UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

POST-MODIFICATION TEST DOCUMENT

Ingegneria del Software: Tecniche Avanzate - Post-Modification Test Document

Luigi Auriemma

Matricola: NF22500161

Ivan Chiello

Matricola: NF22500167

Repository: `https://github.com/LuigiAuriemma/smell_ai.git`

Anno Accademico 2025-2026


sesa
lab
SOFTWARE ENGINEERING
SALERNO

Indice

Elenco delle Figure **iv**

Elenco delle Tabelle **v**

1 Introduzione **1**

2 Regression Testing **2**

2.1 Regression Testing per la Change Request 01 (CR-01) 2

2.1.1 Metodologia Operativa 2

2.1.2 Caso 1: Regressione nei Test di Integrazione 3

2.1.3 Caso 2: Regressione nei Unit Test della CLI 4

2.1.4 Analisi di Impatto su Project Analyzer 5

2.2 Regression Testing per la Change Request 02 (CR-02) 5

2.2.1 Strategia di Regressione 5

2.2.2 Esecuzione Suite Frontend 5

2.2.3 Analisi Backend e Gateway 6

2.2.4 Conclusione 6

3 Copertura **7**

3.1 Metodologia 7

3.1.1 Copertura Backend 7

3.1.2 Copertura Frontend 8
3.1.3 Risultati 8
3.2 Analisi 8

4 Unit Testing **10**
4.1 Unit Testing per la Change Request 01 (CR-01) 10
4.1.1 Tecnologie e Framework 10
4.1.2 Logica Core: CallGraphExtractor 11
4.1.3 Modifiche ai test esistenti 11
4.1.4 Interfaccia CLI 12
4.2 Unit Testing per la Change Request 02 (CR-02) 12
4.2.1 Stack Tecnologico di Testing 12
4.2.2 Backend Testing (FastAPI) 13
4.2.3 Frontend Testing (React) 13
4.2.4 Gateway Testing (Proxy) 14

5 Integretion Testing **15**
5.1 Integration Testing per la Change Request 01 (CR-01) 15
5.1.1 Strategia di Testing 15
5.1.2 Suite di Test: `test_cli_call_graph.py` 16
5.2 Integration Testing per la Change Request 02 (CR-02) 17
5.2.1 Strumenti Utilizzati 17
5.2.2 Componenti Sotto Test 18
5.2.3 Suite di Test 18
5.2.4 Risultati 19

6 System Testing **20**
6.1 Aggiornamento Categorie 20
6.1.1 Nuovo Parametro: Struttura delle Dipendenze (SD) 20
6.1.2 Nuovo Parametro: Configurazione Call Graph (CG) 21
6.2 Nuovi Test Case (TC) 21

7 WebApp Testing (End-to-End) 23

7.1 Strategia di Testing E2E 23

7.2 Casi di Test E2E Automatizzati 24

Elenco delle figure

Elenco delle tabelle

3.1 Metriche di Copertura del Codice per l'Intero Progetto 8

6.1 Nuovo Parametro: Struttura delle Dipendenze 21
6.2 Nuovo Parametro di Configurazione: Call Graph 21

CAPITOLO 1

Introduzione

CAPITOLO 2

Regression Testing

2.1 Regression Testing per la Change Request 01 (CR-01)

Durante l'implementazione della CR-01, è stata eseguita un'analisi approfondita per identificare e risolvere le regressioni introdotte dalle modifiche alle interfacce dei componenti core.

2.1.1 Metodologia Operativa

Il processo di regression testing è stato condotto seguendo un approccio iterativo per garantire che le nuove funzionalità non compromettessero la stabilità del sistema esistente. Il flusso di lavoro seguito è stato il seguente:

1. **Implementazione delle Modifiche**: Sono state applicate le modifiche al codice di produzione necessarie per la CR-01.
2. **Esecuzione della Suite di Test**: È stata eseguita l'intera suite di test esistente.
3. **Identificazione dei Fallimenti**: L'esecuzione ha evidenziato immediati fallimenti in test che precedentemente passavano.

4. **Analisi delle Cause**: Analizzando i messaggi di errore è stato determinato che la causa radice risiedeva nella mancata corrispondenza tra i Mock configurati nei test e le nuove firme dei metodi aggiornati.

5. **Risoluzione**: I test sono stati refattorizzati per allineare i Mock e le asserzioni alla nuova logica, senza modificare il comportamento del sistema.

6. **Verifica Finale**: Una ri-esecuzione completa della suite ha confermato la risoluzione delle regressioni.

2.1.2 Caso 1: Regressione nei Test di Integrazione

File: `test/integration_testing/test_cli_to_analyzer.py`
Questo test verifica l'integrazione tra il runner CLI e l'analizzatore di progetto utilizzando dei Mock per simulare gli argomenti della riga di comando.

Il Problema

Il test falliva riportando un'invocazione inattesa:

```
E   AssertionError: expected call not found.
E   Expected: analyze_project('/fake/input')
E   Actual: analyze_project('/fake/input',
            generate_graph=<Mock name...>)
```

Causa: La nuova logica della CLI recupera sempre l'attributo `call_graph` dagli argomenti per passarlo al metodo `analyze_project`. Il Mock degli argomenti esistente non definiva questo attributo, causando la creazione automatica di un MagicMock (valutato come True/Oggetto) che veniva passato erroneamente al metodo, violando l'asserzione stretta.

Soluzione Implementata

È stato aggiornato il setup del Mock nel test per definire esplicitamente `call_graph=False`, e aggiornata l'asserzione per verificare che il parametro `generate_graph=False` sia passato correttamente.

2.1.3 Caso 2: Regressione nei Unit Test della CLI

File: `test/unit_testing/cli/test_cli_runner.py`
Questa classe di test verifica isolatamente che `CodeSmileCLI` invochi i metodi corretti di `ProjectAnalyzer`.

Il Problema

I test unitari utilizzano `assert_called_once_with` per verificare l'esattezza delle chiamate. Con l'aggiornamento del metodo `execute`, la CLI ha iniziato a passare il parametro `generate_graph` in tutte le chiamate, rompendo le asserzioni esistenti che non prevedevano questo argomento.
Esempio di test impattati:

- `test_execute_with_valid_arguments`
- `test_execute_with_sequential_execution`
- `test_execute_with_parallel_execution`

Soluzione Implementata

I test sono stati aggiornati per accettare la nuova firma delle chiamate. È sato utilizzato il matcher `ANY` per mantenere i test flessibili riguardo al valore specifico del grafo ove non rilevante per il test specifico, oppure aggiornata la firma attesa.

```
# Prima
mock_analyzer.analyze_project.assert_called_once_with("mock_input")
# Dopo (Fix)
mock_analyzer.analyze_project.assert_called_once_with(
    "mock_input",
    generate_graph=ANY
)
```

2.1.4 Analisi di Impatto su Project Analyzer

File: `test/unit_testing/components/test_project_analyzer.py`
Sebbene questo file sia stato modificato per aggiungere nuovi test, non ha subito regressioni funzionali. Questo è dovuto all'uso di **parametri con valori di default** (`generate_graph=False`) nella firma aggiornata di `analyze_project`.
Questa scelta progettuale ha garantito la *Backward Compatibility*, permettendo a tutte le chiamate esistenti nei test di continuare a funzionare senza modifiche.

2.2 Regression Testing per la Change Request 02 (CR-02)

A differenza della CR-01, che modificava componenti core esistenti (CLI e ProjectAnalyzer), la CR-02 è stata implementata principalmente come un'estensione additiva (nuovi componenti Frontend e nuovi Endpoint Backend). Tuttavia, è stato eseguito il regression testing per garantire che l'integrazione dei nuovi moduli non abbia introdotto effetti collaterali.

2.2.1 Strategia di Regressione

La strategia si è concentrata su due livelli:

1. **Frontend Regression**: Verifica che l'introduzione delle nuove pagine e dipendenze grafiche non abbia rotto le pagine esistenti (Upload, Report).
2. **Backend Isolation**: Verifica che l'aggiunta del nuovo router `call_graph` nel Gateway e nel Service non abbia alterato gli endpoint esistenti.

2.2.2 Esecuzione Suite Frontend

È stata eseguita l'intera suite di test del frontend (`webapp/_tests_`) utilizzando `npm test`.
Risultati:

- **Test Suites**: 7 passati su 7.
- **Total Tests**: 29 passati su 29.

Aree verificate:

- `UploadProjectPage` e `UploadPythonPage`: Il flusso di caricamento file è rimasto inalterato.
- `ReportsPage`: La visualizzazione dei report precedenti continua a funzionare nonostante l'aggiornamento delle librerie condivise.
- `ProjectContext`: La gestione dello stato globale dell'applicazione non ha subito regressioni.

L'assenza di fallimenti conferma che le modifiche al `package.json` (aggiunta di librerie per i grafi) e alla struttura delle cartelle non hanno impattato le funzionalità legacy.

2.2.3 Analisi Backend e Gateway

Nel backend, le modifiche sono state strettamente confinate:

- **Service Layer**: Il nuovo modulo `routers/call_graph.py` è isolato. L'unico punto di contatto è l'inclusione del router in `main.py`, che è stata verificata tramite i test di avvio e unitari.
- **Gateway Layer**: L'aggiunta dell'endpoint proxy `/api/generate_call_graph` non ha modificato le rotte esistenti (`detect_smell_ai`, `detect_smell_static`).

2.2.4 Conclusione

Il Regression Testing per la CR-02 ha dato esito positivo al primo passaggio sui test esistenti, confermando l'efficacia dell'architettura modulare che ha permesso di "innestare" la nuova funzionalità senza destabilizzare il sistema.

CAPITOLO 3

Copertura

In questo capitolo vengono presentati i risultati della copertura del codice (code coverage) per i componenti Backend (Python) e Frontend (React/Next.js) dell'intero progetto.

3.1 Metodologia

Per garantire una valutazione completa dell'efficacia della suite di test, sono state misurate sia la *Statement Coverage* (Copertura delle Istruzioni) che la *Branch Coverage* (Copertura dei Rami).

3.1.1 Copertura Backend

La copertura del backend è stata calcolata utilizzando `pytest` combinato con il plugin `pytest-cov`. Lo strumento è stato configurato per misurare la copertura dei rami al fine di identificare i percorsi di esecuzione che potrebbero essere sfuggiti a una semplice verifica delle linee.

- **Strumento**: `pytest`, `pytest-cov`, `coverage.py`
- **Comando**:

```
pytest --cov=. --cov-branch --cov-report=json
```

- **Configurazione**: L'analisi ha incluso tutti i moduli di estrazione, le regole di rilevamento e il runner CLI, escludendo i file di configurazione e i test stessi.

3.1.2 Copertura Frontend

La copertura del frontend è stata calcolata utilizzando il framework di test `Jest`, che possiede funzionalità integrate di code coverage basate su `Istanbul`.

- **Strumento**: `Jest`
- **Comando**:

```
npm test -- --coverage
```

- **Configurazione**: Focalizzata sui test unitari e di integrazione all'interno della directory `webapp`.

3.1.3 Risultati

La tabella seguente riassume le metriche di copertura ottenute.

Componente	Statement Coverage	Branch Coverage
Backend	84.70%	79.53%
Frontend	85.75%	73.73%

Tabella 3.1: Metriche di Copertura del Codice per l'Intero Progetto

3.2 Analisi

Backend: Il backend ha raggiunto un buon grado di copertura complessiva (84.70% Statement, 79.53% Branch). I componenti core, inclusi gli estrattori e il costruttore

del grafo delle dipendenze, mantengono livelli di copertura elevati, assicurando la stabilità della logica di business principale del progetto.
Frontend: La copertura del frontend si attesta al 85.75% per le istruzioni. È importante notare che, inizialmente, a seguito dell'implementazione delle funzionalità previste dal Change Request 02 (CR-02), la copertura dei rami (Branch Coverage) si attestava intorno al 58%. Questo valore è stato ritenuto insufficiente per garantire la robustezza richiesta.
Di conseguenza, sono stati introdotti test mirati per colmare questo divario:

1. **Gestione Caricamento Progetti**: Sono stati aggiunti scenari di test aggiuntivi per il componente `UploadProjectPage` per coprire edge case specifici come errori API, liste di file vuote e gestione delle eccezioni durante l'upload, che non erano precedentemente verificati dai path "felici".

2. **Node Details Modal**: È stata creata una suite di test dedicata per il componente `NodeDetailsModal`, che in precedenza aveva una copertura minima. I nuovi test verificano il rendering condizionale in presenza o assenza di code smells e la visualizzazione corretta del codice sorgente.

Grazie a questi interventi, la Branch Coverage finale ha raggiunto il 73.73%, superando la soglia obiettivo del 70%.

CAPITOLO 4

Unit Testing

4.1 Unit Testing per la Change Request 01 (CR-01)

Questa sezione descrive gli aggiornamenti di testing e la copertura implementata per la **Change Request 01 (CR-01): Implementazione del call graph per la versione CLI**. L'obiettivo della CR-01 era introdurre le funzionalità di generazione del call graph nella versione CLI di CodeSmile. Per garantire l'affidabilità e prevenire regressioni, abbiamo implementato una serie completa di unit test.

4.1.1 Tecnologie e Framework

Per lo sviluppo della suite di test sono stati impiegati i seguenti strumenti e librerie:

- **Pytest**: Framework di testing primario, utilizzato per la definizione dei casi di test e la gestione delle fixture.
- **Unittest.mock**: Utilizzato per la creazione di *mock objects* e stub, essenziali per isolare le unità di codice (es. file system, builder del grafo) durante i test.
- **AST**: Il modulo `ast` nativo di Python è stato utilizzato per validare la logica di parsing sintattico del `CallGraphExtractor`.

4.1.2 Logica Core: CallGraphExtractor

File: `test/unit_testing/code_extractor/test_call_graph_extractor.py`
Il `CallGraphExtractor` è responsabile del parsing degli AST Python e dell'identificazione delle definizioni di funzioni e delle chiamate. Abbiamo creato una suite di test dedicata per verificare questa logica in isolamento.

- **`test_extract_simple_function`**: Verifica che le definizioni di funzioni standard siano identificate correttamente (nome, numeri di riga, tipo).
- **`test_extract_async_function`**: Assicura che le definizioni `async def` siano gestite correttamente con il tipo `ASYNC_FUNCTION`.
- **`test_extract_calls_within_function`**: Verifica specifica che le chiamate di funzione all'interno del corpo di una funzione siano catturate con la corretta relazione chiamante/chiamato.
- **`test_extract_calls_global_scope`**: Verifica che le chiamate effettuate a livello di modulo (nessuna funzione genitore) siano catturate con un chiamante `None`.
- **`test_extract_method_calls`**: Controlla l'estrazione delle chiamate ai metodi (es. `obj.method()`) dove il nome della funzione è un attributo.
- **`test_nested_functions`**: Verifica che le funzioni interne/nidificate siano identificate correttamente e che le chiamate al loro interno siano attribuite allo scope corretto.
- **`test_extract_call_name_edge_cases`**: Assicura che l'extractor gestisca i casi limite delle chiamate (come le lambda) senza arresti anomali.

4.1.3 Modifiche ai test esistenti

File: `test/unit_testing/components/test_project_analyzer.py` (Aggiornato)

Il `ProjectAnalyzer` è l'orchestratore principale. Abbiamo aggiornato i suoi test per verificare che utilizzi correttamente il `DependencyGraphBuilder` quando il flag di generazione del grafo è impostato.

- **`test_analyze_project_with_graph_generation`**: Un nuovo test case che simula (mock) il `DependencyGraphBuilder`. Asserisce che quando viene passato `generate_graph=True` a `analyze_project`, il builder viene inizializzato e `build_graph()` viene chiamato con la lista di file corretta.

4.1.4 Interfaccia CLI

File: `test/unit_testing/cli/test_cli_runner.py` (Aggiornato)
Il punto di ingresso per l'utente è la CLI. Dovevamo assicurarci che l'argomento della riga di comando `-call-graph` fosse analizzato correttamente e propagato all'analizzatore.

- **`test_execute_call_graph_flag`**: Un nuovo test case che simula l'esecuzione della CLI con `-call-graph`. Verifica che `analyzer.analyze_project` venga successivamente chiamato con `generate_graph=True`.

4.2 Unit Testing per la Change Request 02 (CR-02)

La CR-02 introduce la visualizzazione del grafo delle chiamate (Call Graph) nella Webapp. La strategia di testing unitario ha coperto tutti i livelli dell'architettura: Backend (FastAPI), Gateway (Proxy) e Frontend (React).

4.2.1 Stack Tecnologico di Testing

Data la natura ibrida dell'applicazione, sono stati adottati strumenti specifici per ogni livello architetturale:

- **Pytest e Unittest.Mock**: Utilizzati per testare i servizi Backend (FastAPI) e il Gateway. La scelta è dettata dalla necessità di validare la logica complessa di analisi AST e manipolazione dei grafi (NetworkX) scritta in Python, isolando le dipendenze esterne.

- **Jest e React Testing Library**: Utilizzati per il Frontend. Questi strumenti permettono di verificare il rendering dei componenti React, la gestione dello stato e le interazioni utente in un ambiente DOM simulato (JSDOM), essenziale per validare la UI del Call Graph.

4.2.2 Backend Testing (FastAPI)

File: `test/unit_testing/webapp/test_call_graph_route.py`
Il backend espone l'endpoint `/generate_call_graph`. I test unitari utilizzano `unittest.mock` per isolare la logica del router dalle dipendenze esterne (`Inspector` per gli smell e `DependencyGraphBuilder` per il grafo).

- **test_generate_call_graph_success**: Verifica che, dato un payload valido, l'endpoint restituisca un JSON strutturato correttamente secondo lo schema `CallGraphResponse`
  e che i metodi dei componenti sottostanti vengano invocati.

- **test_generate_call_graph_with_smells**: Verifica l'integrazione logica (mockata) tra i risultati dell'analisi degli smell e la costruzione del grafo. Assicura che, se l'Inspector restituisce degli smell, questi vengano correttamente arricchiti nei nodi del grafo ("merging logic").

- **test_generate_call_graph_error_handling**: Simula eccezioni nei componenti core per garantire che l'API restituisca un messaggio di errore gestito gracefuly e non un 500 generico non controllato.

4.2.3 Frontend Testing (React)

File: `webapp/_tests_/components/CallGraphVisualizer.test.tsx`
Il componente `CallGraphVisualizer` è stato testato utilizzando `Jest` e `React Testing Library`. Poiché il componente utilizza `react-plotly.js` (che dipende dal DOM del browser non pienamente supportato da JSDOM), è stato utilizzato un Mock del componente grafico.

- **Gestione Stati Vuoti**: Verifica che venga mostrato il messaggio "No graph data available" quando i dati sono `null` o vuoti.

- **Rendering del Grafo**: Verifica che, in presenza di dati validi, il componente `Plot` (mockato) venga renderizzato nel DOM.

4.2.4 Gateway Testing (Proxy)

File: `test/unit_testing/webapp/test_gateway.py`
L'API Gateway agisce da proxy verso il servizio di analisi statica. Il test unitario verifica che la richiesta venga inoltrata correttamente.

- **test_generate_call_graph_proxy**: Utilizza `AsyncMock` per intercettare la chiamata `httpx.AsyncClient.post`. Verifica che il Gateway inoltri la richiesta all'URL corretto del microservizio (`STATIC_ANALYSIS_SERVICE`) e restituisca la risposta al client.

CAPITOLO 5

Integretion Testing

5.1 Integration Testing per la Change Request 01 (CR-01)

Questa sezione documenta specificamente le attività di Integration Testing svolte per validare la **Change Request 01 (CR-01): Implementazione del call graph per la versione CLI**.
Mentre lo unit testing isola i singoli componenti, l'integration testing verifica che questi componenti collaborino correttamente tra loro e producano l'output atteso in uno scenario quasi-reale.

5.1.1 Strategia di Testing

Per i test di integrazione, abbiamo adottato una strategia *Black-box* parziale:

- **Ambiente Reale**: I test utilizzano il file system reale (tramite directory temporanee isolate) invece di mockare l'I/O dei file. Questo garantisce che il parser legga i file correttamente dal disco.

- **Esecuzione End-to-End**: Viene invocata la classe `CodeSmileCLI` esattamente come farebbe un utente da riga di comando, passando gli argomenti parserizzati.
- **Validazione dell'Output**: Non ispezioniamo lo stato interno degli oggetti durante l'esecuzione, ma verifichiamo esclusivamente l'artefatto prodotto in output (`call_graph.json`).

5.1.2 Suite di Test: `test_cli_call_graph.py`

È stato creato un nuovo file di test dedicato in `test/integration_testing/test_cli_cal`
Questa suite copre due scenari fondamentali:

1. Scenario Singolo File (Intra-module Dependency)

Test Case: `test_cli_call_graph_integration`
Obiettivo: Verificare che il sistema generi un grafo corretto per le chiamate di funzione all'interno dello stesso file.
Setup: Viene creato dinamicamente un file `main.py` contenente:

```
def func_a():
    func_b()


def func_b():
    print("hello")
```

Verifica: Il test asserisce che:

- Il file JSON venga creato nella cartella di output specificata.
- Il JSON contenga una lista di nodi e una lista di archi (links).
- Esistano i nodi `main.py::func_a` e `main.py::func_b`.
- Esista un arco diretto da `func_a` a `func_b`.

2. Scenario Multi-File (Inter-module Dependency)

Test Case: `test_cli_call_graph_multifile`
Obiettivo: Verificare che il sistema risolva correttamente le dipendenze e le chiamate tra file diversi del progetto, una caratteristica critica per la generazione di call graph utili.
Setup: Viene creata una struttura di progetto simulata:

- `utils.py`: Definisce una funzione `helper()`.
- `main.py`: Importa `utils` e chiama `utils.helper()`.

Verifica: Il test asserisce che:

- Entrambi i file vengano analizzati e i rispettivi nodi (`main.py::main` e `utils.py::helper`) siano presenti nel grafo.
- Venga creato correttamente un arco che collega il chiamante nel file `main.py` al chiamato nel file `utils.py`.

5.2 Integration Testing per la Change Request 02 (CR-02)

L'Integration Testing per la CR-02 ha l'obiettivo di verificare il corretto funzionamento della catena di esecuzione che parte dall'API Gateway e coinvolge i servizi di analisi statica e generazione grafi.

5.2.1 Strumenti Utilizzati

Per l'esecuzione dei test di integrazione sono stati impiegati i seguenti strumenti:

- **Pytest**: Framework di testing principale utilizzato come runner per l'intera suite.
- **FastAPI TestClient**: Client HTTP sincrono basato su `httpx`, utilizzato per simulare le chiamate REST verso il Gateway e i Microservizi senza la necessità

di avviare un server TCP reale. Questo permette di testare l'intera catena di middleware e routing in-process.

5.2.2 Componenti Sotto Test

I test di integrazione coinvolgono la cooperazione tra:

- **API Gateway**: Punto di ingresso REST (`/api/generate_call_graph`).
- **Static Analysis Service**: Servizio che orchestra l'analisi.
- **Inspector**: Componente che esegue la rilevazione degli smell.
- **DependencyGraphBuilder**: Componente che costruisce il grafo AST.

5.2.3 Suite di Test

File: `webapp/integration_tests/test_gateway_to_call_graph.py`

Test 1: Generazione del Grafo (Struttura)

Nome: `test_generate_call_graph_integration`
Scopo: Verificare che il sistema sia in grado di parsare un codice Python valido con chiamate di funzione annidate e restituire una struttura a grafo coerente.
Execution Flow:

1. Invio di un payload contenente due funzioni (`function_a` chiama `function_b`).
2. Verifica HTTP 200 OK.
3. Verifica presenza chiavi 'nodes' ed 'edges' nel JSON.
4. Verifica che i nodi 'function_a' e 'function_b' siano presenti.
5. Verifica che esista almeno un arco (edge) che collega i nodi.

Test 2: Integrazione Smell e Grafo (Merging)

Nome: `test_generate_call_graph_with_smells_integration`
Scopo: Verificare il requisito fondamentale della CR-02: visualizzare gli smell all'interno del call graph ("Smell-aware Graph").
Execution Flow:

1. Invio di un payload contenente codice noto per violare la regola `columns_and_datatype_not_explicitly_set` (viene usato Pandas DataFrame senza dtype).
2. Verifica HTTP 200 OK.
3. Ricerca del nodo associato alla funzione vulnerabile (`smelly_function`).
4. Asserzione sulla proprietà `has_smell=True`.
5. Asserzione che la lista `smell_details` contenga lo smell specifico atteso.

5.2.4 Risultati

Tutti i test di integrazione sono stati eseguiti con successo, confermando che il sistema interpola correttamente i dati provenienti dal motore di analisi statica con quelli strutturali del parser AST.

CAPITOLO 6

System Testing

A seguito dell'implementazione delle Change Request CR-01 (Generazione Call Graph Backend) e CR-02 (Visualizzazione Grafica Frontend), la strategia di System Testing è stata aggiornata. Oltre alla riesecuzione dei test esistenti (TC1-TC21) descritti nel documento "Pre_Modification_Test_Document", sono stati introdotti nuovi casi di test per coprire le nuove funzionalità di analisi strutturale.

6.1 Aggiornamento Categorie

Le categorie sono state estese per includere i parametri rilevanti per la generazione del grafo delle chiamate.
Oltre alle categorie già definite nel documento pre-modifica (relative ai Code Smells e ai parametri base di input/output), sono state aggiunte le seguenti categorie specifiche per la nuova funzionalità:

6.1.1 Nuovo Parametro: Struttura delle Dipendenze (SD)

Definisce la complessità topologica delle relazioni tra le funzioni nel codice analizzato.

Categoria	Codice	Scelte (Classi di Equivalenza)
Tipologia di Flusso (SD) Caratteristica strutturale dominante delle invocazioni nel codice target.	SD1	Chiamate Lineari/Annidate (Fan-In/Fan-Out)
	SD2	Ricorsione (Self-Loops)
	SD3	Strutture OOP (Metodi di classe/Istanza)
	SD4	Dipendenze Inter-modulari (Import)

Tabella 6.1: Nuovo Parametro: Struttura delle Dipendenze

6.1.2 Nuovo Parametro: Configurazione Call Graph (CG)

Estensione dei Parametri di Configurazione ed Esecuzione per la nuova funzionalità.

Categoria	Codice	Scelte (Classi di Equivalenza)
Call Graph Generation (CG) Attivazione della generazione ed esportazione del grafo delle dipendenze.	CG0	False (Default)
	CG1	True (Flag `-call-graph`)

Tabella 6.2: Nuovo Parametro di Configurazione: Call Graph

6.2 Nuovi Test Case (TC)

In base alle nuove categorie identificate, sono stati progettati e implementati i seguenti Test Case aggiuntivi.

TC ID	Test Frame (Parametri)	Oracolo / Risultato Atteso
TC_22	NF1, EF0, NP0, SP0, NCS0, TCS0, ME0, OUT1, **SD1, SD2, CG1**	**Backend Call Graph:** Verifica identificazione nodi in chiamate annidate e gestione cicli ricorsivi in `call_graph_complex.py`.
TC_23	NF1, EF0, NP0, SP0, NCS0, TCS0, ME0, OUT1, **SD3, CG1**	**OOP Structure:** Verifica cattura scope metodi di classe e chiamate interne `self` in `class_calls.py`.
TC_24	NF2, EF0, NP0, SP0, NCS0, TCS0, ME0, OUT1, **SD4, CG1**	**Modular Graph:** Verifica archi che attraversano confini dei file tramite import in `main_service.py`.
TC_25	NF1, EF0, NP0, SP0, NCS2, TCS2, ME0, OUT1, **SD2, SD3, CG1**	**Hybrid Analysis:** Generazione grafo su struttura OOP/Ricorsiva con rilevamento simultaneo di AI-Smells in `hybrid_smell_graph.py`.

Dettagli Implementativi dei Nuovi TC I file sorgente (Mock) per questi test sono stati strutturati come segue:

- `test/system_testing/TC22/MockDirectory/call_graph_complex.py`: Include `input_data()`, `process()` (chiamate annidate) e `recurse()` (ricorsione).
- `test/system_testing/TC23/MockDirectory/class_calls.py`: Include la classe `Processor` con metodi `load`, `validate`, `process_items`.
- `test/system_testing/TC24/Project1/MockDirectory`: Contiene `main_servic` che importa funzioni da `utils.py`.
- `test/system_testing/TC25/MockDirectory/hybrid_smell_graph.py`: Include classe `DataProcessor` che combina ricorsione e antipatterns Pandas (chain indexing, iterazione inefficiente).

CAPITOLO 7

WebApp Testing (End-to-End)

Parallelamente ai System Test, è stata implementata una suite di test End-to-End (E2E) automatizzati per validare l'esperienza utente sulla WebApp (CR-02).
La suite di test è stata sviluppata utilizzando il framework **Cypress**, che permette di simulare interazioni reali dell'utente (click, upload, navigazione) in un ambiente controllato.

7.1 Strategia di Testing E2E

I test E2E si concentrano sul flusso della nuova funzionalità "Call Graph Visualization", verificando l'integrazione completa tra i componenti Frontend (Next.js/React) e la risposta simulata del Backend.
La strategia adottata verifica:

- **UI Consistency**: Presenza e stato iniziale degli elementi (bottoni, dropzone).
- **Happy Path**: Flusso corretto di caricamento e visualizzazione.
- **Negative Testing**: Gestione di file non validi ed errori server.
- **Data Binding**: Corretta visualizzazione dei dettagli di Smell vs Clean code.

7.2 Casi di Test E2E Automatizzati

Di seguito sono riportati i Test Case implementati nel file `call-graph.spec.cy.ts`. Ogni test è identificato da un ID univoco E2E.

TC ID	Descrizione Scenario	Oracolo / Risultato Atteso
E2E-01	**UI Rendering Iniziale** Verifica dello stato della pagina al primo caricamento (`/call-graph`).	La Drop Zone è visibile; il bottone "Select File" è presente; il bottone "Generate Graph" è **disabilitato** (prevenzione click prematuri).
E2E-02	**Upload e Visualizzazione Grafo (Happy Path)** Caricamento di un file Python valido e click su Generate.	Il file viene accettato; il loader appare durante la richiesta; il grafo viene renderizzato con i nodi corretti (verificato tramite presenza label "entry" e "helper").
E2E-03	**Input Validation (File Invalido)** Tentativo di upload (Drag & Drop) di un file non supportato (es. .txt).	L'azione viene bloccata; appare un Toast Message di errore: "Please upload a valid .py file"; il bottone Generate resta disabilitato.
E2E-04	**Interazione Nodi (Drill-down)** Click su un nodo del grafo renderizzato.	Apertura modale (Modal); visualizzazione corretta delle metadati del nodo (Nome, File, Linee di codice).
E2E-05	**Visualizzazione Smells** Interazione con un nodo identificato come "Smelly" dal backend.	Il modale mostra la sezione "Detected Smells" in **rosso**; i dettagli dello smell (Nome, Linea) corrispondono al JSON di risposta mockato.

TC ID	Descrizione Scenario	Oracolo / Risultato Atteso
E2E-06	**Verifica Clean Code** Interazione con un nodo privo di smells.	Il modale mostra un feedback positivo in **verde** ("Excellent! No smell detected"); nessuna sezione di allerta smells è presente.
E2E-07	**Gestione Errori Backend** Simulazione fallimento API (HTTP 500) durante la generazione.	L'applicazione gestisce l'eccezione senza crash; viene mostrato un Toast Message con "Failed to generate graph".