Specifica Tecnica

Versione: 0.5.1 18/03/2025

Redattori

Maria Fuensanta Trigueros Hernandez
Marco Perazzolo
Ion Cainareanu
Malik Giafar Mohamed

Verifica

Ion Cainareanu
Malik Giafar Mohamed
Luca Parise
Marco Perazzolo
Stefano Baso

Approvazione

Uso Esterno

nextsoftpadova@gmail.com

Registro dei cambiamenti

Versione	Data	Autore	Descrizione	Verifica
0.5.1	12/05/2025	Marco Perazzolo, Ion Cainareanu	Aggiornamento dei diagrammi UML del frontend e correzione di errori relativi alle tecnologie utilizzate	
0.5.0	12/05/2025	Malik Giafar Mohamed	Stesura sezione architettura di deployment	Marco Perazzolo, Ion Cainareanu
0.4.1	06/05/2025	Marco Perazzolo	Migliorato lo stile del documento e aggiornati i diagrammi UML	Stefano Baso
0.4.0	02/05/2025	Malik Giafar Mohamed	Stesura sezione architettura backend di dettaglio	Marco Perazzolo
0.3.0	27/04/2025	Ion Cainareanu	Stesura sezione architettura frontend di dettaglio	Luca Parise
0.2.0	24/04/2025	Marco Perazzolo	Aggiornamento delle tecnologie utilizzate e aggiunta di diagrammi UML delle classi	Malik Giafar Mohamed
0.1.0	18/03/2025	Maria Fuensanta Trigueros Hernandez, Marco Perazzolo	Creazione del documento e stesura dei capitoli Introduzione e Tecnologie	Ion Cainareanu

Indice

Elenco delle tabelle

Elenco delle figure

1 Introduzione

1.1 Scopo del documento

L'obiettivo di questo documento è fornire una descrizione dettagliata e completa dell'architettura del prodotto sviluppato. Sono incluse le tecnologie utilizzate e i requisiti necessari per il suo funzionamento.

Il documento presenta la struttura del prodotto attraverso diagrammi delle classi. Inoltre, giustifica l'uso di alcuni pattern di progettazione impiegati nell'implementazione

1.2 Scopo del prodotto

Il prodotto, un *plug-in*[G] per *Visual Studio Code*[G] chiamato "Requirement Tracker Plug-in", è progettato per automatizzare il tracciamento dei *requisiti*[G] nei progetti software complessi, con un focus particolare sull'ambito *embedded*[G]. L'obiettivo principale è migliorare la qualità e la chiarezza dei requisiti, fornendo suggerimenti basati sull'analisi di un'*intelligenza artificiale*[G], riducendo al contempo i tempi e gli errori legati alla verifica manuale dell'implementazione nel codice sorgente. Il plug-in adotta un'architettura modulare che consente un'estensibilità semplice, rendendolo facilmente adattabile a nuove funzionalità o esigenze future. Inoltre, supporta gli sviluppatori avendo la capacità di utilizzare documenti tecnici come *knowledge*[G], ad esempio datasheet e manuali, in modo da garantire una corretta implementazione dei requisiti.

1.3 Glossario

I termini che potrebbero risultare ambigui sono contrassegnati alla loro prima apparizione con un apice [G]. La loro definizione completa è consultabile nel documento del glossario, in cui sono riportati in ordine alfabetico.

1.4 Riferimenti

1.4.1 Riferimenti normativi

- Capitolato C8 : Requirement Tracker - Plug-in VS Code
  - https://www.math.unipd.it/~tullio/IS-1/2024/Progetto/C8.pdf
- Norme di Progetto v2.0.0
- Analisi dei Requisiti v2.0.0

1.4.2 Riferimenti informativi

- Progettazione e programmazione: Diagrammi delle classi (UML)
  - https://www.math.unipd.it/~rcardin/swea/2023/Diagrammi%20delle%20Classi.pdf
- Progettazione: I pattern architetturali

  - https://www.math.unipd.it/~rcardin/swea/2022/Software%20Architecture%20Patterns.pdf
- Progettazione software (T6)
  - https://www.math.unipd.it/~tullio/IS-1/2024/Dispense/T06.pdf
- Progettazione: Il pattern Dependency Injection
  - https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Architetturali%20-%20Dependency%20Injection.pdf
- Progettazione: il pattern Model-View-Controller e derivati
  - https://www.math.unipd.it/~rcardin/sweb/2022/L02.pdf
- Progettazione: i pattern creazionali (GoF)
  - https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Creazionali.pdf
- Progettazione: I pattern strutturali (GoF)
  - https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Strutturali.pdf
- Progettazione: I pattern di comportamento (GoF)
  - https://drive.google.com/file/d/1cpi6rORMxFtC91nI6_sPrG1Xn-28z8eI/view?usp=sharing
- Programmazione: SOLID programming
  - https://drive.google.com/file/d/1o1Xun2dVVc3mDiaGyN0FrDJhhoO3lfLQ/view?usp=sharing

2 Tecnologie utilizzate

Tecnologia	Versione	Descrizione
Linguaggio		
TypeScript	5.8.2	Linguaggio di programmazione ad alto livello, *open-source*[G], che estende JavaScript aggiungendo *tipizzazione statica*[G], *interfacce*[G] e controllo statico degli errori.
HTML	5	Linguaggio di markup utilizzato per creare pagine web. Viene utilizzato per definire la struttura e il contenuto dell'interfaccia del plug-in.
CSS	3	Linguaggio di stile utilizzato per descrivere l'aspetto e la formattazione di interfacce web o applicazioni. Nel nostro caso serve a definire lo stile dell'interfaccia del plug-in.
Strumenti		
Visual Studio Code	1.95.0	Editor open source di codice sviluppato da Microsoft, utilizzato per scrivere, testare e fare *debugging*[G] di codice. Permette l'installazione di estensioni per estendere le funzionalità dell'editor.
npm	10.9.0	Gestore di pacchetti che consente di gestire librerie e *dipendenze*[G] in JavaScript.
Node.js	20.x	Ambiente di esecuzione JavaScript multipiattaforma e open-source, utilizzato per sviluppare applicazioni lato server ed *API*[G].
VS Code Extension API	1.95.0	API ufficiale di Visual Studio Code per creare estensioni che interagiscono e si integrano con l'editor.
Git	2.42.0	Git è un sistema di controllo di versione distribuito, utilizzato per la gestione del codice.
Nest CLI	11.0.0	Interfaccia a riga di comando ufficiale di NestJS per generare progetti, moduli, controller e servizi.

Docker	28.0.1	Piattaforma per sviluppare, spedire ed eseguire applicazioni in container. Permette di isolare le dipendenze e semplificare il deployment.
Docker Compose	2.33.1	Strumento per definire e gestire applicazioni multi-container. Permette di configurare i servizi, le reti e i volumi necessari per l'applicazione.
Librerie e Framework		
NestJS	11.0.1	Framework progressivo per costruire applicazioni server-side efficienti e scalabili per Node.js con TypeScript.
Ollama	0.6.5	*Framework*[G] open-source che permette di eseguire modelli *LLM*[G] in locale.
Axios	1.9.0	Libreria JavaScript per effettuare richieste HTTP da browser e Node.js, permette la gestione di richieste asincrone.
Testing e Qualità del Codice		
Jest	29.7.0	Framework di testing JavaScript, utilizzato per eseguire test unitari e di integrazione. Integrato con NestJS.
Supertest	7.0.0	Libreria per testare API HTTP in Node.js, usata nei test di integrazione di NestJS.
ESLint	9.18.0	Strumento di *linting*[G] per identificare problemi di stile e errori nel codice TypeScript.
Prettier	3.4.2	Formattatore di codice opinato, usato per mantenere uno stile di codice coerente.
Mocha	10.0.10	Framework di test JavaScript, usato per eseguire test di integrazione per il plugin di VS Code. Attualmente, è l'unico framework di test di integrazione supportato da VS Code Extension API.

Table 1: Tecnologie coinvolte nel progetto

3 Architettura di deployment

Il sistema è basato su un'architettura distribuita *client-server*[G] a due tier, in cui la logica applicativa è suddivisa tra due componenti distinti: un client, ovvero l'estensione per Visual Studio Code, e un server, ovvero un'applicazione backend sviluppata con il framework NestJS.

Il client è rappresentato dall'estensione VS Code, installabile localmente sull'ambiente di sviluppo dell'utente. Questa componente si occupa della gestione dell'interfaccia utente, dell'interazione con l'utente e di una parte della logica applicativa, organizzata secondo un'architettura a livelli. In particolare, il client include un core layer che implementa alcune regole di dominio e funzionalità specifiche dell'estensione.

Il server consiste in un'applicazione backend realizzata con NestJS e containerizzata tramite Docker. Questa componente centralizza la logica di dominio principale e funge da punto di accesso per l'intero ecosistema attraverso un'interfaccia RESTful. Il servizio è pensato per essere eseguito in un container dedicato, facilitando il deployment.

La comunicazione tra i due tier avviene attraverso richieste HTTP, in particolare tramite API REST esposte dal backend. Questa separazione consente di distribuire e aggiornare in modo indipendente il client e il server. Inoltre, l'approccio distribuito facilita il debugging, l'estensibilità e l'evoluzione del sistema nel tempo.

Sebbene non si tratti di un'architettura a microservizi, la soluzione adottata offre una chiara separazione dei ruoli, garantendo una distribuzione efficace delle responsabilità. Questo approccio combina semplicità di gestione con modularità e scalabilità, rendendo l'intero sistema un esempio di architettura distribuita adatta a progetti in continua evoluzione.

4 Architettura logica

L'architettura del prodotto *Requirement Tracker Plug-in* è composta da due parti principali:

- **Frontend:** L'estensione per Visual Studio Code è stata sviluppata utilizzando una *layered architecture*[G], in modo da separare le responsabilità in moduli distinti. Il frontend è il responsabile dell'interfaccia utente e della comunicazione con il backend tramite API RESTful. Utilizza le *VS Code Extension API* per interagire con l'editor e fornire funzionalità come la visualizzazione dei requisiti e la generazione di suggerimenti.
  La decisione di applicare il pattern architetturale *Layered Architecture* per il frontend è stata motivata da diversi fattori:
  1. **Separazione delle responsabilità:** La *Layered Architecture* consente di separare le diverse responsabilità del sistema in moduli distinti, facilitando la comprensione

e la manutenibilità del codice. Ogni layer ha compiti specifici e può essere sviluppato e testato in modo indipendente.

2. **Estensibilità:** La struttura modulare consente di aggiungere nuove funzionalità o modificare quelle esistenti senza influire su altre parti del sistema. Ad esempio, è possibile aggiungere nuovi servizi o componenti visivi senza dover riscrivere l'intero codice.
3. **Testabilità:** La separazione dei layer facilita il testing, poiché ogni layer può essere testato in modo indipendente. Questo consente di identificare e risolvere i problemi più rapidamente, migliorando la qualità del codice.
4. **Manutenibilità:** La *Layered Architecture* rende il codice più manutenibile, poiché le modifiche possono essere apportate a un layer senza influire sugli altri. Questo è particolarmente utile in progetti complessi, dove le modifiche possono avere effetti a catena su altre parti del sistema.
5. **Facilità di integrazione:** La struttura modulare facilita l'integrazione con altre librerie o strumenti, consentendo di sfruttare le funzionalità esistenti senza dover riscrivere il codice.
6. **Chiarezza:** La *Layered Architecture* offre una chiara separazione tra i vari livelli del sistema, rendendo più facile per gli sviluppatori comprendere come funziona il sistema e come interagiscono le diverse parti.

- **Backend:** Il server NestJS è stato sviluppato con un *architettura esagonale*[G] ed è il responsabile della logica di *business* e dell'interazione con il modello LLM. Comunica con il frontend esponendo API RESTful e gestisce le richieste provenienti dall'estensione. Utilizza librerie come *Axios* per effettuare chiamate HTTP verso il server Ollama, che espone API per l'interazione con i modelli di *AI*[G] (LLM e di *embedding*[G]). Il server è progettato per essere facilmente estensibile e manutenibile, consentendo l'aggiunta di nuove funzionalità in modo semplice.
  Questo tipo di architettura porta numerosi vantaggi:
  - **Isolamento della logica di business:** La logica di business è separata dalle tecnologie esterne, facilitando la manutenibilità e il testing.
  - **Facilità di test:** La separazione tra *dominio* e *adapter* consente di testare la logica di business in modo isolato, senza dipendenze esterne. Questo semplifica la scrittura di test unitari e di integrazione.
  - **Modularità:** L'architettura esagonale consente di aggiungere o modificare facilmente gli *adapter* senza influire sulla logica di business. Questo facilita l'integrazione con nuove tecnologie o servizi esterni.

5 Pattern utilizzati

5.1 Dependency Injection (DI)

Dependency Injection è un pattern di progettazione ampiamente adottato per implementare efficacemente il principio di inversione delle dipendenze (*Dependency Inversion Principle*). Questo pattern si basa sul fornire a un oggetto (il "dipendente") le dipendenze di cui necessita dall'esterno, invece di lasciare che sia l'oggetto stesso a crearle o a cercarle attivamente al suo interno. Le dipendenze possono essere "iniettate" in vari modi, i più comuni dei quali includono l'iniezione tramite costruttore, l'iniezione tramite metodo setter o l'iniezione tramite interfaccia. L'utilizzo della *DI* semplifica notevolmente la gestione delle dipendenze, facilita l'isolamento dei componenti per i test unitari (consentendo l'uso di *mock* o *stub*) e promuove un'architettura software più pulita e basata sull'interazione tra astrazioni piuttosto che su implementazioni concrete rigidamente accoppiate. Nel progetto è stato utilizzato il pattern di *dependency injection* tramite costruttore.

5.2 Facade

Il pattern *Facade* è un design pattern strutturale che fornisce un'interfaccia semplice per interagire con un sistema complesso. Aiuta a ridurre la complessità nascondendo i dettagli interni e fornendo un unico punto di accesso. L'uso del pattern *Facade* migliora la leggibilità del codice e semplifica l'interazione dei client con il sistema, promuovendo una chiara separazione tra le componenti interne e l'accesso esterno.

5.3 Adapter Pattern

L'*Adapter Pattern* è un design pattern strutturale che consente a classi con interfacce incompatibili di lavorare insieme. Funziona come un intermediario, traducendo le chiamate tra due interfacce diverse in modo che possano comunicare senza modifiche al codice esistente.

6 Architettura di dettaglio

6.1 Frontend

6.1.1 Architettura a strati

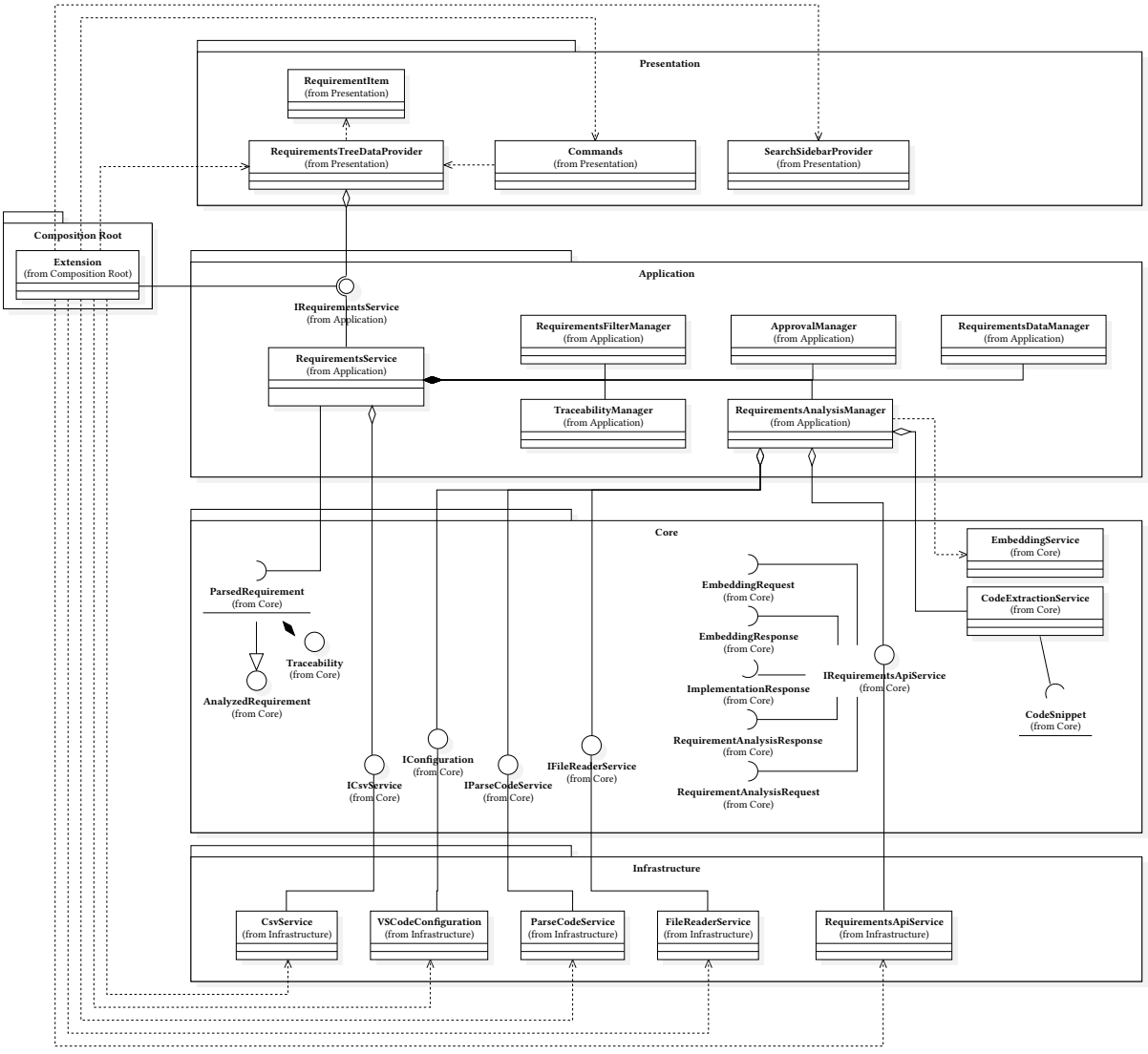

Figure 1: Diagramma generale del Frontend

L'architettura a strati è organizzata in quattro livelli principali: *Presentation*, *Application*, *Core* e *Infrastructure*. Ogni strato ha responsabilità specifiche per garantire una separazione delle responsabilità e facilitare la manutenibilità e l'estensibilità del sistema.

- **Presentation Layer**: Questo strato è responsabile dell'interfaccia utente e dell'interazione con l'utente finale. Si occupa di visualizzare i dati e raccogliere input, comunicando direttamente con l'*Application Layer* per eseguire le operazioni richieste.
- **Application Layer**: Funziona come intermediario tra il *Presentation Layer* e il *Core Layer*. Coordina le operazioni dell'applicazione, applica la logica specifica del caso d'uso e gestisce il flusso di dati tra i vari strati.
- **Core Layer**: Il cuore della logica di business, dove risiedono i modelli e le regole di business. Questo strato definisce contratti (interfacce) per servizi che lo strato di infrastruttura deve implementare e che saranno successivamente iniettati in altri strati grazie alla *dependency injection*. È completamente indipendente dagli altri strati.
- **Infrastructure Layer**: Fornisce servizi tecnici e interfacce per l'accesso ai dati, la comunicazione con servizi esterni e altre funzionalità infrastrutturali. È l'unico strato che comunica direttamente con i sistemi esterni, mantenendo gli altri strati isolati da dettagli implementativi.

L'ordine delle dipendenze è strettamente controllato: ogni strato può dipendere solo dallo strato sottostante, prevenendo accoppiamenti ciclici e garantendo un'architettura scalabile e facilmente manutenibile.

6.1.2 Presentation Layer

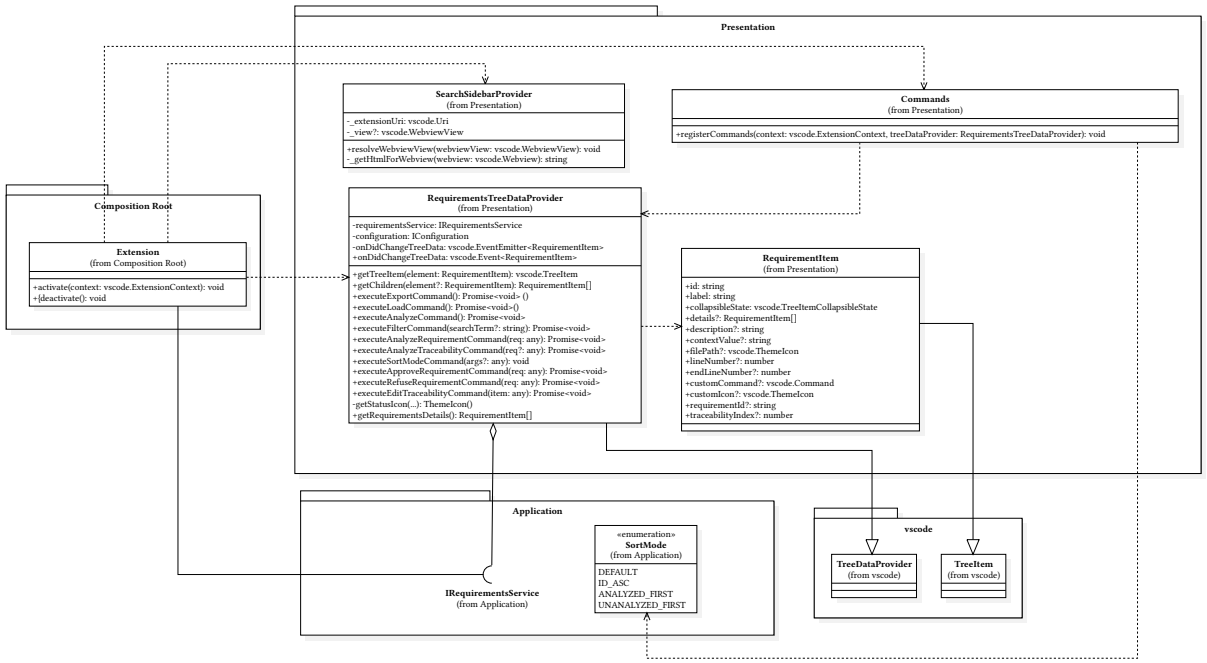

Figure 2: Diagramma dello strato Presentation

Il *Presentation Layer* è responsabile dell'interfaccia utente e dell'interazione con l'utente finale. Questo strato gestisce la visualizzazione dei dati e la raccolta degli input, comunicando con l'*Application Layer* per eseguire le operazioni richieste. Nel contesto dell'estensione VS Code, il *Presentation Layer* include i componenti che interagiscono direttamente con l'API di VS Code per fornire funzionalità di visualizzazione e interazione.

6.1.2.1 Classe: `RequirementsTreeDataProvider`

Implementa l'interfaccia `vscode.TreeDataProvider` per fornire i dati necessari alla visualizzazione dell'albero dei requisiti nell'interfaccia di VS Code. Funge da ponte tra l'interfaccia utente (il plugin) e il layer applicativo, delegando le operazioni al `RequirementsService`. Utilizza *dependency injection* per ricevere le dipendenze tramite il costruttore. Aggiunge la visualizzazione degli errori e dei messaggi di avviso relativi alle operazioni.

Metodi:

- **`getTreeItem`**: Restituisce l'elemento dell'albero corrispondente a un determinato `RequirementItem`. Questo metodo è richiesto dall'interfaccia `TreeDataProvider` di VS Code.

- **`getChildren`**: Restituisce i figli di un elemento dell'albero o gli elementi di primo livello se non viene specificato alcun elemento. Utilizza il `RequirementsService` per ottenere i requisiti filtrati.
- **`executeLoadCommand`**: Gestisce il comando per caricare i requisiti da un file CSV. Mostra un dialogo per la selezione del file e utilizza il `RequirementsService` per importare i dati.
- **`executeExportCommand`**: Gestisce il comando per esportare i requisiti in un file CSV utilizzando il `RequirementsService`.
- **`executeAnalyzeCommand`**: Gestisce il comando per analizzare tutti i requisiti, verificando prima se ci sono requisiti caricati e se hanno informazioni di tracciabilità. Mostra avvisi appropriati e utilizza una barra di progresso per indicare lo stato dell'analisi.
- **`executeFilterCommand`**: Gestisce il comando per filtrare i requisiti in base a un termine di ricerca. Se non viene fornito un termine, mostra un *input box*[G] per richiederlo all'utente.
- **`executeAnalyzeRequirementCommand`**: Gestisce il comando per analizzare un singolo requisito selezionato. Verifica prima se il requisito ha informazioni di tracciabilità e mostra messaggi appropriati durante il processo.
- **`executeAnalyzeTraceabilityCommand`**: Gestisce il comando per analizzare la tracciabilità di un requisito o di tutti i requisiti. Utilizza una barra di progresso per indicare lo stato dell'analisi quando si analizzano tutti i requisiti.
- **`executeSortModeCommand`**: Gestisce il comando per impostare la modalità di ordinamento dei requisiti.
- **`executeApproveRequirementCommand`**: Gestisce il comando per approvare un requisito. Verifica prima se il requisito può essere approvato e mostra messaggi appropriati.
- **`executeRefuseRequirementCommand`**: Gestisce il comando per rifiutare un requisito. Verifica prima se il requisito può essere rifiutato e mostra messaggi appropriati.
- **`executeEditTraceabilityCommand`**: Gestisce il comando per modificare un collegamento di tracciabilità. Mostra un input box per consentire all'utente di modificare il percorso del file e i numeri di riga.
- **`getStatusIcon`**: Metodo privato che determina l'icona di stato appropriata per un requisito in base al suo stato di analisi e approvazione. Utilizza diverse icone per rappresentare stati come approvato, rifiutato, conforme o non conforme.
- **`getRequirementsDetails`**: Metodo privato che costruisce la struttura ad albero dei requisiti con tutti i dettagli e le informazioni di analisi. Crea una gerarchia di

`RequirementItem` per visualizzare requisiti, tracciabilità, risultati dell'analisi, problemi e suggerimenti.

Campi:

- **`_onDidChangeTreeData`**: Emettitore di eventi per notificare i cambiamenti nei dati dell'albero.
- **`onDidChangeTreeData`**: Evento che viene attivato quando i dati dell'albero cambiano.
- **`requirementsService`**: Servizio che gestisce i requisiti e le operazioni correlate.
- **`configuration`**: Servizio che gestisce la configurazione dell'estensione.

6.1.2.2 Classe: `RequirementItem`

Estende la classe `vscode.TreeItem` di VS Code per rappresentare un elemento nell'albero dei requisiti. Ogni istanza rappresenta un requisito o un dettaglio di un requisito nell'interfaccia utente.

Campi:

- **`id`**: Identificatore univoco del requisito.
- **`label`**: Etichetta visualizzata nell'interfaccia utente.
- **`collapsibleState`**: Stato di espansione dell'elemento nell'albero.
- **`details`**: Elementi figli che rappresentano i dettagli del requisito.
- **`description`**: Descrizione aggiuntiva visualizzata accanto all'etichetta.
- **`contextValue`**: Valore di contesto utilizzato per determinare quali comandi sono disponibili per l'elemento.
- **`filePath`**: Percorso del file associato al requisito per la tracciabilità.
- **`lineNumber`**: Numero di riga nel file associato.
- **`endLineNumber`**: Numero di riga finale nel file associato per evidenziare un intervallo.
- **`customCommand`**: Comando personalizzato da eseguire quando l'elemento viene selezionato.
- **`customIcon`**: Icona personalizzata da visualizzare accanto all'elemento.
- **`requirementId`**: ID del requisito associato (utilizzato per gli elementi di tracciabilità).
- **`traceabilityIndex`**: Indice dell'elemento di tracciabilità (utilizzato per la modifica).

- **`command`**: Comando che viene eseguito quando l'elemento viene selezionato (ereditato da `TreeItem`).
- **`iconPath`**: Percorso dell'icona o icona tematica da visualizzare (ereditato da `TreeItem`).

6.1.2.3 Classe: `SearchSidebarProvider`

Implementa l'interfaccia `vscode.WebviewViewProvider` per fornire una vista *webview* personalizzata che consente agli utenti di importare, esportare, filtrare, tracciare e analizzare i requisiti attraverso un'interfaccia grafica.

Metodi:

- **`resolveWebviewView`**: Inizializza la vista webview con HTML, CSS e JavaScript. Configura i gestori di messaggi per comunicare tra la webview e l'estensione.
- **`_getHtmlForWebview`**: Genera il codice HTML per la webview, includendo i riferimenti ai file CSS e JavaScript necessari.

Campi:

- **`_view`**: Riferimento alla vista webview corrente.
- **`_extensionUri`**: URI dell'estensione, utilizzato per risolvere i percorsi delle risorse.

6.1.2.4 Modulo: `commands`

Modulo che registra tutti i comandi dell'estensione con l'API di VS Code. Questi comandi possono essere invocati dall'interfaccia utente o da altre parti dell'estensione.

Metodi:

- **`registerCommands`**: Registra tutti i comandi dell'estensione con VS Code e li collega alle funzioni corrispondenti nel `treeDataProvider`.

Comandi registrati:

- **`requirementsTree.analyze`**: Analizza tutti i requisiti.
- **`requirementsTree.filter`**: Filtra i requisiti in base a un termine di ricerca.
- **`requirementsTree.analyzeRequirement`**: Analizza un singolo requisito.
- **`requirementsTree.openFileAtLine`**: Apre un file nell'editor a una riga specifica.
- **`requirementsTree.load`**: Carica i requisiti da un file CSV.
- **`requirementsTree.analyzeTraceability`**: Analizza la tracciabilità dei requisiti.

- **`requirementsTree.setSortMode`**: Imposta la modalità di ordinamento.
- **`requirementsTree.setSortAnalyzedFirst`**: Ordina mostrando prima i requisiti analizzati.
- **`requirementsTree.setSortUnanalyzedFirst`**: Ordina mostrando prima i requisiti non analizzati.
- **`requirementsTree.approveRequirement`**: Approva un requisito.
- **`requirementsTree.refuseRequirement`**: Rifiuta un requisito.
- **`requirementsTree.editTraceability`**: Modifica un collegamento di tracciabilità.
- **`requirementsTree.exportCSV`**: Esporta i requisiti in un file CSV.

6.1.3 Application Layer

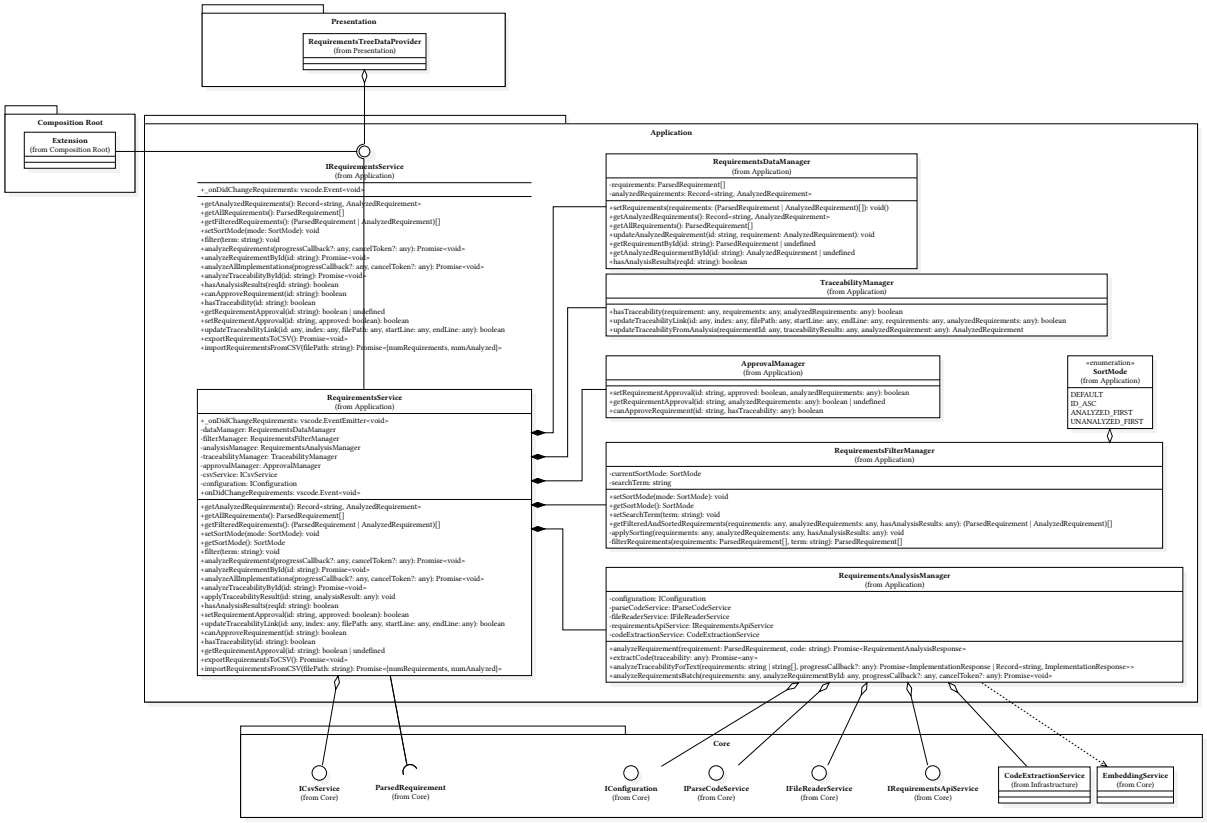

Figure 3: Diagramma dello strato Application

L'*Application Layer* funge da intermediario tra il *Presentation Layer* e il *Core Layer*. Questo strato coordina le operazioni dell'applicazione, applica la logica specifica dei casi d'uso e gestisce il flusso di dati tra i vari strati. Implementa il pattern *Facade* per semplificare l'interazione con i *servizi* sottostanti e organizza la logica in *manager* specializzati.

6.1.3.1 Interfaccia: `IRequirementsService`

Definisce il contratto per il *servizio* di gestione dei requisiti. Questa interfaccia stabilisce tutte le operazioni che il *Presentation Layer* può richiedere all'*Application Layer*, fornendo una chiara separazione tra i due strati.

Metodi:

- **`getAnalyzedRequirements`**: Restituisce un record di tutti i requisiti analizzati, indicizzati per ID.
- **`getAllRequirements`**: Restituisce un array di tutti i requisiti.

- **`getFilteredRequirements`**: Restituisce i requisiti filtrati in base ai criteri di ricerca e ordinamento correnti.
- **`setSortMode`**: Imposta la modalità di ordinamento per i requisiti.
- **`filter`**: Applica un filtro di ricerca ai requisiti.
- **`analyzeRequirements`**: Analizza tutti i requisiti, con supporto per callback di progresso e cancellazione.
- **`analyzeRequirementById`**: Analizza un singolo requisito in base al suo ID.
- **`analyzeAllImplementations`**: Analizza l'implementazione di tutti i requisiti.
- **`analyzeTraceabilityById`**: Analizza la tracciabilità di un requisito specifico.
- **`hasAnalysisResults`**: Verifica se un requisito ha risultati di analisi disponibili.
- **`canApproveRequirement`**: Verifica se un requisito può essere approvato.
- **`hasTraceability`**: Verifica se un requisito ha informazioni di tracciabilità.
- **`getRequirementApproval`**: Restituisce lo stato di un requisito (approvato/non approvato).
- **`setRequirementApproval`**: Imposta lo stato di approvazione di un requisito.
- **`updateTraceabilityLink`**: Aggiorna un collegamento di tracciabilità per un requisito.
- **`exportRequirementsToCSV`**: Esporta i requisiti in un file CSV.
- **`importRequirementsFromCSV`**: Importa i requisiti da un file CSV.

Eventi:

- **`onDidChangeRequirements`**: Evento emesso quando i requisiti cambiano.

Pattern: Definisce un'interfaccia per il pattern *Facade* che nasconde la complessità dei *manager* sottostanti.

6.1.3.2 Classe: `RequirementsService`

Implementazione dell'interfaccia `IRequirementsService` che agisce come *Facade* per i vari *manager* specializzati. Coordina le operazioni tra i *manager* e fornisce un'interfaccia unificata al *Presentation Layer*.

Metodi:

- **`setRequirements`**: Delega al `DataManager` l'impostazione dei requisiti e notifica i cambiamenti.
- **`getAnalyzedRequirements`**: Delega al `DataManager` per ottenere i requisiti analizzati.

- **`getAllRequirements`**: Delega al `DataManager` per ottenere tutti i requisiti.
- **`getFilteredRequirements`**: Utilizza il `FilterManager` per ottenere i requisiti filtrati e ordinati.
- **`getSortMode`**: Delega al `FilterManager` per ottenere la modalità di ordinamento corrente.
- **`filter`**: Delega al `FilterManager` l'applicazione del filtro di ricerca.
- **`analyzeRequirements`**: Coordina l'`AnalysisManager` per analizzare tutti i requisiti in batch.
- **`analyzeRequirementById`**: Coordina l'`AnalysisManager` per analizzare un singolo requisito.
- **`analyzeAllImplementations`**: Coordina l'`AnalysisManager` per analizzare l'implementazione di tutti i requisiti.
- **`analyzeTraceabilityById`**: Coordina l'`AnalysisManager` per analizzare la tracciabilità di un requisito specifico.
- **`hasAnalysisResults`**: Verifica nel `DataManager` se un requisito ha risultati di analisi.
- **`canApproveRequirement`**: Utilizza l'`ApprovalManager` per verificare se un requisito può essere approvato.
- **`hasTraceability`**: Verifica nel `DataManager` se un requisito ha informazioni di tracciabilità.
- **`getRequirementApproval`**: Utilizza l'`ApprovalManager` per ottenere lo stato di approvazione.
- **`setRequirementApproval`**: Utilizza l'`ApprovalManager` per impostare lo stato di approvazione.
- **`updateTraceabilityLink`**: Utilizza il `TraceabilityManager` per aggiornare un collegamento di tracciabilità.
- **`applyTraceabilityResult`**: Utilizza il `TraceabilityManager` per applicare i risultati dell'analisi.
- **`exportRequirementsToCSV`**: Utilizza il `CsvService` per esportare i requisiti.
- **`importRequirementsFromCSV`**: Utilizza il `CsvService` per importare i requisiti.

Campi:

- **`onDidChangeRequirements`**: Emettitore di eventi per notificare i cambiamenti nei requisiti.

- **`requirementsDataManager`**: Gestisce lo stato e l'accesso ai dati dei requisiti.
- **`requirementsFilterManager`**: Gestisce il filtraggio e l'ordinamento dei requisiti.
- **`requirementsAnalysisManager`**: Gestisce l'analisi dei requisiti e le chiamate API.
- **`requirementsApprovalManager`**: Gestisce l'approvazione dei requisiti.
- **`requirementsTraceabilityManager`**: Gestisce la tracciabilità dei requisiti.
- **`csvService`**: Servizio per l'importazione e l'esportazione di CSV.

6.1.3.3 Classe: `RequirementsDataManager`

Gestisce l'archiviazione e l'accesso ai dati dei requisiti. Mantiene lo stato dei requisiti e fornisce metodi per accedervi e modificarli.

Metodi:

- **`setRequirements`**: Imposta l'elenco completo dei requisiti.
- **`getAllRequirements`**: Restituisce tutti i requisiti.
- **`getAnalyzedRequirements`**: Restituisce i requisiti che hanno risultati di analisi.
- **`getRequirementById`**: Trova un requisito specifico per ID.
- **`getAnalyzedRequirementById`**: Trova un requisito analizzato specifico per ID.
- **`updateAnalyzedRequirement`**: Aggiorna un requisito con i risultati dell'analisi.
- **`hasAnalysisResults`**: Verifica se un requisito ha risultati di analisi disponibili.

Campi:

- **`requirements`**: Array di tutti i requisiti.
- **`analyzedRequirements`**: Record dei requisiti analizzati, indicizzati per ID.

6.1.3.4 Enum: `SortMode`

Definisce le modalità di ordinamento disponibili per i requisiti.

Valori:

- **`DEFAULT`**: Nessun ordinamento applicato.
- **`ID_ASC`**: Ordinamento per ID in ordine crescente.
- **`ANALYZED_FIRST`**: Requisiti analizzati mostrati per primi.
- **`UNANALYZED_FIRST`**: Requisiti non analizzati mostrati per primi.

6.1.3.5 Classe: `RequirementsFilterManager`

Gestisce il filtraggio e l'ordinamento dei requisiti in base a criteri specificati dall'utente.

Metodi:

- **`setSearchTerm`**: Imposta il termine di ricerca per il filtraggio.
- **`setSortMode`**: Imposta la modalità di ordinamento.
- **`getSortMode`**: Ottiene la modalità di ordinamento corrente.
- **`getFilteredAndSortedRequirements`**: Applica filtri e ordinamento ai requisiti.
- **`applySorting`**: Applica l'ordinamento ai requisiti in base alla modalità corrente.
- **`filterRequirements`**: Filtra i requisiti in base al termine di ricerca.

Campi:

- **`currentSortMode`**: Modalità di ordinamento corrente.
- **`searchTerm`**: Termine di ricerca corrente.

6.1.3.6 Classe: `RequirementsAnalysisManager`

Gestisce l'analisi dei requisiti e del codice sorgente. Coordina le chiamate ai servizi esterni per l'analisi e l'elaborazione dei risultati.

Metodi:

- **`analyzeRequirement`**: Analizza un singolo requisito rispetto al codice fornito.
- **`extractCode`**: Estrae il codice sorgente dalle informazioni di tracciabilità.
- **`analyzeRequirementsBatch`**: Analizza un batch di requisiti con supporto per progresso e cancellazione.
- **`analyzeTraceabilityForText`**: Analizza la tracciabilità per uno o più testi di requisiti.

Campi:

- **`configuration`**: Configurazione dell'estensione.
- **`parseCodeService`**: Servizio per il parsing del codice.
- **`fileReaderService`**: Servizio per la lettura dei file.
- **`requirementsApiService`**: Servizio per le chiamate API relative ai requisiti.

6.1.3.7 Classe: `TraceabilityManager`

Gestisce le informazioni di tracciabilità tra requisiti e codice sorgente.

Metodi:

- **`hasTraceability`**: Verifica se un requisito ha informazioni di tracciabilità.
- **`updateTraceabilityLink`**: Aggiorna un collegamento di tracciabilità specifico.
- **`updateTraceabilityFromAnalysis`**: Aggiorna la tracciabilità di un requisito con i risultati dell'analisi.

6.1.3.8 Classe: `ApprovalManager`

Gestisce lo stato di approvazione dei requisiti.

Metodi:

- **`canApproveRequirement`**: Verifica se un requisito può essere approvato.
- **`setRequirementApproval`**: Imposta lo stato di approvazione di un requisito.
- **`getRequirementApproval`**: Ottiene lo stato di approvazione di un requisito.

6.1.4 Core Layer

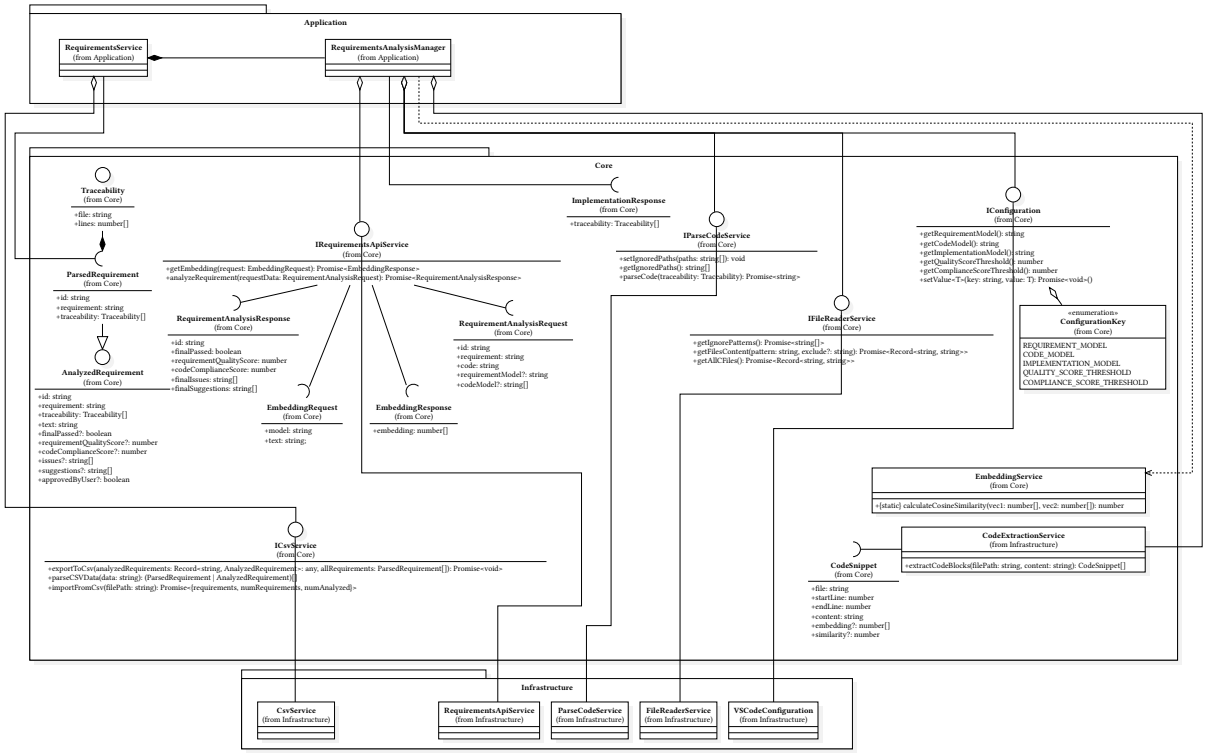

Figure 4: Diagramma dello strato Core

Il *Core Layer* rappresenta il cuore della logica di business dell'applicazione, dove risiedono i modelli e le regole di business fondamentali. Questo strato definisce contratti (interfacce) per *servizi* che lo strato di infrastruttura deve implementare e che saranno successivamente iniettati in altri strati grazie alla *dependency injection*. È completamente indipendente dagli altri strati e non contiene riferimenti a tecnologie esterne.

6.1.4.1 Interfaccia: `IConfiguration`

Definisce il contratto per il servizio di configurazione dell'estensione. Questa interfaccia permette di accedere alle impostazioni di configurazione in modo indipendente dall'implementazione specifica.

Metodi:

- **`getRequirementModel`**: Ottiene il nome del modello AI per l'analisi dei requisiti.
- **`getCodeModel`**: Ottiene il nome del modello AI per l'analisi del codice.

- **`getImplementationModel`**: Ottiene il nome del modello AI per l'analisi dell'implementazione.
- **`getQualityScoreThreshold`**: Ottiene la soglia di punteggio di qualità per i requisiti.
- **`getComplianceScoreThreshold`**: Ottiene la soglia di punteggio di conformità per i requisiti.
- **`setValue`**: Imposta un valore di configurazione.
- **`onConfigurationChanged`**: Registra un listener per i cambiamenti di configurazione.

Enum associato: **`ConfigurationKey`**

- **`REQUIREMENT_MODEL`**: Chiave per il modello di analisi dei requisiti.
- **`CODE_MODEL`**: Chiave per il modello di analisi del codice.
- **`IMPLEMENTATION_MODEL`**: Chiave per il modello di analisi dell'implementazione.
- **`QUALITY_SCORE_THRESHOLD`**: Chiave per la soglia di punteggio di qualità.
- **`COMPLIANCE_SCORE_THRESHOLD`**: Chiave per la soglia di punteggio di conformità.

6.1.4.2 Interfaccia: `ICsvService`

Definisce il contratto per il servizio di importazione ed esportazione di dati in formato CSV. Questa interfaccia permette di gestire le operazioni di I/O relative ai requisiti in modo indipendente dall'implementazione.

Metodi:

- **`exportToCsv`**: Esporta i requisiti in un file CSV.
- **`parseCSVData`**: Analizza i dati CSV e li converte in oggetti requisito.
- **`importFromCsv`**: Importa requisiti da un file CSV.

6.1.4.3 Interfaccia: `IFileReaderService`

Definisce il contratto per il servizio di lettura dei file. Questa interfaccia permette di accedere al filesystem in modo indipendente dall'implementazione specifica.

Metodi:

- **`getIgnorePatterns`**: Ottiene i pattern di esclusione dal file .reqignore.
- **`getFilesContent`**: Ottiene il contenuto di tutti i file che corrispondono a un pattern.
- **`getAllCFiles`**: Ottiene tutti i file C nel workspace corrente.

6.1.4.4 Interfaccia: `IParseCodeService`

Definisce il contratto per il servizio di parsing del codice sorgente. Questa interfaccia permette di estrarre informazioni strutturali dal codice in modo indipendente dall'implementazione.

Metodi:

- **`setIgnoredPaths`**: Imposta i percorsi da ignorare durante la ricerca dei file.
- **`getIgnoredPaths`**: Ottiene i percorsi attualmente ignorati.
- **`parseCode`**: Analizza il codice basato sulle informazioni di tracciabilità.

6.1.4.5 Interfaccia: `IRequirementsApiService`

Definisce il contratto per il servizio di comunicazione con l'API dei requisiti. Questa interfaccia permette di effettuare chiamate API in modo indipendente dall'implementazione specifica.

Metodi:

- **`analyzeRequirement`**: Invia una richiesta di analisi dei requisiti all'API.
- **`getEmbedding`**: Ottiene un embedding vettoriale per un testo dall'API.

6.1.4.6 Modelli

Definisce i modelli di dati fondamentali utilizzati nell'applicazione per rappresentare requisiti, analisi e tracciabilità.

Interfacce:

- **`Traceability`** Rappresenta un collegamento di tracciabilità tra un requisito e il codice sorgente.
  - **`file`**: Percorso del file.
  - **`lines`**: Numeri di riga nel file.
- **`CodeSnippet`** Rappresenta un frammento di codice estratto per l'analisi.
  - **`file`**: Percorso del file.
  - **`startLine`**: Riga di inizio.
  - **`endLine`**: Riga di fine.
  - **`content`**: Contenuto del frammento.

  - **`embedding`**: Vettore di embedding opzionale.
  - **`similarity`**: Punteggio di similarità opzionale.
- **`EmbeddingRequest`** Rappresenta una richiesta per ottenere un embedding vettoriale.
  - **`model`**: Nome del modello da utilizzare.
  - **`text`**: Testo da convertire in embedding.
- **`EmbeddingResponse`** Rappresenta la risposta a una richiesta di embedding.
  - **`embedding`**: Vettore di embedding risultante.
- **`ImplementationResponse`** Rappresenta la risposta a una richiesta di analisi dell'implementazione.
  - **`traceability`**: Collegamenti di tracciabilità trovati.
- **`RequirementAnalysisRequest`** Rappresenta una richiesta di analisi dei requisiti.
  - **`id`**: ID del requisito.
  - **`requirement`**: Testo del requisito.
  - **`code`**: Codice da analizzare.
  - **`requirementModel`**: Modello per l'analisi dei requisiti.
  - **`codeModel`**: Modello per l'analisi del codice.
- **`RequirementAnalysisResponse`** Rappresenta la risposta a una richiesta di analisi dei requisiti.
  - **`id`**: ID del requisito.
  - **`finalPassed`**: Indica se il requisito è soddisfatto.
  - **`requirementQualityScore`**: Punteggio di qualità del requisito.
  - **`codeComplianceScore`**: Punteggio di conformità del codice.
  - **`finalIssues`**: Problemi identificati.
  - **`finalSuggestions`**: Suggerimenti per miglioramenti.
- **`ParsedRequirement`** Rappresenta un requisito importato da CSV.
  - **`id`**: ID del requisito.
  - **`requirement`**: Testo del requisito.
  - **`traceability`**: Collegamenti di tracciabilità.
- **`AnalyzedRequirement`** Estende `ParsedRequirement` con i risultati dell'analisi.
  - **`text`**: Testo del requisito.
  - **`finalPassed`**: Indica se il requisito è soddisfatto.

- **`requirementQualityScore`**: Punteggio di qualità.
- **`codeComplianceScore`**: Punteggio di conformità.
- **`issues`**: Problemi identificati.
- **`suggestions`**: Suggerimenti.
- **`approvedByUser`**: Indica se l'utente ha approvato il requisito.

6.1.4.7 Servizio: `embeddingService`

Fornisce funzionalità per il calcolo della similarità tra vettori di embedding, utilizzato per l'analisi della tracciabilità.

Metodi:

- **`calculateCosineSimilarity`** Calcola la similarità del coseno tra due vettori di embedding.

6.1.4.8 Servizio: `codeExtractionService`

Fornisce funzionalità per l'estrazione di blocchi di codice dai file sorgente.

Metodi:

- **`extractCodeBlocks`** Estrae blocchi di codice significativi da un file sorgente.

6.1.5 Infrastructure Layer

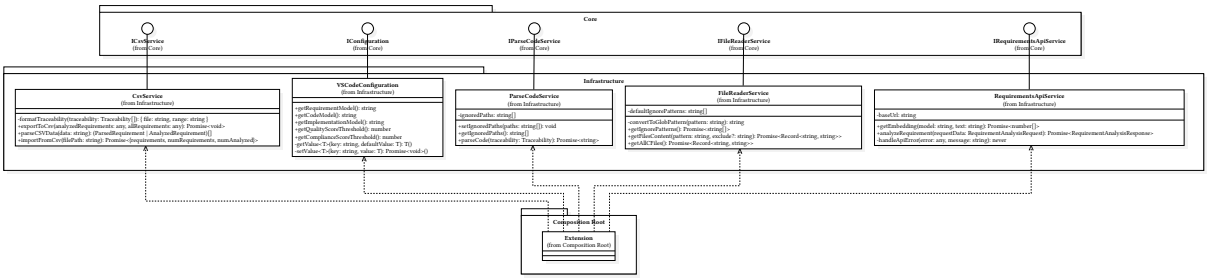

Figure 5: Diagramma dello strato Infrastructure

L'*Infrastructure Layer* fornisce servizi tecnici e interfacce per l'accesso ai dati, la comunicazione con servizi esterni e altre funzionalità infrastrutturali. È l'unico strato che comunica direttamente con i sistemi esterni, mantenendo gli altri strati isolati da dettagli implementativi. Questo strato implementa le interfacce definite nel *Core Layer*, seguendo il principio di inversione delle dipendenze.

6.1.5.1 Classe: `RequirementsApiService`

Implementa l'interfaccia `IRequirementsApiService` per gestire le comunicazioni con l'API di analisi dei requisiti. Questa classe è responsabile di effettuare le chiamate HTTP all'API esterna e di gestire le risposte e gli errori.

Metodi:

- **`getEmbedding`**: Invia una richiesta per ottenere un embedding vettoriale per un testo. Gestisce la comunicazione HTTP con l'endpoint appropriato dell'API e restituisce la risposta formattata.
- **`analyzeRequirement`**: Invia una richiesta di analisi dei requisiti all'API. Gestisce la comunicazione HTTP con l'endpoint appropriato dell'API e restituisce la risposta formattata.
- **`handleApiError`**: Metodo privato che gestisce gli errori delle chiamate API in modo consistente, fornendo messaggi di errore informativi.

Campi:

- **`baseUrl`**: URL di base dell'API con cui comunicare.

6.1.5.2 Classe: `VSCodeConfiguration`

Implementa l'interfaccia `IConfiguration` per gestire l'accesso alle impostazioni di configurazione dell'estensione. Utilizza l'API di configurazione di VS Code per leggere e scrivere le impostazioni.

Metodi:

- **`getRequirementModel`**: Ottiene il nome del modello AI configurato per l'analisi dei requisiti dalle impostazioni di VS Code.
- **`getCodeModel`**: Ottiene il nome del modello AI configurato per l'analisi del codice dalle impostazioni di VS Code.
- **`getImplementationModel`**: Ottiene il nome del modello AI configurato per l'analisi dell'implementazione dalle impostazioni di VS Code.
- **`getQualityScoreThreshold`**: Ottiene la soglia di punteggio di qualità configurata per i requisiti dalle impostazioni di VS Code.
- **`getComplianceScoreThreshold`**: Ottiene la soglia di punteggio di conformità configurata per i requisiti dalle impostazioni di VS Code.
- **`getValue`**: Metodo privato che ottiene un valore di configurazione con un valore predefinito di fallback.
- **`setValue`**: Imposta un valore di configurazione nelle impostazioni di VS Code.
- **`onConfigurationChanged`**: Registra un listener per i cambiamenti di configurazione in VS Code.

6.1.5.3 Classe: `CsvService`

Implementa l'interfaccia `ICsvService` per gestire l'importazione e l'esportazione di dati in formato CSV. Questa classe è responsabile della lettura e scrittura di file CSV e della conversione tra il formato CSV e gli oggetti di *dominio*.

Metodi:

- **`exportToCsv`**: Esporta i requisiti in un file CSV. Mostra un dialogo per la selezione del file di destinazione e scrive i dati nel formato appropriato.
- **`parseCSVData`**: Analizza i dati CSV e li converte in oggetti requisito. Gestisce il parsing delle colonne e la creazione degli oggetti di dominio.
- **`importFromCsv`**: Importa requisiti da un file CSV. Legge il file, analizza i dati e restituisce gli oggetti di dominio insieme a statistiche sull'importazione.

- **`formatTraceability`**: Metodo privato che formatta i dati di tracciabilità per l'esportazione CSV, convertendo l'array di oggetti `Traceability` in stringhe formattate per file e range.

6.1.5.4 Classe: `FileReaderService`

Implementa l'interfaccia `IFileReaderService` per gestire la lettura dei file dal filesystem. Questa classe è responsabile dell'accesso al filesystem e della lettura dei contenuti dei file.

Metodi:

- **`getAllCFiles`**: Ottiene tutti i file C nel workspace corrente. Cerca ricorsivamente i file con estensione .c e restituisce un record con i percorsi dei file e i loro contenuti.
- **`getFilesContent`**: Ottiene il contenuto di tutti i file che corrispondono a un pattern specificato, escludendo i file che corrispondono ai pattern di esclusione.
- **`getIgnorePatterns`**: Legge il file .reqignore e ottiene i pattern di esclusione. Se il file non esiste, utilizza i pattern di esclusione predefiniti.
- **`convertToGlobPattern`**: Metodo privato che converte un pattern di percorso in un pattern glob compatibile con VSCode.

Campi:

- **`defaultIgnorePatterns`**: Array di pattern predefiniti da ignorare per i progetti C, come cartelle di build, bin, lib, ecc.

6.1.5.5 Classe: `ParseCodeService`

Implementa l'interfaccia `IParseCodeService` per gestire il parsing del codice sorgente. Questa classe è responsabile dell'estrazione e dell'analisi del codice sorgente dai file specificati.

Metodi:

- **`parseCode`**: Analizza il codice basato sulle informazioni di tracciabilità. Estrae il codice dal file specificato e dalle righe indicate nelle informazioni di tracciabilità.
- **`setIgnoredPaths`**: Imposta i percorsi da ignorare durante la ricerca dei file.
- **`getIgnoredPaths`**: Ottiene i percorsi attualmente ignorati durante la ricerca dei file.

Campi:

- **`ignoredPaths`**: Array di pattern di percorsi da ignorare durante la ricerca dei file.

6.2 Backend

6.2.1 Architettura Esagonale

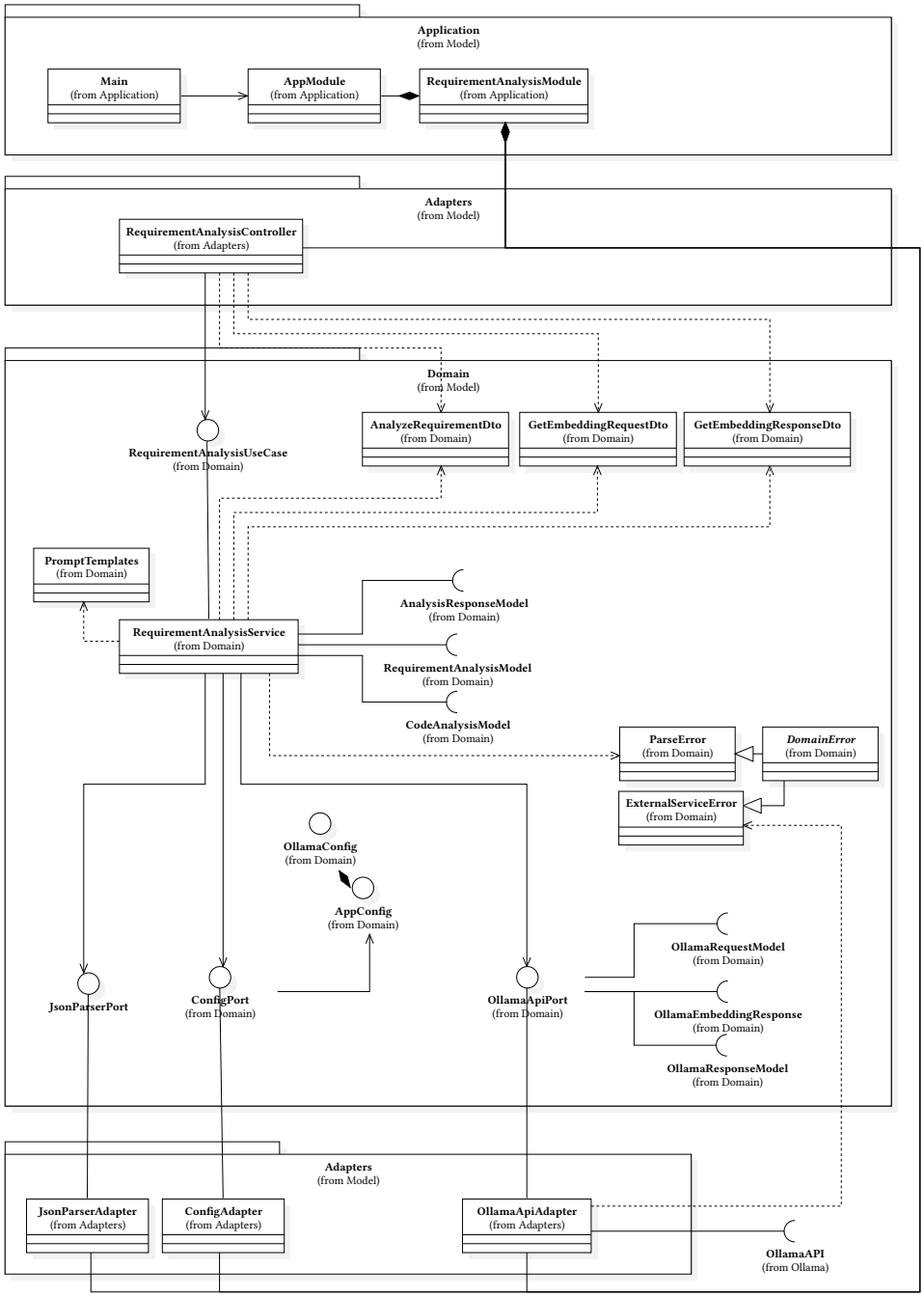

Figure 6: Diagramma dell'architettura esagonale del backend

L'architettura del backend è basata sul pattern architetturale *Hexagonal Architecture* (o *Ports and Adapters*), che consente di separare la logica di business dalle tecnologie esterne. Questa scelta architetturale facilita la manutenibilità e l'estensibilità del codice, consentendo di apportare modifiche senza influire su altre parti del sistema. La struttura del backend è organizzata in diversi livelli, ognuno con compiti specifici.

6.2.2 Domain

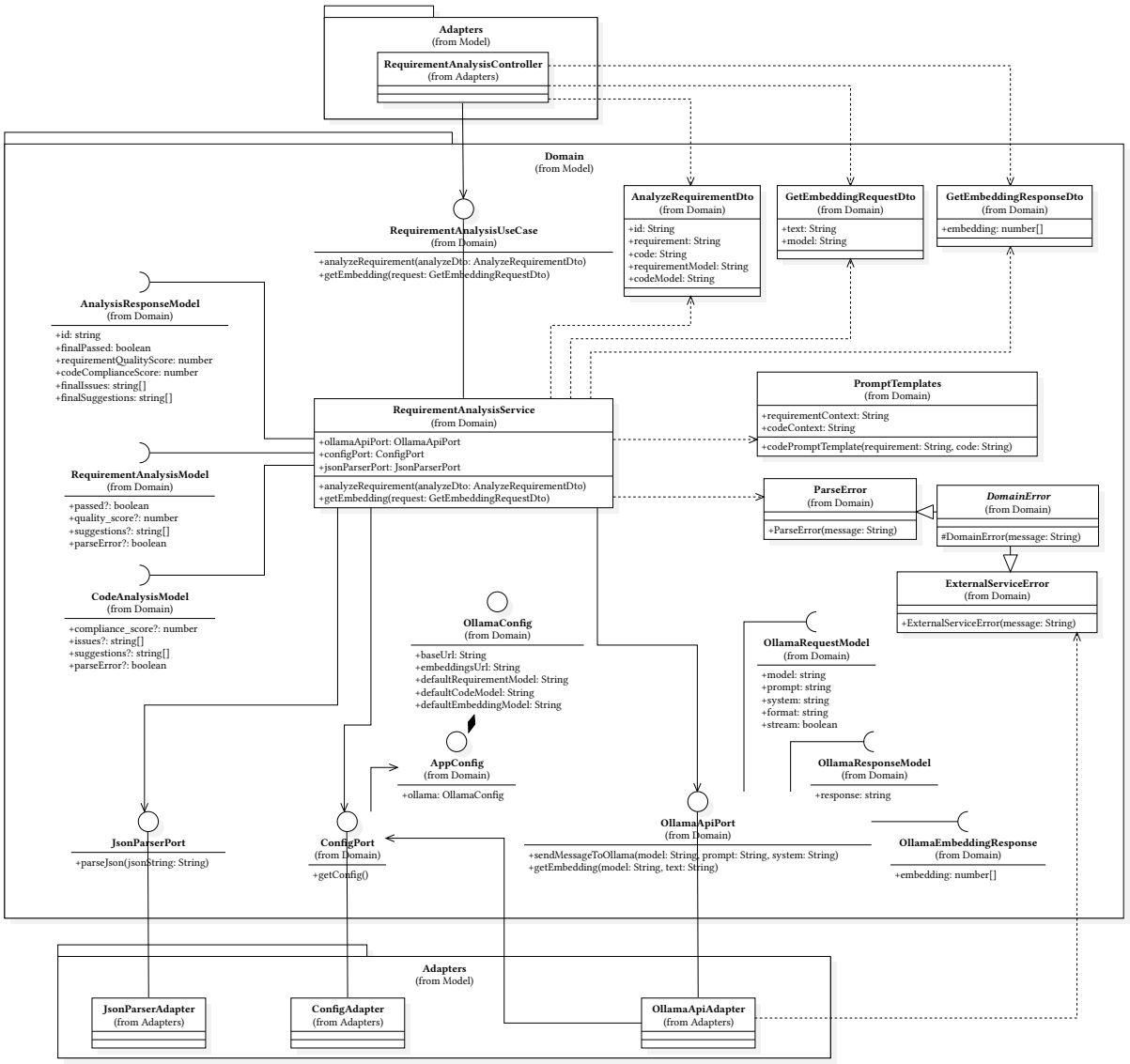

Figure 7: Diagramma architetturale del Domain

Il *Domain* dell'architettura esagonale racchiude tutta la business logic del backend, che include i modelli e i servizi per l'analisi. Il dominio è indipendente dalle tecnologie esterne e può essere testato in modo isolato.

6.2.2.1 Domain Services

Un servizio del dominio è una classe che utilizza la logica di business per fornire un servizio. I servizi del dominio sono responsabili dell'implementazione degli *inbound port* e della gestione delle interazioni tra i modeli. Nel nostro caso, esiste solo un servizio a livello di dominio.

6.2.2.1.1 Classe: `RequirementAnalysisService`

Questo servizio gestisce l'analisi dei requisiti e del codice, l'embedding e l'interpretazione delle risposte da parte del server di Ollama.

Metodi

- **`analyzeRequirement`**: Questo metodo si occupa di comunicare con Ollama per analizzare i requisiti forniti come input. Riceve un oggetto `AnalyzeRequirementDto` contenente i dati necessari per l'analisi e restituisce un oggetto `AnalysisResponseModel` che rappresenta i risultati dell'analisi, avvenuta tramite una richiesta HTTP al server di Ollama.
- **`getEmbedding`**: Questo metodo comunica con Ollama per generare gli embedding per i dati forniti. Riceve un oggetto `GetEmbeddingRequestDto` con le informazioni necessarie per calcolare gli embedding e restituisce un oggetto `GetEmbeddingResponseDto` contenente i risultati degli embedding.

Campi

- **`ollamaApiPort`**: *Outbound Port* per comunicare con l'API esterna Ollama.
- **`configPort`**: *Outbound Port* per la gestione delle configurazioni del server API. Viene iniettata tramite *dependency injection* e utilizzata per accedere alle configurazioni necessarie per l'analisi
- **`jsonParserPort`**: *Outbound Port* per la gestione dei risultati delle analisi. Viene iniettata tramite *dependency injection* e utilizzata per accedere alle configurazioni necessarie per l'analisi

6.2.2.2 Domain Models

I models sono classi che rappresentano gli oggetti principali del dominio applicativo. Essi contengono la logica di business e, nel nostro caso, sono principalmente interfacce che definiscono le proprietà che un oggetto dovrà possedere.

6.2.2.2.1 Interfaccia: `RequirementAnalysisModel`

Modello che rappresenta i risultati di un'analisi dei requisiti. Contiene informazioni legate ai risultati complessivi dell'analisi di un singolo requisito.

Campi:

- **`passed`**: Indica se l'analisi del requisito è stata superata.
- **`quality_score`**: Punteggio di qualità associato al requisito.
- **`suggestions`**: Suggerimenti per migliorare il requisito.
- **`parseError`**: Indica se si è verificato un errore durante il parsing.

6.2.2.2.2 Interfaccia: `CodeAnalysisModel`

Modello che descrive le proprietà dei risultati dell'analisi del codice associato a un requisito, come la qualità del codice, la copertura dei requisiti e i potenziali problemi.

Campi:

- **`compliance_score`**: Punteggio di conformità del codice rispetto ai requisiti.
- **`issues`**: Elenco dei problemi del codice riscontrati durante l'analisi.
- **`suggestions`**: Suggerimenti per migliorare il codice.
- **`parseError`**: Indica se si è verificato un errore durante il parsing.

6.2.2.2.3 Interfaccia: `AnalysisResponseModel`

Modello che descrive la struttura delle risposte delle analisi, includendo i risultati aggregati e i dettagli rilevanti.

Campi:

- **`id`**: Identificativo univoco del requisito.
- **`finalPassed`**: Indica se il requisito può essere considerato soddisfatto oppure no.
- **`requirementQualityScore`**: Punteggio di qualità dei requisiti.
- **`codeComplianceScore`**: Punteggio di conformità del codice.
- **`finalIssues`**: Elenco complessivo dei problemi riscontrati.
- **`finalSuggestions`**: Suggerimenti per migliorare il requisito, sia a livello di codice che a livello di correttezza del requisito.

6.2.2.2.4 Interfaccia: `OllamaResponseModel`

Modello che descrive la struttura della risposta ricevuta dall'API Odi llama.

Campi:

- **`response`**: Risposta ricevuta dall'API.

6.2.2.2.5 Interfaccia: `OllamaRequestModel`

Modello che rappresenta una richiesta inviata all'API Ollama.

Campi:

- **`model`**: Modello utilizzato per l'analisi del requisito.
- **`prompt`**: Prompt inviato all'API.
- **`system`**: *system prompt* utilizzato per l'analisi.
- **`format`**: Formato della risposta richiesta. Il formato di default utilizzato è il JSON.
- **`stream`**: Campo booleano per indicare se inviare la risposta in uno stream di oggetti o in un unico oggetto che rappresenti una risposta, nel nostro caso sarà sempre la seconda opzione.

6.2.2.2.6 Interfaccia: `OllamaEmbeddingResponse`

Modello che rappresenta una risposta di embedding dall'API Ollama.

Campi:

- **`embedding`**: Array di valori numerici che rappresentano l'embedding.

6.2.2.2.7 Interfaccia: `AppConfig`

Modello che rappresenta la configurazione dell'applicazione. Contiene informazioni relative alla configurazione del server API e dei modelli utilizzati.

Campi:

- **`ollama`**: Impostazioni di configurazione relative a Ollama.

6.2.2.2.8 Interfaccia: `OllamaConfig`

Modello che rappresenta la configurazione di Ollama. Contiene informazioni relative all'API di Ollama e ai modelli utilizzati.

Campi:

- **`baseUrl`**: URL di base dell'API di Ollama.
- **`embeddingsUrl`**: URL per ottenere gli embedding.
- **`defaultRequirementModel`**: Modello predefinito per l'analisi dei requisiti.
- **`defaultCodeModel`**: Modello predefinito per l'analisi del codice.
- **`defaultEmbeddingModel`**: Modello predefinito per l'embedding.

6.2.3 DTO

I *Data Transfer Objects* sono oggetti utilizzati per trasferire dati tra `ports` e `adapters`. I principali *DTO* sono i seguenti.

6.2.3.1 Classe: `AnalyzeRequirementDto`

DTO utilizzato per trasferire i dati relativi all'analisi dei requisiti.

Campi

- **`id`**: Identificativo univoco del requisito
- **`requirement`**: Testo del requisito da analizzare
- **`code`**: Codice sorgente associato al requisito
- **`requirementModel`**: Modello opzionale utilizzato per l'analisi del requisito
- **`codeModel`**: Modello opzionale utilizzato per l'analisi del codice

6.2.3.2 Classe: `GetEmbeddingRequestDto`

DTO che rappresenta una richiesta per ottenere gli embedding da un modello esterno.

Campi

- **`text`**: Testo per il quale calcolare gli embedding
- **`model`**: Modello opzionale utilizzato per generare gli embedding

6.2.3.3 Classe: `GetEmbeddingResponseDto`

DTO che rappresenta la risposta contenente gli embedding ottenuti.

Campi

- **`embedding`**: Array di numeri che rappresentano gli embedding calcolati.

6.2.4 Ports

I *Ports* sono interfacce che definiscono le operazioni che mettono in comunicazione il *domain* con l'esterno, come per esempio la comunicazione con Ollama per l'analisi dei requisiti. I *ports* fungono da contratti tra il *dominio* e gli *adapter*, consentendo inoltre di aggiungere o modificare l'implementazione di alcune operazioni senza influire sulla business logic.

6.2.4.1 Inbound Ports (Use Cases)

Gli *Inbound Ports* rappresentano le operazioni che possono essere invocate dai *controller* o da altri componenti esterni. Questi *ports* definiscono le interfacce per le operazioni principali del dominio.

6.2.4.1.1 Interfaccia: `RequirementAnalysisUseCase`

Interfaccia utilizzata per implementare le funzioni di analisi dei requisiti e di reperimento degli embeddings.

Metodi

- **`analyzeRequirement`**: Metodo che consente di analizzare un requisito specifico. Riceve un oggetto `AnalyzeRequirementDto` contenente i dati necessari per l'analisi e restituisce un oggetto `AnalysisResponseModel` con i risultati dell'analisi.
- **`getEmbedding`**: Metodo che consente di generare gli embedding per una determinata porzione di codice. Riceve un oggetto `GetEmbeddingRequestDto` con le informazioni necessarie e restituisce un oggetto `GetEmbeddingResponseDto` contenente gli embedding calcolati.

6.2.4.2 Outbound Ports

Gli *outbound ports* rappresentano le interfacce per le operazioni che il dominio può eseguire verso l'esterno, come la comunicazione con servizi esterni. Questi *ports* definiscono le interfacce per gli *adapter* che implementano le funzionalità necessarie.

6.2.4.2.1 Interfaccia: `OllamaApiPort`

Interfaccia che fornisce metodi di comunicazione con l'API del server di Ollama.

Metodi:

- **`sendMessageToOllama`**: Metodo per inviare un messaggio all'API Ollama, specificando il modello, il prompt e il system prompt. Restituisce un oggetto di tipo `OllamaResponseModel`.
- **`getEmbedding`**: Metodo per ottenere gli embedding da Ollama per un determinato testo, specificando il modello. Restituisce un array di numeri.

6.2.4.2.2 Interfaccia: `ConfigPort`

Interfaccia per la gestione della configurazione del server API.

Metodi:

- **`getConfig`**: Metodo per recuperare la configurazione del server API.

6.2.4.2.3 Interfaccia: `JsonParserPort`

Interfaccia che offre metodi per il parsing JSON. **Metodi**:

- **`parseJson`**: Metodo per deserializzare una stringa JSON in un oggetto.

6.2.5 Adapters

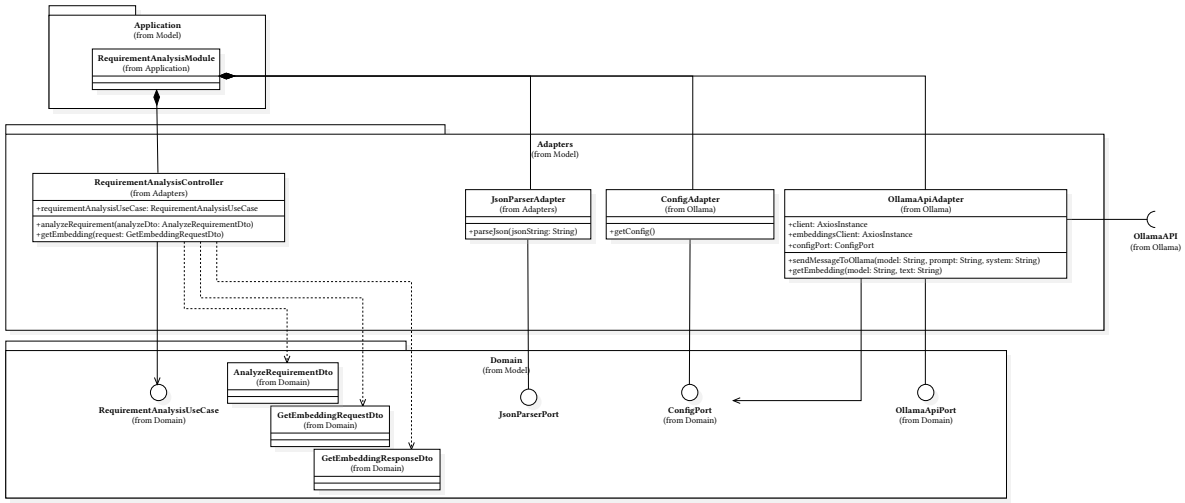

Figure 8: Diagramma archiettturale degli Adapters

Gli *Adapters* sono implementazioni concrete dei ports che consentono al *domain* di interagire con il mondo esterno, garantendo che il dominio rimanga isolato e indipendente.

6.2.5.1 Inbound Adapters (Controllers)

I *controllers* sono componenti che gestiscono le richieste da parte di client vari e inoltrano le operazioni da svolgere ai servizi del *dominio*. I *controller* sono implementati come *adapter* e si interfacciano con i *ports* per invocare la logica di business.

6.2.5.1.1 Classe: `RequirementAnalysisController`

Questo *adapter* è responsabile di ricevere le richieste dall'esterno (ad esempio, dal frontend), validarle e inoltrarle ai servizi del *dominio* tramite i ports. Inoltre, si occupa di formattare le risposte provenienti dal *dominio* in un formato comprensibile per il client. Infine, esso gestisce gli endpoint HTTP esposti dal backend, in modo da permettere la comunicazione dei requisiti da analizzare da parte del frontend.

Metodi

- **`analyzeRequirement`**: Utilizza i metodi forniti dall'*inbound port* `RequirementAnalysisUseCase` per analizzare i requisiti e restituire i risultati al client.
- **`getEmbedding`**: Utilizza i metodi forniti dall'*inbound port* `RequirementAnalysisUseCase` per generare gli embedding e restituirli al client.

Campi

- **`requirementAnalysisUseCase`**: attributo rappresentante l'omonimo *inbound port*. Una istanza effettiva di questo *port* viene iniettata a runtime tramite *dependency injection* e utilizzata per accedere alle funzionalità di analisi dei requisiti e generazione degli embedding.

6.2.5.2 Outbound Adapters

Gli *outbound adapters* sono implementazioni concrete degli *outbound ports* che consentono al *dominio* di interagire con servizi esterni, come API o sistemi di archiviazione. Questi *adapter* si interfacciano con i *ports* per invocare le operazioni necessarie.

6.2.5.2.1 Classe: `OllamaApiAdapter`

Questo *adapter* implementa l'interfaccia `OLLAMA_API_PORT` e si occupa di comunicare con l'API esterna Ollama. Gestisce le richieste HTTP verso l'API, il parsing delle risposte e la gestione degli errori derivanti da problemi di rete o di servizio.

Metodi

- **`sendMessageToOllama`**: Metodo per inviare un messaggio all'API Ollama, specificando il modello, il prompt e il `system prompt`. Restituisce un oggetto di tipo `OllamaResponseModel`.

- **`getEmbedding`**: Metodo per ottenere gli embedding da Ollama per un determinato testo, specificando il modello. Restituisce un array di numeri.

Campi

- **`client`**: Oggetto di tipo `AxiosInstance` che rappresenterà l'API rest di Ollama.
- **`embeddingsClient`**: Oggetto di tipo `AxiosInstance` che rappresenterà l'API rest di Ollama utilizzata esclusivamente per gli embeddings.
- **`configPort`**: Oggetto di tipo `ConfigPort` iniettato tramite *dependency injection*, che servirà per ottenere i parametri di configurazione del server API necessari per utilizzarli nella comunicazione con Ollama

6.2.5.2.2 Classe: `ConfigAdapter`

Implementa l'interfaccia `CONFIG_PORT` e fornisce un accesso centralizzato alle configurazioni dell'applicazione. Questo *adapter* può leggere configurazioni da file, variabili d'ambiente o altri sistemi di configurazione.

Metodi

- **`getConfig`**: Metodo per recuperare la configurazione del server API. Restituisce un oggetto di tipo `ConfigModel` che rappresenta le configurazioni necessarie per l'analisi dei requisiti e la comunicazione con Ollama.

6.2.5.2.3 Classe: `JsonParserAdapter`

Implementa l'interfaccia `JSON_PARSER_PORT` e offre un parsing JSON sicuro e affidabile. Si occupa di serializzare e deserializzare i dati JSON, garantendo la gestione di errori come formati non validi o dati mancanti.

Metodi

- **`parseJson`**: Metodo per deserializzare una stringa JSON in un oggetto.

6.2.6 Application

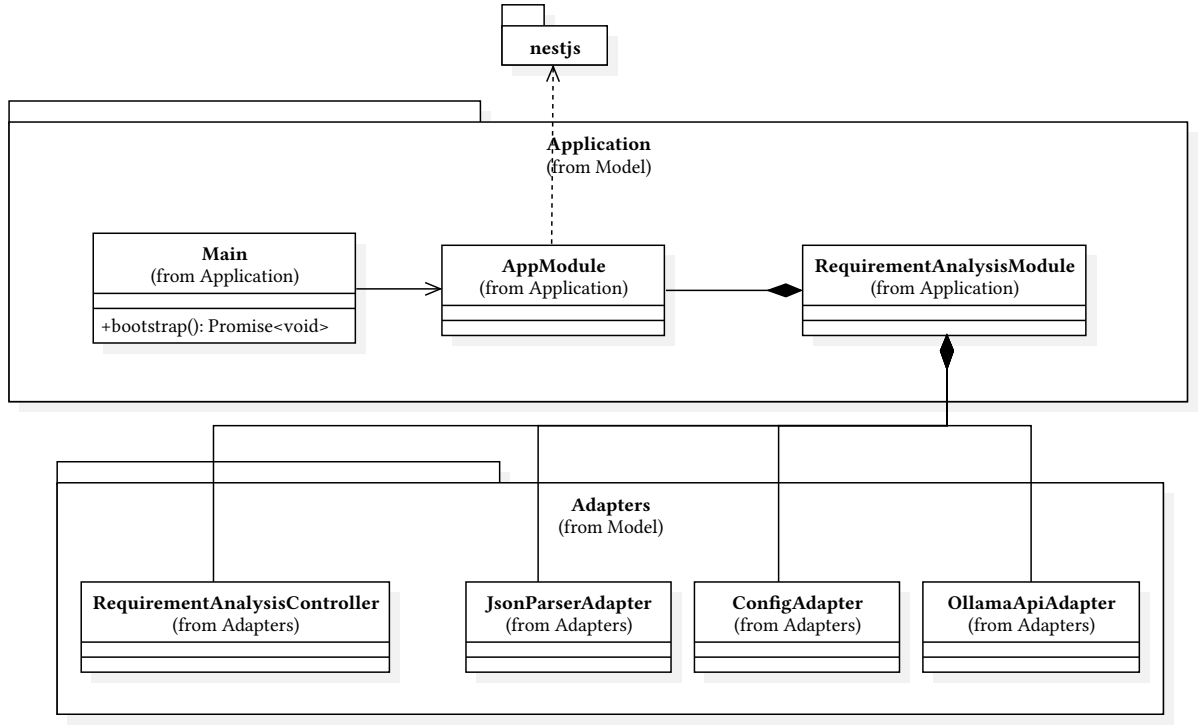

Figure 9: Diagramma architetturale dell'Application

L'*Application layer* si occupa di assemblare le dipendenze e configurare i componenti tramite *dependency injection*. Questo layer è responsabile di garantire che tutti i componenti siano correttamente inizializzati e pronti per l'uso.

6.2.6.1 Classe: `Main`

Questo è il punto di ingresso dell'applicazione. Si occupa di avviare l'applicazione e gestire il ciclo di vita dei componenti principali.

Metodi

- **`bootstrap`**: Metodo per avviare l'applicazione e inizializzare i componenti principali.

6.2.6.2 Modulo: `AppModule`

È il modulo principale dell'applicazione e funge da punto di ingresso per l'intero sistema. Configura il modulo `RequirementAnalysisModule` in modo da far registrare i *controller*, i *servizi* e gli `adapter` necessari per il funzionamento del backend.

6.2.6.3 Modulo: `RequirementAnalysisModule`

Modulo specifico per la gestione dell'analisi dei requisiti. Contiene la configurazione dei *servizi*, dei *ports* e degli *adapter* relativi all'analisi dei requisiti. Questo modulo garantisce che tutte le dipendenze necessarie per l'analisi siano correttamente iniettate e configurate.

6.2.7 Utility

6.2.7.1 Gestione degli errori

La gestione degli errori è un aspetto cruciale per garantire la robustezza del backend. Le classi per la gestione degli errori forniscono un meccanismo strutturato per identificare e gestire i problemi che possono verificarsi durante l'esecuzione. Queste includono:

- **`DomainError`**: Classe base per tutti gli errori del dominio. Estende la classe `Error` di JavaScript e aggiunge un contesto specifico per gli errori legati alla logica di business.
- **`ParseError`**: Estende `DomainError` e rappresenta un errore specifico che si verifica durante il parsing dei dati, come JSON non valido.
- **`ExternalServiceError`**: Estende `DomainError` e rappresenta un errore che si verifica durante la comunicazione con servizi esterni, come l'API Ollama.

6.2.7.2 Prompt Templates

I modelli di prompt sono utilizzati per generare input strutturati per l'API Ollama. Questi modelli includono:

- **`requirementContext`**: Contesto predefinito per l'analisi dei requisiti.
- **`codeContext`**: Contesto predefinito per l'analisi del codice.
- **`codePromptTemplate`**: Funzione che genera un prompt personalizzato combinando i requisiti e il codice da analizzare. Questo approccio consente di standardizzare e ottimizzare le richieste inviate all'API.