Ingegneria del Software

Cos'è l'Ingegneria del Software

L'ingegneria del software è un ramo dell'informatica che si occupa della progettazione, sviluppo, manutenzione e riutilizzo dei programmi informatici (software). L'obiettivo principale è quello di realizzare software di alta qualità, riducendo al contempo i costi di produzione e facilitando la gestione nel tempo.

Un software non è semplicemente un insieme di programmi: include anche tutta la documentazione necessaria per la sua progettazione, realizzazione, utilizzo e manutenzione.

Le fasi fondamentali

L'intero processo di sviluppo del software è suddiviso in diverse fasi, che nel loro insieme costituiscono il **ciclo di vita del software**. Queste fasi permettono di organizzare il lavoro in maniera sistematica e controllata.

1. Analisi

Questa è la fase iniziale e una delle più importanti. Serve a comprendere cosa vuole il cliente e quali sono le esigenze reali. Non si tratta solo di raccogliere idee, ma di formalizzarle in modo chiaro e strutturato.

Durante l'analisi si realizzano diversi tipi di studio:

- **Analisi di fattibilità**: stabilisce se il progetto è realizzabile sia dal punto di vista tecnico che economico.
- **Analisi dei requisiti**: definisce in modo dettagliato cosa deve fare il sistema (requisiti funzionali) e con quali vincoli (requisiti non funzionali).
- **Analisi dell'interfaccia utente**: studia come sarà l'interazione tra l'utente e il sistema, rendendola chiara e accessibile.
- **Analisi dell'architettura**: inizia a definire la struttura generale del sistema.
- **Analisi dei dati**: individua e struttura i dati che saranno usati.
- **Analisi dei test**: pianifica le prove per verificare il corretto funzionamento.

2. Progettazione

Questa fase si basa sui risultati dell'analisi. Serve a stabilire come sarà costruito il sistema:

- **Progettazione architetturale**: definisce le componenti principali del sistema e come queste interagiranno tra loro.
- **Progettazione di dettaglio**: specifica nel dettaglio ogni componente, descrivendo algoritmi, strutture dati, flussi di lavoro e interazioni.

3. Realizzazione

Durante questa fase, il software viene effettivamente scritto e realizzato:

- **Programmazione**: si codifica il sistema nel linguaggio scelto.
- **Primi test**: si correggono errori evidenti (bug sintattici o logici).
- **Integrazione e test di sistema**: le varie componenti vengono assemblate e testate insieme, per verificarne la compatibilità e il corretto funzionamento complessivo.

4. Rilascio e avviamento

Questa è la fase in cui il software viene consegnato e messo in funzione:

- **Rilascio del prodotto**: installazione presso il cliente e consegna della documentazione.
- **Collaudo**: verifica del funzionamento con dati reali e primi corsi formativi per gli utenti.
- **Avviamento**: il sistema entra in funzione; il fornitore supporta il cliente nella fase iniziale.

5. Manutenzione

Il software viene mantenuto aggiornato e funzionante per tutto il suo ciclo di vita:

- **Manutenzione correttiva**: risoluzione di errori rilevati dopo il rilascio.
- **Manutenzione adattativa**: modifiche per adattarsi a nuove normative o condizioni.
- **Manutenzione evolutiva**: aggiunta di nuove funzionalità o miglioramenti.

Gli attori coinvolti

Nel processo di sviluppo del software sono coinvolti diversi soggetti, detti **stakeholder**:

- **Clienti**:
  - *Utente finale*: chi usa direttamente il sistema.
  - *Committente*: chi commissiona e finanzia il progetto.
- **Fornitori**:
  - Tecnici, analisti, progettisti, programmatori, commerciali.

La comunicazione tra clienti e fornitori è spesso la causa di problemi: per questo è fondamentale usare un linguaggio comune e formalizzare le decisioni prese.

La qualità del software

Un software può essere definito di qualità se:

1. Risponde davvero alle esigenze del cliente.
2. È facilmente manutenibile nel tempo.
3. È accompagnato da documentazione completa e aggiornata.

Importanza delle fasi iniziali

Secondo Barry Boehm, rilevare e correggere un errore in fase di analisi costa molto meno rispetto a farlo dopo il rilascio (fino a 200 volte meno). Questo dimostra quanto sia importante investire tempo e attenzione nella fase di analisi e progettazione.

Riassunto Esteso: Casi d'Uso, Scenari e User Stories

I requisiti funzionali rappresentano i comportamenti attesi di un sistema informatico e descrivono ciò che il sistema deve fare in termini di servizi e risposte agli stimoli esterni. Uno degli strumenti più diffusi per descrivere i requisiti funzionali è il **caso d'uso** (*use case*), largamente impiegato nel modello di sviluppo orientato agli oggetti e formalizzato tramite il linguaggio UML.

Casi d'Uso e Attori

Un **caso d'uso** descrive una sequenza di azioni che il sistema compie per fornire un servizio a un attore, ovvero un'entità esterna che interagisce con il sistema. Gli attori possono essere:

- **Primari**: iniziano l'interazione con il sistema per ottenere un determinato risultato.
- **Secondari**: supportano il sistema nel fornire il servizio ma non iniziano l'interazione.

Ogni caso d'uso deve essere:

- scritto dal punto di vista dell'attore;
- delimitato da confini chiari (inizio e fine dell'interazione);
- auto-contenuto, ossia descrivere una sequenza logica completa;
- focalizzato su comportamenti osservabili e misurabili.

Viene spesso utilizzata una descrizione narrativa per evidenziare come il sistema risponde a determinati eventi.

Scenari: Definizione e Tipologie

Ogni caso d'uso è composto da uno o più **scenari**, che rappresentano varianti specifiche del comportamento del sistema. Gli scenari descrivono interazioni concrete tra attori e sistema e aiutano ad esplorare più a fondo le possibili situazioni operative.

Le principali tipologie di scenario sono:

- **As-is**: fotografano la situazione attuale, utile per identificare criticità e inefficienze.
- **Visionary**: mostrano come il sistema dovrebbe funzionare in futuro, una volta implementato.
- **Evaluation**: utilizzati nei test, descrivono situazioni da simulare per verificare il corretto funzionamento.
- **Training**: finalizzati alla formazione degli utenti, illustrano casi pratici d'uso.

Per costruire scenari validi è necessario analizzare le attività svolte dagli utenti, i dati in gioco e le interazioni possibili.

Diagrammi dei Casi d'Uso e Relazioni

I **diagrammi dei casi d'uso UML** sono strumenti grafici che rappresentano attori, casi d'uso e le relazioni tra di essi. Gli attori sono visualizzati come omini stilizzati, mentre i casi d'uso sono ellissi con il nome del caso al centro. Le linee che li collegano rappresentano le interazioni.

Tra i casi d'uso possono esistere relazioni che aiutano a semplificare e strutturare meglio il diagramma:

- **Inclusione (<<include>>)**: permette di riutilizzare una sequenza comune di azioni in più casi d'uso. È utile quando diversi casi condividono uno stesso sotto-processo.
- **Estensione (<<extend>>)**: definisce comportamenti opzionali che si attivano solo in certe condizioni. Aiuta a mantenere i diagrammi ordinati e a evitare ridondanze.

Queste relazioni rendono il modello più modulare e favoriscono il riutilizzo e la chiarezza.

Documentazione di un Caso d'Uso

Ogni caso d'uso viene documentato seguendo una struttura standard che garantisce uniformità e facilita la lettura da parte degli stakeholder:

- **Nome**: identificativo del caso.
- **Descrizione**: obiettivo generale e funzione svolta.
- **Attori coinvolti**: elenco di attori primari e secondari.
- **Scenario principale**: flusso regolare delle operazioni.
- **Estensioni (variazioni)**: eventuali alternative, eccezioni o errori.
- **Precondizioni**: condizioni necessarie prima di iniziare il caso.
- **Postcondizioni**: risultati attesi alla fine del caso.
- **Requisiti speciali**: vincoli tecnici o normativi.
- **Autore/data**: persona che ha scritto il caso e data di redazione.

Questa documentazione è utile per lo sviluppo, il collaudo e la manutenzione del sistema.

Le User Stories nel Modello Agile

Nel contesto **agile**, i requisiti vengono descritti mediante **user stories**, una forma sintetica e flessibile per rappresentare le esigenze dell'utente. Ogni user story si compone di tre parti:

1. **Situazione (as)**: chi compie l'azione (es. utente, amministratore);
2. **Motivazione (I want)**: cosa vuole fare;
3. **Aspettativa (so that)**: quale beneficio si aspetta.

Le user stories non sono casi d'uso completi ma ne rappresentano un'estensione semplificata, utile a evidenziare le funzionalità attese in termini di valore per l'utente.

Ogni story deve essere:

- chiara e comprensibile,
- realizzabile entro una o due iterazioni,
- collegata a criteri di accettazione condivisi.

Le user stories vengono raccolte nel **backlog**, uno strumento fondamentale del metodo agile, e sono considerate **partizioni verticali** perché includono una funzionalità completa dal punto di vista dell'utente. Tuttavia, per motivi pratici, si tende spesso a organizzarle **orizzontalmente** secondo criteri di efficienza e facilità di implementazione.

Questo approccio favorisce una gestione più dinamica e iterativa dei requisiti durante tutto il ciclo di sviluppo.

Tipi di Raccolta dei Requisiti

A seconda del progetto, la raccolta dei requisiti degli attori coinvolti può seguire tre principali approcci:

- **Greenfield engineering**: lo sviluppo parte da zero, senza un sistema precedente da sostituire. I requisiti vengono forniti dall'azienda committente e dai suoi stakeholder. Lo sviluppatore valuta anche eventuali soluzioni già disponibili sul mercato.
- **Re-engineering**: si lavora su un sistema esistente che necessita di essere riprogettato per motivi tecnologici o funzionali. Si analizzano pregi e difetti del sistema attuale per mantenere, migliorare o migrare funzionalità.
- **Interface engineering**: si aggiorna solo l'interfaccia utente, mantenendo intatto il sistema legacy. I requisiti non cambiano, ma si riprogetta la presentazione.

Fase di Esplorazione

Questa fase è fondamentale per analizzare a fondo il problema da risolvere. Serve a scomporre ogni aspetto, capire tutti i bisogni e definire le priorità. Viene anche detta "elicitation" o "discovery", in quanto i requisiti spesso emergono solo tramite un'attenta analisi.

Per redigere un documento dei requisiti efficace si parte dall'esaminare le richieste del committente, considerato il principale referente. Successivamente, si identificano e si coinvolgono gli stakeholder, ovvero coloro che avranno un interesse o saranno influenzati dal sistema.

Stakeholder Engagement

Engagement significa coinvolgimento attivo degli stakeholder, non una semplice raccolta di opinioni. Gli ingegneri del software instaurano un dialogo per integrare i bisogni degli attori nelle decisioni progettuali. Poiché questi operano su livelli diversi, offrono prospettive che potrebbero rivelare criticità altrimenti ignorate.

È importante garantire una comunicazione interattiva, evitando approcci passivi o imposti. Gli stakeholder devono sentirsi parte del progetto. Per ottenere risultati utili, devono essere scelti con attenzione, assicurando rappresentatività e inclusività. La raccolta dei requisiti prende il nome di requirements elicitation e richiede di analizzare il sistema da tutti i viewpoint possibili.

Tecniche di Esplorazione

Le tecniche usate in questa fase sono molteplici, ciascuna con vantaggi e limiti:

- Le **interviste individuali** approfondiscono aspetti specifici tramite un confronto diretto. Richiedono tempo, ma permettono un controllo puntuale.
- I **focus group** fanno emergere diverse opinioni, evidenziando conflitti e aree di consenso. La loro efficacia dipende dalla regia del facilitatore.
- Le **osservazioni sul campo** permettono di cogliere l'effettivo comportamento degli utenti, mostrando attività spesso non dichiarate.
- I **suggerimenti spontanei** sono contributi che arrivano dagli utenti senza sollecitazione, utili per cogliere miglioramenti precisi.
- I **questionari** raggiungono molti utenti e consentono analisi statistiche, ma devono essere ben progettati e soffrono di bassa affidabilità.
- L'**analisi della concorrenza e delle best practice** consente di adottare soluzioni già collaudate, evitando errori e riducendo costi.
- I **casi d'uso** descrivono in dettaglio le operazioni del sistema e sono fondamentali anche per i test.

Interviste Individuali

Questa tecnica è una delle più efficaci. Si parte dal committente, che stabilisce obiettivi e tempi, e si estende agli stakeholder rilevanti. Più persone vengono ascoltate, più è probabile raccogliere informazioni utili. Tuttavia, possono emergere contraddizioni, quindi è fondamentale selezionare gli intervistati con criterio, suddividendoli in gruppi omogenei.

Strutturazione delle Interviste

Le interviste si distinguono in:

- **Non strutturate**: conversazioni aperte che consentono flessibilità e approfondimento.
- **Strutturate**: domande fisse, adatte a raccogliere dati comparabili.
- **Semi-strutturate**: combinazione dei due approcci, con domande aperte e chiuse.

La riuscita di un'intervista dipende dalla competenza dell'intervistatore, che deve mettere a proprio agio l'intervistato, evitando di influenzarlo e facilitando l'espressione di requisiti impliciti.

Problemi nella Fase di Esplorazione

Durante la raccolta dei requisiti emergono diversi ostacoli:

- Spesso gli utenti confondono desideri con bisogni reali.
- Le barriere comunicative derivano da linguaggi tecnici diversi.
- È difficile trovare il giusto livello di dettaglio: si rischia di restare vaghi o essere troppo specifici.
- I requisiti possono essere espressi in modo contraddittorio da stakeholder diversi.
- La **volatilità** dei requisiti porta a modifiche nel tempo a causa di fattori esterni o interni.

Per questo, i requisiti devono essere sempre validati con il committente prima di procedere oltre nello sviluppo.

Requisiti Software e Stakeholder

Un requisito è una proprietà richiesta o auspicabile del prodotto. Il documento dei requisiti contiene una descrizione di tutte le proprietà desiderate e include ogni informazione circa la funzionalità, i servizi, le modalità operative e di gestione del sistema da sviluppare. La definizione dei requisiti rappresenta un'analisi completa dei bisogni dell'utente e del dominio del problema, allo scopo di determinare cosa il sistema deve fare.

La raccolta dei requisiti è un processo complesso, che coinvolge membri del team di sviluppo, rappresentanti dell'azienda cliente e, talvolta, consulenti esterni. Gli stakeholder sono tutti coloro che, a vario titolo e livello organizzativo, hanno un interesse nella realizzazione del sistema. Ogni stakeholder può offrire una visione generale del sistema, che l'analista deve poi interpretare e formalizzare in termini tecnici e dettagliati.

È importante considerare anche i fattori esterni al sistema, poiché i requisiti non sono fissi ma possono evolversi durante tutto il ciclo di sviluppo. Errori in questa fase di analisi possono compromettere l'intero progetto. La normativa ISO 13407 (Human-centred design process) fornisce una guida all'identificazione dei requisiti e degli obiettivi di un sistema usabile, considerando:

- le prestazioni richieste dal sistema in relazione agli obiettivi operativi ed economici;
- i requisiti normativi e legislativi, incluse le normative su sicurezza e salute;
- la comunicazione e cooperazione tra utenti e altri attori rilevanti;
- le attività degli utenti, compresa la ripartizione dei compiti;
- la progettazione dei flussi di lavoro e dell'organizzazione;
- la gestione del cambiamento, incluse attività di formazione e personale coinvolto;
- la fattibilità delle operazioni, incluse quelle di manutenzione;
- la progettazione dei posti di lavoro e dell'interfaccia uomo-computer.

Classificazione dei Requisiti

I requisiti software si classificano secondo due punti di vista principali:

Livello di dettaglio

- **Requisiti utente**: espressi in linguaggio comprensibile al cliente, riflettono le necessità percepite dall'utente finale. Sono presentati al team di sviluppo, che propone soluzioni anche alternative. Sono detti anche "requisiti aperti".
- **Requisiti di sistema**: imposti da vincoli tecnici, normativi o infrastrutturali. Sono scritti in linguaggio tecnico, spesso formale o semi-formale, e non lasciano margini all'inventiva. Possono non essere noti all'utente ma ben conosciuti dallo sviluppatore.

Tipo di requisito

- **Requisiti funzionali**: definiscono le funzionalità e i servizi offerti dal sistema. Devono essere:
  - completi (includere tutti i servizi richiesti);
  - coerenti (non contenere contraddizioni);
- **Requisiti non funzionali**: riguardano le modalità operative e il ciclo di vita del prodotto. Imposti dall'organizzazione o da fattori esterni, sono classificati da Sommerville in:
  - requisiti di prodotto;
  - requisiti organizzativi;
  - requisiti esterni;
- **Requisiti di dominio**: dipendono dal contesto specifico del sistema. Esempi includono la riservatezza dei dati, le leggi fisiche o normative settoriali (es. sicurezza sul lavoro). Un esempio tipico è la richiesta di login per accedere ad aree protette.

Una seconda classificazione è il modello **FURPS**, che distingue:

- **Functionality**: caratteristiche, funzioni, sicurezza del sistema;
- **Usability**: facilità d'apprendimento e d'uso, qualità della documentazione e dell'help;
- **Reliability**: accuratezza, frequenza delle failure, recuperabilità, predicibilità del comportamento;
- **Performance**: tempi di risposta, risorse impiegate, efficienza complessiva;
- **Supportability**: manutenibilità, estendibilità, compatibilità, adattabilità.

A questi si aggiungono i cosiddetti **vincoli** o **pseudorequisiti**, che includono:

- vincoli di **implementazione** (strumenti, linguaggi, piattaforme);
- vincoli di **interfaccia** (interazioni con altri sistemi);
- vincoli di **operazione** (modalità di gestione e amministrazione);
- vincoli di **packaging** (modalità di distribuzione);
- vincoli **legali** (licenze, regolamenti, certificazioni);

Verifica e Validazione dei Requisiti

I requisiti funzionali sono verificabili tramite collaudi che coinvolgono gli utenti finali. La validazione, che deve avvenire durante tutto il ciclo di sviluppo, implica il controllo dei seguenti criteri:

- **Correttezza**: la specifica riflette esattamente ciò che è richiesto;
- **Completezza**: la specifica copre tutti gli scenari previsti;
- **Coerenza**: assenza di contraddizioni tra i requisiti;
- **Chiarezza**: i requisiti devono essere espressi in modo non ambiguo;
- **Realismo**: i requisiti devono essere realizzabili nei limiti imposti;
- **Verificabilità**: deve essere possibile testare ogni requisito;
- **Tracciabilità**: ogni funzione implementata deve essere collegabile a un requisito preciso.

Anche i requisiti di dominio sono verificabili collaudando le interazioni del sistema con software esistenti, sistemi di terze parti e rispetto delle normative. I requisiti non funzionali, invece, sono spesso vaghi e difficili da misurare in modo binario. In questi casi si adottano metriche quantitative per valutare il grado di soddisfazione.

Riassunto: Specifica dei Requisiti Software (SRS)

La Specifica dei Requisiti Software (SRS) rappresenta il documento conclusivo della fase di analisi di un progetto software. Il suo obiettivo è raccogliere in maniera strutturata tutte le esigenze funzionali e non funzionali espresse dal committente. Questo documento è un punto di riferimento per garantire una comunicazione chiara e coerente tra analisti, sviluppatori, utenti e cliente. Assicura che tutte le parti coinvolte abbiano una visione condivisa del sistema da realizzare e costituisce la base per pianificare e controllare lo sviluppo.

Importanza dell'SRS

Redigere un SRS accurato è essenziale per evitare errori che, se scoperti nelle fasi avanzate di sviluppo, possono comportare ritardi significativi e costi elevati. Un requisito ignorato o mal formulato all'inizio può compromettere l'intero progetto. Al contrario, un SRS ben fatto consente di stimare correttamente tempi, risorse e budget, oltre a fungere da base per la validazione e il collaudo del sistema.

Contenuto dell'SRS

Il documento deve includere le seguenti informazioni:

- analisi dei bisogni e dei requisiti degli utenti;
- descrizione del contesto operativo;
- elenco delle funzionalità richieste;
- definizione dei vincoli tecnici e operativi, come requisiti di performance, sicurezza, affidabilità e compatibilità;
- rappresentazione tramite casi d'uso e modelli di sistema.

Secondo lo standard IEEE 830-1998, la struttura dell'SRS si articola in:

1. Introduzione;
2. Descrizione generale del sistema;
3. Requisiti specifici (funzionali e non);
4. Modelli e architettura;
5. Appendici tecniche.

Sommerville propone una classificazione simile, ponendo particolare attenzione alla chiarezza e alla logica interna dei contenuti.

Ruoli e responsabilità

Il documento distingue chiaramente i diversi attori coinvolti nel progetto:

- **Contractor**: l'organizzazione che fornisce il sistema;
- **Customer**: il soggetto che richiede il sistema;
- **Supplier**: chi sviluppa concretamente il sistema;
- **User**: l'utente finale che interagirà con il software.

Validazione dei requisiti

Affinché sia efficace, un SRS deve essere:

- completo e privo di ambiguità;
- verificabile, cioè suscettibile di essere testato;
- coerente con i vincoli e le specifiche tecniche;
- costantemente validato insieme a clienti e utenti;

- facilmente modificabile e aggiornabile;
- tracciabile, con requisiti numerati e collegati ad attori, scenari o componenti.

Ambiguità e dettagli

Un linguaggio preciso riduce le interpretazioni errate. Requisiti ben scritti contengono formule misurabili, come ad esempio:

- "Il sistema deve rispondere entro 10 secondi";
- "Il tasso di errore non deve superare 1 su 300 richieste";
- "Il programma deve avviarsi entro 5 secondi".

Questo tipo di formulazioni consente di verificare concretamente se il sistema rispetta o meno i requisiti specificati.

Convalida delle specifiche

I requisiti possono cambiare nel tempo, anche a causa di nuove esigenze o modifiche organizzative. Per questo motivo, la convalida continua del documento è fondamentale. Gli errori più comuni includono requisiti vaghi o omessi, informazioni incomplete, conflitti interni tra requisiti e richieste irrealistiche. Talvolta, accordi verbali non documentati possono generare ambiguità.

Le checklist risultano molto utili per controllare che ogni aspetto critico sia stato coperto. Ad esempio, aiutano a verificare se sono stati specificati i limiti temporali, le eccezioni, i vincoli hardware o software, e la presenza di modifiche future prevedibili.

Revisioni e ispezioni

Per migliorare la qualità del documento è essenziale sottoporlo a revisioni frequenti e sistematiche. Questo processo coinvolge tutte le parti interessate: autore del documento, cliente, progettista, esperti di qualità. Le revisioni devono essere pianificate per funzione o area tematica, prevedendo una lettura preliminare individuale e una discussione collettiva per identificare eventuali errori o ambiguità.

Tecniche di Reading

Il metodo del "reading" è molto efficace per rilevare errori latenti. Si concentra sull'eliminazione di ciò che non deve essere presente. Le buone pratiche prevedono:

- evitare termini generici e poco precisi;
- mantenere uno stile narrativo coerente e uniforme;
- formulare requisiti oggettivi e verificabili;
- usare esempi numerici ove possibile;
- assicurare coerenza nella struttura e nei contenuti.

Introduzione

I principi SOLID sono una raccolta di linee guida fondamentali per la progettazione orientata agli oggetti. L'acronimo deriva dalle iniziali dei cinque principi formulati da Robert C. Martin. Questi principi aiutano a sviluppare software comprensibile, flessibile e manutenibile.

S - Single Responsibility Principle (principio di responsabilità singola)

Una classe deve avere una sola responsabilità, ovvero un solo motivo per essere modificata. In questo modo il codice diventa più leggibile e semplice da aggiornare o testare.

O - Open/Closed Principle (principio aperto/chiuso)

Un modulo, una classe, una funzione deve essere aperto all'estensione ma chiuso alla modifica. Questo si ottiene introducendo un'astrazione. Il concetto è rendere il codice estendibile attraverso nuove classi o metodi senza alterare il codice già esistente.

L - Liskov Substitution Principle (principio di sostituzione di Liskov)

Questo principio afferma che un oggetto di una classe derivata deve poter sostituire un oggetto della classe base senza alterare il funzionamento del programma. Si applica al polimorfismo e alla progettazione con classi e interfacce.

I - Interface Segregation Principle (principio di segregazione delle interfacce)

Questo principio afferma che è meglio creare interfacce più piccole e specifiche piuttosto che una sola interfaccia grande. I client non devono essere costretti a implementare metodi che non usano.

D - Dependency Inversion Principle (principio di inversione delle dipendenze)

Le classi ad alto livello non devono dipendere da classi a basso livello, ma entrambe devono dipendere da astrazioni. Le astrazioni non devono dipendere dai dettagli, ma i dettagli devono dipendere dalle astrazioni. Questo consente di separare le responsabilità e di cambiare le classi concrete senza modificare le classi che le usano.

Conclusione

I principi SOLID offrono un modello efficace per progettare codice di qualità, ridurre la complessità e migliorare la manutenibilità. La loro applicazione porta vantaggi significativi nello sviluppo software, soprattutto su larga scala.