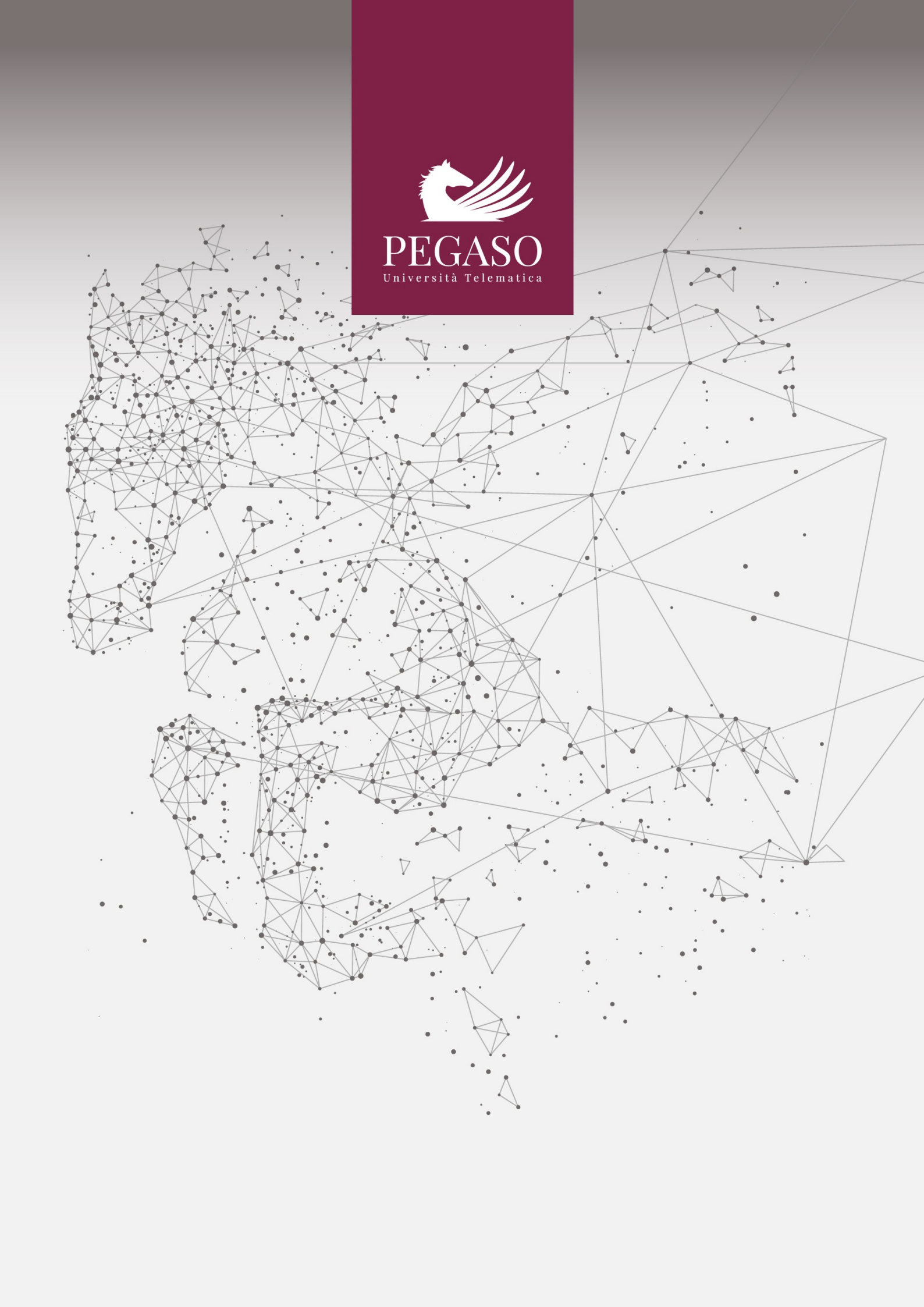
PEGASO
Università Telematica

Indice

1. Premessa

La progettazione dei sistemi, o **system design**, rappresenta una fase cruciale nello sviluppo di software complessi, in quanto costituisce il passaggio chiave tra l'analisi dei requisiti e l'implementazione concreta. Essa si occupa della strutturazione interna del sistema, definendo l'organizzazione dei suoi componenti principali con l'obiettivo di ottenere un'architettura che sia **modulare**, **manutenibile**, **scalabile**, ma anche **robusta** e **flessibile** rispetto ai cambiamenti futuri. In particolare, una buona progettazione consente di isolare le aree del sistema soggette a frequenti modifiche da quelle più stabili, minimizzando gli impatti collaterali delle evoluzioni funzionali o tecnologiche.

Nel contesto di questa lezione, esploreremo i concetti fondamentali della progettazione dei sistemi informatici con un approccio orientato agli oggetti. Analizzeremo nel dettaglio la **decomposizione in sottosistemi**, ovvero la suddivisione logica del sistema in componenti più piccoli e indipendenti, ciascuno dotato di una propria responsabilità funzionale. Approfondiremo la **definizione dei servizi** offerti da ciascun sottosistema e delle **interfacce** che ne regolano l'interazione, illustrando le modalità con cui queste contribuiscono a una progettazione modulare. Esamineremo poi le proprietà di **coesione** e **accoppiamento**, due parametri critici che influenzano direttamente la qualità e la manutenibilità dell'architettura software. Infine, discuteremo approcci strutturali consolidati come la **stratificazione** e il **partizionamento**, evidenziando come questi possano guidare l'organizzazione globale del sistema e supportare la separazione delle responsabilità.

Particolare attenzione sarà data all'impatto di queste scelte architetturali sulla **qualità del software**, con riferimento a metriche riconosciute e a **buone pratiche** derivate dall'esperienza industriale. Analizzeremo anche i principali **criteri di progettazione**, condivisi e formalizzati nella comunità ingegneristica, che permettono di prendere decisioni consapevoli, orientate alla realizzazione di sistemi **efficienti**, **affidabili** e **facilmente evolvibili** nel tempo.

2. Sottosistemi e Classi

La **decomposizione del sistema** in sottosistemi rappresenta una strategia essenziale per affrontare la complessità intrinseca dei sistemi software moderni. Un **sottosistema** è una parte autonoma del sistema che incapsula una specifica responsabilità funzionale ed è progettato per essere **indipendente** dagli altri. Questa indipendenza è ottenuta tramite **interfacce ben definite**, che consentono la comunicazione tra sottosistemi senza rivelare dettagli interni. I sottosistemi, a loro volta, sono composti da **classi**, che rappresentano le entità fondamentali della modellazione orientata agli oggetti. Una classe definisce uno **stato** (tramite attributi) e un **comportamento** (tramite metodi), ed è alla base della strutturazione del codice nel dominio della soluzione.

La relazione tra classi e sottosistemi è analoga a quella tra elementi atomici e molecole: le classi sono unità fondamentali, mentre i sottosistemi rappresentano **aggregazioni coerenti** di classi che cooperano per fornire una funzionalità. Questa organizzazione gerarchica consente di **ridurre la complessità cognitiva**, favorendo la **distribuzione del lavoro** tra team differenti e la **riusabilità** dei componenti. Ad esempio, in un sistema di gestione incidenti, possiamo identificare sottosistemi come l'interfaccia del dispatcher, la gestione delle risorse, la mappatura degli eventi e la notifica. Ognuno di questi raggruppa classi con responsabilità affini, permettendo una gestione autonoma e mirata. Inoltre, la possibilità di **decomporre ulteriormente** un sottosistema complesso in sottosistemi più semplici garantisce una maggiore granularità nella progettazione.

Questa modalità di organizzazione riflette un principio ingegneristico fondamentale: suddividere un problema complesso in problemi più piccoli e risolvibili in modo indipendente. I vantaggi sono molteplici: maggiore chiarezza nella gestione delle responsabilità, migliore tracciabilità delle funzionalità, incremento nella facilità di test e verifica, nonché una superiore capacità di adattamento a nuove esigenze o modifiche di requisiti. La **modularità** introdotta dalla suddivisione in sottosistemi favorisce anche l'adozione di metodologie agili, che si basano su iterazioni rapide e incrementali: ogni sprint può concentrarsi su un sottosistema specifico, mantenendo sotto controllo la complessità globale del progetto.

Un altro aspetto fondamentale riguarda la **documentazione** della decomposizione: ogni sottosistema deve essere documentato accuratamente, indipendentemente dal supporto offerto dal linguaggio di programmazione scelto. Questo è particolarmente rilevante in ambienti in cui la realizzazione dei sottosistemi è distribuita su team distinti. La documentazione deve includere non solo l'elenco delle classi appartenenti al sottosistema, ma anche la descrizione dei servizi offerti e delle dipendenze con altri componenti. Il concetto di sottosistema, pur essendo astratto, si traduce in elementi concreti del progetto,

come moduli, pacchetti o directory, a seconda del linguaggio adottato. In definitiva, la progettazione orientata ai sottosistemi rappresenta uno strumento potente per affrontare la complessità e promuovere l'**ingegnerizzazione del software** su larga scala.

3. Servizi e Interfacce dei Sottosistemi

Nel contesto della progettazione dei sistemi, la definizione dei **servizi** e delle **interfacce** rappresenta uno degli elementi più critici per garantire **modularità**, **flessibilità** e **interoperabilità** tra i componenti del sistema. Un **servizio** è un insieme coerente di operazioni che un sottosistema mette a disposizione degli altri, con uno scopo ben definito e comprensibile. Le **interfacce**, invece, costituiscono il mezzo attraverso il quale questi servizi sono resi accessibili. Definire correttamente i servizi e le interfacce permette di ottenere sistemi altamente **coesi** internamente ma **debolmente accoppiati** tra loro, favorendo la scalabilità e la manutenzione del software. Inoltre, queste definizioni sono il punto di partenza per ogni strategia di **integrazione continua** e di **testing automatico**, poiché consentono di simulare il comportamento dei sottosistemi attraverso l'uso di stub o mock.

Durante la fase di **system design**, si pone l'accento su cosa ogni sottosistema deve offrire, piuttosto che su come queste funzionalità saranno implementate. Questa separazione concettuale consente di definire le **responsabilità** di ciascun componente in termini di **contratti funzionali** che devono essere rispettati, indipendentemente dai dettagli realizzativi. Ad esempio, un sottosistema dedicato alla **gestione delle risorse** in un sistema di emergenza può offrire servizi per localizzare, allocare e aggiornare lo stato delle risorse (come ambulanze o autopompe), senza che gli altri sottosistemi debbano conoscere il modo in cui tali operazioni sono effettivamente realizzate. Questo tipo di astrazione è cruciale per garantire l'**intercambiabilità dei moduli** e per supportare l'evoluzione del sistema nel tempo.

Le **interfacce dei sottosistemi** comprendono la definizione dei nomi delle operazioni, i parametri in ingresso, i tipi restituiti e i possibili messaggi di errore. Una buona progettazione delle interfacce mira a mantenere **l'indipendenza dell'implementazione**, evitando riferimenti a strutture dati interne o algoritmi specifici. Questo approccio riduce sensibilmente l'impatto delle modifiche future, poiché eventuali cambiamenti interni non propagano effetti collaterali agli altri componenti. Le interfacce, dunque, agiscono come **barriere di protezione** tra sottosistemi, favorendo l'evoluzione indipendente dei moduli. In ambienti distribuiti o altamente dinamici, la **stabilità dell'interfaccia** diventa un criterio prioritario, poiché un cambiamento imprevisto può compromettere l'interoperabilità tra moduli sviluppati da team o fornitori diversi.

Dal punto di vista pratico, la definizione delle interfacce è supportata da notazioni formali come i **diagrammi UML**. Tra queste, la notazione **ball-and-socket** è particolarmente utile per rappresentare le dipendenze tra componenti: l'elemento "ball" (o lollipop) indica un'interfaccia **fornita** da un sottosistema, mentre il "socket" rappresenta un'interfaccia **richiesta**. Questa rappresentazione visiva consente di

tracciare in maniera chiara le **relazioni di servizio** tra i componenti, rendendo esplicite le aspettative e le responsabilità. Ad esempio, il sottosistema dell'interfaccia utente di un operatore può richiedere un servizio di aggiornamento delle risorse, offerto dal sottosistema di gestione risorse; questa relazione è facilmente modellabile con un collegamento ball-and-socket. In fase avanzata, la mappatura tra i diagrammi e le **specifiche tecniche formali** (come WSDL per servizi web o IDL per sistemi CORBA) consente di passare dalla modellazione astratta all'implementazione concreta.

Un ulteriore passaggio fondamentale avviene nella transizione dalla progettazione di sistema alla progettazione orientata agli oggetti: i servizi vengono raffinati e formalizzati come **API (Application Programming Interface)**. In questa fase, l'attenzione si sposta sull'implementazione concreta delle operazioni, includendo dettagli tecnici come la gestione delle eccezioni, il controllo dei tipi e le politiche di sicurezza. Tuttavia, il principio di **separazione delle responsabilità** rimane centrale: l'interfaccia continua a rappresentare un **contratto** tra componenti, la cui stabilità è essenziale per il corretto funzionamento del sistema.

In conclusione, una progettazione accurata dei servizi e delle interfacce dei sottosistemi non solo migliora la qualità architetturale del sistema, ma costituisce anche una base solida per la futura evoluzione e manutenzione. Essa permette di raggiungere un equilibrio virtuoso tra **trasparenza funzionale**, **incapsulamento**, **riusabilità** e **manutenibilità**, elementi chiave per la realizzazione di software industriale robusto ed efficace. La chiarezza e la precisione nella definizione delle interfacce sono, in definitiva, indicatori fondamentali della **maturità architetturale** di un progetto software.

4. Coesione e Accoppiamento

Nel contesto della progettazione software, i concetti di **coesione** e **accoppiamento** rappresentano due pilastri fondamentali per valutare e guidare la qualità dell'architettura di un sistema. La **coesione** misura il grado con cui gli elementi interni a un modulo (o sottosistema) collaborano per realizzare un'unica responsabilità funzionale. L'**accoppiamento**, invece, quantifica il livello di dipendenza tra moduli differenti. Un buon design mira a massimizzare la coesione e minimizzare l'accoppiamento, ottenendo così un sistema modulare, flessibile e facilmente manutenibile. Questo equilibrio diventa particolarmente cruciale nei sistemi su larga scala, distribuiti o soggetti a frequenti evoluzioni.

Una **coesione elevata** implica che tutte le classi e le funzioni di un sottosistema lavorano insieme per raggiungere uno scopo comune. Questo si traduce in una maggiore chiarezza concettuale, codice più leggibile e semplice da mantenere, e una più alta probabilità di riutilizzo del modulo in contesti diversi. Per esempio, un modulo per la gestione degli utenti che comprende solo le funzionalità di autenticazione, gestione dei profili e ruoli, mostra una coesione elevata. Al contrario, un modulo che include funzionalità disparate come gestione utenti, ordini e pagamenti, presenta una coesione bassa, rendendo il codice più difficile da gestire e testare. Una bassa coesione può inoltre rendere più complessa la suddivisione del lavoro tra team e rallentare i cicli di sviluppo.

L'**accoppiamento**, d'altra parte, riguarda la quantità e la natura delle dipendenze tra moduli. Un **accoppiamento debole** indica che i moduli interagiscono tra loro tramite interfacce ben definite, senza accedere direttamente agli elementi interni degli altri. Questo rende possibile modificare, sostituire o estendere un modulo senza dover apportare modifiche agli altri. Un **accoppiamento forte**, invece, implica una stretta interdipendenza, spesso attraverso l'accesso diretto a strutture dati o funzioni interne, con conseguenze negative in termini di manutenzione, test e riuso. Inoltre, un forte accoppiamento può ostacolare il riutilizzo del modulo in altri progetti e rendere più complesso l'adattamento a nuovi requisiti.

Il giusto bilanciamento tra coesione e accoppiamento è spesso soggetto a **compromessi progettuali**. A volte, per aumentare la coesione, è necessario introdurre un maggior numero di sottosistemi, aumentando così il numero di interfacce e, potenzialmente, il livello di accoppiamento. In questi casi, è fondamentale adottare **strategie di astrazione** e **definizione chiara delle responsabilità**, per contenere l'accoppiamento entro livelli accettabili. Ad esempio, un modulo per la gestione degli ordini può comunicare con il modulo delle spedizioni attraverso un'API stabile, che fornisce tutte le operazioni necessarie, senza esporre i dettagli interni del modulo spedizioni. Il design dell'interfaccia deve quindi

prevedere un set di operazioni ben progettato, in grado di esprimere tutte le esigenze funzionali senza richiedere conoscenza del funzionamento interno.

Un caso interessante si verifica quando si introduce un livello di astrazione intermedio, come nel caso di un **modulo Storage** che funge da intermediario tra il sottosistema logico e la base dati. In questo modo, eventuali cambiamenti nel database (ad esempio, un cambio di tecnologia o di schema) richiedono modifiche solo nel modulo Storage, e non in tutti i moduli client. Questo approccio dimostra come sia possibile ridurre l'accoppiamento, migliorando al contempo l'isolamento dei cambiamenti e la **robustezza architetturale**. Inoltre, può consentire l'utilizzo di sistemi eterogenei o la migrazione graduale verso nuove tecnologie, senza compromettere il funzionamento dell'intero sistema.

In conclusione, il principio guida di una buona progettazione è quello di creare moduli **altamente coesi e debolmente accoppiati**. Questa combinazione consente una maggiore facilità di sviluppo parallelo, una minore probabilità di errori regressivi e una maggiore adattabilità del sistema a nuove esigenze. La padronanza di questi concetti è essenziale per ogni ingegnere del software che miri a costruire sistemi **sostenibili nel tempo**, **resilienti ai cambiamenti**, e **progettati per evolversi** in modo efficiente e controllato.

5. Stratificazione e Partizionamento

La progettazione di un sistema software non si limita alla suddivisione in sottosistemi, ma richiede anche una riflessione sull'**organizzazione complessiva** dell'architettura. In questo contesto, le tecniche di **stratificazione** (layering) e **partizionamento** (partitioning) rappresentano due approcci complementari per gestire la complessità e promuovere la modularità. Esse consentono di distribuire le responsabilità funzionali in modo strutturato e chiaro, facilitando il lavoro di team multipli e il mantenimento a lungo termine del sistema. In architetture moderne e distribuite, queste tecniche risultano particolarmente efficaci per supportare l'evoluzione incrementale del sistema, la scalabilità orizzontale e il deploy selettivo dei componenti.

La **stratificazione** consiste nella suddivisione del sistema in **livelli verticali**, ciascuno dei quali fornisce servizi al livello superiore e utilizza i servizi del livello inferiore. I livelli sono organizzati in modo gerarchico, secondo un principio di **dipendenza unidirezionale**: un livello non può accedere direttamente a quelli superiori. Questo approccio permette di isolare le preoccupazioni funzionali e favorisce il riuso, l'indipendenza e la sostituibilità dei componenti. Un esempio classico è l'architettura a tre livelli: **Presentation Layer** (interfaccia utente), **Application Layer** (logica di business), e **Data Layer** (accesso ai dati). Questo modello è ampiamente usato in ambito enterprise per garantire flessibilità, facilità di testing e separazione delle responsabilità.

La stratificazione può essere di due tipi: **chiusa** e **aperta**. In una stratificazione **chiusa** (opaque layering), ogni livello interagisce solo con il livello immediatamente sottostante. Questo garantisce una forte separazione delle responsabilità e semplifica la manutenzione, poiché le modifiche a un livello non si propagano facilmente agli altri. Al contrario, una stratificazione **aperta** (transparent layering) permette a un livello di accedere direttamente a più livelli sottostanti. Questa flessibilità può essere utile in presenza di esigenze di **ottimizzazione delle prestazioni** o di **personalizzazione funzionale**, ma comporta un rischio maggiore di accoppiamento. La scelta tra i due approcci dipende dalla natura del sistema e dai requisiti di manutenibilità, sicurezza e prestazioni.

Il **partizionamento**, invece, si concentra sulla suddivisione **orizzontale** del sistema in sottosistemi o moduli **peer-to-peer**, ognuno dei quali è responsabile di una specifica funzionalità. A differenza della stratificazione, i moduli partizionati non sono organizzati gerarchicamente e possono interagire tra loro in maniera più libera. Questo approccio è ideale quando il sistema deve gestire un insieme di servizi eterogenei, indipendenti ma cooperanti, come in un sistema di e-commerce con moduli per gestione

utenti, ordini, pagamenti e spedizioni. In questi scenari, il partizionamento consente una migliore gestione del ciclo di vita dei singoli moduli, favorendo il riutilizzo e la sostituzione modulare.

Un vantaggio rilevante del partizionamento è la possibilità di **parallelizzare lo sviluppo**: team distinti possono lavorare contemporaneamente su moduli diversi, riducendo i tempi di consegna. Inoltre, i moduli partizionati possono essere **riutilizzati** in altri sistemi o applicazioni. Tuttavia, l'assenza di una struttura gerarchica richiede un'attenta definizione delle interfacce e dei protocolli di comunicazione, per evitare un'eccessiva complessità nelle interazioni tra i moduli. Una scarsa standardizzazione può infatti portare a un aumento del debito tecnico e a difficoltà di integrazione tra componenti sviluppati in modo indipendente.

Nella pratica, un'architettura efficace combina spesso stratificazione e partizionamento. La progettazione parte tipicamente da una **partizione ad alto livello** in sottosistemi principali, ciascuno dei quali può essere **stratificato internamente** per organizzare meglio la logica e le dipendenze. Questo approccio misto consente di bilanciare **specializzazione funzionale** e **controllo delle dipendenze**, migliorando la **manutenibilità**, la **scalabilità** e la **trasparenza** dell'intera architettura. Tali sistemi risultano più semplici da comprendere, testare e aggiornare, anche in contesti caratterizzati da requisiti in continua evoluzione.

Infine, è fondamentale valutare la **profondità della stratificazione**: secondo il principio cognitivo dei "7 ± 2 elementi", un'architettura non dovrebbe contenere più di 7 livelli o moduli per livello, per evitare una complessità eccessiva nella comprensione e gestione del sistema. Una buona stratificazione, unita a un partizionamento efficace, rappresenta quindi un potente strumento per dominare la complessità del software moderno. L'efficacia di questi approcci non risiede solo nella loro struttura logica, ma anche nella loro capacità di adattarsi alle necessità di evoluzione, interoperabilità e manutenzione a lungo termine dei sistemi informativi.

6. Conclusioni e sintesi

In questa lezione abbiamo evidenziato l'importanza di un approccio sistematico alla **decomposizione**, all'**organizzazione architetturale** e alla **definizione delle interfacce**. Ogni concetto affrontato – dalla scomposizione in sottosistemi, alla gestione della coesione e dell'accoppiamento, fino alle tecniche di stratificazione e partizionamento – contribuisce in modo decisivo a costruire sistemi più **robusti**, **scalabili** e **facilmente manutenibili**.

Uno dei principi cardine emersi è che un sistema ben progettato nasce da **decisioni consapevoli** sul piano architetturale. La capacità di suddividere la complessità in elementi comprensibili, attraverso sottosistemi dotati di **interfacce ben definite**, consente di isolare le responsabilità e facilitare lo sviluppo parallelo. In questo modo, più team possono lavorare contemporaneamente su parti diverse del sistema, riducendo i tempi complessivi di sviluppo. Questo approccio si rivela particolarmente vantaggioso nei contesti industriali complessi, dove la scalabilità organizzativa si riflette direttamente nella scalabilità tecnica della soluzione.

Un altro elemento centrale è rappresentato dalla **qualità delle interfacce**. Interfacce chiare e stabili, fondate su una corretta definizione dei servizi, consentono un'elevata **indipendenza tra i componenti**, condizione necessaria per favorire la modularità e il riuso. La possibilità di modificare l'implementazione interna di un sottosistema senza impatti sui consumatori delle sue interfacce è uno dei tratti distintivi delle architetture professionali e longeve. In un'ottica di evoluzione continua, l'adozione di interfacce stabili rappresenta un prerequisito per l'adozione di pratiche DevOps, continuous integration e continuous delivery.

I concetti di **coesione** e **accoppiamento** offrono una guida essenziale nella valutazione della qualità strutturale del sistema. Un'elevata coesione assicura che ogni modulo sia ben focalizzato e facilmente gestibile, mentre un basso accoppiamento garantisce la separazione tra i moduli, riducendo la propagazione degli errori e dei cambiamenti. Insieme, questi due concetti supportano l'agilità e la sostenibilità del software nel tempo. In fase di refactoring, questi parametri rappresentano metriche chiave per guidare miglioramenti incrementali e per orientare la modularizzazione del codice.

Infine, abbiamo osservato come **stratificazione** e **partizionamento** siano strumenti complementari per strutturare l'architettura a livello macro. Mentre la stratificazione aiuta a organizzare il flusso di dipendenze verticali, il partizionamento permette di raggruppare funzionalità orizzontali indipendenti. Combinati, questi due approcci rendono possibile costruire architetture che siano al contempo **modulari**, **estensibili** e **adattabili ai cambiamenti**. Nei sistemi distribuiti, questa combinazione è particolarmente

potente, in quanto consente sia una governance chiara delle responsabilità, sia l'adattabilità dell'infrastruttura tecnica.

In sintesi, un sistema ben progettato non è frutto del caso, ma della **padronanza dei principi architetturali** e della loro applicazione coerente in ogni fase dello sviluppo. La qualità dell'ingegneria del software dipende dalla capacità di operare scelte progettuali che bilancino esigenze funzionali, evolutive e operative. Solo così è possibile costruire sistemi realmente **conquistatori della complessità**, capaci di affrontare con successo le sfide del mondo reale. La progettazione architetturale, quindi, non è solo una fase tecnica ma un'attività strategica che incide direttamente sulla longevità e sul successo di qualsiasi progetto software.

Bibliografia

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