UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

TEST PLAN

INGEGNERIA, GESTIONE ED EVOLUZIONE DEL SOFTWARE

DOCENTI

Prof. Andrea De Lucia

TUTOR

Dott. Stefano Lambiase

Università degli Studi di Salerno

STUDENTI

Francesco Maria Torino (0522501879)

Francesco Alessandro Pinto (0522501981)

Stefano Guida (0522502054)

Anno Accademico 2024-2025

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

Il presente documento costituisce il **Test Plan** relativo al progetto di manutenzione evolutiva del sistema *GUIDO* (Gathering and Understanding Socio-Technical Aspects in Development Organizations). L'obiettivo principale è descrivere in modo sistematico le strategie, gli strumenti e i criteri adottati per la verifica e la validazione delle funzionalità già esistenti e di quelle che saranno introdotte tramite l'integrazione del tool *TOAD* (Tool for Organizational and Activity Diagnosis).

1.1 Obiettivi del Test

Il Test Plan si propone di garantire che tutte le funzionalità del sistema, sia preesistenti sia di nuova implementazione, soddisfino i requisiti funzionali specificati e si comportino come atteso in condizioni operative reali. Gli obiettivi generali del processo di testing sono i seguenti:

- Verificare la correttezza e la stabilità delle funzionalità già presenti in GUIDO prima dell'attività di manutenzione;
- Valutare l'integrazione tecnica del tool TOAD all'interno dell'ecosistema GUIDO, con particolare attenzione alla comunicazione tramite interfacce RESTful;

- Validare la GUI sviluppata per l'interazione con TOAD, inclusa la corretta visualizzazione delle metriche e delle informazioni grafiche;
- Assicurare la continuità e l'assenza di regressioni nelle funzionalità preesistenti, a seguito delle modifiche introdotte.

1.2 Funzionalità da Testare

Le funzionalità che saranno oggetto di test sono riportate nella Tabella 1.1 e comprendono sia le funzionalità esistenti in GUIDO prima della manutenzione (RF01-RF04), sia quella che verrà aggiunta a seguito dell'integrazione di TOAD (RF05).

ID	Nome	Descrizione
RF01	Info	Permette all'utente di richiedere al chatbot informazioni sulle sue funzionalità e sui community smells che è in grado di rilevare.
RF02	Get Smells	Consente all'utente di richiedere al chatbot un'analisi di una repository GitHub per ottenere un report con tutti i community smells rilevati.
RF03	Get Smells By Date	Consente all'utente di richiedere al chatbot un'analisi di una repository GitHub, limitata a un determinato intervallo temporale, per ottenere un report sui community smells rilevati.
RF04	Get Cultural Dispersion	Consente all'utente di richiedere al Culture Inspector informazioni sulla dispersione culturale e geografica di una community o di un team di sviluppatori.
RF05	Get Community Pattern	Consente all'utente di richiedere tramite il Community Inspector (TOAD) un'analisi di una repository Github in una specifica finestra temporale di 3 mesi, per ottenere un report contenente i community pattern rilevati, le metriche calcolate ed un grafo che rappresenta le relazioni collaborative tra i membri del team.

Tabella 1.1: Requisiti Funzionali da Testare

CAPITOLO 2

Analisi dei Test Preesistenti

In questa sezione si analizzano i test già presenti nel sistema *GUIDO* prima dell'attività di manutenzione evolutiva. L'obiettivo è documentare l'attuale copertura funzionale, valutare la qualità dei test esistenti e fornire un quadro di riferimento per i test successivi all'integrazione del tool TOAD.

2.1 Struttura e Tipologia dei Test

Il sistema dispone di una suite di test articolata su tre livelli:

- **Unit Test**: implementati con `pytest`, mirano a verificare il comportamento di singole unità funzionali (funzioni e classi). È stato adottato un approccio di tipo **white-box**.
- **Integration Test**: anch'essi basati su `pytest`, si concentrano sulla verifica delle interazioni tra moduli software. L'approccio seguito è **bottom-up**, partendo dall'integrazione delle unità più elementari fino ad arrivare ai componenti più complessi.
- **System Test**: realizzati mediante `Selenium`, sono strutturati secondo la tecnica del **category partitioning** e testano il comportamento del sistema nel suo complesso, simulando l'interazione utente attraverso GUI.

2.2 Risultati dell'Esecuzione

L'esecuzione complessiva della suite ha prodotto i seguenti risultati:

- **Totale test eseguiti**: 55
- **Test superati**: 44
- **Test falliti**: 11
- **Tempo totale di esecuzione**: 91.02 secondi
- **Ambiente**: Python 3.10.0, pytest 8.2.0, Windows

2.2.1 Unit Test

Nome del Test	Esito
TestIntentResolverWebService::test_missing_data	Passed
TestIntentResolverWebService::test_missing_entities	Passed
TestIntentResolverWebService::test_resolver_exception_handling	Passed
TestIntentResolverWebService::test_resolve_success	Passed
TestCultureInspectorUnit::test_check_list_null	Failed
TestCultureInspectorUnit::test_check_list_format	Failed
TestCultureInspectorUnit::test_success	Passed
TestIntentManager::test_detect_intent_report	Failed
TestIntentManager::test_detect_intent_info	Passed
TestIntentManager::test_detect_intent_get_smells	Passed
TestIntentManager::test_detect_intent_get_smells_date	Failed
TestToolSelectorUT::test_setter	Passed
TestToolSelectorUT::test_run	Passed
TestLanguageHandlerUT::test_return_same_instance	Passed
TestLanguageHandlerUT::test_detect_language	Passed
TestLanguageHandlerUT::test_get_current_language	Passed
TestLanguageHandlerUT::test_replace_message	Passed
TestUtils::test_valid_link_with_valid_url	Passed
TestUtils::test_valid_link_with_invalid_url	Passed
TestUtils::test_valid_date_with_valid_date	Passed

TestUtils::test_valid_date_with_valid_date_exception	Passed
TestUtils::test_valid_date_with_invalid_date	Passed
TestUtils::test_enum_values	Passed
TestIntentResolverUT::test_resolve_intent_get_smells_and_date1	Passed
TestIntentResolverUT::test_resolve_intent_get_smells_and_date2	Passed
TestIntentResolverUT::test_resolve_intent_info_and_report1	Passed
TestIntentResolverUT::test_resolve_intent_info_and_report2	Passed
TestCsDetectorToolUT::test_execute_tool_w_date[data0-files0]	Passed
TestCsDetectorToolUT::test_execute_tool_w_date[data1-files1]	Passed
TestCsDetectorToolUT::test_execute_tool_w_date[data2-files2]	Passed

Totale test: 30 **Passati:** 26 **Falliti:** 4

2.2.2 Integration Test

Nome del Test	Esito
TestIntentResolverIntegration::test_resolve_intent_with_message	Failed
TestIntentResolverIntegration::test_resolve_intent_with_entities	Failed
TestIntentResolverIntegration::test_resolve_intent_invalid_request	Passed
TestIntentResolverIntegration::test_resolve_intent_exception	Passed
TestCultureInspectorIntegration::test_check_list_null	Failed
TestCultureInspectorIntegration::test_check_list_format	Failed
TestCultureInspectorIntegration::test_success	Failed
TestToolSelectorIT::test_setter	Passed
TestToolSelectorIT::test_run	Passed
TestIntentResolverIT::test_resolve_intent_get_smells_and_date1	Passed
TestIntentResolverIT::test_resolve_intent_get_smells_and_date2	Passed
TestIntentResolverIT::test_resolve_intent_info_and_report1	Passed
TestIntentResolverIT::test_resolve_intent_info_and_report2	Passed

Totale test: 13 **Passati:** 8 **Falliti:** 5

2.2.3 System Test

Nome del Test	Esito
RF1_get_smells::test_tc_GS1	Passed
RF1_get_smells::test_tc_GS_2	Passed
RF1_get_smells::test_tc_GS_pass	Passed
RF1_get_smells_by_date::test_GSBD_fail	Failed
RF1_get_smells_by_date::test_GSBD_pass	Passed
RF3_info::test_tc_IN1	Passed
RF3_info::test_tc_IN_PASS	Passed
RF4_geo_dispersion_tool::test_compute	Passed

Totale test: 8 **Passati:** 7 **Falliti:** 1

2.2.4 Copertura del Codice

L'analisi della **branch coverage** sui test preesistenti mostra risultati soddisfacenti con una branch coverage totale del 58%. Alcuni file, come `tool_selector.py` e `utils.py`, raggiungono il 100% di copertura dei rami condizionali, indicando una validazione completa delle logiche interne. Anche i moduli `language_handler.py` e `intent_web_service.py` presentano valori superiori al 90%, confermando un'adeguata qualità del testing condotto. Alcuni moduli meno recenti o deprecati risultano invece non coperti (es. `oauth.py`, `custmException.py`) e potrebbero essere oggetto di rimozione o refactoring.

File	Stmt	Miss	Branch	Part	Cov.%
src\intent_handling\cadocs_intent.py	7	0	0	0	100.0
src\intent_handling\tool_selector.py	15	0	0	0	100.0
src\service\utils.py	22	0	4	0	100.0
src\service\language_handler.py	24	0	4	1	96.0
src\intent_web_service.py	42	2	8	2	92.0
src\intent_handling\intent_resolver.py	27	2	12	2	90.0
src\chatbot\intent_manager.py	23	1	8	2	90.0
src\nlu_model\CADOCS.py	35	4	0	0	89.0
src\nlu_model\model_selector.py	15	2	0	0	87.0
src\intent_handling\tool_strategy.py	6	1	0	0	83.0

src\intent_handling\tools.py	33	7	4	1	78.0
src\nlu_model\prediction_service.py	15	3	2	1	76.0
src\chatbot\cadocs_utils.py	38	22	8	1	41.0
src\service\cadocs_messages.py	125	87	66	7	27.0
src\intent_handling\custmException.py	14	14	4	0	0.0
src\oauth\oauth.py	30	30	2	0	0.0
Totale	**491**	**175**	**122**	**17**	**58.0**

2.3 Category Partitioning Test di Sistema

In questa sezione vengono presentati i test case e i relativi test frame associati ai test di sistema preesistenti, progettati secondo la tecnica del *Category Partitioning*. Le informazioni qui riportate sono state ricavate direttamente dalla documentazione tecnica lasciata dagli sviluppatori precedenti.

2.3.1 RF01-Info

Descrizione	
Il sistema deve interpretare correttamente una richiesta di tipo info	
Parametro: richiesta	
Nome Categoria	**Scelte**
Intent [IR]	1. Tipo intent $\neq$ info **[ERROR]**
	2. Tipo intent = info

Test Case ID	Test Frame	Risultato
TC_IN_1	IR1	Errore: la richiesta non è interpretata correttamente
TC_IN_2	IR2	Corretto

2.3.2 RF02-Get Smells

Descrizione	
Il sistema deve interpretare correttamente una richiesta di tipo get_smells	
Parametro: richiesta	
Nome Categoria	**Scelte**
Intent [IR]	1. Tipo intent ≠ get_smells **[ERROR]**
	2. Tipo intent = get_smells [Property IR_OK]
Parametro: Link	
Formato: `(?:https:)?(?www\.)?github.com/([a-zA-Z0-9_]+)(/[a-zA-Z0-9_]+)`	
Nome Categoria	**Scelte**
Formato [FL]	1. Formato link ≠ true **[ERROR]**
	2. Formato link = true [Property IR_OK]

Test Case ID	Test Frame	Risultato
TC_GS_1	IR1	Errore: la richiesta non è interpretata correttamente
TC_GS_2	IR2, FL1	Errore: il formato del link non è corretto
TC_GS_3	IR2, FL2	Corretto

2.3.3 RF03-Get Smells By Date

Descrizione	
Il sistema deve interpretare correttamente una richiesta get_smells_by_date	
Parametro: richiesta	
Nome Categoria	**Scelte**
Intent [IR]	1. Tipo intent ≠ get_smells [**ERROR**]
	2. Tipo intent = get_smells [Property IR_OK]
Parametro: Link	
Formato: `(?:https:)?(?www\.)?github.com/([a-zA-Z0-9_]+)(/[a-zA-Z0-9_]+)`	
Nome Categoria	**Scelte**
Formato [FL]	1. Formato link = false [**ERROR**]
	2. Formato link = true [IF IR_OK][Property FL_OK]
Parametro: Data	
Formato: DD/MM/YYYY	
Nome Categoria	**Scelte**
Formato [FD]	1. Formato data = false [**ERROR**]
	2. Formato data = true [IF FL_OK][Property FD_OK]
Validità [VD]	1. Validità data = false [**ERROR**]
	2. Validità data = true [IF FD_OK]

Test Case ID	Test Frame	Risultato
TC_GSD_1	IR1	Errore: la richiesta non è interpretata correttamente
TC_GSD_2	IR2, FL1	Errore: il formato del link non è corretto
TC_GSD_3	IR2, FL2, FD1	Errore: il formato della data non è corretto
TC_GSD_4	IR2, FL2, FD2, VD1	Errore: la data non è valida
TC_GSD_5	IR2, FL2, FD2, VD2	Corretto

2.3.4 RF04-Get Cultural Dispersion

Descrizione	
Il sistema deve interpretare correttamente una richiesta di tipo geodispersion	
Parametro: intent	
Nome Categoria	**Scelte**
Intent [IR]	1. Tipo intent $\neq$ geodispersion [**ERROR**]
	2. Tipo intent = geodispersion
Parametro: Nazionalità	
Nome Categoria	**Scelte**
Nazionalità [NA]	1. Nazionalità NOT in csv [**ERROR**]
	2. Nazionalità in CSV
Num Partecipanti [NUM]	1. Numero Partecipanti < 0 [**ERROR**]
	2. Numero Partecipanti > 0
Parametro: GeoDispersionValue	
Nome Categoria	**Scelte**
Value [VA]	1. Value > 100 OR Value < 0 [**ERROR**]
	2. Value > 0 AND Value < 100

Test Case ID	Test Frame	Risultato
TC_CI_1	IR1	Errore: la richiesta non è interpretata correttamente
TC_CI_2	IR2, NA1	Errore: la nazionalità non ha valori di Hofstede
TC_CI_3	IR2, NA2, NUM1	Errore: il numero di partecipanti non è tollerato
TC_CI_4	IR2, NA2, NUM2, VA1	Errore: il valore per la geodispersione non è tollerato
TC_CI_5	IR2, NA2, NUM2, VA2	Corretto

2.4 Considerazioni Finali

L'analisi della suite di test preesistente nel sistema GUIDO evidenzia, nel complesso, una base solida e articolata. La presenza di test a livello di unità, integrazione e sistema, unita a una copertura del codice pari al 58%, testimonia un'attenzione già consolidata alle pratiche di verifica nelle fasi precedenti dello sviluppo. Tuttavia, l'esecuzione della suite ha messo in luce alcune criticità: 11 test su 55 non sono stati superati, con un tasso di fallimento del 20%. Gli errori sono distribuiti su tutti i livelli e richiedono un'analisi dettagliata e mirata.

Un'analisi più approfondita ha evidenziato come principale fonte di instabilità i test associati al componente *Culture Inspector*, introdotto nelle fasi più recenti di evoluzione del sistema. Le difficoltà sembrano derivare da una configurazione ambientale non documentata, il che rende oggi complicata la riproduzione accurata delle condizioni di esecuzione. Per quanto riguarda i test di sistema, progettati mediante Category Partitioning, si osservano alcune debolezze metodologiche: in particolare, la definizione delle categorie risulta talvolta eccessivamente generica o non realistica. Inoltre, dei cinque test case specificati per il requisito RF04, solo uno risulta effettivamente implementato, il che solleva dubbi sulla coerenza tra documentazione e implementazione.
Nonostante le problematiche evidenziate, la suite rappresenta un punto di partenza valido per le future attività di regression testing. I test funzionanti saranno mantenuti come riferimento, mentre quelli falliti o incompleti verranno corretti e integrati, in parallelo con lo sviluppo dei nuovi casi di test. Infine, si raccomanda di preservare la coerenza metodologica già adottata (white-box per i test di unità, bottom-up per quelli di integrazione, black-box con category partitioning per quelli di sistema) al fine di garantire uniformità e tracciabilità all'interno del processo di verifica e validazione.

CAPITOLO 3

Strategie di Test

La presente sezione descrive le strategie di testing che verranno adottate nel contesto dell'attività di manutenzione evolutiva del sistema GUIDO. L'obiettivo è garantire la verifica e la validazione delle nuove funzionalità introdotte, assicurando al contempo la continuità operativa del sistema e la non regressione delle componenti preesistenti. Le strategie saranno articolate su più livelli, in coerenza con il modello classico a livelli del software testing: testing di unità, testing di integrazione e testing di sistema.

Per mantenere la coerenza metodologica e facilitare l'integrazione dei nuovi test con quelli già presenti nel sistema, le decisioni relative agli strumenti da adottare e alle tecniche di progettazione dei test si baseranno anche sulle scelte già effettuate nelle versioni precedenti. Ciò consente di evitare discontinuità operative e ridurre il rischio di conflitti metodologici, garantendo una maggiore uniformità nell'esecuzione e nella manutenzione futura della suite di test.

3.1 Testing di Unità

Il testing di unità ha lo scopo di verificare il corretto funzionamento delle singole unità del software, intese come le più piccole componenti testabili in modo isolato, tipicamente funzioni o classi. Questo livello di test è fondamentale per intercettare errori logici e comportamenti anomali già nelle prime fasi di sviluppo, contribuendo a migliorare la qualità complessiva del sistema.

Nel contesto della presente attività di manutenzione evolutiva, i test di unità saranno progettati con un approccio **white-box**, che consente di valutare il comportamento interno delle componenti a partire dalla conoscenza del codice sorgente. Verranno quindi sviluppati nuovi test per coprire le funzionalità introdotte attraverso le Change Request, con particolare attenzione alla logica dei moduli sviluppati per l'integrazione del tool TOAD. Parallelamente, i test di unità già presenti nel sistema ma che risultano non funzionanti verranno opportunamente analizzati, corretti e integrati all'interno della nuova suite, in modo da garantire continuità nella verifica delle funzionalità preesistenti.

Per l'implementazione e l'esecuzione dei test verrà utilizzato il framework **pytest**, già adottato all'interno del sistema GUIDO. Verranno inoltre impiegate librerie di supporto per la creazione di mock e stub, ove necessario, al fine di isolare le unità testate e simulare il comportamento delle dipendenze esterne.

3.2 Testing di Integrazione

Il testing di integrazione si concentra sulla verifica del corretto funzionamento delle interazioni tra più unità software, con l'obiettivo di identificare eventuali anomalie legate alla comunicazione tra moduli o alla condivisione dei dati. A differenza del testing di unità, che isola le singole componenti, il testing di integrazione mira a validare i percorsi logici e funzionali che emergono dall'accorpamento progressivo dei componenti del sistema.

In linea con le strategie già adottate all'interno del progetto GUIDO, verrà seguito un approccio **bottom-up**, che prevede l'integrazione progressiva dei moduli a partire da quelli di livello più basso, fino ad arrivare ai componenti più complessi e interconnessi. Questa strategia è particolarmente adatta nel contesto attuale, in quanto consente di verificare prima le dipendenze fondamentali e poi la loro orchestrazione nei livelli superiori.

I test di integrazione verranno sviluppati principalmente per verificare il corretto inserimento delle nuove componenti collegate al tool TOAD, con particolare attenzione all'invocazione dei servizi, alla gestione delle risposte e all'elaborazione dei dati restituiti. Ugualmente ai test di unità, anche i test di integrazione già presenti nel sistema e che attualmente risultano non funzionanti verranno opportunamente corretti, aggiornati e integrati all'interno della nuova suite di test.

Anche in questo caso verrà utilizzato il framework **pytest**, che consente di organizzare i test in maniera modulare ed estendibile. Per simulare il comportamento delle componenti esterne o ancora non disponibili, verranno impiegate librerie per la *mocking* e la *patching*, al fine di isolare le porzioni di sistema sotto test.

3.3 Testing di Sistema

Il testing di sistema rappresenta il livello più alto del processo di verifica del software e ha l'obiettivo di valutare il comportamento del sistema nel suo complesso, considerando le funzionalità implementate e il modo in cui esse vengono percepite dall'utente finale. Questo tipo di testing viene effettuato su un sistema completamente integrato e si concentra sull'osservazione dell'output in risposta a specifici input.

Per l'attività corrente verrà adottato un approccio **black-box**, con particolare riferimento alla tecnica del **category partitioning**. Tale tecnica prevede l'individuazione delle categorie rilevanti degli input e la definizione delle relative combinazioni significative, da cui derivare i test frame necessari a coprire in modo sistematico i casi d'uso funzionali del sistema. Questo metodo consente di mantenere una copertura ragionata dello spazio delle possibilità, riducendo il rischio di omettere condizioni critiche.

I test di sistema saranno implementati utilizzando il framework **Selenium**, che consente l'automatizzazione dell'interazione con l'interfaccia grafica del sistema, riproducendo le azioni dell'utente e verificando la correttezza delle risposte visuali e dei dati restituiti. L'uso di Selenium è in continuità con le scelte progettuali già adottate in precedenza.

Durante la fase di progettazione e sviluppo dei nuovi casi di test, sarà inoltre oggetto di valutazione la qualità del category partitioning già presente nei test di sistema esistenti. In diversi casi, infatti, la definizione delle categorie risulta non del tutto ottimale. Verrà quindi considerata la possibilità di migliorare tali test attraverso una ridefinizione più precisa delle partizioni e l'implementazione dei test frame mancanti, compatibilmente con le risorse temporali disponibili.

3.4 Testing di Regressione

Il testing di regressione ha lo scopo di verificare che le modifiche introdotte nel sistema, sia sotto forma di nuove funzionalità che di correzioni, non compromettano il comportamento corretto delle componenti esistenti. Si tratta quindi di un'attività fondamentale per preservare la stabilità complessiva del sistema software nel tempo.

Nel caso specifico del sistema GUIDO, verrà adottata una strategia di tipo **TEST ALL**, che prevede la riesecuzione completa di tutti i test già esistenti e attualmente funzionanti. Tale scelta è motivata dal fatto che il numero complessivo dei test presenti nel sistema non è eccessivo, e il tempo richiesto per la loro esecuzione è contenuto. Di conseguenza, non si ritiene necessario ricorrere a tecniche di selezione o prioritizzazione dei test, che risulterebbero sproporzionate rispetto alla scala del progetto.

Per quanto riguarda il tool TOAD, che verrà integrato in GUIDO come componente esterno, non è attualmente disponibile alcuna test suite automatizzata. Di conseguenza, per la verifica delle eventuali modifiche che si renderanno necessarie per realizzare le Change Request, verrà adottato un approccio di regressione manuale basato sull'esecuzione comparata. In particolare, TOAD verrà eseguito su un insieme selezionato di repository GitHub, e i risultati ottenuti saranno salvati e documentati. Dopo l'introduzione delle modifiche, lo strumento verrà nuovamente eseguito sulle stesse repository, e i risultati delle due esecuzioni saranno confrontati al fine di rilevare eventuali differenze non desiderate. Questo approccio, sebbene manuale, consente di preservare la coerenza comportamentale dello strumento e di verificarne la stabilità a fronte di modifiche strutturali.

CAPITOLO 4

Criteri di Accettazione

Al fine di garantire la qualità delle modifiche apportate al sistema GUIDO e la loro corretta integrazione nell'ecosistema esistente, ogni intervento sarà considerato completato e accettabile solo se soddisfa **tutti** i seguenti criteri:

- **Implementazione funzionale**: Tutte le funzionalità previste dalla Change Request devono essere correttamente implementate e operative, secondo quanto descritto nei requisiti specifici.
- **Verifica unitaria e di integrazione**: I test di unità e di integrazione devono essere stati sviluppati, eseguiti e superati con esito positivo.
- **Validazione del sistema**: I test di sistema devono dimostrare il corretto funzionamento delle funzionalità in scenari d'uso realistici e l'interazione fluida tra GUI, GUIDO e TOAD.
- **Assenza di regressioni**: L'intera suite di test preesistenti (unitari, di integrazione e di sistema) deve essere eseguita e completata con successo, senza introdurre regressioni o malfunzionamenti nelle funzionalità già consolidate.
- **Conformità ambientale (Docker)**: Qualora la Change Request impatti sull'ambiente di esecuzione (es. dockerizzazione), il sistema deve risultare completamente operativo anche nella nuova configurazione containerizzata.
- **Copertura del codice**: La *branch coverage* complessiva del progetto, misurata mediante strumenti automatici (es. `coverage.py`, `pytest-cov`), deve essere **almeno pari al 75%**. Tale soglia garantisce che la logica condizionale interna sia sufficientemente esercitata dai test automatizzati.