UNIVERSITÀ DEGLI STUDI
DI SALERNO

Test Plan

CodeFace4Smells

Versione	2.0
Data	27/10/2025
Destinatario	Prof. Andrea De Lucia
Presentato da	Gabriele Santoro Pasquale Sorrentino

Composizione Gruppo	
Gabriele Santoro	0522502066
Pasquale Sorrentino	0522501954

Cronologia Revisioni

Data	Versione	Descrizione	Autori
29/09/2025	0.1	Inizio stesura del documento	Gabriele Santoro Pasquale Sorrentino
29/09/2025	0.2	Suddivisione del documento in capitoli	Gabriele Santoro Pasquale Sorrentino
30/09/2025	0.3	Realizzazione dei test case	Gabriele Santoro Pasquale Sorrentino
01/10/2025	0.4	Definizione dell'approccio al testing	Gabriele Santoro Pasquale Sorrentino
02/10/2025	0.5	Realizzazione dei test d'Unità	Gabriele Santoro Pasquale Sorrentino
03/10/2025	1.0	Stesura finale del documento	Gabriele Santoro Pasquale Sorrentino
13/10/2025	1.1	Modifica dei test **TC_CD_1, TU_CD_1** per via della **CR_01**	Gabriele Santoro Pasquale Sorrentino
21/10/2025	2.0	Revisione del documento	Gabriele Santoro Pasquale Sorrentino

Indice

1 Introduzione

L'obiettivo principale di questo documento è quello 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 progetto ***Codeface4Smells*** si basa sullo strumento open source *CodeFace*, usato per effettuare un'analisi dell'evoluzione del software, e l'individuazione di **community smells**. Questi ultimi sono delle forme di debito tecnico che si presentano come problematiche di collaborazione all'interno della community di sviluppo, rendendo la manutenzione del software più dispendiosa.

CodeFace4Smells integra metriche tecniche e metriche sociali, combinando i dati provenienti da Git, mailing list e commit log.

I sottosistemi principali sono:

- Bug Extractor: Estrae informazioni sui bug basandosi sui commit
- Smell Detection: Rileva pattern sinstattici e strutturali che potrebbero causare problematiche
- id_service: Servizio Node.js che unifica gli alias degli sviluppatori all'interno della repository
- Community Metrics: Calcolano le metriche di tipo socio-tecnico, combinando i dati delle mailing list e dei repository Git.
- Experiments & Performance: Consente l'esecuzione di esperimenti che forniscono in output analisi statistiche
- Integration Scripts: Scripts che gestiscono l'esecuzione generale del progetto, a partire dall'installazione
- rpackages: Ulteriori scripts che gestiscono l'installazione dei pacchetti R richiesti dal sistema per funzionare.
- Report Generator: Sistema automatico per la generazione dei report PDF riguardanti le analisi eseguite
- Ambiente di deploy preconfigurato: Grazie all'uso di Docker, di script di installazione R e Python, e configurazioni di sistema, l'intero ambiente di analisi è facilmente replicabile.

3 Funzionalità da testare

La seguente tabella riporta per ogni requisito funzionale presente, quelli sottoposti a test.

Categoria	ID Requisito	Descrizione	Test
Gestione Containerizzazione e Deploy	**RF_CD_1**	Il sistema deve supportare il deployment automatico di tutti i servizi attraverso Docker Compose con gestione delle dipendenze	**TU_CD_1**
	RF_CD_2	Il sistema deve supportare l'installazione delle librerie necessarie al suo funzionamento	**TC_CD_1** **TU_CD_2** **TU_CD_3**
	RF_CD_3	Il sistema deve supportare l'installazione e la configurazione del database MySQL	**TC_CD_2**
Analisi ed Elaborazione Dati	**RF_AD_4**	Il sistema deve consentire l'import di repository da diverse sorgenti	**TC_AD_1**
	RF_AD_5	Il sistema deve consentire l'analisi focalizzata su release e intervalli temporali specifici	**TC_AD_2** **TU_AD_1** **TU_AD_2** **TU_AD_3**
	RF_AD_6	Il sistema deve implementare algoritmi per il rilevamento di community smell	**TC_AD_3** **TU_AD_3**
	RF_AD_7	Il sistema deve consentire l'analisi dei commit e tag nel tempo	**TC_AD_4**
	RF_AD_8	Il sistema deve generare gradi di clusterizzazione degli sviluppatori	**TC_AD_5** **TU_AD_4**

	RF_AD_9	Il sistema deve esporre delle API REST per la fruizione dei dati	**TC_AD_6**
Output e Visualizzazione	**RF_OV_10**	Il sistema deve generare report completi	**TC_OV_1**
	RF_OV_11	Il sistema deve fornire una dashboard web per l'esplorazione dei risultati in tempo reale	**TC_OV_2** **TU_OV_1**
Testing	**RF_TE_12**	Il sistema supportare l'attività di testing tramite comandi appositi	**TC_TE_1** **TU_TE_1**
	RF_TE_13	Il sistema dovrà produrre file di log dettagliati	**TC_TE_2**

4 Criteri pass/failed

Per ogni failure riscontrata durante l'esecuzione verrà individuato il relativo fault e ne sarà effettuata la correzione. Successivamente, la fase di testing sarà ripetuta per verificare che le modifiche introdotte non abbiano avuto effetti negativi sulle altre componenti del sistema.

5 Approccio

L'approccio al testing è stato organizzato in maniera incrementale, con l'obiettivo di garantire la stabilità e la correttezza del sistema CodeFace4Smell a seguito degli interventi di aggiornamento della piattaforma.

Non è stato possibile adottare una strategia di testing comparativa pre/post-refactoring, poiché il sistema risultava inizialmente non funzionante.

Pertanto l'approccio utilizzato è stato il seguente:

1. **Fase di Stabilizzazione**

   In una prima fase si è intervenuti per risolvere gli errori riscontrati, attraverso:
   a. attività di debug e revisione del codice Python e R,

b. correzione delle dipendenze mancanti o dei percorsi non corretti,
c. configurazione del database e definizione dello schema.

2. **Testing Incrementale**

   Ripristinato il corretto funzionamento del programma, si è passati alla fase di testing, che ha previsto:
   a. l'esecuzione di test di unità sulle componenti Python,
   b. test di sistema e di integrazione, con particolare attenzione agli script R e alla loro interazione con il database,
   c. l'analisi degli output prodotti, al fine di garantire la correttezza e l'affidabilità.

In seguito all'analisi della Change Request CR_01 relativa alla sostituzione di Vagrant con Docker, è stato necessario eseguire un aggiornamento della suite di test per garantire che la validazione del sistema riflettesse la nuova architettura containerizzata. Nello specifico, il Test Case TC_CD_1 (dedicato alla validazione della procedura di build e containerizzazione) e il Test di Unità TU_CD_1 (focalizzato sul verificare la corretta esecuzione dei processi di installazione e configurazione) sono stati riadattati alle nuove modalità di deployment.

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 di analisi socio-tecnica del software, 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 del sistema; la seconda fase è stata effettuata al termine delle modifiche, per verificare che gli aggiornamenti introdotti non compromettesse 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 commit, clustering di sviluppatori, analisi temporale, rilevamento di code smells, esportazione dati e visualizzazione tramite interfaccia web Shiny.

L'esecuzione dei test di sistema è stata effettuata manualmente tramite interfaccia a linea di comando (*codeface run*, *codeface test*) e mediante la consultazione diretta dei file di output, log, database MySQL e grafici generati.

Per quanto concerne i test di integrazione, è stata adottata una strategia di unificazione con i test di sistema piuttosto che una separazione formale. Questa scelta metodologica è motivata dal fatto che il comportamento osservabile dei singoli componenti risulta già ampiamente verificato attraverso i test black-box condotti sull'intero flusso funzionale.
L'esecuzione dei comandi codeface run, codeface test e degli script R ha infatti permesso di testare in modalità end-to-end l'integrazione completa tra i diversi moduli del sistema, includendo il caricamento delle configurazioni, il parsing dei repository, il salvataggio nel database, l'esecuzione degli script di analisi e la generazione dei report. Tale approccio ha reso superflua la creazione di test d'integrazione separati, garantendo al contempo una copertura completa delle interazioni inter-modulari.

I test-case sono stati formalizzati in tabelle descrittive che riportano obiettivo, input, azione, output atteso ed esito previsto per ogni requisito funzionale. Questa strategia integrata ha consentito di ottenere una verifica efficace e completa del comportamento dei moduli integrati tramite CLI, eliminando ridondanze e ottimizzando l'effort di testing complessivo.

Il comportamento end-to-end rappresenta l'integrazione completa tra i moduli di analisi Python (commit analysis, clustering), i moduli R (time series analysis, socio-technical analysis, code smell detection), il database MySQL per la persistenza dei dati, e l'interfaccia web Shiny per la visualizzazione dei risultati.

5.2 Testing di Unità

Il testing di unità, a differenza del testing di sistema, 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 con gestione degli errori di parsing e validazione dei parametri (test_configuration.py);

- L'integrazione e l'esecuzione dell'analisi statica tramite cppstats per l'estrazione di feature strutturali da codice C/C++ (test_cppstats_works.py);

- La logica di clustering degli sviluppatori sulla base della co-evoluzione dei commit e il calcolo delle dipendenze logiche (test_cluster.py);

- L'estrazione e parsing di feature strutturali da codice sorgente C/C++ con gestione di costrutti condizionali complessi (#if, #elif, #else) (test_getFeatureLines.py);

- La generazione e verifica dei log con formattazione colorata e gestione dei livelli di logging (test_logger.py);

- L'esecuzione degli script R tramite Rscript e la verifica dell'output risultante utilizzando il framework testthat (test_R_code.py);

- Il comportamento della CLI codeface in risposta a comandi validi o incompleti con validazione degli argomenti (test_cli.py);

- La creazione ed esecuzione dei job batch tramite parametri di configurazione con gestione delle dipendenze e degli errori (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. file di configurazione malformati, errori di parsing YAML, dipendenze logiche errate), assicurando che il sistema reagisse in modo controllato tramite eccezioni specifiche.

Tutti i test sono stati strutturati in moduli separati e automatizzabili, eseguibili tramite framework standard (unittest per Python, testthat per R, Rscript per l'esecuzione degli script R), rendendo semplice la loro esecuzione in ambienti virtualizzati tramite Docker. Questa organizzazione ha facilitato il debugging e la validazione durante le fasi di refactoring, permettendo di individuare tempestivamente regressioni nei moduli di analisi commit, clustering, estrazione feature e gestione delle configurazioni.

5.2.1 TU_CD_1 Testing avvio della macchina

Obiettivo	Testare la configurazione e l'avvio della macchina
Input	Dockerfile configurato correttamente
Azione	Eseguire comando *docker build . -t codeface4smell* e verificare il corretto avvio del sistema
Output atteso	Il sistema si avvia correttamente
Esito	PASS se il sistema si avvia ed è pronto all'uso

5.2.2 TU_CD_2 Testing configurazione del sistema

Obiettivo	Testare la configurazione del sistema
Input	File di testing *test_configuration.py*
Azione	Eseguire lo script in input
Output atteso	L'ambiente è configurato correttamente
Esito	PASS se lo script ritorna OK

5.2.3 TU_CD_3 Testing librerie R

Obiettivo	Testare il corretto funzionamento delle librerie R
Input	File di testing *analyse_ts.r*
Azione	Eseguire lo script in input
Output atteso	File e grafici in output
Esito	PASS se i file vengono prodotti correttamente

5.2.4 TU_AD_1 Testing CppStats

Obiettivo	Verificare che le analisi cppstats funzionino
Input	File da analizzare e script *test_cppstats_works.py*
Azione	Eseguire lo script *test_cppstats_works.py*
Output atteso	File .csv con metriche cppstats
Esito	PASS se il file viene prodotto correttamente

5.2.5 TU_AD_2 Rilevamento Revisioni/Tag

Obiettivo	Validare l'esecuzione dell'analisi
Input	Script *test_batchjob.py*
Azione	Eseguire lo script *test_batchjob.py*
Output atteso	Revisioni e tag vengono correttamente rilevati
Esito	PASS se l'analisi viene portata a termine

5.2.6 TU_AD_3 Test CLI Codeface

Obiettivo	Verificare che la CLI di Codeface funzioni correttamente
Input	Script *test_cli.py*
Azione	Eseguire lo script *test_cli.py*
Output atteso	Tabella con le possibili opzioni
Esito	PASS se le feature sono correttamente mostrate

5.2.7 TU_AD_4 Test clustering

Obiettivo	Verificare la corretta generazione di cluster
Input	File di configurazione e script *test_cluster.py*
Azione	Eseguire lo script *test_cluster.py*
Output atteso	Vengono generati file con gruppi di sviluppatori
Esito	PASS se i cluster vengono generati correttamente

5.2.8 TU_OV_1 Testing Dashboard

Obiettivo	Verificare il corretto funzionamento della dashboard
Input	Script *test_dashboard.py*
Azione	Eseguire lo script *test_dashboard.py*
Output atteso	La dashboard funziona correttamente
Esito	PASS se la dashboard funziona correttamente

5.2.9 TU_TE_1 Test generazione Log

Obiettivo	Verificare che vengano correttamente generati i file di log
Input	Script *test_logger.py*
Azione	Eseguire lo script *test_logger.py*
Output atteso	Log completi delle azioni del sistema
Esito	PASS se i log descrivono correttamente le attività

5.3 Analisi della Copertura del Codice

Per valutare l'efficacia e la completezza della strategia di testing adottata, è stata condotta un'analisi quantitativa della code coverage utilizzando lo strumento coverage.py, un tool standard per la misurazione della copertura del codice nei progetti Python.

Coverage.py è stato integrato nel workflow di testing per fornire metriche oggettive sulla percentuale di codice effettivamente eseguito durante l'esecuzione dei test, consentendo di identificare eventuali porzioni di codice non testate e di valutare la qualità complessiva della suite di test.

5.3.1 Configurazione del Tool

Il file di configurazione *.coveragerc* è stato personalizzato per:

- Includere tutti i moduli sorgente del progetto
- Escludere file di configurazione, script di setup e dipendenze esterne
- Omettere linee contenenti commenti, docstring e gestione delle eccezioni non critiche
- Generare report sia in formato console che HTML per facilitare l'analisi

5.3.2 Risultati della Coverage Analysis

L'analisi ha prodotto le seguenti metriche di copertura:

- Coverage complessiva: 65% del codice sorgente
- Linee totali analizzate: 4286 linee di codice
- Linee eseguite: 2785 linee coperte dai test
- Linee non coperte: 1501 linee non raggiunte durante l'esecuzione

6 Materiale per il testing

Gli strumenti usati per l'attività di testing sono:

- Containerizzazione tramite l'uso di Docker
- Web browser per monitorare la dashboard
- Dataset di Mailing List e GIT
- Script di testing in Python

7 Test Case

7.1 Gestione Containerizzazione e Deploy

7.1.1 TC_CD_1 - Installazione librerie necessarie

Obiettivo	Verificare che le librerie siano state installate correttamente
Input	Dockerfile
Azione	Eseguire comando *docker build . -t codeface4smell* e verificare che all'interno dei log non vi siano errori
Output atteso	Tutte le librerie vengono correttamente installate

Esito	PASS se il sistema si avvia e le librerie sono installate

7.1.2 TC_CD_2 - Configurazione MySQL

Obiettivo	Verificare che MySQL venga installato e configurato correttamente
Input	Servizio MySQL, e schema del database
Azione	Eseguire lo script di configurazione MySQL, o procedere alla installazione manuale
Output atteso	Tutte le tabelle presenti nello schema vengono create correttamente
Esito	PASS se il database è popolato da tutte le tabelle

7.2 Analisi ed Elaborazione Dati

7.2.1 TC_AD_1 - Importazione di Repository

Obiettivo	Verificare le repository GIT siano importabili correttamente
Input	URL di un progetto GIT
Azione	Eseguire un comando *git clone* all'interno della cartella git-repos
Output atteso	Il progetto GIT viene correttamente importato
Esito	PASS il repository viene clonato correttamente

7.2.2 TC_AD_2 - Esecuzione Analisi

Obiettivo	Verificare che l'analisi venga eseguita correttamente
Input	Sistema correttamente installato, e file *.conf*

Azione	Eseguire il comando *codeface run -c config.conf -p project.conf repo-dir/ results-dir/*
Output atteso	File di output generati nella cartella specificata dal comando
Esito	PASS l'output viene generato ed è corretto

7.2.3 TC_AD_3 - Rilevamento code smells

Obiettivo	Verificare l'identificazione degli smells nel codice
Input	Codice con smell noti
Azione	Eseguire il comando *codeface st -c codeface.conf -p project.conf results/*
Output atteso	File di output generato contenente i code smells rilevati
Esito	PASS se gli smell sono rilevati correttamente

7.2.4 TC_AD_4 - Analisi delle revisioni nel tempo e dei tag

Obiettivo	Verificare che le revisioni e i tag vengano rilevati correttamente
Input	Repository GIT scaricato, file di configurazione *.conf*
Azione	Eseguire l'analisi completa del progetto
Output atteso	Lista completa dei dati rilevati dall'analisi dei tag e delle revisioni
Esito	PASS se le analisi vengono generate e sono corrette

7.2.5 TC_AD_5 - Generazione di Cluster

Obiettivo	Verificare che i cluster vengono generati correttamente
Input	Repository GIT scaricato, file di configurazione *.conf*

Azione	Eseguire l'analisi completa del progetto
Output atteso	Cluster di sviluppatori generati correttamente
Esito	PASS se le analisi vengono generate e sono corrette

7.2.6 TC_AD_6 - Esposizione API REST

Obiettivo	Verificare che i servizi di API vengano esposti correttamente
Input	Progetto funzionante, script *id_service*
Azione	Lanciare il comando *node id_service.js ../codeface_testing.conf*
Output atteso	Servizio API REST funzionante
Esito	PASS se le API sono funzionanti, e rispondono correttamente

7.3 Output e Visualizzazione

7.3.1 TC_OV_1 - Generazione Report

Obiettivo	Verificare che i report siano effettivamente visibili
Input	Analisi completata e dati presenti
Azione	Eseguire il programma di generazione report
Output atteso	Report generati e visualizzabili correttamente
Esito	PASS se i report sono completi e visualizzabili

7.3.2 TC_OV_2 - Gestione Interfaccia Web

Obiettivo	Verificare che la dashboard sia attiva e funzionante
Input	Progetti analizzati già disponibili
Azione	Accedere alla dashboard web con indirizzo *localhost:8081*

Output atteso	La dashboard è correttamente funzionante e visualizza i progetti
Esito	PASS se la dashboard viene raggiunta, e i progetti sono presenti

7.4 Gestione Testing

7.4.1 TC_TE_1 - Funzione comando di Testing

Obiettivo	Verificare che il comando di testing funzioni
Input	Sistema pronto all'uso
Azione	Tramite la CLI di codeface, lanciare il comando *codeface test*
Output atteso	Dashboard correttamente funzionante e visualizza i progetti
Esito	PASS se tutti i test siano eseguiti correttamente

7.4.2 TC_TE_2 - Gestione logs di sistema

Obiettivo	Verificare che i log siano correttamente visualizzati e salvati
Input	Sistema pronto all'uso
Azione	Dopo la scansione di uno o più progetti, verificare che i log di sistema vengano correttamente popolati
Output atteso	Log completi e coerenti
Esito	PASS se i log descrivono correttamente l'attività