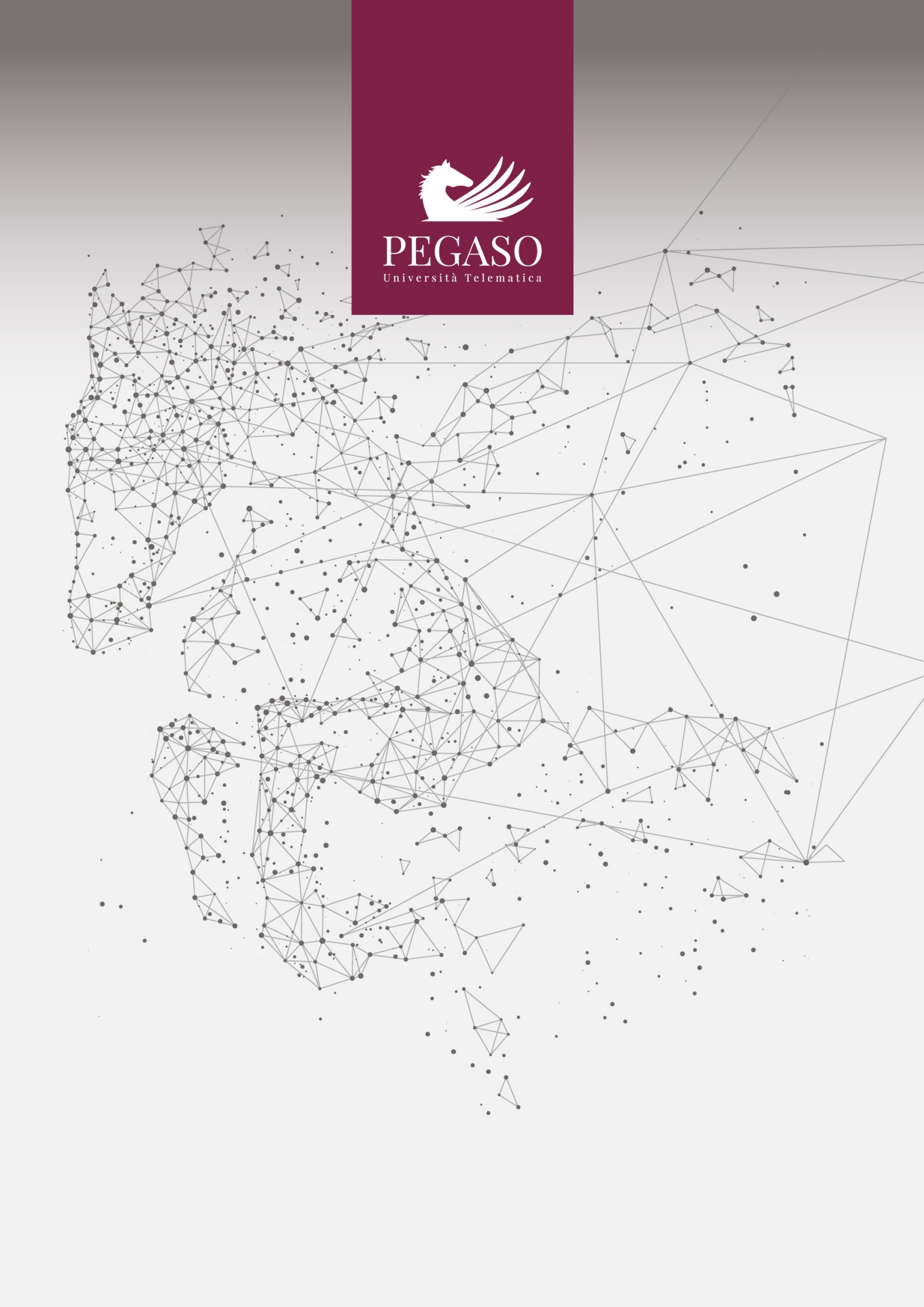
PEGASO
Università Telematica

Indice

1. PREMESSA 3
2. DAL MODELLO DI ANALISI ALLA STRUTTURA DEL SISTEMA 4
3. IDENTIFICAZIONE DEI SOTTOSISTEMI 5
4. IDENTIFICAZIONE DEGLI OBIETTIVI DI DESIGN 7
5. CRITERI DI DESIGN E COMPROMESSI 9
6. CONCLUSIONI E SINTESI 11
BIBLIOGRAFIA 12

1. Premessa

Questa lezione esplora un passaggio cruciale nel processo di **ingegneria del software**: la transizione dal modello di analisi al modello di design. Tale transizione non rappresenta semplicemente un cambiamento formale, ma una vera e propria **evoluzione concettuale**, in cui il progettista assume una nuova prospettiva: si passa dalla descrizione del problema (modellato attraverso oggetti, classi e relazioni concettuali) alla progettazione della **soluzione tecnica**. In questa nuova visione, l'attenzione si sposta dalle singole entità alla loro organizzazione in **unità strutturali più ampie e coese**, ovvero i sottosistemi, capaci di racchiudere in sé una responsabilità funzionale ben definita e autonomamente gestibile.

L'obiettivo fondamentale di questa fase è tradurre i **requisiti funzionali** e **non funzionali** raccolti durante l'analisi in un'architettura software concreta e realizzabile, capace di fungere da base per tutte le attività successive del ciclo di sviluppo: implementazione, test, deployment, manutenzione ed evoluzione del sistema. In altre parole, si costruisce una **visione tecnica e operativa del sistema**, che potrà guidare con precisione e coerenza tutte le decisioni progettuali e implementative future.

Il processo di design del sistema si articola secondo una sequenza di attività fondamentali: l'identificazione degli **obiettivi di design**, che esprimono le qualità desiderate del sistema in base ai requisiti e al contesto operativo; la definizione dei **criteri progettuali**, che forniscono linee guida per valutare le possibili scelte architetturali; e infine la **decomposizione del sistema** in sottosistemi, ognuno con compiti e confini ben delineati. Questo approccio consente di affrontare in modo strutturato la **complessità del software**, facilitando una progettazione modulare, mantenibile, scalabile e riutilizzabile.

Nel corso della lezione analizzeremo il processo di definizione dell'architettura, ponendo particolare attenzione ai **criteri di coesione**, **separazione delle responsabilità** e **minimizzazione delle dipendenze**, che costituiscono i fondamenti per un design robusto ed evolutivo.

2. Dal Modello di Analisi alla Struttura del Sistema

Il primo passo verso la costruzione dell'architettura di sistema è la **trasformazione del modello di analisi**—basato su oggetti, classi e relazioni concettuali—verso una struttura tecnica fatta di **componenti e sottosistemi**. Questo passaggio implica un cambio di paradigma: non si ragiona più su singole entità isolate, ma si cerca di costruire **unità logiche coese** che possano essere sviluppate, testate e mantenute in modo indipendente. Si passa quindi da una prospettiva descrittiva e statica, centrata sull'identificazione dei concetti fondamentali del dominio, a una visione più ingegneristica e orientata alla realizzazione concreta del software.

Nella pratica progettuale, questo passaggio si concretizza nella ridefinizione degli oggetti, delle classi e delle associazioni identificate in fase di analisi, che vengono riorganizzati in **componenti modulari**. Queste componenti non solo rappresentano raggruppamenti logici, ma diventano anche **unità di deployment e manutenzione**, ciascuna dotata di un'interfaccia esplicita e ben documentata. Una tale organizzazione consente di gestire con maggiore efficacia la complessità, distribuendo le responsabilità in maniera ordinata e facilitando il lavoro parallelo di team differenti.

In questo contesto, è cruciale individuare i **criteri per un raggruppamento efficace** degli oggetti. Le linee guida più utilizzate includono:

- **Funzionalità condivisa**: raggruppare oggetti che collaborano per realizzare uno stesso caso d'uso, garantendo coesione interna;
- **Dati condivisi**: identificare insiemi di oggetti che operano sugli stessi dataset, per assicurare consistenza e ottimizzazione nell'accesso ai dati;
- **Collaborazione intensa**: riconoscere quegli oggetti che comunicano frequentemente tra loro, minimizzando così la necessità di interfacce esterne;
- **Stesso ciclo di vita**: considerare oggetti che vengono creati, utilizzati e distrutti congiuntamente, così da semplificare la gestione della memoria e delle risorse.

Il risultato finale di questa attività è una **struttura del sistema robusta ed evolvibile**, nella quale le responsabilità sono chiaramente distribuite e i componenti sono progettati per essere riutilizzabili e sostituibili. Questa **modularizzazione** consente inoltre di rispondere in modo più efficace ai cambiamenti nei requisiti, migliorando la **scalabilità dell'architettura** e supportando una manutenzione agile e sostenibile. In sistemi complessi e in continua evoluzione, come quelli destinati a un'utenza ampia e variegata, una tale impostazione iniziale può fare la differenza tra un progetto gestibile e uno destinato a fallire.

3. Identificazione dei Sottosistemi

Un **sottosistema** è una parte autonoma e coesa del sistema, composta da un insieme di oggetti correlati che condividono una funzionalità comune e che interagiscono con il resto del sistema attraverso un'interfaccia pubblica ben definita. In pratica, ogni sottosistema rappresenta un'unità logica che incapsula una specifica area funzionale del software, fornendo un insieme di servizi o comportamenti che possono essere utilizzati da altri moduli. Questa strategia di suddivisione contribuisce in modo significativo alla **gestione della complessità del sistema**, alla sua **manutenibilità** e alla possibilità di effettuare evoluzioni incrementali nel tempo.

L'identificazione dei sottosistemi rappresenta un passaggio centrale nel design architetturale, poiché consente di **suddividere il sistema in unità logiche**, riducendo la complessità e facilitando il lavoro parallelo di diversi team. Ogni sottosistema può essere sviluppato, testato e documentato in maniera indipendente, agevolando non solo la **scalabilità del processo di sviluppo**, ma anche la sua **qualità complessiva**.

L'obiettivo è quello di minimizzare le dipendenze tra i sottosistemi, mantenendo la maggior parte della logica all'interno di contesti ben delimitati. Inoltre, tale separazione permette di applicare il **principio di singola responsabilità** e di isolare le aree di cambiamento, facilitando future modifiche o estensioni del sistema senza impatti trasversali.

Per identificare i sottosistemi in modo sistematico si adottano le seguenti tecniche:

- **Raggruppamento per casi d'uso**: ogni scenario operativo genera un insieme di oggetti da collocare nello stesso modulo;
- **Separazione delle responsabilità**: secondo livelli architetturali (presentazione, dominio, persistenza);
- **Minimizzazione delle dipendenze**: evitare legami diretti tra moduli;
- **Riconoscimento di oggetti condivisi**: questi vengono centralizzati in moduli comuni o gestiti tramite interfacce.

È fondamentale anche il principio di **information hiding**, secondo cui i dettagli interni di un sottosistema devono rimanere nascosti agli altri moduli, garantendo una forte **incapsulamento**. Questo permette di ridurre il rischio di dipendenze non controllate e di mantenere alta la **coesione interna** dei sottosistemi.

Il risultato è un'**architettura modulare**, in cui i sottosistemi interagiscono tramite **API pubbliche**, favorendo la manutenzione, il test, la riusabilità e la sostituzione dei componenti. Una tale organizzazione costituisce la base per un sistema resiliente e in grado di evolvere con efficienza nel tempo.

4. Identificazione degli Obiettivi di Design

Nel processo di **System Design**, l'individuazione degli **obiettivi di design** rappresenta una fase fondamentale, in quanto stabilisce le qualità che il sistema software dovrà garantire al di là delle funzionalità richieste. Tali obiettivi derivano principalmente dai **requisiti non funzionali**, dal **contesto operativo** in cui il sistema dovrà essere impiegato, nonché dalle **aspettative degli stakeholder**. Gli obiettivi di design non rispondono alla domanda "che cosa deve fare il sistema?", bensì a "come deve farlo?", ponendo quindi l'accento su **prestazioni, affidabilità, sicurezza, scalabilità, manutenibilità** e altri aspetti qualitativi.

La **formalizzazione esplicita degli obiettivi di design** serve a guidare in modo coerente le decisioni architetturali: ogni scelta progettuale deve poter essere ricondotta a uno o più obiettivi dichiarati. Inoltre, quando si presentano alternative progettuali equivalenti sul piano funzionale, saranno proprio gli obiettivi di design a suggerire quale opzione sia preferibile. Questa pratica favorisce anche la **tracciabilità delle decisioni progettuali**, in quanto ogni compromesso effettuato può essere giustificato rispetto agli obiettivi perseguiti. In fase di validazione, inoltre, la presenza di obiettivi chiari permette di valutare se il sistema risultante soddisfa davvero le aspettative iniziali.

Una classificazione utile per organizzare gli obiettivi di design include cinque macro-categorie:

1. **Prestazioni**: riguardano la rapidità e l'efficienza delle operazioni svolte dal sistema (es. tempo di risposta, throughput, utilizzo delle risorse);
2. **Affidabilità e sicurezza**: concernono la capacità del sistema di operare correttamente anche in presenza di anomalie, attacchi o condizioni ambientali avverse (es. tolleranza ai guasti, robustezza, protezione dei dati);
3. **Costo**: include gli aspetti economici legati allo sviluppo, alla distribuzione, alla formazione degli utenti, alla manutenzione e all'amministrazione;
4. **Manutenibilità**: indica la facilità con cui il sistema può essere modificato, esteso, adattato a nuovi contesti o letto e compreso da sviluppatori terzi (es. estendibilità, portabilità, leggibilità del codice);
5. **Esperienza utente**: valuta quanto il sistema sia realmente utile, facile da usare, accessibile e adattabile alle diverse esigenze dell'utenza finale (es. usabilità, personalizzazione, chiarezza dell'interfaccia).

Ogni sistema avrà il proprio profilo di priorità tra questi obiettivi, in funzione del dominio applicativo e dei vincoli esterni. In ambito aerospaziale o sanitario, ad esempio, la **sicurezza funzionale** e l'**affidabilità** avranno un peso preponderante; per un sistema embedded per l'automotive, la **robustezza** e

la **disponibilità** saranno imprescindibili; mentre in un'app mobile consumer, probabilmente la **responsività**, la **scalabilità** e l'**esperienza utente** saranno prioritari, anche in termini di competitività sul mercato.

5. Criteri di Design e Compromessi

Una volta stabiliti gli obiettivi di design, il progettista deve disporre di strumenti per valutare le alternative progettuali in modo sistematico. È qui che entrano in gioco i **criteri di design**, ossia un insieme di linee guida che traducono gli obiettivi in specifiche tecniche e misurabili. Questi criteri costituiscono una sorta di **grammatica della qualità** del software, rendendo possibile la comparazione tra differenti soluzioni architetturali. Essi permettono anche di standardizzare il linguaggio tecnico all'interno del team di sviluppo, agevolando il confronto tra soluzioni e facilitando la documentazione e la revisione delle decisioni progettuali.

I criteri principali si articolano nelle stesse cinque categorie viste in precedenza. Per esempio:

- **Performance**: include parametri fondamentali quali il tempo di risposta alle richieste, la capacità di gestire numerose operazioni contemporaneamente (throughput), e il consumo delle risorse di sistema (memoria, CPU, banda). In contesti come i sistemi in tempo reale, i requisiti di performance sono stringenti e devono essere trattati con priorità assoluta.
- **Dependability**: si riferisce all'affidabilità del sistema nel tempo, alla disponibilità continua dei servizi, alla capacità di resistere a guasti o anomalie (fault tolerance), alla protezione contro accessi non autorizzati (security) e alla tutela della sicurezza fisica degli utenti e dei beni (safety). Questo è particolarmente critico in settori regolamentati o in applicazioni mission-critical.
- **Costo**: comprende il costo di sviluppo iniziale, i costi di licenza e distribuzione, il costo di formazione degli utenti, quello legato alla manutenzione correttiva ed evolutiva, e infine i costi di amministrazione quotidiana del sistema. Le decisioni di design devono quindi essere allineate anche con le strategie economiche del progetto.
- **Manutenzione**: valuta quanto facilmente il sistema possa essere aggiornato, esteso o adattato a nuovi scenari. Rientrano in questa categoria anche la leggibilità del codice, la sua documentazione e la capacità di tracciare ogni modifica alle specifiche originali. Sistemi concepiti per vivere a lungo nel tempo devono puntare fortemente sulla manutenibilità.
- **Esperienza utente**: si concentra sull'interazione tra utente e sistema, prendendo in considerazione aspetti come l'intuitività dell'interfaccia, la curva di apprendimento, il livello di personalizzazione, e il supporto attivo al lavoro dell'utente. Questo aspetto è determinante per il successo di applicazioni consumer e sistemi ad alto tasso di interazione.

Tuttavia, **non tutti i criteri possono essere ottimizzati simultaneamente**. Spesso occorre operare dei **compromessi (trade-off)** tra obiettivi in conflitto. Ad esempio:

- Ottimizzare le prestazioni può comportare un aumento nell'uso della memoria o della potenza computazionale, rendendo il sistema meno sostenibile su dispositivi a risorse limitate;
- Aumentare la sicurezza, ad esempio attraverso cifrature avanzate e autenticazioni multiple, può rallentare l'usabilità e ostacolare l'esperienza utente;
- Migliorare la portabilità di un'applicazione può richiedere l'adozione di strumenti e librerie meno efficienti, limitando l'utilizzo di specifiche funzionalità native delle piattaforme target.

Il progettista deve essere in grado di **riconoscere e documentare questi compromessi**, discutendoli con gli stakeholder e motivandoli rispetto agli obiettivi principali. È buona prassi mantenere traccia delle decisioni in una sezione specifica della documentazione architetturale, esplicitando non solo il compromesso accettato, ma anche le motivazioni che ne giustificano la scelta. Questo approccio trasparente aiuta anche nei processi di revisione tecnica, audit e gestione del rischio.

Solo in questo modo sarà possibile realizzare un sistema che, pur non essendo ottimale su ogni fronte, risponda in modo equilibrato alle esigenze concrete del progetto, rispettando tempi, budget e aspettative degli utenti finali. Il valore di un'architettura non sta infatti nella perfezione teorica, ma nella sua **capacità di offrire un equilibrio sostenibile** tra qualità tecniche, costi e vincoli operativi.

6. Conclusioni e sintesi

Il percorso affrontato in questa lezione ha guidato attraverso le fasi chiave del **System Design**, evidenziando come sia possibile trasformare una rappresentazione concettuale del dominio in una struttura tecnica articolata e sostenibile. Dall'analisi degli oggetti e delle relazioni si è giunti alla **decomposizione del sistema in sottosistemi**, passando per l'identificazione di **obiettivi di design** e la valutazione dei **criteri progettuali**. Ogni passaggio ha avuto lo scopo di ridurre la complessità, aumentare la modularità e garantire qualità architetturale. È emersa con chiarezza la centralità della progettazione sistemica, intesa non solo come attività tecnica ma come processo strategico che traduce le esigenze del dominio applicativo in soluzioni reali, concrete e sostenibili nel tempo.

Uno dei concetti centrali emersi è che **un buon design non è mai casuale**, bensì frutto di **decisioni consapevoli**, orientate da obiettivi chiari e condivisi. Il progettista non si limita a costruire qualcosa che "funziona", ma crea una **struttura duratura**, in grado di resistere all'evoluzione dei requisiti, all'introduzione di nuove tecnologie e alle sfide operative. Il design efficace si fonda su una comprensione profonda del contesto d'uso, sull'analisi dei trade-off progettuali e sull'adozione di buone pratiche ingegneristiche. La **separazione delle responsabilità**, l'**incapsulamento**, la **coesione interna** e la **minimizzazione delle dipendenze** sono strumenti fondamentali per raggiungere questi risultati, consentendo al sistema di evolvere mantenendo intatta la propria affidabilità.

Infine, si è visto come la presenza di **criteri di design ben articolati** consenta di affrontare le scelte progettuali con maggiore rigore, e come la gestione dei **compromessi** tra criteri in conflitto sia una competenza chiave del progettista. L'architettura del software è quindi tanto un prodotto quanto un processo: una **mediazione continua** tra esigenze tecniche, economiche e umane. In questo senso, progettare significa costruire un equilibrio dinamico, una visione che sappia coniugare funzionalità, qualità e sostenibilità nel tempo. La progettazione di sistemi complessi è quindi una forma di pensiero ingegneristico, analitico e creativo al tempo stesso, capace di generare valore attraverso la strutturazione efficace dell'informazione e della responsabilità.

Bibliografia

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