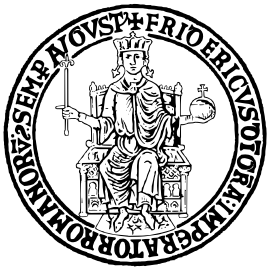
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Triennale in Informatica

Fragilità dei test per applicazioni web template based tramite generazione di mutazioni: sperimentazione

Relatore
Prof. Porfirio Tramontana

Candidato
Simone Liberti
N86004252

Anno Accademico 2024–2025

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

Fragilità dei test per applicazioni web template based tramite generazione di mutazioni: sperimentazione

Relatore
Prof. Porfirio Tramontana

Candidato
Simone Liberti
N86004252

Anno Accademico 2024–2025

Abstract

End-to-End (E2E) testing plays a pivotal role in ensuring the quality of web applications. However, this approach presents a significant challenge: even minor variations within the user interface can lead to test failures (test breakage), consequently increasing maintenance costs. The fragility of E2E tests is closely related to the strategy adopted for locating elements within the Document Object Model (DOM). This thesis presents an experimental process aimed at analyzing, generating, and testing mutations on graphical user interfaces based on *AngularJS*. Specifically, it aims to evaluate the robustness of tests and their respective mutations by comparing seven different XPath-based locator strategies: Absolute, Relative, Hook-Based, ROBULA, ROBULA+, Selenium, and Katalon. To achieve this goal, a tool was designed and developed to automate both the generation of mutants—through the static analysis of the application's source files—and the subsequent test execution phase. This project builds upon previous research, extending the work described in the theses "Techniques for generating mutants for template-based web applications" (A. Volpe) and "An experimental process for evaluating locator fragility in the context of E2E testing" (A. Paolella). Following the generation and execution process, the obtained results were validated and verified in order to rigorously quantify the fragility of the analyzed locators and the actual correctness of the generated mutants.

Sommario

Il testing End-to-End (E2E) riveste un ruolo di primaria importanza nel garantire la qualità delle applicazioni web. Tuttavia, tale approccio presenta una criticità significativa: minime variazioni all'interno dell'interfaccia utente possono causare il fallimento dei test (test breakage), incrementando di conseguenza i costi di manutenzione. La fragilità dei test E2E è strettamente correlata alla strategia adottata per la localizzazione degli elementi all'interno del Document Object Model (DOM). Il presente elaborato illustra un processo sperimentale volto ad analizzare, generare e collaudare mutazioni su interfacce grafiche basate su *AngularJS*. Nello specifico, si intende valutare la robustezza dei test e delle relative mutazioni confrontando sette diverse strategie di localizzazione basate su XPath: Absolute, Relative, Hook-Based, ROBULA, ROBULA+, Selenium e Katalon. Al fine di perseguire tale obiettivo, è stato progettato e sviluppato uno strumento in grado di automatizzare sia la generazione dei mutanti — tramite l'analisi statica dei file dell'applicazione — sia la successiva fase di esecuzione dei test. Il presente progetto si pone in continuità con le ricerche condotte in precedenza, estendendo i lavori descritti nelle tesi "Tecniche di generazione di mutanti per applicazioni web template-based" (A. Volpe) e "Un processo sperimentale per la valutazione della fragilità dei locatori nel contesto di testing E2E" (A. Paolella). A valle del processo di generazione ed esecuzione, i risultati ottenuti sono stati oggetto di validazione e verifica, allo scopo di quantificare in modo rigoroso la fragilità dei locatori analizzati e l'effettiva correttezza dei mutanti prodotti.

Contents

Introduzione

Il collaudo del software è una fase critica nel ciclo di vita dello sviluppo, e nel contesto delle applicazioni web, il testing End-to-End (E2E) rappresenta l'approccio più completo per validare l'intero sistema. Simulando l'interazione reale di un utente con l'interfaccia grafica (GUI), i test E2E verificano che tutti i componenti, dal *front-end* al *back-end*, comunichino correttamente.

Testing End-to-End e cause di fragilità

Nel panorama dell'automazione del software, i test E2E si dividono tipicamente in diverse tipologie e approcci, tra cui:

- **Test basati su script:** I test vengono scritti programmaticamente (es. tramite framework come *Selenium* o *Cypress*) interagendo direttamente con il Document Object Model (DOM) per individuare ed eseguire azioni sugli elementi della pagina.
- **Test Record-and-Playback:** Strumenti che registrano le azioni dell'utente e generano automaticamente lo script di test. Sebbene siano di facile implementazione, generano spesso locatori molto rigidi.
- **Visual Regression Testing:** Test che confrontano screenshot dell'interfaccia per rilevare anomalie grafiche, indipendentemente dalla struttura del codice sottostante.

Nonostante la loro importanza, i test E2E basati sul DOM (i più diffusi nel settore) soffrono di un'elevata **fragilità**. Questa criticità si manifesta quando un test fallisce non a causa di un reale difetto nel sistema (bug funzionale), ma per modifiche strutturali o grafiche della GUI. Risulta di fondamentale importanza distinguere le cause di tali fallimenti, differenziando tra fragilità e obsolescenza:

- **Obsolescenza dei test:** Si verifica quando un test fallisce a causa di una reale e voluta modifica nella logica di business o nei requisiti funzionali dell'applicazione. Ad esempio, se un flusso di pagamento a tre step viene ridotto a due, il test originale fallirà perché cerca di validare un comportamento che non fa più parte del sistema. In questo scenario, il fallimento è "giustificato": il test è semplicemente diventato obsoleto e necessita di essere aggiornato o rimosso.

- **Fragilità dei test:** Si manifesta quando un test fallisce a causa di modifiche puramente strutturali o estetiche della GUI, sebbene la funzionalità sottostante rimanga inalterata e corretta. In questo caso, il test riporta un "falso positivo". Le cause principali di questo fenomeno includono l'evoluzione continua del design, l'utilizzo di framework che generano attributi dinamici a runtime e, soprattutto, l'elevata dipendenza (coupling) dei locatori dalla struttura dell'albero DOM (come nel caso degli XPath assoluti).

Il Mutation Testing come strumento di benchmarking

Per misurare e mitigare la fragilità dei test E2E, la ricerca empirica in ambito software engineering si avvale della *Mutation Analysis* (o *Mutation Testing*). Tradizionalmente usata per valutare la qualità dei test unitari inserendo errori intenzionali nel codice sorgente, questa tecnica può essere adattata alle interfacce web. Nel contesto di questa tesi, un mutante è una versione alterata dell'interfaccia utente originale in cui viene introdotta una singola modifica sintattica o strutturale nel DOM (ad esempio, la rimozione di un attributo, il cambiamento di un tag o l'alterazione di una classe CSS). L'utilizzo dei mutanti permette di stabilire un benchmark rigoroso: eseguendo la suite di test E2E sia sull'applicazione originale sia sulle versioni mutate, è possibile quantificare oggettivamente quanto una specifica strategia di localizzazione sia resiliente ai cambiamenti. Se un locatore riesce a individuare l'elemento target anche nell'interfaccia mutata, si dimostra robusto; se fallisce, si dimostra fragile. Questo approccio trasforma la valutazione della fragilità da un concetto qualitativo a una metrica quantitativa e misurabile.

Obiettivo della Tesi

Alla luce delle criticità esposte, la presente tesi si pone l'obiettivo primario di definire ed eseguire un processo sperimentale per valutare la robustezza di diverse strategie di localizzazione nel testing E2E, applicate ad applicazioni web basate sul framework *AngularJS*. Nello specifico, gli obiettivi del lavoro si articolano nei seguenti punti:

- **Analisi comparativa delle strategie XPath:** Valutare e confrontare empiricamente la fragilità di sette specifiche strategie di localizzazione: Absolute, Relative, Hook-Based, ROBULA, ROBULA+, Selenium e Katalon.
- **Sviluppo di uno strumento di automazione:** Progettare e implementare un tool in grado di automatizzare il processo di generazione dei mutanti attraverso l'analisi statica dei file dell'applicazione, riducendo l'overhead manuale.
- **Esecuzione automatizzata dei test:** Automatizzare la fase di collaudo, eseguendo le diverse strategie di localizzazione contro le interfacce mutate per raccogliere i dati di test.

- **Validazione dei risultati:** Verificare la correttezza dei mutanti generati e analizzare i dati raccolti per quantificare in modo rigoroso le prestazioni e i limiti di ciascun locatore.

Struttura del documento

Il presente lavoro di tesi è articolato in cinque capitoli, organizzati come di seguito illustrato:

- **Capitolo 1:** Definisce il contesto teorico e tecnologico. Analizza il funzionamento delle applicazioni web basate su template, lo scopo del testing E2E e la distinzione tra le diverse cause di fallimento dei test. Introduce inoltre gli strumenti adottati e descrive in dettaglio le strategie di localizzazione oggetto di studio.
- **Capitolo 2:** Illustra le metodologie e i processi utilizzati per la generazione dei locatori da sottoporre ad analisi.
- **Capitolo 3:** Descrive l'architettura e il funzionamento dello strumento sviluppato per l'automazione, dettagliando le modalità pratiche di esecuzione dei test.
- **Capitolo 4:** Presenta la test suite impiegata per la sperimentazione, definendone le caratteristiche e i casi di test selezionati.
- **Capitolo 5:** Riporta la validazione dei risultati emersi dal processo sperimentale, discutendo criticamente le evidenze raccolte e rispondendo accuratamanete alle domande di ricerca poste.
- **Capitolo 6:** Riporta le conclusioni finali del lavoro e fornisce uno spunto su possibili lavori futuri.

Si precisa fin d'ora che, per garantire la totale trasparenza e riproducibilità dello studio, l'intero ecosistema software sviluppato (inclusi gli script di mutazione e la suite di collaudo) e i dataset grezzi estratti sono stati resi pubblicamente disponibili. La descrizione dettagliata dell'organizzazione di tale materiale tecnico è riportata nell'**Appendice A**.

–1–
Contesto Teorico e Tecnologico

1.1 Applicazioni Web Basate su Template

Lo sviluppo di applicazioni web moderne è un processo ingegneristico complesso che richiede metodologie rigorose per garantire affidabilità, sicurezza e un'esperienza utente ottimale. In questo contesto, il collaudo del software (*software testing*) assume un ruolo centrale all'interno del ciclo di vita dello sviluppo. Esso si configura come un processo sistematico e strutturato, volto a valutare e verificare che un prodotto informatico soddisfi i requisiti prestabiliti e si comporti come atteso in molteplici condizioni operative.
A differenza degli approcci tradizionali basati sull'elaborazione server-side (come *Java Servlet* o *JSP*), la maggior parte delle soluzioni moderne si fonda sul principio di separazione degli strati: logica di business, presentazione e dati. Tale disaccoppiamento favorisce la modularità, consentendo di ottenere applicazioni più facilmente sviluppabili, scalabili e manutenibili. Attualmente, gran parte delle applicazioni web adotta il pattern architetturale MVC (*Model-View-Controller*) o le sue varianti, come MVVM (*Model-View-ViewModel*). In questo panorama, ha acquisito notevole popolarità l'utilizzo di framework e architetture di sviluppo basate su template e template engine (motori di rendering). Un template ha il compito specifico di definire la presentazione visiva delle informazioni. Esso è tipicamente costituito da codice di markup standard arricchito da blocchi di tag e direttive all'interno dei quali vengono iniettati i dati dinamici. Durante l'esecuzione, il template engine elabora questi componenti: riceve in input i dati dal modello e le stringhe tokenizzate del template, compila le direttive e produce in output il codice HTML renderizzato finale.
È proprio questa generazione dinamica del Document Object Model (DOM) da parte dei template engine a introdurre un notevole livello di variabilità nell'interfaccia utente finale, ponendo le basi per le sfide di automazione e per le criticità legate alla fragilità dei test che verranno esplorate nelle sezioni successive.

1.2 Il testing End-to-End (E2E): Obiettivi e Caratteristiche

Il testing End-to-End (E2E) rappresenta una metodologia di collaudo software che mira a validare il comportamento di un'applicazione simulando le interazioni di un utente reale dall'inizio alla fine del suo percorso. L'obiettivo primario di questa tecnica non è la verifica di una singola funzione o di un modulo isolato, bensì il controllo dell'intero flusso operativo dell'applicazione, assicurandosi che tutti i sottosistemi comunichino correttamente tra loro. Mentre i test unitari e di integrazione si concentrano sul codice sorgente e sulle API interne, i test E2E operano tipicamente secondo un approccio black-box. Il sistema viene testato esclusivamente attraverso la sua Interfaccia Utente (UI), verificando che le azioni compiute sul front-end si traducano in elaborazioni corrette nel back-end, interrogazioni esatte al database e risposte coerenti da eventuali servizi di terze parti (come gateway di pagamento o API esterne). A titolo esemplificativo, un test E2E non si limita a verificare se il componente di login accetta un input, ma automatizza l'intero scenario di utilizzo: inserimento delle credenziali nella maschera web, click sul pulsante di accesso, attesa della risposta del server e verifica che l'utente venga correttamente reindirizzato alla propria area riservata con i permessi adeguati. Affinché queste operazioni possano essere eseguite in modo efficiente e ripetibile all'interno di una pipeline di sviluppo, l'interazione manuale dell'utente viene sostituita da script automatizzati. Tali script si avvalgono di framework di automazione che pilotano direttamente il browser web (DOM interaction), localizzando gli elementi dell'interfaccia ed eseguendo eventi come click, digitazione di testo e navigazione. Tuttavia, proprio questa stretta dipendenza dall'interfaccia grafica e dalla struttura del Document Object Model (DOM) rappresenta il punto debole di tale approccio, rendendo i test E2E particolarmente suscettibili a fallimenti non legati a reali difetti del software.

1.3 Dinamiche di Fallimento dei Test: Fragilità vs Obsolescenza

Come precedentemente accennato, l'elevato livello di interazione con l'interfaccia utente rende i test E2E intrinsecamente complessi e soggetti a frequenti fallimenti. In ambito accademico e industriale, è di fondamentale importanza analizzare e classificare le cause alla base di un test fallito (*test failure*), al fine di adottare le corrette strategie di manutenzione e mitigazione. Le principali dinamiche di fallimento si possono categorizzare in tre macro-aree:

- **Instabilità (Flakiness):** Si verifica quando un test produce risultati non deterministici, alternando fallimenti e successi in esecuzioni consecutive pur in assenza di modifiche al codice sorgente. Questo comportamento è frequentemente riconducibile a problemi di temporizzazione (timing issues), asincronicità delle richieste

di rete, tempi di caricamento imprevedibili o dipendenze da stati globali. La flakiness compromette significativamente la fiducia del team di sviluppo nell'affidabilità dell'intera suite di collaudo.

- **Obsolescenza:** Si manifesta quando un test fallisce a causa di una reale e voluta modifica nella logica di business o nei requisiti funzionali dell'applicazione. Ad esempio, qualora un flusso di registrazione a tre passaggi venga ottimizzato e ridotto a due, lo script di test originale fallirà cercando di validare un comportamento che non fa più parte del sistema. In questo scenario, il fallimento è giustificato e atteso: il test è divenuto obsoleto e necessita di essere aggiornato o rimosso per allinearsi ai nuovi requisiti.
- **Fragilità:** Data l'elevata sensibilità alle variazioni dell'interfaccia grafica, anche modifiche marginali o puramente estetiche della struttura HTML possono causare il fallimento del test (*test breakage*). Questo fenomeno genera numerosi falsi positivi, richiedendo continui e onerosi interventi di manutenzione da parte degli sviluppatori per aggiornare le strategie di localizzazione degli elementi.

Il presente elaborato concentra il proprio campo di indagine esclusivamente sul problema della fragilità. Nello specifico, la ricerca mira ad analizzare come l'adozione di differenti strategie per l'identificazione degli elementi web (locatori) possa influenzare la resilienza degli script di collaudo di fronte all'evoluzione strutturale, e non funzionale, dell'interfaccia utente.

1.4 Strumenti Tecnologici a Supporto della Sperimentazione

Per condurre l'analisi empirica e automatizzare il processo di collaudo, la presente ricerca si è avvalsa di tecnologie consolidate nel panorama del software testing. Di seguito vengono descritti i principali framework di terze parti utilizzati come base tecnologica per l'interazione con le interfacce web e la valutazione delle strategie di localizzazione:

- **Selenium WebDriver (Java):** Rappresenta il framework *de facto* per l'automazione dei browser web. Nel contesto di questo lavoro, i *binding* in linguaggio Java di Selenium sono stati impiegati sia come motore di esecuzione automatizzata dei test (pilotando il browser per interagire con il DOM), sia come strategia nativa per l'estrazione di specifici locatori.
- **Strumenti per l'estrazione dei locatori:** Per ottenere gli XPath da sottoporre a test, sono stati combinati approcci manuali e automatizzati. L'estensione per browser **SelectorsHub** (in ambiente Google Chrome) è stata utilizzata per ricavare locatori assoluti e relativi. Parallelamente, il motore di **Katalon** è stato impiegato per catturare i locatori generati dalla sua euristica proprietaria, mentre per le strategie

algoritmiche più complesse (**ROBULA, ROBULA+** e **Hook-Based**) sono stati implementati appositi script dedicati. La metodologia dettagliata di generazione di tali locatori sarà oggetto del Capitolo 2.

- **JSoup:** Una potente libreria Java progettata per il parsing, l'estrazione e la manipolazione di documenti HTML. JSoup è in grado di interpretare il codice sorgente restituendo un albero DOM navigabile, permettendo di ricercare e alterare programmaticamente nodi e attributi.

Al fine di orchestrare queste tecnologie all'interno del processo di *Mutation Analysis*, è stato progettato e sviluppato **uno strumento software ad hoc scritto in linguaggio Java**. Tale applicativo sfrutta JSoup per analizzare staticamente i file del progetto *AngularJS*, individuare gli elementi target e iniettare l'insieme di mutazioni previste. Successivamente, lo stesso applicativo pilota Selenium WebDriver per eseguire la suite di collaudo sulle interfacce alterate.

Essendo questo strumento il nucleo operativo dell'intero processo sperimentale, la descrizione dettagliata della sua architettura e delle sue logiche di funzionamento verrà ampiamente trattata nel **Capitolo 3** del presente elaborato.

1.5 Mutation Testing e Tipologie di Mutazioni Utilizzate

Il *Mutation Testing* (o Analisi delle Mutazioni) è una tecnica di collaudo avanzata, basata sull'iniezione intenzionale di difetti (*fault injections*) all'interno di un sistema software. Tradizionalmente impiegata nel contesto dei test unitari, questa metodologia prevede la creazione di molteplici versioni alterate del programma originale, definite **mutanti**. Lo scopo è valutare l'efficacia (*fault-findig capability*) di una suite di test: se un test fallisce eseguendo un mutante, si dice che ha "ucciso" il mutante, dimostrando di essere in grado di intercettare quell'errore; se il test passa, il mutante "sopravvive", evidenziando una potenziale lacuna nella suite di collaudo.

Tuttavia, nel contesto della presente tesi e del testing End-to-End (E2E), il paradigma della *Mutation Analysis* viene riadattato con un obiettivo diametralmente opposto: misurare la **resilienza** (o, inversamente, la fragilità) delle strategie di localizzazione. In questo scenario sperimentale, l'iniezione dei difetti non avviene a livello di logica di business, bensì a livello di presentazione, applicando alterazioni sintattiche e strutturali al Document Object Model (DOM) dell'applicazione web basata su *AngularJS*. Un test E2E che continua a superare l'esecuzione (*pass*) su un'interfaccia mutata dimostra che il locatore utilizzato (ad esempio, un particolare XPath) è sufficientemente robusto da resistere a quel cambiamento. Viceversa, un fallimento (*test breakage*) indica l'eccessiva fragilità del locatore rispetto all'evoluzione della UI.

1.5.1 Regole per le Mutazioni

Per generare i mutanti in modo sistematico, la letteratura scientifica di riferimento (tra cui i lavori in continuità con questo elaborato) definisce una serie di **operatori di mutazione**, ovvero regole formali di trasformazione applicabili ai nodi dell'albero HTML.

Tramite lo strumento di analisi statica sviluppato in Java (basato su JSoup), sono state automatizzate specifiche tipologie di alterazione, progettate per simulare i cambiamenti più comuni che avvengono durante l'evoluzione e la manutenzione di un'interfaccia utente. La Tabella 1.1 che segue elenca in modo dettagliato le regole di mutazione, con esempi pratici di codice prima e dopo. Il *Listing* 1.1 successivo, descrive la gerarchia dei tag su cui saranno applicate le mutazioni definite.

Table 1.1: Tipologie di Mutazioni.

Object Type	Change ID	Change Type Description	Original Code	Modified Code
Attribute	a	Attribute Value Modification	*< tag attr = "value" > text < /tag >*	*< tag attr = "newValue" > text < /tag >*
	b	Attribute Removal	*< tag attr = "value" > text < /tag >*	*< tag > text < /tag >*
	c	Attribute Identifier Modification	*< tag attr = "value" > text < /tag >*	*< tag newAttr = "value" > text < /tag >*
Text	d	Text Content Modification	*< tag attr = "value" > text < /tag >*	*< tag attr = "value" > newText < /tag >*
	e	Text Content Removal	*< tag attr = "value" > text < /tag >*	*< tag attr = "value" > < /tag >*
Tag	f	HTML Tag Movement (within a container)	*< div >* *< tag > < /tag >* *< tag attr = "value" > text < /tag >* *< /div >*	*< div >* *< tag attr = "value" > text < /tag >* *< tag > < /tag >* *< /div >*
	g	HTML Tag Movement (in any point of the HTML tree)	*< div >* *< tag attr = "value" > text < /tag >* *< tag > < /tag >* *< /div >*	*< tag attr = "value" > text < /tag >* *< div >* *< tag > < /tag >* *< /div >*
	h	HTML Tag Movement (between two templates)	*< template1 >* *< tag attr = "value" > text < /tag >* *< /template1 >* *< template2 >* *< /template2 >*	*< template1 >* *< /template1 >* *< template2 >* *< tag attr = "value" > text < /tag >* *< /template2 >*
	i	HTML Tag Removal	*< tag attr = "value" > text < /tag >*	*text*
	j	HTML Tag Type Modification	*< tag attr = "value" > text < /tag >*	*< newTag attr = "value" > text < /newTag >*
	k	HTML Tag Insertion	*< tag attr = "value" > text < /tag >*	*< tag > < /tag >* *< tag attr = "newValue" > text < /tag >*

Listing 1.1 Tipologie di Tag

```
1: <template>                    <!-- ϵ (template)    -->
2:   <p>                         <!-- γ (ancestor)    -->
3:     <div>                     <!-- β (parent)      -->
4:       <a onclick="go()">      <!-- α (target tag) -->
5:       <a href="#">            <!-- σ (sibling)     -->
6:     </div>
7:   </p>
8: </template>
```

1.6 Analisi delle strategie di Localizzazione (XPath)

Come evidenziato nelle sezioni precedenti, l'affidabilità di uno script di collaudo E2E dipende in larga misura dalla robustezza del meccanismo utilizzato per identificare gli elementi all'interno del Document Object Model (DOM). Il linguaggio standard universalmente adottato per interrogare e navigare l'albero HTML è XPath (*XML Path Language*).

Tuttavia, dato un singolo nodo target, esistono innumerevoli espressioni XPath valide per selezionarlo. Una "strategia di localizzazione" definisce l'approccio algoritmico o l'euristica con cui tale espressione viene costruita. Nelle sottosezioni seguenti vengono analizzate nel dettaglio le strategie di localizzazione oggetto della presente valutazione sperimentale, raggruppate per caratteristiche sintattiche e metodologie di generazione.

1.6.1 Locatori Assoluti e Relativi

Locatori Assoluti

Un locatore assoluto è rappresentato da un'espressione XPath che traccia l'intero percorso gerarchico, partendo dal nodo radice del documento (tipicamente il tag <html>) fino a raggiungere l'elemento target. Qualora uno o più nodi lungo il percorso presentino elementi fratelli (sibling) nell'albero HTML, vengono impiegati degli indici posizionali per disambiguare e selezionare il nodo corretto. Di seguito sono mostrati alcuni esempi di locatori assoluti:

Listing 1.2 Esempi di Locatori Assoluti

```
1: /html/body/angular-spotify-root/as-layout/as-nav-bar/ul
     /li[1]/a
2: /html/body/angular-spotify-root/as-layout/as-nav-bar/ul
     /li[4]/a
```

Locatori Relativi

A differenza dell'approccio assoluto, un locatore relativo è definito da un'espressione XPath che non ha origine dalla radice del documento. Esso sfrutta combinazioni di riferimenti al tag target, ai suoi antenati (ancestor) e a specifici attributi o contenuti

testuali, con l'obiettivo di identificare l'elemento in modo univoco indipendentemente dalla sua profondità nell'albero DOM. Di seguito sono mostrati alcuni esempi di locatori relativi:

Listing 1.3 Esempi di Locatori Relativi

```
1: //as-playlist-track[1]//as-media-table-row[1]//as-track
     -main-info[1]//div[2]//div[1]
2: //input[@placeholder='Artists, songs, albums, or
     playlists']
```

1.6.2 Algoritmi ROBULA e ROBULA+

Locatori ROBULA

ROBULA (Robust Locator Algorithm) [1] è una tecnica algoritmica per la costruzione di locatori che adotta un approccio top-down iterativo. L'algoritmo parte dall'espressione XPath più generica possibile per poi specializzarla attraverso successivi step di trasformazione (ad esempio, aggiungendo informazioni sul tipo di tag e sui relativi attributi). Se l'espressione risultante identifica univocamente l'elemento target, l'algoritmo termina restituendo il locatore; in caso contrario, le trasformazioni vengono reiterate risalendo la gerarchia verso i nodi padre. Nel caso limite in cui non sia possibile ricavare un'espressione univoca, l'algoritmo ripiega sulla generazione di un locatore assoluto. Di seguito sono mostrati alcuni esempi di locatori generati con ROBULA:

Listing 1.4 Esempi di Locatori Relativi

```
1: //div[3]/div
2: //svg-icon[@ng-reflect-key='times']
```

Locatori ROBULA+

ROBULA+ [2] rappresenta un'evoluzione e un'estensione dell'algoritmo ROBULA. Esso introduce un set più ampio e sofisticato di trasformazioni applicabili al locatore generico iniziale, con il preciso intento di costruire identificatori dotati di una robustezza e di una resilienza superiori rispetto alla versione originaria. Di seguito sono mostrati alcuni esempi di locatori generati con ROBULA+:

Listing 1.5 Esempi di Locatori Relativi

```
1: //*[3]/*[@class='common-grid']
2: //*[@ng-reflect-router-link='/search']
```

1.6.3 Strategie basate su Selenium e Katalon

Locatori Selenium

Oltre a essere un framework di esecuzione, *Selenium* integra strumenti ampiamente utilizzati per registrare casi di test basati su GUI tramite un approccio *Capture and Replay* (C&R) [3]. Il suo motore interno offre diverse euristiche per la localizzazione degli elementi, generando espressioni che combinano tag, attributi, valori specifici e contenuti testuali per garantire l'univocità dell'identificazione. Oltre alle espressioni XPath, Selenium supporta nativamente strategie di localizzazione basate su selettori CSS. Di seguito sono mostrati alcuni esempi di locatori generati nativamente da Selenium:

Listing 1.6 Esempi di Locatori Relativi

```
1: xpath=//a[contains(text(),'Home')]
2: css=.input-container > .ng-untouched
```

Locatori Katalon

Katalon Recorder implementa tecnologie proprietarie per la sintesi dei locatori, il cui obiettivo teorico è affine a quello della famiglia di algoritmi ROBULA: identificare univocamente un elemento target generando un'espressione concisa e dotata di un numero limitato di riferimenti rigidi. Per raggiungere tale scopo, le espressioni generate dall'euristica di Katalon combinano riferimenti agli attributi del nodo target, indici posizionali, porzioni di testo visibile e relazioni spaziali con i tag adiacenti (following, preceding). Di seguito sono mostrati alcuni esempi di locatori generati con Katalon:

Listing 1.7 Esempi di Locatori Relativi

```
1: xpath=(.//*[normalize-space(text()) and normalize-space
     (.)='Home'])[1]/following::a[1]
2: xpath=(.//*[normalize-space(text()) and normalize-space
     (.)='Artists'])[1]/following::div[1]
```

1.6.4 Approccio Hook-Based

Gli Hook [4] si definiscono come attributi personalizzati iniettati esplicitamente all'interno dei tag HTML. Un locatore Hook-Based è costituito da un'espressione XPath che sfrutta tali marcatori per l'identificazione degli elementi. Questa strategia rappresenta un'alternativa robusta ai locatori tradizionali, risultando particolarmente indicata per le applicazioni web sviluppate tramite tecnologie basate su template. Affinché l'approccio sia efficace, è necessario che a ogni nodo di interesse venga associato un attributo hook il cui nome risulti univocamente identificabile nel contesto dell'intera applicazione.

Poiché questi attributi sono inseriti esclusivamente per finalità di automazione del collaudo, essi risultano del tutto disaccoppiati dalla logica di presentazione visiva. Di conseguenza, sono intrinsecamente meno soggetti a modifiche involontarie o strutturali

durante i cicli di sviluppo della User Interface (UI), riducendo drasticamente il problema della fragilità dei test.

Nel contesto del presente lavoro, i locatori Hook-Based sono stati generati e catturati utilizzando Katalon Recorder, integrato con uno script di estensione sviluppato ad hoc. La convenzione di naming adottata per la generazione automatica degli attributi segue la sintassi `x-test-{tipo}-{tag}-{n}`, dove:

- **{tipo}:** Assume il valore "hook" o "tpl", distinguendo se l'elemento appartiene o meno a un componente template generato dinamicamente.
- **{tag}:** Riporta la tipologia del nodo HTML (ad esempio div, button, input), arricchendo l'attributo di significato semantico.
- **{n}:** Rappresenta un contatore numerico progressivo volto a garantire l'univocità assoluta dell'attributo all'interno del DOM.

Di seguito sono mostrati alcuni esempi di locatori basati su Hook:

Listing 1.8 Esempi di Locatori Relativi

```
1: //*[@x-test-tpl-html-1]//*[@x-test-hook-angular-spotify
     -root-29]//*[@x-test-tpl-as-main-view-3]//*[@x-test-
     tpl-div-2]//*[@x-test-tpl-div-1]//*[@x-test-hook-div
     -16]
2: //*[@x-test-tpl-html-1]//*[@x-test-hook-angular-spotify
     -root-29]//*[@x-test-tpl-as-nav-bar-1]//*[@x-test-
     tpl-ul-3]//*[@x-test-hook-li-4][1]//*[@x-test-hook-a
     -5]
```

–2–
Metodologia di Estrazione e Generazione dei Locatori

A valle dell'inquadramento teorico fornito nel capitolo precedente, in cui sono state definite le caratteristiche strutturali delle diverse strategie XPath, il presente capitolo illustra nel dettaglio il processo operativo adottato per la loro generazione pratica.

In ambito di ingegneria del software sperimentale, la riproducibilità dell'esperimento rappresenta un requisito fondamentale. Pertanto, l'obiettivo di questa sezione è documentare in modo rigoroso e trasparente il workflow metodologico attraverso il quale i locatori sono stati ricavati, catturati o sintetizzati a partire dall'applicazione web AngularJS oggetto di studio, ancor prima di sottoporli al processo di mutazione del DOM.

Poiché le sette strategie di localizzazione selezionate (Absolute, Relative, Hook-Based, ROBULA, ROBULA+, Selenium e Katalon) differiscono radicalmente per logica e provenienza, la fase di preparazione del dataset di test ha richiesto l'adozione di approcci eterogenei. Tali approcci possono essere raggruppati in tre macro-categorie operative, che verranno analizzate nelle sezioni successive:

- **Estrazione assistita tramite ispezione del DOM:** Processo semi-manuale utilizzato per ricavare i locatori Absolute e Relative avvalendosi degli strumenti di sviluppo del browser (DevTools) e dell'estensione SelectorsHub.
- **Generazione algoritmica tramite script:** Esecuzione di procedure automatizzate ad hoc per percorrere l'albero HTML e sintetizzare i locatori complessi, conformemente alle regole matematiche ed euristiche degli algoritmi ROBULA e ROBULA+.
- C**attura dinamica tramite approccio Capture&Replay:** Utilizzo dei motori di registrazione integrati in framework come Katalon Recorder e Selenium per intercettare le azioni dell'utente e generare automaticamente i locatori corrispondenti (inclusa l'identificazione degli attributi custom per l'approccio Hook-Based).

Questo lavoro preparatorio ha permesso di costituire un set di dati eterogeneo e rappresentativo, base indispensabile per la successiva fase di iniezione delle mutazioni e per la valutazione quantitativa della fragilità.

2.1 Estrazione dei Locatori Assoluti e Relativi

Per l'estrazione dei locatori appartenenti alle categorie *Absolute* e *Relative*, è stato adottato un approccio semi-automatizzato, avvalendosi in modo combinato degli strumenti di sviluppo nativi del browser (DevTools) e dell'estensione specifica *SelectorsHub*.

2.1.1 Estrazione dei Locatori Assoluti

L'acquisizione degli XPath assoluti è stata effettuata sfruttando le funzionalità di ispezione del codice sorgente offerte dai moderni browser web (come ad esempio i DevTools di Google Chrome). Attraverso l'interfaccia di esplorazione del Document Object Model (tab "Elements"), è possibile espandere e navigare l'albero HTML fino a individuare il nodo target.

Una volta evidenziato l'elemento di interesse, aprendo il menu contestuale a esso associato, si procede selezionando la funzione nativa di copia del percorso XPath completo (generalmente denominata "Copy full XPath"). Questa operazione estrae il percorso esatto dalla radice del documento fino al nodo, come illustrato nella Figura 2.1.

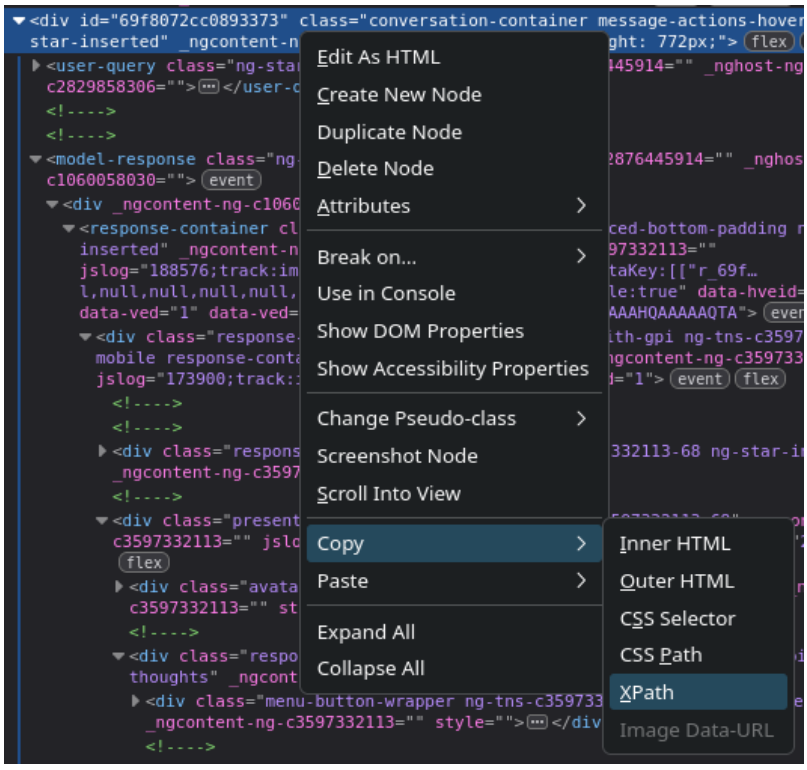


Figure 2.1: Estrazione di un locatore assoluto tramite i DevTools del browser

2.1.2 Estrazione dei Locatori Relativi

Per l'individuazione dei locatori relativi, il processo si è avvalso di *SelectorsHub*, un'estensione per browser progettata appositamente per la generazione, la scrittura e la validazione di selettori complessi.

Una volta attivato il plugin all'interno della pagina web in esame, è stato utilizzato lo strumento di selezione visiva (funzione di ispezione dell'estensione) per puntare direttamente l'elemento grafico target. A seguito della selezione, il motore di *SelectorsHub* analizza in tempo reale il contesto del nodo all'interno del DOM e calcola un'espressione XPath relativa ottimizzata. Tale espressione viene immediatamente mostrata nell'interfaccia dello strumento, rendendola disponibile per la copia e per eventuali raffinamenti manuali, come evidenziato nelle Figure 2.2 e 2.3.

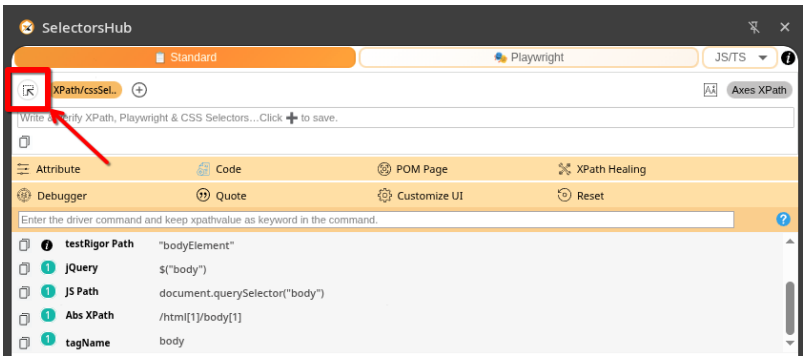


Figure 2.2: Selezione di un elemento tramite SelectorsHub

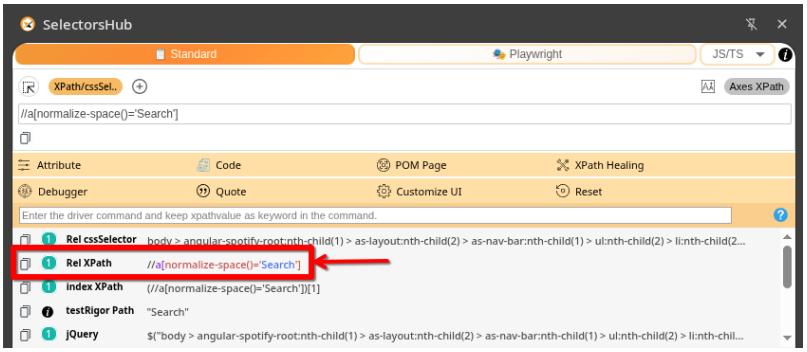


Figure 2.3: Selezione del locatore Relative tramite SelectorsHub

2.2 Generazione Algoritmica dei Locatori ROBULA e ROBULA+

Per la sintesi dei locatori basati sulle euristiche ROBULA e ROBULA+, ci si è avvalsi di un modulo software sviluppato in linguaggio Java. Trattandosi di uno strumento implemen-

tato specificamente per le esigenze di questo processo sperimentale, la fase di inizializzazione dei parametri e l'impostazione del dataset di analisi avvengono in modo programmatico, operando direttamente sui file sorgente che fungono da *entry point* dell'applicativo (rispettivamente `robula/Robula.java` e `robulaplus/RobulaPlus.java`).
Il processo operativo per l'esecuzione degli algoritmi si articola in tre fasi metodologiche:

1. **Acquisizione del contesto DOM:** Affinché gli algoritmi possano navigare e analizzare l'albero HTML per valutare l'univocità delle trasformazioni, è necessario fornire i file sorgente dell'interfaccia web. Tali documenti vengono preventivamente estratti e posizionati all'interno della directory `resources` del modulo, fungendo da base di conoscenza statica per lo strumento.

2. **Definizione programmatica dei target:** All'interno della funzione principale della classe di esecuzione, il set di elementi da localizzare viene istanziato popolando una specifica struttura dati (una lista di oggetti di tipo `element`). Ogni oggetto richiede la definizione di tre parametri fondamentali per la corretta esecuzione dell'algoritmo:

   - Un identificativo logico (Nome) per mappare i risultati a valle dell'esperimento.
   - Il locatore XPath assoluto di partenza, essenziale affinché l'algoritmo identifichi con esattezza il nodo target all'interno del DOM prima di iniziare il processo di riduzione top-down.
   - Il riferimento testuale al file sorgente HTML corrispondente.

3. **Esecuzione e sintesi:** A seguito della configurazione, la compilazione e l'esecuzione del modulo avviano la generazione procedurale. Il sistema elabora la lista degli elementi target, applica ricorsivamente i set di trasformazioni previsti (standard per ROBULA, estesi per ROBULA+) e restituisce in output i locatori robusti elaborati.

Di seguito è riportato un frammento di codice che illustra l'inizializzazione programmatica di un elemento target all'interno dell'ambiente di esecuzione:

Listing 2.1 Esempio di configurazione dei target per l'algoritmo ROBULA/ROBULA+

```
1: elements.add(new element(
2:   "SongsNavbar",
3:   "/html/body/angular-root/as-nav-bar/ul/li[6]/a",
4:   "home.html"
5: ));
```

2.3 Generazione Dinamica tramite Capture & Replay

Per l'estrazione dei locatori basati sulle euristiche di Selenium e Katalon, è stato adottato un approccio dinamico fondato sul paradigma *Capture and Replay* (C&R). Questa metodologia

prevede l'interazione diretta e manuale con l'interfaccia grafica dell'applicazione web, durante la quale specifici strumenti di automazione, posti in ascolto in background, intercettano gli eventi scatenati dall'utente (come i click) e sintetizzano simultaneamente i locatori per i nodi DOM coinvolti.

2.3.1 Estrazione dei Locatori Selenium

La generazione dei locatori nativi di Selenium è stata effettuata avvalendosi del suo ambiente di registrazione ufficiale (Selenium IDE). La procedura operativa consiste nell'avviare la fase di registrazione di un nuovo scenario di test e nell'eseguire l'azione di interesse sull'elemento target all'interno dell'interfaccia *AngularJS*.

A ogni interazione, il motore interno di Selenium intercetta l'evento e compila automaticamente un set di selettori, applicando le proprie regole di generazione. Una volta terminata l'interazione, accedendo all'interfaccia dello strumento, è possibile ispezionare il comando registrato ed estrarre l'espressione XPath o CSS primaria associata all'elemento testato, come illustrato nella Figura 2.4.

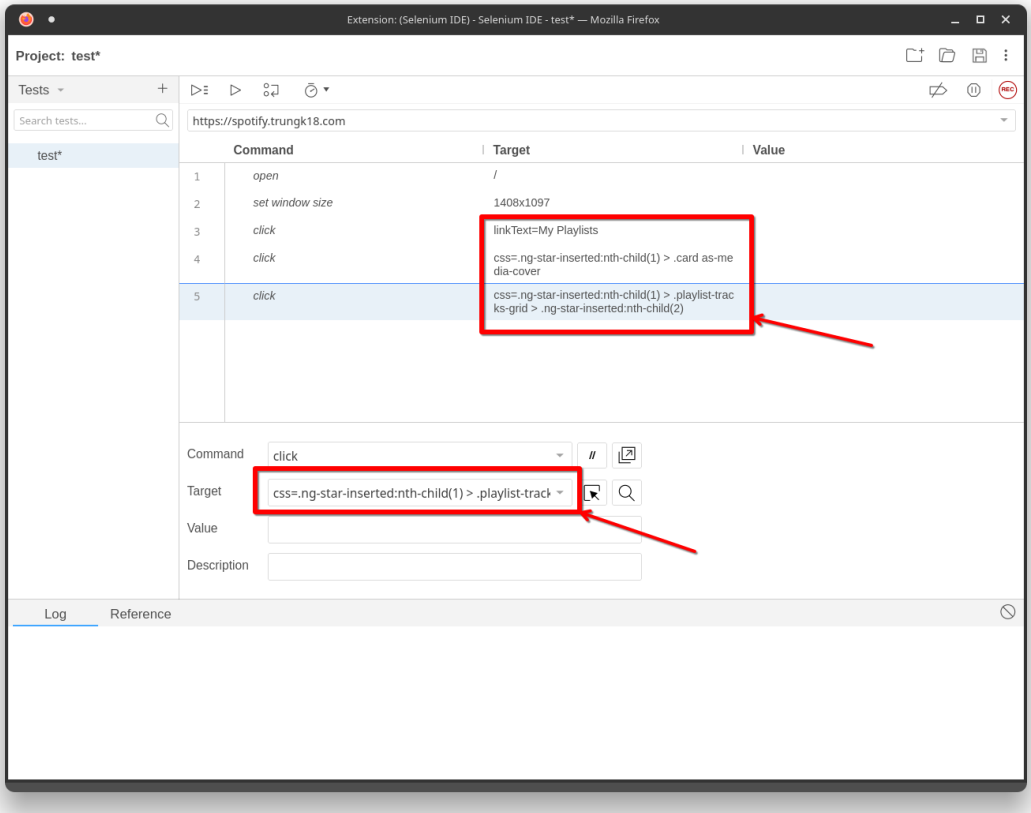


Figure 2.4: Interfaccia di Selenium IDE durante la fase di registrazione e sintesi dei locatori

2.3.2 Estrazione dei Locatori Katalon

Parallelamente, per l'ottenimento dei locatori basati sull'algoritmo di Katalon, è stata impiegata l'estensione browser *Katalon Recorder*. Analogamente al processo precedente, l'ambiente viene attivato in modalità di registrazione prima di interagire visivamente con la pagina.

Il valore aggiunto di questo strumento risiede nel suo motore di sintesi proprietario: a seguito del click su un nodo target, Katalon Recorder elabora dinamicamente molteplici varianti di espressioni XPath (combinando attributi testuali, relazioni spaziali come *following* o *preceding*, e indici posizionali). Queste varianti vengono proposte e ordinate all'interno del pannello dei dettagli in base a una stima interna di priorità e affidabilità. Per le finalità della presente sperimentazione, è stato sistematicamente estratto il locatore XPath principale suggerito dall'euristica dello strumento (Figura 2.5).

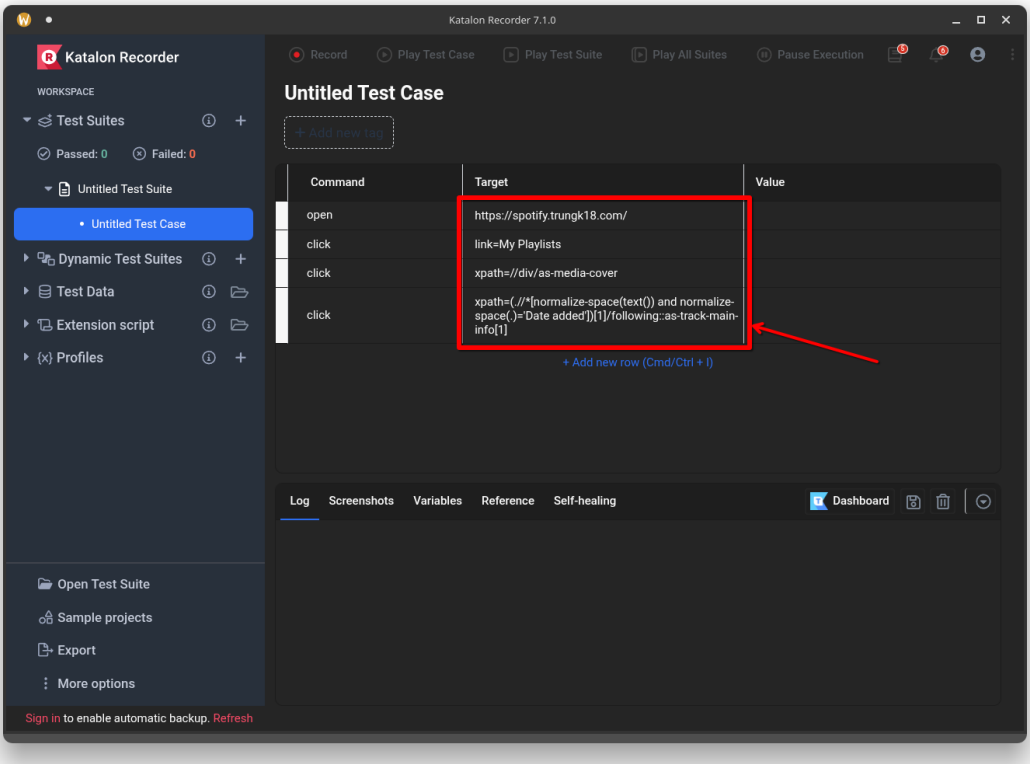


Figure 2.5: Pannello di Katalon Recorder che mostra l'elenco dei locatori generati a seguito di un'interazione utente

2.4 Generazione dei Locatori Hook-Based

Come delineato nel capitolo precedente, l'efficacia della strategia Hook-Based è intrinsecamente legata alla presenza di attributi personalizzati all'interno del codice sorgente.

Pertanto, la generazione di questi locatori ha richiesto un processo articolato in due fasi distinte: un'iniezione preliminare nel DOM e la successiva sintesi tramite script.

2.4.1 Iniezione Preliminare degli Attributi

Al fine di predisporre l'applicazione web all'analisi, è stata eseguita una fase preparatoria di iniezione degli *Hook*. Tale operazione automatizzata è stata demandata allo strumento software sviluppato appositamente per questo studio (di cui si tratterà approfonditamente nel Capitolo 3). Il tool ha il compito di scansionare i file *template* del progetto *AngularJS* e di inserire in modo programmatico gli attributi necessari nei nodi HTML rilevanti. Per i dettagli implementativi e le istruzioni operative relative all'esecuzione di questa fase di iniezione, si rimanda alla documentazione tecnica presente all'interno del *repository* GitHub del progetto [5].

2.4.2 Sintesi dei Locatori tramite Estensione Katalon

A valle dell'iniezione, per la generazione effettiva delle espressioni XPath a partire dal DOM modificato, è stato sviluppato lo script dedicato `attributeHooksLocators.js`, caricato come estensione personalizzata all'interno di Katalon Recorder (Figura 2.6).

L'obiettivo primario di tale estensione è generare locatori robusti sfruttando esclusivamente gli attributi personalizzati, intervenendo a livello architetturale per sovrascrivere l'ordine di preferenza nativo di Katalon e garantendo priorità assoluta alla strategia Hook-Based. Il funzionamento logico dello script si articola in tre fasi principali:

- **Riconoscimento degli Hook:** Nella fase di inizializzazione, lo script definisce due costanti, `hookPrefix` e `templatePrefix`, corrispondenti ai prefissi degli attributi custom (rispettivamente `"x-test-hook-"` e `"x-test-tpl-"`). Attraverso l'ausilio di espressioni regolari (RegEx), l'estensione è in grado di intercettare e riconoscere dinamicamente qualsiasi attributo associato a tale pattern testuale.
- **Algoritmo di costruzione del locatore:** Il *core* dell'estensione risiede nella funzione `myLocatorBuilder`. Quando Katalon avvia la generazione di un locatore per un elemento HTML target, questa funzione implementa una logica di attraversamento *bottom-up* dell'albero DOM:
  - L'algoritmo ha origine dall'elemento selezionato e risale gerarchicamente verso la radice del documento.
  - A ogni livello esplorato, l'esecuzione ricerca la presenza di un attributo di tipo Hook.
  - Qualora venga individuato un Hook, lo script verifica l'eventuale esistenza di nodi fratelli (*sibling*) provvisti del medesimo attributo. In caso affermativo, l'algoritmo inietta un indice posizionale per garantire l'univocità ed evitare ambiguità di selezione.

– Il locatore finale viene sintetizzato concatenando i vari attributi intercettati durante la risalita. In questa fase viene conferita elevata priorità ai nodi di tipo *template*, assicurando che essi vengano rigorosamente inclusi nel percorso gerarchico risultante.

L'output di questo processo consiste nella produzione di locatori ottimizzati, declinati sia in formato XPath sia come selettori CSS.

- **Registrazione e Assegnazione di Priorità:** Come ultima operazione, lo script registra le due nuove strategie di localizzazione (XPath e CSS) all'interno del motore di Katalon, imponendole come scelte preferenziali. In virtù di questo override, a seguito di un'interazione dell'utente, la prima espressione suggerita e utilizzata da Katalon Recorder coinciderà sempre con il locatore XPath basato sugli Hook generato dallo script custom.

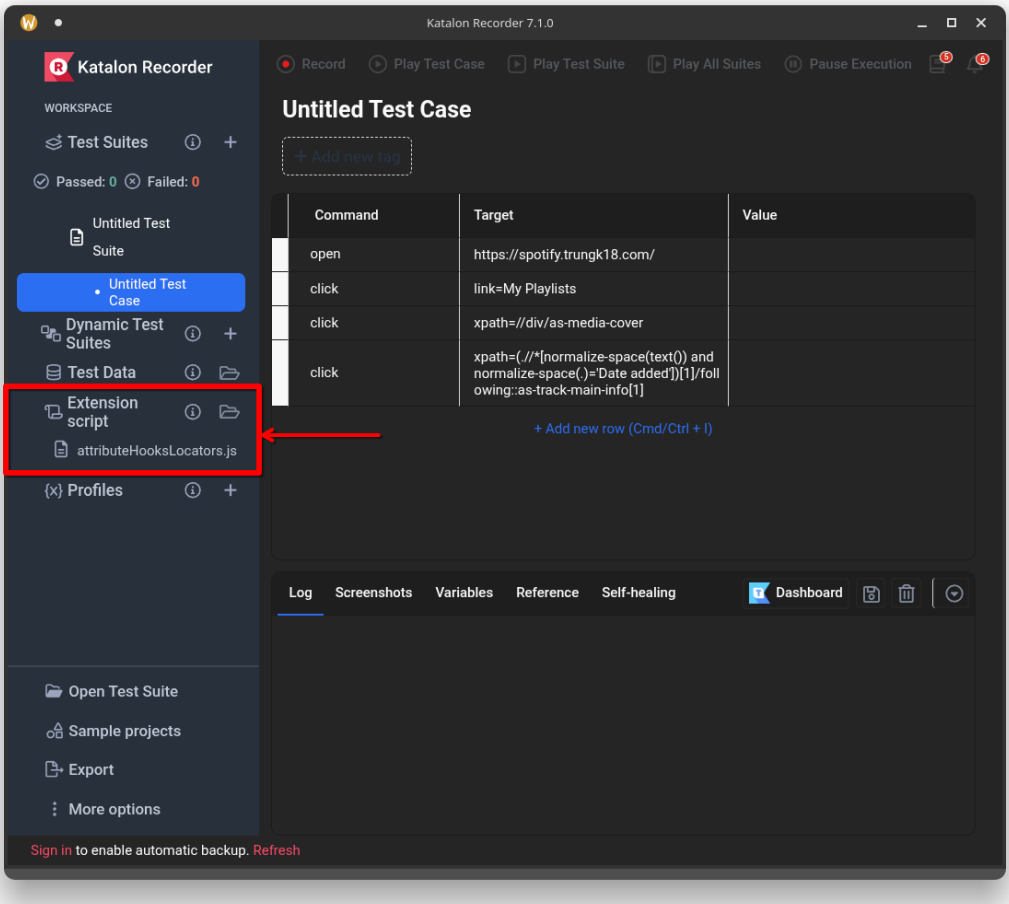


Figure 2.6: Sezione del pannello di Katalon dove inserire lo script custom

–3–
Architettura e Funzionamento dello Strumento di Automazione

Al fine di condurre l'analisi sperimentale delineata nei capitoli precedenti, è stato progettato e sviluppato uno strumento software dedicato [5]. L'applicativo ha il compito di orchestrare l'intero processo di Mutation Analysis: dall'alterazione strutturale del codice sorgente dell'applicazione AngularJS, fino all'esecuzione automatizzata e al monitoraggio delle suite di test E2E.

Per i dettagli implementativi a basso livello, i file di configurazione specifici per l'avvio della suite e per consultare il codice sorgente integrale dei moduli descritti in questo capitolo, si rimanda alla repository GitHub ufficiale del progetto, la cui struttura logica è illustrata nell'**Appendice A**.

3.1 Architettura Generale dello Strumento

Lo strumento è stato ingegnerizzato come un progetto multimodulo basato su *Maven*, definito dal file descrittore principale `pom.xml`. Questa scelta architetturale garantisce un'elevata modularità e scalabilità, separando nettamente le logiche di generazione da quelle di collaudo. L'intera suite è governata da un file di configurazione centralizzato (`generator-config.json`), il quale permette all'utente di definire i parametri di esecuzione, i percorsi dei file e i target di riferimento senza dover intervenire sul codice sorgente.

L'architettura del sistema si articola nei seguenti moduli principali:

- **`custom-locators`**: Modulo deputato alla logica di generazione dei locatori algoritmici. Include l'implementazione Java per le strategie ROBULA e ROBULA+, nonché lo script JavaScript da integrare in Katalon per la sintesi dei locatori Hook-Based.
- **`hook-injector`**: Modulo di supporto responsabile dell'iniezione programmatica degli attributi custom (Hook) all'interno dell'applicazione target, fase propedeutica alla generazione dei relativi locatori.

- `mutation-generator`: Il core logico per l'alterazione del Document Object Model (DOM). Si suddivide in sottomoduli, tra cui `common` per le utility condivise e `static-generator` per l'analisi statica. È stata inoltre predisposta l'infrastruttura per un futuro modulo `llm-generator` (attualmente non implementato), concepito per la generazione di mutazioni guidata dall'Intelligenza Artificiale.
- `mutation-tester`: Modulo orchestratore incaricato di eseguire l'applicazione mutata, lanciare i casi di test precompilati e raccoglierne gli esiti.

3.2 Modulo di Analisi Statica e Manipolazione del DOM

Il primo pilastro architetturale dello strumento risiede nella sua capacità di ispezionare e comprendere il codice sorgente dell'applicazione target prima ancora che questa venga eseguita nel browser. Tale operazione, nota come *analisi statica*, è stata demandata all'utilizzo integrato della libreria Java **JSoup**.

A differenza degli approcci dinamici che interrogano il DOM a runtime tramite un *WebDriver*, l'analisi statica permette di elaborare i file *template* di *AngularJS* in modo isolato, estremamente rapido e privo degli overhead legati al rendering grafico. Il processo si articola nelle seguenti fasi operative:

- **Parsing e Inizializzazione:** Il modulo legge dal file di configurazione il percorso esatto del documento da mutare (`file_path`) e utilizza JSoup per processarne il contenuto testuale. La libreria analizza la sintassi del markup e costruisce in memoria una rappresentazione gerarchica ad albero.
- **Mappatura del Contesto e del Target:** Per individuare il nodo da alterare, lo strumento legge l'oggetto `target_matcher` dalla configurazione, il quale definisce le regole di matching (es. per classe, per attributo o per valore testuale). Il sistema non si limita a identificare l'elemento target, ma ne mappa l'intero "ecosistema" gerarchico: individua il nodo padre, un antenato (*ancestor*), l'eventuale *template* di appartenenza e un nodo fratello (*sibling*) casuale. Quest'ultimo viene selezionato in modo pseudo-casuale ma deterministico, sfruttando un *seed* globale impostato nella configurazione.
- **Salvataggio degli Indici e Manipolazione:** Una volta identificati, gli indici strutturali di questi nodi vengono salvati all'interno di un'apposita struttura dati. La manipolazione successiva (alterazione o spostamento) avviene sfruttando tali indici operando direttamente sugli oggetti strutturali di JSoup, garantendo che qualsiasi modifica mantenga intatta la validità sintattica del documento HTML risultante prima del salvataggio.

Una volta che l'albero DOM è stato caricato e i target identificati, la struttura in memoria viene passata al motore centrale per l'applicazione effettiva delle mutazioni.

3.3 Il Motore di Generazione dei Mutanti

Il cuore logico per l'iniezione dei difetti (*fault injection*) è implementato all'interno del sottomodulo `static-generator`. Questo componente non si limita ad applicare le modifiche al DOM, ma orchestra l'intero ciclo di preparazione e persistenza dei dati, ottimizzando le performance di esecuzione.

Per gestire in modo efficiente la generazione dei mutanti, il motore opera secondo un flusso di lavoro strutturato:

- **Configurazione Iniziale:** L'esecuzione si avvia con il parsing del file di configurazione `generator-config.json`. Questo documento centralizzato definisce il parametro di generazione (`seed`), la root del repository (`repositoryRootPath`), il comando di avvio dell'applicazione (`npmRunCommand`) e l'array dettagliato di tutte le mutazioni che il sistema dovrà applicare in sequenza.

- **Indicizzazione dei Componenti (*ComponentIndexer*):** Le applicazioni *AngularJS* sono tipicamente costituite da innumerevoli componenti frammentati. Per evitare costose e ripetitive operazioni di Input/Output sul disco, lo strumento istanzia un *singleton* denominato `ComponentIndexer`. Questa classe scansiona preventivamente l'intero albero delle directory alla ricerca di file *TypeScript* (`.component.ts`). Tramite l'uso di espressioni regolari (RegEx), estrae il `selector` del componente e lo associa al rispettivo file *template* HTML. Tutti i percorsi e i contenuti vengono caricati in memoria in una mappa strutturata. Questa indicizzazione globale è fondamentale per supportare mutazioni avanzate, come ad esempio lo spostamento di un tag DOM da un componente all'altro, avendo già a disposizione i riferimenti incrociati.

- **Persistenza su Database Relazionale (SQLite):** Poiché un'applicazione reale prevede che una singola modifica possa ripercuotersi su più porzioni di codice o file (e poiché è necessario mantenere traccia esatta di quale codice corrisponda a quale mutante), il sistema non sovrascrive i file originali sul file system. I risultati vengono invece salvati all'interno di un database relazionale locale basato su **SQLite**.

La scelta di adottare un database integrato e *serverless* come SQLite risponde alla necessità di avere un archivio dati leggero, portabile e strutturato. Il modello dati è organizzato in due tabelle principali:

1. **`mutations`:** Una tabella anagrafica che memorizza le proprietà, i metadati e la tipologia logica della mutazione generata (es. "Rimozione ID sul tag Button").

2. **`mutated_files`:** Una tabella di mappatura che associa ogni specifica mutazione ai file sorgente che hanno subito alterazioni. Per ogni record, viene salvato l'esatto percorso del file e la stringa HTML corrispondente al codice sorgente mutato. Questa granularità garantisce che il modulo di testing possa, nella fase successiva, estrarre e ricostruire l'intero contesto dell'applicazione *AngularJS* esattamente come alterato dal generatore.

Al fine di garantire la tracciabilità del processo e consentire una valutazione analitica delle performance del generatore, il modulo produce in output un report diagnostico strutturato in formato CSV, denominato `mutations.csv`.

Questo documento traccia l'esito di ogni tentativo di iniezione, riportando per ciascuna istanza lo stato dell'operazione (classificato come OK per le mutazioni generate con successo o KO per quelle fallite) ed eventuali messaggi di errore o eccezioni sollevate durante l'elaborazione. È fondamentale sottolineare che l'architettura del sistema adotta un approccio selettivo basato sui vincoli strutturali del DOM: il motore tenta la generazione esclusivamente sugli elementi effettivamente identificati e mappati durante la fase di analisi preliminare. A titolo esemplificativo, qualora un nodo target non presenti elementi fratelli (sibling) nell'albero HTML, il corrispondente riferimento logico σ (come specificato nel Listing 1.1) non verrà incluso nella lista dei target ammissibili. Di conseguenza, le tipologie di mutazione dipendenti da tale elemento non verranno catalogate come "errori di generazione", bensì non verranno neppure tentate, ottimizzando i tempi di elaborazione ed evitando la produzione di falsi negativi nel report.

Di seguito è riportato un estratto esemplificativo del file `mutations.csv` generato al termine del processo, strutturato in forma tabellare per semplificare la lettura:

Nome	**Elemento**	**Mutazione**	**Stato**	**Errore**
NavbarLinkA	alpha	a	OK	
NavbarLinkA	beta	c	KO	Target element has no id attribute
NavbarLinkLi	alpha	e	KO	Target element text content is empty
NavbarLinkUl	beta	f	KO	Target element with body tag cannot be mutated

Table 3.1: Esempio di output del file mutations.csv con esiti di generazione

3.4 Modulo di Automazione ed Esecuzione dei Test

L'ultima fase del processo di *Mutation Analysis* consiste nell'esecuzione iterativa e automatizzata della suite di collaudo. Questa responsabilità è data al modulo `mutation-tester`, progettato per operare in modo completamente agnostico rispetto ai casi di test specifici scritti dall'utente.

Dal punto di vista architetturale, il modulo si articola in tre componenti principali, ciascuno con responsabilità ben definite all'interno della pipeline di esecuzione:

- **Entry Point (`MutationTester`):** appresenta la classe principale che ospita il metodo `main`. Il suo compito primario è validare la coerenza delle configurazioni di avvio e acquisire in input il percorso della directory (fornito dall'utente) in cui risiedono i casi di test già compilati (file `.class`).

- **Integrazione con l'Ambiente di Sviluppo `NpmConsoleWrapper`:** Per testare un'applicazione *AngularJS*, è necessario che questa sia in esecuzione su un server locale. Questo componente funge da wrapper per la riga di comando, lanciando il processo Node.js tramite il comando npm specificato nel file JSON di configurazione. Oltre ad avviare il server, il wrapper monitora costantemente l'output della console per assicurarsi che l'applicazione si avvii correttamente e intercetta gli eventi di avvenuta ricompilazione (*hot-reload*) che si innescano ogni volta che il sistema applica o rimuove una mutazione.
- **Motore di Esecuzione `TestRunnerEngine`:** Costituisce il cuore logico dell'intero processo di valutazione. Questo componente agisce da orchestratore: interroga il database SQLite per recuperare la lista delle mutazioni previste, le inietta nei file dell'applicazione in esecuzione e, a ricompilazione avvenuta, scatena le classi di test. Al termine di ogni esecuzione, valuta l'esito del test (se il locatore è sopravvissuto o se è incorso in un test breakage), esegue il ripristino del file alterato e produce due report in formato CSV: uno contenente il dettaglio granulare di ogni singolo test e uno aggregato con le statistiche di robustezza per ciascun locatore analizzato.

3.5 Flusso Operativo e Raccolta dei Risultati

L'architettura modulare descritta nelle sezioni precedenti opera in modo sinergico per automatizzare l'intero ciclo di vita della *Mutation Analysis*. Il flusso operativo *end-to-end* dello strumento può essere riassunto in quattro fasi consequenziali:

1. **Inizializzazione e Analisi Statica:** Il sistema indicizza i componenti e mappa i nodi target nel DOM.
2. **Generazione e Persistenza:** Le mutazioni vengono calcolate in memoria tramite JSoup e salvate nel database SQLite, evitando di inquinare il file system originale.
3. **Esecuzione Dinamica:** Il *wrapper* avvia l'applicazione mutata, mentre il motore carica a runtime le classi di test pertinenti, eseguendole contro il nuovo DOM.
4. **Valutazione e Ripristino:** L'esito del test viene registrato, il codice sorgente viene ripristinato allo stato originale e il ciclo riparte per la mutazione successiva.

3.5.1 Struttura dei Dati di Output

A conclusione dell'intero ciclo iterativo, il `TestRunnerEngine` consolida i risultati in memoria ed effettua il *dump* dei dati su file system. Per supportare la successiva fase di analisi empirica, il modulo esporta due distinti report in formato CSV, i quali rappresentano l'output tangibile dell'intero strumento.

Report Dettagliato delle Esecuzioni

Questo file traccia l'esito granulare di ogni singolo test eseguito. Per ciascuna combinazione di locatore e mutazione, il sistema registra se l'elemento è stato identificato con successo (*Passed*), se la mutazione ha causato la rottura del test (*Failed*) o se ci sono stati errori o eccezioni interne al motore di esecuzione (*Broken*). Questo livello di dettaglio è fondamentale per identificare quali specifiche alterazioni del DOM mettono in crisi determinate strategie. Di seguito è riportato un estratto strutturale del report dettagliato, strutturato in forma tabellare per semplificare la lettura:

Name	**Id**	**Locator**	**Tag**	**Status**	**Error**
NavbarLinkA	mutation_j_tag_type_mod_mut_beta	Absolute	beta	FAILED	no such element: Unable to locate element: "/html/body/angular-spotify-root/as-layout/as-nav-bar/ul/li[1]/a"
NavbarLinkUl	mutation_k_tag_ins_mut_delta	Selenium	delta	PASSED	null
NavbarLinkUl	mutation_f_tag_mov_container_mut_alpha	RobulaPlus	alpha	FAILED	java.lang.AssertionError

Table 3.2: Esempio di output del file stats.csv con statistiche globali

Report Aggregato delle Statistiche

Parallelamente al log dettagliato, il sistema elabora un file di sintesi che aggrega i risultati calcolando le metriche di robustezza e diagnosticando le specifiche cause di fallimento a livello globale. Per ogni strategia di localizzazione testata, il report quantifica il numero totale di mutazioni affrontate e suddivide gli esiti in specifiche categorie analitiche:

- **Success:** Numero di mutazioni a cui il locatore è sopravvissuto, riuscendo a identificare correttamente l'elemento target.
- **Fragility:** Numero di esecuzioni fallite esclusivamente a causa della fragilità del locatore. In questi casi, la mutazione rappresenta una modifica marginale (strutturale o estetica) a cui altre strategie di localizzazione sono riuscite a resistere, configurando il fallimento come un "falso positivo".
- **Obsolescence:** Numero di esecuzioni in cui l'intero batch di test ha registrato un fallimento totale, impedendo a qualsiasi locatore di identificare l'elemento target. Questo scenario suggerisce che l'alterazione strutturale potrebbe aver compromesso la funzionalità stessa del componente (ad esempio, a causa della rimozione integrale

del nodo dal DOM). Poiché lo strumento automatizzato non può dedurre l'intento logico della modifica, queste casistiche vengono classificate esclusivamente come "possibili obsolescenze"; è pertanto richiesta una successiva validazione manuale per confermare se il test sia effettivamente divenuto obsoleto e necessiti di essere rimosso o riscritto.

- **Not Compiled:** Numero di mutazioni scartate poiché l'alterazione sintattica del *template* ha causato un errore fatale di compilazione nel motore di *AngularJS*, impedendo il riavvio dell'applicazione e la conseguente esecuzione del test.

Questa suddivisione diagnostica costituisce la base dati primaria per la valutazione comparativa delle sette strategie analizzate. Di seguito è riportato un estratto reale della struttura di questo file generato dallo strumento, strutturato in forma tabellare per semplificare la lettura:

Locator	**Total**	**Success**	**Fragility**	**Obsolescence**	**Not Compiled**
Relative	50	26	2	6	14
Absolute	50	17	13	6	14
Robula	50	26	4	6	14
Katalon	50	25	5	6	14
Selenium	50	25	5	6	14
RobulaPlus	50	25	5	6	14
Hook	50	20	10	6	14

Table 3.3: Esempio di output del file stats.csv con statistiche globali

–4–
Disegno Sperimentale e Definizione delle Metriche

Dopo aver illustrato l'architettura del sistema di *Mutation Testing*, il presente capitolo formalizza il protocollo sperimentale adottato. Vengono definiti gli obiettivi primari dello studio, le domande di ricerca (*Research Questions*), gli oggetti sottoposti ad analisi e le metriche quantitative utilizzate per valutare sia la robustezza delle strategie di localizzazione sia l'affidabilità dello strumento di generazione sviluppato.

4.1 Obiettivi dell'Esperimento (Goal)

L'esperimento si pone un duplice obiettivo, mirato a validare sia l'oggetto della ricerca (i locatori) sia lo strumento utilizzato per condurla:

- **Validazione dello Strumento di Generazione:** Verificare l'efficacia e la correttezza sintattica del motore di mutazione implementato, quantificando la sua capacità di produrre alterazioni del codice sorgente che siano valide, compilabili dal framework Angular e in grado di rappresentare scenari di test significativi (evitando tassi eccessivi di errori di compilazione o mutazioni "vuote").
- **Valutazione della Robustezza:** Analizzare comparativamente la resilienza di sette differenti strategie di localizzazione XPath (tra cui approcci algoritmici come ROBULA+ e approcci basati su Hook dedicati) quando sottoposte a mutazioni del DOM in un'applicazione reale complessa.

4.2 Domande di Ricerca (Research Questions)

Sulla base degli obiettivi dichiarati e dei dati raccolti tramite l'esecuzione automatizzata, sono state formulate le seguenti domande di ricerca:

- **RQ1 (Applicabilità delle Mutazioni):** Qual è il grado di applicabilità pratica degli operatori di mutazione standard all'interno di un DOM reale e in che percentuale il framework AngularJS tollera tali alterazioni sintattiche senza incorrere in fallimenti di compilazione (Build Failure)?
- **RQ2 (Robustezza Comparativa):** Quale strategia di localizzazione garantisce il minor tasso di fragilità (*Fragility Rate*) e il maggior tasso di sopravvivenza (*Survival Rate*) a fronte di modifiche strutturali ed estetiche dell'interfaccia utente?
- **RQ3 (Analisi dell'Impatto):** Esiste una correlazione tra specifiche tipologie di mutazione e il fallimento di determinate famiglie di locatori?

4.3 Oggetto dello Studio (Objects)

4.3.1 L'Applicazione Target: Angular-Spotify

La scelta del caso di studio è ricaduta su *Angular-Spotify*, un clone open-source del celebre client di streaming musicale. Tale applicazione è stata selezionata in quanto rappresenta un artefatto software "production-ready", caratterizzato da una complessità architetturale e funzionale nettamente superiore ai comuni esempi didattici ("*toy applications*").

Il progetto, che vanta oltre 2.800 stelle sulla piattaforma GitHub, è sviluppato con il framework Angular e implementa pattern avanzati tipici delle moderne applicazioni enterprise, tra cui:

- Gestione complessa dello stato e routing applicativo.
- Interazioni asincrone con le API ufficiali di Spotify.
- Struttura del DOM profonda e dinamica, ricca di componenti annidati.

Queste caratteristiche rendono *Angular-Spotify* il banco di prova ideale per stressare le capacità di localizzazione degli algoritmi e valutare la resilienza dello strumento di mutazione creato.

4.3.2 La Suite di Test E2E

Per validare le funzionalità critiche dell'applicazione, è stata progettata e implementata una suite di test End-to-End (E2E) completa. I casi di test (*Test Cases* - TC) sono stati suddivisi in tre macro-aree funzionali, progettate per coprire le interazioni più frequenti dell'utente: Navigazione Strutturale, Gestione Libreria e Motore di Ricerca.

Navigazione e Struttura

Questa area verifica il corretto funzionamento della barra laterale di navigazione, elemento persistente e critico per l'usabilità dell'applicazione.

Table 4.1: Test Case strutturati per l'area di test "Navigazione e Struttura"

ID Test	Precondizione	Input	Oracolo	Postcondizione
TC-SIDE-01	None	1. Click su "Home".	L'URI nel percorso è uguale a "/".	None
	None	2. Click su "Search".	L'URI nel percorso è uguale a "/search".	None
	None	3. Click su "Browse".	L'URI nel percorso è uguale a "/browse".	None
	None	4. Click su "My Playlists".	L'URI nel percorso è uguale a "/collection/-playlists".	None
	None	5. Click su "My Albums".	L'URI nel percorso è uguale a "/albums".	None
	None	6. Click su "Liked Songs."	L'URI nel percorso è uguale a "/collection/-tracks".	None
TC-SIDE-02	None	1. Click su "Home".	L'elemento cliccato ha classe css `active`.	None
	None	2. Click su "Search".	L'elemento cliccato ha classe css `active`.	None
	None	3. Click su "Browse".	L'elemento cliccato ha classe css `active`.	None

Continua nella pagina successiva...

Table 4.1: Test Case strutturati per l'area di test "Navigazione e Struttura"

ID Test	Precondizione	Input	Oracolo	Postcondizione
	None	4. Click su "My Playlists".	L'elemento cliccato ha classe css `active`.	None
	None	5. Click su "My Albums".	L'elemento cliccato ha classe css `active`.	None
	None	6. Click su "Liked Songs".	L'elemento cliccato ha classe css `active`.	None
TC-SIDE-03	None	1. Click sulla prima libreria.	Il nome della libreria aperta corrisponde con il nome della libreria nella sidebar.	None
	None	2. Click sulla seconda libreria.	Il nome della libreria aperta corrisponde con il nome della libreria nella sidebar.	None
	None	3. Click sulla terza libreria.	Il nome della libreria aperta corrisponde con il nome della libreria nella sidebar.	None
	None	4. Click sulla quarta libreria.	Il nome della libreria aperta corrisponde con il nome della libreria nella sidebar.	None

Continua nella pagina successiva...

Table 4.1: Test Case strutturati per l'area di test "Navigazione e Struttura"

ID Test	Precondizione	Input	Oracolo	Postcondizione
	None	5. Click sulla quinta libreria.	Il nome della libreria aperta corrisponde con il nome della libreria nella sidebar.	None
	None	6. Click sulla sesta libreria.	Il nome della libreria aperta corrisponde con il nome della libreria nella sidebar.	None
TC-SIDE-04	None	1. Click sulla prima libreria.	L'elemento cliccato ha classe css `active`.	None
	None	2. Click sulla seconda libreria.	L'elemento cliccato ha classe css `active`.	None
	None	3. Click sulla terza libreria.	L'elemento cliccato ha classe css `active`.	None
	None	4. Click sulla quarta libreria.	L'elemento cliccato ha classe css `active`.	None
	None	5. Click sulla quinta libreria.	L'elemento cliccato ha classe css `active`.	None
	None	6. Click sulla sesta libreria.	L'elemento cliccato ha classe css `active`.	None

Libreria e Playlist

Questa sezione collauda la logica di visualizzazione dei dati, la cardinalità delle liste e le funzionalità di riproduzione musicale.

Table 4.2: Test Case strutturati per l'area di test "Libreria e Playlist"

ID Test	Precondizione	Input	Oracolo	Postcondizione
TC-LIB-01	Pagina "My Playlists" aperta.	Click sulla prima playlist.	Il testo dell'intestazione è uguale al nome della playlist aperta.	None
TC-LIB-02	Una playlist contenente almeno un brano è aperta a schermo.	Conteggio del numero di righe presenti nell'elenco dei brani.	Il numero di elementi deve corrispondere al numero di tracce presente nell'apposito campo testuale.	None
TC-LIB-03	Una playlist contenente almeno un brano è aperta a schermo.	Doppio click sul primo brano dell'elenco	**1.** Il testo del brano ha classe css text-primary. **2.** Il nome della canzone in esecuzione è uguale al nome del primo brano.	None
TC-LIB-04	**1.** Una playlist contenente almeno un brano è aperta a schermo. **2.** Almeno un brano della playlist è in esecuzione.	Click sul pulsante "Next" nel player globale.	Il nome della canzone in esecuzione è uguale al nome del secondo brano dell'elenco.	None
TC-LIB-05	Pagina "My Playlists" aperta.	Click sulla prima playlist.	Nella sidebar, la voce corrispondente ha classe css `active`.	None

Motore di Ricerca

L'ultima macro-area verifica la capacità del sistema di filtrare i risultati e gestire input utente dinamici.

Table 4.3: Test Case strutturati per l'area di test "Motore di Ricerca"

ID Test	Precondizione	Input	Oracolo	Postcondizione
TC-SRCH-01	Pagina "Search" aperta.	Inserire il nome di un brano esistente nel campo di ricerca.	La lista dei risultati dei brani contiene almeno un risultato che corrisponde alla stringa inserita.	None
TC-SRCH-02	Pagina "Search" aperta.	Inserire il nome di un brano non esistente nel campo di ricerca.	La lista dei risultati dei brani non contiene nessun risultato che corrisponde alla stringa inserita.	None
TC-SRCH-03	Pagina "Search" aperta.	Inserire il nome di un artista esistente nel campo di ricerca.	La lista dei risultati degli artisti contiene almeno un risultato che corrisponde alla stringa inserita.	None
TC-SRCH-04	Pagina "Search" aperta.	Inserire il nome di un artista non esistente nel campo di ricerca.	La lista dei risultati degli artisti non contiene nessun risultato che corrisponde alla stringa inserita.	None
TC-SRCH-05	**1.** Pagina "Search" aperta. **2.** Testo presente nel campo di ricerca.	Click sull'icona "X" nel campo di ricerca.	Il testo nel campo di ricerca è vuoto.	None

4.4 Variabili dell'Esperimento

4.4.1 Variabili Indipendenti

Le variabili manipolate per osservare gli effetti sul sistema sono:

- **Strategia di Localizzazione:** I 7 approcci analizzati (Absolute, Relative, Selenium, Katalon, Robula, Robula+, Hook).
- **Operatore di Mutazione:** Le diverse tipologie di alterazione iniettate dal tool (es. rimozione ID, cambio del tipo di tag, spostamento in altro template) come specificate nella Tabella 1.1.

4.4.2 Variabili Dipendenti (Metriche)

Le variabili dipendenti rappresentano le misurazioni effettuate per rispondere alle RQ. Esse vengono calcolate a partire dai dati grezzi estratti dai report CSV (Total, Success, Fragility, Obsolescence, Not Compiled).

Metriche per la RQ1

Per valutare l'efficacia del processo di iniezione dei difetti, l'analisi non si limita a verificare la corretta esecuzione del software, ma indaga quanto gli operatori di mutazione teorici siano effettivamente compatibili con la struttura di una moderna Single Page Application. A tale scopo sono state definite due metriche:

- **Tasso di Applicabilità (Mutation Applicability Rate - MAR):** Questa metrica quantifica la percentuale di mutazioni teoriche che hanno trovato le condizioni strutturali adatte per essere istanziate sul DOM. Calcolata a partire dai log di generazione (`mutations.csv`), misura il rapporto tra le mutazioni generate con successo e il totale delle mutazioni tentate.

$$MAR = \frac{N_{generatedOK}}{N_{attempted}} \times 100 \tag{4.1}$$

  Un valore di MAR inferiore al 100% non indica un malfunzionamento dello strumento, bensì riflette i vincoli intrinseci del codice sorgente: lo strumento, operando correttamente, scarta i mutanti inapplicabili (ad esempio, interrompendo la mutazione "Attribute Identifier Modification" qualora l'elemento target sia sprovvisto di tale attributo), garantendo la coerenza logica dell'esperimento.

- **Tasso di Tolleranza del Framework (Framework Tolerance Rate - FTR):** Questa metrica valuta l'impatto semantico delle mutazioni sull'ecosistema *AngularJS*. Calcolata esclusivamente sull'insieme dei mutanti effettivamente applicati (MAR), quantifica la percentuale di alterazioni che il compilatore AOT (*Ahead-of-Time*)

del framework ha accettato come codice sintatticamente e logicamente valido, permettendo l'avvio dell'applicazione.

$$FTR = \left(1 - \frac{Not\_Compiled}{Total\_Applied}\right) \times 100 \tag{4.2}$$

L'FTR evidenzia il grado di rigidità strutturale del framework: fallimenti in questa fase dimostrano l'intrinseca invalidità logica di alcune manipolazioni (come lo spostamento di direttive `*ngIf` in contesti privi del corretto *scope* delle variabili), che causano un'inevitabile *Build Failure*.

Metriche per la RQ2

Per valutare oggettivamente le performance di ciascuna strategia, sono state definite due metriche principali, calcolate escludendo dal totale delle esecuzioni i casi in cui il test non è stato compilato (**Not Compiled**) o è divenuto semanticamente irraggiungibile (**Obsolescence**), poiché in quei casi il fallimento non è imputabile al locatore.

- **Tasso di Sopravvivenza (Survival Rate - SR):** Misura la capacità del locatore di adattarsi al cambiamento.

$$SR = \frac{Success}{Total - (NotCompiled + Obsolescence)} \times 100 \tag{4.3}$$

- **Tasso di Fragilità (Fragility Rate - FR):** Misura la percentuale di fallimenti dovuti specificamente alla rottura del locatore.

$$FR = \frac{Fragility}{Total - (NotCompiled + Obsolescence)} \times 100 \tag{4.4}$$

È fondamentale sottolineare che le due metriche selezionate sono matematicamente complementari. Esse valutano il medesimo fenomeno (l'esito di un'esecuzione valida) da due prospettive opposte: l'adattabilità al cambiamento e la suscettibilità all'errore. Poiché la loro somma copre l'intero dominio dei test validi valutabili, esse risultano inversamente proporzionali ($SR + FR = 100\%$); pertanto, all'ottimizzazione di una corrisponde necessariamente la minimizzazione dell'altra.

Metriche per la RQ3

Per determinare se esista una correlazione tra specifici costrutti HTML alterati e il fallimento di particolari famiglie di locatori, l'analisi non si basa sui dati aggregati globali, ma incrocia i dati del *Report Dettagliato delle Esecuzioni*.

- **Tasso di Fragilità Specifico (Specific Fragility Rate - SFR):** Questa metrica scompone il calcolo della fragilità raggruppandolo per la variabile indipendente "Tipologia di Mutazione" (es. m =mutation_j_tag_type_mod_mut_beta). Permette di isolare il "punto debole" di ogni algoritmo. Viene calcolato come:

$$SFR = \frac{Fragility_m}{Total_m - (NotCompiled_m + Obsolescence_m)} \times 100 \tag{4.5}$$

È importante sottolineare che lo *Specific Fragility Rate* non è stato calcolato esclusivamente per tipologia di mutazione. Mantenendo invariata la formula base, il denominatore dei test validi e il numeratore dei falsi positivi sono stati riaggregati per calcolare un secondo set di SFR, basato sulla **relazione gerarchica** tra l'elemento mutato e l'elemento target (target, padre, fratello, *ancestor*, *template*). Questa duplice aggregazione dei dati garantisce una mappatura completa delle vulnerabilità di ciascun locatore, sia a livello operativo che topologico.

–5–
Analisi dei Risultati e Discussione

5.1 Introduzione e Panoramica dell'esecuzione

Questo capitolo espone e analizza i risultati quantitativi raccolti a valle dell'esecuzione automatizzata della suite di *Mutation Testing* sull'applicazione *Angular-Spotify*. L'analisi è strutturata in modo sequenziale per rispondere alle tre Domande di Ricerca (RQ) formulate nel capitolo precedente, valutando dapprima l'efficacia dello strumento ingegnerizzato e, successivamente, la resilienza comparativa delle sette strategie di localizzazione.

Prima di procedere con la disaggregazione dei dati, è opportuno inquadrare la dimensione quantitativa dell'esperimento. La campagna di collaudo ha coinvolto l'esecuzione sistematica di 14 scenari di test *End-to-End* (suddivisi nelle aree *Sidebar*, *Libreria* e *Ricerca*), ciascuno replicato per le 7 strategie di localizzazione oggetto di studio.

Il motore di analisi statica ha scansionato i *template* dei componenti interessati, individuando i nodi target e calcolando un totale di **932** mutazioni teoriche applicabili. L'intero ciclo iterativo — comprensivo di iniezione del difetto nel codice sorgente, attesa della ricompilazione *hot-reload* del framework *Angular*, esecuzione della batteria di test tramite Selenium WebDriver e ripristino finale del *clean state* — è stato gestito in completa autonomia dal modulo `mutation-tester`.

L'elaborazione complessiva ha richiesto un tempo di esecuzione di circa **720** minuti, operando su un ambiente di test locale ed eseguendo un totale di **6237** iterazioni di test. Al termine del processo, i log diagnostici e i risultati dei collaudi sono stati consolidati nei report CSV, che costituiscono la base di dati primari per le misurazioni discusse nelle sezioni seguenti. Al termine del processo, i log diagnostici e i risultati dei collaudi sono stati consolidati nei report CSV, che costituiscono la base di dati primari per le misurazioni discusse nelle sezioni seguenti.

A coronamento di questa fase di esecuzione automatizzata, la Tabella 5.1 riporta il consolidato globale dei risultati grezzi. Tali dati aggregano gli esiti di tutte le iterazioni, suddividendo il campione totale delle mutazioni nelle quattro categorie diagnostiche definite in fase di progettazione (*Success*, *Fragility*, *Obsolescence* e *Not Compiled*). Per fornire ulteriore granularità sulle dimensioni del dataset, la Tabella X contestualizza il

volume delle mutazioni generate, ripartendole per singola tipologia di alterazione.

I log grezzi di esecuzione e i file in formato `.csv` dai quali sono state estratte queste metriche, comprensivi di tutti gli output originali della suite di test, sono liberamente consultabili e riproducibili seguendo le direttive riportate nell'**Appendice A**.

Locator	Total	Success	Fragility	Obsolescence	Not Compiled
Relative	891	344	173	60	314
Absolute	891	314	203	60	314
Robula	891	324	193	60	314
Katalon	891	272	245	60	314
Selenium	891	334	183	60	314
Robula+	891	332	185	60	314
Hook	891	414	103	60	314

Table 5.1: Risultati globali aggregati per ciascuna strategia di localizzazione al termine dell'esperimento

Mutazione	Generazioni Totali	Esecuzioni Totali
(a) Attribute Value Modification	60	420
(b) Attribute Removal	60	420
(d) Text Content Modification	41	287
(e) Text Content Removal	44	308
(f) Tag Movement (container)	57	399
(g) Tag Movement (HTML tree)	127	889
(h) Tag Movement (templates)	127	889
(i) Tag Removal	125	875
(j) Tag Type Modification	125	875
(k) Tag Insertion	125	875

Table 5.2: Distribuzione delle mutazioni valide generate per singola tipologia di operatore e relativo volume di esecuzioni di collaudo

Da questo set di dati primari verranno ora estratti e calcolati gli indici percentuali per rispondere progressivamente alle Domande di Ricerca.

5.2 Applicabilità delle Mutazioni e Tolleranza del Framework (RQ1)

RQ1: *Qual è il grado di applicabilità pratica degli operatori di mutazione standard all'interno di un DOM reale e in che percentuale il framework AngularJS tollera tali alterazioni sintattiche senza incorrere in fallimenti di compilazione (Build Failure)?*

La prima fase dell'analisi si concentra sulla validazione dell'interazione tra lo strumento di iniezione statica (basato su JSoup) e i vincoli architetturali dell'applicazione AngularJS. L'elaborazione dei log di generazione (`mutations.csv`) ha permesso di calcolare il **Mutation Applicability Rate (MAR)**.

Su un pool teorico di **1378** mutazioni possibili, il sistema ha istanziato con successo **891** mutanti, registrando un MAR del **64.65%**. La porzione di mutazioni classificate come non applicabili (esito `KO` in fase di generazione) non rappresenta un'anomalia software, bensì conferma la consapevolezza contestuale dello strumento: il motore ha correttamente interrotto l'iniezione in scenari logicamente incoerenti, come il tentativo di applicare l'operatore "Modifica Identificativo" su un nodo nativamente sprovvisto di tale attributo, o la ricerca di un nodo *parent* in elementi isolati.

Una volta accertata l'applicabilità, l'analisi si è spostata sull'esecuzione dinamica per calcolare il **Framework Tolerance Rate (FTR)**. L'ecosistema Angular, basato sul compilatore rigoroso Ahead-of-Time (AOT), impone che i file template HTML rispettino rigidi vincoli semantici e di data-binding.

Come evidenziato dai report aggregati, su **891** mutazioni effettivamente iniettate e testate per ogni singola strategia, **314** istanze hanno causato un blocco critico del server di sviluppo. Questo si traduce in un FTR del **64.76%**.

L'ispezione dei log ha rivelato che la totalità dei casi `Not Compiled` (pari a circa il **35.2%** del volume di test) è imputabile a mutazioni strutturali invasive. Lo spostamento gerarchico di elementi contenenti direttive strutturali esclusive (come `*ngIf` o `*ngFor`) al di fuori del corretto *scope* della variabile TypeScript associata, corrompe inevitabilmente l'albero di rendering e blocca il riavvio dell'applicazione.
Sulla base dei risultati ottenuti, possiamo rispondere alla domanda di ricerca **RQ1** come segue:
I dati dimostrano che lo strumento generatore è abbastanza affidabile e capace di manipolare il DOM in modo sintatticamente valido. Tuttavia, si evince in modo inequivocabile che l'applicazione del Mutation Testing su framework frontend moderni e fortemente tipizzati come Angular presenta un limite fisiologico: oltre un terzo delle mutazioni strutturali standard, sebbene ampiamente documentate in letteratura per le vecchie applicazioni web statiche, risulta intrinsecamente ineseguibile poiché non tollerato dall'architettura a componenti.

5.3 Analisi della Robustezza Comparativa (RQ2)

RQ2: *Quale strategia di localizzazione garantisce il minor tasso di fragilità (Fragility Rate) e il maggior tasso di sopravvivenza (Survival Rate) a fronte di modifiche strutturali ed estetiche dell'interfaccia utente?*

Al fine di rispondere in modo rigoroso a questa domanda, i dati grezzi estratti dalla campagna di collaudo sono stati "depurati" dalle variabili non imputabili alle strategie di localizzazione. Nello specifico, per il calcolo delle metriche di *Survival Rate* (**SR**) e *Fragility Rate* (**FR**), sono state escluse dal denominatore le 314 mutazioni `Not Compiled` (poiché il test non è stato avviato) e le 60 mutazioni classificate come `Obsolescence` (poiché il fallimento potrebbe non essere imputabile a una fragilità del locatore).

Il nuovo dominio di valutazione è pertanto costituito da **517** mutazioni valide e testabili. Su questa base matematica, la formula **SR+FR=100%** permette di tracciare il profilo di resilienza esatto per le 7 strategie analizzate.

5.3.1 Valutazione Globale

Analizzando i risultati aggregati sull'intero perimetro dell'applicazione *Angular-Spotify*, emerge un quadro prestazionale estremamente eterogeneo che conferma e, per certi versi, estremizza le assunzioni teoriche espresse in letteratura. L'esame del campione di 517 esecuzioni valide permette di tracciare un profilo di resilienza netto, evidenziando il divario tra gli approcci intrusivi, le euristiche algoritmiche e gli strumenti tradizionali.

Il dato di maggiore spicco è senza dubbio il dominio della strategia **Hook-Based**. Svincolandosi totalmente dalla complessa gerarchia dell'albero HTML e dall'estetica dei componenti, questo approccio ha superato indenne **414 mutazioni**, registrando un *Survival Rate* dell'**80.08%**. Questo risultato dimostra empiricamente che l'effort iniziale richiesto per l'iniezione preventiva di identificatori univoci nel codice sorgente è ampiamente ripagato da una drastica riduzione della manutenzione: il tasso di fragilità (*Fragility Rate*) viene infatti confinato al **19.92%**, assorbendo la quasi totalità delle operazioni di *refactoring* simulato.

A debita distanza dal vertice si posiziona un compatto "gruppo di mezzo", composto sia da algoritmi accademici che da approcci relativi classici. Le strategie basate sul **Relative XPath** (**66.54%**), sulle registrazioni native di **Selenium** (**64.60%**) e sulle euristiche di riduzione **ROBULA** e **ROBULA+** (rispettivamente al **62.67%** e **64.22%**) hanno mostrato un comportamento pressoché analogo a livello globale. Pur garantendo un tasso di sopravvivenza sufficiente, l'intera fascia accumula un *Fragility Rate* medio superiore al 35%. Questo tetto prestazionale conferma che affidarsi ad ancore testuali o a gerarchie parziali, per quanto calcolate in modo intelligente, espone fisiologicamente le suite E2E a rotture impreviste quando il DOM subisce alterazioni strutturali.

Infine, l'indagine sulle performance più fragili offre gli spunti di riflessione più inaspettati. Se da un lato il crollo dell'**Absolute XPath** era un fenomeno ampiamente previsto dalla teoria — con un *Fragility Rate* sfiorante il **40%** a causa della sua intrinseca e totale

rigidità gerarchica —, il vero elemento di anomalia è rappresentato da **Katalon Recorder**. L'euristica nativa di questo popolare strumento commerciale ha registrato la prestazione peggiore in assoluto dell'intero esperimento, fallendo 245 esecuzioni valide e sprofondando a un *Survival Rate* di appena il **52.61%**. Un tracollo così marcato (quasi un test su due fallito per cause non legate alla logica applicativa) evidenzia come i selettori generati in fase di *Capture & Replay* da Katalon tendano all'iper-specificità: concatenando attributi multipli o relazioni posizionali complesse per garantire l'univocità iniziale, lo strumento produce locatori estremamente fragili e del tutto inadatti a sopravvivere nell'ecosistema dinamico e mutabile di una *Single Page Application*.
Sulla base dei risultati ottenuti, possiamo rispondere alla domanda di ricerca **RQ1** come segue:
I risultati dimostrano inequivocabilmente che la strategia intrusiva basata su attributi custom (Hook-Based) è l'approccio più robusto, garantendo il maggior Survival Rate globale (80.08%) e il minor Fragility Rate (19.92%). Mentre le euristiche non intrusive (algoritmiche o di Capture & Replay) mostrano marcate fluttuazioni prestazionali e gravi vulnerabilità dipendenti dal livello di dinamicità dell'area testata, l'approccio Hook-Based si conferma l'unico in grado di svincolare sistematicamente i test dalle alterazioni strutturali ed estetiche del DOM in qualsiasi contesto architetturale della Single Page Application.

5.3.2 Valutazione Contestuale per Area di Test

Per ottenere una comprensione granulare del comportamento dei locatori, l'analisi globale è stata disaggregata per mappare i tassi di sopravvivenza sulle tre macro-aree funzionali definite nel Capitolo 4: *Navigazione (Sidebar)*, *Libreria (Playlist)* e *Motore di Ricerca*.

Questa frammentazione è cruciale poiché tali aree presentano livelli di volatilità del DOM eterogenei: la Sidebar è tendenzialmente statica, mentre Ricerca e Libreria sono popolate asincronamente tramite *data-binding* e direttive di iterazione. La Tabella 5.3 mostra i risultati e i valori di *Survival Rate* specifici per ogni area di test (il valore di FR è omesso, data la complementarità delle metriche).

L'analisi parziale dei dati ha rivelato che:

- **Navigazione e Struttura:** I risultati di questa porzione dell'applicativo, caratterizzata da una struttura gerarchica stabile e da una ricca semantica di base, hanno rivelato un andamento in controtendenza rispetto alla media globale. Gli algoritmi di ottimizzazione **ROBULA** e **ROBULA+** hanno registrato le performance migliori in assoluto, raggiungendo un *Survival Rate* rispettivamente dell'**89.52%** e dell'**88.57%**, superando persino la strategia **Hook-Based** (che si assesta comunque su un solido **80.00%**). Questo scostamento indica che, in porzioni di DOM prive di estrema volatilità indotta dal rendering asincrono, le euristiche algoritmiche riescono a ricalcolare percorsi alternativi in modo estremamente efficace. Sorprende in negativo, invece, il tracollo degli strumenti di *Capture & Replay*: **Katalon** (**45.71%**) e **Selenium** (**47.62%**) hanno registrato tassi di sopravvivenza inferiori ai più rigidi locatori **Absolute** (**60.95%**) e **Relative** (**70.47%**), dimostrando che i loro selettori nativi risultano ipersensibili alle alterazioni tipicamente frequenti nei menu di navigazione.

- **Libreria e Playlist:** Spostando l'analisi su una porzione dell'applicativo dominata dal *data-binding* asincrono e dalla generazione dinamica di elementi ripetuti, lo scenario si capovolge drasticamente. Il dato più allarmante appartiene a **Katalon Recorder**, che sprofonda a un *Survival Rate* del **12.83%**, registrando la performance peggiore in assoluto. Questo tracollo evidenzia come i suoi selettori nativi, spesso basati su concatenazioni di attributi generati automaticamente o indici posizionali rigidi, siano totalmente inadatti a gestire liste virtualizzate. Parallelamente, anche gli algoritmi **ROBULA** e **ROBULA+** subiscono un vero e proprio collasso strutturale, precipitando rispettivamente al **18.18%** e **21.92%**. Tale anomalia si spiega con la natura dell'euristica: in presenza di liste con innumerevoli nodi fratelli strutturalmente identici, ROBULA fatica a identificare ancore univoche e una singola mutazione porta il locatore a selezionare la riga sbagliata. In questo contesto ostile, la strategia **Hook-Based** riafferma la sua netta superiorità, mantenendo un solido tasso di sopravvivenza del **72.19%**. Risulta infine interessante il comportamento di Selenium (**58.82%**), che riesce a mitigare in parte i fallimenti — superando nettamente il competitor Katalon e i locatori puramente posizionali — probabilmente grazie alla sua tendenza ad ancorarsi ai testi visibili, i quali tendono a sopravvivere

alle mutazioni gerarchiche.

- **Motore di Ricerca:** I risultati di questa terza macro-area restituiscono uno scenario ancora diverso, caratterizzato da un innalzamento generale del *Survival Rate* per tutte le strategie e da un inaspettato ribaltamento delle gerarchie. Il dato più eclatante è la performance dell'euristica di **Katalon Recorder**, che passa dall'essere il fanalino di coda nella Libreria a dominare quest'area con un tasso di sopravvivenza dell'**88.89%**. Seguono a brevissima distanza gli algoritmi **ROBULA+** (**88.00%**) e **ROBULA** (**87.11%**). Questo recupero è spiegabile analizzando i casi di test associati (interazioni con il campo di input della ricerca e con l'icona di reset): trattandosi di elementi isolati e semanticamente forti, dotati di attributi nativi molto specifici (come `placeholder` o `type`), gli strumenti di *Capture & Replay* riescono a generare selettori estremamente stabili. Allo stesso tempo, l'assenza di nodi fratelli identici permette agli algoritmi di ricavare percorsi univoci e brevi. In questo contesto di generale efficacia, la strategia **Hook-Based** si attesta all'**86.67%**, confermando un dato empirico di assoluta rilevanza: pur non primeggiando in questa singola area, l'approccio basato su attributi custom è l'unica strategia ad aver mantenuto un *Survival Rate* superiore al **72%** in *tutti e tre* i contesti architetturali affrontati, eleggendosi a strumento più affidabile per garantire la manutenibilità di una suite E2E su vasta scala.

Area Navigazione e Struttura						
Locator	**Total**	**Success**	**Fragility**	**Obsolescence**	**Not Compiled**	**SR**
Relative	225	74	29	43	77	70.47%
Absolute	225	64	41	43	77	60.95%
Robula	225	94	11	43	77	89.52%
Katalon	225	48	57	43	77	45.71%
Selenium	225	50	55	43	77	47.62%
Robula+	225	93	12	43	77	88.57%
Hook	225	84	21	43	77	80.00%
Area Libreria e Playlist						
Locator	**Total**	**Success**	**Fragility**	**Obsolescence**	**Not Compiled**	**SR**
Relative	311	86	101	5	119	45.98%
Absolute	311	85	102	5	119	45.45%
Robula	311	34	153	5	119	18.18%
Katalon	311	24	163	5	119	12.83%
Selenium	311	110	77	5	119	58.82%
Robula+	311	41	146	5	119	21.92%
Hook	311	135	52	5	119	72.19%
Area Motore di Ricerca						
Locator	**Total**	**Success**	**Fragility**	**Obsolescence**	**Not Compiled**	**SR**
Relative	355	184	41	12	118	81.78%
Absolute	355	165	60	12	118	73.33%
Robula	355	196	29	12	118	87.11%
Katalon	355	200	25	12	118	88.89%
Selenium	355	174	51	12	118	77.33%
Robula+	355	198	27	12	118	88.00%
Hook	355	195	30	12	118	86.67%

Table 5.3: Risultati e Tasso di Sopravvivenza (SR) aggregato per le tre macro-aree funzionali dell'applicazione

5.4 Analisi dell'Impatto per Tipologia di Mutazione (RQ3)

RQ3: *Esiste una correlazione tra specifiche tipologie di mutazione e il fallimento di determinate famiglie di locatori?*

Per comprendere a fondo la semantica dei fallimenti e rispondere all'ultima domanda di ricerca, i dati globali sono stati disaggregati per incrociare le sette strategie di localizzazione con le singole tipologie di mutazione iniettate nel codice sorgente. Tramite questa indagine incrociata (*Specific Fragility Analysis*), sono stati isolati esclusivamente i fallimenti per fragilità, calcolando il relativo *Specific Fragility Rate* (**SFR**) per mappare le vulnerabilità specifiche di ciascun locatore.

Dall'analisi dei risultati emerge un primo pattern di rottura netto relativo alle mutazioni strutturali e gerarchiche, come l'inserimento di tag (`tag_ins_mut`) o gli spostamenti ad altro template (`tag_mov_temp_mut`). Le alterazioni che modificano la profondità dell'albero HTML hanno generato un impatto devastante sugli strumenti basati su percorsi e indici rigidi. Il caso più eclatante è rappresentato da **Katalon Recorder**, che registra uno SFR del **41.67%** sulla singola mutazione di inserimento tag, superando negativamente gli altri locatori. Questo dimostra che l'uso intensivo di relazioni posizionali nelle sue registrazioni di *Capture & Replay* lo rende del tutto inadatto a tollerare l'aggiunta di nuovi livelli gerarchici intermedi. Analogamente, l'**Absolute XPath** subisce pesantemente gli spostamenti degli elementi all'interno del layout, raggiungendo un tasso di fragilità dell'**85.86%**. Al contrario, le euristiche ad albero come **ROBULA+** e la strategia **Hook-Based** riescono ad assorbire l'inserimento di nuovi contenitori con tassi di fragilità nettamente inferiori, attestandosi rispettivamente al **33.33%** e a un trascurabile **4.17%**.

Le gerarchie prestazionali si invertono radicalmente quando l'operatore di mutazione altera le proprietà intrinseche ed estetiche del nodo target. Strumenti come **Selenium** e **Katalon** subiscono il loro picco di fragilità semantica a fronte della rimozione di attributi (`attr_rem_mut`), registrando uno SFR rispettivamente del **52.17%** e del **50.00%**. Tale collasso conferma la loro estrema dipendenza dalle classi CSS e dagli ID nativi per l'identificazione univoca degli elementi. In questo cluster di mutazioni spicca in negativo anche l'**Absolute XPath** a fronte della modifica del nome del tag (`tag_type_mod_mut`, es. da `<span>` a `<div>`): essendo la nomenclatura del suo percorso assoluto "*hardcoded*", la strategia colleziona un preoccupante SFR del **69.49%**, risultando in assoluto la più vulnerabile a questo specifico refactoring.

L'analisi granulare permette infine di contestualizzare le vulnerabilità residue della strategia intrusiva. Pur risultando quasi totalmente immune alle alterazioni strutturali, l'approccio **Hook-Based** registra un picco di fragilità (SFR pari al **28.26%**) in corrispondenza della rimozione degli attributi (`attr_rem_mut`). Poiché lo strumento di mutazione è stato esplicitamente configurato per ignorare e preservare i metadati custom dedicati al testing (es. `x-test`), questo tasso di fallimento non è imputabile a un errore di localizzazione del nodo nel DOM. Piuttosto, tale dato evidenzia una dinamica fondamentale del collaudo E2E: la corretta identificazione di un elemento non ne garantisce la corretta

fruibilità. La rimozione casuale di altri attributi funzionali o direttive di *data-binding* (es. `href`, `disabled`, o classi che regolano la visibilità del componente) finisce per alterare lo stato o il comportamento dell'interfaccia. Di conseguenza, pur riuscendo ad agganciare correttamente il *target*, il test script fallisce nelle successive fasi di interazione (*Action*) o verifica (*Assertion*). Questa evidenza empirica dimostra che l'infallibilità del locatore è una condizione necessaria, ma non sempre sufficiente, per garantire l'assoluta resilienza di un test di fronte a refactoring distruttivi della logica di presentazione.
I valori percentuali relativi allo Specific Fragility Rate per ogni operatore di mutazione, con dettaglio dei valori assoluti, sono schematizzati nella Tabella 5.4. La colonna "Mutazione" riprende gli identificativi delle mutazioni indicate nella tabella 1.1. È opportuno sottolineare l'assenza della mutazione *c* (mutazione dell'identificativo) dai risultati; nell'applicazione in esame, infatti, il campione di elementi target non presentava alcun identificativo ID di base, rendendo di fatto inapplicabile questa specifica alterazione semantica.

Al fine di condurre un'analisi ancora più granulare, le esecuzioni sono state disaggregate valutando l'impatto delle mutazioni in base alla posizione topologica del nodo alterato. La Tabella 5.5 riporta nel dettaglio i valori percentuali e assoluti dello *Specific Fragility Rate* a seconda che la mutazione abbia colpito direttamente il target, un suo fratello, padre, *ancestor* o *template* (secondo le casistiche introdotte nel Listing 1.1).

Sulla base dei risultati ottenuti, possiamo rispondere alla domanda di ricerca **RQ3** come segue:
I log sperimentali dimostrano una correlazione diretta, misurabile e prevedibile tra il tipo di refactoring simulato e la rottura dei test. Le strategie strutturali collassano inevitabilmente di fronte alle mutazioni gerarchiche, mentre le euristiche basate sulla registrazione soffrono in modo sistematico l'alterazione degli identificatori visivi. Questa polarizzazione conferma in via definitiva che l'affidamento alle proprietà native e instabili del DOM espone la suite E2E a inevitabili rotture contestuali, criticità che possono essere eradicate unicamente adottando identificatori disaccoppiati dalla logica di business e di presentazione.

	SFR per locatore						
Mutazione	**Absolute**	**Hook**	**Katalon**	**Relative**	**Robula**	**Robula+**	**Selenium**
a	88.33%	88.33%	88.33%	88.33%	88.33%	88.33%	90.00%
	(53/60)	(53/60)	(53/60)	(53/60)	(53/60)	(53/60)	(54/60)
b	34.78%	28.26%	50.00%	32.61%	39.13%	39.13%	52.17%
	(16/46)	(13/46)	(23/46)	(15/46)	(18/46)	(18/46)	(24/46)
d	11.76%	8.82%	50.00%	35.29%	35.29%	29.41%	26.47%
	(4/34)	(3/34)	(17/34)	(12/34)	(12/34)	(10/34)	(9/34)
e	82.05%	82.05%	89.74%	87.18%	84.62%	84.62%	84.62%
	(32/39)	(32/39)	(35/39)	(34/39)	(33/39)	(33/39)	(33/39)
f	28.30%	16.98%	49.06%	26.42%	37.74%	37.74%	35.85%
	(15/53)	(9/53)	(26/53)	(14/53)	(20/53)	(20/53)	(19/53)
g	89.52%	88.46%	91.35%	88.57%	88.57%	88.57%	89.52%
	(94/105)	(92/104)	(95/104)	(93/105)	(93/105)	(93/105)	(94/105)
h	85.86%	81.63%	88.78%	81.82%	88.00%	87.88%	85.00%
	(85/99)	(80/98)	(87/98)	(81/99)	(88/100)	(87/99)	(85/100)
i	58.00%	45.54%	49.02%	41.00%	40.59%	38.61%	41.58%
	(58/100)	(46/101)	(50/102)	(41/100)	(41/101)	(39/101)	(42/101)
j	69.49%	25.41%	55.93%	52.94%	48.31%	43.33%	47.11%
	(82/118)	(31/122)	(66/118)	(63/119)	(57/118)	(52/120)	(57/121)
k	17.50%	4.17%	41.67%	20.83%	30.83%	33.33%	23.33%
	(21/120)	(5/120)	(50/120)	(25/120)	(37/120)	(40/120)	(28/120)

Table 5.4: Specific Fragility Rate (SFR) per tipologia di mutazione, con dettaglio dei valori assoluti (Fragilità / Test Validi)

	SFR per tipo di tag						
Tag	**Absolute**	**Hook**	**Katalon**	**Relative**	**Robula**	**Robula+**	**Selenium**
alpha	58.55%	47.44%	65.09%	55.17%	58.80%	59.05%	57.26%
	(137/234)	(111/234)	(151/232)	(128/232)	(137/233)	(137/232)	(134/234)
beta	67.87%	52.47%	67.87%	62.16%	60.36%	58.74%	64.13%
	(150/221)	(117/223)	(150/221)	(138/222)	(134/222)	(131/223)	(143/223)
gamma	49.02%	40.69%	65.20%	50.00%	58.82%	57.84%	52.45%
	(100/204)	(83/204)	(133/204)	(102/204)	(120/204)	(118/204)	(107/204)
delta	63.48%	45.69%	58.12%	53.85%	52.14%	50.00%	51.69%
	(73/115)	(53/116)	(68/117)	(63/117)	(61/117)	(59/118)	(61/118)

Table 5.5: Specific Fragility Rate (SFR) per tipologia di tag, con dettaglio dei valori assoluti (Fragilità / Test Validi)

5.5 Minacce alla Validità

Come in ogni studio empirico, i risultati ottenuti in questa ricerca sono soggetti a diverse minacce alla validità, che sono state identificate e, ove possibile, mitigate durante la progettazione dell'esperimento.

5.5.1 Minacce alla Validità di Costrutto (Construct Validity)

Questa categoria riguarda il rapporto tra la teoria e l'esperimento: stiamo davvero misurando ciò che crediamo di misurare?

- **Rappresentatività e Correttezza delle Mutazioni:** Una minaccia interinseca al *Mutation Testing* applicato alla UI è la fedeltà degli operatori di mutazioni rispetto ai reali *refactoring* degli sviluppatori. Sebbene il generatore sia stato progettato per simulare alterazioni comuni, esiste il rischio che alcune mutazioni generate sintatticamente non abbiano senso semantico nel contesto specifico di Angular.
- **Metriche di Valutazione:** L'uso dello Specific Fragility Rate (SFR) e del Survival Rate assume che un test fallito corrisponda sempre a un locatore rotto. Come emerso nell'analisi della strategia Hook-Based, a volte il test fallisce per un'asserzione di business non verificata pur avendo localizzato l'elemento. Questo sovrappone parzialmente la fragilità del locatore con la fragilità della logica di test.

5.5.2 Minacce alla Validità Interna (Internal Validity)

Questa categoria riguarda i fattori che possono aver influenzato i risultati senza che ce ne accorgessimo.

- **Test Flakiness e Sincronizzazione:** I test E2E sono notoriamente inclini al flakiness a causa di latenze di rete, tempi di rendering del browser o colli di bottiglia computazionali della macchina host. Nelle *Single Page Applications* come Angular, il rendering asincrono può causare fallimenti ("*timeout*") che non dipendono dall'effettiva rottura del locatore, ma da attese non sufficientemente calibrate nel codice di test di fronte a lievi rallentamenti ambientali.
- **Configurazione degli Strumenti:** Strumenti come Selenium e Katalon possiedono parametri di configurazione (es. timeout di ricerca). L'esperimento li ha utilizzati in configurazione standard per valutare l'approccio out-of-the-box, ma un tuning iper-specifico di questi parametri da parte di un ingegnere esperto potrebbe mitigare alcuni dei fallimenti registrati.

5.5.3 Minacce alla Validità Esterna (External Validity)

Questa categoria risponde alla domanda: possiamo generalizzare questi risultati al di fuori di questo esperimento?

- **Generalizzazione del Dominio Applicativo:** L'esperimento è stato condotto su una singola applicazione target (*Angular-Spotify*). Sebbene rappresenti un'ottima *baseline* per le moderne architetture basate su componenti, i risultati potrebbero variare applicando la medesima suite su framework concettualmente diversi (come React o Vue.js), o su applicazioni legacy stratificate che non utilizzano lo Shadow DOM o framework reattivi.
- **Generalizzazione dei Locatori:** Lo studio ha valutato sette strategie specifiche. Sebbene siano altamente rappresentative delle tecniche industriali e accademiche attuali, l'introduzione di framework di nuova generazione (come Cypress o Playwright, che possiedono meccanismi di auto-attesa e locatori nativi profondamente diversi da WebDriver) o locatori basati su Intelligenza Artificiale (AI-healing) potrebbe generare tassi di resilienza differenti.

–6–
Conclusioni

6.1 Riepilogo del contesto

Negli ultimi anni, l'adozione di architetture moderne come le Single Page Application (SPA) ha reso le interfacce web estremamente dinamiche e reattive. In questo ecosistema, il collaudo End-to-End (E2E) rappresenta uno snodo cruciale per garantire la qualità del software, ma sconta un elevato costo di manutenzione a causa della "fragilità" dei test: i locatori utilizzati per identificare gli elementi della UI tendono infatti a rompersi frequentemente a fronte di normali evoluzioni stilistiche o strutturali del DOM (refactoring).

6.2 Sintesi del processo sperimentale

Per analizzare e misurare oggettivamente questo fenomeno, il presente lavoro di tesi ha condotto uno studio empirico basato sul Mutation Testing applicato alla UI. Utilizzando un'applicazione Angular open-source di riferimento (*Angular-Spotify*), sono state iniettate in modo automatizzato centinaia di mutazioni sintetiche (sia strutturali che semantiche) nel codice sorgente. Su questo DOM alterato, è stata valutata la resilienza di sette diverse strategie di localizzazione: approcci posizionali (Absolute e Relative XPath), strumenti industriali di Capture & Replay (Selenium e Katalon), algoritmi euristici (ROBULA e ROBULA+) e un approccio intrusivo basato su attributi custom (Hook-Based).

6.3 Presentazione dei risultati

I dati raccolti su oltre 500 mutazioni valide (con un totale di più di 3000 esecuzioni significative) hanno fornito risposte inequivocabili alle Domande di Ricerca. A livello globale, la strategia **Hook-Based** si è confermata la più robusta in assoluto, garantendo un Survival Rate superiore all'**80%**. Disaggregando i dati per contesti architetturali e tipologie di mutazione, è emerso che non esiste un silver bullet tra le alternative non intrusive: gli algoritmi come **ROBULA** eccellono nelle aree statiche ma collassano sulle

liste dinamiche, mentre i tool commerciali come **Katalon** si dimostrano efficaci sugli input isolati ma sprofondano a fronte di alterazioni gerarchiche, a causa della loro dipendenza da indici posizionali e classi CSS.

I risultati di questo studio offrono una direttiva chiara per i team di sviluppo. Affidarsi alle proprietà native e instabili del DOM per il testing E2E espone inevitabilmente la suite a specifici *blind spot* e a continue fragilità. Investire un effort iniziale nell'iniezione di metadati dedicati esclusivamente al collaudo (es. x-test-hook) rappresenta l'unico compromesso tecnico in grado di disaccoppiare la logica di test dalle naturali evoluzioni della UI, abbattendo drasticamente i costi di manutenzione a lungo termine.

6.4 Limiti dello studio

Lo studio presenta dei limiti fisiologici legati alla natura empirica dell'esperimento. In primo luogo, la valutazione è stata condotta su una singola applicazione target, limitando parzialmente la generalizzazione assoluta dei tassi di fallimento ad altri framework (come React o Vue.js). Inoltre, l'utilizzo degli strumenti di localizzazione nella loro configurazione out-of-the-box e la presenza intrinseca di flakiness dovuta ai tempi di rendering asincrono di Angular rappresentano minacce alla validità che, seppur mitigate metodologicamente, potrebbero essere ulteriormente affinate tramite un tuning iper-specifico dei framework di test.

6.5 Lavori futuri

Partendo dalle evidenze di questo lavoro, i futuri sviluppi di ricerca potranno esplorare diverse direzioni. Sarà interessante estendere il framework di *Mutation Testing* ad altre librerie moderne per validare l'universalità dei pattern di rottura individuati. Inoltre, un filone di ricerca promettente riguarda lo sviluppo di plugin per l'iniezione automatizzata e trasparente degli Hook durante il processo di build dell'applicazione, annullando così l'overhead per gli sviluppatori. Infine, si potrebbe valutare l'efficacia dei nuovi locatori basati su Intelligenza Artificiale (AI-healing), per comprendere se le moderne reti neurali riescano a superare i limiti strutturali che affliggono le attuali euristiche algoritmiche.

–A–
Riproducibilità dello Studio e Organizzazione del Materiale

Al fine di garantire la totale trasparenza scientifica e la riproducibilità dello studio empirico condotto, l'intero ecosistema software sviluppato, i file di configurazione e i dataset generati sono stati resi pubblicamente disponibili. L'intero materiale è centralizzato all'interno di un'unica repository GitHub [5], organizzata strutturalmente per facilitarne la navigazione e il riutilizzo da parte di terzi.

Nello specifico, la repository è articolata nelle seguenti macro-aree:

- **Codice Sorgente e Software Eseguibile:** L'ambiente di lavoro principale ospita tutti i moduli *Maven* necessari per la compilazione e l'esecuzione dell'applicativo. Per agevolare l'immediato utilizzo dello strumento da parte di altri sviluppatori, aggirando le tempistiche di build dell'ambiente locale, la sezione *Releases* della repository include gli artefatti precompilati in formato `.jar`, pronti per essere eseguiti in modalità *plug-and-play*.

- **Documentazione Tecnica Modulare:** Le istruzioni operative e i dettagli di configurazione sono stati mantenuti nella loro forma nativa all'interno di file `README.md` dedicati. La documentazione è stata segmentata tematicamente per fornire una guida d'uso generale, affiancata da manuali specifici per task avanzati, quali le direttive per la generazione dei percorsi tramite gli algoritmi ROBULA e ROBULA+ e i comandi per l'iniezione degli attributi *Hook-Based* nel codice sorgente.

- **Dataset e Replicabilità dell'Esperimento (`test-suite`):** Per permettere la replica esatta dei risultati discussi in questo elaborato, è stata predisposta una directory specifica denominata `test-suite`. Al suo interno sono archiviati tutti i file di configurazione impiegati durante le sessioni di collaudo, unitamente all'interezza degli output originali generati dalla suite di test. Quest'area contiene i log e i dataset grezzi (la cui struttura e schematizzazione tabellare sono state precedentemente dettagliate nel Capitolo 3), offrendo al lettore la possibilità di verificare e ricostruire autonomamente le metriche statistiche calcolate.

Per garantire una corretta replicabilità dei test eseguiti, è disponibile un apposito file readme che spiega nel dettaglio la configurazione corretta dell'applicazione testata (Angular-Spotify).

Questa organizzazione organica del materiale allegato non solo corrobora la validità dei dati presentati, ma mette a disposizione della comunità un'infrastruttura completa e documentata per future evoluzioni del Mutation Testing applicato alle interfacce web.

Bibliografia

[1] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. “Reducing web test cases aging by means of robust xpath locators”. In: *25th IEEE International Symposium on Software Reliability Engineering Workshops* (2014), pp. 449–454.

[2] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. “Robula+: An algorithm for generating robust xpath locators for web testing”. In: *Journal of Software: Evolution and Process 28.3* (2016), pp. 177–204.

[3] A. R. Fasolino and P. Tramontana. “Towards the Generation of Robust E2E Test Cases in Template-based Web Applications”. In: *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE.* (2022), pp. 104–111.

[4] A. R. Fasolino and P. Tramontana. “Investigating the robustness of locators in template-based Web application testing using a GUI change classification model”. In: *The Journal of Systems and Software 210 (2024) 111932* (2024).

[5] Liberti S. “Automatic Angular E2E Testing Suite”. In: *Github Repository* (2025). DOI: `https://github.com/sim-liberti/Automatic-Angular-E2E-Testing-Suite.git`.