TP - TEST PLAN

CodeFace4Smells

Riferimento	
Versione	1.0
Data	07/07/2025
Destinatario	Prof. Andrea De Lucia
Presentato da	Antonio Ferrentino, Francesco Perilli, Giuseppe Napolitano

Composizione gruppo	
Antonio Ferrentino	0522501898
Francesco Perilli	0522502073
Giuseppe Napolitano	0522501961

Cronologia revisioni

Data	Versione	Descrizione	Autori
10/07/2025	0.1	Inizio stesura del documento	Giuseppe Napolitano
11/07/2025	0.2	Stesura dei Test Case e dei Capitoli	Giuseppe Napolitano
12/07/2025	0.3	Completamento Test Case	Giuseppe Napolitano
13/07/2025	0.4	Revisione e correzioni	Antonio Ferrentino
14/07/2025	0.5	Revisione, ulteriori correzioni e modifiche	Giuseppe Napolitano, Antonio Ferrentino
16/07/2025	1.0	Revisione documento	Giuseppe Napolitano, Antonio Ferrentino, Francesco Perilli

Contents

1 Introduzione

L'obiettivo principale di questa sezione è quella di presentare l'attività di testing svolta sul software "Codeface4Smell", identificando gli elementi e le funzionalità da testare, insieme alle strategie di testing impiegate.

2 Panoramica del sistema

Il sistema CodeFace4Smell è un'estensione del progetto open source CodeFace, pensata per l'analisi del software e l'individuazione di **community smells**. Questi ultimi rappresentano forme latenti di debito tecnico che si manifestano come problematiche di collaborazione all'interno della community di sviluppo, rendendo la manutenzione del software più dispendiosa.

CodeFace4Smell integra metriche tecniche del codice e metriche sociali, combinando dati provenienti da repository Git, mailing list e commit log. Le analisi si basano su indicatori di bassa qualità del design software, dati storici e pattern di interazione tra sviluppatori.

I sottosistemi principali sono:

- **Bug Extractor**: estrae informazioni sui bug basandosi sui commit
- **Smell Detection**: rileva pattern sintattici e strutturali che potrebbero causare problematiche
- **id_service**: servizio Node.js che unifica gli alias degli sviluppatori all'interno delle repository.
- **Community metrics**: calcolano le metriche di tipo **socio-tecnico**, combinando i dati delle mailing list e dei repositori Git; le metriche principali individuate sono: **numero di commit**, **indice di attività**, **distribuzione delle modifiche**, etc.
- **Experiments & Performance**: consente l'esecuzione di esperimentim che forniscono in output analisi statistiche (code smells + metriche sociali).
- **Integration Scripts**: scripts che orchestrano l'esecuzione sequenziale dei moduli ed il collegamento tra i vari dati.
- **Report Generator**: sistema automatico per la generazione di report PDF delle analisi eseguite.
- **Ambiente di deploy preconfigurato**: grazie a Vagrant, script R/Python e configurazioni di sistema, l'intero ambiente di analisi può essere replicato facilmente.

3 Funzionalità da testare

La seguente tabella riporta, per ogni requisito funzionale presente nel sistema, coloro che sono sottoposti a test, con il relativo identificativo del test case.

Identificativo	Descrizione	Test Case
RF_CF_1	Installazione automatica	TCS_11
RF_CF_2	Setup ambientale	TCU_01
RF_CF_3	Configurazione MySQL	TCU_09, TCS_12
RF_CF_4	Importazione Repository	TCU_10, TCS_13
RF_CF_5	Esecuzione Analisi	TCU_02, TCU_07, TCS_014
RF_CF_6	Rilevamento revisioni/tag	TCU_04, TCS_15
RF_CF_7	Analisi temporale	TCU_03, TCS_16
RF_CF_8	Analisi strutturale	TCU_05, TCS_17
RF_CF_9	Rilevamento Code Smells	TCU_06, TCS_18
RF_CF_10	Generazione report	TCS_19
RF_CF_11	Visualizzazione grafi	TCS_20
RF_CF_12	Esportazione risultati	TCS_21
RF_CF_13	Esecuzione test	TCS_22
RF_CF_14	Gestione ambienti separati	TCS_23
RF_CF_15	Logging	TCU_08

4 Criteri pass/failed

Per ogni failure che è stato riscontrato durante l'esecuzione, verrà individuato il relativo fault e si procederà alla correzione di quest'ultimo. Successivamente, sarà reiterata la fase di testing per verificare che le modifiche apportate non abbiano avuto un impatto dannoso sulle altre componenti del sistema.

5 Approccio

L'attività di testing è stata strutturata in modo incrementale, con l'obiettivo di garantire la stabilità e la correttezza del sistema **CodeFace4Smell**, in seguito agli interventi di refactoring e aggiornamento effettuati sugli script in **R** e **Python**.

A differenza di un classico approccio *pre/post-refactoring*, in questo caso non è stato possibile effettuare un testing preliminare sulle versioni originali, in quanto il sistema risultava non funzionante e impossibilitato ad avviarsi correttamente.

Pertanto, l'approccio adottato si è articolato come segue:

1. **Fase di stabilizzazione**
   Inizialmente si è lavorato alla risoluzione progressiva degli errori bloccanti, tramite:

   - Debug e revisione del codice Python e R.

- Correzione delle dipendenze e dei percorsi errati.
- Configurazione del database e validazione dei file di progetto.

Questa fase è stata fondamentale per rendere il sistema eseguibile e raggiungere uno stato minimo funzionante.

2. **Testing incrementale**
   Una volta ristabilita l'esecuzione del sistema, è stata avviata una fase di testing, comprendente:

   - Test di unità su componenti Python e moduli di parsing.
   - Test di sistema e integrazione, con verifica degli script R e dell'interazione con il database.
   - Analisi degli output prodotti (bug, smell, cluster, report) al fine di garantire la correttezza dei flussi.

   Il confronto dei risultati ottenuti con le aspettative teoriche ha permesso di validare la correttezza funzionale del sistema aggiornato.

5.1 Testing di sistema e di integrazione

Il testing di sistema e di integrazione per il progetto CodeFace4Smell ha l'obiettivo di verificare il corretto funzionamento delle funzionalità offerte dal sistema, così come descritte nei requisiti funzionali. Tale attività è stata svolta in due fasi distinte: una prima esecuzione è stata condotta prima dell'attività di manutenzione evolutiva, al fine di raccogliere lo stato attuale; la seconda fase è stata effettuata al termine delle modifiche, per verificare che gli aggiornamenti introdotti non compromettessero le funzionalità esistenti.

Il testing è stato condotto adottando un approccio black-box, in cui la definizione dei casi di test è avvenuta esclusivamente sulla base dei requisiti funzionali e del comportamento atteso in termini di input/output. La verifica dei risultati (oracolo) è stata formulata in riferimento ai documenti di specifica, con particolare attenzione agli output generati dai moduli di analisi, clustering, esportazione dati e visualizzazione.

L'esecuzione dei test di sistema è stata effettuata manualmente tramite interfaccia a linea di comando (codeface run, codeface test, vagrant up) e mediante la consultazione diretta dei file di output, log, database e grafici generati. I test-case sono stati formalizzati in tabelle descrittive che riportano obiettivo, input, azione, output atteso ed esito previsto per ogni requisito.

I test di integrazione sono stati assorbiti nei test di sistema, poiché ciascun test-case automatizza la verifica completa del comportamento dei moduli integrati tramite CLI (codeface run, codeface test) e non è stato necessario effettuare test di integrazione separati. Il comportamento end-to-end rappresenta già la verifica dell'interazione tra componenti (configurazione, parser, DB, script R).

5.2 Testing di Unità

Il testing di unità è stato progettato secondo un approccio **white-box**, in cui ciascun test verifica il comportamento interno dei singoli moduli Python e R sviluppati o modificati durante l'attività di manutenzione evolutiva. L'accesso diretto al codice sorgente ha permesso di progettare i test conoscendo in anticipo l'implementazione delle funzioni, la logica di controllo (`if/else`, `try/except`, cicli) e le variabili locali utilizzate. A differenza del testing black-box, l'attenzione è posta sul funzionamento interno dei moduli piuttosto che sul solo comportamento osservabile.

I moduli testati comprendono:

- Il caricamento e la validazione delle configurazioni YAML (`test_configuration.py`);
- L'integrazione e l'esecuzione dell'analisi statica tramite `cppstats` (`test_cppstats_works.py`);
- La logica di clustering degli sviluppatori sulla base della co-evoluzione dei commit (`test_cluster.py`);
- L'estrazione di feature strutturali da codice sorgente C/C++ (`test_getFeatureLines.py`);
- La generazione e verifica dei log (`test_logger.py`);
- L'esecuzione degli script R e la verifica dell'output risultante (`test_R_code.py`);
- Il comportamento della CLI `codeface` in risposta a comandi validi o incompleti (`test_cli.py`);
- La creazione ed esecuzione dei job batch tramite parametri di configurazione (`test_batchjob.py`).

L'approccio white-box adottato è riconducibile alla tecnica di **statement coverage**, in cui l'obiettivo è garantire che tutte le istruzioni del codice vengano eseguite almeno una volta durante i test. Sono stati inclusi anche test negativi, per simulare scenari di errore realistici (es. assenza di file, directory non scrivibili, input malformati), assicurando che il sistema reagisse in modo controllato.

Tutti i test sono stati strutturati in moduli separati e automatizzabili, eseguibili tramite framework standard (`unittest`, `pytest`, `Rscript`), rendendo semplice la loro esecuzione in ambienti virtualizzati. Questa organizzazione ha facilitato il debugging e la validazione durante le fasi di refactoring, permettendo di individuare tempestivamente regressioni e assicurare la stabilità del sistema. Di seguito riportiamo i test unit ricavati:

TCU_01 – Test configurazione ambiente

Obiettivo	Verificare la corretta creazione dell'ambiente Python e delle dipendenze tramite `test_configuration.py`.
Input	Sistema Python con script `test_configuration.py` disponibile.
Azione	Eseguire lo script `python test_configuration.py`.
Output atteso	Ambiente creato con tutte le dipendenze soddisfatte.
Esito	**PASS** se l'ambiente risulta funzionante e pronto all'uso.

TCU_02 – Test analisi cppstats

Obiettivo	Verificare che l'analisi `cppstats` venga eseguita correttamente.
Input	File sorgente da analizzare.
Azione	Eseguire lo script che invoca `cppstats`.
Output atteso	Report `.csv` con metadati e metriche cppstats.
Esito	**PASS** se il file viene prodotto correttamente.

TCU_03 – Test script R

Obiettivo	Controllare l'integrazione e il corretto funzionamento degli script R per l'analisi.
Input	Script R e file di configurazione corretti.
Azione	Eseguire `Rscript analyse-ts.R`.
Output atteso	File statistici e grafici di output.
Esito	**PASS** se l'analisi viene completata senza errori.

TCU_04 – Test configurazione batchjob

Obiettivo	Validare l'esecuzione delle analisi con parametri corretti e il rilevamento di revisioni/tag.
Input	File `.conf` con progetto Git e parametri.
Azione	Eseguire il batchjob di analisi.
Output atteso	Revisoni e tag correttamente rilevati.
Esito	**PASS** se il progetto viene analizzato senza errori.

TCU_05 – Test clustering sviluppatori

Obiettivo	Verificare la corretta generazione dei cluster di sviluppatori dai commit.
Input	Log Git con commit.
Azione	Eseguire lo script `test_cluster.py`.
Output atteso	File con gruppi di sviluppatori.
Esito	**PASS** se i cluster sono ben formati.

TCU_06 – Test estrazione feature linee

Obiettivo	Testare la rilevazione di feature nei file.
Input	File contenente linee di codice annotate.
Azione	Eseguire lo script Python di estrazione feature.
Output atteso	Tabella con feature identificative.
Esito	**PASS** se le feature sono estratte in modo coerente.

TCU_07 – Test CLI Codeface

Obiettivo	Verificare che il comando CLI `codeface run` funzioni correttamente.
Input	File `.conf` e repository Git valido.
Azione	Eseguire `codeface run -c config -p project -o outputDir`.
Output atteso	Cartella di output popolata con log e report.
Esito	**PASS** se il comando termina senza errori.

TCU_08 – Test logger

Obiettivo	Verificare che vengano generati correttamente i file di log.
Input	Esecuzione batchjob completa.
Azione	Controllare la presenza e il contenuto dei file `.log`.
Output atteso	Log completi, dettagliati e coerenti.
Esito	**PASS** se i log descrivono correttamente le attività.

TCU_09 – Test database MySQL

Obiettivo	Verificare la creazione, connessione e gestione del database MySQL.
Input	Credenziali MySQL e file `schema.sql`.
Azione	Eseguire script di setup e test di connessione.
Output atteso	Connessione stabile e struttura DB corretta.
Esito	**PASS** se il DB è accessibile e utilizzabile.

TCU_10 – Test configurazione progetto

Obiettivo	Verificare l'importazione e la configurazione corretta del progetto Git.
Input	Repository Git da analizzare.
Azione	Eseguire importazione e inizializzazione progetto.
Output atteso	Dati iniziali presenti e consistenti.
Esito	**PASS** se il progetto è pronto per l'analisi.

5.3 Analisi della Copertura del Codice e Considerazioni sui Test di Integrazione

Nel corso delle attività di testing, è stata eseguita un'analisi approfondita della copertura del codice (*code coverage*) al fine di valutare l'effettiva efficacia dei test implementati, in particolare dopo la fase di manutenzione evolutiva del sistema `CodeFace4Smell`. L'analisi è stata condotta utilizzando il tool `coverage.py`, con successiva generazione di report in formato HTML. Il risultato ha evidenziato una copertura media del **57.56%**, distribuita su **65 file analizzati**.

La copertura è risultata elevata nei moduli unitari e ausiliari (come `logger.py`, `configuration.py`, `test/unit/`) grazie all'adozione di test di unità granulari e mirati. Al contrario, i moduli più critici per il funzionamento del sistema (es. `project.py`, `cluster.py`, `dbmanager.py`, `commit_analysis.py`) presentano coperture inferiori al 30%, principalmente a causa della loro dipendenza da risorse esterne (file `.conf`, database MySQL, script R), della complessità del flusso di esecuzione e della mancanza di isolamento della logica interna. Questi moduli richiederebbero strategie di testing più avanzate come *mocking*, *sandboxing* o *refactoring*, che non rientravano nei vincoli di progetto previsti.

Per quanto riguarda i test di integrazione, si è scelto consapevolmente di non separarli formalmente dai test di sistema, poiché il comportamento osservabile dei componenti è già stato verificato tramite test *black-box* condotti sull'intero flusso funzionale. In particolare, l'invocazione dei comandi `codeface run`, `codeface test` e degli script R ha consentito di testare in modo *end-to-end* l'integrazione tra moduli (caricamento configurazioni, parsing dei repository, salvataggio nel database, esecuzione di script di analisi, generazione di report), rendendo ridondante la creazione di test d'integrazione separati. Ogni test-case di sistema, infatti,

implica già la cooperazione tra più componenti software e garantisce la copertura funzionale dell'intero stack.

Questa scelta è stata documentata all'interno del presente piano di test e riflette una pratica consolidata nei progetti a esecuzione *batch* o basati su orchestrazione `CLI`, nei quali i test *black-box* sui comandi principali risultano sufficienti a garantire la validazione dell'interazione tra i moduli.

6 Materiale per il testing

Gli elementi utilizzati per effettuare il testing sono stati i seguenti:

- Ambiente virtuale **Vagrant** (tramite VM con Ubuntu)
- Dataset **Git** e **Mailing** list
- Log **CSV**, **JSON** generati dagli script

7 Test Case

TCS_11 – Installazione automatica

Obiettivo	Verificare che il sistema venga installato tramite Vagrant con provisioning automatico.
Input	Sistema con Vagrant installato (progetto clonato).
Azione	Eseguire '**vagrant up**' dalla root.
Output atteso	Ambiente pronto, provisioning completato senza errori.
Esito	PASS se la macchina è accessibile e pronta all'uso.

TCS_12 – Configurazione MySQL

Obiettivo	Verificare la configurazione automatica o manuale del database MySQL.
Input	Sistema MySQL attivo, file schema disponibile.
Azione	Eseguire script di configurazione MySQL o configurare manualmente.
Output atteso	Database creato correttamente con schema inizializzato.
Esito	PASS se si può accedere ai dati da CodeFace.

TCS_13 – Importazione repository Git

Obiettivo	Verificare che un repository Git venga importato correttamente.
Input	URL o path di un progetto Git.
Azione	Eseguire 'codeface import', o comando equivalente.
Output atteso	Repository disponibile nella struttura dati del sistema.
Esito	PASS se il progetto è accessibile da codeface.

TCS_14 – Esecuzione analisi

Obiettivo	Verificare che l'analisi venga eseguita correttamente.
Input	File '**.conf**' valido.
Azione	Eseguire '**codeface run -c config.conf -p project.conf outputDir**'.
Output atteso	File di output generati in directory specificata.
Esito	PASS se output presenti e corretti.

TCS_15 – Rilevamento revisioni e tag

Obiettivo	Verificare che revisioni e tag siano rilevati automaticamente.
Input	Repository Git valido.
Azione	Eseguire analisi su progetto con più tag/versioni.
Output atteso	Lista completa di revisioni e tag rilevati.
Esito	PASS se revisioni e tag appaiono correttamente nei dati.

TCS_16 – Analisi temporale

Obiettivo	Verificare la produzione dell'analisi temporale.
Input	File '**.conf**', script R attivi.
Azione	Avviare script '**analyse-ts.R**'.
Output atteso	Serie temporali in database e grafici associati.
Esito	PASS se le analisi sono consistenti con i dati temporali.

TCS_17 – Analisi strutturale

Obiettivo	Verificare la generazione di grafi e cluster.
Input	Repository con cronologia di commit.
Azione	Eseguire pipeline strutturale.
Output atteso	Cluster e grafi popolati nel database.
Esito	PASS se i risultati sono visibili nei report.

TCS_18 – Rilevamento code smells

Obiettivo	Verificare l'identificazione degli smells nel codice.
Input	Codice con smell noti.
Azione	Eseguire modulo '**detect-smells**'.
Output atteso	JSON contenente code smells rilevati.
Esito	PASS se gli smell sono rilevati correttamente.

TCS_19 – Generazione report

Obiettivo	Verificare la generazione dei report **PDF** e **HTML**.
Input	Analisi completata, template presenti.
Azione	Eseguire comando di generazione report.
Output atteso	Report generati e salvati correttamente.
Esito	PASS se report contengono grafici e metriche attese.

TCS_20 – Visualizzazione grafi

Obiettivo	Verificare che i grafi siano correttamente generati e visualizzabili.
Input	Output analisi strutturale.
Azione	Aprire file **SVG/HTML** generati.
Output atteso	Grafi chiari e completi.
Esito	PASS se tutti i nodi e connessioni sono visibili.

TCS_21 – Esportazione risultati

Obiettivo	Verificare l'esportazione dei risultati in CSV o SQLite.
Input	Output analisi presente.
Azione	Controllare directory output.
Output atteso	File esportati in formato valido.
Esito	PASS se i file sono corretti e coerenti.

TCS_22 – Esecuzione test

Obiettivo	Verificare il corretto funzionamento del comando 'codeface test'.
Input	File '**.conf**' di test.
Azione	Eseguire '**codeface test -c conf**'.
Output atteso	Output test comprensivo di successi/errori.
Esito	PASS se tutti i test previsti sono eseguiti.

TCS_23 – Gestione ambienti separati

Obiettivo	Verificare la separazione tra ambienti di test e produzione.
Input	Configurazioni DB distinte.
Azione	Eseguire test e analisi su DB separati.
Output atteso	Nessuna interferenza tra ambienti.
Esito	PASS se i dati restano confinati nel DB corretto.