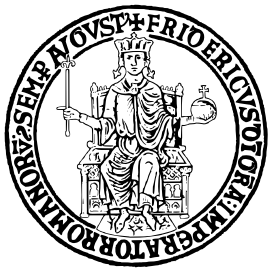
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Triennale in Informatica

Morfify: Un ambiente per metamorphic testing di applicazioni web

Relatore
Professor Adriano Peron

Correlatore
Dott. Luigi Libero Lucio starace

Candidato
Ivan Capasso
N86002587

Anno Accademico 2019–2020

Prefazione

Sono un ragazzo di ventidue anni con una passione a 360° per l'informatica fin da quando ero bambino. Non ho esperienze lavorative pregresse, solo progetti amatoriali di piccole entità; i più importanti, riguardano hosting e sviluppo di server pubblici di svariati videogiochi online.

Non volevo basare la tesi, nonché l'intera attività di tirocinio, a un argomento non di mio interesse. Essendo il dominio dei Web Services non molto trattato dal CdL, il tirocinio proposto riguardante questa tecnica di Metamorphic Testing a me sconosciuta, applicata nel dominio delle Web Application, ha catturato subito l'attenzione.

Questo studio è destinato a tutti i developer di Web Application che sono alla ricerca di una tecnica di testing per untestable tests, con annesso ambiente in cui usarla.
Il lavoro inerente a Morfify è stata svolta durante l'attività di tirocinio interna, il restante per la stesura della tesi.

Ringrazio il professore Adriano Peron, relatore di questa tesi di laurea, oltre che per l'aiuto fornitomi in tutti questi anni e la conoscenza che mi ha donato, per la disponibilità e precisione dimostratemi durante tutto il periodo di stesura.
Ringrazio il mio correlatore Luigi Libero Lucio Starace, sempre disponibile per ogni piccolo problema che gli si presentava d'avanti, sia durante il periodo di tirocinio che di stesura della tesi.

Ringrazio tutte le persone che mi sono state vicine durante tutto il percorso accademico.
A tutti quelli che hanno creduto in me fino alla fine, posso finalmente dire "Ce l'ho fatta".

Abstract

Morfify è un ambiente di metamorphic testing per applicazioni web.
Utilizza la tecnica di Metamorphic Testing per testare potenzialmente tutte le funzionalità di qualsiasi programma, partendo semplicemente da un caso di test (Source test case) registrato con Selenium IDE.

Grazie all'utilizzo di una Metamorphic Relation, ovvero una proprietà espressa in codice di una funzionalità del programma previsto, è in grado di creare dei test case con leggere variazioni sull'input, chiamati Follow-up test case e di verificare se questi test siano "simili" in termini di contenuti delle pagine visitate durante la loro esecuzione.

Supponiamo di dover testare un e-commerce di una catena di supermercati, l'utente accede alla homepage in cui è presente una form per ricercare gli articoli in base al nome e può filtrare in base alla categoria.
L'utente effettua una ricerca per nome scrivendo "pasta",senza filtri; sono usciti 462 prodotti corrispondenti alla ricerca, come faccio a sapere se l'output è corretto? manca qualche tipo di pasta nell'insieme dei risultati? nell'insieme sono presenti prodotti che non sono paste? Rispondere con esattezza a queste domande impiegherebbe un grande quantitativo di tempo.

Questo è un chiaro esempio di problema dell'oracolo, una problematica che coinvolge casi di test che non hanno un oracolo noto, ovvero un meccanismo esterno all'applicativo usato per determinare se un test ha avuto successo oppure è fallito.

Grazie a Morfify, si è in grado di eseguire un primo caso di test (Source) con la form sopra citata e di andare a creare ed eseguire altri casi di test (Follow-Up) con variazioni sull'input dettate dalla Metamorphic Relation che si è deciso di usare, anch'essa parte integrante dell'input di Morfify.
Il test originario verrà confrontato con quello dei suoi cloni e se risulteranno "quasi simili" tra di loro il Source test case avrà avuto successo, altrimenti è fallito.

In questa tesi di laurea vedremo nel dettaglio cos'è la tecnica di Metamorphic Testing, cos'è una Metamorphic Relation, come si costruisce, quanto e dove viene utilizzata questa tecnica dalle realtà software.
Vedremo com'è stata costruita Morfify, esamineremo le sue componenti e le tecnologie che utilizza.

Infine vedremo cos'è in grado di fare Morfify con un esempio d'uso di una web application di sample chiamata PetClinic per poi concludere con la verifica degli obbiettivi posti e cosa andrà migliorato di Morfify con degli sviluppi futuri.

Indice

Lista delle figure

Elenco delle abbreviazioni

- MT: Metamorphic Testing
- MR: Metamorphic Relation
- SQA: Software Quality Assurance
- DBMS: DataBase Manengment System
- JPA: Java Persistence API
- Source: caso di test sorgente
- Follow-Up: caso di test Follow-Up
- DSL: Domain Specific Languange
- JSON: JavaScript Object Notation
- CRUD: Create,Read,Update,Delete

Introduzione

Lo studio si propone come una presentazione di un framework di test automatico a livello di interfaccia utente chiamato Morfify, un ambiente di metamorphic testing per applicazioni web.
Per farlo, partiremo dal definire che cos'è la tecnica di Metamorphic Testing, quando va usata e, esaminando vari articoli in letteratura, quali sono le realtà software e le funzionalità di un programma che beneficiano nell'utilizzo di questa tecnica.

Successivamente spiegheremo cos'è Morfify, quali sono le sue componenti principali e come comunicano tra di loro. Infine, vedremo un esempio d'uso di quest'ambiente e cos'è in grado di fare Morfify, nonché quali feature potrebbero essere sviluppate in futuro.

Le motivazioni che mi hanno spinto ad approfondire la tecnica di Metamorphic Testing e a svilupparci un ambiente di testing sono di varia natura.
Innanzitutto Morfify era già stato sviluppato come attività di tirocinio interno, sempre svolto con il professore Adriano Peron.

L'interesse in quest'attività è stato sicuramente influenzato da un interesse personale verso i web services, le web application e il loro sviluppo software.

Dopo essersi documentati sui vari ambienti di testing utilizzati in software B2B, sia manuali che automatici, si era arrivati alla conclusione che nonostante la tecnica di MT risulti particolarmente efficace per determinate azioni che si possono svolgere in una web application, non ci fosse un ambiente di testing che utilizzasse la tecnica di MT.

Si è deciso di riempire questo vuoto: Morfify si vuole proporre come un ambiente di testing valido ed efficace per applicazioni web.
L'elaborato, in questo modo, mira ad approfondire e a conoscere la tecnica di MT, per poi applicarla in domini di applicazione anche diversi da quelle delle applicazioni web, dato che, si ritiene che la tecnica è molto flessibile e può variare forma e contenuto in base al tipo di applicativo su cui viene usata, senza perdere di efficienza.

Gli obbiettivi della tesi sono presentati sotto forma di domande:

- Quali sono i vantaggi nell'uso della tecnica di Metamorphic Testing rispetto ad altre tecniche di testing?
- Nel dominio delle Web Application, quant'è utile la tecnica di Metamorphic Testing?
- Cos'è Morfify? Quali sono le realtà software che potrebbero usarlo e perché.

Si darà una risposta a queste domande durante tutto l'elaborato, in particolare si risponderà esplicitamente ad ogni singola domanda nel capitolo conclusivo.

La tesi è articolata in cinque capitoli:

- Nel primo capitolo esamineremo il contesto e descriveremo il problema che ci vogliamo porre. Per farlo, spiegheremo cos'è un caso di test, quali ci interessano e in che modo possono essere generati.
  Spiegheremo cos'è la tecnica di Metamorphic Testing e cos'è una Metamorphic Relation, con tanto di esempi.
  Infine analizzeremo dei dati raccolti in letteratura e ne trarremmo conclusioni su dove quando e come viene usata la tecnica di Metamorphic Testing e quando quest'ultima risulta essere limitante.
- Nel secondo capitoli esamineremo le tecnologie utilizzate, prime tra tutte Selenium, un framework portatile per testare applicazioni web.
  Fornisce sia uno strumento di riproduzione per la creazione di test funzionali, ovvero L'API

Selenium WebDriver, ma anche uno strumento di capture & replay di casi di test, ovvero Selenium IDE.

- Nel terzo capitolo presenteremo Morfify, la nostra soluzione al problema posto.
  Mostreremo l'architettura del sistema e sulla sua creazione. Mostreremo le 5 componenti principali di cui è composto e sviluppato il nostro ambiente di testing: Creazione Source test case, Creazione Follow-up, esecuzione di test case, raccolta delle pagine e verifica.

- Nel quarto capitolo useremo un web application di sample chiamata PetClinic come esempio d'uso su una qualsiasi web application che vuole usare Morfify come ambiente di testing (Anche solo parziale).
  Per valutare l'efficacia di Morfify, sono stati inseriti manualmente dei fault realistici nel codice dell'applicazione web considerata. Quindi, si è proceduto all'esecuzione del tool per verificare se i fault venivano rilevati.

- Nel quinto ed ultimo capitolo infine faremo delle conclusioni rispondendo esplicitamente alle domande poste precedentemente.
  Faremo anche delle critiche verificando quali obbiettivi sono stati soddisfatti dalla premessa di Morfify e quali sono le funzionalità di quest'applicativo che potrebbero essere sviluppate in un futuro prossimo.

1 Descrizione del problema

In questo capitolo esamineremo alcuni concetti fondamentali nel settore del testing.
Nella prima parte definiremo cosa si intende per software testing, caso di test, quando e in che modo si intende rendere questi test automatici utilizzando la tecnica di metamorphic testing; per poi spiegare cos'è una Metamorphic Relation, con esempi e guide sulla creazione.

Nella seconda parte analizzeremo dati raccolti in letteratura riguardanti la tecnica di Metamorphic testing e della sua applicazione a seconda del dominio di applicazione.
Vedremo quindi in che modo viene usata la tecnica di MT da realtà software reali e ne trarremmo informazioni sull'efficacia e sulle limitazioni che la tecnica può avere.

1.1 Piramide di testing

Il software testing prevede l'esecuzione di una componente software per valutare una proprietà.
In generale, queste proprietà vengono definite per verificare se una componente:

- soddisfa i requisiti di sistema, accordati in fase di progettazione;
- sia installata ed eseguita sull'ambiente di sviluppo su cui si è progettata;
- risponda correttamente agli input dati;
- sia facile e veloce da usare.

In tutti i processi di testing si utilizzano delle strategie per selezionare dei test adeguati alle risorse e al tempo a disposizione.

In contesti lavorativi con Software Quality Assurance sempre più ferree e, in generale, realtà software sempre più complesse, è necessario definire un'architettura su come costruire, collaudare e incorporare i diversi test previsti per le funzionalità di un programma.
Definiamo la cosiddetta piramide dei test, uno schema per definire un ordine sia temporale che di importanza su quali tipi di test bisogna svolgere, andando a creare una gerarchia di test multistrato.

Lo scopo del test multistrato è catturare diversi tipi di problemi. Ogni tipo di test si rivolge a diversi livelli dell'intero sistema software e verifica il comportamento all'interno di tale intervallo.

Un test quindi, può essere suddiviso nei seguenti livelli:

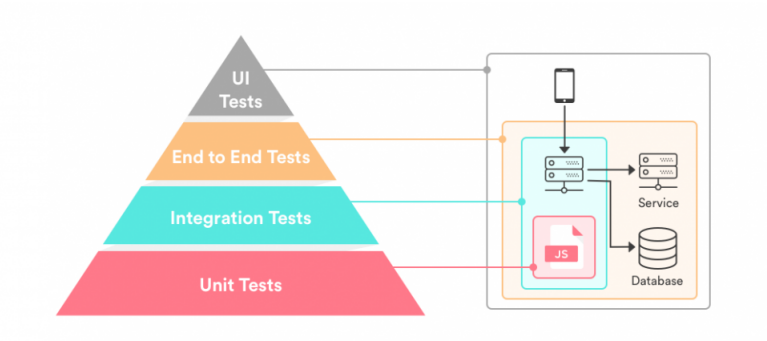


Fig. 1: Pyramid testing

- Unit test (test d'unità): serve per collaudare piccole unità software, ovvero le più piccole parti di un programma (modulo) indipendenti tra di loro.
  Utilizzando i test d'unità, è possibile modificare facilmente il codice del modulo in un secondo momento (refactoring) e assicurarsi che il modulo continuerà a funzionare normalmente.

Il processo consiste nello scrivere casi di test per tutte le funzioni e/o metodi in modo che se la modifica non supera il test, la modifica responsabile può essere facilmente identificata.

- Integration test(test di integrazione):Test atti a verificare il comportamento esterno al singolo servizio, ovvero un unione di uno o più moduli.
  Il framework di test avvia l'istanza del servizio, quindi utilizza l'interfaccia del servizio esterno per eseguire la logica di business del servizio allo stesso modo del servizio esterno integrato e del servizio testato. Ad esempio, il servizio API REST verrà testato effettuando una richiesta HTTP al servizio.

- Test End-to-end: Verifica il flusso e il corretto funzionamento dell'intero programma dall'inizio alla fine di un servizio richiesto.
  Questo metodo include il test delle interfacce esterne e delle dipendenze, come l'ambiente o il back-end in cui viene eseguito. Attraverso i test end-to-end, non solo le proprietà funzionali possono essere verificate, ma anche le prestazioni o l'affidabilità.
  Non svolge testing sull'interfaccia utente, ma banalmente la usa per verificare il successo (Success) del caso di test.

- UI Testing (Test sull'interfaccia grafica): verificano il comportamento dell'intera piattaforma a partire dall'interfaccia utente del client.
  Questa ultima tipologia di test si occuperà della logica del client e del sistema di back-end per controllare se il client può comunicare con il sistema di back-end e se lo stato tra questi due sia corretto e aggiornato. Nei capitoli successivi, quando ci riferiremo a casi di test e framework di test, facciamo riferimento a testing end-to-end, essendo quello generato da Selenium, nonché dall'applicativo da noi proposto.

1.2 Necessità di test automatici

Un articolo di Oracle " *When to automate your testing and when not to*"[4] indica che secondo un'indagine aziendale il 75% delle società scrive i suoi test funzionali manualmente.
I motivi principali, aggiunge l'articolo, riguardano il tempo, sia per istruire i developer, sia per gli analisti nell'imparare nuovi approcci alla fase di testing.

Si aggiunge uno sforzo iniziale nell'apprendere nuovi tool e un'indifferenza da parte di analisti e QA verso questi strumenti, il cui uso precluderebbe lavoro precedentemente svolto con framework di testing tradizionali.

Innanzitutto bisogna chiarire che non sempre il Testing automatico è migliore di quello manuale.
Immaginiamo applicativi molto complessi: per progettare casi di test automatici, a prescindere dalla tecnica utilizzata, occorrono persone con forte conoscenza del dominio, molto più di quanto occorra per test manuali.
In una circostanza simile, è facile immaginare che i tempi di progettazione superano i benefici, rendendo i test automatici una cattiva scelta; altro caso riguarda la creazioni di nuovi software: non è una buona idea usare test automatici per applicativi sotto sviluppo, poiché si dovrà sempre modificare sempre qualcosa nello script di automazione.

Invece, applicativi già lanciati, con casi di test semplici,ripetitivi o con grandi quantità di input (Nel caso di applicazioni web: svariati tipi di form, ricerche, applicazione di filtri etc...),possono essere ottimi candidati per usare test automatici.
Inoltre, se l'applicativo è disponibile per vari browser o sistemi operativi, allora la quantità di test da eseguire per ogni modifica aumenta notevolmente.Usando i test automatici invece si devono modificare pochi parametri tra un test per piattaforme diverse e può essere avviato un numero illimitato di volte senza costi aggiuntivi.

Usare i test automatici, quando ci si ritrova nei casi sopra descritti, significa facilitare la manutenzione

del software, rilasciare aggiornamenti più sicuri e affidabili, riducendo notevolmente le possibilità a lungo termine di trovare bug da parte del cliente (essendo testing end-to-end).

1.3 Il problema dell'oracolo

Altro aspetto da tenere in considerazione per scegliere tra test automatici e manuali è la disponibilità o meno di un oracolo noto.
L'output prodotto dall'esecuzione di un test case, infatti, deve essere dato ad un oracolo: un meccanismo usato per determinare se un test ha avuto successo o è fallito.

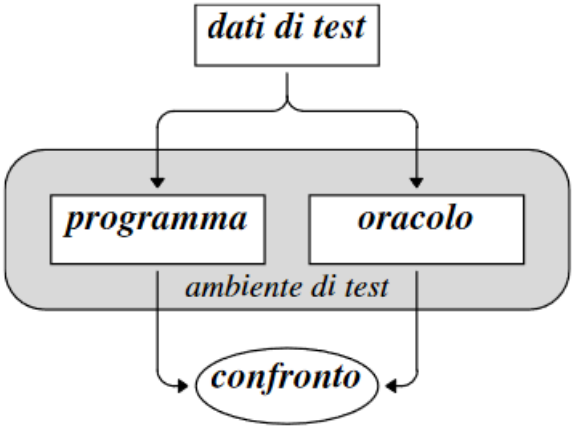


Fig. 2: Ruolo dell'oracolo durante l'esecuzione di un test case

Solitamente sono separati dall'applicativo da testare e può presentarsi sotto tante forme, tra cui:

- Specifiche di sistema, documentazione etc...
- Software di Oracolo: Programmi simili a quello da testare ma con diversi algoritmi e implementazioni al suo interno, l'applicativo da testare deve produrre lo stesso risultato del software di oracolo.
- Oracolo Statistico: Utilizza dati statistici del problema in esame del test.
- Oracolo basato sui modelli (model-based testing): utilizza modelli di testing per generare e verificare il comportamento del sistema.
- Banalmente, il giudizio di un essere umano.

Per ogni test case quindi, bisognerebbe avere un oracolo: non in tutte le realtà software questo è possibile, basti pensare a realtà complesse come motori di ricerca, sistemi di machine learning e simulatori vari, in che modo potrò stabilire il "Successo" del mio test?

Un altro esempio, potrebbe essere la ricerca di elementi all'interno di una web application: usando una ricerca tramite stringhe, effettuo un'interrogazione del DB ed esso mi restituisce una query con N risultati, come possiamo essere certi che i valori restituiti dal DB sono corretti?

Se l'oracolo fosse noto, riusciremmo a creare un meccanismo esterno al nostro applicativo che sà a priori il risultato corretto. Di conseguenza, sarebbe facile implementare un caso di test che verifica se il valore restituito dal DB equivalga a quello dell'oracolo.
Purtroppo la ricerca di un oracolo non può risultare sempre una buona scelta, non è detto che si conosca il risultato corretto per qualunque input possa essere dato al sistema.

Questo genere di problemi prendono il nome di "problema dell'oracolo" e i test case che ne sono afflitti prendono il nome di test non verificabili (untestable tests).
Per definizione, un test non verificabile non ha un oracolo, o quanto meno esso non è noto, questo rende impossibile definire se il loro output sia "Success" o "Fail".

Quella che proponiamo nel paragrafo seguente è una delle tante soluzioni proposte in letteratura per la risoluzione di questi test.

1.4 Metamorphic Testing

Presentiamo la tecnica di Metamorphic Testing (MT), un approccio al software testing per alleviare il problema dell'oracolo per i test non verificabili.
La tecnica consiste nel concentrarsi non sull'output di un singolo test, come farebbe un oracolo, ma sull'output prodotto dall'esecuzione di più test.

Questi test devono soddisfare delle condizioni atte a verificare la funzionalità del programma: se le condizioni sono soddisfatte il test originario ha avuto successo (Success), altrimenti è fallito (Faulty).
La tecnica per poter funzionare prende in input un test già esistente, ovvero il Source test case e l'insieme delle condizioni che l' esecuzioni successive devono rispettare, ovvero le Metamorphic Relation.

Approfondiremo le relazioni metamorfiche nei due capitoli successivi, limitiamoci a dire che trasformano il test originario (Source test case) in tanti nuovi test, chiamati Follow-Up test cases.
I Follow-Up test cases sono molto simili al test originario, tant'è vero che essi deve testare la stessa funzionalità del Source, ma ciò che li distingue sono delle leggere variazioni sul loro input.
Esempi inerenti al dominio delle Web application potrebbero essere: Browser utilizzato, inserire una stringa casuale all'interno di una form, cliccare il pulsante Y invece che X, aprire un link in una nuova scheda del browser.

In linea di massima, vorremmo che le modifiche di questi Follow-Up test cases non cambiassero di molto, se non addirittura per nulla, l'output del Source Test case.
Ci sono anche casi in cui vorremo che l'output fosse completamente differente, dipende da come è stata scritta la metamorphic relation.

Per l'implementazione, si può usare qualsiasi unit-testing-framework, come ad esempio JUnit in ambiente Java: Si inizia con la scritta del caso Source Test Case e si procede con la verifica della relazione.

Quello che vedremo con la nostra soluzione, che si focalizza sul dominio delle web application, è un ambiente di sviluppo indipendente da framework di testing.

1.5 Metamorphic relation

Una relazione metamorfica (MR) è una proprietà necessaria di una funzionalità del programma previsto.
Può essere espressa come statement, in una prima fase di identificazione delle relazioni, oppure come costrutto matematico (Applicativi numerici). Una Metamorphic relation è divisa in:

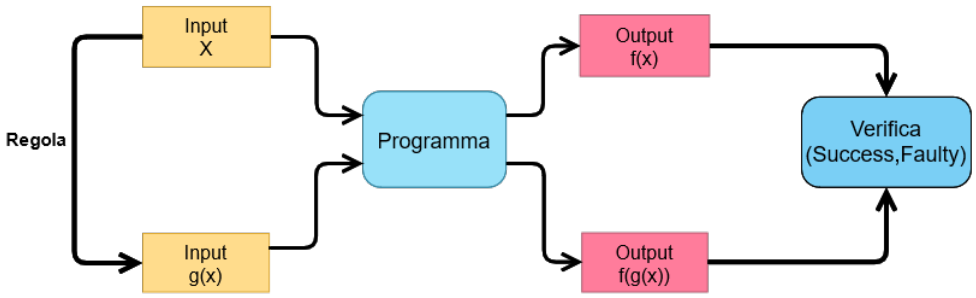


Fig. 3: Applicazione di una Metamorphic Relation

- Regola : E' la parte della relazione che si occupa di modificare il Source test case, generando dei test (Follow-Up test cases) con leggere variazioni sull'input.
- Verifica :Viene interpellata dopo l'esecuzione dei Source e Follow-Up test cases, si occupa di stabilire, se il test ha avuto successo (Success) oppure è fallito (Faulty).

In una relazione metamorfica la regola può anche non esserci (Eseguire multiple volte lo stesso test per verificare una coerenza temporale, per esempio) mentre la verifica è obbligatoria.

1.5.1 Esempi

Consideriamo come esempio il programma MAX(a,b,c), che prende in input a,b,c array e restituisce il massimo elemento in comune a tutti gli array (se non presente, restituisce null). Escludiamo l'uso di un oracolo per verificare l'output del programma e cerchiamo di creare una MR. Una prima relazione potrebbe riguardare l'ordine degli input, quindi verificare:
MAX(a,b,c)=MAX(a,c,b)=MAX(b,a,c)=MAX(b,c,a)=MAX(c,a,b)=MAX(c,b,a).
Quindi, immaginiamoci una prima esecuzione del test con input a=[1,2] e b=[2,3] c=[2,4]; si calcola MAX([1,2],[2,3],[2,4]) (Source) e subito dopo si crea un Follow-Up che calcola MAX([1,2],[2,4],[2,3]).
se la relazione è violata, ovvero se MAX([1,2],[2,3],[2,4])!= MAX([1,2],[2,4],[2,3]), possiamo essere sicuri che il test non ha avuto successo (Faulty).

Si continua con la creazione dei follow-up test cases restanti, che dovranno essere generati a partire dalla regola (es: continua le permutazioni).
Se la verifica è stata soddisfatta nell'esecuzione di tutti i Follow-up, allora il test originario (Source) ha avuto successo (Success).

Abbiamo creato una Metamorphic relation MR1 definita in questo modo: Regola: Modificare l'ordine dell'input del programma in un elemento casuale dell'insieme delle permutazioni (x,y,z).
Verifica: MAX(a,b,c)=MAX(x,y,z). Un altro esempio più inerente al nostro dominio di applicazione potrebbe riguardare l'esempio generico trattato in 1.3.
Supponiamo di dover testare un e-commerce di una catena di supermercati.

L'utente accede alla homepage in cui è presente una form per ricercare gli articoli in base al nome, può filtrare in base alla categoria, nonché sull'origine dei prodotti (Italiani oppure Esteri) e ordinare in base al prezzo.
L'utente effettua una ricerca per nome scrivendo "pasta", senza filtri aggiuntivi; sono stati trovati 462 prodotti corrispondenti alla ricerca, in che modo potrò dire se l'output è corretto? Manca qualche tipo di pasta nell'insieme dei risultati? Nell'insieme sono presenti prodotti che non sono paste?

Rispondere con esattezza a queste domande usando un oracolo potrebbe occupare un grande quantitativo di tempo con svariate interrogazioni al DB per verificare in prima battuta che il problema non sia lì, per poi controllare che il server manda la query di ricerca correttamente.

Ricordiamo che essendo testing end-to-end il nostro obbiettivo è testare server e DB del nostro applicativo: se si verificherà un Faulty non sapremmo dire dove trovare il fault seed senza usare i test di unità e di integrazione che reggono il nostro test end-to-end (Vedi 1.1).

Definiamo le seguenti Metamorphic relation, atte a verificare un test sorgente in cui si ricerca senza filtri la parola "pasta".

- MR1: Regola: rieffettuare la ricerca aggiungendo un filtro sulla categoria "Pasta e riso".
  Verifica: l'insieme dei risultati deve essere un sottoinsieme dell'insieme generato dal Source Test Case.
- MR2: Regola: rieseguire due volte la ricerca appliccando il filtro sulla provenienza (Una ricerca con filtro su prodotti made in italy, un'altra su prodotti importati).
  Verifica: I due sott'insiemi dei risultati devono costituire una partizione del l'insieme generato dal Source Test Case.
- MR3: Regola: rieseguire la ricerca ordinando gli elementi in ordine crescente e decrescente (supponiamo che di default non ci sia un' ordinamento a noi noto).
  Verifica: Gli insieme risultati sono uguali all'insieme generato dal Source Test Case.

La prima potrebbe risultare particolarmente utile nel caso in cui nei 462 prodotti del Source test case ci siano dei prodotti che sono tipi di riso (Stiamo escludendo il caso in cui un prodotto identificabile come riso contenga la parola "pasta" al suo interno...).
Gli ultimi due invece possono essere particolarmente utili per verificare sia il funzionamento di filtri e ordinamenti vari; ma anche per verificare che ognuno dei 462 prodotti del Source test case abbia effettivamente una provenienza ed un prezzo (Se viene richiesto un ordinamento sul prezzo, i prodotti senza prezzo non compaiono).

Applicativi e realtà software che utilizzano la tecnica di MT l'esaminiamo in 1.6.

1.5.2 Costruzione di una metamorphic relation

L'Efficienza nell'uso della tecnica di MT è strettamente correlata da una buona definizione delle MR. L'identificazione di queste relazioni è un lavoro che richiede creatività e conoscenza del dominio, ma ci sono delle linee guida da poter seguire durante il processo di costruzione della relazione.

Per una singola MR il numero di regole e verifiche è molto ampio, in un primo momento si può essere tentati di creare un unica relazione con tutte le componenti al suo interno.
Vedremo invece come creare la relazione in modo mirato alla risoluzione di una determinata funzionalità di un programma.

In particolare, proponiamo 3 tipi di approcci [2]:

- Basato sull'input (Input-Driven Approach): Si concentra sul numero ed il tipo di input che vengono dati al programma; un possibile cambiamento sull'input, potrebbe essere, per esempio modificarlo in base alle operazione che quel tipo di dato permette di fare.
  Quindi, se devo testare una query di ricerca, come nell'esempio precedente, potrei aggiungere, rimuovere, applicare filtri e ordinamenti vari alla mia ricerca.
  Oppure, se ho un tipo di dato riconducibile ad un'insieme (Set) potrei effettuare azioni di aggiunta, rimozione e modifica all'interno della mia collezione, prima di darlo in pasto al programma. Queste possibili modifiche forniscono indizi su come modificare i Source test case per generare nuovi casi di test di follow-up per identificare relazioni metamorfiche.
  L'approccio è usato particolarmente in ambiente matematico e in teoria dei grafi.
- Basato sull'output(Output-Driven Approach): In contrapposizione con l'approccio precedente, l'approccio basato sull'output esamina l'output generato da diverse esecuzioni dello stesso test e cerca di trarne delle proprietà basandosi sulle modifiche dell'input.

Un esempio di quest'approccio lo troviamo in operazioni di ricerca, una relazione tipica è porre una verifica che impone una correlazione tra l'insieme dei risultati di partenza e quello della fine, come per esempio la relazione MR1 dell'esempio sul sito di e-commerce.
In quel caso abbiamo imposto una verifica sul Source test case che doveva essere un sottoinsieme di quelli che venivano generati dal Follow-Up.
Approccio particolarmente utile se già si possiedono test case simili tra loro, si per programmi che accedono a repository di dati come sistemi di informazione, motori di ricerca etc...

- Generazione automatica di regole: Partendo da un test di partenza, si costruiscono test con variazioni sempre più grandi sull'input, si verifica quali sono le variabili più o meno comuni per tutti i test e ci si costruisce sopra una regola.
  La differenza con l'approccio basata sull'output e che si basa più sulle similitudini che alle differenze dell'output. Approccio tipico in contesti di Machine Learning.

In linea generale, qualsiasi relazione che si focalizza sulla regole sarà basata sull'input, viceversa, se si focalizza sulla verifica, sarà basata sull'output.

Indipendentemente dal metodo utilizzato, studi precedenti hanno suggerito che le relazioni metamorfiche dovrebbero essere diverse, il che significa che dovrebbero contenere sia approcci basati sull'input che sull'output, cercando di mantenere un equilibro tra i due approcci[9].

1.6 Domini di applicazione

La tecnica di Metamorphic Testing è stata scoperta da T.Y Chen nel 1998 [7], da allora sono state pubblicate svariate ricerche che approfondissero in che modo costruire le relazioni metamorfiche, in che modo automatizzare questo processo [9], che come abbiamo detto, richiede molta creatività e conoscenza del dominio, nonchè in che modo applicare la tecnica a seconda del dominio di riferimento.

Un'articolo pubblicato da S.Segura "Metamorphic Testing: Testing the Untestable" nel 2020 [6] ha svolto un sondaggio che ha coinvolto 84 casi di studio svolti tra il 1998 e il 2018.
Questi casi di studio riguardano non il numero di studi sulla tecnica in generale (altrimenti sarebbero molti di più), ma sull'uso e quando conviene usare la tecnica tra i vari domini di applicazione.

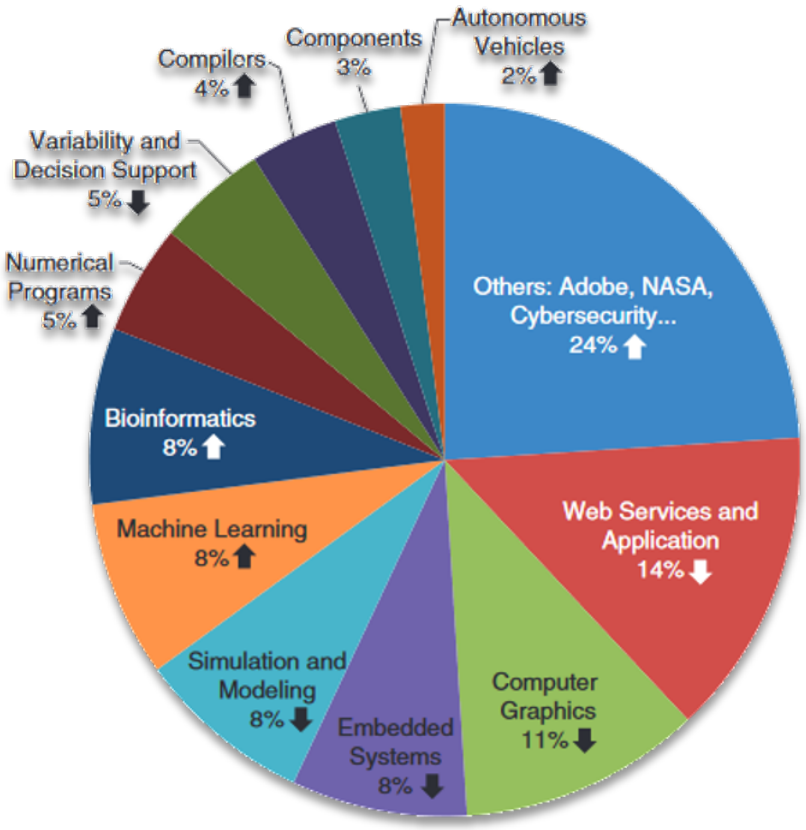


Fig. 4: Casi di studi riguardanti la tecnica di MT

I settori più popolari sono sicuramente Web application e Computer Graphics, ma c'è anche una varietà di settori come software finanziario, programmi di ottimizzazione, cybersecurity e data analytics, nonché applicazioni industriali in organizzazioni come NASA e Adobe. Il grafico lascia intuire una crescita dell'uso di queste tecniche nei settori incentrati più sull'IA, come Machine Learning, Bioinformatica e Self-Driving.

Alcuni articoli invece hanno proposto la tecnica di MT per trovare bug nei compilatori: applicando una regola per rimuovere o aggiungere codice non usato da un sorgente di un programma ed una verifica atta a confermare la non variazione dell'output.
Quest'approccio ha fatto trovare 147 bug (110 risolti) nelle prime versioni di un compilatore GCC[10].
Altro dominio particolarmente interessante è quello dell'IA, dove la relazione metamorfica, in particolare la regola, può essere usata in fase di training con i seed input per poi usare la verifica per controllare il comportamento del sistema.

Un applicativo degno di nota in quest'ambito è DeepTest [8], un tool di testing per identificare automaticamente comportamenti erronei di veicoli di guida autonoma (DNN-automated driven vehicles.

Questo tool usa l'informazioni della macchina in movimento (Immagine della strada, direzione prevista, dimensioni del veicolo,sensori vari etc...) e identifica quando la macchina assume comportamenti erronei, dalle più stupide come frenate non necessarie alle più fatali come incidenti con altri veicoli. il tool cerca di simulare tutte le condizione di guida possibili, generando automaticamente test case basati su condizioni di guida come pioggia o nebbia; condizioni dove l'auto dovrebbe comportarsi in

modo simile alla sua situazione di partenza.

Invece, sono stati trovati migliaia di azioni errate in diverse condizioni di guida realistiche, alcune delle quali avrebbero portato ad incidenti mortali.

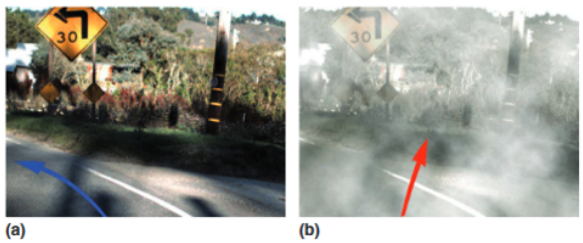


Fig. 5: Un esempio di immagine raccolta da DeepTest: a destra il test sorgente con relativa direzione della macchina, a sinistra la stessa foto ma con applicati dei filtri per simulare condizioni di pioggia o nebbia, il comportamento dovrebbe essere quasi simile invece la direzione della macchina cambia totalmente.[6]

1.7 Valutazioni sperimentali

In questo paragrafo analizziamo dati raccolti dall'articolo: "A Survey on Metamorphic Testing" sempre di S.Segura del 2016 [5].
L'articolo esamina 145 paper riguardanti la tecnica di MT, tra questi articoli , 62 presentano soluzioni software utilizzando la tecnica di MT: si tratta di problemi reali e di realtà applicative non piccole.

I restanti sono articoli di natura teorica, ma con esempi particolarmente articolati al loro interno.
Di questi programmi, sono stati raccolte informazioni sul loro uso della tecnica di MT in particolare:

- Dominio di applicazione
- integrazione con altre tecniche di testing
- numero di relazione metamorfiche proposte
- dettagli sulla valutazione
- problematiche riscontrate

Questi programmi sono stati scritti principalmente in Java (46.2%) e C/C++ (35.5 %) con righe di codice sorgente fino a 12,795.
Solitamente, una nuova tecnologia è considerata matura quando il numero di programmi "reali" (commerciali o open-source) superano quelli di ricerca: il grafico seguente mette a confronto il numero di programmi sviluppati in entrambi gli ambiti e lascia intuire che la tecnica nel corso del 2010-2015 stia ancora maturando.

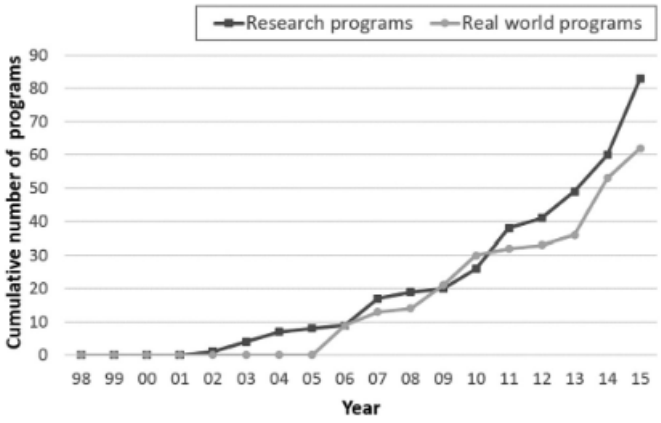


Fig. 6: Numero di articoli per anno di Ricerca VS Applicativi reali

La tecnica di MT richiede un Source test case, questo test dovrà essere usato come seed per generare dei Follow-Up Test.

I Source test cases possono essere generati usando varie tecniche di testing: Il grafico mostra che la maggioranza(57%) dei casi di studio usa la tecnica di test casuale (Random Testing, vengono generati input casuali e indipendenti e si verifica manualmente che l'output rispecchi le specifiche).

Seguono i casi di test che usano suite di test (34%), mentre altri articoli (6%) usano modelli di testing(model-based testing, testing di ricerca (search testing) o esecuzione simbolica (symbolic execution).

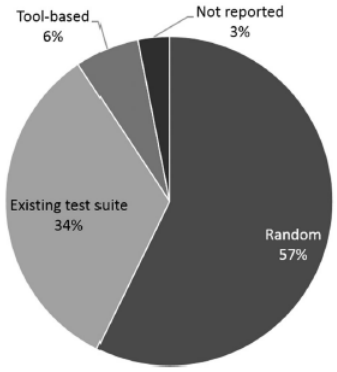


Fig. 7: Generazione di Source test case

Crediamo che la varietà di modi proposti per ottenere un Source test case dimostri L'applicabilità della tecnica di MT nelle varie realtà software.
Altro dato interessante da esaminare è il numero fault trovati, nonchè il parametro più importante per stabilire un uso corretto di un qualsiasi tecnica di testing.

I paper di ricerca dello studio in questione tendevano a creare artificialmente casi di fault (chiamati anche mutanti), sia manualmente che utilizzando tool automatici (fault generator).

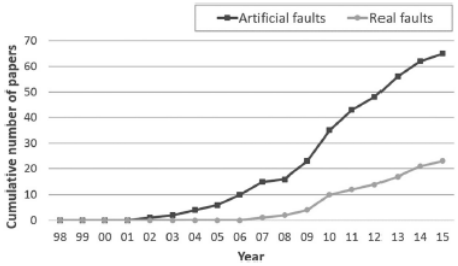


Fig. 8: Tipi di fault: Artificiali VS Reali

Il grafico seguente mette a confronto i fault trovati per così dire "reali" (non previsti, latenti, sconosciuti) e quelli artificiali: Sono stati presi in esame 36 (23 dei quali reali) programmi che hanno indivituato 295 fault distinti (Come si evince dal grafico, la maggioranza è artificiale). La tecnica di MT quindi si dimostra efficace per l'individuazione di bug.

Ultimo dato che si vuole esaminare è il numero di MR definite per testare un singolo programma: crediamo che il numero di relazioni dia un'idea del tempo e del costo richiesto per applicare la tecnica di MT.

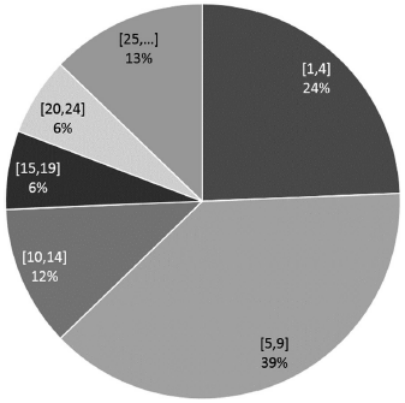


Fig. 9: Numero di Metamorphic relation

La maggior parte dei programmi tende sul costruire dai cinque ad i nove MR (39%), seguito da 1 a 4 quattro (24%): la maggior parte sono i paper di ricerca che, per forza di cose, avranno problematiche accademiche da risolvere, invece che casi reali.
Si continua con un 12% che ha definito dai 10 ad i 14 MR, un 6% dai 20 ad i 24, e infine un 13% che ha optato per 25 o più.
L'articolo ha dato una lettura approfondita a quest'ultima porzione di paper (9) ed ha scoperto che il numero così alto di Metamorphic Relation era dovuto da un'identificazione sbagliata di quest'ultime, tornando indietro, alcuni articoli dichiarano, si sarebbero fermati a 10-14 MR più complesse e ben definite.

1.8 Limitazioni

La tecnica di Metamorphic Testing e le relazioni metamorfiche si adattano bene a molte realtà software, ma non in tutti i domini potrebbe risultare una tecnica efficace e produttiva.

Innanzitutto, la tecnica non va usata in domini che hanno bisogno di certezze: ricordiamoci che il Metamorphic Testing allevia il problema dell'oracolo ma non lo risolve del tutto; in questi casi è più saggio impiegare le risorse nella ricerca di un oracolo.
Nulla vieta di integrare la tecnica con altri tipi di testing, l'importante è saper individuare i casi di test in cui sarà opportuno ricercare un oracolo e quali no.

Altra problematica è l'individuazione delle Metamorphic Relation: essendo un processo manuale lungo e dispendioso di tempo e di risorse umane (Analisti, QA, etc...) ci si può ritrovare in una situazione in cui non si è prestato abbastanza tempo nell'individuazione di una relazione, scrivendola male, o saltandone alcune.
Sono in corso diverse ricerche sulla creazione automatica di relazioni metamorfiche, ancora molto acerbe.[1] [3]

L'identificazione delle Metamorphic relation è un lavoro creativo e per tanto ha i suoi tempi: un modo per rendere più veloce questo processo è allenarsi a risolvere problemi nei vari domini di applicazione.[2]
Ultimo aspetto da tenere in considerazione sulle relazioni metamorfiche è la loro scelta: a seconda del problema che si sta risolvendo, una relazione potrebbe essere più efficace rispetto ad un'altra, ma in che modo potremmo saperlo?

Dalle limitazioni mostrate, è chiaro come il MT sia in continuo sviluppo e che alcuni suoi aspetti devono essere ancora coperti del tutto, nonostante ciò i framework di testing e realtà software che utilizzano questa tecnica esistono e continuano ad invertirci sopra.

2 Tecnologie utilizzate

2.1 Introduzione

Prima di presentare la soluzione del problema posto, esaminiamo strumenti e tecnologie utilizzate durante la creazione dell'applicativo.

Il linguaggio di programmazione scelto è stato Java, oltre per la necessità di usare un approccio ad oggetti, anche per la documentazione e materiale disponibile nel settore di Web Testing; altrettanto ottima alternativa poteva essere python per i medesimi motivi.
Come core framework è stato utilizzato Spring: Oramai essenziale per sviluppare il proprio applicativo in un ambiente di programmazione e configurazione aziendale in ambiente Java.
Permette di concentrarsi sulla business logic a livello di applicazione, senza legami inutili con ambienti di distribuzione specifici: Si adatta a tutti i contesti ed ha una libreria pronta per ogni piccolo problema.

2.2 Selenium

Il framework oggetto di studio principale della tesi è stato Selenium, un tool open source per la gestione automatizzata dei browser.

È un framework di testing ed indica una suite, composta da diversi strumenti: Selenium IDE, Selenium WebDriver, Selenium Builder,Selenium Server, Selenium Grid; nel nostro caso, sono stati utilizzati solo i primi due.
Fornisce un linguaggio di dominio (DSL), ovvero un linguaggio di specifica dedicato a problemi di un dominio specifico o a una particolare soluzione tecnica.

Permette di scrivere test per alcuni tra i maggiori linguaggi di programmazione, tra cui Java, Python, Ruby, C# e PHP.
Si tratta infine di uno strumento multi piattaforma, disponibile per Windows, Linux e Mac.

Una valida alternativa poteva essere Micro Focus Unified Functional Testing (UFT), conosciuto anche come QuickTest Professional (QTP).
È un software proprietario di HP per test automatici (Desktop e Web), comprende tutto un ambiente di sviluppo (IDE) che permette di fare test di regressione.

Selenium è più vantaggioso per le nostre esigenze in termini di costi, essendo Selenium gratuito, e flessibile, poiché in grado di girare per tutti i browser e in grado di simulare tutti i comportamenti possibili in una pagina web (Selenium IDE).
Permette l'esecuzione di test in parallelo, a differenza di QTP dove si possono eseguire solo sequenzialmente, inoltre Selenium può eseguire i test mentre è ridotto ad icona, mentre QTP utilizzando un tool basato a icone non può fare. Infine Selenium permette l'esecuzione di script in JavaScript durante la cattura ed esecuzione di un test, rendendolo utilizzabile per la maggior parte dei siti dinamici(Features non pienamente implementata nella soluzione da noi proposta).

2.2.1 Selenium Web Driver

Selenium WebDriver è una API e un protocollo che definisce un linguaggio neutro(indipendente dal linguaggio di programmazione) per controllare il comportamento di un browser. Ogni browser supporta una specifica implementazione (di terze parti, esterne a Selenium) di un WebDriver, chiamata driver.

Il driver è colui che delega le istruzioni al browser e gestisce la comunicazione da e verso Selenium.

Selenium unisce le diverse implementazioni dei driver attraverso un'interfaccia rivolta all'utente che consente di utilizzare diversi browser in modo trasparente, consentendo l'automazione cross-browser e multi piattaforma.
È importante capire che il Web Driver NON è la componente che si occupa di eseguire i test, il suo

unico scopo è quello di comunicare con il browser: per rendere Selenium WebDriver un tool di testing deve essere affiancato a un framework di testing come JUnit in ambiente Java.

2.2.2 Selenium IDE

Selenium IDE è lo strumento più user-friendly dell'intera suite: è un add-on per browser che crea in modo semplice e immediato tramite funzionalità di capture and replay un test case per i vari framework di test, tra cui JUnit in ambiente Java, NUnit in ambiente C#,pytest in ambiente Python.
Richiede conoscenze molto basilari di HTML, DOM e JavaScript, il che lo rende un ottimo candidato per creare prototipi di test case da perfezionare una volta esportato in framework di test.

Si incomincia creando un progetto dove memorizzare tutti i test case che si vorranno registrare, si crea un test case specificando l'URL sulla quale, quando si vorrà iniziare a registrare il test, si aprirà una finestra del browser sull'indirizzo specificato.
Una volta avviata la registrazione, Selenium IDE tiene traccia di ogni azione che l'utente svolge all'interno delle pagine web che sta visitando, ed una volta finita la registrazione si procede con l'esportazione sotto forma di codice che esegue i comandi replicati dall'utente in fase di registrazione sotto forma di API di Selenium WebDriver.

Quello che è stato fatto nell'applicativo per portare il test in ambiente di sviluppo non è stato semplicemente quello di esportare codice utilizzante le API di Selenium WebDriver, tutt'altro.
Si è approfondito il funzionamento di Selenium IDE e si è constatato che ad ognuno di questi progetti (Insieme di test registrati) è un file JSON con estensione .side. È stato preso il progetto salvato in questo formato e ci si è creato sopra un vero e proprio interprete per la formattazione di questo tipo di file che salva Selenium IDE (Vedi 3.2.2).

È stata presa questa scelta per poter astrarre la creazione dei test case al di fuori del tool Selenium IDE; potendo partire da un file vuoto, purché rispecchi il formato dei file che vengano salvati dallo strumento(Vedi 5.2).

2.3 Altri tool utilizzati

Descriviamo altri strumenti utilizzati usati di rado, o semplicemente di minore importanza per la creazione del nostro applicativo; Si escludono le librerie standard JDK e del Framework Spring.

Si lascerà un link Maven Repository/Github corrispondente alle librerie degli oggetti in questione, per evitare uso di librerie omonime.

StringSimilarity: classe atta al confronto delle pagine visitate dal Source test case e dal Follow-Up test case; Usa un'implementazione dell'algoritmo delle distanze minime di Levenshtein.[1] Dei test da confrontare si prendono le singole pagine HTML (Vedi 3.5) e le si confronta come stringhe.
L'algoritmo prende in input due stringhe A e B e misura il numero delle modifiche elementari (eliminazione,sostituzione e inserimento) necessarie per trasformare la stringa A (Source) nella stringa B (Follow-Up).
L'implementazione scelta offre un valore in percentuale di questa differenza: si può manualmente (Vedi 4.3.1) impostare un limite entro il quale due pagine sono considerate "quasi-simili" (Default 75%, può cambiare molto in base alla varietà di elementi e selettori nella pagina HTML).
JSONSimple[2] (Vedi 3.2.1): Libreria utilizzata dal TestCaseCreator per avere sotto forma di oggetto il file JSON dato in input del Source test case, essenziale per la creazione del primo (Source) oggetto test case.
Generex[3]: Classe usata per creare espressioni regolari in ambiente Java.
Partendo da una stringa equivalente a una REGEX scitta con notazione standard (Argomento del

[1] https://stackoverflow.com/questions/13564464/problems-with-levenshtein-algorithm-in-java
[2] https://mvnrepository.com/artifact/org.json/json-simple/20090211
[3] https://github.com/mifmif/Generex

costruttore), si crea un oggetto equivalente a un'automa a stati finiti che permette il confronto con stringhe o con altre sue REGEX.
E' usato principalmente nella verifica delle regole durante l'esecuzione dei Follow-Up Test case (Vedi 3.4.2).
JUnit[4]:Framework usato in ambiente java per effettuare test di unità.
E' stato ampiamente utilizzato in una prima fase di sviluppo, in particolare per fare debugging sul modo in cui venivano convertite in ambiente Java i singoli comandi di Selenium IDE (Vedi 3.3).

Una meccanica a cui vale la pena spendere due parole è quella della serializzazione.
La serializzazione in Java ci consente di convertire un oggetto in un flusso che possiamo inviare sulla rete o salvarlo come file o archiviarlo in DB per un utilizzo successivo.
In Java, questo avviene tramite l'interfaccia Serializable, e alle varie classi presenti nelle JDK per il formato del testo di I/O (Nel nostro caso si citano:File,FileOutputStream,ObjectOutputStream).
Il processo di serializzazione è stato usato per salvare gli oggetti test case in modo persistente (vedi 3.6).

[4]https://mvnrepository.com/artifact/junit/junit

3 Morfify: A Metamorphic Test solution for Web Application

Presentiamo Morfify, un ambiente per metamorphic testing per applicazioni web.
Morfify può essere usato per fare testing end-to-end per la propria web application, infatti, a partire da un singolo caso di test (sorgente), l'applicativo è in grado di generare test molto simili, con variazioni imposte a grandi linee dall'utente (follow-up).
Durante l'esecuzione, l'applicativo tiene traccia delle pagine che vengono visitate durante tutto il caso di test e usa quest'informazioni generate dai Follow-up per stabilire se il test sorgente ha avuto Successo (Success) oppure no (Faulty).

In questo capitolo vedremo in prima battuta il problema posto da risolvere e gli obbiettivi che si è riuscito a raggiungere con Morfify; vedremo nel dettaglio le cinque componenti principali del programma e in che modo sono state implementate e infine lasceremo un piccolo manuale d'uso.

L'applicativo è stato sviluppato per l'attività di tirocinio interno svolto sempre con il professore Adriano Peron.

3.1 Architettura del sistema

Siamo interessati alla creazione di un ambiente di testing utilizzante la tecnica di MT, come visto durante la descrizione del problema, essa si presenta particolarmente efficace, nonché abbastanza gettonata, quando si tratta di fare testing su Web Application (Vedi 1.6).

Vogliamo che il nostro sistema venga usato in realtà software di vari calibri e che si presenti come uno strumento efficace, completo e facile da usare.
I requisiti per usare Morfify sono:

- Una web application da testare;
- Se possibile, un testing di unità e di integrazione ben scritto, per reggere la piramide di testing (Vedi 1.1);
- un test case con un oracolo non noto (untestable test);
- Delle MR da applicare all'applicativo, in particolare un insieme di regole da applicare per costruire i Follow-Up.

L'ambiente Morfify è composto da 5 componenti principali, approfondiremo ulteriormente le loro proprietà nei paragrafi successivi:

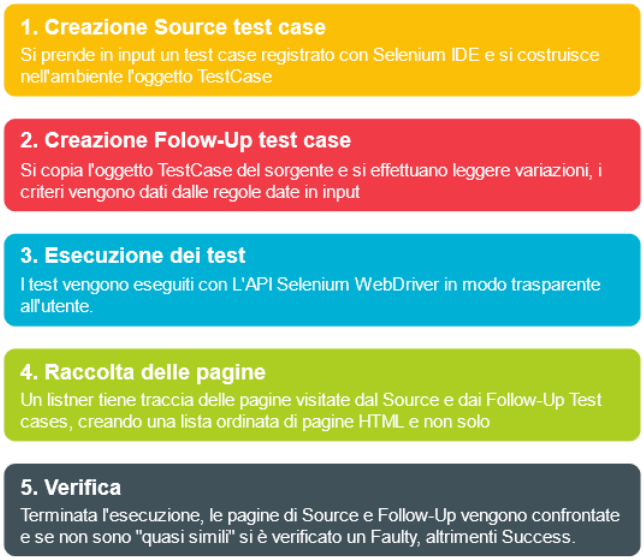


Fig. 10: Descrizione delle componenti di Morfify

- Creazione Source Test case: E' la componente che si occupa di prendere in input il test case da dover eseguire, si tratta di un parser JSON (TestCaseCreator) che prende in input un file generato da Selenium IDE e lo trasforma in un oggetto della classe TestCase; banalmente è la componente che porta l'input dentro all envirement.
- Creazione Follow-Up Test cases: E' la componente che si occupa, partendo da una copia del caso di test sorgente, di applicare le modifiche imposte dall'insieme delle regole date in input.
  L'output del sistema in questione è sempre uno o più oggetti TestCase definito come un Follow-Up.
- Esecuzione Test Case: tramite la classe TestRunner, che svolge il ruolo da esecutore dei TestCase, vedremo come vengono eseguiti Source e Follow-Up test.
- Raccolta delle pagine: durante l'esecuzione, è affiancato un listner che tiene traccia dei cambiamenti di pagina, sia da un punto di vista di codice (HTML,CSS etc...) che banalmente visivo (Crea uno screenshot per ogni pagina), l'esecuzione di un test case diventa una lista ordinata di pagine che vengono visitate e sarà parte integrante dell'Output offerto da Morfify.
- Verifica: Tramite L'informazioni raccolte durante l'esecuzione di Source e Follow-Up test cases, si mettono a confronto le pagine visitate dal sorgente e quelle dei vari Follow-up; le pagine devono essere "quasi-simili" tra di loro altrimenti si sarà verificato un Faulty del Source test case.

3.1.1 Diagrammi di flusso

Presentiamo due diagrammi di flusso per capire al meglio i passaggi sequenziali riguardanti la creazione ed esecuzione dei Source e dei Follow-Up test cases.

Tutte le classi citate nei diagrammi verranno approfondite nei paragrafi successivi.

La prima cosa che viene effettuata da Morfify è la creazione, ovvero portare nell'ambiente di testing sotto forma di oggetto, ed esecuzione del Source test case.
Possono essere dati in input più test case da usare come Source, per facilitare la comprensione sottintenderemo che nel file di input è presente un solo Source test Case.

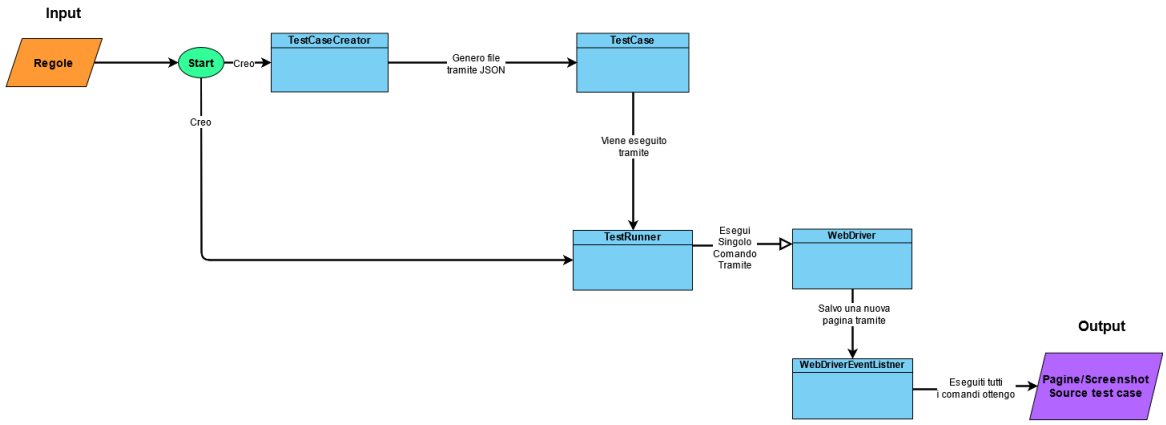


Fig. 11: Flowchart per la creazione ed esecuzione dei Source test case

Il TestCaseCreator (vedi 3.2.4) usa un parser in JSON per interpretare il file dato in input e crea un oggetto test case corrispondente al sorgente.
Il TestCase viene eseguito dal TestRunner che ha lo scopo di interpetrare i comandi da eseguire nella lista di comandi dell'oggetto TestCase ed eseguirli usando le API di Selenium WebDriver.

A contorno delle API di Selenium WebDriver è stato messo un Listner, chiamato EventDriverWeb-Listner, che si "attiva" ogni volta che viene verificata una differenza tra pagine.
Il listner salva le pagine e crea una lista ordinata di pagine da dare alla componente atta a verificare la somiglianza tra le pagine visitate nel Source e nei Follow-Up Test Cases.

Vediamo adesso come vengono creati ed eseguiti i Follow-Up Test cases.
Morfify esegue temporalmente prima il Source Test Case, motivo per cui l'abbiamo presentato per primo: ciò che deve essere chiaro però è che l'esecuzione tra un Source e un Follow-Up test case è indipendente l'una dall'altra.
Questo perchè la verifica della MR viene effettuata dopo l'esecuzione di tutti i due tipi di test, per poter confrontare le pagine visitate.

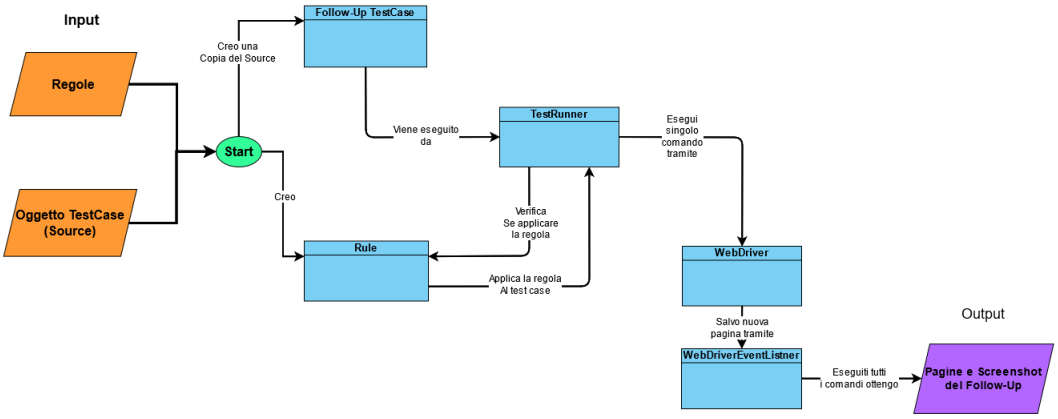


Fig. 12: Flowchart per la creazione ed esecuzione dei Follow-Up test cases

Si inizia effettuando una copia del Source test case: essa è la base per l'esecuzione del Follow-Up,

l'uso delle regole date in input tramite la classe Rule invece, serviranno per modificare questa copia e renderla leggermente diversa sull'input.
La modifica (sempre se la regola sarà attivata) dell'oggetto test case avviene durante la sua esecuzione dal test runner, in particolare comando dopo comando.
La classe Rule restituisce al TestRunner le modifiche apportate all'oggetto TestCase, permettendo al TestRunner di eseguire il comando modificato, e di far sempre attivare il listner sopra citato, per poter salvare le differenza tra pagine.

3.1.2 Requisiti di sistema

Vogliamo costruire un ambiente di sviluppo per testing in linguaggio Java di Web Application chiamato Morfify.

Iniziamo identificando i requisiti.
Essi sono composti da un identificativo, un nome ed una descrizione.

Requisiti utente:

ID:	RE01
Nome:	Esecuzione di test sorgente
Descrizione:	L'utente deve avere la possibilità di dare in input il test sorgente,che dovrà essere eseguito e visualizzato dall'utente nell'esatto modo in cui l'ha creato

ID:	RE02
Nome:	Creazione Follow-Up
Descrizione:	L'utente deve avere la possibilità di dare in input un'insieme di regole, dovrà poter modificare tutti gli attributi presenti in ogni comando di un test case

ID:	RE03
Nome:	Visualizzazione Follow-Up
Descrizione:	L'utente deve avere la possibilità di dare in input un'insieme di regole e vedere eseguite i test che vengono generati.

ID:	RE04
Nome:	Pagine visitate
Descrizione:	Al termine dell'esecuzione dell'applicativo, l'utente deve poter visualizzare i test case sorgenti staccati dai follow-up generati. Per l'utente deve essere immediato capire se dei Follow-Up Test case fanno dismatch con il sorgente, con particolare riferimento all'identificazione delle pagine coinvolte.

Requisiti non funzionali:

ID:	RENF01
Nome:	Insieme di regole con REGEX
Descrizione:	Le regole vanno implementate con delle REGEX sia per identificare quando attivarla e sia per cosa.

ID:	RENF02
Nome:	Performance esecuzione
Descrizione:	L'applicazione deve funzionare a prescindere dal caso di test e dalla pagina web data in input.

3.2 Creazione dei source test case

Esaminiamo la prima componente che si incontra durante l'esecuzione di Morfify: la creazione di un test case sorgente all'interno dell'applicativo.

Essendo uno dei 3 input richiesti dal programma, è più corretto specificare che questo primo processo di esecuzione più che "creare" in realtà effettua un trasferimento da un test case registrato su un file (Vedi 3.2.4) all'ambiente di Morfify, ovvero sotto forma di oggetto.

Selenium IDE permette la creazione di un progetto, dentro il quale possono risiedere vari test, l'utente che usa Morfify sarà libero di inserire tutti i Source Test Case che desidera all'interno di Selenium, il nostro parser creerà un oggetto TestCase per ognuno dei test che trova all'interno del file JSON.

3.2.1 Classe TestCase

La classe TestCase è la rappresentazione di un caso di test all'interno dell'ambiente Morfify.
I casi di test quindi vengono proiettati nel sistema sotto forma di oggetti della classe TestCase, sia che essi siano dei test case sorgenti o follow-up.

Esaminiamo gli attributi della classe principali:

```
private String id;
private String name;
private String url;
private Boolean isFollowUp;
private List<Command> commands;
```

- ID: Ogni test case ha un suo identificativo usato internamente per effettuare controlli di vario genere.
- Name: Nome del test case.
- Url: Url iniziale su cui far partire il test
- isFollowUp: Identifica se si tratta di un oggetto rappresentate un test follow-up oppure un Source.
- commands: Racchiunde una lista di comandi che identificano il percorso svolto dal test.

3.2.2 Design pattern BUILDER

Il builder pattern è uno dei più importanti pattern, conosciuto anche come GoF design pattern.

Questo modello è molto popolare perché separa la costruzione di oggetti complessi dalle loro rappresentazioni in modo che il processo di costruzione stesso possa creare rappresentazioni diverse.
In questo modo, l'algoritmo utilizzato per creare oggetti complessi non ha nulla a che fare con le parti che compongono l'oggetto e come vengono assemblati. Ciò ha l'effetto diretto di semplificare la classe, consentendo alla classe del generatrice di concentrarsi sulla costruzione corretta dell'istanza, lasciando che la classe astratta si concentri sul funzionamento dell'oggetto.

Questa funzione è particolarmente utile quando si desidera assicurarsi che l'oggetto sia valido prima di crearne un'istanza e non si desidera che la logica di controllo appaia nel costruttore dell'oggetto.

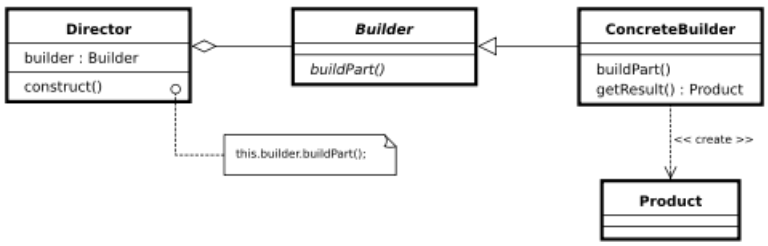


Fig. 13: Design pattern builder

Non è richiesto che la classe Builder sia una classe a parte: La classe builder può essere definita come una classe interna statica della classe di definizione dell'oggetto.
Nel nostro caso appunto si è preferito delegare la creazione delle specializzazioni sempre al "Builder" (TestCaseGenerator) ma di usare metodi diversi in base da cosa si parte per in stanziare l'oggetto

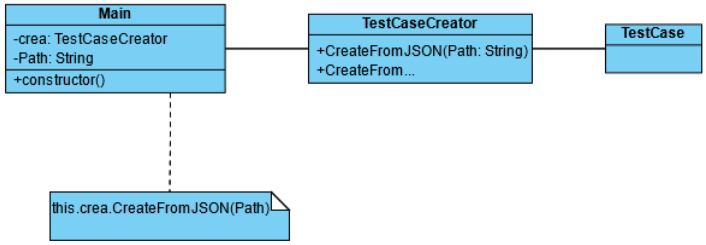


Fig. 14: Design pattern builder in Morfify

Attualmente l'unico modo di dare il test case alla classe TestCaseGenerator è tramite il Path del file contenente il file JSON generato da Selenium IDE (O un file JSON con sintassi simile a quello generato dal tool), si è però cercato di creare con questo design pattern una classe che un domani potesse ospitare altre modi di prendere in input questo test case e portalo nell'ambiente Morfify (Vedi 5.2).

3.2.3 JSON

JSON (Javascript Object Notation) è un formato strutturato per rappresentare un'insieme di dati. Immaginiamo che dobbiamo trasferire dati anagrafici di una persona (Nome,Cognome,Data di nascita etc...) tra client e server.

Gli array contenenti queste informazioni devono essere trasmessi e gestiti sulla rete e questo solleva alcuni grossi problemi: in teoria, i dati dovrebbero essere serializzati prima della trasmissione, quindi trasmessi e infine riorganizzati a destinazione.

Il formato JSON aggira in qualche modo questo problema, anche se in modo poco ortodosso e ne facilita la gestione: grazie ad esso l'array diventa una stringa con un formato specifico.
Passare da un elenco di oggetti (array) a JSON e viceversa, è facile, sempre possibile e reversibile il che è molto interessante per la creazione di servizi web.
Inoltre, JSON è anche indipendente dal linguaggio di programmazione utilizzato, il che lo rende adatto all'uso nelle situazioni più complesse.

Morfify ha utilizzato un Parser JSON (JSONParser[5])per riuscire a leggere il file dato in input, in

[5]https://javaee.github.io/javaee-spec/javadocs/javax/json/stream/JsonParser.html

particolare sono state prelevate tutte le informazioni, sia del test case in sè (Nome,Url etc...) ma anche le informazioni relative ad i singoli comandi.

Il nostro parser infatti preleva le informazioni dei comandi e le ordina, inserendole nella lista dei comandi presente come attributo della classe TestCase

3.2.4 Implementazione

Esaminiamo il metodo della classe TestCaseCreator CreateFromJSON, il metodo prende in ingresso il path del file contenente i test case in formato JSON e resitutisce una lista di TestCase equivalenti a quelli presenti nel file.

```
public List<TestCase> CreateFromJSON(String jsonPathFile) {
   List<TestCase> testCases = new ArrayList<>();
   JSONParser parser = new JSONParser();
   Object obj = null;
   obj = parser.parse(new FileReader(jsonPathFile));
   JSONObject testObj = (JSONObject) obj;
   JSONArray jsonTests = (JSONArray) testObj.get("tests");
   Iterator<JSONObject> iterTests = jsonTests.iterator();
   while (iterTests.hasNext()) {
      // itero sui test case nel file
      JSONObject testCaseJSON = iterTests.next();
      TestCase testCase = new TestCase();
      // Passaggio da testJSON a Test Case
      testCase.setUrl((String) testObj.get("url"));
      testCase.setName((String) testCaseJSON.get("name"));
      // Retrieving command
      List<Command> risCommand = new ArrayList<>();
      JSONArray jsonCommands = (JSONArray) testCaseJSON.get("commands");
      Iterator<JSONObject> listCommands = jsonCommands.iterator();
      while (listCommands.hasNext()) {
         JSONObject commandJSON = listCommands.next();
         // gestione targets
         List<Pair<String, String>> listAlternative = new ArrayList<>();
         JSONArray jsonAlternative = (JSONArray) commandJSON.get("targets");
         Iterator<JSONArray> iterOfAlternative = jsonAlternative.iterator();
         while (iterOfAlternative.hasNext()) {
            JSONArray pairJSON = iterOfAlternative.next();
               if(pairJSON.size()>1)listAlternative.add(new Pair<String, String>((String)
                   pairJSON.get(0), (String) pairJSON.get(1)));
             else listAlternative.add(new Pair<String, String>((String) pairJSON.get(0),
                 null));
      }
         // Creazione command
         Command command = new Command((String) commandJSON.get("comment"), (String)
             commandJSON.get("command"),
              (String) commandJSON.get("target"), (String) commandJSON.get("value"),
                  listAlternative);
         risCommand.add(command);
      }
      testCase.setCommands(risCommand);
      // Aggiungo il test nella lista da restituire
      testCases.add(testCase);
      return testCases;
   }
```

Si parte istanziando la lista di testCases (I Source test Case possono essere anche più di uno) e l'oggetto JSONParser, che grazie al suo metodo parse trasforma il nostro file JSON in un unico oggetto, da convertire successivamente in un JSONbject o in un JSONArray, in base all'elemento che ci viene presentato in ordine sequenziale.

Un esempio sono le righe 6,7 e 8: sapendo che il file JSON di partenza partiva col definire un oggetto, ho effettuato un cast esplicito a JSONObject; continuando, sapendo che dentro a quest'oggetto c'era un'array di nome test, richiamo il metodo get della classe JSONObject che mi restituisce un array (JSONArray) di JSONObject su cui posso iterare.
Ogni elemento dell'array equivale a un test case pronto da ispezionare, motivo per cui creo un'oggetto TestCase ad ogni ciclo e lo popolo con i campi presenti nel JSONObject.
Si arriva a riga 20 dove si necessita di prelevare le informazioni contenenti i singoli comandi, esamineremo le componenti di un singolo comando in 3.3.2, per il momento ci basti sapere che per ogni test case è associata una lista di comandi sotto forma di array di nome "commands", quindi, iteriamo su quest'oggetto e otteniamo così un oggetto (commandJSON, riga 21) contenente il singolo comando.
Da quest'oggetto estriamo ed elaboriamo tutti gli attributi di nostro interesse e creiamo un oggetto command (riga 32) con tutte le informazioni necessaria da importare su Morfify tramite l'oggetto TestCase.

3.3 Esecuzione test case

In questo paragrafo esaminiamo la terza componente di Morfify: L'esecuzione di un caso di test all'interno dell'ambiente Morfify.

Approfondiamo prima questa componente, invece che la Creazione dei Follow-Up, poichè dopo la creazione dei Source Test Case, essa è la successiva ad essere "invocata".

Per esecuzione intendiamo la simulazione delle azioni registrate nel caso di test all'interno di una finestra browser isolata.
L'esecuzione viene effettuata tramite le API di Selenium IDE , come già spiegato in 2.2 , questa libreria è composta da un driver, ovvero colui che delega le istruzioni al browser e gestisce la comunicazione da e verso Selenium.

L'uso del driver avviene nella classe oggetto di questo paragrafo: la classe TestRunner.
la classe TestRunner viene creata dal main e contiene due metodi per eseguire un Source oppure un Follow-Up test case.
La modifica principale tra i due metodi consiste nel richiamare il metodo di matching della regola (vedi 3.4.2).

La classe TestRunner , per ogni esecuzione di test case, crea un driver a cui viene dato un URL su cui iniziare l'esecuzione del test, si continua con la singola esecuzione dei comandi.

Attualmente Morfify esegue i suoi casi di test soltanto nel browser Google Chrome, l'architettura di questa componente è fatta in modo di inserire facilmente in futuro altri browser come Firefox o Microsoft Edge (Vedi 5.2).

3.3.1 Esecuzione di un comando

Approfondiamo l'uso di WebDriver e andiamo a vedere in che modo viene eseguito un comando all'interno di Morfify.

Come già detto nei capitoli precedenti, Morfify usa il framework Selenium per creare (Selenium IDE) ed eseguire (Selenium WebDriver) i casi di test.
Sarebbe un'errore però pensare, che ci si è limitati nell'uso "standard" di questi potenti tool.
Selenium IDE permette l'esportazione dei propri casi di test in vari linguaggi di programmazione e in vari framework di unità (JUnit,KUnit etc...).

Quindi, a partire dal file JSON generato da Selenium IDE, nonché mero salvataggio del progetto registrato dall'utente, si utilizza un interprete interno al plugin-in per creare un file contenente il test case scritto nel linguaggio e nel framework di riferimento.

Ciò che a noi interessava era l'interpretazione dei singoli comandi ovvero, dato un comando di Selenium IDE, qual'era la sua rispettiva/e linea/e di codice dell'API Selenium WebDriver.

Osservando le variazione dell'output di Selenium IDE su svariati casi di test, si è riuscito a creare un interprete, completamente autonomo a Selenium (Che non utilizza codice proprietario) che riuscisse ad eseguire un comando scritto con la sintassi presente nel file di salvataggio del progetto.
Questo interprete è presente nella classe TestRunner con il metodo executeCommand, questo metodo prende in ingresso il test case che si sta eseguendo in quel momento e un suo corrispettivo comando ed usando le API di Selenium WebDriver effettua l'azione descritta nel comando.
Non mostriamo il codice per una mera questione di leggibilità: il metodo è lungo oltre 200 linee di codice ed non è nient'altro che un grande switch case che come argomento prende la stringa risiedente nell'attributo "command" della classe Command.

Nonostante il grande lavoro per dare spazio un pò a tutti i tipi di comandi che Selenium IDE permette di usare nei suoi ambienti; alcuni comandi, come l'utilizzo di costrutti if, for, while etc... non sono supportate in Morfify.
Quindi, è importante tenere a mente (vedi 3.6) che se si vuole dare in input a Morfify un caso di test registrato con SeleniumIDE esso non dovrà contenere questi comandi. Come spiegheremo in 3.3.1, l'attributo "command" identifica il tipo di azione che si sta andando ad effettuare (Click,scritture su tastiera, drag & drop etc...), questo valore è sufficiente per poter definire il corrispettivo comando della libreria di Selenium WebDriver da usare.

3.3.2 Esecuzione di Source e Follow-up

Una volta spiegato il contenuto della funzione executeCommand, esaminiamo altri metodi della classe TestRunner.

In particolare faremo riferimento a due metodi di facile comprensione utilizzati per eseguire rispettivamente Source e Follow-up test case.

```
public Pair<List<WebPage>,List<File>> executeSource(TestCase test) {
   // Creazione driver
   initRunner();
   for (Command currCommand : test.getCommands()) {
      executeCommand(test, currCommand);
   }
   driver.quit();
   return new Pair<>(listener.getHtmlPages(),listener.getScreenshotPages());
}
```

il metodo executeSource prende in input un oggetto TestCase e instanzia un driver e tutte le componenti per farlo funzionare nel metodo initRunner().
L'esecuzione consisterà nell'iterare sulla lista dei comandi presenti come attributo nell'oggetto TestCase e richiamare il metodo executeCommand con il comando corrente.
Si chiude il WebDriver per poter far avviare correttamente il successivo test e si restituisce il contenuto raccolto dal listner (vedi 3.5.2).

Come si può vedere, non ci sono controlli sul tipo di testCase che si sta eseguendo (Verificare se si tratta di Source o Follow-Up), in linea di massima il metodo serve per eseguire un test case senza dover applicare delle regole su di esso.

Esaminiamo invece l'esecuzione dei Follow-up:

```
public Triple<TestCase, List<WebPage>,List<File>> executeFollowupTestCase(TestCase
    testSource, List<Rule> rules) {
   // Creazione driver etc...
   initRunner();
   TestCase followUp = new TestCase(testSource);
   for (Command currCommand : followUp.getCommands()) {
      // Fa il match delle regole e restituisce la modifica della prima regola a
      // fare il match
      verifyRules(rules, currCommand);
      // Eseguo il comando modificato, sto eseguendo il followUp
      executeCommand(followUp, currCommand);
      driver.quit();
      return new Triple<TestCase, List<WebPage>,List<File>>(followUp,
          listener.getHtmlPages(),listener.getScreenshotPages());
}
```

il metodo executeFollowUpTestCase prende in input un oggetto di tipo TestCase , equivalente al Source TestCase oramai già eseguito, e una lista di regole da applicare (Vedi 3.4.2).

Si inizia creando una copia del Source test case, questo grazie a un costruttore della classe TestCase che ha il ruolo di clonatore.
I comandi di quest'oggetto TestCase clonato saranno dati al metodo verifyRules (Vedi 3.5.2) che modificherà, se le guardie delle regole si attiveranno, l'oggetto.
Si procede con l'esecuzione dei singoli comandi che a questo punto, se si dovevano modificare rispetto al Source test case, saranno stati modificati e si restituisce come output sempre il contenuto raccolto dal listner (Vedi 3.5.2).

3.4 Creazione follow-up

In questo paragrafo esaminiamo la seconda componente di Morfify: la creazione dei Follow-Up test cases.

Come già visto in 1.5 , una MR è composta da:

Regola : E' la parte della relazione che si occupa di modificare il Source test case, generando dei test (Follow-Up test cases) con leggere variazioni sull'input.

Verifica: Si occupa di stabilire, se il test ha avuto successo (Success) oppure no (Faulty). La verifica viene effettuata dalla quinta componente che vedremo meglio in 3.5.3, concentriamoci ora sulla regola.

Una regola di per se può essere implementata in tanti modi, dipende da che tipo di informazioni possiamo avere sull'applicativo da testare.

L'obbiettivo di Morfify è quello di porsi come uno strumento versatile per qualsiasi Web Application. E' facile dedurre che più tipologie di applicativi ho intenzione di testare, più le informazioni con cui posso scrivere (ciò che posso dare per scontato) siano limitate.

Consideriamo la più elementare delle azioni che con Morfify potrei essere interessato ad eseguire in automatico durante un caso di test: un click di un bottone.

Siamo interessati a dare in input un caso di test dove a un certo punto è richiesto un click ad un bottone X , vorrei testare il mio applicativo al fine di verificare che se si clicca invece il bottone Y il programma continui a funzionare correttamente, Ovvero che le pagine che verranno visitate dal Follow-up che avrà premuto il bottone Y, siano "quasi-simili" a quella del test originale.
In che modo potrò testare il mio applicativo se nell'ambiente Morfify non so a priori dell'esistenza del bottone Y? E chi dovrebbe darmi quest'informazione?

La risposta è l'insieme delle regole, esse devono specificare esattamente quale tipologie di comandi, quali tipi di elementi e quali tipi di valori devono essere modificati.
E' chiaro quindi che l'insieme delle regole deve essere dato in input e che queste regole devono operare sui singoli comandi.

3.4.1 La classe Command

Esaminiamo la classe Command che racchiude tutte le informazioni per poter eseguire un azione usando le API di Selenium WebDriver.
Un oggetto della classe Command viene creato durante il processo di parsing del file JSON (vedi 3.2.2), nonchè durante il processo di clonazione del Source come instanzazione dei Follow-Up.
questi comandi sono stati trascritti dal file JSON cercando di mantenere un formato simile, anche se sono state modificati alcuni aspetti e modi di contenere le varie informazioni dei comandi.

```
private String id;
private String command;
private String target;
private String value;
private List<Pair<String, String>> targets;
private String comment;
```

Esaminiamo gli attributi delle classe Command:

- ID: unico per ogni oggetto creato (Attualmente obsoleto).
- command: E' una stringa che identifica il nome del comando che si deve eseguire (Esempi: click, doubleclick, open, close, type, sendKeys etc...).
- target: Identifica il selettore a cui deve essere effettuato il comando, ovvero è quell'elemento della pagina che sarà "bersagliato" dal tipo di comando.
- value: è il valore da associare all'azione effettuata; varia in base al tipo di command (se voglio scrivere su una form, il comando sarà type e value conterrà il valore da scrivere, per esempio).
- targets: In una pagina HTML, ci si può riferire ad un'elemento della pagina usando diversi selettori; Se il selettore usato in target, un giorno, dovesse cambiare (un cambio ID in una form, per esempio), ci si ritroverebbe ad effettuare un'azione ad un selettore inestistente, comportanto il lancio di un eccezione da parte del WebDriver; targets è una lista dei diversi selettori in cui l'elemento scelto in target può essere identificato.
- comment: Ad ogni comando è associato un commento puramente facoltativo.

Selenium IDE ha un proprietà interna al suo sistema che gli permette di ottenere diversi selettori in base all'elemento: Morfify si limita a proiettare questi selettori nell'ambiente ma non vengono veramente interpellati da nessuna delle sue componenti.

Tra gli sviluppi futuri (Vedi 5.2) c'è anche la volontà di utilizzare, in caso il selettore presenti in target non venga trovato, la lista dei selettori presenti in targets.

Durante l'esecuzione di un Source Test Case, i comandi vengono banalmente eseguiti, mentre un Follow-Up Test case prima di eseguire un suo comando necessità di passare dalla classe Rule.

3.4.2 La classe Rule

La classe Rule, come lascia intuire il suo nome, racchiude le funzionalità richieste sulla implementazione della regola riguardante la MR che intende verificare Morfify presentata in 3.1.
I suoi oggetti vengono instanziati dal main, e sono parte integrante dell'input di Morfify.

Una regola, volendo, può essere divisa in due parti:

- Guardia: Racchiude una serie di condizioni da dover rispettare per poter applicare il cambiamento.
- Cambiamento: banalmente, è la modifica da applicare per effettuare una variazione al Follow-Up test case.

Anche la classe rule è "divisa" in queste due fasi, esaminiamo gli attributi di questa classe vediamo in quanti modi si può applicare una Guardia e quali solo le modifiche concesse dal cambiamento.

```
 //Guardia
 private final String urlTarget;
private final String commandMatch;
private final String targetMatch;
//Cambiamento
private final String mutationType;
private final String patternValue;
private final String patternTarget;
```

Innanzitutto notiamo che questa classe ha tutti gli attributi final, questo perchè la regola è definita dall'utente e non viene effettuata una modifica.
Tutti gli attributi sono delle stringhe, come vedremo nel paragrafo successivo, in realtà si tratta di REGEX (Tranne mutationType) che possono racchiudere vari valori rispettando la sintassi dell'espressone regolare.
Un esempio l'abbiamo con il primo attributo della guardia: urlTarget, essa è una REGEX che possiamo usare per identificare un insieme di URL che possono essere visitati durante l'esecuzione dei nostri casi di test.

Per la costruzione delle esperessioni regolari, si usa la notazione PCRE2 (da PHP 7+), se non si è familiari con le REGEX, si consiglia la lettura del manuale d'uso[6].

Immaginiamoci un'applicativo web che in una pagina contiene una tabelle contententi vari persone con i loro dati anagrafici, se si clicca sul nome di un proprietario, si viene portati in una pagina contentente tutte le informazioni su quest'ultimo.
Un'implementazione valida per questo tipo di situazione, prevederebbe la distinzione degli URL tramite un identificativo del proprietario (Es: home/1/info racchiude le informazioni di Mario Rossi, home/2/info racchiude le informazioni di Marco Bianchi).
Se fossi interessati a "bersagliare" tutti i proprietari presenti nella tabella, non potremmo limitarci a quello che ha raccolto il Source Test Case, poichè esso avrà un URL unico ed associato ad un singolo proprietario.

Quest'attributo serve proprio per incorporare vari tipi di URL che potrebbero essere "validi" per l'applicazione del cambiamento.
Se l'url in cui viene eseguito il comando attuale rientra nella REGEX, ovvero l'automa a stati finiti indotto dall'espressione regolare riesce ad eseguire il programma, si va avanti a verificare se anche il comando fà matching, con l'attributo commandMatch.
Essendo anche lui una REGEX, quest'attributo può essere usato per racchiundere un insieme di comandi che siamo interessati a modificare (per esempio, modificando il valore di un comando type, oppure invece di un click effettuare un type etc).

Se si è arrivati ad avere un matching con questi primi due attributi, la guardia si conclude con la verifica del taget: quest'attributo può essere usato per includere un insieme di selettori che siamo interessati a modificare (magari vogliamo cambiare il bersaglio di un comando).

[6] https://www.pcre.org/current/doc/html/pcre2test.html

Una volta che la guardia è stata attivata, si procede con il cambiamento del comando, questo può avvenire principalmente in due modi:

L'attributo mutationType identifica se la modifica che deve applicare la regole riguarda solo il target (SELECTOR), il valore (VALUE) o entrambi (BOTH) del comando che ha attivato la guardia.
Gli attributi patternValue e patternTarget sono delle REGEX che possono effettuare cambiamenti rispettivamente sul valore del comando oppure sul suo target.

Potremmo essere interessati a modificare solo un valore(Es: Voglio cambiare il valore scritto da un comando type, oppure solo il selettore (voglio effettuare un click dal selettore X a quello Y) oppure entrambi.

3.4.3 Applicazione di una regola

Esaminiamo il metodo verifyRules della classe TestRunner.

Il metodo è presente nella classe TestRunner per incapacità progettuali, era più opportuno implementarla nella Classe Rule essendo che è la regola a dover verificare e cambiare l'input del test case.
Il metodo prende in input la lista di regole date in input dal main ed il comando che bisogna "verificare" e restituisce il comando inalterato (nel caso la gurdia non sia stata attivata) o modificato secondo il cambiamento definito nella regola.

```
    private void verifyRules(List<Rule> rules, Command currCommand) {
      for (Rule rule : rules) {
         //Match URL
         Pattern patternUrl= Pattern.compile(rule.getUrlTarget());
         Matcher matchUrl=patternUrl.matcher(driver.getCurrentUrl());
            if (matchUrl.matches()){
               //Match Command
               Pattern patternCommand= Pattern.compile(rule.getCommandMatch());
               Matcher matchCommand=patternCommand.matcher(currCommand.getCommand());
               if (matchCommand.matches()){
                  //Match Target
                  Pattern patternTarget= Pattern.compile(rule.getTargetMatch());
                  Matcher matchTarget=patternTarget.matcher(currCommand.getTarget());
                  if (matchTarget.matches()){
                          //Verify Type of mutation and set
                          if (rule.getMutationType().equals("VALUES")) {
                             Generex generex = new Generex(rule.getPatternValue());
                             currCommand.setValue(generex.random());
                          } else if (rule.getMutationType().equals("SELECTOR")) {
                             Generex generex = new Generex(rule.getPatternTarget());
                             currCommand.setTarget(generex.random());
                          } else if (rule.getMutationType().equals("BOTH")) {
                             Generex generexTargetMu = new Generex(rule.getPatternTarget());
                             Generex generexValueMu = new Generex(rule.getPatternValue());
                             currCommand.setValue(generexTargetMu.random());
                             currCommand.setValue(generexValueMu.random());
                          }
                          //Prima regola soddisfatta
                          break;
               }
            }
         }
      }
    }
```

Per ogni regola iterata, si verifica se le REGEX contenute negli attributi.

urlTarget,patterCommand,patternValue siano soddisfatte. Questo è possibile grazie alle classi Pattern[7] e alla classe Matcher[8].
Il primo crea un automa a stati finiti partendo da una espressione regolare sotto forma di stringa, questa classe ha anche un metodo matcher, che prende in input una stringa, Anch'egli, espressione regolare, e restituisce un oggetto Matcher contenente l'esito del confronto.

Per tutte e 3 le guardie che si devono attivare quindi, è stato richiamato il metodo statico della classe Pattern compile che prende in input la regex e crea l'automa, subito dopo dall'oggetto appena creato si prende il contenuto del comando e si usa il metodo matcher per verificare se il comando attivi la guardia.
Se la guardia viene attivata per L'URL, per il comando e per il target (selettore), si procede con l'applicazione del cambiamento della regola.

Come già spiegato nel paragrafo precedente,
ci sono 3 tipi di cambiamenti (VALUES,SELECTOR,BOTH).
Da riga 18 fino a 27 si verifica quale cambiamento fare, e in base a quello ricavare dalla REGEX degli attributi patternValue e/o patterTarget un valore casuale tra quelli disponibili dall'espressione regolare.
Questo è stato possibile grazie alla classe Generex , che prendendo in input una stringa equivalente a una REGEX, restituisce, con il suo metodo random, una stringa per che fà matching con quell'espressione regolare.

E' responsabilità dell'utente creare una regola che rispetti, oltre il formato delle REGEX, il formato dei selettori, se si sbaglia a scrivere un cambiamento di una regola si potrebbero effettuare dei lanci di eccezione da pate dell'API di Selenium WebDriver durante l'esecuzione del test case modificiato.

3.5 Raccolta e Verifica delle pagine

In questo paragrafo esaminiamo le ultime due componenti di Morfify, analizzeremo in che modo avviene la raccolta delle pagine e la logica che c'è dietro, per poi concentrarci sulla verifica della MR di Morfify e in che modo è stata implementata.

3.5.1 Design pattern OBSERVER

Il pattern Observer (o anche chiamato pattern Listner) consente di definire le dipendenze uno-a-molti tra gli oggetti, in modo che se un oggetto cambia il suo stato interno, notificherà e aggiornerà automaticamente ogni oggetto dipendente. L'osservatore è colui che necessità di mantenere un alto grado di coerenza tra le classi correlate senza avere un forti dipendenze o accorpamenti.

Il patter viene usato quando diversi oggetti (Observer) devono conoscere lo stato di un oggetto (subject).
Il soggetto viene "osservato" e tanti oggetti (osservatori) che "osservano" i cambiamenti di quest'ultimo.

Analizziamo in dettaglio i componenti in esecuzione mostrati nel diagramma UML di seguito:

[7] https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html
[8] https://docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html

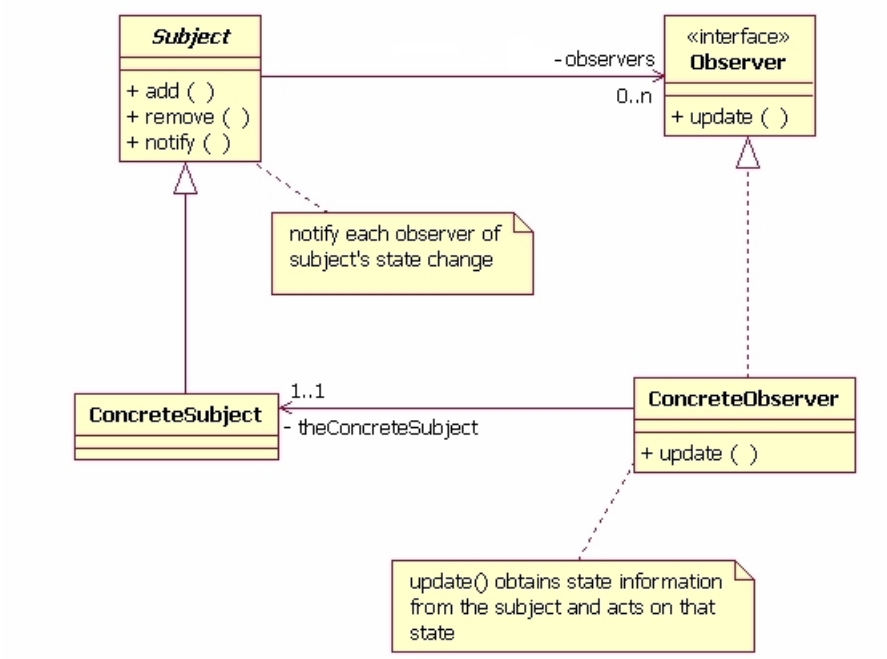


Fig. 15: Design pattern observer

- Subject: Chiamata anche classe Observable, sa chi sono i suoi observer e fornisce metodi atti ad aggiungere o eliminare quest'ultimi.
- Observer: Intesa sempre come un interfaccia, ha lo scopo di notificare gli eventi a un determinato Subject.
- ConcreteSubject: chiamata anche ObservedSubject, alla sua modifica, vengono notificati tutti gli observer associati ad esso; questo deve avvenire tramite i metodi messi a disposizione da Subject per comunicare con i ConcreteObserver
- ConcreteObserver: definisce il comportamente in caso di cambio di stato di un determinato oggetto ConcreteSubject.

Questo modello è ampiamente utilizzato in molte librerie, toolkit GUI, modelli di architettura MVC, messaggistica e realtime.

I vantaggi sono noti:

- Favorisce l'accorpamento(loosely).
- Consente di inviare in modo efficiente i dati ad altri oggetti senza modificare la classe Subject o Observer.
- Il soggetto e l'osservatore sono indipendenti e le modifiche a uno di questi componenti non richiedono modifiche agli altri componenti.
- Gli osservatori possono essere aggiunti/rimossi in qualsiasi momento.

Vediamo invece com'è stato implementato il pattern su Morfify.

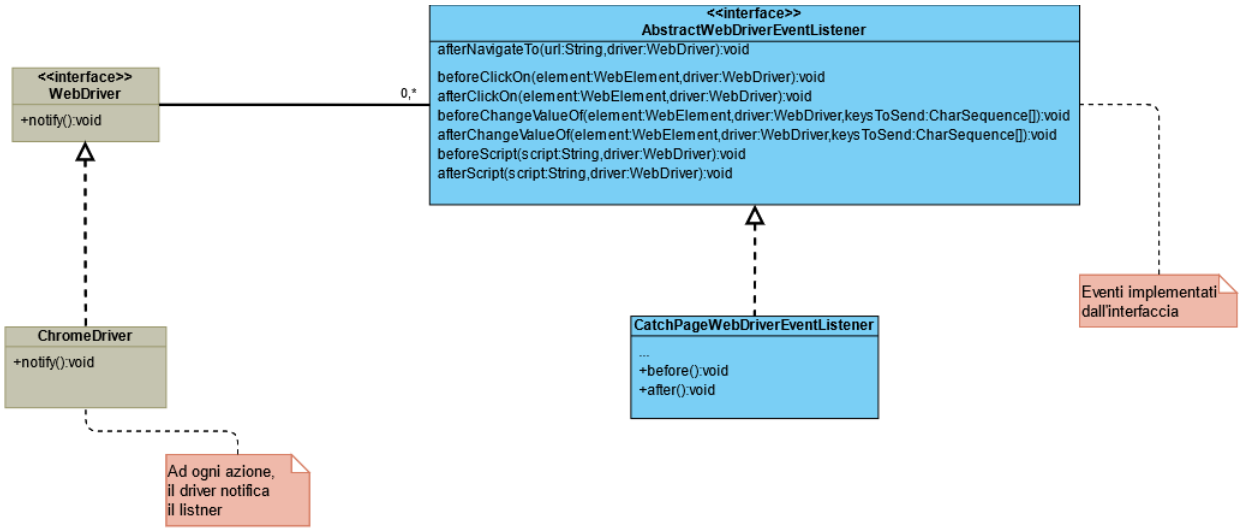


Fig. 16: Design pattern observer in Morfify

Il ChromeDriver svolge il ruolo di concreteSubject, essendo colui che esegue i comandi dal TestRunner, questo driver implementa il metodo notify (astratto, sono un insieme di chiamate a metodi a varie classi del framework), esso si occupa di far attivare l'EventListner associato a quello specifico driver.

Questo, a differenza del diagramma visto sopra, non avviene direttamente con un riferimento del listner all'interno del concrete Subject, ma tramite l'interfaccia WebDriver che comunica all'interfaccia AbstractWebDriverEventListner (e di conseguenza anche alla sua implementazione) il cambiamento di stato dell'oggetto.
Il concrete observer, chiamato CatchPageWebDriverEventListner, verrà interpellato ad ogni azione effettuata dal ChromeDriver che possa comportare un possibile cambiamento di pagina (Click a pulsanti, esecuzioni di script etc...).

In particolare, l'implementazione dei metodi presenti della sua interfaccia sono state tutte scritte usando dei metodi chiamati before
(per beforeClickOn,beforeScript) e after (per afterNavigateTo,afterClickOn,afterScript)

```
public void before(WebDriver driver) {
   currentPage = new WebPage(driver.getPageSource());
}
```

Il metodo before semplicemente salva la pagina in cui il driver si trova in quel momento, prima che venga effettuata l'azione.

```
   public void after(WebDriver driver) {
   String after = driver.getPageSource();
   if (similarity(currentPage.getHtmlPage(), after) < StringSimilarity.MAX_DIFFERENCE) {
      htmlPages.add(new WebPage(after));
      TakesScreenshot scrShot =((TakesScreenshot)driver);
      //Call getScreenshotAs method to create image file
      File srcFile=scrShot.getScreenshotAs(OutputType.FILE);
      screenshotPages.add(srcFile);
   }
}
```

Mentre il metodo after salva la pagina in cui si trova il driver dopo che il comando è stato eseguito, e utilizza una funzione di similarità (Vedi paragrafo successivo) per verificare se tra la pagina che aveva salvato in before e quella attuale sono "quasi-simili"; ovvero si cerca di capire se durante un qualsiasi tipo di comando, venga effettuato un cambiamento di pagina prima e dopo l'esecuzione del comando stesso.

Se il cambiamento è avvenuto, ovvero le due pagine non sono "quasi-simili" si procede a salvare la pagina prima del suo cambio e di aggiugerla alla lista delle pagine visitate, insieme ad uno screenshot di quest'ultimo.
Eseguire casi di test con un EventDriverWebListner permette la separazione tra l'esecuzione dei casi di test (terza componente) dalla necessità di raccogliere la lista ordinata delle pagine visitate, il che rende il codice più accorpato e leggibile.
Nell'interfaccia AbstractWebDriverEventListner sono presenti tanti eventi che possono essere "intercettati", noi abbiamo selezionato solo gli eventi che si attivano dai comandi che potenzialmente possono portare ad un cambiamento di pagina.
Il ChromeDriver deve delegare alla sua interfaccia la comunicazione agli observer, aspettare che venga eseguito il metodo before() del ConcreteObserver (si intende i casi in cui vengono eseguiti i comandi che richiamano gli eventi dell'interfaccia implementati), eseguire il comando, e comunicarlo al WebDriver, che attiverà il metodo after() del concreteObserver.

3.5.2 Verifica

Esaminiamo la quinta ed ultima componente di Morfify: La Verifica delle similarità tra pagine visitate dal Source con quelle visitate dai Follow-Up Test cases.

E' la più semplice, ma costituisce anche l'output principale di tutto l'applicativo, quindi va esaminata approfonditamente.
Nel paragrafo precedente si è accennati ad un algoritmo di similarità tra le pagine. quest'algoritmo viene utilizzato sia nella circostanza di verificare se si è passati da una pagina nuova tra l'esecuzione di un comando e l'altro, sia per verificare la similarità tra una pagina visitata dal Source con quelle visitate dal Follow-Up test case.

Si parla delle distanze minime di Levenshtein, un'unità di misura per calcolare la differenza tra due stringhe.
Date due parole A e B, la distanza di Levenshtein tra queste due parole è il numero minimo di modifiche di un carattere singolo (inserimento, cancellazione, sostituzione) necessario per modificare la parola A nella parola B.
La classe atta a contenere le pagine web è WebPage, essa attualmente è semplicemente un contenitore della stringa contenente l'html, tra gli sviluppi futuri(Vedi 5.2), c'è la volontà di usare un parser DOM per ottenere informazioni dettagliate sulle pagine, andando a identificare una differenza tra quest'ultime tramite l'analisi dei tag contenenti le pagine, piuttosto che una mera differenza tra stringhe.
l'algoritmo che utilizza le distanze minime di leveshtein è stato utilizzato per confrontare le pagine visitate; le pagine sono state trattate come delle stringhe.

L'implementazione utilizzata, in particolare, utilizza una classe StringSimilarity[9] che usa due metodi statici.
Il primo è editDistance, che calcola la differenza tra le due stringhe, il secondo è similarity, che utilizza la differenza generata dal metodo precedente per assegnare una percentuale di differenza, tramite un double che và da 0 (le pagine sono identiche) a 1 (le pagine sono completamente diverse).

[9]https://stackoverflow.com/questions/955110/similarity-string-comparison-in-java/16018452

Ritorniamo a parlare dell'algoritmo in sè: esso non è dei migliori quando si parla di verificare una similarità tra stringhe così lunghe e piene di parole, questo limite si fà sentire a seconda delle tipologie di pagine che si sta visitando.
In linea di massima, più il codice HTML della pagina è lungo, più la verifica delle similarità tra pagine può fallire.
Si è cercato di rendere il tasso di accettabilità (percentuale che definisce quando una pagina non p più "quasi-simile" ad un'altra) più versatile possibile, utilizzando una costante definita nella classe StringSimilarity.

```
public static final double SIMILARITYOFTWOPAGE = 0.75
```

Questa costante, è lasciata al 75%: dopo varie esecuzioni su più applicativi diversi e test case vari, si è arrivati alla conclusione che questo numero è ottimale per identificare una vera differenza tra le pagine.

In caso dovessero essere riscontrati falsi positivi, si consiglia di aumentare il valore di questa percentuale, viceversa, se pagine molto diverse tra di loro non vengono identificate, si consiglia di dimuire la costante.

3.5.3 Implementazione

Concludiamo mostrando come avviene la verifica delle similarità tra le pagine dei Source e dei Follow-Up test cases.

```
//Confronto delle pagine
Set<Pair<List<WebPage>,List<File>>> sourceTestCase=allTestCases.keySet();
for (Pair<List<WebPage>,List<File>> test: sourceTestCase){
   for(Triple<TestCase,List<WebPage>,List<File>> follow_Up : allTestCases.get(test)){
      TestCase testCaseFollowUp=follow_Up.getFirst();
      Iterator<WebPage> iterFollowUp=follow_Up.getSecond().iterator();
      Iterator<File> iterFollowUpFile=follow_Up.getThird().iterator();
      Iterator<WebPage> iterSource=test.getKey().iterator();
      List<Triple<WebPage,Boolean,File>> warningPages=new ArrayList<>();
      while(iterSource.hasNext() && iterFollowUp.hasNext() &&
          iterFollowUpFile.hasNext()){
         WebPage sourceWebPage=iterSource.next();
         WebPage followUpWebPage=iterFollowUp.next();
         File followUpFile=iterFollowUpFile.next();
         String pageHtmlSource=sourceWebPage.getHtmlPage();
         String pageHtmlFollowUp=followUpWebPage.getHtmlPage();
         if(similarity(pageHtmlSource,pageHtmlFollowUp) <
             StringSimilarity.SIMILARITYOFTWOPAGE) {
            System.out.println("Le due pagine sono troppo diverse!");
            if(!testCaseFollowUp.getFaulty() &&
                !testCaseFollowUp.getWarning())testCaseFollowUp.setWarning(true);
            warningPages.add(new Triple<>(followUpWebPage,true,followUpFile));
            printSimilarity(pageHtmlSource,pageHtmlFollowUp);
         }
         else warningPages.add(new Triple<>(followUpWebPage,false,followUpFile));
      }
      if(testCaseFollowUp.getFaulty())
          testCaseFollowUp.createFile("Faulty",warningPages);
      else if(testCaseFollowUp.getWarning())
          testCaseFollowUp.createFile("Warning",warningPages);
      else testCaseFollowUp.createFile("Success",warningPages);
   }
```

```
    }
}
```

Ci troviamo nell'ultime linee di codice di Morfify , questa porzione risiede nel main e viene eseguita dopo l'esecuzione dell'ultimo Follow-Up Test case.

Durante l'esecuzione dei test, è stata popolata una variabile chiamata allTestCase, che contiene:

- Tutti le liste di pagine e screenshot visitati dai Source test case.
- Tutti i TestCase, liste di pagine e screensgot visitati dai Follow-Up test cases.

La prima iterazione avviene per tutti i Source TestCase, ricordiamo che può essere dato un file contenente più Source Test Case.
per ogni Source TestCase quindi, si itera sui Follow-Up associati ad esso.

Da riga 5 a 9 vengono creati gli iteratori per esaminare la lista delle pagine di source e Follow-up, da 10 a 15 vengono estratti i contenuti degli iteratori,che oltre a servire per il controllo, dovranno essere serializzati (Vedi 2.3).
Da 16 in poi si verifica se il test runner ha individuato dei Warning, dei Faulty o dei Success tra un confronto tra pagine e l'altro.
Per Warning si intende un caso di test follow-up andato a buon fine, ma che al suo interno ha delle pagine che non sono "quasi-simili" con quelle del Source test case.
Per Faulty invece si intende un caso di test che non è riuscito ad essere stato eseguito correttamente, per via di eccezioni lanciate da parte del WebDriver (Ancora da sviluppare, vedi 5.2).
Per Success infine si intende un caso di test follow-up eseguito correttamente, in cui tutte le pagine sono state confrontate con il Source test case e sono risultate "quasi-simili" tutte tra di loro.

3.6 Gestione I/O

Morfify prende in input:

- un test case registrato con Selenium IDE (Formato .side, non esportato), alternativamente un file JSON con lo stesso formato dati.
  ATTENZIONE: non tutti i comandi utilizzabili in Selenium IDE sono supportati da Morfify.
- un'insieme di regole per definire la MR indotta dall'applicativo.
- numero di follow-up test cases che vuole vengano generati per il confronto.

Restituisce:

- Una cartella contenente le pagine visitate, con rispettivi screenshot di Source e Follow-Up test cases (Vedi 4.3).
- un messaggio per avvisare se il Source test case ha avuto successo (Success) oppure no (Faulty).

4 Valutazione sperimentale

In questo capitolo vedremo un esempio d'uso di Morfify utilizzando un applicativo web chiamato PetClinic.

Vedremo cosa permette di fare questa piccola web application, quali funzionalità del programma si è deciso di testare, in che modo è stato dato l'input a Morfify e infine come ha risposto il nostro ambiente.

4.1 PetClinic

Petclinic[10] è un applicativo di esempio (sample) per neofiti nell'utilizzo del framework Spring.
E' stato sviluppato dai fondatori di Spring Boot ed ha lo scopo di mostrare come utilizzare lo stack Spring per creare applicazioni semplici ma potenti orientate al database. PetClinic è stato largamente usato in un gran numero di studi empirici di ingegneria del software nell'ambito di applicazioni web.

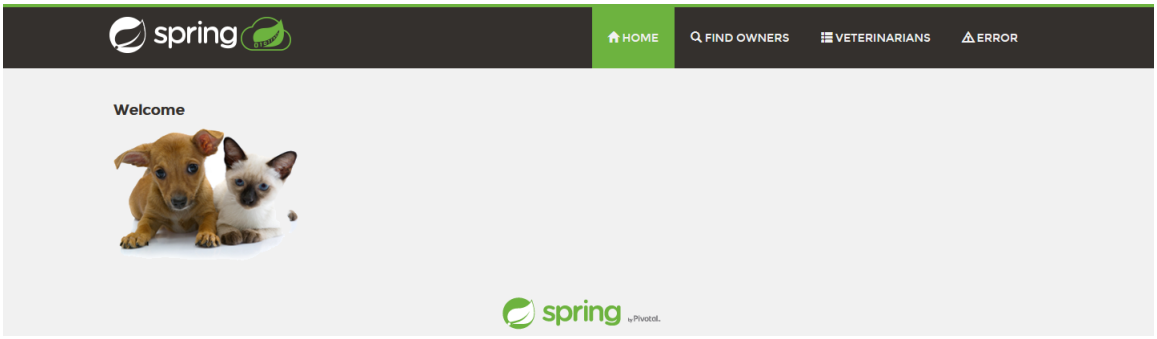


Fig. 17: Homepage di PetClinic

Si presenta come un sito di gestione di una clinica veterinaria, dalla homepage è possibile cercare i proprietari dei vari animali.
E' possibile aggiungere delle viste ad un animale, oppure aggiungere un nuovo animale per un proprietario. L'applicativo è molto facile da istallare in qualsiasi IDE moderno:
Le librerie esterne al framework Spring vengono aggiunte grazie a Maven e di default supporta un sistema di archiviazione con tecnologia in-memory chiamato H2, rendendo superflua l'istallazione di un DBMS sulla/e macchina/e ospitante/i.

La versione da noi utilizzata mostra come impostare Spring Boot con Spring MVC e Spring Data JPA. la web application dispone di un suo codice per testing di unità con framework JUnit, quello che si è cercato di fare è di fare testing end-to-end con Morfify.

Per dei meri esempi, ci si è limitati ad individuare 3 funzionalità da testare, sono stati scelti quindi 3 casi d'uso generati con Selenium IDE molto eterogeneo tra di loro, cercando di racchiudere le principali funzionalità da testare su una web application.
Essendo il nostro testing end-to-end siamo interessati a vedere se si riesce a soddisfare le funzionalità di un test esaminando se il comportamento del back-end è corretto.
PetClinic si presta molto bene per questo tipo di operazioni, poichè è usato principalmente proprio per studiare la comunicazione tra server e database per le operazioni CRUD.

In particolare, ci siamo focalizzati sulla creazione e l'inserimento di elementi nel DB.

[10] https://github.com/spring-projects/spring-petclinic

4.1.1 Funzionalità e regole

Abbiamo quindi individuato 3 casi di test da registrare:

- Aggiunta di un proprietario: l'applicativo permette l'aggiunta di un proprietario di animali, questi proprietari sono visibili sia ricercandoli tramite nome, sia in una tabella contenente tutti i proprietari.
  Vengono richieste informazioni come Nome, Cognome, Indirizzo etc...
- Aggiunta di un animale: dato un proprietario, l'applicativo permette l'inserimento di animali associati a quel determinato proprietario. Vengono richieste informazioni come Nome, data di nascita, tipo di animale (cane,gatto,uccello,criceto,lucertola,serpente).
- Aggiunta di una visita: Ogni animale ha delle visite associate, sono composte da una data e una descrizione (Es: visita di controllo).

Sono quindi stati registrati 3 casi di test con Selenium IDE:

- Dalla homepage, si clicca su "Find Owner", si clicca "Add Owner", si compila il form con particolare attenzione al numero di telefono uguale a 333333333.
- Dalla homepage, si clicca su "Find Owner", si clicca "Find Owner", si clicca su "Mario Rossi", si clicca su "Add Pet", si compila il form con particolare attenzione al nome dell'animale (ovver "Source").
- Dalla homepage, si clicca su "Find Owner", si clicca "Find Owner", si clicca su "Mario Rossi", si clicca su "Add Visit" in riferimento all'animale "Source", si compila il form con particolare attenzione alla data (2023-01-01).

Abbrevieremo nel resto del capitolo questi source test case registrati come ST1,ST2,ST3.

Andiamo a vedere le regole che sono state scelte per effettuare le variazioni sull'input (Per la composizione di una regola vedi 3.4.2 e 3.4.3):

```
List<Rule> regole = new ArrayList<>();
Rule regola1 = new Rule("http://localhost:8080/owners\\?lastName=", "click",
    "linkText=Mario␣Rossi",
     "linkText=Mario␣Rossi|linkText=Ciro␣Esposito|linkText=Mario␣
         Bianchi|linkText=Michele␣Russo|linkText=Francesca␣Ferrara",
     "SELECTOR");
Rule regola2 = new Rule("http://localhost:8080/owners/[0-9]*/pets/new", "type",
    "id=name",
     "([A-z]|[0-9]){10}[?!]", "VALUES");
Rule regola3 = new Rule("http://localhost:8080/owners/new", "type", "id=telephone",
    "([0-9]){8}[123!]",
     "VALUES");
Rule regola4 = new Rule("http://localhost:8080/owners/[0-9]*/pets/[0-9]*/visits/new",
    "type", "id=date",
     "20[0-2][0-9]-0[123456789]-[0-3]{2}", "VALUES");
regole.add(regola1);
regole.add(regola2);
regole.add(regola3);
regole.add(regola4);
```

- regola1: L'url identifica la pagina con la tabella dei proprietari; fà si che, invece di cliccare sul proprietario Mario Rossi, si clicchi su uno a caso di altri proprietari (Ciro Esposito, Mario Bianchi etc...).
  Da notare come questa regole verrà applicata sia per la ST2 che per la ST3.

- regola2: L'url indentifica una creazione di un nuovo animale di un qualsiasi proprietario; fà si che, quando si scrive sul campo relativo al nome dell'animale, venga invece creata una sequenza casuale di lettere e numeri, con particolarità, l'uso o di un punto interrogativo oppure esclamativo alla fine del nome.
  Questa regola è usata per la ST2.
- regola3:L'url indentifica una creazione di un nuovo proprietario; fa si che, quando si scrive il numero di telefono di questo nuovo proprietario, esso sia una sequenza casuale di numeri, con particolare riferimento all'ultimo numero che potrebbe anche essere un punto escalamativo.
  La regola è usata per la ST3.
- regola4: L'url identifica una creazione di una nuova visita; fa si che, quando si scrive sul campo relativo alla data, venga invece inserita una data che possa variare dall'anno 2000 fino al 2022.
  Questa regola è usata per la ST3.

4.2 Seeding di fault

Per mostrare le potenzialità di Morfify sono state introdotti dei seeding di fault volontariamente (anche detti mutanti), per poter verificare che il nostro ambiente segnali correttamente dei Follow-Up test case con differenze sul Source (Verificare che i Warning vengano individuati).

Queste porzioni di codice sono presenti nel codice sorgente di PetClinic e sono state modificate per inserire volutamente dei lanci di eccezione; ricordiamo che PetClinic si trova su GitHub ed è Open Source.

```
 @PostMapping("/pets/new")
public String processCreationForm(Owner owner, @Valid Pet pet, BindingResult result,
    ModelMap model) {
   if (pet.getName().contains("!"))
      throw new RuntimeExecption(Qualcosa e andato storto nell inserimento del nuovo
          animale);
   if (StringUtils.hasLength(pet.getName()) && pet.isNew() && owner.getPet(pet.getName(),
       true) != null) {
      result.rejectValue("name", "duplicate", "already␣exists");
   }
   owner.addPet(pet);
   if (result.hasErrors()) {
      model.put("pet", pet);
      return VIEWS_PETS_CREATE_OR_UPDATE_FORM;
   }
   else {
      this.pets.save(pet);
      return "redirect:/owners/{ownerId}";
   }
}
```

Ci troviamo nella classe PetController, e per chiunque conosca un minimo il framework Spring, nonché l'uso delle annotazioni bind che mette a disposizione quest'ultimo, questo metodo viene eseguito dopo l'invio del form di inserimento di un nuovo animale lato client.
E' stato inserito un controllo per verificare se il nome dell'animale contenga un punto esclamativo, se così fosse verrebbe lanciata un'eccezione lato server.

Quando PetClinic lancia un'eccezione ci si ritrova nella schermata seguente.

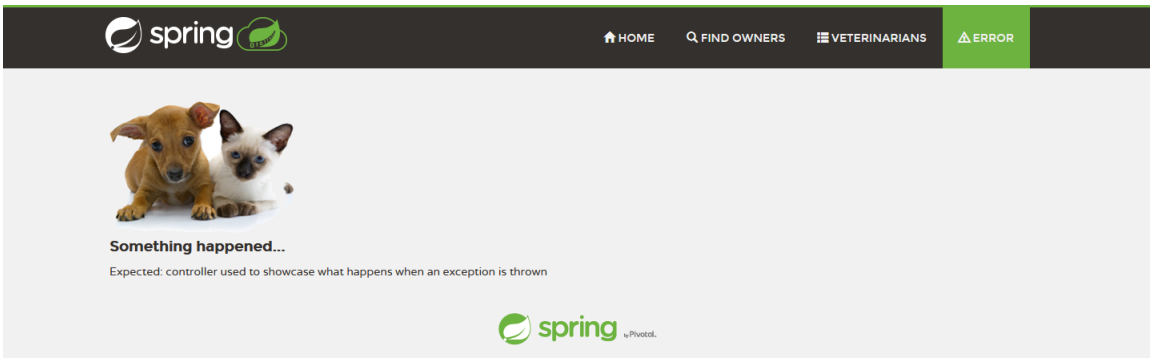


Fig. 18: Pagina di errore di PetClinic

Se durante l'esecuzione di uno dei follow-up test cases si arrivi a una schermata del genere, è chiaro immaginare come quel caso di test sia quanto meno un Warning, dato che il confronto tra una pagina del genere di errore e una che dovrebbe costituire il successo dell'operazione che si voleva portare a termine risulteranno necessariamente non "quasi-simili".

```
   // Spring MVC calls method loadPetWithVisit(...) before processNewVisitForm is called
  @PostMapping("/owners/{ownerId}/pets/{petId}/visits/new")
  public String processNewVisitForm(@Valid Visit visit, BindingResult result) {
     if (visit.getDate().compareTo(
           new
               Date(System.currentTimeMillis()).toInstant().atZone(ZoneId.systemDefault()).toLocalDate())
               < 0)
        throw new RuntimeExecption(Qualcosa e andato storto nell inserimento della visita);
     if (result.hasErrors()) {
        return "pets/createOrUpdateVisitForm";
     }
     else {

        this.visits.save(visit);
        return "redirect:/owners/{ownerId}";
     }
  }
```

Un seeding di fault è stato inserito anche nell'inserimento di una visita di un animale.
In particolare, se la data inserita è inferiore della data corrente, verrà lanciata un'eccezione.

4.3 Esempio d'uso

Esaminiamo l'esecuzione dei tre casi di test sopra descritti da Morfify e vediamo quali Follow-Up test cases verranno prodotti con le regole che abbiamo imposto.

Ci aspettiamo creazioni di Follow-Up test case che in qualche modo siano eterogenei tra di loro, dato che si è cercato di fare in modo che la probabilità di entrare nella porzione di codice mutante è proporzionale a quella della corretta esecuzione.

4.3.1 Input

Ricordiamo gli input che vengono chiesti da Morfify (Vedi 3.1 e 3.6):

- Source test case: Sono stati dati i test registrati con Selenium IDE ST1,ST2,ST3.
  Questi test sono tutti presenti nel file "PetclinicSourceTest.side" presente nella cartella "testFile".
- Lista di regole: Nel main è stato possibile creare oggetti di tipo Rule, nonchè una lista di questi oggetti.
  Tramite il metodo createFollowupTestCase è stato possibile passare all'esecuzione di un follow-up una lista di regole.
- Numero di iterazioni: è una costante presente nel main è indica quanti follow-up test cases devono essere generati (Nel nostro caso tre per ogni Source test).

4.3.2 Output

L'output è disponibile nella cartella "Output" del nostro ambiente, esso è diviso in due cartelle:

- Source: Comprende tutti i Source test case eseguiti correttamente (Se si è presentato un ipotetico Faulty non dovrebbe esserci in questa cartella).
- Follow-Up: Comprende tutti i Follow-Up test cases.

Esaminiamo il contenuto della cartella Follow-up:

- E' composto da un'insieme di cartelle chiamate "Source-N.1","Source-N.2","Source-N.3"; esse racchiudono le pagine e gli screenshot effettuati durante l'esecuzione del Source test case corrispondente.
  L'ordine di esecuzione, nonché banalmente l'identificazione numerica, avviene in ordine di test presenti nel file .side.
- Dentro ognuna di queste cartelle, sono presenti altre tre cartelle di nome "1!","2","3!".
  Il numero rappresenta il Follow-Up (in ordine cronologico) ed il punto esclamativo, come già spiegato in 3.5.2, rappresenta un Follow-Up test case che non fa matching con il Source, ovvero un Follow-up test case che ha al suo interno delle pagine ritenute non "quasi-simili" con quelle eseguite e catturate dal Source.

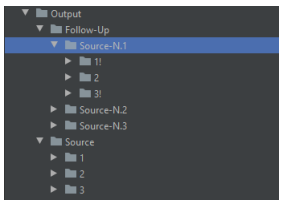


Fig. 19: Cartelle prodotte dall'output di PetClinic

Mentre, dentro ognuno di queste cartelle è presente:

- Sorgente della pagina: per ogni "passaggio" tra una pagina e l'altro (vedi 3.5.1) viene creato un file .html contenente la DOM della pagina.
- Screenshot della pagina: viene effettuato uno screenshot della nuova pagina che si sta visitando e salvato in formato .png
- File testcase: è un file con estensione .txt che racchiude le informazioni chiave dell'oggetto test case che è stato creato da Morfify; è la serializzazione dell'oggetto test case, con l'aggiunta di un commento, in base che si tratti di un caso di test risultato Success, Warning o Faulty (Vedi 3.5.3).

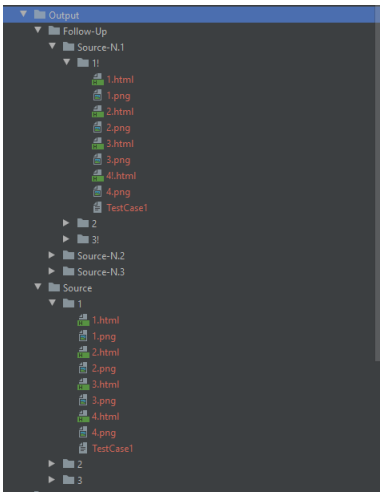


Fig. 20: Cartelle prodotte dall'output di PetClinic (Con file)

Il sorgente e gli screenshot sono serializzati in ordine cronologico, per esempio il file "1.html" ed "1.png" sono le prime pagine che vengono individuate dall'esecuzione del test (URL iniziale, l'homepage).

Questi file possono presentare un punto esclamativo
dopo il numero della pagina visitata (Es: "1!.html"), queste sono le pagine che sono risultate troppo diverse rispetto a quelle confrontate con il Source test case.
E' intuitivo capire che se si vuole sapere cos'è andato storto con l'esecuzione di un Follow-Up test case bisognerà confrontare gli screenshot (per poi magari in un secondo momento il file HTML) della pagina incriminata con quella del Source test case.

Esaminiamo un paio di output prodotti dal nostro esempio d'uso specifico: Sono stati prodotti un totale di 9 Follow-Up test cases (3 per ogni Source in input) di cui solo 2 hanno avuto un match positivo con il loro rispettivo Source (Success), esaminiamo prima questi tipi di test.

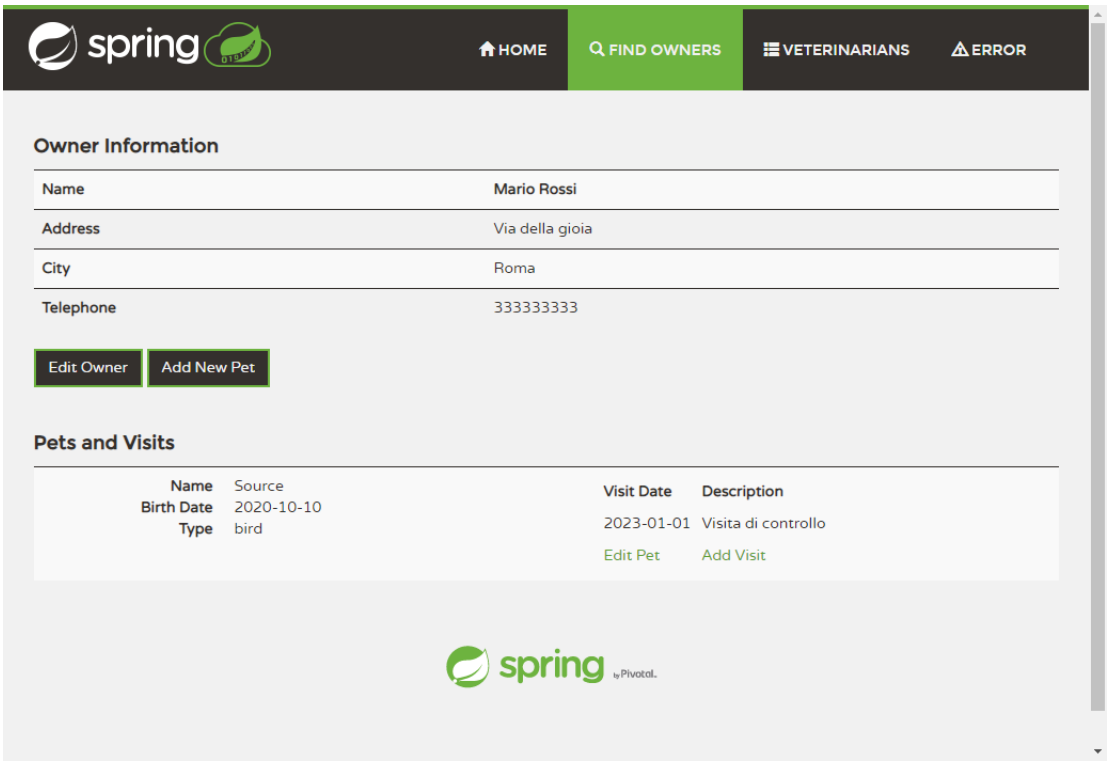


Fig. 21: ST3: Esecuzione Source test case

Questa è l'ultima pagina che viene catturata dal WebDriver prima di concludere il test ST3: si vede che il Source test case è riuscito ad inserire normalmente la visita da noi desiderata.

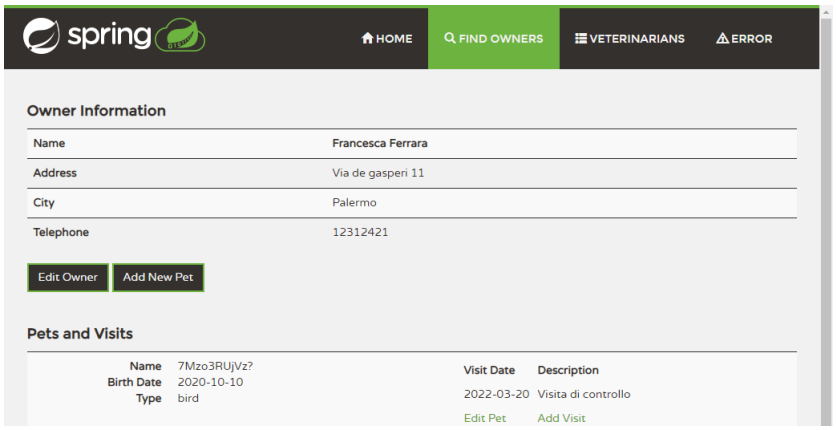


Fig. 22: ST3: Esecuzione Follow-Up test case numero 1

Questa è l'ultima pagina che viene catturata dal WebDriver prima di concludere l'esecuzione del Follow-Up del test ST3: si vede che, anche se cambia il nome del proprietario e animale le pagine risultino ancora "quasi-simili".
Questo è assolutamente voluto, dato che vorremmo identificare delle differenze più sulla struttura di una pagina e non sul contenuto, che per la nostra MR, vogliamo che possa cambiare.

Notiamo anche che la data inserita porta l'anno 2022, quindi il nostro seeding di fault non può essere ancora trovato (Vedi 4.2).

Vediamo adesso sempre un Follow-up appartenente al ST3, ma segnato da Morfify come Warning.

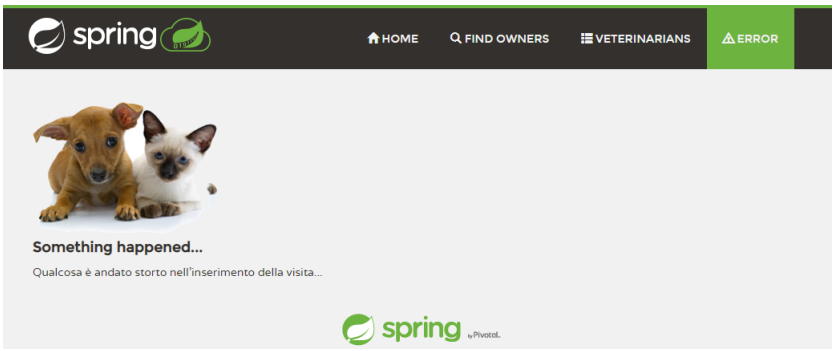


Fig. 23: ST3: Esecuzione Follow-Up test case numero 3, ultima pagina visitata

Questo messaggio di errore si è presentato dopo che si è provato a creare una visita con data inferiore a quella attuale (In questo caso la data portava l'anno 2015).
E' stata quindi lanciata un'eccezione lato server che ha portato a questa pagina di errore, essendo questa pagina chiaramente diversa dalla figura mostrata precedentemente, l'esecuzione di questo Follow-up test case è risultata fallita (Warning).

5 Conclusioni

In questo capitolo riassumeremo quello che abbiamo visto nei capitoli precedenti e cercheremo di rispondere alle domande poste nell'introduzione.

Vedremo quando e come usare la tecnica di Metamorphic Testing,
faremo delle riflessioni sull'applicativo creato, cos'è in grado di offrire e quali sono le features future che verranno implementate.

5.1 Verifica degli obbiettivi

La prima domanda che ci si era posti era: "Quali sono i vantaggi nell'uso della tecnica di Metamorphic Testing rispetto ad altre tecniche di testing?".

Durante tutto il capitolo 1, abbiamo visto quando usare test automatici e in che modo la tecnica di Metamorphic Testing, tramite la creazione di relazioni metamorfiche, riduce fortemente i tempi di sviluppo rispetto a tecniche di testing tradizionali.

La seconda domanda invece chiedeva l'utilità nell'uso della tecnica di Metamorphic testing per web application.
Per rispondere a questa domanda sono state riportati dati raccolti da diversi articoli di ricerca che da una popolazione di programmi software in svariati domini di applicazione, il settore dei Web Services è quello con il maggior numero di programmi utilizzanti la tecnica di MT (Vedi 1.6 e 1.7).

La ragione è presto detta: la tecnica di MT è particolarmente efficace per testare funzionalità ripetitive, semplici o con grandi quantità di input. Quindi la tecnica di MT trova spazio in tante piccole funzionalità come svariati tipi di form, ricerche, filtri e ordinamenti etc...

Rispondiamo alle domande inerenti a Morfify e cosa si è riuscito a rispettare dei requisiti di sistema.
La terza domanda chiede: "Quali sono i vantaggi nell'uso della tecnica di Metamorphic Testing rispetto ad altre tecniche di testing?".
Morfify è un ambiente di testing utilizzante la tecnica di MT, Vogliamo che venga usato in realtà software di vari calibri e che si presenti come uno strumento efficace, completo e facile da usare.
Al tal fine, è stato esposto come, grazie all'uso indiretto di tecnologie come Selenium, Morfify riesce ad essere usato per verificare casi di test di applicativi complessi e non.
Può essere usato potenzialmente per testare qualsiasi funzionalità di un sito web che prevede una MR incentrata sulla similarità delle pagine.

5.2 Sviluppi futuri

Durante tutto lo studio sono state fatte menzioni a parti di applicativo, nonché alcune intere funzionalità dell'ambiente, non ancora sviluppate.

Ricordiamo che Morfify è su GitHub[11] e che eventuali implementazione delle funzionalità sotto citate saranno disponibili e documentate qui:

- Creazione di un DSL: Sebbene Morfify non utilizzi direttamente il tool Selenium IDE, attualmente è parecchio scomoda la creazione dei casi di test senza l'utilizzo del file JSON creato dal progetto di questo tool.
  E' stato implementato un design pattern a doc per poter aggiungere in un secondo momento un tipo di creatore di Source Test case diverso dal parser JSON; questo potrebbe avenire con un linguaggio di dominio apposito, con comandi e sintassi testuali a righe di comando (Azione 1: Clicca bottone X, Azione 2: Scrivi in form A la stringa "X" etc...)

[11] https://github.com/Noctino52/MetamorphicTesting

- Supporto per Firefox e Microsoft Edge: Morfify attualmente permette l'esecuzione di casi di test solamente su Chrome e non è stato fatto in modo di poter cambiare tanto facilmente browser.
- Uso dei selettori in targets: Quando si cerca di eseguire un test case è comune che un elemento di una pagina durante la creazione del Source test case venga riconosciuto e localizzato da Selenium WebDriver; ma che durante l'esecuzione il selettore presente in target non venga trovato.
  In queste circostanze, invece che far lanciare un eccezione al WebDriver, potrebbe essere eseguito un selettore presente nella lista dei selettori presenti in targets.
- Verifica con un parser DOM: L'ultima componente di Morfify prevede la verifica delle similarità tra le pagine tramite l'utilizzo dell'algorimto delle distanze minime di Levinstain.
  Come già spiegato in 3.5.2 non è un buon algoritmo per le nostre circostanze, poichè si limita ad esaminare il contenuto raccolto dalla quarta compenente come semplice stringhe.
  Quello che vorremmo noi è un parser DOM per ottenere informazioni dettagliate sulle pagine, andando a identificare una differenza tra quest'ultime tramite l'analisi dei tag HTML contenenti le pagine, piuttosto che una mera differenza tra stringhe.

- Supportare casi di test con if, cicli for e while: Alcuni comandi che permette di svolgere Selenium IDE durante la registrazione del test, nonchè i comandi che permette di eseguire l'API di Selenium WebDriver, non sono supportati da Morfify, questo perchè per il momento ci limitiamo a verificare casi di test per pagine molto semplici e con un numero di iterazioni sulle operazioni da fare ripetibili manualmente nel caso di test.
  Per completezza, sarebbe opportuno implementare i comandi mancanti.
- Faulty test case: Se si verifica un problema durante l'esecuzione di un test case, Morfify si interromperà e non verrà prodotto un output relativo a tutti i test case precedentemente eseguiti. Sarebbe invece opportuno innanzitutto individuare queste categorie di test case, per poi includere anche loro nell'output ed aggiungere in quest'ultimo informazioni relative all'eccezione lanciata.

Bibliografia

•

[1] Upulee Kanewala James M. Bieman Asa Ben-Hur. "Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels". In: *National Institute Of General Medical Sciences.* (2015). DOI: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1594.

[2] Dave Towey Huai Liu Fei-Ching Kuo. "How Effectively Does Metamorphic Testing Alleviate the Oracle Problem?" In: *IEEE Transactions on Software Engineering* (2014). DOI: https://ieeexplore.ieee.org/document/6613484.

[3] Dan Hao Jie Zhang Junjie Chen. "Search-based inference of polynomial metamorphic relations". In: *ASE '14: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering* (2014). DOI: https://dl.acm.org/doi/10.1145/2642937.2642994.

[4] Alex Di Fonzo Joe Fernandes. "When to Automate Your Testing (and When Not To)". In: *Oracle* (2017).

[5] Ana B. Sanchez Sergio Segura Gordon Fraser. "A Survey on Metamorphic Testing". In: *IEEE Transactions on Software Engineering* (2016). DOI: https://doi.org/10.1109/TSE.2016.2532875.

[6] Zhi Quan Zhou Sergio Segura Dave Towey. "Metamorphic Testing: Testing the Untestable". In: *IEEE Software* (2020). DOI: https://doi.org/10.1109/MS.2018.2875968.

[7] S.M. Yiu T.Y. Chen S.C. Cheung. "Metamorphic Testing: A New Approach for Generating Next Test Cases". In: *The Hong Kong University of Science and Technology* (1998). DOI: https://arxiv.org/abs/2002.12543.

[8] Yuchi Tian. "DeepTest: Automated Testing ofDeep-Neural-Network-driven Autonomous Cars". In: *ICSE '18: 40th International Conference on Software Engineering* (2018). DOI: https://doi.org/10.1145/2666356.2594334.

[9] Pak- Lok Poon Xiaoyuan Xie Tsong YuehChen. "METRIC: METamorphic Relation Identification based on the Category-choice framework". In: *IEEE Transactions on Software Engineering* (2016). DOI: https://doi.org/10.1016/j.jss.2015.07.037.

[10] Mehrdad Afshari Vu Le. "Compiler validation via equivalence modulo inputs". In: *ACM SIGPLAN Notices* (2014). DOI: https://doi.org/10.1145/2666356.2594334.