'Uso La comprensione delle relazioni ‘Include’, ‘Extend’ e ‘Generalizzazione’ è cruciale per modellare accuratamente il comportamento del sistema e i requisiti opzionali o riutilizzabili.

Relazione fra casi d'uso

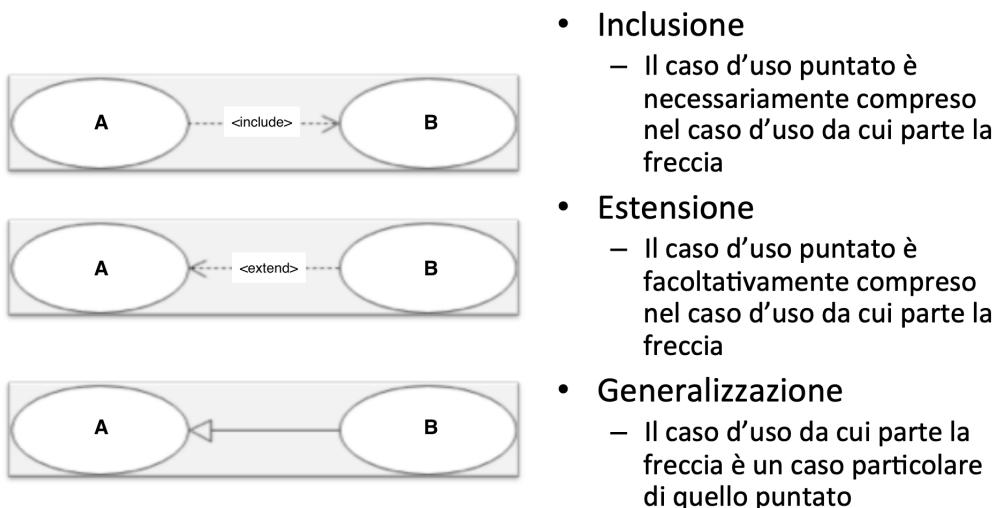


Figura 5.1: Rappresentazione grafica delle relazioni comuni tra Casi d'Uso in UML (Inclusione, Estensione e Generalizzazione).

Esempio Completo: Sistema di Ristorazione Un esempio pratico aiuta a consolidare la comprensione di come i casi d'uso e le loro relazioni si applichino in un contesto reale. Il diagramma seguente mostra le funzionalità di un sistema di gestione per un ristorante, illustrando le interazioni tra clienti, personale e il sistema stesso.

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 (dati), i loro metodi (operazioni) e le relazioni tra le classi. Sono fondamentali per la progettazione orientata agli oggetti e per modellare il design logico del database.

- **Scopo:** Modellare il design logico del sistema, la struttura del codice, le relazioni tra le classi e le dipendenze. Utile per visualizzare l'architettura del software a livello di classi.
- **Componenti Principali:**

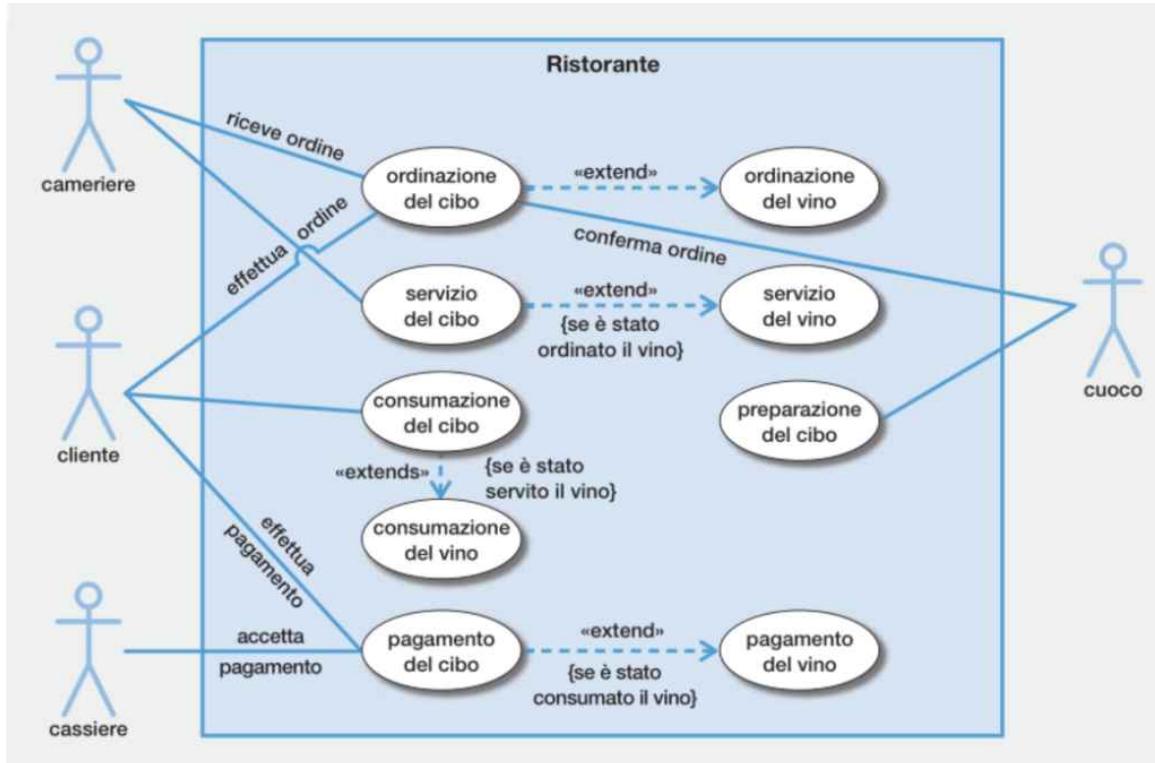


Figura 5.2: Esempio completo di Diagramma dei Casi d'Uso per un sistema di Ristorazione, che illustra le interazioni tra attori (Cameriere, Cliente, Cuoco, Cassiere) e le funzionalità del sistema.

- **Classe:** Rappresentata da un rettangolo diviso in tre sezioni: nome della classe, attributi (con visibilità e tipo), e metodi (con visibilità, parametri e tipo di ritorno).
- **Visibilità (Visibility):** Indica l'accessibilità degli attributi e metodi, rappresentata da simboli specifici (+, -, \#, \textasciitilde).
- **Relazioni:** Specificano le associazioni tra le classi.

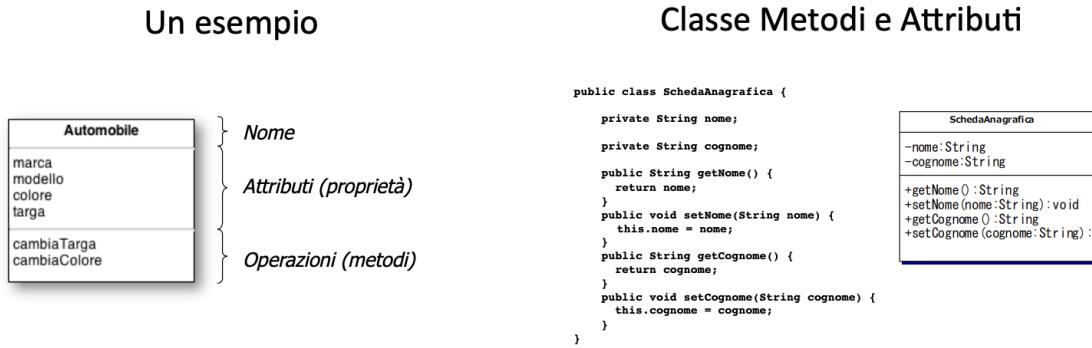
Struttura Base e Attributi Un diagramma delle classi inizia con la rappresentazione della singola classe, evidenziando il suo nome, i suoi attributi (le proprietà) e le sue operazioni (i metodi).

L'uso di modificatori di visibilità definisce l'accessibilità di questi attributi e metodi, come mostrato nell'esempio dei Clienti e Fornitori.

Relazioni di Generalizzazione (Ereditarietà) La generalizzazione indica che una classe (sottoclasse) eredita proprietà e comportamenti da un'altra classe (superclasse). Questo promuove il riutilizzo del codice. In contesti come Java, dove l'ereditarietà multipla non è ammessa (a differenza di C++), le interfacce vengono utilizzate per ovviare a questa limitazione, permettendo a una classe di implementare più interfacce.

Relazioni Strutturali: Aggregazione e Composizione Aggregazione e composizione sono forme specifiche di associazione che esprimono relazioni "parte di" tra classi, distinguendosi per la dipendenza esistenziale della parte rispetto al tutto.

- **Aggregazione:** Rappresenta una relazione uno a molti in cui l'oggetto "parte" può esistere indipendentemente dall'oggetto "tutto" (es. un libro può esistere senza una



(a) Struttura di una Classe (Automobile)

(b) Classe con Metodi e Attributi (Scheda Anagrafica)

Figura 5.3: Esempi di rappresentazione base di una classe UML, con attributi e metodi, e come si relaziona al codice.

Modificatori



- **+Public:** Libero Accesso
- **#Protected:** Accessibile dalle Sottoclassi
- **-Private:** Accessibile solo all'interno della classe
- **Static:** Accessibili anche senza creare istanze

Figura 5.4: Esempio di Diagramma delle Classi che illustra l'uso dei modificatori di visibilità (public, private, protected) per attributi e metodi.

mensola specifica). Viene rappresentata con una freccia con la punta a diamante vuota all'estremità del "tutto".

- **Composizione:** Una forma più forte di aggregazione, che implica una esclusività. La "parte" non può esistere da sola senza il "tutto", e la distruzione del "tutto" comporta la distruzione delle "parti" (es. una pagina non può esistere senza il suo libro). Il diamante si disegna pieno.

Esempi Complessi di Diagrammi delle Classi Per illustrare l'applicazione di queste relazioni in contesti più ampi, consideriamo sistemi con diverse classi interconnesse, come un sistema e-commerce o un'applicazione di chat. Questi esempi mostrano come le classi interagiscono per formare un sistema coerente.

Ereditarietà

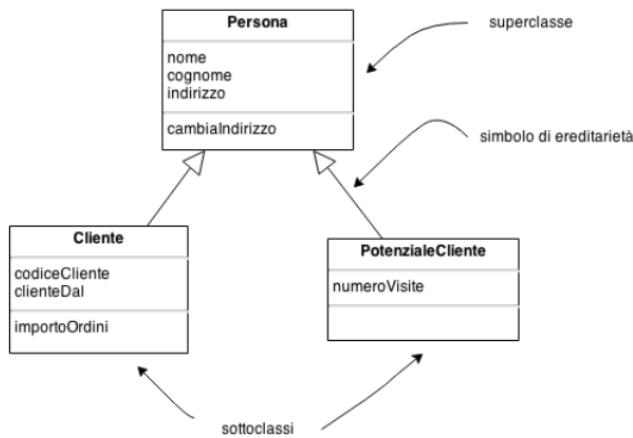


Figura 5.5: Esempio di Diagramma delle Classi che mostra la relazione di ereditarietà tra le classi Persona, Cliente e Potenziale Cliente.

Ereditarietà multipla

- ➊ In Java per esempio non è ammessa l'ereditarietà multipla (possibile in C++)
- ➋ Le interfacce permettono di ovviare a questo problema: una classe può ereditare da una sola classe ma implementare varie interfacce

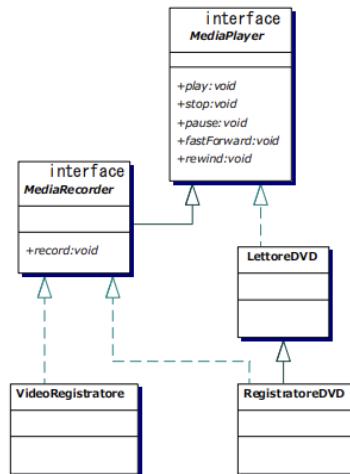
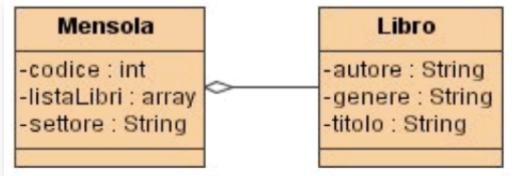


Figura 5.6: Esempio di Diagramma delle Classi che illustra come le interfacce consentono di simulare l'ereditarietà multipla in linguaggi che non la supportano nativamente.

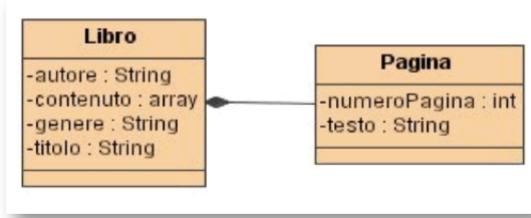
Diagrammi di Sequenza (Sequence Diagrams)

Un **Diagramma di Sequenza**, o in inglese *Sequence Diagram*, è un diagramma appartenente alla famiglia del linguaggio **UML (Unified Modeling Language)**, che viene utilizzato principalmente per rappresentare le interazioni tra gli oggetti, rispettando l'ordine sequenziale in cui avvengono questi scambi di messaggi. Sono utili non solo agli sviluppatori per modellare il comportamento di un'applicazione, ma anche ai dirigenti di un'azienda, in quanto possono riprodurre il comportamento dei vari elementi del sistema che costituiscono il business, mo-

Esempio di Aggregazione



Esempio di Composizione



(a) Esempio di Aggregazione (Mensola e Libro) (b) Esempio di Composizione (Libro e Pagina)

Figura 5.7: Confronto visivo tra Aggregazione e Composizione nei Diagrammi delle Classi UML.

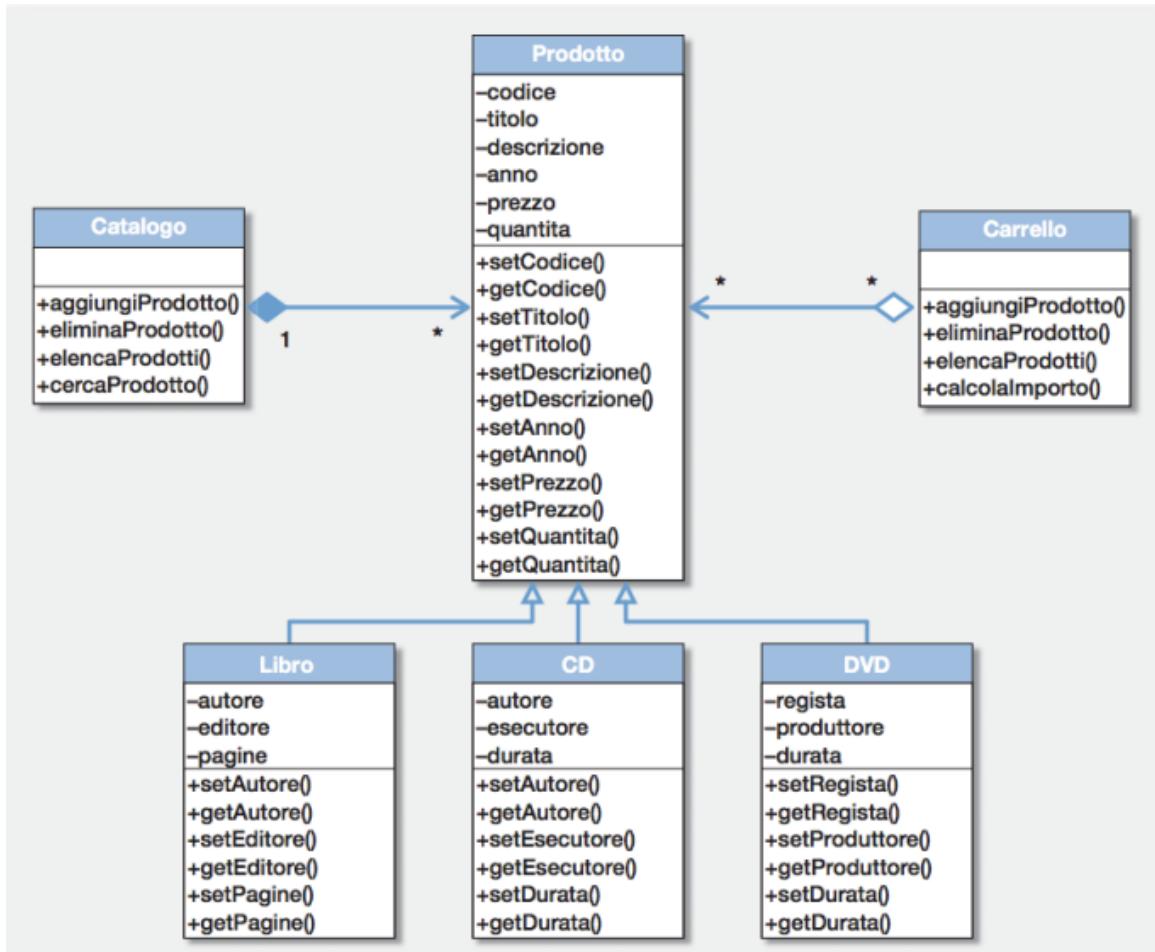


Figura 5.8: Diagramma delle Classi per un sistema di gestione prodotti/catalogo/carrello, che illustra varie classi e le loro relazioni (es. Prodotto, Carrello, DVD, Libro).

strand come interagiscono tra loro. Lo staff tecnico di un'organizzazione potrebbe avvalersi di questi diagrammi per documentare un comportamento. Ad esempio, durante la fase di progettazione, architetti e sviluppatori possono utilizzare questo schema per aggiungere o eliminare le interazioni tra le componenti del sistema.

È possibile utilizzare i diagrammi di sequenza a diversi livelli durante il processo di sviluppo:

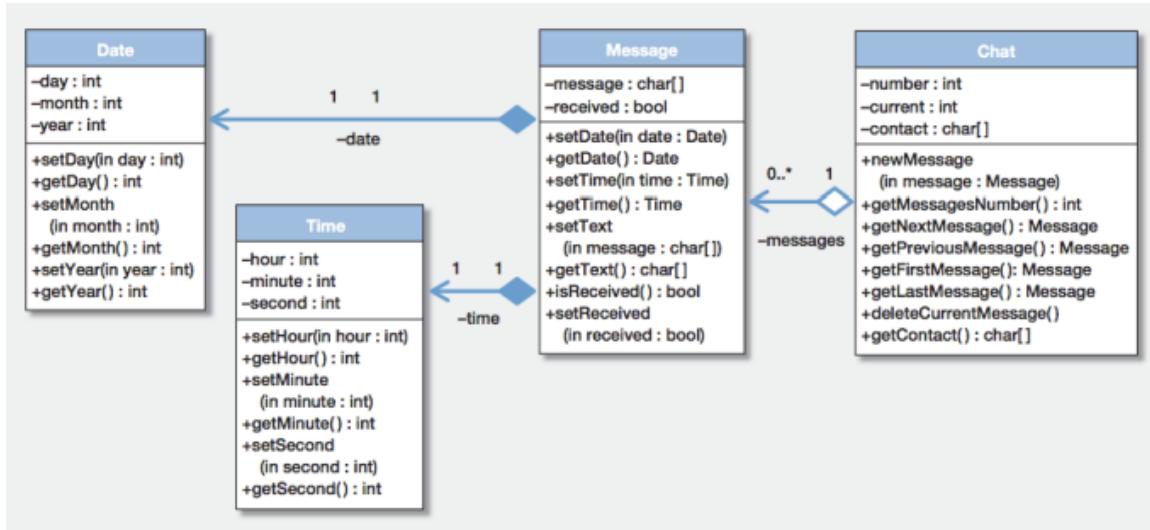


Figura 5.9: Diagramma delle Classi per un sistema di Chat, che mostra le interazioni tra classi come Utente, Chat e Messaggio, e le loro molteplicità.

- Nella fase di **analisi**, possono aiutare ad identificare le classi di cui un sistema ha bisogno e ciò che gli oggetti di classe fanno nelle interazioni.
- Nella fase di **progettazione**, spiegano come il sistema funziona per compiere le interazioni.
- Durante la costruzione di un'**architettura di un sistema**, è possibile utilizzarli per mostrare il funzionamento dei modelli di progettazione ed i meccanismi che il sistema utilizza.

Uno dei principali usi del diagramma è quello di raffinare, in uno o più diagrammi, i requisiti espressi nei diagrammi dei casi d'uso UML, che vengono usati per la descrizione delle funzioni o servizi offerti da un sistema, così come sono percepiti e utilizzati dagli attori che interagiscono col sistema stesso. Lo scopo principale è quello di definire sequenze di eventi che portano a qualche risultato desiderato. L'attenzione si concentra sull'ordine in cui si verificano i messaggi piuttosto che sul loro contenuto. Il diagramma trasmette queste informazioni lungo le dimensioni orizzontali e verticali:

- In **verticale**, dall'alto verso il basso, è evidenziata la sequenza temporale dei messaggi secondo l'ordine in cui si verificano.
- In **orizzontale**, da sinistra a destra, le istanze degli oggetti a cui sono inviati i messaggi.

Componenti Principali dei Diagrammi di Sequenza

- **Lifeline (Linea di Vita)**: Rappresenta la partecipazione di un oggetto o attore all'interazione, mostrata come una linea verticale tratteggiata. In cima alla lifeline c'è un rettangolo (per gli oggetti) o un omino (per gli attori).
- **Attore**: Utente o sistema esterno che avvia o partecipa all'interazione.
- **Messaggio**: Una comunicazione o chiamata di metodo tra lifeline, rappresentata da una freccia orizzontale. Possono essere:
 - **Sincroni**: Freccia piena, indica una chiamata bloccante in attesa di risposta.

- **Asincroni:** Freccia con punta aperta, indica una chiamata non bloccante.
- **Risposta:** Freccia tratteggiata, indica il ritorno di un valore o la fine di una chiamata sincrona.
- **Barra di Attivazione (Activation Bar/Execution Occurrence):** Un rettangolo sottile posizionato verticalmente sulla lifeline, che indica il periodo di tempo durante il quale un oggetto è attivo e sta eseguendo un'operazione o aspettando una risposta.
- **Frammenti Combinati (Combined Fragments):** Costrutti per mostrare strutture di controllo logico:
 - **alt (alternative):** Per mostrare blocchi if-else.
 - **opt (optional):** Per mostrare un blocco if.
 - **loop (loop):** Per mostrare iterazioni.
 - **par (parallel):** Per mostrare esecuzioni parallele.

Esempio 1: Flusso di Aggiornamento Dati in MVC Questo esempio illustra un tipico flusso di aggiornamento dei dati in un'architettura Model-View-Controller (MVC), mostrando le interazioni tra l'utente, la View, il Controller e il Model. È un esempio comune per descrivere come le modifiche da parte dell'utente si propagano attraverso i livelli di un'applicazione.

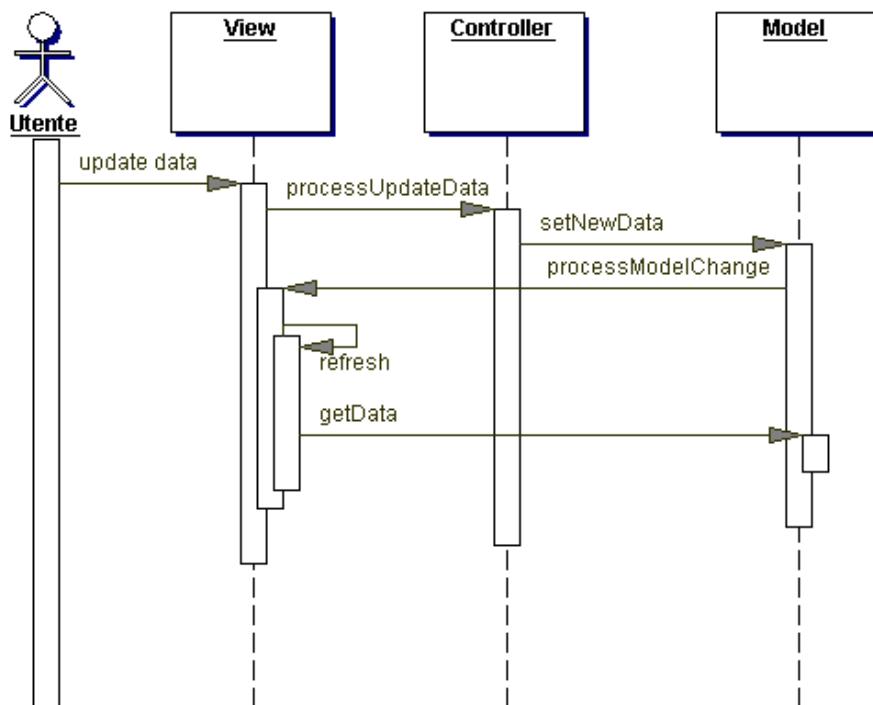


Figura 5.10: Diagramma di Sequenza UML che illustra il flusso di aggiornamento dei dati in un'architettura Model-View-Controller (MVC).

Esempio 2: Flusso di Prestito in un Sistema Biblioteca Questo diagramma mostra un processo più orientato al dominio di business, come la gestione del prestito di un libro in un sistema di biblioteca. Evidenzia l'interazione tra l'utente, un gestore dei prestiti, l'entità libro, e i registri dei prestiti totali, illustrando come i sistemi gestiscono gli stati e le verifiche per completare un'operazione.

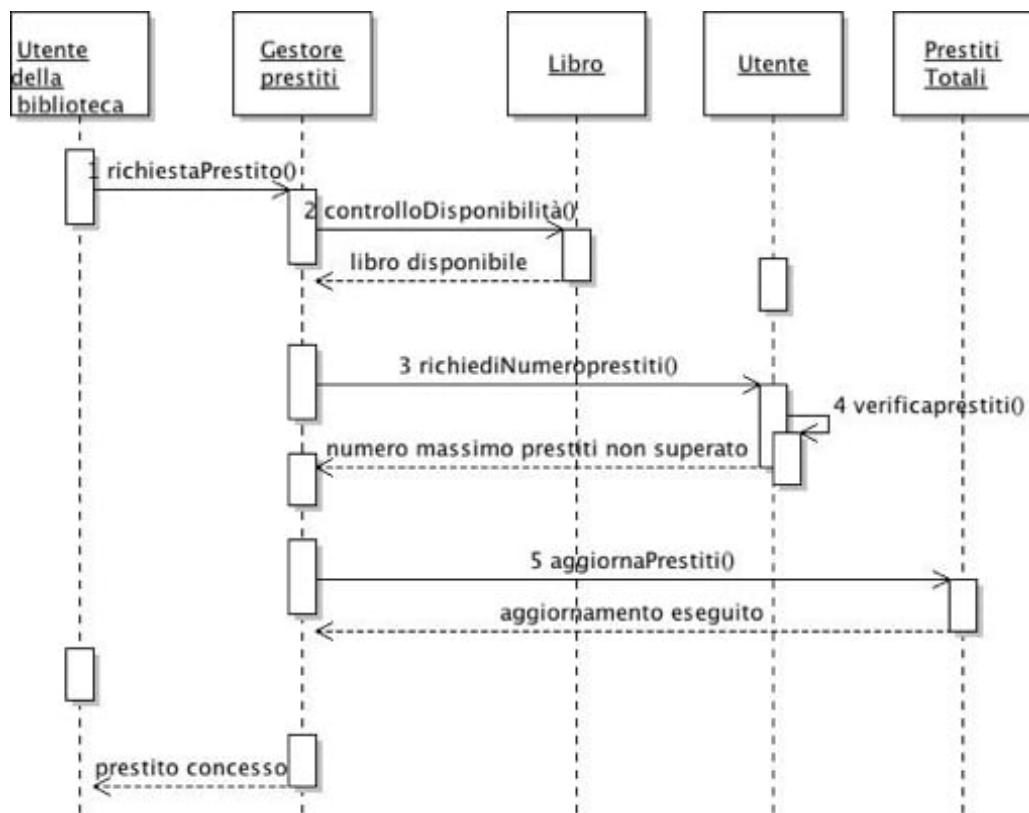


Figura 5.11: Diagramma di Sequenza UML che illustra il flusso di prestito di un libro in un sistema di gestione bibliotecaria.

Capitolo 6

Elettronica

L'**Elettronica** è la branca dell'ingegneria e della fisica che si occupa del controllo del flusso di elettroni, tipicamente attraverso dispositivi semiconduttori, per costruire circuiti e sistemi che elaborano informazioni o controllano energia. È alla base di tutti i dispositivi digitali e di molteplici sistemi analogici moderni.

6.1 Transistore MOS (Metal-Oxide-Semiconductor Field-Effect Transistor)

Il **transistore MOS** è il dispositivo fondamentale della microelettronica moderna e la spina dorsale di microprocessori, memorie e altri circuiti integrati. È un transistore a effetto di campo (FET), dove la corrente tra due terminali (Source e Drain) è controllata da un campo elettrico generato da una tensione applicata a un terzo terminale (Gate).

6.1.1 Struttura del Transistore MOS (nMOS)

Per comprendere il funzionamento, si consideri la realizzazione più comune: l'nMOS (n-channel Metal-Oxide-Semiconductor).

- **Substrato (Bulk/Body):** Generalmente di tipo P per un nMOS. È il materiale semiconduttore di base su cui viene costruito il dispositivo.
- **Source (S) e Drain (D):** Due regioni altamente drogate (N+ per nMOS) impiantate nel substrato P. Sono simmetriche e collegate ai terminali esterni. Il Source è tipicamente la sorgente di portatori di carica, il Drain è dove fluiscono.
- **Canale:** La regione del substrato (P) tra Source e Drain. È qui che si formerà il canale di conduzione per il flusso di corrente.
- **Ossido di Gate (SiO_2):** Uno strato isolante molto sottile (diossido di silicio) deposto sopra il canale. Agisce come dielettrico, isolando elettricamente il Gate dal canale.
- **Gate (G):** Uno strato conduttivo (metallo o polisilicio) deposto sopra l'ossido di gate. È il terminale di controllo attraverso il quale si applica la tensione per creare il campo elettrico.
- **Terminali:** Gate (G), Source (S), Drain (D), Bulk/Substrate (B). Spesso il Bulk è collegato al Source o a una tensione fissa (es. massa per nMOS, V_{DD} per pMOS).

6.1.2 Principio di Funzionamento (nMOS)

Il funzionamento del transistore MOS dipende dalla tensione applicata tra Gate e Source (V_{GS}) e tra Drain e Source (V_{DS}). Si basa sulla modulazione della conduttività del canale tramite un campo elettrico.

- **Interdizione (Cut-off Region):**

- **Condizione:** $V_{GS} < V_{TH}$ (Tensione di Soglia).
- **Descrizione:** Non c'è un campo elettrico sufficiente per attrarre elettroni al canale. Non si forma un canale di conduzione tra Source e Drain. La corrente I_{DS} è praticamente nulla (solo una piccola corrente di leakage). Il transistore agisce come un interruttore aperto.

- **Conduzione (quando $V_{GS} > V_{TH}$):**

- Quando una tensione positiva V_{GS} sufficiente (maggiore di V_{TH}) viene applicata al Gate, il campo elettrico attrae gli elettroni (portatori minoritari nel substrato P) verso la superficie del semiconduttore, sotto l'ossido.
- Si forma uno strato sottile di elettroni, creando un **canale di conduzione** a bassa resistenza tra Source e Drain.
- Se si applica una tensione $V_{DS} > 0$, una corrente I_{DS} fluirà attraverso questo canale.

6.1.3 Regimi di Funzionamento (per nMOS con $V_{GS} > V_{TH}$)

Una volta che il transistore è in conduzione, il suo comportamento (e la corrente I_{DS}) dipende dalla relazione tra V_{GS} e V_{DS} .

- **Regione di Triodo (o Lineare/Ohmica):**

- **Condizioni:** $V_{GS} > V_{TH}$ e $V_{DS} < (V_{GS} - V_{TH})$.
- **Descrizione:** Il canale è completamente formato e la sua profondità è relativamente uniforme. Il transistore si comporta come una resistenza controllata in tensione, e la corrente I_{DS} è approssimativamente lineare rispetto a V_{DS} .
- **Applicazione:** Usata come interruttore chiuso (ON) in circuiti digitali (con bassa resistenza) o come resistenza variabile in circuiti analogici.

- **Regione di Saturazione:**

- **Condizioni:** $V_{GS} > V_{TH}$ e $V_{DS} \geq (V_{GS} - V_{TH})$.
- **Descrizione:** Aumentando V_{DS} , la tensione canale-gate verso il lato del Drain si riduce. Quando V_{DS} raggiunge $V_{GS} - V_{TH}$, il canale si "pizzica" (pinch-off) vicino al Drain. Oltre questo punto, ulteriori aumenti di V_{DS} non aumentano significativamente la corrente I_{DS} , che diventa quasi costante.
- **Applicazione:** Questa è la regione di funzionamento preferita per gli amplificatori (circuiti analogici) in quanto si comporta come una sorgente di corrente controllata in tensione, e per gli stati ON (logico '1') in circuiti digitali per un'elevata impedenza di uscita.

6.1.4 Vantaggi del MOS rispetto al Transistore Bipolare (BJT)

Il transistore MOS ha soppiantato in gran parte il BJT nella microelettronica digitale e in molte applicazioni analogiche grazie a diversi vantaggi chiave:

- **Alta Impedenza di Ingresso:** Il gate del MOS è isolato dall'ossido, il che lo rende quasi idealmente un circuito aperto con una corrente di gate praticamente nulla ($I_G \approx 0$). Questo riduce il carico sui circuiti precedenti e semplifica il design degli stadi di ingresso. Il BJT, invece, è controllato in corrente (corrente di base) e presenta un'impedenza di ingresso inferiore.
- **Minore Dissipazione di Potenza Statica (in CMOS):** Nelle configurazioni Complementary MOS (CMOS), quando il circuito è in stato stabile (non commuta), uno dei transistori è sempre spento, eliminando un percorso di corrente diretto tra alimentazione e massa. Questo porta a una dissipazione di potenza statica estremamente bassa, fondamentale per dispositivi a batteria e circuiti ad alta integrazione. I BJT, anche quando "spenti", possono avere correnti di leakage maggiori e configurazioni logiche basate su BJT tendono a dissipare più potenza.
- **Scalabilità e Densità di Integrazione:** I MOS possono essere miniaturizzati molto più facilmente rispetto ai BJT, permettendo la realizzazione di circuiti integrati con miliardi di transistori su un singolo chip. La loro struttura planare si presta bene alla fabbricazione su larga scala.
- **Costo di Fabbricazione:** Il processo di fabbricazione MOS è generalmente più semplice e meno costoso rispetto a quello BJT, che richiede più passaggi di droggaggio e diffusione.
- **Immunità al Rumore (in CMOS):** I circuiti CMOS hanno buoni margini di rumore, rendendoli robusti contro le fluttuazioni di tensione indesiderate.

Nonostante ciò, i BJT mantengono vantaggi in alcune applicazioni specifiche (es. alta frequenza, alta potenza) grazie a una maggiore transconduttanza e velocità in particolari contesti.

6.2 Circuiti CMOS (Complementary MOS)

La tecnologia **CMOS** è il design più diffuso per la realizzazione di circuiti integrati digitali, caratterizzata dall'uso complementare di transistori nMOS e pMOS.

6.2.1 Invertitore CMOS

L'**invertitore CMOS** è la porta logica fondamentale in tecnologia CMOS, che realizza la funzione NOT (negazione logica). Produce un'uscita opposta all'ingresso.

- **Realizzazione:** È composto da due transistori MOS collegati in serie tra V_{DD} e GND, con i gate collegati insieme a formare l'ingresso (IN) e i drain collegati a formare l'uscita (OUT).
 - Un **pMOS** (pull-up network) è collegato tra V_{DD} e OUT. Il pMOS conduce quando il suo gate è basso.
 - Un **nMOS** (pull-down network) è collegato tra OUT e GND. L'nMOS conduce quando il suo gate è alto.
- **Funzionamento:**

- **Ingresso Alto (Logico '1', $V_{IN} = V_{DD}$):** L'nMOS è ON (conduce), creando un percorso a bassa resistenza tra OUT e GND. Il pMOS è OFF (interdetto). L'uscita V_{OUT} è tirata a GND (Logico '0').
- **Ingresso Basso (Logico '0', $V_{IN} = GND$):** Il pMOS è ON (conduce), creando un percorso a bassa resistenza tra OUT e V_{DD} . L'nMOS è OFF (interdetto). L'uscita V_{OUT} è tirata a V_{DD} (Logico '1').

6.2.2 Vantaggi dell'Invertitore CMOS rispetto agli Invertitori con Carico Resistivo

Gli invertitori CMOS offrono vantaggi significativi rispetto alle implementazioni più vecchie che usavano un transistore (BJT o MOS) con una resistenza di carico:

- **Minore Dissipazione di Potenza Statica (Vantaggio Primario):** Nello stato stabile (ingresso alto o basso), uno dei due transistori (nMOS o pMOS) è sempre spento. Non c'è un percorso di corrente diretto da V_{DD} a GND. La corrente statica è limitata a correnti di leakage quasi nulle, risultando in un consumo di potenza estremamente basso a riposo. Gli invertitori con carico resistivo, invece, dissipano continuamente potenza quando l'uscita è nello stato "basso", poiché la corrente fluisce attraverso il resistore di carico e il transistore acceso.
- **Migliori Margini di Rumore:** Le curve di trasferimento tensione-tensione degli invertitori CMOS sono quasi ideali, con una transizione molto ripida tra i due stati logici. Questo fornisce ampi margini di rumore, rendendo i circuiti più robusti alle fluttuazioni di tensione indesiderate.
- **Prestazioni Simmetriche (Potenziale):** Con un corretto dimensionamento dei transistori (rapporto W/L), i tempi di salita (charge time) e di discesa (discharge time) dell'uscita possono essere resi quasi simmetrici. Negli invertitori con resistore, il tempo di salita (carica del condensatore di carico attraverso il resistore) è tipicamente più lento del tempo di discesa.
- **Densità di Integrazione:** I transistori MOS occupano meno area rispetto ai resistori su chip, consentendo una maggiore densità di circuiti.
- **Ampio Range di Tensione Operativa:** I circuiti CMOS possono operare su un'ampia gamma di tensioni di alimentazione mantenendo buone prestazioni.

6.2.3 Applicazioni Notevoli dell'Invertitore CMOS

L'invertitore CMOS è un blocco costruttivo fondamentale per una vasta gamma di applicazioni:

- **Porta Logica Fondamentale:** È il componente di base da cui vengono costruite tutte le altre porte logiche digitali (NAND, NOR, XOR, latch, flip-flop) in tecnologia CMOS.
- **Buffer e Driver:** Utilizzando cascate di invertitori, si possono creare buffer (per aumentare la capacità di pilotaggio di un segnale) o driver per linee lunghe o carichi capacitivi elevati (es. clock network nei microprocessori).
- **Celle di Memoria:** Due invertitori CMOS collegati in back-to-back formano un latch, che è la cella di memoria fondamentale nelle SRAM (Static RAM), capace di memorizzare un bit.

- **Oscillatori ad Anello (Ring Oscillators):** Una catena di un numero dispari di invertitori, con l'uscita dell'ultimo collegata all'ingresso del primo, crea un oscillatore che produce un'onda quadra. Usati per testare le prestazioni di velocità e per generare segnali di clock.
- **Livelli di Tensione:** Possono essere usati per convertire livelli di tensione logici tra diversi standard o blocchi di circuiti.

Capitolo 7

Sistemi Numerici

I **sistemi numerici** sono metodi per rappresentare i numeri utilizzando simboli specifici e regole ben definite. In informatica, la rappresentazione binaria è fondamentale, ma è cruciale anche capire come i numeri negativi vengono gestiti, in particolare tramite il complemento a due.

7.1 Rappresentazione per Numeri Interi

I numeri interi possono essere rappresentati in diversi modi all'interno di un sistema digitale. Le rappresentazioni più comuni per i numeri con segno includono segno e modulo, complemento a uno, e complemento a due. Il complemento a due è la rappresentazione più utilizzata nei sistemi digitali moderni per la sua efficienza nelle operazioni aritmetiche.

7.1.1 Complemento a Due

La rappresentazione in **complemento a due** è il metodo più diffuso per rappresentare numeri interi con segno nei sistemi digitali. Offre il vantaggio di semplificare le operazioni di somma e sottrazione, poiché la sottrazione può essere implementata come una somma con il complemento a due del sottraendo, eliminando la necessità di circuiti dedicati alla sottrazione.

Principio di Funzionamento

- Un numero positivo in complemento a due è rappresentato esattamente come nella notazione binaria pura (senza segno), con il bit più significativo (MSB, most significant byte) pari a 0.
- Un numero negativo in complemento a due è ottenuto complementando (invertendo) tutti i bit del suo valore assoluto (passando da 0 a 1 e viceversa) e poi sommando 1 al risultato. Il bit più significativo (MSB) sarà sempre 1 per i numeri negativi.
- Il bit più significativo (MSB) indica il segno: 0 per i positivi, 1 per i negativi.
- Il range di valori rappresentabile con N bit in complemento a due va da -2^{N-1} a $2^{N-1} - 1$. Ad esempio, con 8 bit, si possono rappresentare numeri da -128 a 127.

Derivazione della Rappresentazione in Complemento a Due

Per convertire un numero decimale negativo in complemento a due (con N bit):

1. Prendi il valore assoluto del numero decimale (positivo).

2. Converti il valore assoluto in binario su N bit.
3. Inverti tutti i bit (complemento a uno, 0 diventa 1, 1 diventa 0).
4. Somma 1 al risultato binario.

Per convertire un numero binario in complemento a due a decimale:

- Se il MSB è 0: il numero è positivo. Convertilo come un normale binario senza segno.
- Se il MSB è 1: il numero è negativo.
 1. Inverti tutti i bit del numero binario.
 2. Somma 1 al risultato binario.
 3. Converti questo risultato binario in decimale e anteponi il segno meno.

Esempio: Rappresentazione in Binario a 16 bit del numero decimale -15

Per rappresentare il numero decimale -15 in complemento a due su 16 bit:

1. **Valore assoluto:** $|-15| = 15$.
2. **15 in binario su 16 bit:** 0000 0000 0000 1111₂.
3. **Inverti tutti i bit (complemento a uno):** 1111 1111 1111 0000₂.
4. **Somma 1:** 1111 1111 1111 0000₂ + 1₂ = 1111 1111 1111 0001₂.

Quindi, la rappresentazione in complemento a due di -15 su 16 bit è 1111 1111 1111 0001₂.

7.2 Operazioni Aritmetiche con Complemento a Due

Il vantaggio principale del complemento a due è che le operazioni di addizione e sottrazione possono essere eseguite utilizzando lo stesso circuito per l'addizione.

7.2.1 Addizione

L'addizione di due numeri (positivi o negativi) in complemento a due viene eseguita come una normale addizione binaria. Qualsiasi bit di riporto che esce dal bit più significativo (MSB) viene semplicemente ignorato.

- **Esempio:** $5 + (-2)$ su 4 bit ($N = 4$, range da -8 a 7)
 - $5_{10} = 0101_2$
 - -2_{10} :
 - * $2_{10} = 0010_2$
 - * Complemento a uno: 1101₂
 - * Somma 1: $1101_2 + 1_2 = 1110_2$ (che è -2_{10} in complemento a due)
 - **Somma:**

0101	(5)
+ 1110	(-2)

10011	

Il bit di riporto (il primo 1 a sinistra) viene ignorato. Il risultato è $0011_2 = 3_{10}$, che è corretto.

7.2.2 Sottrazione

La sottrazione $A - B$ è implementata come la somma $A + (-B)$. Per calcolare $-B$, si determina il complemento a due di B .

- **Esempio:** $5 - 2$ su 4 bit ($N = 4$, range da -8 a 7)

- $5_{10} = 0101_2$
- $-2_{10} = 1110_2$ (già calcolato sopra)
- **Sottrazione come somma:** $5 + (-2)$

$$\begin{array}{r} 0101 \quad (5) \\ + 1110 \quad (-2) \\ \hline 10011 \end{array}$$

Ignorando il riporto, il risultato è $0011_2 = 3_{10}$, che è corretto.

7.2.3 Overflow

L'**overflow** si verifica quando il risultato di un'operazione aritmetica supera il range di valori rappresentabili con il numero di bit a disposizione.

- **Rilevamento:** Si verifica un overflow se:

- Si sommano due numeri positivi e il risultato è negativo.
- Si sommano due numeri negativi e il risultato è positivo.
- Un modo più formale per rilevarlo è controllare se il riporto nel bit del segno (MSB) è diverso dal riporto fuori dal bit del segno.

