Relazione finale CodeReviewBot

Progettino IS 09.03.26 - Laura Polastri

1 Introduzione

Il progetto ***CodeReviewBot*** è uno strumento di analisi automatica del codice progettato per supportare il processo di revisione attraverso l'individuazione di errori comuni, cattive pratiche e problemi di qualità del software.
Il sistema analizza i file sorgente di un progetto e genera un report dettagliato contenente le problematiche rilevate, classificandole per categoria e livello di severità, e producendo un punteggio complessivo di qualità del codice.

L'***obiettivo principale*** del sistema è supportare sviluppatori e team di sviluppo nell'identificazione di problemi nel codice, migliorando la leggibilità, la manutenibilità e la qualità complessiva del software.

Dal punto di vista funzionale, CodeReviewBot consente di:

- ***analizzare*** automaticamente ***i file sorgente*** di un progetto software
- ***applicare*** un insieme di ***regole*** di analisi configurabili
- ***individuare*** potenziali ***problemi di qualità*** del codice
- ***generare report*** dettagliati con classificazione delle issue rilevate
- ***produrre un punteggio sintetico*** di qualità del codice.

2 Architettura del sistema

Il sistema è organizzato secondo una ***layered architecture***, che suddivide l'applicazione in livelli con responsabilità ben definite. I moduli più esterni gestiscono l'interazione con l'utente e le componenti tecniche, mentre il dominio contiene la logica principale.

Questa struttura consente di separare la logica di dominio dai dettagli tecnici dell'infrastruttura e dall'interfaccia utente, migliorando la modularità e la manutenibilità del sistema.

L’architettura del progetto è organizzata nei seguenti package principali:

CLI : interfaccia utente per eseguire l’analisi
↓
Application : coordina i servizi di analisi
↓
Domain : contiene entità e regole di analisi
↓
Infrastructure : gestisce parser, file system, logging e report

Ogni livello è progettato per interagire con gli altri livelli attraverso interfacce ben definite.

2.1 CLI Layer

Il livello ***CLI*** rappresenta il punto di ingresso dell’applicazione e gestisce l’interazione con l’utente tramite riga di comando.

In questo livello viene eseguito il parsing degli argomenti forniti dall’utente e viene avviato il processo di analisi del progetto. La classe principale (Main) svolge inoltre il ruolo di composition root, occupandosi dell’istanziazione e della configurazione dei principali componenti del sistema, come parser, loader, regole di analisi ed exporter di report.

2.2 Application Layer

Il livello ***application*** contiene i servizi applicativi che coordinano i principali casi d’uso del sistema.

In particolare:

- ***AnalisiService*** gestisce il flusso completo dell’analisi del progetto, orchestrando il caricamento dei file, il parsing del codice e l’applicazione delle regole di analisi.

- ***ReportService*** gestisce la generazione e l’esportazione dei report prodotti dal sistema.

Questo livello agisce come intermediario tra il dominio e i componenti infrastrutturali, coordinando le operazioni senza contenere logica di dominio complessa.

2.3 Domain Layer

Il livello ***domain*** rappresenta il nucleo logico del sistema e contiene le entità principali e le regole di analisi del codice.

Tra le principali componenti del dominio si trovano:

- ***Progetto***, che rappresenta il progetto software analizzato
- ***FileSorgente***, che modella un file sorgente
- ***Analisi***, che rappresenta l'esecuzione di un processo di analisi
- ***Issue***, che rappresenta un problema rilevato nel codice
- ***Report***, che rappresenta il risultato finale dell'analisi
- ***RegolaAnalisi***, che definisce il comportamento delle regole di analisi applicate al codice.

Questo livello contiene la logica principale del sistema ed è progettato per essere indipendente dai dettagli tecnici dell'infrastruttura.

2.4 Infrastructure Layer

Il livello ***infrastructure*** contiene le componenti tecniche che interagiscono con risorse esterne o che implementano funzionalità specifiche necessarie al funzionamento del sistema.

Tra queste componenti troviamo:

- ***ProjectLoader***, responsabile del caricamento dei file del progetto dal filesystem
- ***Parser***, responsabile della costruzione della rappresentazione AST dei file sorgente
- ***Logger***, utilizzato per la gestione dei messaggi di log del sistema
- ***ReportExporter***, responsabile della generazione dei report in diversi formati (HTML, JSON, PDF).

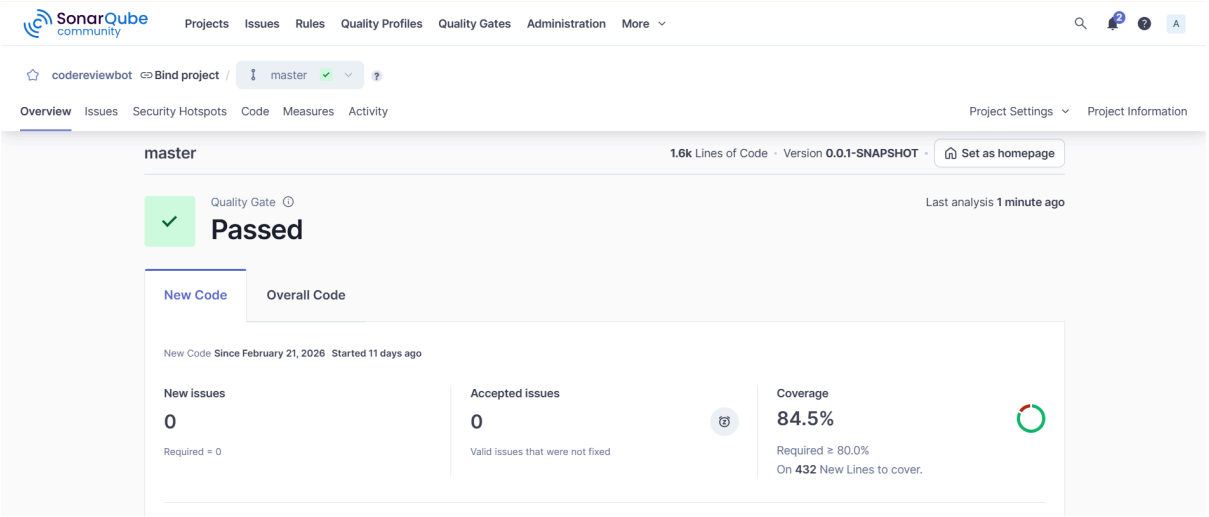
Questo livello fornisce servizi tecnici utilizzati dagli altri livelli dell'applicazione, mantenendo separata la logica di dominio dai dettagli implementativi.

3. Analisi codice SonarQube

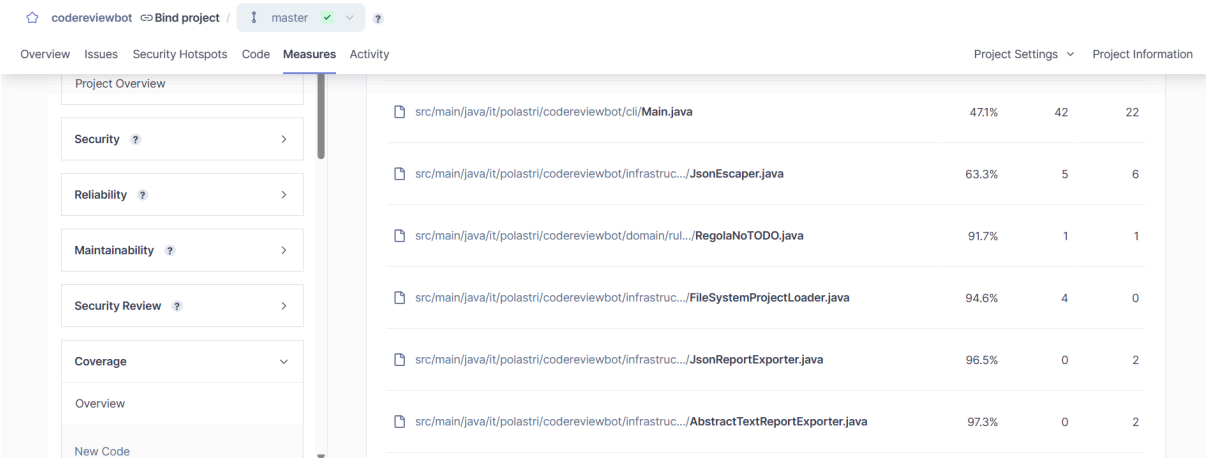
Per valutare la qualità del codice è stata utilizzata la piattaforma di analisi statica ***SonarQube***, che permette di individuare potenziali problemi di qualità, vulnerabilità di sicurezza, duplicazioni di codice e complessità del software.

L’analisi del progetto è stata effettuata nel periodo compreso tra il 22 febbraio e il 4 marzo, monitorando progressivamente il miglioramento della qualità del codice.

3.1 Overview e issues

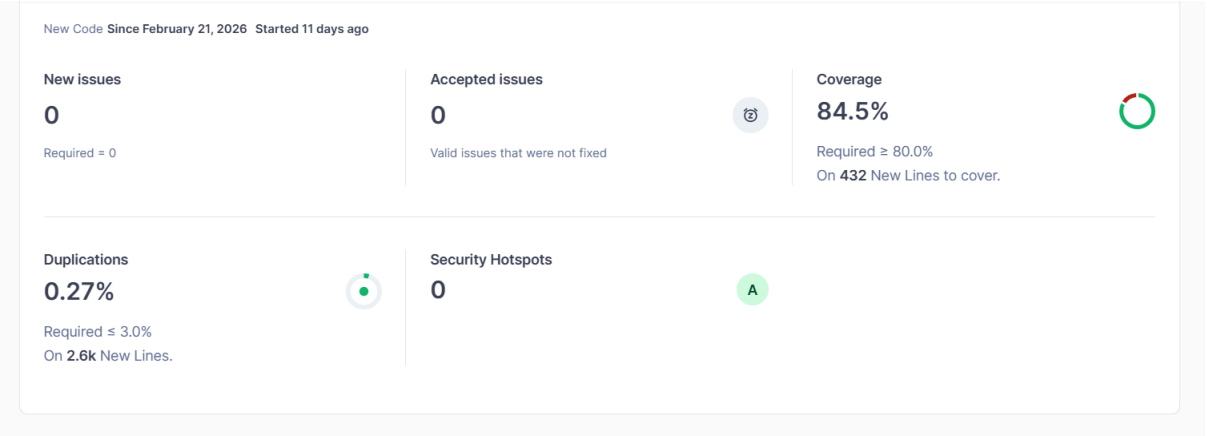


3.2 Code coverage



L’analisi mostra una ***code coverage*** dell’84.5%, valore che indica una buona copertura del codice da parte dei test automatici.

Una copertura elevata consente di verificare il corretto funzionamento delle principali componenti del sistema, ridurre il rischio di regressioni durante modifiche future e aumentare l’affidabilità del software.

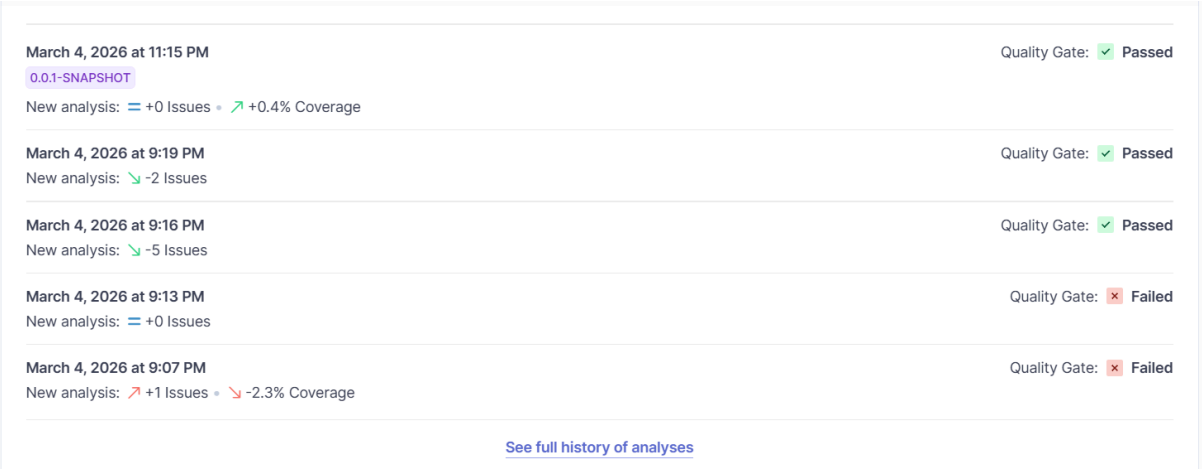


Il progetto presenta un livello molto basso di ***duplicazione del codice*** (0.27%), segno che le funzionalità comuni sono state correttamente riutilizzate attraverso classi di supporto e astrazioni condivise.

Ad esempio, nel sistema di esportazione dei report è stata introdotta una classe astratta (AbstractTextReportExporter) che centralizza il comportamento comune degli exporter, evitando duplicazioni tra le diverse implementazioni.

SonarQube ha inoltre evidenziato un buon livello di maintainability complessiva del codice.

3.3 Refactoring nel tempo



Durante le prime analisi del codice erano presenti 32 issue, principalmente relative a TODOs e presenza di metodi troppo corposi. Nel corso dello sviluppo sono stati effettuati diversi interventi di ***refactoring*** e miglioramento del codice che hanno permesso di ridurre progressivamente tali segnalazioni fino ad eliminarle completamente.

Questo ha permesso al progetto di soddisfare completamente il Quality Gate definito dallo strumento.

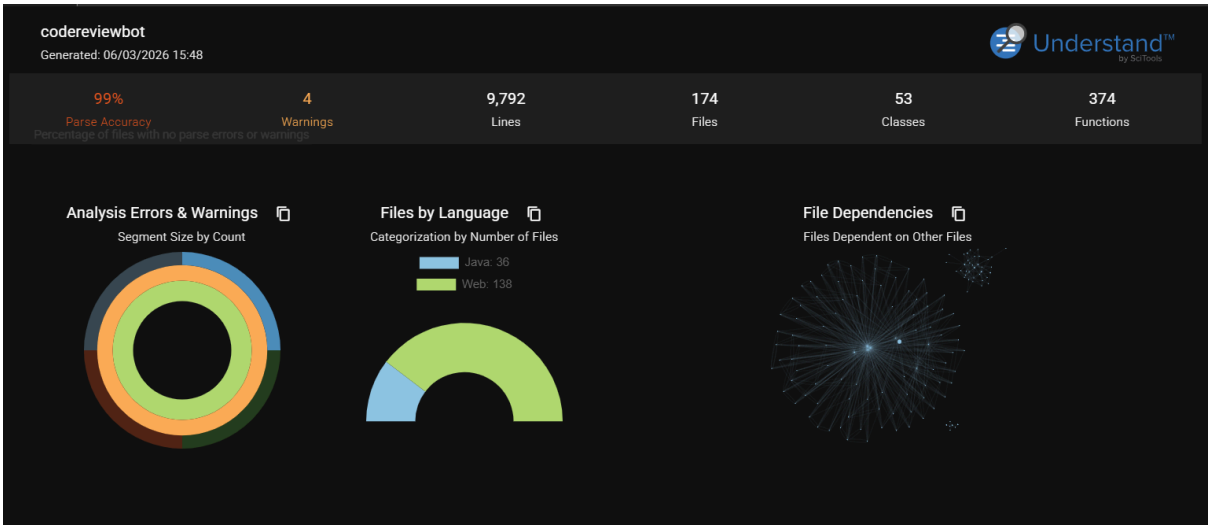
Dopo aver analizzato la qualità del codice tramite SonarQube, è stata effettuata un'analisi della struttura architetturale del progetto utilizzando ***Understand***, che consente di visualizzare dipendenze, complessità e organizzazione dei componenti software.

4. Analisi Understand

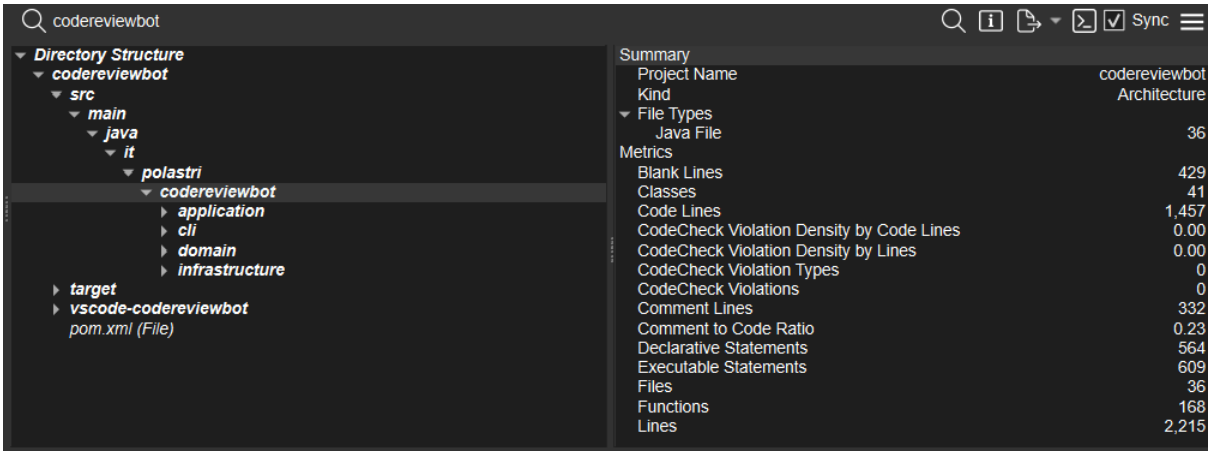
4.1 Overview del progetto

La ***dashboard*** generata dallo strumento di analisi mostra una panoramica generale del progetto. Il sistema contiene complessivamente 53 classi e 374 funzioni, distribuite su circa 9792 linee di codice. L'analisi ha evidenziato un'accuratezza del parsing pari al 99%, con un numero limitato di warning.

Sono state ignorate le classi di testing e le librerie esterne.



4.2 Metriche di complessità



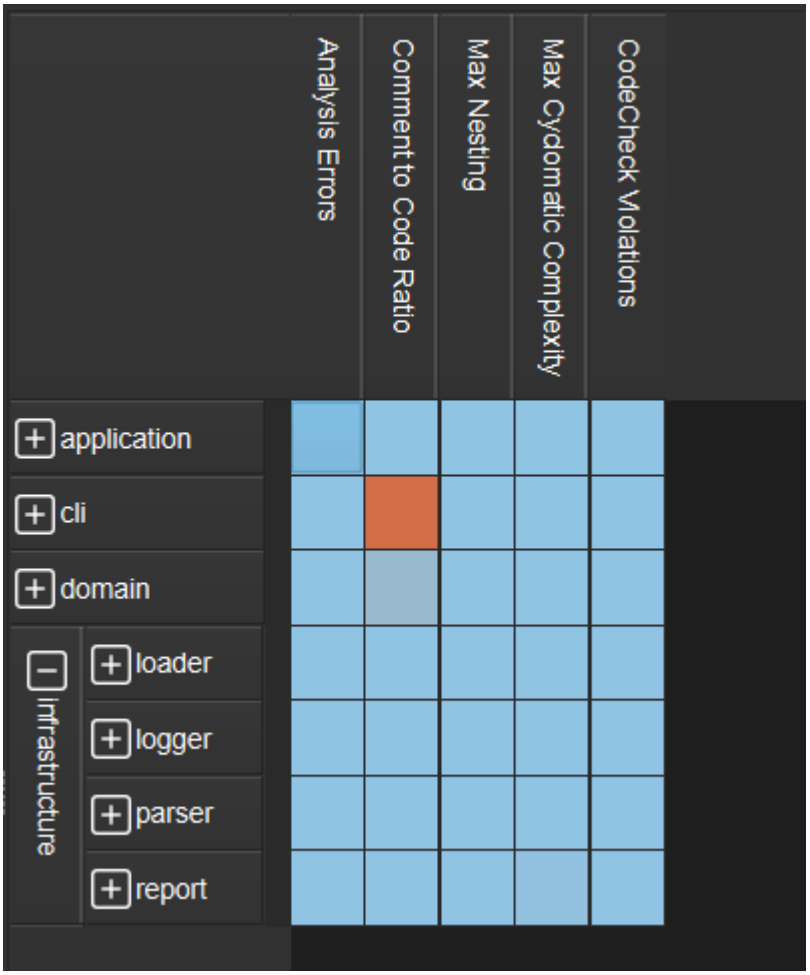
L'analisi delle ***metriche di complessità*** evidenzia alcune classi con valori più elevati di complessità ciclomatica e nesting, in particolare quelle appartenenti al package domain. Tuttavia, nel complesso la complessità del codice risulta distribuita e non emergono classi eccessivamente critiche.

La differenza tra il numero complessivo di linee di codice riportato dalla dashboard e le linee di codice Java effettivamente analizzate è dovuta alla presenza di altri componenti del progetto, come file relativi al frontend, che vengono conteggiati nelle metriche generali ma non analizzati nel dettaglio come codice Java.

L'***analisi delle dimensioni*** delle classi mostra inoltre che alcune componenti, come quelle relative alla gestione dell'analisi del codice e alla generazione dei report, presentano un numero maggiore di linee di codice. Tuttavia, la distribuzione delle dimensioni appare relativamente equilibrata.

4.3 Metrics heatmap

Le ***metrics heatmap*** generate con Understand permettono di visualizzare in modo sintetico la distribuzione di alcune metriche del codice tra i diversi package.

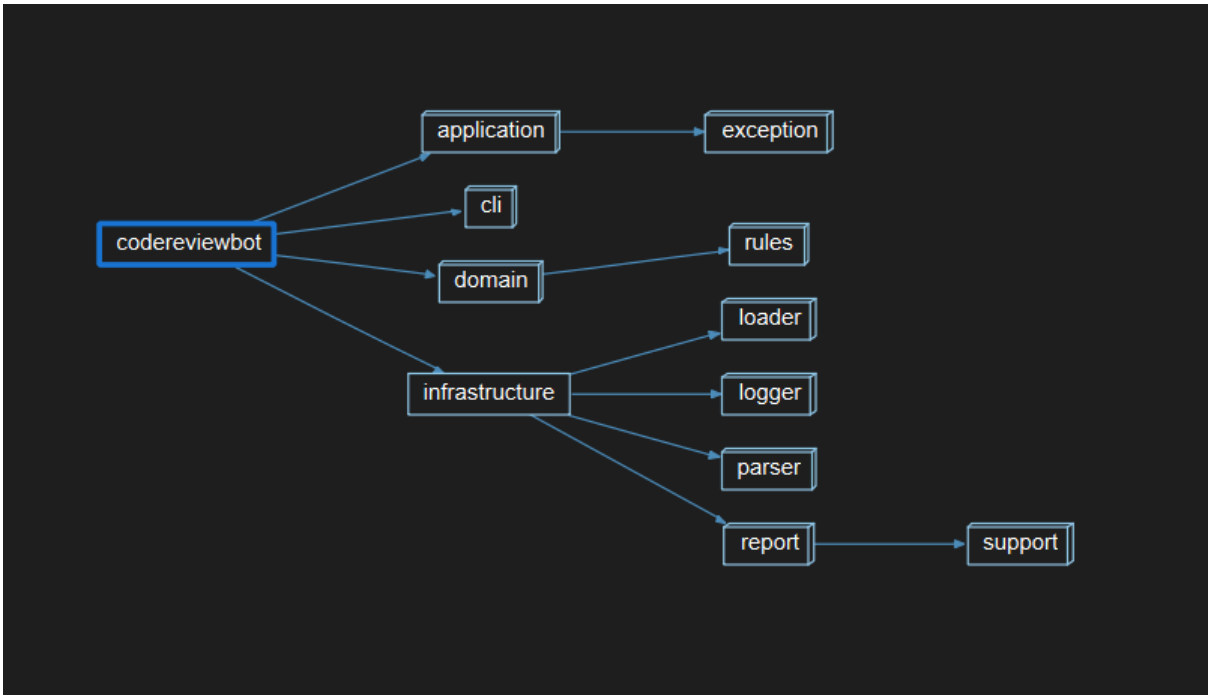


Dall'analisi emerge che ciascun package presenta valori relativamente bassi per le metriche di complessità, suggerendo che la struttura del codice risulta semplice e ben distribuita tra i diversi moduli.

Il package cli mostra un valore relativamente più alto nel comment to code ratio, indicando una maggiore presenza di commenti nel codice, probabilmente dovuta al ruolo di punto di ingresso dell'applicazione e alla necessità di documentare il flusso di esecuzione principale.

Nel complesso, il sistema mantiene una buona leggibilità e un livello di complessità controllato.

4.4 Dipendenze tra package, classi e architettura del sistema

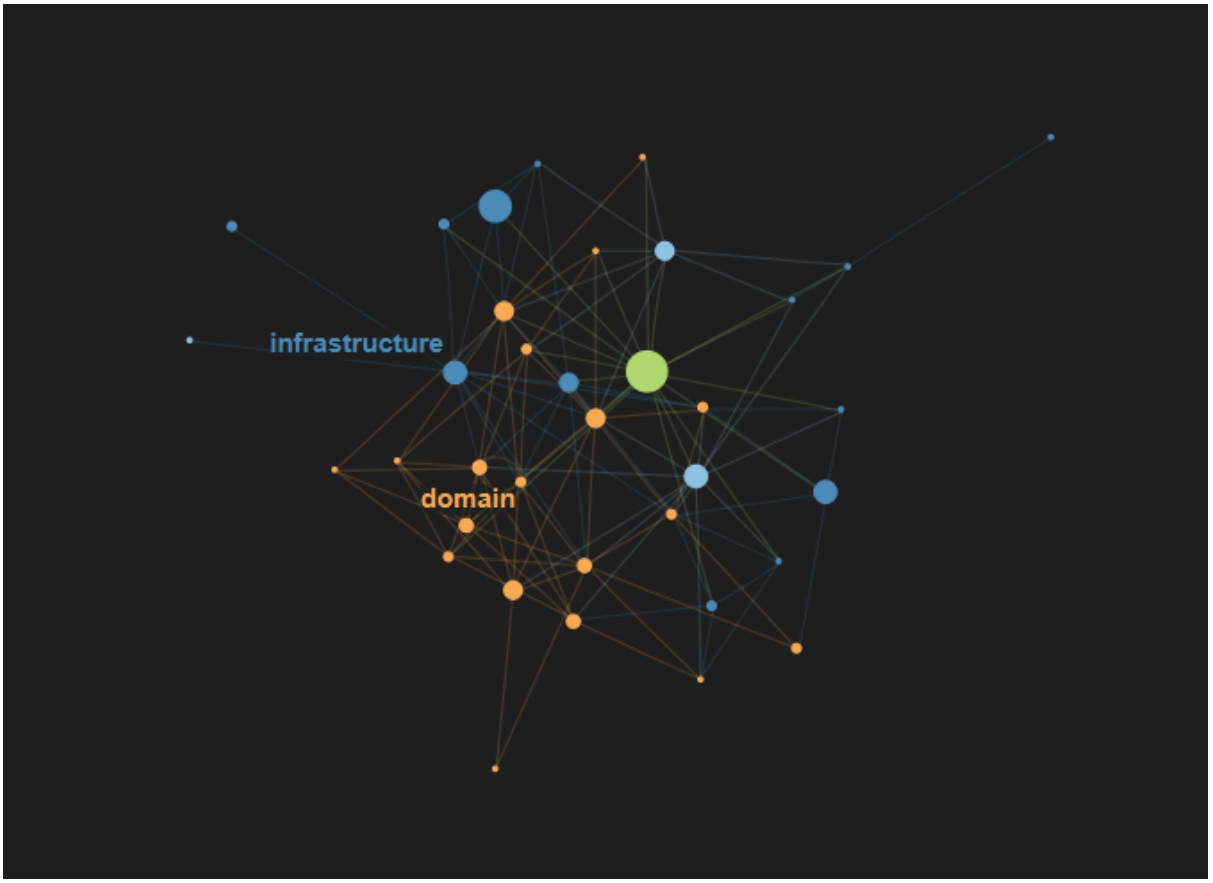


L'analisi delle ***dipendenze tra package*** evidenzia la struttura modulare organizzata in layer: CLI, application, domain e infrastructure.

Il grafo delle ***dipendenze tra classi*** mostra che le classi del domain sono le più centrali nel sistema, indicando che la logica principale è concentrata in questo modulo. I package

application e infrastructure dipendono dal domain per implementare rispettivamente la logica applicativa e i servizi tecnici.

Questa struttura suggerisce una separazione tra logica applicativa e componenti infrastrutturali.



Il grafo mostra ancora una volta una forte centralità del package domain. Sebbene ciò sia coerente con il ruolo del dominio nel sistema, una concentrazione eccessiva di dipendenze potrebbe rendere alcune classi particolarmente critiche dal punto di vista della manutenzione.

Il domain risulta infatti avere molte connessioni esterne (linee più concentrate), mentre infrastructure presenta meno dipendenze.

5 Code smells e possibili antipattern strutturali

L'analisi effettuata tramite Understand consente di individuare eventuali criticità strutturali del sistema. Sono stati rilevati alcuni possibili ***code smells e antipattern architetturali***,

evidenziando le principali aree del progetto che potrebbero rappresentare punti a cui fare attenzione dal punto di vista della manutenzione del software.

5.1 Forte centralità del package domain

Come evidenziato anche dai grafici delle dipendenze analizzati nella sezione precedente, il package *domain* rappresenta il nucleo centrale del sistema. Questa scelta è coerente con l'architettura a livelli adottata, in cui il dominio contiene le principali entità e regole di analisi del codice.

Tuttavia, una forte concentrazione di dipendenze su un singolo modulo può essere associata al concetto di ***Hub Risk***, un antipattern architetturale che si verifica quando un componente diventa il punto centrale di molte dipendenze nel sistema. In tali situazioni eventuali modifiche a questo modulo potrebbero propagarsi ad altri componenti dell'applicazione.

Nel caso specifico del progetto, questa centralità non rappresenta un problema critico, ma riflette il ruolo naturale del dominio all'interno dell'architettura del sistema.

5.2 Coesione non elevata del package infrastructure

L'analisi dei package evidenzia che il modulo infrastructure raccoglie componenti tecnici eterogenei, tra cui parsing, logging, caricamento dei file e generazione dei report. Questa scelta è coerente con il ruolo del livello infrastrutturale, che ha il compito di fornire servizi tecnici utilizzati dagli altri livelli dell'applicazione.

Tuttavia, la presenza di responsabilità piuttosto diverse all'interno dello stesso modulo può suggerire una coesione interna non elevata. Questo fenomeno può essere associato al concetto di ***Low Cohesion***, in cui un modulo accorpa funzionalità non strettamente correlate, rendendo potenzialmente più complessa l'evoluzione e la manutenzione del sistema nel caso in cui il numero di componenti infrastrutturali aumenti.

5.3 Accoppiamento marcato tra application e infrastructure

Il grafo delle dipendenze mostra una dipendenza diretta del package application da infrastructure. Questa scelta può essere accettabile in un progetto di dimensioni moderate, ma riduce parzialmente l'isolamento della logica applicativa rispetto ai dettagli tecnici.

Tale situazione può essere ricondotta al problema del ***Tight Coupling***, ovvero un forte accoppiamento tra moduli che limita l'indipendenza dei componenti e può aumentare l'impatto delle modifiche nel sistema.

In sistemi più complessi questo tipo di dipendenza può contribuire alla formazione di strutture architetturali fragili, in cui una modifica in un modulo tende a propagarsi facilmente agli altri componenti.

5.4 Complessità concentrata in alcune classi del dominio

Le metriche di complessità mostrano che alcune classi del package domain presentano valori più elevati rispetto alla media. Ciò suggerisce la presenza di componenti maggiormente critiche dal punto di vista della manutenzione e del refactoring.

Questo fenomeno può essere collegato al rischio di ***High Complexity*** o alla possibile evoluzione verso una ***God Class***, ovvero una classe che tende ad accumulare troppe responsabilità e dipendenze.

6 Pattern architetturali e di progettazione

Dall'analisi della struttura del progetto e delle classi implementate è possibile individuare diversi pattern architetturali e di progettazione. L'utilizzo di tali pattern contribuisce a migliorare la modularità del sistema, favorire il riutilizzo del codice e ridurre il grado di accoppiamento tra i componenti.

6.1 Layered Architecture

Il progetto adotta una ***Layered Architecture***, organizzando il sistema in quattro package.

Ogni livello ha responsabilità ben definite:

- ***CLI layer***
  gestisce l'interazione con l'utente tramite l'interfaccia a riga di comando.

- ***Application layer***
  coordina i casi d'uso principali del sistema.

- ***Domain layer***
  contiene le entità principali e la logica di analisi del codice.

- ***Infrastructure layer***
  implementa componenti tecnici come parser, logging, caricamento dei file e generazione dei report.

Questa organizzazione favorisce una chiara separazione delle responsabilità e riduce la dipendenza diretta tra componenti di livelli diversi.

6.2 Application Service Pattern

Il ***Service Pattern*** prevede l'introduzione di componenti dedicati alla gestione dei casi d'uso dell'applicazione, separando la logica di coordinamento dalle entità di dominio. I service orchestrano le operazioni tra diversi componenti del sistema senza contenere direttamente lo stato principale del dominio.

Nel livello applicativo sono presenti classi che implementano chiaramente questo pattern, incaricate di coordinare le operazioni principali del sistema.

In particolare:

- ***AnalisiService*** gestisce il flusso completo dell'analisi del progetto, coordinando il caricamento dei file, il parsing del codice e l'applicazione delle regole di analisi.

- ***ReportService*** gestisce la generazione e l'esportazione dei report di qualità.

Queste classi non rappresentano entità di dominio ma fungono da orchestratori dei casi d'uso principali del sistema.

6.3 Strategy Pattern

Lo ***Strategy Pattern*** consente di definire una famiglia di algoritmi incapsulati in classi separate e intercambiabili. Ogni strategia implementa un comportamento specifico, permettendo al sistema di selezionare dinamicamente l'algoritmo da utilizzare senza modificare il codice client.

Questo pattern viene utilizzato in due parti del sistema:

Regole di analisi del codice

Le regole di analisi sono modellate tramite l'astrazione ***RegolaAnalisi***, mentre le classi concrete implementano differenti strategie di controllo del codice, in questo caso: ***RegolaNoTODO*** e ***RegolaNoSystemOutPrintln.***

Questo consente di aggiungere facilmente nuove regole senza modificare il codice esistente.

Esportazione dei report

Anche il sistema di esportazione dei report utilizza una struttura strategica: ***ReportExporter*** definisce l'interfaccia comune, implementata dalle classi concrete ***HtmlReportExporter, JsonReportExporter*** e ***PdfReportExporter***.

Ogni classe implementa una strategia diversa per la generazione del report, consentendo di aggiungere nuovi format senza la necessità di modificare il codice esistente.

6.4 Template Method

Nel sottosistema di generazione dei report è presente una classe astratta di supporto ***AbstractTextReportExporter***.

Questa classe fornisce il comportamento comune necessario alla generazione dei report, tra cui:

- validazione dei parametri
- gestione della scrittura su file
- rendering del contenuto testuale del report
- gestione degli errori di I/O

Le classi concrete (***HtmlReportExporter, JsonReportExporter e PdfReportExporter***) specializzano il comportamento implementando la specifica modalità di esportazione.

Questa struttura corrisponde al ***Template Method Pattern***, in cui la classe base definisce l'algoritmo generale mentre le sottoclassi ne specializzano alcune parti.

6.5 Factory Method

Il ***Factory Method*** è un pattern che delega la creazione degli oggetti a un metodo dedicato invece di istanziarli direttamente nel codice client. Questo permette di centralizzare la logica di creazione e di nascondere i dettagli costruttivi degli oggetti.

Nel dominio sono infatti presenti metodi statici che centralizzano la creazione di oggetti complessi, ovvero

Report.creaDa(...) e RisultatoAnalisi.creaDa(...).

Questi metodi costruiscono oggetti derivati a partire dai risultati dell'analisi, incapsulando la logica di creazione e mantenendo le classi client indipendenti dai dettagli costruttivi.

Questo approccio è assimilabile all'uso di ***Factory Method statici***.

6.6 Dependency Injection

La ***Dependency Injection*** è una tecnica di progettazione in cui le dipendenze di una classe vengono fornite dall'esterno, invece di essere create direttamente al suo interno. In questo modo le classi risultano meno accoppiate tra loro e il sistema diventa più facile da modificare e testare.

Il progetto utilizza una forma di ***Dependency Injection costruttoriale***.

Ad esempio:

- ***AnalisiService*** riceve nel costruttore:
    - ProjectLoader
    - Parser
    - lista di RegolaAnalisi
    - Logger
- ***ReportService*** riceve:
    - QualityScoreService
    - i binding degli exporter
    - Logger

Questa scelta consente di ridurre l'accoppiamento tra componenti e di migliorare la modularità e la testabilità del sistema.

6.7 Domain Service

Un ***Domain Service*** rappresenta un servizio che incapsula logica di dominio quando questa non può essere attribuita in modo naturale a una singola entità. In questo modo si mantiene il modello di dominio più chiaro e le responsabilità delle classi rimangono ben separate.

La classe ***QualityScoreService*** rappresenta un esempio di Domain Service, poiché incapsula una logica di calcolo che non appartiene naturalmente a una singola entità del dominio.

Il servizio calcola il punteggio di qualità complessivo dell'analisi sulla base delle issue rilevate.

L'uso combinato di pattern architetturali e di progettazione contribuisce a rendere il sistema modulare, estendibile e manutenibile.

7 Design Principles

Durante la progettazione del sistema sono stati applicati diversi design principles orientati a migliorare modularità, estendibilità e manutenibilità del software. In particolare, il progetto segue diversi principi della famiglia ***SOLID***, introdotti da Robert C. ***Martin***.

7.1 Single Responsibility Principle (SRP)

Il ***Single Responsibility Principle*** stabilisce che una classe dovrebbe avere una sola responsabilità e un solo motivo di cambiamento.

Nel progetto questo principio è applicato separando chiaramente le diverse responsabilità tra le classi.

Ad esempio:

- ***AnalisiService*** si occupa esclusivamente della gestione del processo di analisi del progetto.
- ***ReportService*** gestisce la generazione e l'esportazione dei report.
- ***FileSystemProjectLoader*** è responsabile del caricamento dei file sorgente dal filesystem.
- ***ConsoleLogger*** gestisce esclusivamente la registrazione dei messaggi di log.

Questa separazione consente di modificare una funzionalità senza influenzare direttamente le altre componenti del sistema.

7.2 Open/Closed Principle (OCP)

L'***Open/Closed Principle*** stabilisce che i componenti software dovrebbero essere aperti all'estensione ma chiusi alla modifica. Questo significa che è possibile aggiungere nuovi comportamenti estendendo il sistema (nuove classi o implementazioni) senza dover modificare il codice già esistente.

Nel progetto questo principio è applicato in diversi punti.

Un esempio significativo riguarda il sistema di esportazione dei report:

- l'interfaccia ***ReportExporter*** definisce il comportamento comune
- le classi concrete (***HtmlReportExporter, JsonReportExporter, PdfReportExporte***r) implementano modalità diverse di generazione del report.

Questo permette di aggiungere nuovi formati di esportazione senza modificare il codice esistente.

Lo stesso principio è applicato anche alle regole di analisi, dove nuove regole possono essere introdotte implementando l'astrazione ***RegolaAnalisi***.

7.3 Dependency Inversion Principle (DIP)

Il ***Dependency Inversion Principle*** stabilisce che i moduli di alto livello non dovrebbero dipendere direttamente dai moduli di basso livello, ma entrambi dovrebbero dipendere da astrazioni.

Nel progetto questo principio è applicato tramite l'uso di interfacce per diversi componenti del sistema.

Ad esempio:

- ***Parser*** definisce un'astrazione per il parsing dei file sorgente
- ***ProjectLoader*** definisce l'interfaccia per il caricamento dei progetti

- ***Logger*** definisce l'astrazione per il sistema di logging
- ***ReportExporter*** definisce il comportamento comune degli exporter.

Le implementazioni concrete (***StubParser, FileSystemProjectLoader, ConsoleLogger, Html/Json/PdfReportExporter***) possono quindi essere sostituite senza modificare il codice dei livelli superiori.

7.4 Interface Segregation Principle (ISP)

L'***Interface Segregation Principle*** stabilisce che le interfacce dovrebbero essere specifiche e focalizzate su un insieme limitato di operazioni.

Nel progetto questo principio è rispettato tramite l'uso di interfacce semplici e mirate.

Ad esempio:

- ***Parser*** espone solo il metodo necessario per il parsing dei file sorgente.
- ***ProjectLoader*** espone esclusivamente l'operazione di caricamento di un progetto.
- ***Logger*** definisce un'operazione generale di logging e alcuni metodi di supporto per i diversi livelli di severità.

Questo evita la creazione di interfacce troppo complesse e facilita l'implementazione di nuove componenti.

7.5 Acyclic Dependency Principle (ADP)

L'***Acyclic Dependency Principle*** stabilisce che le dipendenze tra moduli o package non dovrebbero formare cicli.

L'analisi delle dipendenze effettuata tramite Understand mostra che i package del sistema seguono una struttura gerarchica coerente con l'architettura a livelli, evitando cicli di dipendenza tra i moduli principali.

7.6 Separation of Concerns

Un ulteriore principio applicato nel progetto è la ***Separation of Concerns***, che consiste nel separare le diverse responsabilità del sistema in moduli indipendenti.

Nel progetto questo principio è evidente nella suddivisione architetturale nei diversi livelli:

- ***CLI*** per l'interazione con l'utente
- ***Application*** per la gestione dei casi d'uso
- ***Domain*** per la logica di business
- ***Infrastructure*** per i servizi tecnici.

Questa organizzazione riduce l'accoppiamento tra i componenti e facilita l'evoluzione futura del sistema.

L'analisi delle metriche, dei pattern architetturali e dei design principles permette quindi di ottenere una visione complessiva della qualità strutturale del sistema.

8 Conclusioni

Il presente progetto ha avuto l'obiettivo di analizzare e valutare la qualità e la struttura architetturale di un sistema software, utilizzando strumenti di analisi statica del codice e tecniche di progettazione orientate agli oggetti.

L'***analisi della qualità*** del codice è stata effettuata tramite ***SonarQube***, che ha permesso di individuare inizialmente diverse issue legate principalmente a code smells e aspetti di manutenibilità del codice. Nel corso dello sviluppo sono stati effettuati interventi progressivi di refactoring che hanno consentito il miglioramento del codice.

Successivamente è stata effettuata un'***analisi della struttura architetturale*** del sistema tramite ***SciTools Understand***, che ha consentito di studiare le dipendenze tra i componenti e di analizzare diverse metriche strutturali del progetto. I grafici generati mostrano una chiara organizzazione modulare del sistema, con una struttura basata su livelli che separa le responsabilità tra interfaccia utente, logica applicativa, dominio e componenti infrastrutturali.

L'***analisi del codice*** ha inoltre permesso di individuare diversi ***pattern di progettazione*** utilizzati all'interno del sistema. In particolare, sono stati identificati lo ***Strategy Pattern***, utilizzato per implementare sia le regole di analisi del codice sia le diverse modalità di esportazione dei report; il ***Template Method Pattern*** nel sottosistema di generazione dei report; il ***Factory Method*** per la creazione di alcuni oggetti di dominio. A livello architetturale, il sistema segue una ***Layered Architecture***, mentre nel livello applicativo sono presenti classi che implementano il ***Service Pattern*** coordinando i principali casi d'uso del sistema.

In conclusione, il lavoro svolto ha evidenziato l***'importanza dell'integrazione tra buone pratiche di progettazione software e tecniche di analisi automatica del codice***, dimostrando come l'applicazione congiunta di tali tecniche, dei ***principi di progettazione*** e di pratiche di ***refactoring continuo*** possa contribuire in modo significativo al miglioramento della qualità del software, permettendo di individuare precocemente problemi strutturali e di mantenere il sistema più semplice da comprendere ed evolvere nel tempo.

9. Gantt e flusso di lavoro

La pianificazione iniziale del flusso di lavoro del progetto si è rivelata complessivamente realistica. In particolare, la fase di ***analisi e progettazione*** e la successiva ***implementazione del core*** dell'applicazione hanno rispettato in larga parte le tempistiche previste dal ***diagramma di Gantt***.

Lo sviluppo è stato organizzato procedendo inizialmente con l'implementazione delle classi di dominio, affrontate progressivamente in ordine di complessità. Successivamente sono stati realizzati i service applicativi, responsabili del coordinamento dei principali casi d'uso del sistema.

In una fase successiva è stato implementato il layer di infrastructure, includendo le componenti necessarie per il caricamento dei progetti, il parsing e la generazione dei report. Questa fase ha comportato anche l'adattamento e l'integrazione delle classi già esistenti con le nuove componenti introdotte. Tutte le principali parti del sistema sono state accompagnate dalla scrittura dei relativi test.

Infine sono state implementate le versioni concrete delle regole di analisi e degli exporter dei report, concludendo la parte ***backend*** con una prima generica stesura del punto di ingresso dell'applicazione, rappresentato dalla classe Main.

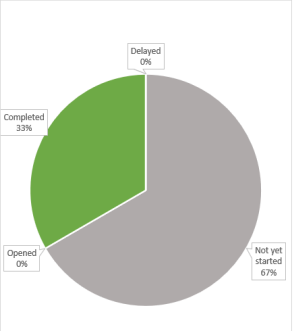
Solo in un secondo momento è stata creata la cartella *"vscode-codereviewbot"*, contenente i file necessari alla realizzazione della parte ***frontend*** dell'applicazione.

	Date	Name
Created:	17/01/2026	LauraPolastri
Changed:		

Project name: CodeReviewBot

List of open points

No.	Date opened	Topic/ Category	Actions	Due date	%	Status
1	09/02/2026	Avvio e setup progetto	Setup github repository e workflow, build e struttura progetto	09/02/2026	100%	completed
2	09/02/2026	Analisi e progettazione	Modellazione diagrammi vpp: use case, dominio, classi, SD e SSD. Contratti, CU e diregole di dominio	16/02/2026	100%	completed
3	16/02/2026	Implementazione core applicaizione	Implementazione dominio, servizi applicativi, analisi statica e gestione issues, generazione report e quality score, persistenza output su file	23/02/2026	0%	not yet started
4	24/02/2026	Qualità, test e hardening	Esecuzione test unitari e integrazione, analisi SonarQube e refactoring, test di usabilità	01/03/2026	0%	not yet started
5	02/03/2026	Interfaccia utente (CLI) e usabilità	Progettazione interazione utente, messaggi d'errore, help, validazione input, guida installazione e uso	07/03/2026	0%	not yet started
6	07/03/2026	Documentazione e consegna	Stesura relazione finale, check finale e tag di consegna	09/03/2026	0%	not yet started



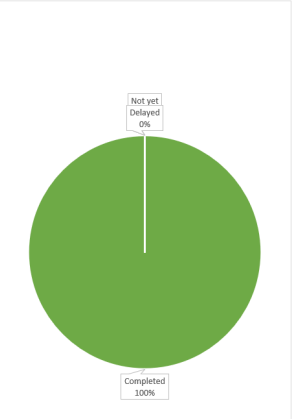
Contrariamente a quanto previsto nella pianificazione iniziale, la fase di ***test e hardening*** del codice, così come l'analisi tramite SonarQube, non è stata svolta in un'unica fase finale, ma è avvenuta progressivamente durante l'intero sviluppo del codice Java. Prima di ogni commit è stata infatti eseguita un'analisi completa, seguita da eventuali interventi di refactoring necessari a migliorare la qualità complessiva del progetto.

Successivamente è stata introdotta una fase aggiuntiva dedicata all'implementazione delle ***GitHub Actions*** per la build automatica dell'applicazione. Questa attività ha richiesto ulteriori interventi di refactoring del codice e della configurazione del progetto fino al raggiungimento di una build stabile e correttamente eseguita tramite workflow automatico.

Created:	17/01/2026	LauraPolastri
Changed:	08/03/2026	LauraPolastri
Project name:	CodeReviewBot	

List of open points

No.	Date opened	Topic/ Category	Actions	Due date	%	Status
1	09/02/2026	Avvio e setup progetto	Setup github repository e workflow, build e struttura progetto	09/02/2026	100%	completed
2	09/02/2026	Analisi e progettazione	Modellazione diagrammi vpp: use case, dominio, classi, SD e SSD. Contratti, CU e diregole di dominio	16/02/2026	100%	completed
3	16/02/2026	Implementazione core applicaizione	Implementazione dominio, servizi applicativi, analisi statica e gestione issues, generazione report e quality score, persistenza output su file	25/02/2026	100%	completed
4	16/02/2026	Qualità, test e hardening	Esecuzione test unitari e integrazione, analisi SonarQube e refactoring, test di usabilità	25/02/2026	100%	completed
5	26/02/2026	Github Actions e CLI	Aggiunta build per applicazione, stesura Main per interazione da linea di comando con il client	28/02/2026	100%	completed
6	01/03/2026	Interfaccia utente e usabilità	Progettazione interfaccia utente, estensioneVS Code, guida installazione e uso con prova su pc "pulito"	05/03/2026	100%	Completed
7	05/03/2026	Allineamento progettazione iniziale	Correzione dei file generati nella prima fase di lavoro, in modo da non generare incongruenze con la loro implementazione nel codice	08/03/2026	100%	completed
8	06/03/2026	Documentazione e consegna	Integrazione analisi Understand, stesura relazione finale, check finale e tag di consegna	09/03/2026	100%	completed



La realizzazione dell'***interfaccia utente*** si è articolata in due fasi principali. In un primo momento è stata implementata la classe ***Main***, responsabile della validazione dell'input proveniente dal client e del coordinamento delle diverse operazioni dell'applicazione. Successivamente è stato sviluppato il codice dell'estensione per ***Visual Studio Code***, comprendente la realizzazione dell'interfaccia, l'integrazione con il backend e le relative attività di test e refactoring.

È stato inoltre testato il processo completo di installazione e utilizzo dell'applicazione su un ***PC "pulito"***, partendo esclusivamente dal repository GitHub. Questo test ha permesso di individuare alcune criticità nel processo di installazione, successivamente risolte tramite correzioni e integrazioni del file ***README.md***.

Infine è stata necessaria una fase aggiuntiva di circa tre o quattro giorni dedicata all'***allineamento del file di analisi e progettazione*** .vpp con il codice effettivamente implementato. Durante lo sviluppo sono infatti emerse alcune esigenze progettuali non previste inizialmente, che hanno portato a modifiche del flusso di lavoro e dell'architettura del sistema. L'aggiornamento del modello progettuale è stato quindi necessario per evitare incongruenze evidenti tra la documentazione di progetto e l'implementazione finale.

Nel complesso, la pianificazione iniziale ha fornito una base efficace per l'organizzazione delle attività di sviluppo. Alcune fasi sono state adattate nel corso del progetto, in particolare per integrare maggiormente test, refactoring e analisi del codice durante l'implementazione. Questo approccio iterativo ha permesso di mantenere un buon livello di qualità del software e di completare il progetto in modo coerente con gli obiettivi prefissati.