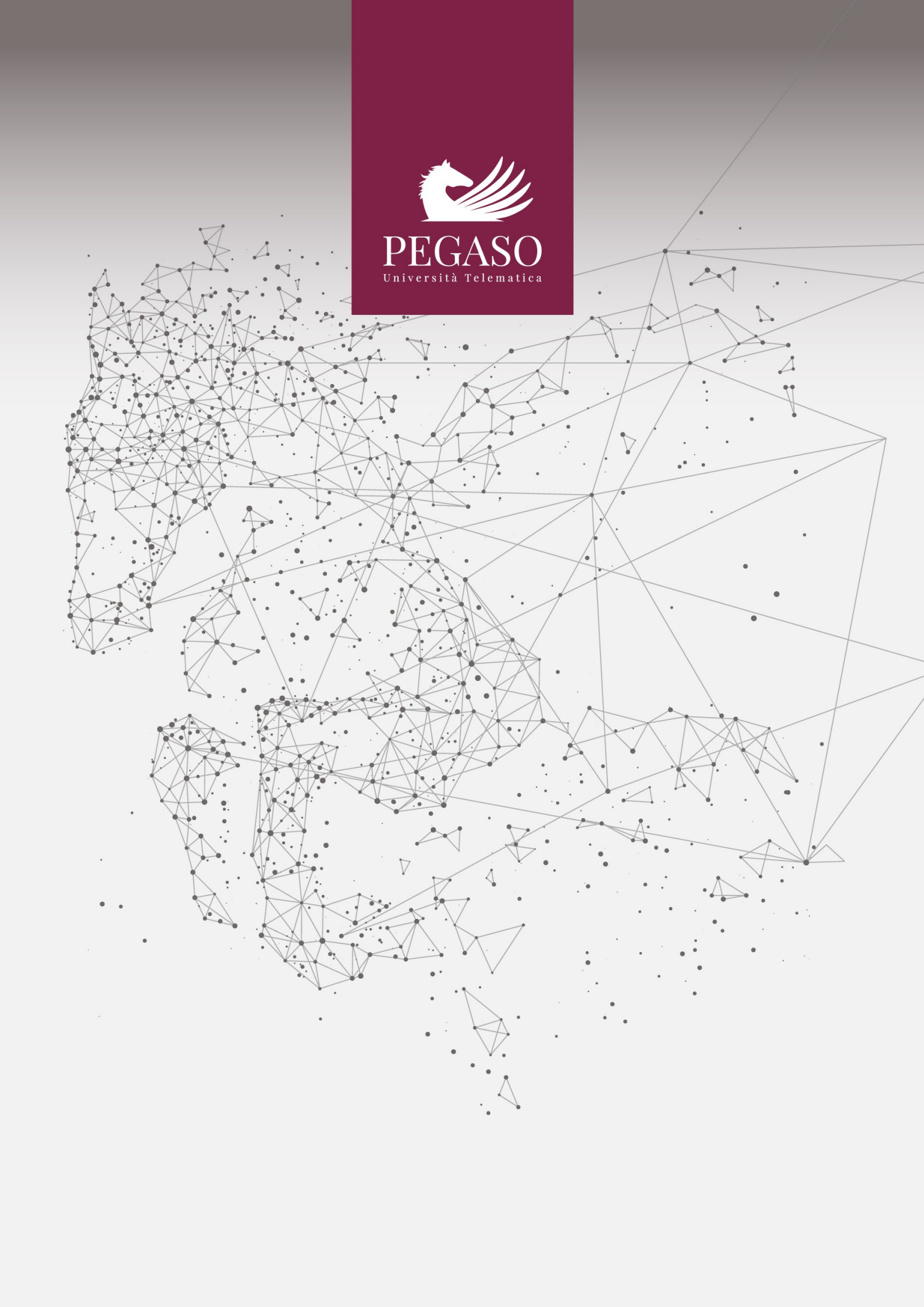
PEGASO
Università Telematica

Indice

1. Premessa

La progettazione dei sistemi software non si limita alla definizione delle funzionalità principali, ma implica anche l'individuazione e il soddisfacimento degli **obiettivi non funzionali**. Questi ultimi comprendono aspetti critici quali **prestazioni**, **affidabilità**, **sicurezza** e **manutenibilità**, che spesso determinano il successo o il fallimento di un sistema nel lungo periodo. Tali aspetti, spesso trascurati nella fase iniziale del progetto, acquisiscono un'importanza crescente man mano che il sistema evolve e viene impiegato in scenari reali, dove fattori come la robustezza alle anomalie o la capacità di adattarsi a modifiche future diventano determinanti.

La lezione qui presentata si propone di offrire una visione sistematica delle attività fondamentali del **system design**, focalizzandosi sulla **decomposizione in sottosistemi** e sull'analisi delle principali decisioni architetturali. L'obiettivo è mostrare come tali attività non siano isolate, ma interconnesse e in grado di influenzarsi reciprocamente: ad esempio, la scelta di un determinato modello di persistenza dei dati può condizionare la struttura del flusso di controllo o richiedere specifici meccanismi di accesso.

Attraverso un approccio concreto, esploreremo le attività che contribuiscono a definire la struttura del sistema in relazione agli obiettivi di progetto. Tra queste, la **mappatura hardware/software**, la **gestione dei dati persistenti**, il **controllo degli accessi**, la **progettazione del flusso di controllo globale** e l'analisi delle **condizioni limite**. Ogni decisione presa in queste fasi ha implicazioni dirette sulla struttura del sistema e sulle modalità con cui risponde ai requisiti di qualità. In tale contesto, ogni obiettivo non funzionale deve essere esplicitamente considerato e tracciato nel progetto, per evitare che scelte locali compromettano la coerenza dell'intero sistema.

La lezione ha l'obiettivo di fornire **strumenti concettuali e operativi** per affrontare con metodo la progettazione di sistemi software complessi. Particolare attenzione sarà dedicata alla classificazione delle scelte progettuali secondo le dimensioni di qualità del software, e all'adozione di pratiche consolidate nell'industria per la gestione della sicurezza, della persistenza dei dati e della configurazione dei sistemi distribuiti., e all'adozione di pratiche consolidate nell'industria per la gestione della sicurezza e della persistenza dei dati.

2. Mapping hardware/software e Flusso di controllo

La **mappatura hardware/software** rappresenta una delle attività cardine del system design, in quanto determina **su quali nodi fisici o ambienti virtuali** verranno eseguiti i vari sottosistemi software. Questa scelta ha impatti rilevanti su parametri come la **scalabilità**, la **latency**, la **tolleranza ai guasti** e la **complessità della comunicazione inter-sottosistemi**. Inoltre, incide profondamente sull'efficienza energetica, sulla gestibilità del sistema e sulla possibilità di effettuare aggiornamenti o manutenzioni senza interrompere il servizio.

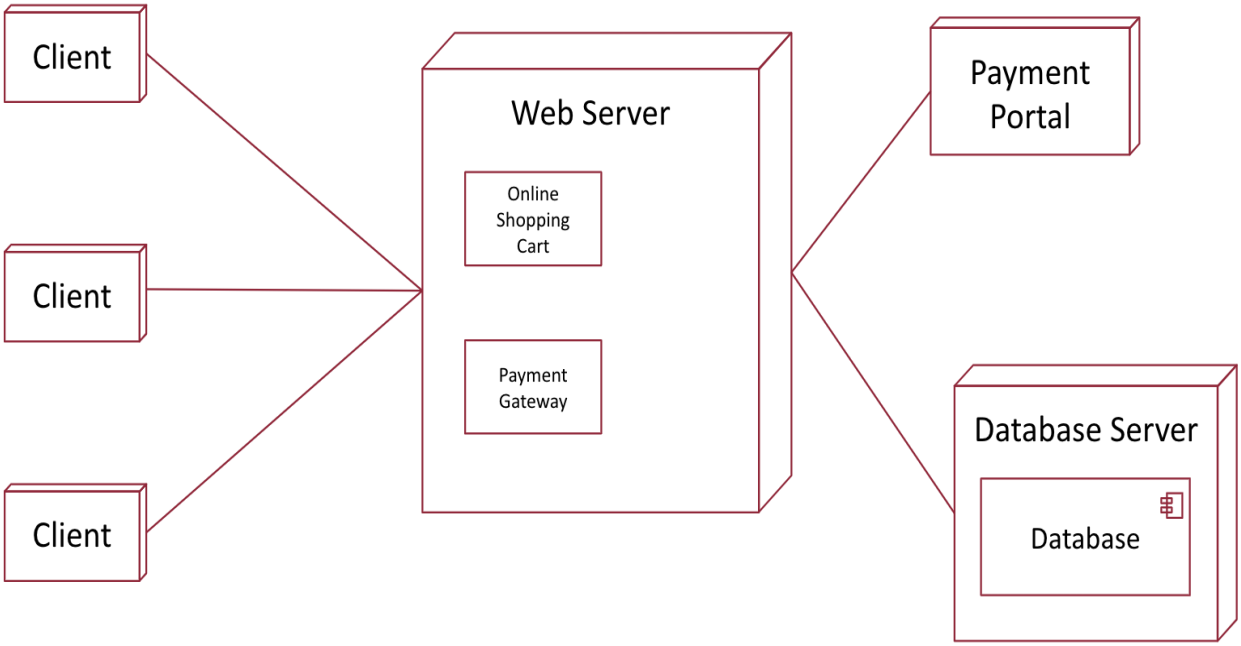

Figura 1. *Deployment Diagram*

Nel contesto moderno, l'adozione di **architetture distribuite** è spesso inevitabile, soprattutto in scenari in cui si desidera offrire funzionalità avanzate come il supporto a client remoti, la continuità operativa o l'integrazione con sistemi eterogenei. La distribuzione dei sottosistemi permette di **separare le responsabilità**, migliorare le prestazioni e aumentare l'affidabilità, ma introduce al contempo sfide legate alla **coerenza dei dati**, alla **gestione degli errori**, alla **latenza nella comunicazione** e alla **sincronizzazione degli stati**.

Per modellare tali scelte si utilizzano i **diagrammi UML di deployment (vedi Figura 1)**, nei quali i **nodi** rappresentano dispositivi fisici (es. server, dispositivi mobili) o ambienti di esecuzione (es. container, JVM), e le **connessioni** esplicitano i protocolli di comunicazione (HTTP, JDBC, TCP/IP). Questi diagrammi

offrono una rappresentazione ad alto livello della struttura distribuita del sistema, evidenziando **la distribuzione dei componenti**, le relazioni tra nodi, i **servizi offerti e richiesti**, e gli eventuali vincoli di affidabilità e sicurezza. L'uso appropriato dei deployment diagram consente di valutare anticipatamente **colli di bottiglia**, **punti singoli di fallimento** e possibili ottimizzazioni infrastrutturali.

Contemporaneamente alla mappatura, è necessario progettare il **flusso di controllo globale**, che definisce l'ordine e le condizioni con cui il sistema esegue le operazioni in risposta a **input esterni**, **eventi** o **scadenze temporali**. Questa progettazione è cruciale per garantire **reattività**, **coerenza**, e una **distribuzione efficiente del carico**. Sono possibili tre modelli principali:

1. **Controllo procedurale**, adatto a sistemi legacy, in cui ogni componente procede linearmente dopo aver ricevuto input. Questo modello, sebbene semplice da comprendere, risulta poco flessibile e difficile da manutenere in ambienti dinamici o distribuiti.
2. **Controllo basato su eventi**, in cui un ciclo centrale riceve eventi e li distribuisce agli handler appropriati. Questo approccio, molto diffuso in applicazioni web e interfacce utente, consente una maggiore modularità e reattività, ma può complicare la gestione delle sequenze logiche più articolate.
3. **Controllo basato su thread**, in cui ciascun evento genera un nuovo flusso di esecuzione concorrente. Questo modello consente un elevato grado di parallelismo e una gestione più naturale delle attività asincrone, ma introduce problemi legati alla **sincronizzazione**, **race conditions** e **debug**.

La scelta del modello più adatto dipende da requisiti come **responsività**, **concorrenza**, **robustezza** e **manutenibilità**, e influisce direttamente sull'organizzazione interna del sistema, sulla **scalabilità futura** e sulla definizione delle **interfacce tra sottosistemi**. Una progettazione consapevole e coerente del flusso di controllo permette di affrontare in modo efficace l'evoluzione del sistema e la gestione degli scenari eccezionali.

3. Gestione dei dati persistenti

La **gestione dei dati persistenti** è una componente essenziale nella progettazione di un sistema software, in quanto determina come e dove vengono salvate le informazioni che devono sopravvivere alla terminazione dell'esecuzione. I **dati persistenti** includono informazioni utente, configurazioni, stati intermedi e tutto ciò che deve essere recuperato anche dopo un riavvio o un crash del sistema. Una cattiva gestione della persistenza può comportare la **perdita di dati**, **incoerenze** o **degradazione delle prestazioni**, compromettendo l'intero sistema.

La prima attività in quest'ambito è l'**identificazione degli oggetti persistenti**, ovvero quegli elementi del dominio applicativo che devono essere conservati nel tempo. Questi possono essere entità centrali (es. ordini, utenti, prenotazioni) ma anche oggetti ausiliari, come configurazioni personalizzate, preferenze utente o informazioni di logging. Un criterio utile per l'identificazione è valutare quali dati devono essere ripristinabili dopo una disconnessione o un arresto anomalo. Inoltre, è fondamentale comprendere il ciclo di vita di ciascun oggetto: se un oggetto viene creato e utilizzato oltre una singola sessione del sistema, esso è un candidato per la persistenza.

Una volta identificati, è necessario stabilire **come memorizzare questi dati**. Le principali strategie comprendono:

- **File flat**: semplici da implementare, adatti a dati temporanei o non strutturati. Offrono prestazioni elevate, ma poca sicurezza e gestione della concorrenza. L'accesso diretto ai file può essere efficiente per piccoli dataset, ma risulta inadeguato per applicazioni multi-utente o per gestire grandi volumi di dati.
- **Database relazionali**: ideali per dati strutturati e accesso multi-utente. Offrono strumenti robusti per **transazioni**, **query complesse**, **backup** e **ripristino**. Sono particolarmente efficaci quando è richiesta **coerenza ACID** (Atomicità, Coerenza, Isolamento, Durabilità), e supportano una modellazione rigorosa delle relazioni tra entità.
- **Database orientati agli oggetti**: memorizzano direttamente oggetti e associazioni, riducendo la necessità di mapping. Adatti a sistemi con forte enfasi su relazioni complesse tra dati e dinamiche di navigazione nei grafi di oggetti. Facilitano la serializzazione e deserializzazione automatica, ma possono risultare più lenti in operazioni massive.

La scelta della tecnologia dipende da vari fattori: **complessità dei dati**, **frequenza di accesso**, **necessità di transazioni**, **scalabilità**, **budget** e **requisiti di prestazione**. Bisogna anche considerare la **compatibilità con tecnologie esistenti**, la **disponibilità di skill nel team**, e la **facilità di manutenzione a**

lungo termine. Spesso si adottano **soluzioni ibride**, dove ad esempio file vengono usati per il salvataggio rapido, mentre i database conservano i dati condivisi e permanenti. In alcuni casi, si può anche impiegare la **caching in memoria** per velocizzare l'accesso a dati frequentemente utilizzati, utilizzando strutture come memcached o Redis, che permettono di alleggerire il carico sui sistemi di storage persistente e migliorare la reattività delle applicazioni in tempo reale.

La progettazione della persistenza dovrebbe includere anche la **gestione della concorrenza** (lock, transazioni concorrenti), la **protezione contro la corruzione dei dati**, la **disponibilità dei backup** e la possibilità di **ripristino in caso di crash**. Un sistema ben progettato prevede inoltre **strumenti per la migrazione dei dati**, per supportare aggiornamenti o evoluzioni del modello persistente.

4. Controllo degli accessi

Il **controllo degli accessi** è fondamentale nei sistemi multi-utente, dove non tutti gli utenti devono avere gli stessi permessi su dati e funzionalità. Serve a definire **chi può fare cosa**, tutelando **sicurezza**, **riservatezza** e **integrità** del sistema. Senza un controllo efficace, il sistema diventa vulnerabile ad accessi non autorizzati e a potenziali violazioni dei dati. Inoltre, l'assenza di un modello ben definito per la gestione dei permessi può generare **ambiguità nei comportamenti del sistema**, creare **incoerenze nei dati** e rendere estremamente difficile la verifica della conformità alle normative vigenti, come il GDPR.

Esistono diverse tecniche per implementare il controllo degli accessi:

- **RBAC (Role-Based Access Control)**: i permessi sono assegnati a **ruoli**, e gli utenti ereditano i permessi dei ruoli cui appartengono. È un modello scalabile, adatto a organizzazioni strutturate dove esistono responsabilità chiaramente delineate. RBAC è particolarmente efficace nei contesti in cui i ruoli sono relativamente stabili e ben definiti, come in ambito aziendale o sanitario.
- **ABAC (Attribute-Based Access Control)**: le decisioni di accesso sono basate su **attributi** degli utenti, delle risorse, delle azioni e del contesto (es. orario, posizione geografica). Offre flessibilità e granularità, risultando adatto a sistemi **dinamici** e **complessi**, dove i ruoli non sono sempre sufficienti a descrivere le condizioni di accesso. Tuttavia, la gestione delle policy ABAC può essere più onerosa in termini di configurazione e verifica.
- **ACL (Access Control List)**: ogni oggetto ha una lista di utenti e permessi associati. È un modello semplice e diretto, ma difficile da scalare in sistemi complessi con numerosi utenti e risorse. È spesso utilizzato in combinazione con altri modelli per implementare controlli puntuali su oggetti specifici.

Risorsa	Cliente	Venditore	Admin
Ordine	Visualizza, Crea, Modifica	Visualizza, Modifica	Tutti
Prodotto	Visualizza	Visualizza, Crea, Modifica	Tutti
Recensione	Visualizza, Crea, Modifica	Visualizza	Visualizza, Elimina
Utente	Modifica profilo	-	Tutti
Statistiche	-	Visualizza base	Visualizza tutte

Tabella 1. Matrice degli accessi - Esempio

Un concetto chiave è la **matrice degli accessi (vedi Tabella 1)**, una rappresentazione tabellare delle autorizzazioni tra attori e risorse. Tale matrice può essere realizzata tramite liste di controllo, capability o regole dinamiche. Nei sistemi complessi, è utile adottare strumenti di **audit e logging centralizzati** per monitorare in tempo reale gli accessi e identificare tempestivamente comportamenti anomali o non conformi.

Il controllo può essere ulteriormente rafforzato attraverso **meccanismi di autenticazione**, che servono a verificare l'identità degli utenti prima di concedere l'accesso. Tra questi: l'uso di **password robuste**, **autenticazione a due fattori**, **smart card**, o **sistemi biometrici** come il riconoscimento facciale o delle impronte digitali.

Infine, è fondamentale considerare la **cifratura dei dati** e delle comunicazioni: anche qualora un accesso non autorizzato riuscisse ad aggirare i meccanismi di controllo, la protezione crittografica rappresenta un'ulteriore barriera alla violazione dei dati. L'adozione di protocolli sicuri (es. TLS/SSL), la cifratura end-to-end e la protezione delle chiavi crittografiche sono elementi imprescindibili per la sicurezza complessiva del sistema.

5. Condizioni limite

Le **condizioni limite** comprendono tutti quei comportamenti del sistema che si manifestano **prima, dopo o al di fuori del flusso operativo standard**. Questi includono l'**avvio del sistema**, lo **shutdown**, la **gestione delle eccezioni** e le **attività amministrative**. Spesso trascurati nei requisiti funzionali, questi aspetti sono invece fondamentali per garantire **robustezza**, **recupero da errori** e **manutenibilità**. La corretta gestione delle condizioni limite costituisce infatti una barriera protettiva che assicura la continuità operativa del sistema anche in presenza di eventi imprevisti o scenari degradati.

Durante l'avvio, il sistema deve caricare configurazioni, verificare la consistenza dei dati, ripristinare stati precedenti e avviare componenti software secondo una sequenza ben definita. In sistemi distribuiti, ciò implica anche la sincronizzazione tra nodi, l'inizializzazione di connessioni di rete sicure e l'eventuale validazione di licenze o certificati. Analogamente, durante lo shutdown il sistema deve salvare i dati in modo sicuro, chiudere connessioni attive, liberare risorse di sistema (file, memoria, thread) e notificare i sistemi remoti. Gli **errori di avvio o spegnimento** sono una causa frequente di malfunzionamenti in ambienti distribuiti, poiché possono interrompere flussi critici o lasciare il sistema in uno stato inconsistente.

La **gestione delle eccezioni** deve considerare errori hardware (es. guasti a dischi, memoria), errori di rete (es. timeout, disconnessioni) ed errori software (es. eccezioni non catturate, overflow, violazioni di accesso). Le strategie comuni includono **tolleranza** (es. ripetizione automatica dell'operazione, uso di dati temporanei), **recupero** (es. rollback transazionale, ripristino da backup), e **notifica all'utente o al log**, eventualmente con livelli di criticità differenziati (warning, error, fatal). L'adozione di un **sistema di gestione centralizzata degli errori** consente di monitorare e intervenire tempestivamente sulle anomalie più gravi, migliorando la resilienza complessiva.

Anche gli aggiornamenti e la manutenzione rientrano a pieno titolo tra le condizioni limite. Il sistema deve poter gestire con sicurezza operazioni come aggiornamenti del software, migrazioni di dati, modifica di configurazioni, aggiornamento di certificati. Questo richiede l'adozione di **procedure di hot-swap**, **modalità di manutenzione**, **test di rollback** e meccanismi di validazione post-deployment. Un sistema robusto consente anche aggiornamenti in background senza interruzioni del servizio.

Un buon system design prevede **use case amministrativi** specifici (es. gestione utenti, aggiornamento mappe, configurazione server) e meccanismi automatici per la rilevazione e la gestione di anomalie. Inoltre, l'introduzione di **sistemi di logging strutturato**, **audit trail** per la tracciabilità delle azioni critiche, e **interfacce di amministrazione sicure** consente una gestione efficace degli scenari limite. Tali

strumenti, se progettati correttamente, supportano anche la conformità a normative sulla sicurezza e sulla continuità operativa (es. ISO/IEC 27001, GDPR, ITIL).

6. Conclusioni e sintesi

La progettazione di un sistema software orientata agli obiettivi non funzionali richiede una visione d'insieme e decisioni architetturali consapevoli. Dalla mappatura distribuita dei sottosistemi, alla gestione del flusso di controllo, dalla scelta della persistenza dei dati al controllo degli accessi, fino alla gestione delle condizioni limite: ogni elemento contribuisce a rendere il sistema **scalabile**, **affidabile**, **sicuro** e **manutenibile**. La progettazione architetturale, pertanto, non può essere considerata come un esercizio puramente tecnico, ma come un'attività strategica che impatta direttamente sull'efficacia operativa e sulla capacità evolutiva dell'intero ecosistema software.

Un buon system design non si limita a far funzionare il sistema, ma lo prepara a **resistere agli imprevisti**, **evolvere nel tempo**, e **rispondere con efficacia alle esigenze degli utenti e dei gestori**. In altre parole, un'architettura ben congegnata è quella che rende semplice il difficile, nasconde la complessità senza sacrificarne la gestione, e permette al sistema di adattarsi senza comprometterne l'integrità. Inoltre, consente di ridurre i costi di manutenzione, favorire il riuso dei componenti e semplificare l'integrazione con altre piattaforme o tecnologie emergenti. Un sistema progettato con lungimiranza non solo è più semplice da evolvere, ma si rivela anche più robusto di fronte a cambiamenti nei requisiti, nella tecnologia o nel contesto d'uso.

La padronanza delle tecniche e degli strumenti illustrati consente al progettista di compiere scelte informate e robuste, investendo nella **qualità complessiva del software**, nella **riduzione del debito tecnico** e nella **sua sostenibilità futura** in termini di aggiornabilità, monitoraggio, scalabilità e sicurezza. In sintesi, il system design rappresenta il punto di sintesi tra le ambizioni progettuali e la concretezza dell'implementazione, e incarna la visione che guida ogni altra scelta lungo il ciclo di vita del software.

Bibliografia

- Bruegge, B., & Dutoit, A. H. (2010). Object-oriented software engineering. Using UML.