PROGETTO “JUNIT-TO-JMH”

Impact Analysis Document

Nome	Matricola
Benito Pigna	NF22500110
Michele Cillo	NF22500124

Sommario

INTRODUZIONE

Questo documento di Impact Analysis ha lo scopo di esaminare e valutare le conseguenze delle Change Request (CR) introdotte nel progetto junit-to-jmh. L'intento principale è guidare le scelte di refactoring e sviluppo, minimizzando la possibilità di introdurre effetti collaterali indesiderati. Per raggiungere questo scopo, è stata eseguita un'analisi statica del codice sorgente. Questo approccio permette di mappare l'architettura del software suddividendola in moduli, servizi e classi, mettendo in luce le dipendenze che determinano come una singola modifica si propaga all'interno del sistema. Lo strumento principale scelto per illustrare visivamente queste interazioni è il *call graph* . Questo artefatto è essenziale in quanto consente di:

1 Evidenziare in modo chiaro i flussi di invocazione e il grado di accoppiamento tra i vari componenti.

2 Identificare i nodi nevralgici dell'architettura e i punti di orchestrazione.

3 Ricavare metodicamente gli insiemi di impatto (SIS e CIS), riducendo al minimo il rischio di trascurare file coinvolti.

4 Giustificare le decisioni implementative mettendo a confronto l'architettura pianificata con quella realmente sviluppata

Per garantire un'ottima leggibilità, i call graph presentati in questo documento sono stati calibrati su un livello di astrazione adeguato alla specifica CR trattata (ad esempio, limitandosi alle relazioni tra servizi o classi principali), tralasciando dettagli a grana troppo fine che rischierebbero di appesantire inutilmente l'analisi.

A supporto dei call graph, l'analisi si avvale di una matrice di raggiungibilità applicata ai componenti chiave. I valori all'interno delle celle (i, j) indicano il tipo di relazione intercorrente:

- **0**: Assenza totale di dipendenza tra il componente *i* e il componente *j*.
- **1**: Dipendenza diretta.
- **2+**: Dipendenza indiretta, mediata da uno o più componenti intermedi.

L'utilizzo combinato del grafo e della matrice è fondamentale per quantificare quanto "in profondità" si spingono le chiamate tra i moduli, offrendo una stima precisa dell'onda d'urto causata dalle future modifiche.

1.1 Obiettivo

Gli scopi principali di questo documento si articolano nei seguenti punti:

1. **Mappatura architetturale:** Eseguire un'analisi statica del codice sorgente per delineare la struttura di **junit-to-jmh**. Questo studio, focalizzato sulle interazioni tra classi e servizi, mira a identificare le dipendenze cruciali che determinano la propagazione delle modifiche all'interno del sistema. L'analisi si avvale del parsing statico del codice Java e dell'utilizzo di tool per la generazione automatizzata dei call graph.

2. **Valutazione delle modifiche:** Definire e caratterizzare in modo dettagliato l'impatto specifico di ogni singola Change Request (CR).

3. **Tracciabilità delle stime:** Fornire una giustificazione empirica alle previsioni di impatto basandosi su artefatti strutturali oggettivi, ovvero i call graph e le matrici di raggiungibilità. Questo approccio garantisce un collegamento logico e completamente tracciabile tra le dipendenze osservate e i set di impatto generati.
4. **Misurazione dell'accuratezza:** Validare l'affidabilità dell'analisi preventiva attraverso metriche standard di valutazione, nello specifico Precision e Recall. Tali metriche vengono calcolate confrontando l'insieme dei componenti stimati come "potenzialmente impattati" con quelli che hanno effettivamente subito alterazioni in fase di implementazione, misurando così la reale capacità predittiva dello studio.

1.2 Descrizione delle Change Request

ID	Titolo	Descrizione Breve
CR01	Supporto JUnit 5	Aggiornamento del sistema per supportare, riconoscere e convertire le annotazioni introdotte da JUnit 5.
CR02	Integrazione AMBER	Aggiornamento delle dipendenze e configurazione del parser per elaborare correttamente i file contenenti i nuovi costrutti di AMBER (supporto al livello linguistico Java 17/21).
CR03	Controllo granularità	Introduzione di un meccanismo per consentire all'utente di specificare singolarmente i metodi di test da convertire, superando la limitazione della conversione dell'intera classe.

1.3 Definizioni e acronimi

- **SIS (Starting Impact Set):** Insieme delle componenti direttamente (o inizialmente) coinvolte dalla Change Request.
- **CIS (Candidate Impact Set):** Insieme delle componenti potenzialmente colpite a livello indiretto.
- **AIS (Actual Impact Set):** Insieme delle componenti realmente modificate durante l’implementazione.
- **FPIS (False Positive Impact Set):** Componenti inizialmente considerate a rischio ma che non subiscono modifiche.
- **DIS (Discovered Impact Set):** Componenti non previste inizialmente ma risultate impattate in corso d’opera

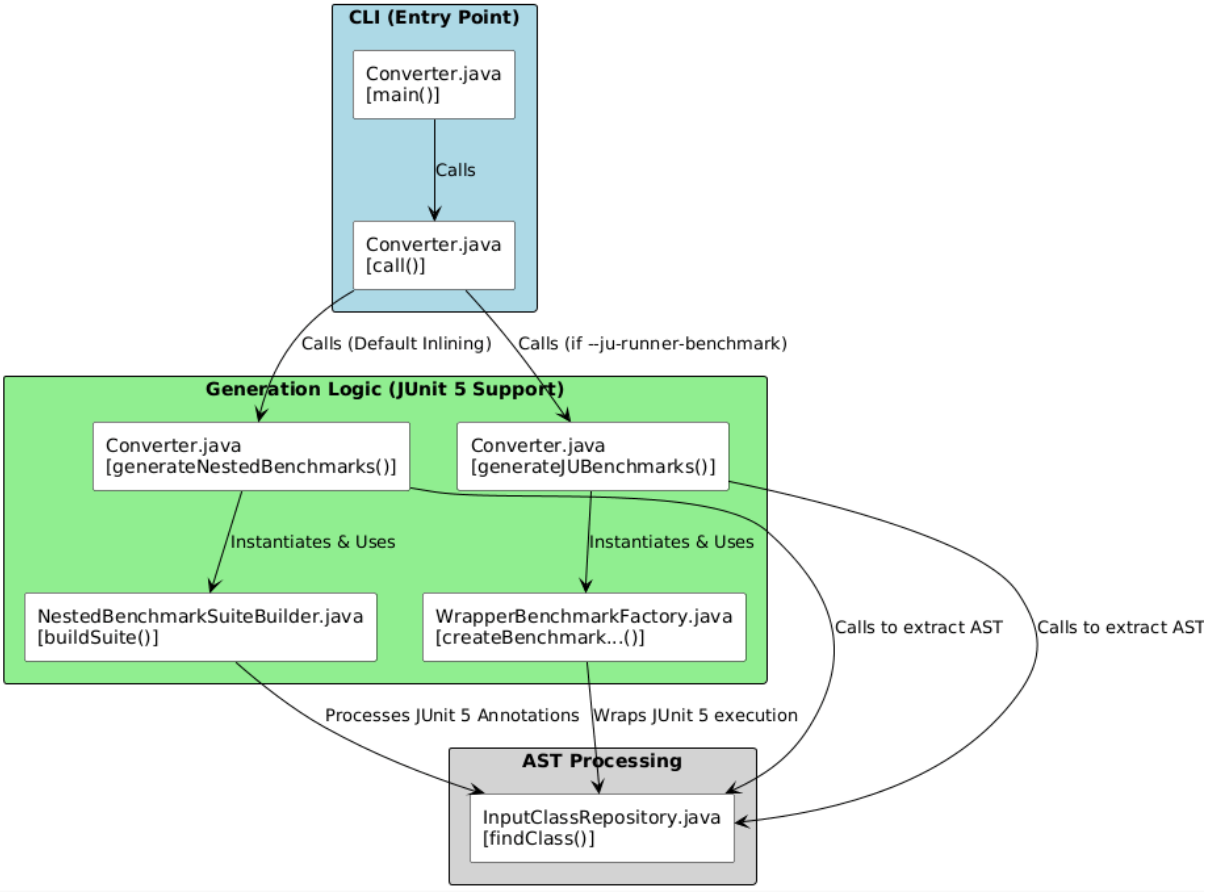
2 Change Request CR01

La presente Change Request introduce un'evoluzione fondamentale per il sistema *ju2jmh*, estendendone le capacità per supportare le suite di test scritte in JUnit 5 (Jupiter), lo standard attualmente predominante nello sviluppo Java.

- **Situazione attuale:** Il sistema originale era progettato con un forte accoppiamento verso JUnit 4. Il tool era in grado di analizzare e convertire esclusivamente classi basate su tale framework, ignorando le annotazioni moderne e il nuovo ciclo di vita dei test.
- **Situazione desiderata:** Estendere le capacità del convertitore affinché riconosca, analizzi e traduca correttamente le annotazioni introdotte da JUnit 5 (es. @Test, @BeforeEach, @AfterEach, @BeforeAll e @AfterAll). Inoltre, l'architettura deve essere arricchita per permettere di delegare l'esecuzione al *JUnit Platform Launcher* per i test più complessi, tramite l'introduzione di nuovi flag e factory dedicate. Oltre al controllo sui metodi, l'esigenza di eseguire generazioni mirate in momenti diversi ha fatto emergere il requisito implicito di dover gestire in modo sicuro i file di output, evitando sovrascritture distruttive dei benchmark generati in precedenza.

2.1 Analisi statica

Il seguente call graph architetturale illustra le relazioni tra le classi principali responsabili dell'individuazione delle classi di test e della generazione dei benchmark, riflettendo l'architettura necessaria per gestire il doppio flusso (JUnit 4 e JUnit 5).



A completamento del call graph, la seguente matrice di raggiungibilità rappresenta le dipendenze tra i componenti architetturali principali coinvolti nella CR01:

	Converter	**InputClass Repository**	**NestedBenchmark SuiteBuilder**	**WrapperBenchmark Factory**
Converter	0	1	2	1
InputClassReposito ry	-	0	1	0
NestedBenchmark SuiteBuilder	-	-	0	0
WrapperBenchmar kFactory	-	-	-	0

2.2 Impact Analysis

2.2.1 SIS (Start Impact Set)

I componenti direttamente coinvolti in questa CR, identificati inizialmente durante la fase di pianificazione (come riportato nel documento di Change Request), riguardano il modulo di parsing delle classi e il builder per la generazione dei benchmark. I file individuati per l'intervento diretto sono:

1. **se.chalmers.ju2jmh.InputClassRepository.java**
2. **se.chalmers.ju2jmh.NestedBenchmarkSuiteBuilder.java**

SIS = {InputClassRepository.java, NestedBenchmarkSuiteBuilder.java}

2.2.2 CIS (Candidate Impact Set)

Considerando le dipendenze strutturali, si è ipotizzato un potenziale impatto indiretto sul componente principale che orchestra il flusso e richiama il repository.

CIS = SIS ∪ {Converter.java}

2.3 Implentazione della Change Request

2.3.1 AIS (Actual Impact Set)

L'implementazione della CR ha richiesto un intervento più esteso del previsto. Oltre ad aggiornare il parsing per riconoscere l'AST di JUnit 5, è stato necessario introdurre nuove dipendenze, gestire un nuovo flag CLI (--ju-runner-benchmark) nel Converter e, soprattutto, creare una nuova factory per delegare l'esecuzione al JUnit Platform Launcher. I file effettivamente modificati sono:

1. **se.chalmers.ju2jmh.InputClassRepository.java**

2. **se.chalmers.ju2jmh.NestedBenchmarkSuiteBuilder.java**
3. **se.chalmers.ju2jmh.Converter.java**
4. **build.gradle.kts**
5. **se.chalmers.ju2jmh.WrapperBenchmarkFactory.java** (Componente rinominato)

AIS = {InputClassRepository.java, NestedBenchmarkSuiteBuilder.java, Converter.java, build.gradle.kts, WrapperBenchmarkFactory.java}

2.3.2 FPIS (False Positive Impact Set)

Non sono stati riscontrati file inizialmente previsti nel CIS che poi non hanno richiesto modifiche. Tutte le classi stimate hanno subito alterazioni.

FPIS = {∅}

2.3.3 DIS (Discovered Impact Set)

Durante l'implementazione sono emersi due componenti non considerati nell'analisi iniziale:

1. **build.gradle.kts:** Necessario per includere le nuove dipendenze a junit-jupiter-api e junit-jupiter-engine.
2. **WrapperBenchmarkFactory.java**: È stato necessario intervenire sulla vecchia factory (rinominandola e alterandone la logica) per generare i benchmark come wrapper. Questo refactoring architetturale non era emerso durante l'analisi preliminare.

DIS = {build.gradle.kts, WrapperBenchmarkFactory.java}

2.4 Metriche

Per valutare la qualità dell'Impact Analysis preventiva consideriamo le metriche di Precision e Recall:

$$Precision = \frac{|CIS \cap AIS|}{|CIS|} = \frac{3}{3} = 1.00$$

$$Recall = \frac{|CIS \cap AIS|}{|AIS|} = \frac{3}{5} = 0.60$$

Valutazione: L'analisi si è rivelata estremamente precisa (Precision = 1.00), in quanto tutti i componenti previsti nel CIS hanno effettivamente subito modifiche (zero falsi positivi). Tuttavia, ha sottostimato l'impatto reale (Recall = 0.60) poiché, durante l'implementazione pratica del supporto a JUnit 5, sono emerse dipendenze aggiuntive a livello di build system e la necessità architetturale di introdurre una nuova factory (DIS), fattori che hanno ampliato il raggio della modifica.

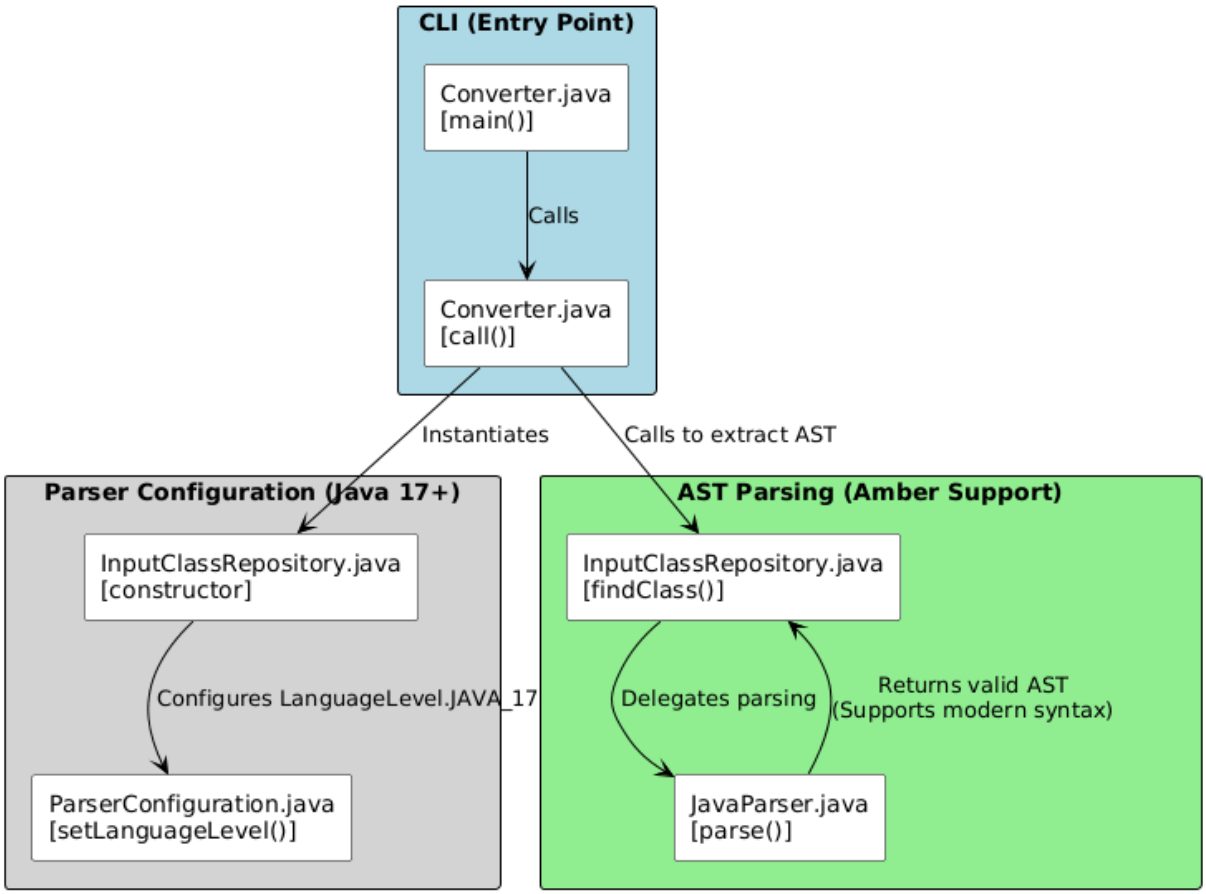
3 Change Request CR02

La presente Change Request affronta una necessità di manutenzione adattiva , volta a modernizzare l'infrastruttura del sistema *ju2jmh* per renderlo compatibile con i costrutti sintattici introdotti dal Project Amber di Java.

- **Situazione attuale:** Il sistema originale utilizzava una configurazione del parser e dipendenze obsolete. L'istanza interna di parsing non era configurata per elaborare i file sorgente contenenti le sintassi moderne di Java (livello 17/21), causando fallimenti nella generazione dell'AST in presenza di tali costrutti.
- **Situazione desiderata:** Aggiornare l'infrastruttura di build per supportare la toolchain di Java 17 e l'ultima versione della libreria *JavaParser*. A livello applicativo, riconfigurare l'istanza del parser interno impostando esplicitamente il LanguageLevel aggiornato per garantire la corretta generazione dell'AST. Infine, integrare una versione locale di JMH compatibile con il nuovo bytecode.

3.1 Analisi statica

Il seguente call graph architetturale illustra l'isolamento di questa modifica. A differenza della CR01, che ha alterato il flusso logico del programma, la CR02 agisce esclusivamente sul livello di configurazione dell'ambiente (Build System) e sull'inizializzazione del motore di parsing.



Di seguito la matrice di raggiungibilità per i componenti interni del progetto coinvolti:

	build.gradle.kts	**InputClassRepository**
build.gradle.kts	0	1
InputClassRepository	-	0

3.2 Impact Analysis

3.2.1 SIS (Start Impact Set)

L'analisi architetturale preventiva ha individuato due punti focali per questa manutenzione adattiva: il file di gestione del progetto e la classe responsabile della lettura dei file sorgente.

SIS = {build.gradle.kts, InputClassRepository.java}

3.2.2 CIS (Candidate Impact Set)

Trattandosi di un aggiornamento di dipendenze e di configurazione di un parser di terze parti (iniettato direttamente nel costruttore), non si sono previsti impatti a cascata sulle altre logiche di business o sul flusso del Converter. L'impatto stimato è rimasto circoscritto al SIS.

CIS = {build.gradle.kts, InputClassRepository.java}

3.2 Implentazione della Change Request

3.3.1 AIS (Actual Impact Set)

L'implementazione ha confermato in pieno l'analisi preventiva. Le modifiche hanno riguardato esclusivamente:

1. **build.gradle.kts:** Aggiornamento della toolchain a Java 17, della dipendenza javaparser-core e modifica della risoluzione (tramite mavenLocal()) per la versione aggiornata di JMH.
2. **InputClassRepository.java:** Refactoring del costruttore per istanziare un oggetto ParserConfiguration con LanguageLevel.JAVA_17, iniettato nella nuova istanza di JavaParser.

AIS = {build.gradle.kts, InputClassRepository.java}

3.3.2 FPIS (False Positive Impact Set)

Tutti i componenti previsti nel CIS hanno effettivamente richiesto un intervento.

FPIS = {∅}

3.3.3 DIS (Discovered Impact Set)

Non è emersa la necessità di modificare alcun componente strutturale imprevisto. L'aggiornamento del LanguageLevel ha risolto nativamente i problemi di parsing dell'AST per i livelli superiori di Java.

DIS = {∅}

3.4 Metriche

Per valutare la qualità dell'Impact Analysis preventiva consideriamo le metriche di Precision e Recall:

$$Precision = \frac{|CIS \cap AIS|}{|CIS|} = \frac{2}{2} = 1.00$$

$$Recall = \frac{|CIS \cap AIS|}{|AIS|} = \frac{2}{2} = 1.00$$

Valutazione: L'analisi per la CR02 si è rivelata perfetta. L'impatto è stato stimato in modo accurato e circoscritto, senza generare falsi positivi (Precision massima) e senza incontrare ramificazioni impreviste nel codice durante lo sviluppo (Recall massima). Questo evidenzia un basso livello di accoppiamento (low coupling) tra la logica di configurazione del parser e il resto del sistema.

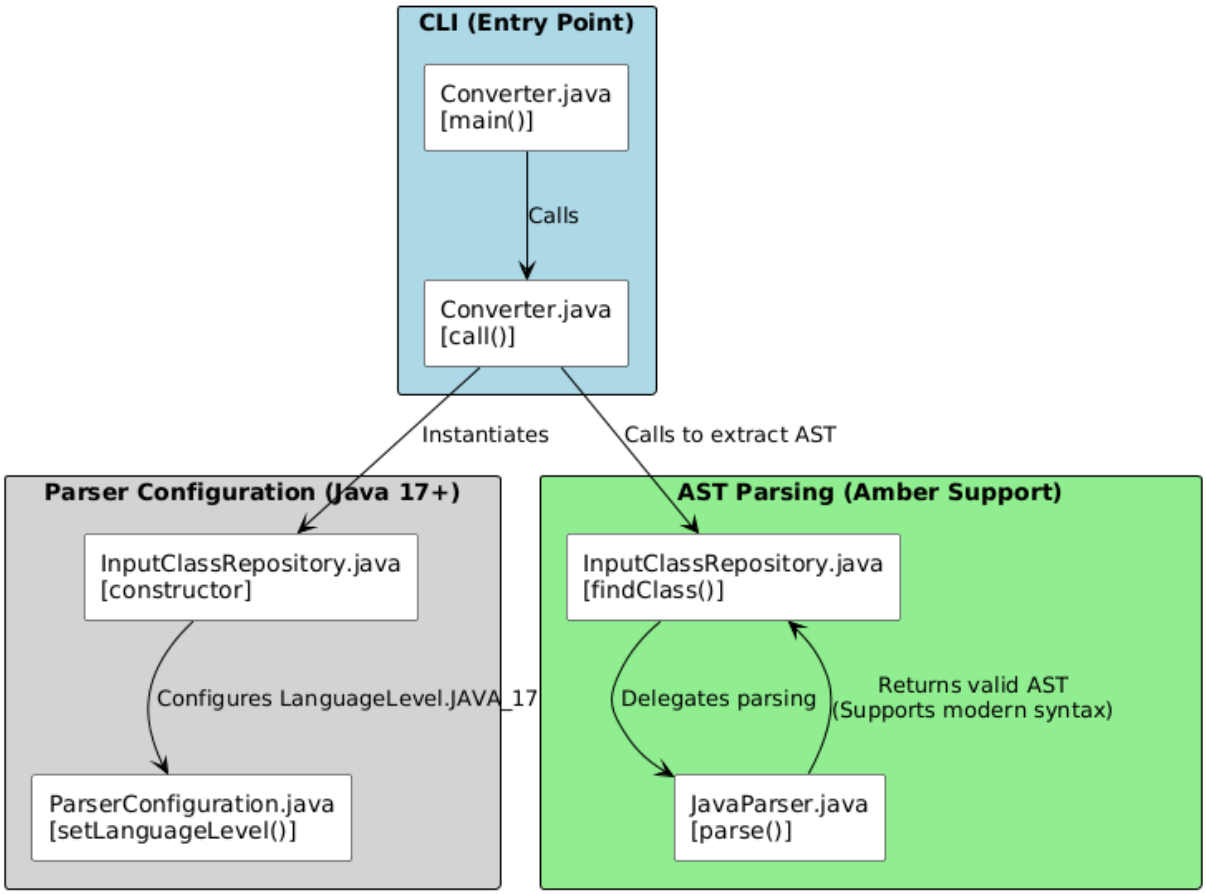
4 Change Request CR03

La presente Change Request introduce un intervento di manutenzione perfettiva, mirato a migliorare l'usabilità e la sicurezza del tool *ju2jmh* offrendo all'utente un controllo granulare sui benchmark generati.

- **Situazione attuale:** Il sistema era in grado di convertire unicamente l'intera classe di test JUnit. Non esisteva alcun meccanismo per isolare specifici metodi, obbligando l'utente a generare (e successivamente cancellare a mano) benchmark non necessari.
- **Situazione desiderata:** Introdurre un meccanismo di selezione dei metodi (es. tramite sintassi MyTest#testA,testB o flag dedicati). Durante l'analisi, il requisito è stato evoluto introducendo un pattern di **Validazione Fail-Fast**: prima di avviare la costosa generazione dei file, il sistema interroga l'AST per verificare l'effettiva esistenza dei metodi richiesti. In caso di errore (es. typo dell'utente), il processo si interrompe immediatamente, prevenendo la creazione di file parziali o corrotti.

4.1 Analisi statica

Il seguente call graph architetturale illustra il nuovo flusso logico. Il Converter assume un ruolo centrale non solo di instradamento, ma di validazione attiva sull'AST prima di delegare il lavoro alle factory di generazione.



Di seguito la matrice di raggiungibilità aggiornata per il nuovo flusso di validazione:

	Converter	InputClass Repository	NestedBenchmark SuiteBuilder	WrapperBenchmark Factory
Converter	0	1	1	1
InputClassRepository	-	0	0	0
NestedBenchmarkSuiteBuilder	-	-	0	0
WrapperBenchmarkFactory	-	-	-	0

4.2 Impact Analysis

4.2.1 SIS (Start Impact Set)

In base all'analisi preliminare riportata nel documento di Change Request, l'impatto stimato per l'introduzione del controllo di granularità era limitato esclusivamente al componente che gestisce l'input dell'utente.

SIS = {Converter.java}

4.2.2 CIS (Candidate Impact Set)

Si ipotizzava che la logica di filtraggio potesse essere risolta interamente a monte, isolando la modifica all'interno del Converter senza impattare le classi a valle.

CIS = {Converter.java}

4.3 Implentazione della Change Request

4.3.1 AIS (Actual Impact Set)

L'implementazione reale ha rivelato che la modifica non poteva essere confinata al solo entry point. L'introduzione della Validazione Fail-Fast ha richiesto l'estensione del Converter (per il parsing degli argomenti CLI tramite # o --methods e la navigazione dell'AST), ma ha anche costretto ad aggiornare le firme e la logica interna dei builder/factory affinché accettassero il set di metodi "filtrati", ignorando il resto della classe. Oltre alla Validazione Fail-Fast, all'interno del Converter.java è stato implementato un intero modulo logico non previsto inizialmente per la Smart Conflict Resolution. Per supportare la generazione iterativa introdotta da questa CR, il Converter ora gestisce un prompt CLI interattivo, clona automaticamente i file in copie di backup (.bak) e sfrutta l'AST per unire (Merge) i nuovi benchmark a quelli esistenti senza perdite di dati.

AIS = {Converter.java, NestedBenchmarkSuiteBuilder.java, WrapperBenchmarkFactory.java}

4.3.2 FPIS (False Positive Impact Set)

Il file inizialmente previsto ha effettivamente subito le modifiche pianificate.

FPIS = {∅}

4.3.3 DIS (Discovered Impact Set)

Durante lo sviluppo sono emersi due componenti aggiuntivi che hanno necessitato di interventi strutturali imprevisti per supportare il passaggio dei metodi filtrati.

DIS = {NestedBenchmarkSuiteBuilder.java, WrapperBenchmarkFactory.java}

4.4 Metriche

Per valutare la qualità dell'Impact Analysis preventiva consideriamo le metriche di Precision e Recall:

$$Precision = \frac{|CIS \cap AIS|}{|CIS|} = \frac{1}{1} = 1.00$$

$$Recall = \frac{|CIS \cap AIS|}{|AIS|} = \frac{1}{3} = 0.33$$

Valutazione: Le metriche illustrano perfettamente la natura di questa implementazione. L'analisi preliminare è stata precisa (Precision = 1.00) nell'individuare il Converter come punto di partenza. Tuttavia, la Recall estremamente bassa (≈ 0.33) evidenzia una **forte sottostima dell'impatto architetturale**. L'evoluzione del requisito verso un pattern di Validazione Fail-Fast ha causato un "effetto a cascata" (ripple effect) imprevisto sulle factory a valle, dimostrando un accoppiamento nei contratti delle interfacce maggiore di quanto ipotizzato inizialmente.