Master Test Plan Document

Versione 1.0

CodeSmile

Team Members:
Matteo Ercolino — 0522501462
Simone Silvestri — 0522501419

Repository: GitHub link

Anno Accademico 2024/2025

Indice

1 Introduzione

Questo documento definisce la strategia generale, l'approccio e le attività di test pianificate per il progetto **CodeSmile**. Il Master Test Plan copre i diversi livelli di testing (unit, integrazione, sistema, e2e) e la gestione della coverage, con particolare attenzione alle funzionalità che sono al centro delle modifiche pianificate (Change Requests).

1.1 Obiettivi del Documento

- Descrivere i livelli di testing pianificati: Unit, Integrazione, Sistema ed E2E.
- Specificare le aree funzionali incluse ed escluse dal testing.
- Definire gli obiettivi di copertura e i criteri di accettazione.
- Fornire una baseline di riferimento per le future attività di regressione e manutenzione.

2 Strategie di Testing

2.1 Testing di Unità

L'attività di **Unit Testing** è finalizzata a validare singolarmente i moduli e le classi fondamentali di CodeSmile, in modo da assicurare il corretto comportamento di ciascuna funzionalità a livello isolato.

2.1.1 Struttura e Organizzazione

La suite di test di unità è organizzata in base alla suddivisione in *package* del sistema. All'interno della directory `test/unit_testing` si trovano sottocartelle che rispecchiano i principali moduli del codice. Tale suddivisione rispecchia la struttura interna di CodeSmile, garantendo una corrispondenza uno-a-uno tra i test e i moduli interessati.

2.1.2 Approccio e Strumenti

White-box Testing. Il **White-box** è stato scelto come approccio principale per i test di unità. Grazie a questa tecnica, si analizza la *branch coverage* di ciascun modulo, verificando che tutti i possibili rami logici del codice vengano effettivamente esercitati durante l'esecuzione dei test.

Coverage. Per misurare la copertura, si utilizza *coverage.py*, che fornisce report dettagliati sulle linee e diramazioni coperte dai test. L'obiettivo prefissato è raggiungere (e possibilmente superare) l'80% di branch coverage per ciascun package core e per le regole di detection più critiche.

Strumenti di Testing. Il framework di riferimento è **pytest**, in abbinamento a:

- `pytest.fixture` per la creazione di *fixture* riutilizzabili (e.g., `tmp_path`, dati di esempio).
- *unittest.mock* per l'uso estensivo di `Mock` e `patch`, con cui simulare funzionalità esterne e isolare il codice in esame.

2.2 Testing di Integrazione

L'attività di **Integration Testing** ha lo scopo di validare le interazioni tra i principali componenti architetturali del sistema CodeSmile. In particolare, l'attenzione si è focalizzata sull'integrazione dei moduli centrali che realizzano il flusso completo dell'analisi: dal front-end (CLI e GUI) fino alla generazione del report finale, passando attraverso l'`Inspector` e il `RuleChecker`.

Approccio

Il testing è stato condotto con approccio **incrementale e top-down**, combinato con tecniche di **mocking strategico** al fine di isolare i singoli punti di interazione tra i moduli.

2.3 Testing di Sistema

Il **System Testing** si concentra sulla verifica end-to-end dell'intero flusso di CodeSmile, dall'input (directory o progetti) fino alla generazione dei report. L'approccio seguito è di tipo

black-box, con casi di test costruiti secondo la tecnica di **Category Partition** (scelta mirata di parametri come numero e tipo di file, struttura del progetto, modalità di esecuzione, ecc.). Il sistema è stato validato tramite una suite di test di sistema strutturata secondo il *Category Partition Method* (vedi documento "Pre-Modifications System Testing").

2.3.1 Organizzazione e Strumenti

I casi di test di sistema sono organizzati in cartelle numerate (`TC1`, `TC2`, ...) nella directory `system_testing`, ciascuna contenente file di esempio (anche con estensioni non `.py`) e directory annidate che rispecchiano vari scenari d'uso. Un file `test_system_runner.py` automatizza l'esecuzione di tutti i test tramite:

- **CLI** di CodeSmile, invocata con parametri diversi (es. `--parallel`, `--resume`).
- **Subprocessi Python**, per simulare l'utente finale che lancia il comando a riga di comando.
- **File non leggibili**, **interruzioni simulate** e **dir empty** per validare la robustezza in condizioni limite.

2.3.2 Benefici e Obiettivi

Questo livello di test garantisce che le varie componenti, già collaudate singolarmente nei test di unità e integrazione, operino in maniera corretta e robusta quando utilizzate in uno scenario più ampio e prossimo a quello reale. I risultati di *System Testing* forniscono inoltre una *baseline* importante per attività di *regression testing* dopo eventuali modifiche o evoluzioni del progetto.

2.4 Testing E2E (End-to-End) per la WebApp

Per la componente di **WebApp**, si è pianificato un insieme di test End-to-End (E2E) volti a validare i principali flussi utente, dall'accesso alla homepage fino alla generazione dei report. L'obiettivo è assicurare che tutte le parti dell'applicazione — front-end, API Gateway e servizi di backend — interagiscano correttamente, offrendo un'esperienza fluida e priva di errori.

2.4.1 Struttura e Strumenti

La struttura dei test E2E è organizzata nella directory `cypress/e2e`, dove ciascun file (`.spec.cy.ts`) corrisponde a una suite di scenari. Come framework di automazione si utilizza **Cypress**, che fornisce:

- **Simulazione delle azioni utente**: clic, upload di file, navigazione, gestione di pop-up/alert.
- **Intercept e mocking di API**: per testare correttamente flussi di rete, eventuali errori o risposte in formato JSON.

2.4.2 Obiettivi Principali

- **Verifica delle rotte e della navigazione**: i pulsanti presenti nella homepage devono reindirizzare correttamente alle varie sezioni (upload e report).
- **Corretto caricamento dei file**: la WebApp deve accettare file Python e progetti, mostrando all'utente i progressi e i risultati dell'analisi.
- **Robustezza in caso di errori**: in presenza di *failure* da parte dell'API (*HTTP 500*), l'applicazione deve avvisare l'utente con un messaggio esplicito senza interrompersi in modo anomalo.

- **Corretta generazione dei report**: dopo l'analisi, la dashboard deve mostrare i risultati (ad esempio, i *chart* con il conteggio dei code smells) e consentire l'eventuale download in PDF.

2.4.3 Criteri di Successo

I test E2E si considerano superati se:

- Il caricamento dei file e la conseguente analisi si concludono senza errori inattesi (ovvero, con risposte corrette e tempi di caricamento accettabili).
- La UI rende sempre disponibili i pulsanti e gli elementi grafici necessari (p.es. *progress bar*, messaggi di successo/errore).
- Tutti i flussi navigazionali (homepage → varie sezioni → visualizzazione report) funzionano correttamente in modo continuo.

3 Organizzazione delle Funzionalità Testate

3.1 Componenti Incluse dal Testing

Tutte le **componenti principali** di CodeSmile sono incluse nel piano di test:

- *components*, *detection_rules* e *extractor* per l'analisi statica.
- *cli* e *gui* per l'esecuzione via terminale e interfaccia desktop.
- *gateway* e *services* per la parte di WebApp (inclusi AI Analysis, Static Analysis, Report).

3.2 Componenti Escluse dal Testing

Sono al momento **escluse** dal piano:

- Funzionalità di gestione e addestramento del modello IA (focalizzate sull'aspetto ML).
- Componenti sperimentali o dedicati alla generazione di dataset, che verranno coperti in piani successivi se richiesto.

3.3 Coverage

L'obiettivo è raggiungere almeno l'80% di *branch coverage* complessiva, escludendo i moduli relativi alla parte sperimentale di IA o dataset. Nel dettaglio:

- **Unit Testing**: Mirare ad una coverage $\geq$80% per i package core (`components`, `detection_rules`, `extractor`).
- **Integrazione**: Valutare la percentuale di codici effettivamente toccati dai test integrati (obiettivo: oltre il 50%).
- **Sistema ed E2E**: Non necessariamente riflettono un'ampia coverage a livello di righe, ma garantiscono che i flussi principali del sistema siano testati in condizioni reali.

4 Pianificazione e Gestione del Testing

4.1 Allineamento con le Change Requests

Il piano di test considera le seguenti richieste di modifica approvate:

- **CR-01**: La nuova dashboard verrà validata tramite test E2E Cypress.
- **CR-02**: L'integrazione CI/CD sarà oggetto di test unitari e di sistema dedicati.
- **CR-03**: La modalità "Quick Scan" sarà oggetto di test unitari e di sistema dedicati.

4.2 Criteri di Accettazione e Uscita

- **Criteri di Accettazione**:
  - Raggiungimento dell'80% di branch coverage nei componenti principali.
  - Assenza di bug critici o blocker nel System Testing e E2E Testing.
  - Capacità del sistema di generare report finali coerenti con i code smells rilevati.
- **Criteri di Uscita**:
  - Il testing si considera concluso quando tutti i test pianificati sono stati eseguiti e approvati.
  - Non ci sono anomalie aperte di priorità alta o critica.

4.3 Integrazione in GitHub Actions

Per assicurare l'esecuzione continua dei test e il controllo della qualità del codice, il progetto CodeSmile utilizza una pipeline **CI** su *GitHub Actions*. Alla *push* o all'apertura di una *pull request* sui branch principali, il workflow di GitHub Actions esegue i passaggi descritti in un file YAML (definito in `.github/workflows/`), comprendenti linting, testing e calcolo della coverage.

Le principali **fasi** del workflow sono:

- **Checkout e setup dell'ambiente Python**: viene clonato il repository e predisposta la versione di Python desiderata (3.11).
- **Installazione dipendenze e linting**: si installano i pacchetti necessari da `requirements.txt` e viene eseguito `flake8` per controllare il codice.
- **Esecuzione dei test con coverage**: `pytest` esegue l'intera suite di test (escludendo eventuali directory non pertinenti), con `coverage.py` che registra la *branch coverage*.
- **Pubblicazione dei risultati**: il report di coverage viene caricato su *Codecov*, così da ottenere statistiche centralizzate e badge di copertura.

In questo modo, ogni modifica al codice è immediatamente validata attraverso l'esecuzione automatizzata di lint, test e analisi della coverage, garantendo trasparenza e stabilità nel processo di sviluppo.