POLITECNICO DI TORINO

Laurea MAGISTRALE in
INGEGNERIA INFORMATICA

Tesi di Laurea MAGISTRALE

Sviluppo di un tool di mutant injection nel contesto di un ambiente di GUI Testing con gamification

Supervisori

Prof. LUCA ARDITO

Dott. RICCARDO COPPOLA

Dott. TOMMASO FULCINI

Candidato

RICCARDO GABELLONE

04/2022

Sommario

Contesto: Il *GUI Testing* è una disciplina del Software Testing basata sulla interazione grafica tra il tester e l'applicazione target tramite i vari elementi dell'interfaccia che la compongono.

Il *Mutation Testing* è una disciplina del Software Testing basata sulla creazione di artificiale di bug all'interno del codice di un software, una tecnica puramente di white box testing.

Nonostante tali pratiche siano un buon mezzo di validazione del software sviluppato, spesso si tende a trascurarle a causa dei costi elevati derivanti da un approccio manuale e dalla fragilità dei test risultanti da metodi puramente automatizzati.

Un approccio più recente è fornito dall'*Augmented Testing*, che prevede di generare sequenze di test con l'ausilio di informazioni visive che facilitano la creazione di test case rispetto a quanto era possibile con tecniche di generazioni precedenti.

Obiettivo: Questa tesi propone la combinazione delle suddette tecniche di testing, cercando di automatizzare la generazione e l'inserimento delle mutazioni all'interno della GUI di un'applicazione web (WA) con l'ausilio dell'integrazione di alcuni concetti di Gamification, ossia l'inserimento di elementi e meccanismi tipici del game design all'interno di contesti non ludici al fine di migliorare la qualità delle test suite risultanti e di stimolare l'aspetto emotivo del tester.

Metodo: La tesi si concentra sullo sviluppo di un tool (in Kotlin) che automatizzi la creazione di *mutants* all'interno del Document Object Model (DOM) di una pagina web, tramite la manipolazione del codice sorgente HTML. In un secondo momento il tool è stato integrato, insieme con alcuni elementi di Gamification, all'interno di un plugin Java per un prototipo di software di Augmented Testing: Scout. L'obiettivo, in questo caso, dell'inserimento di elementi ludici, è comunicare al tester l'eventualità della scoperta del mutant all'interno di una WA, cercando di renderlo più coinvolto durante la sessione di testing.

Risultati: Un campione ristretto composto da studenti, laureati (o laureandi) in Ingegneria Informatica, ha utilizzato il prototipo di Scout con i plugin di Gamification, DisplayStatsOverlay e SeleniumPlugin attivi, con integrata la *mutant injection*: è stato confrontato il numero di mutant trovati dal tool con quelli segnalati manualmente tramite voto, per ogni tester, evidenziando un totale del 63% di mutazioni che erano state correttamente segnalate, buon risultato considerando che circa la metà di quelli generati sia stata segnalata almeno una volta.

Conclusioni: La fase di sperimentazione ha portato ad evidenziare i vantaggi dell'inclusione degli elementi di Gamification ma anche gli svantaggi e le limitazioni sofferte dal tool, sia per quanto riguarda la generazione di alcune mutazioni in alcune WA sia per quanto riguarda l'utilizzo di Scout. Per sviluppi futuri è possibile perfezionare l'inserimento dei mutants nel sorgente della pagina e la generazione di quei tipi che vadano ad interagire anche con gli eventi dinamici della WA.

Ringraziamenti

Arrivare fin qui è segno di orgoglio, per chiunque si trovi a percorrere la stessa strada, per chiunque ha la fortuna di poterlo fare. Probabilmente non mi renderò subito conto di quanto fatto, giusto perché la strada è appena iniziata.

Vorrei ringraziare tutti quanti, tutte le persone che ho avuto il piacere e la fortuna di incontrare in questa strada, quelle lontane e quelle vicine. I colleghi e gli amici, persone uniche che in qualsiasi momento hanno contribuito a migliorarmi grazie a quanto passato insieme e che contribuiscono ancora a farlo.

Grazie a tutta la mia famiglia, ai miei genitori e i miei fratelli, sostegno di tutte le scelte che mi hanno portato fin qui e ad essere quello che sono.

"Life is very short, and there's no time for fussing and fighting, my friend"
John Lennon, Paul McCartney

"Lasciate il mondo un po' migliore di come lo avete trovato"
Baden Powell

"Look up at the stars, not down at your feet"
Stephen Hawking

Indice

Elenco delle tabelle

Elenco delle figure

Acronimi

SE
: Software Engineering

SUT
: System Under Test

GUI
: Graphical User Interface

DOM
: Document Object Model

CI
: Continuous Integration

AT
: Augmented Testing

C&R
: Capture & Replay

API
: Application Programming Interface

Capitolo 1

Introduzione

In questo lavoro di tesi si prevede di affrontare due principali argomenti, ambito di vari studi nell'ingegneria del software: il Testing e la Gamification. In particolare, verrà trattata l'analisi dello sviluppo di un tool per l'inserimento automatico di bug, quelli che chiameremo da ora *mutants* (o *mutations*), all'interno di applicazioni web varie, ad esempio alcuni siti come wikipedia.org o altri. Successivamente verrà trattata la sua integrazione all'interno di un software per il testing pre-esistente di nome Scout, in cui sono stati stati precedentemente inseriti plugin per l'Augmented Testing e per la Gamification.

Il tool sopra citato tratta la ***Mutant Injection***, la tecnica alla base del *Mutation Testing*, in cui si ha l'obbiettivo di creare artificialmente difetti e bug all'interno del codice delle applicazioni software, possibile sia sulle logiche e gli algoritmi interni che sul codice che si riflette direttamente sull'eventuale GUI dell'applicativo, con lo scopo di migliorarne la robustezza durante la fase di testing.

Si andranno perciò a vedere nel dettaglio questi aspetti di background per poi entrare nell'analisi della soluzione scelta nello sviluppare il tool di *mutants injection*. Quest'ultima, utilizza Selenium, framework comune e molto utilizzato nel software testing per la manipolazione del DOM di una pagina web tramite browser, che gli permette così di creare e modificare nuovi elementi, quelli che chiameremo *widget*, all'interno della pagina, i *mutants* per l'appunto, a partire da quelli già esistenti e resi disponibili all'utente non appena viene caricata la GUI.

La soluzione cerca di automatizzare quanto più possibile la creazione casuale di questi *mutants*, nonostante le diverse limitazioni dovute a diversi fattori che affronteremo nel dettaglio più avanti. In generale, si cerca anche di perfezionare il modo con cui vengono inseriti e generati, per renderli il più possibile riconoscibili da parte del tester così da creare una più consapevole partecipazione alla sessione di test stessa. Il successo nell'aver riconosciuto queste mutazioni, verrà premiato, tramite il plugin della Gamification, e tutte le statistiche relative gli verranno mostrate al termine della sessione.

In sintesi, il resto della tesi affronta i capitoli nel modo seguente:

- Nel capitolo 2 si descrive il background del testing di applicazioni web, mutation testing e gamification applicata nell'ingegneria del software
- Nel capitolo 3 si descrive l'architettura utilizzata per creare il tool di injection dei mutants, con relativi risultati attesi
- Nel capitolo 4 si descrive il metodo di validazione usato per il tool precedentemente analizzato, mettendo in risalto i risultati di alcune sessioni affrontate da vari tester volontari, tramite l'ausilio di questionari finali
- Nel capitolo 5 si ricapitolano le limitazioni riscontrate e come sarà possibile risolverle per ricerche e applicazioni future del tool

Capitolo 2

Background

Il focus generale della tesi verte sul mutation testing. Per affrontare lo stato dell'arte che troviamo in letteratura, dobbiamo prima porre l'attenzione su cosa sia il testing all'interno dei processi di creazione di un software, analizzandone vantaggi e svantaggi. (Figura 2.1)

Figura 2.1: Software Testing keywords.

La fase di testing è l'insieme dei processi con lo scopo di garantire il corretto funzionamento del software e quindi la sua qualità da parte dell'utilizzatore finale. L'idea di base è quella di colmare le lacune di correttezza e completezza che possono verificarsi durante lo sviluppo, in modo così da portare il prodotto finale a rispettare le specifiche tecniche e i requisiti dell'azienda o del cliente stesso. Da ciò se ne traggono vari benefici, tra cui

- *guadagno economico*, trovare i bug del software prima della sua messa in produzione è meno dispendioso che farlo dopo
- *sicurezza*, i test servono anche per la fase successiva a quella di risk assessment in cui trovare falle potrebbe essere essenziale per garantire una certa qualità al consumatore
- *soddisfazione utente*, un prodotto migliore aiuta l'esperienza dell'utente ad essere più adeguata e l'azienda a crescere

al contrario comunque, come detto, se si applica la fase di test e bug-fixing in fase di produzione e post-release, lo sforzo economico richiesto risulterebbe maggiore, con un conseguente e possibile calo della soddisfazione dell'utente finale [34, Gallotti]. Nonostante ciò, il testing andrebbe fatto in generale in tutte le fasi dello sviluppo di un software così da minimizzare la possibilità di problemi vari.

Possiamo trovare varie tipologie di test che possono essere svolte durante la relativa fase nello sviluppo di un software, ad esempio: *unit test*, *integration test*, *Graphical User Interface (GUI) test*, *e2e test* e i più generali *white box test* e *black box test*. Per ognuna di queste tipologie è possibile effettuare dei *test automatizzati*, che sono quelli verso cui la tendenza, il trend, degli ultimi decenni nella SE, sta virando. Esistono vari tools che permettono l'esecuzione automatica di test, così da ridurre i costi soprattutto in termini di tempo impiegato. Quelli mirati ad andare a testare le parti grafiche del software in esame (SUT), possono presentare una GUI moderna che aiuta a generare test case specifici e ripeterli con semplicità. Distinguiamo alcuni tipi di tool nella tabella 2.1. I test di tipo *Automated Input Generation*, ad esempio, hanno lo scopo di generare insiemi di valori di input per un programma, o una componente di esso, tipicamente con lo scopo di raggiungere un particolare stato dell'esecuzione [14, Orso et al.]. Questa tecnica, nell'ultima decade,

Tipologia	Descrizione
Automation APIs / Frameworks	Interfaccia formata da tutti gli elementi del software in modo gerarchico. Tendenza all'universalità applicativa
Record and Replay	Facile creazione di script di testing, iterazioni registrate ed eventualmente riprodotte (macro)
Automated Test Input Generation	Generare input, eventi di sistema
Bug Reporting / Monitoring	Creano report automatici sugli eventi di errore nel corso dell'esecuzione del software

Tabella 2.1: Elenco tools di testing.

ha ripreso piede, producendo diversi e affidabili risultati e contributi, dovuta al fatto che, probabilmente, il miglioramento tecnologico, in particolare, capacità di calcolo, dei sistemi moderni abbia aiutato lo sviluppo del processo. In aggiunta a ciò, i ricercatori hanno anche trovato dei metodi per combinare adeguatamente le tecniche di testing con tecniche di verifica. Merita citazione lo [3, Yogi Project at Microsoft Research], che cerca di combinare testing, il quale sotto certe condizioni può creare falsi negativi ma comunque scovare i veri errori, con la verifica statica, la quale può produrre falsi positivi anche se completa: tutto allo scopo di sfruttare i punti forza e ridurre le debolezze del codice.

Ancora in [14, Orso et al.] viene posta l'attenzione sullo stato della pratica nell'area del software testing, dove hanno preso piede i *frameworks* per l'esecuzione dei test e l'adozione sempre più ampia di cicli nell'intero processo, quali la *continuous integration (CI)* ad esempio. Per quanto riguarda i frameworks, il più famoso per così dire e comunque usato è [38, **JUnit**], il quale permette agli sviluppatori di scrivere test ripetibili in quello che è ormai uno standard de-facto. Questo permette di progettare facilmete le 4 parti di un test case: setting stato iniziale; chiamate alle funzionalità sotto test; controllo risultati; applicazione eventuali e necessari pulizie del codice. Un altro contributo, come detto prima, lo da anche la CI. L'idea di base è quella di consegnare (*commit*), una o più volte al giorno, tutte le copie del codice del software in sviluppo. Vengono per cui applicati i test automatizzati di volta in volta a questi commit, questo per assicurare che il codebase rimanga stabile e che possa essere considerato affidabile. Se qualche test fallisce, gli sviluppatori responsabili devono prontamente correggere le falle, ciò permette un aumento generale della velocità nelle fasi dello sviluppo. Seppur vantaggiosa, l'integrazione dei test automatici nella CI non è poco complessa, soprattutto se riguarda i tipi di test end-to-end, dove bisogna coinvolgere e mimare tutto l'ambiente reale di applicazione del software come databases o scambi di pacchetti di rete, oppure se riguarda i test sulla GUI che talvolta possono richiedere un intervento manuale.

Oltre ai metodi manuali e automatici, brevemente sopra descritti, in [29, Nass et al. 2019] viene presentata una nuova tecnica con il nome di *Augmented Testing* (AT), definita come un tecnica di testing visuale, basata sul principio *Record & Replay* (o *Capture & Replay*, C&R), che prevede di generare sequenze di test con l'ausilio di informazioni utili per il task del tester: un'interfaccia si sovrappone alla GUI del SUT, arricchendola con informazioni supplementari, ad esempio, permette di registrare commenti su certe aree dell'interfaccia. Uno schema che riassume il funzionamento base lo troviamo in figura 2.2. In breve, il tester interagisce con l'*Augmented Layer* attraverso azioni che vengono catturate secondo la loro opportuna tipologia (click del mouse, trascinamento, scorrimento, ecc.) e vengono poi riprodotte sulla GUI originale. L'AT viene implementato da **Scout** [31, Nass et al. 2020], un prototipo di tool di web application testing, sviluppato in Java, che applica tali tecniche ad una applicazione web (si veda 2.1.2).

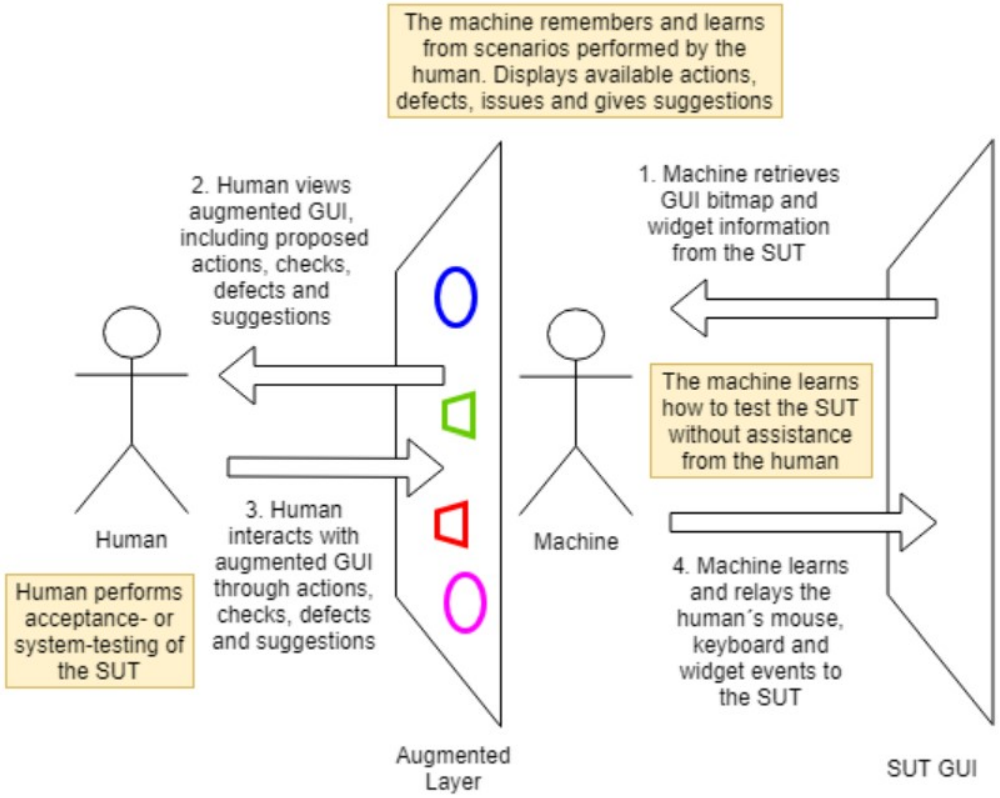


Figura 2.2: Augmented Testing workflow.

2.1 Web Application Testing

Gli esempi visti finora, sui tipi di testing, erano una panoramica generale per ogni tipo di software. Analizziamo qui, in particolare, il testing per le applicazioni web. Innanzitutto, la velocità di diffusione delle applicazioni web, ha favorito l'affermazione di librerie e framework in grado di facilitare la vita degli sviluppatori. Il linguaggio comune nella programmazione web è il *js* (JavaScript) ma sempre più aziende stanno investendo in nuove tecnologie, come *Google* con **Flutter** o *Meta* (ex *Facebook*) con **React**. Per cui è doveroso porre più attenzione alla fase di test, essendo tecnologie in fase di sviluppo, in particolare, l'uso di server e browser, del backend e del frontend, rende le applicazioni web più eterogenee, poiché sviluppate in modo differente, e quindi più inclini all'errore e complesse da testare dato devono garantire un'adeguata interoperabilità. Le difficoltà sono perciò varie, a partire dall'architettura di distribuzione, basata sulle richieste e risposte HTTP tra client e server, dove si devono gestire correttamente tutte le chiamate asincrone per allinearle con lo stato dell'applicazione e dove quindi si possono verificare problemi di collegamento.

In [11, Garousi et al.] viene raccolta la letteratura riguardo al testing sulle applicazioni web e analizzata sotto vari aspetti, tra cui, da porre in rilievo, la differenza tra tecniche manuali e automatiche. Viene indicato, come già detto, che l'automazione nei test riduce sicuramente gli sforzi da parte degli sviluppatori e per questo principale motivo si ritrova ad essere la più popolare tecnica, al contrario della creazione di testing manuale. In [9, Alegroth et al.] viene indicato come le pratiche manuali portino con sé numerosi problemi, essendo inoltre inclini ad errori e a sprechi di risorse in termini di tempo con costi oltre il 40% del costo totale per lo sviluppo. Per cui viene presentato dagli autori *JAutomate*, uno strumento di visual GUI testing che colma il gap delle tecniche manuali combinando il riconoscimento delle immagini con le funzionalità di *Record & Replay*. In [10, Dukes et al.] viene presentato un caso studio sul security testing delle applicazioni web e la differenza usando metodi manuali o tools. Qui, al contrario di quanto detto finora, le tecniche manuali mostrano una maggiore applicabilità, poiché alcune vulnerabilità possono solo essere trovate con attente osservazioni da parte del tester. Ad esempio, usando il testing manuale sono state trovate vulnerabilità riguardanti l'autenticazione, il controllo degli accessi e cross-site scripting che solo in parte sono state riconosciute dai tool automatici. Non ne sono state riconosciute e riportate alcune come

- *Forgot Password feature*, la funzionalità non era implementata nel sito web per cui gli utenti non potevano accedere a essa nel caso in cui avessero dimenticato le credenziali d'accesso
- l'applicazione non presentava una lockout policy per i tentativi di login falliti, il che porta ad un elevato rischio di attacchi brute force
- ecc...

Cercando di fare un confronto generale sui criteri per effettuare i test sulle applicazioni web, [13, Nikfard, 2014] mostra una esauriente tabella 2.2 (qui mostrata in parte) in cui vengono classificati tali criteri a seconda della loro

- *prestazione*, basata sul tempo di esecuzione
- *affidabilità*, basata sull'uso di standard
- *precisione*, basata sull'uso di tecniche logico matematiche

Allo stato dell'arte attuale troviamo vari riferimenti, utili ad illustrare il web testing, anche in [12, Leotta et al.], dove viene presentata un'analisi di costi e benefici di due diverse categorie di approcci automatici al test di applicazioni web: *Capture & Replay*, usando in particolare *Selenium* IDE; web testing programmabile, usando *Selenium* Web Driver (vedi paragrafo successivo). I test *Capture & Replay* (C&R) sono basati sull'assunzione che l'attività di testing sull'applicazione può

Criterio
Generare casi di test da requisiti funzionali
Generare casi di test modellando interazioni con il database
Testing basato sull'interazione nel browser

Tabella 2.2: Elenco approcci di web application testing.

essere automatizzata registrando le azioni eseguite dal tester sulla GUI del sito e generando uno script che ne fornisca una riesecuzione automatica. D'altro canto, i web testing programmabili cercano di unificare il web testing con il testing tradizionale, supportando l'interazione con la pagina web e i suoi elementi (*widgets*) così che, ad esempio, si possa automaticamente inserire del testo in dei form o cliccare su pulsanti e link. I casi di test C&R sono facilmente implementabili, non essendo richiesta alcuna particolare abilità avanzata nel testing in generale né di programmazione: i tester utilizzano solamente la web application come un SUT mentre le loro azioni vengono registrate. Questo porta però ad un chiaro svantaggio, chiamato *GUI test fragility*: se vengono modificate, anche solo in parte, sezioni della GUI dopo che sono state registrate le azioni, queste potrebbero non funzionare più correttamente se non totalmente alla loro riesecuzione. Al contrario, i web test programmabili richiedono un minimo di abilità nel programmare per potere creare adeguati casi di test, per cui, deve essere scritto un insieme di istruzioni e comandi da essere eseguiti e che l'applicazione SUT debba eseguire come atteso. Rispetto all'altra modalità, gli script così preparati sono più robusti e affidabili se si presentano cambiamenti nella GUI del sito web.

2.1.1 Selenium

Uno strumento utile, ai fini del testing in applicazioni web, è [1, **Selenium**], tool open sorce per la gestione automatizzata dei browser, utilizzato come framework di testing per l'automazione di test per applicazioni web. Permette di svolgere operazioni di test quali *acceptance, functional, performance, load, stress*. Si mostrano di seguite alcune caratteristiche tra due varianti di applicativo offerte da Selenium (vedi paragrafo precedente per differenze).

Attraverso Selenium *WebDriver* è possibile interagire con l'applicazione web, in particolare con la GUI, grazie alle API disponibili in vari linguaggi tra cui Java e Python. I casi di test vengono quindi implementati manualmente nel linguaggio scelto e possono essere integrati con le asserzioni di *JUnit*. Una pratica comune è quella di usare il driver applicando il pattern del *page object*, usato per creare un modello della pagina web, il *POM*, così che le funzioni offerte dalla pagina

diventino dei metodi espliciti, facili da richiamare al momento del testing e quindi permettendo allo sviluppatore di lavorare ad un più alto livello di astrazione. Una pagina web con una certa modularità, ben organizzata, implementata attraverso elementi riutilizzabili, così come intere pagine all'interno di tutta l'applicazione, è un fattore che influenza il testing programmabile come quello che permette il WebDriver di Selenium [12, Leotta et al.]. I benefici sono massimizzati se adottati nell fasi iniziali dello sviluppo del software.

Attraverso Selenium *IDE* è possibile scrivere i casi di test, non è solamente possibile applicare la C&R: come suggerisce il nome è un IDE completo (Integrated Development Environment). In particolare

- permette ai tester di registrare, modificare e fare debug dei casi di test, usando *Selenese*, linguaggio composto da triple, ognuna formata da ***(command, target, value)***
- supporta selezione campi *smart*, usando IDs, nomi, xPath
- può salvare i casi di test come codice HTML, Java o altri formati
- suggerisce le varie asserzioni

2.1.2 Scout

Come brevemente detto in precedenza, **Scout** [31, Nass et al. 2020] è un prototipo di tool che offre funzionalità di GUI testing implementando i concetti dell'AT (vedi figura 2.2). All'inizio della fase di test, l'utente ha la possibilità di scegliere il sito web di partenza (*Home Locator*) e il browser di riferimento, che viene utilizzato come motore dietro l'*Augmented Layer*. Tutte le azioni eseguite dal tester, come click del mouse o input dalla tastiera, vengono registrate, attraverso l'uso di un modello a grafo orientato. Le azioni eseguite in sessioni di esecuzione precedente vengono mostrate con riquadri che fanno da contorno al *widget* relativo. In particolare, è possibile, tra le varie azioni, associare una *issue*, ovvero quello che secondo il tester non va come dovrebbe in un determinato widget e viene quindi rilevato come un bug nella pagina web. O ancora, è possibile inserire dei *check*, azione che identifica un widget che ha bisogno di essere verificato successivamente.

Il modello permette quindi di conservare tutti gli stati dell'applicazione (tutte le dipendenze di memoria, dati memorizzati) che contengono le azioni eseguite dal tester, attraverso lo *State Controller*, che permette la corretta sincronizzazione tra la GUI del SUT e le informazioni "aumentate" prodotte.

Scout ha delle similitudini con quanto visto in precedenza nei sistemi di C&R come in *Selenium IDE*, anche se differisce in parecchi punti: Scout registra le azioni direttamente quando vengono eseguite dal tester, prima di essere trasferite alla GUI, così che sia in grado di ripetere il caso di test esattamente nel modo in

cui è stato creato. Inoltre, alcuni risultati del workshop in [31, Nass et al. 2020] mostrano che creare casi di test usando Scout è, in media, l'11% più veloce rispetto al farlo con Selenium: è pur vero che, in generale, il GUI testing con tecniche C&R è più semplice per lo sviluppatore poiché permette di generare una test suite in un tempo minore rispetto allo scrivere test case con dei tool layout-based.

Il prototipo di Scout, oltre ai suoi vantaggi nella fase di testing, presenta comunque alcune limitazioni (verranno descritte nel capitolo successivo quelle relative allo sviluppo del tool) come mostrato in [29, Nass et al. 2019], per elencarne alcune:

- Collaborazione tra tester e sviluppatore: non è previsto un canale di comunicazione diretta integrato tra le parti durante la fase di test;
- Scroll del mouse non funzionante: ciò che viene visualizzato sulla GUI è uno screenshot preso ogni circa 1 secondo dal webDriver, lo scorrimento è disattivato e la rotella del mouse serve per scorrere tra i widget nell'Augmented Layer;
- Feedback dopo click del mouse: è facile fare click nel widget sbagliato, l'utente vorrebbe capire se ha interagito col widget giusto nella pagina;
- Headless mode, ovvero se il browser di supporto deve essere reso nascosto o meno;
- Computazionalmente costoso e pesante;
- Rilevare automaticamente i bug di UI comuni, come dimensione testo piccola o widget non allineati: confronto con gli standard UI;
- Non in grado di gestire le animazioni: essendo una sequenza di screenshots presa dal driver collegato al browser, le possibili animazioni presenti vengono mostrate con numero di frame pari a 1 per ogni circa secondo o quando viene visualizzato la schermata nella GUI

2.2 Mutation Testing

L'idea di base dietro al concetto di *Mutation Testing* è quella di creare casi di test, per il software o applicazione web, sul codice che presenti bug e difetti inseriti artificialmente, così che ne si possa verificare la robustezza nella fase di testing, in particolare poiché, se queste mutazioni al codice non vengono rilevate come si dovrebbe, allora i test vengono considerati non affidabili.

Questi difetti artificiali sono chiamati ***mutants***, ognuno dei quali può riguardare le varie parti del software in modo da evidenziare, ad esempio, errori di programmazione comuni, condizioni errate e, per quanto riguarda la GUI, se presente,

errori generali nell'interfaccia con cui l'utente interagisce. In [30, Papadakis et al.] vengono evidenziati metodi e consigli da seguire per usare il mutation testing, riassunti, in parte, come segue:

- *selezione del mutant*, bisogna spiegare l'appropriata scelta dei mutants rispetto al linguaggio di programmazione usato: è necessario definire le trasformazioni sintattiche che introducono al mutant dal momento che dipende fortemente dall'analisi del linguaggio in cui il software è scritto
- *ridondanza del mutant*, bisogna giustificare l'eventuale presenza ridondante di più mutants, poiché possono avere un impatto non trascurabile nella validità del mutation test: l'idea è che i mutants semanticamente equivalenti e quelli diferrenti ma ridondanti possano essere essere identificati e rimossi per ridurre lo sforzo richiesto ad eseguire il test e migliorarne la precisione
- *presentazione dei risultati*, bisogna chiarire i risultati empirici ottenuti dai test, così da assegnare il giusto peso alle mutazioni rilevate

In [35, Thomas Hamilton] viene mostrato come il mutation testing sia un valido approccio ma presenti un elevato costo nel generare i mutants ed eseguirli per tutti i casi di test. Riassumendo, i vantaggi sono i seguenti:

- Porta lo sviluppatore ad un buon livello di rilevamento degli errori: può essere coperta gran parte della struttura del codice così rilevare le parti non testate adeguatamente;
- Permette di scoprire le ambiguità nel codice sorgente grazie ad eventuali analogie con le mutazioni;
- I difetti artificiali permettono di identificare buchi nella test suite e quindi forniscono un ottimo e concreto suggerimento su come effettuare test aggiuntivi;
- Può far risparmiare tempo nel debugging e quindi migliorarlo se nessun caso di test creato in precedenza copre la logica che il mutant prevede;
- La qualità del codice del software migliora: gli utenti finali possono affidarsi ad un sistema più robusto.

Al contrario, gli svantaggi sono:

- Costo elevato, in termini di tempo, per generare tutti i possibili mutants;
- Mutazioni complesse sono difficile da implementare;
- Dato il costo, è sbagliato affermare che non sia necessario un tool di generazione automatica;

- Non applicabile per il *Black Box testing* essendo necessaria la modifica del codice sorgente.

In generale, grazie all'inserimento dei mutants, lo sviluppatore ha la possibilità di migliorare la qualità sia della test suite, poiché ne si espongono le eventuali debolezze, sia del software stesso, anche se le mutazioni comportano uno sforzo maggiore nella loro creazione.

2.2.1 Visual GUI mutants

A differenza dell'inserimento di mutants nel codice di basso livello, come detto in precedenza, il mutation testing riguardante la GUI del software agisce direttamente sugli elementi che ne compongono l'interfaccia (*widgets*) e i loro parametri, le loro proprietà. In [16, Alegroth et al.] e [18, Oliveira et al.] viene presentata un'analisi sulla classificazione dei possibili mutants a livello dell GUI, si veda tabella 2.3 (verranno prese in esame solo alcune di queste per lo sviluppo del tool, vedi capitolo 3). Questi studi cercano di capire se sia possibile generare automaticamente dei mutants sulla GUI dei software e come quelli proposti in tabella 2.3 possano effettivamente essere usati in applicazioni reali. Partendo da sperimentazioni sulla libreria Java *Swing* all'interno di un applicazione che emuli una calcolatrice e quindi partendo dal codice sorgente, gli autori [18, Oliveira et al.] sono riusciti a concludere che l'utilizzo dei mutants sulla GUI porta il principale beneficio di poter testare direttamente al livello più alto il SUT e, nel caso di tool automatici, di risparmiare nei costi, in termini di tempo, di dover scrivere manualmente i mutants.

2.2.2 Mutants nelle applicazioni web

In [7, Praphamontripong, 2012], l'autore cerca di analizzare in modo sistematico il mutation testing nelle applicazioni web, partendo dal presupposto che le tradizionali tecniche per il software testing sono inadeguate per le applicazioni web a causa della natura dinamica ed in evoluzione delle relative tecnologie. Questa tecnica dell'inserimento di mutazioni può quindi essere usata per identificare più problemi, definiti dall'autore, di interazione, che riguardano la comunicazione tra le componenti web. Ad esempio, alcuni mutants che si possono inserire nel codice HTML possono essere:

- *sostituzione link*, all'interno dei tag `<a>` viene cambiata la destinazione di `href`
- *rimozione link*, all'interno dei tag `<a>` viene rimosso il link di destinazione
- *form field nascosto*, all'interno dei tag `<input>` viene inpostato per esempio **`null`**

#	Mutant
1	Rimozione completa widget esistente
2	Rendere widget invisibile
3	Rimozione listener esistente
4	Aggiunzione widget identico
5	Aggiunzione widget simile
6	Aggiunzione widget differente
7	Aggiunzione listener
8	Modifica dimensione delle finestre così che i widgets si adattino automaticamente
9	Modifica dimensione delle finestre così che i widgets NON si adattino automaticamente
10	Ridurre dimensione delle finestre per nascondere i widgets
11	Modifica posizione widget, specificata
12	Modifica posizione widget, verso gli angoli del layout
13	Modifica posizione widget per renderlo sovrapposto ad un altro
14	Modifica dimensione widget
15	Modifica aspetto widget
16	Modifica tipo di widget (ad es. pulsante cambiato in campo di testo in input)
17	Modifica libreria GUI usata per i widgets
18	Modifica dimensione delle finestre così che i widgets possano adattarsi

Tabella 2.3: Elenco GUI mutants.

In [39, Yandrapally et al.] viene proposto un framework per l'analisi delle mutazioni della GUI nelle applicazioni web, che agisce sul DOM per superare le limitazioni delle tecniche di mutazione basate sulla modifica del codice sorgente. Per cui, il tool necessita soltanto della URL del sito da analizzare, dato che agisce sull'output prelevato dalla risposta HTTP. Questo porta un primo evidente svantaggio: durante la navigazione o il refresh della pagina, la mutazione applicata svanisce, rendendo così vana la modifica effettuata, ne consegue che il test può essere effettuato solo in quelle precise condizioni. Per quanto riguarda i tipi di mutant utilizzati, ne vengono usati 16, molto simili a quelli in tabella 2.3, riguardanti però nello specifico gli elementi web del DOM: si veda la tabella 2.4, dove vengono elencate le mutazioni basate sull'impatto dei bugs nella UI, raggruppate in 5 categorie riguardanti la proprietà dell'elemento da modificare. Ad esempio, si può andare a modificare l'attributo della classe CSS oppure l'ID per fare variare di conseguenza anche l'aspetto dell'elemento. Malgrado l'efficacia del visual GUI

Type	Mutant Name
Attribute	AttributeAdd
	AttributeDelete
	AttributeModify
Event Handler	EventHandlerAdd
	EventHandlerDelete
	EventHandlerModify
Tree	TreeInsert
	TreeDelete
	TreeMove
Content	ContentInsert
	ContentDelete
	ContentModify
Style	StyleVisibility
	StyleColor
	StylePosition
	StyleSize

Tabella 2.4: Elenco Web mutants e relativi tipi.

mutation testing descritta prima, gli autori fanno notare che comunque è possibile che il mutant, talvolta, non abbia un impatto visivo sostanziale, se non addirittura nullo: per esempio, cambiare la posizione di un widget, già invisibile, non ha alcun effetto reale ai fini del testing. Analizzando le proprietà degli elementi del DOM, è possibile andare ad individuare dove applicare la modifica per rendere visibili i mutants, ad esempio, per quanto riguarda gli attributi è possibile cambiare la sorgente **src** di un tag `<img>` per cambiare l'immagine visualizzata o ancora, se si agisce su **style** è possibile modificare le proprietà *CSS* come dimensione, posizione e colore del widget.

In generale, attraverso l'uso di tool, come Selenium (vedi 2.1.1) o Scout (vedi 2.1.2), che permettono l'interazione con gli elementi della pagina di un'applicazione web, è possibile creare mutants, analizzando come il loro inserimento possa influenzare l'interazione del tester e migliorare la fase stessa di testing (vedi capitolo 3).

2.3 Gamification

In una pubblicazione di una conferenza [5, Deterding et al.], gli autori qui citati hanno definito a modo cosa sia la Gamification: rendere ludica una situazione

ed un contesto dove non era previsto l'inserimento di elementi specifici del gioco e dell'intrattenimento, con la finalità di rendere più coinvolti gli attori che vi partecipano. Vi sono vari metodi applicativi della gamification nella vita di tutti i giorni, la maggior parte dovuti al fatto che la recente (negli ultimi venti anni) evoluzione tecnologica e, soprattutto, informatica, stia aiutando non poco il processo. Si parte quindi dal rendere più divertenti contesti come quelli formativi (scuola, università) o lavorativi, dove non si è soliti avere come obiettivo il divertimento ludico [33, Fulcini]. Tanto ad esempio da poter assistere ad una lezione scolastica così coinvolgente che gli studenti si divertono a creare una sana competizione tra loro per arrivare ad essere i primi a rispondere alle domande poste dal docente [8, Di cosa parliamo quando parliamo di Gamification]. Alcuni vantaggi dell'applicazione di elementi ludici possono essere così riassunti:

- Perseguimento dell'obiettivo, rispettando le regole
- Maggiore motivazione
- Socialità e collaborazione
- Problem solving
- Interattività e proattività
- Maggiore coninvolgimento e partecipazione

Altro esempio sono alcuni svariati software applicativi, web e non, che grazie all'inserimento di elementi di gamification rendono l'utente non solo più partecipe ma anche più propenso ad utilizzare il servizio che il software offre (user engagement) tramite i vari sistemi di ricompense (rewarding) strategicamente collegati.

Spostando il focus nello specifico sui processi interni che vengono utilizzati per applicare la gamification, possiamo analizzare gli elementi costitutivi classici del gioco, come nella figura 2.3. In pratica, si evince che implementando una qualsiasi combinazione di queste tecniche, si può raggiungere l'obbiettivo di quel coinvolgimento voluto. A ciò deve essere aggiunta un'altra analisi: l'età del target, del "giocatore" (vedi capitolo 4, dove vengono selezionati target con età inferiore a 50). Questo perché la percezione e la comprensione di alcune dinamiche legate al mondo tech del gaming viene data più per spontanea nelle nuove generazioni che in quelle precedenti.

In [28, Chou, 2019] viene descritto un framework che organizza gli elementi della gamification e la loro influenza a livello cognitivo da parte della persona coinvolta. **Octalysis**, vedi figura 2.4, è composto in 8 punti cardine, come si evince dal nome, che rappresentano i fattori prima citati. Di seguito si riassumono brevemente:

1. *Epic Meaning & Calling*, è il bisogno di prendere parte a qualcosa di più grande di sé ed essere convinti nel farlo

Figura 2.3: Elementi di gioco.

2. *Development & Accomplishment*, è ciò che riguarda la motivazione nel migliorarsi, acquisendo abilità e raggiungendo traguardi

3. *Empowerment of Creativity & Feedback*, quando il soggetto è coinvolto in un processo creativo prova diverse strategie cercando riscontro nelle sue azioni

4. *Ownership & Possession*, riguarda la motivazione della persona nell'accumulare cose concrete, difendendone il possesso e migliorandosi per ottenerne di più

5. *Social Influence & Relatedness*, qui ci si riferisce alle attività motivate dall'influenza delle altre persone sul soggetto, come accettazione sociale e invidia

6. *Scarcity & Impatience*, è ciò che motiva la persona a voler avere, a desiderare qualcosa che non può avere poiché rara o esclusiva e unica

7. *Unpredictability & Curiosity*, riguarda la volontà di scoprire qualcosa di sconosciuto e impredicibile, coinvolgendo il caso poiché non è dato sapere cosa succederà, come nei giochi d'azzardo o lotterie

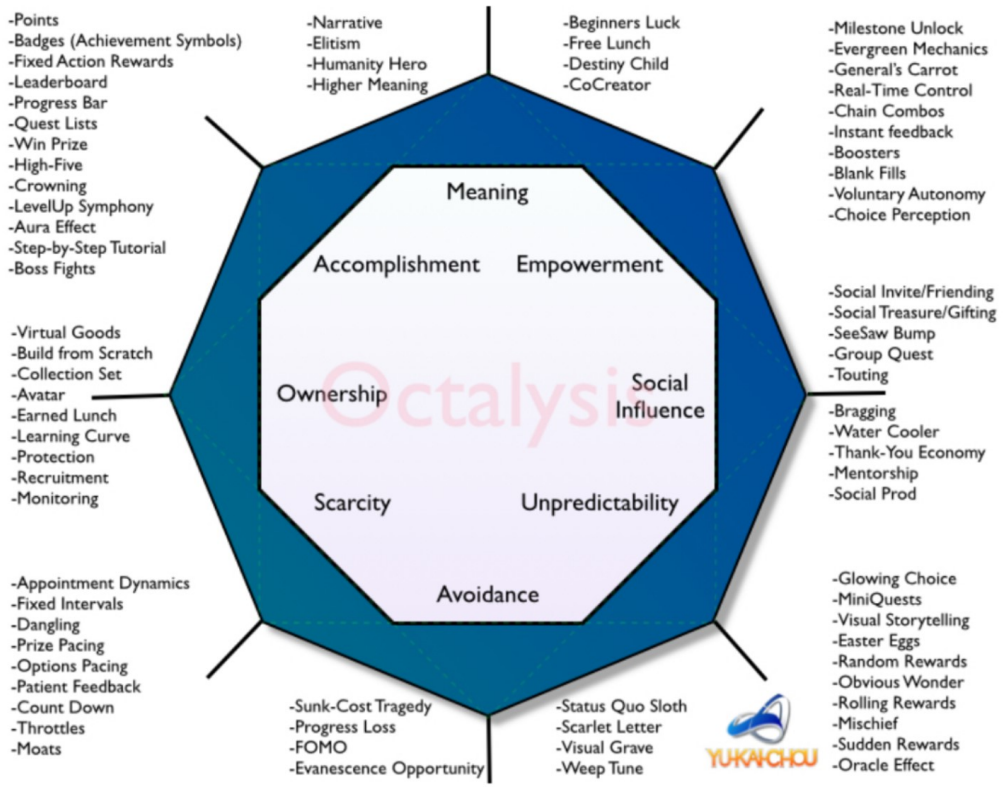


Figura 2.4: Schema Octalysis.

8. *Loss & Avoidance*, ci si riferisce a situazioni, talvolta negative, che la persona coinvolta non vuole che accadano, come il perdere progressi e traguardi raggiunti

9. *Sensation*, nonostante fossero 8, questo punto cardine viene considerato **nascosto** poiché riguarda il piacere personale, fisico ed emotivo che la persona ottiene nel compiere un'azione e può coinvolgere tutti e 5 i sensi, come scegliere un pasto piuttosto che un altro

Il framework è organizzato in modo da presentare sul lato destro il cosiddetto "cervello destro", dove troviamo ciò che è relativo alla creatività, all'espressività personale e agli aspetti e dinamiche sociali. Sul lato sinistro, il "cervello sinistro", troviamo ciò che riguarda la logica, il pensiero analitico e il possesso degli obbiettivi raggiunti. Se consideriamo invece la parte superiore, detta *White Hat*, troviamo le motivazioni considerate positive, ad esempio, qualcosa che valorizza la propria creatività e riesce a far raggiungere le abilità e gli obbiettivi voluti. Dal lato opposto, nella parte inferiore, detta *Black Hat*, abbiamo le motivazioni considerate

negative, ad esempio, qualcosa basato sull'incertezza negli obbiettivi da raggiungere e la paura di perdere ciò che si ottenuto.

2.3.1 Gamification nella disciplina del Software Engineering

Restando nell'ambito del software, bisogna evidenziare l'importanza della gamification e la sua applicazione nei processi di Software Engineering (SE). Secondo gli autori [15, Pedreira et al.], bisogna considerare tutti i processi descritti nello standard ISO 12207 per capire se e quali fanno uso di elementi di gamification e più precisamente se lo scopo di averla inserita sia quello di accrescere la motivazione all'utilizzo da parte dei partecipanti, senza escludere il fatto che rendere ludica una di queste attività possa contribuire e non poco agli obiettivi delle altre, interne allo sviluppo del software stesso. Per elementi di gamification si intendono ad esempio

- *badges*, rappresentano alcuni traguardi (achievements) dell'utente
- *rankings*, classifica con i migliori partecipanti ceh viene presentata a tutti per creare competitività. La posizione può essere calcolata tramite i punti o il livello ad esempio
- *livelli*, all'utente viene attribuito un livello basato su un sistema di attribuzione punti, ottenuti da ricompense per aver completato particolari task
- *quests*, sono i task che l'utente deve completare, sotto forma di domanda con elementi di gioco che possono richiamare a delle storie per renderlo più attrattivo
- *voting*, i partecipanti possono votare gli altri, o meglio, i loro comportamenti. Si trasformano in punti acquisiti
- *betting*, l'utente può scommettere su un certo evento e guadagnare o meno punti e ricompense

Volendo fare invece alcuni esempi di software, in cui possiamo trovare tali elementi inseriti, ne troviamo diversi. Per richiamare l'attenzione è giusto citare [Dencheva et al.] che ha proposto di migliorare la partecipazione ad una Wiki corporate inserendo elementi per influenzare la reputazione dei partecipanti. O ancora, [Grant and Betts] che analizza come la gamification abbia un impatto significativo nel problem solving collaborativo, con particolare focus su [4, StackOverflow].

Merita attenzione anche una recente pubblicazione di [37, Moldon et al.] su come il comportamento degli sviluppatori software possa cambiare in risposta alla modifica degli elementi di gamification all'interno di [2, GitHub], una tra le più comuni piattaforme per la programmazione e sviluppo collaborativa. In particolare, è stato notato che la sola rimozione (non annunciata), dalla GUI della pagina del

profilo, del contatore delle attività giornaliere (lo possiamo definire una sorta di A/B testing) è conseguita in cambio significativo nel comportamento dell'utente. Su un campione di sviluppatori che avrebbero dovuto contribuito per 100 giorni consecutivi, è stato notato che alcuni hanno abbandato il *quest* per rimanere in linea con il contatore che non gli veniva più mostrato.

In [42, Stol et al.] si analizza come una piattaforma di gamification applicata nelle aziende di software porti gli sviluppatori a prenderne parte in modo più agevole e orientato alla soddisfazione del proprio lavoro, il che è un punto fondamentale per la crescita aziendale. Ciò si basa sulla teoria di auto-determinazione (SDT), che indica come le persone abbiano certi bisogni psicologici che devono essere soddisfatti tramite la motivazione, l'autonomia e la competenza di se stessi.

Lo scopo, quindi, della gamification, nel mondo degli sviluppatori, non è solo migliorarne la partecipazione ma anche la fiducia e la collaborazione tra gli utenti, grazie al sistema di *rewarding* che ne evidenzia gli obiettivi raggiunti e la carriera passata.

Altro esempio lo possiamo trovare nelle applicazioni recenti del mondo dello sport a supporto del tifoso basate sulla *blockchain*, come [27, Socios], che inserendo gli elementi della gamification ha provveduto ad aumentare lo *user engagement*, fattore rilevante se si vuole aumentare la *demand side network*. In breve vengono utilizzate combinazioni di *badges*, *rankings*, *livelli*, *quests* e *betting* opportunamente distribuite all'interno del software. Ad esempio, i livelli servono semplicemente ad indicare l'esperienza dell'utente attraverso i relativi punti acquisiti in cambio di particolari azioni effettuate come lasciare voti ai sondaggi proposti dalle squadre o dalla piattaforma o scommettere sul risultato delle partite. O ancora, le ricompense sono considerate tangibili poiché, oltre a ritrovarsi con più punti degli altri utenti, corrispondono a premi reali, riscattabili all'esterno, ad esempio biglietti per assistere alle partite della propria squadra o esperienze vip all'interno degli stadi o centri di allenamento della società.

2.3.2 Gamification nell'ambito del Software Testing

Come già analizzato nel paragrafo 2.1, una delle fasi più dispendiose dello sviluppo di un software in termini di tempo e di risorse è quella di testing. Nonostante bisogna avere una test suite completa ed esaustiva per poter riconoscere e prevenire le problematiche derivanti dai vari bug sparsi nel codice, spesso il software viene testato in maniera blanda e sicuramente non esaustiva, solo ai fini di rientrare nelle scadenze imposte, trascorrendo maggior tempo nella fase di scrittura del codice. In aggiunta la fase di testing è considerata spesso noiosa e ripetitiva e non in grado di stimolare l'attenzione del tester. Per rendere più piacevole tale disciplina sono stati fatti vari tentativi, tra cui l'applicazione della Gamification e dei suoi elementi visti in precedenza [22, Fraser et al.]: questi possono aiutare a superare

le mancanze didattiche ed educative che si hanno con il testing, essendo spesso trascurato, inoltre agevola i tester ad utilizzare strumenti di test avanzato che non avrebbero considerato altrimenti. In [26, de Jesus et al.] vengono classificati e categorizzati gli ambiti di applicazione della gamification nella fase di testing del software, in breve mostrati in tabella 2.5

Categoria	Descrizione	Esempi
APPLICATION CONTEXT	Contesti in cui approcci gamificati possono essere applicati per il software testing	Educativo, Industriale
USED GAME ELEMENTS	Elementi di gioco concreti inseriti in un ambiente gamificato	Achievement, Avatar, Badge, Duel, Leaderboard, Level, Points, Quest, Social Graph, Team, Virtual good
GAMIFICATION GOALS	Risultati attesi dall'uso della gamification	Crescita consapevolezza, Sviluppo creatività, Facilita processo di fixing, Incoraggia abitudini al test, Aumenta engagement, Migliora skills, Accresce motivazione, Stimola collaborazione
TESTING TECHNIQUES	Approcci che si basano su diversi artefatti software per derivare i requisiti di test	Testing funzionale, Testing strutturale, Testing basato sui fallimenti
TESTING PROCESS PHASES	Fasi incluse nei tipici modelli di processi di testing	Pianificazione, Configurazione dati e ambiente, Progettazione casi di test, Esecuzione e valutazione, Monitoraggio e controllo, Manutenzione
TESTING LEVELS	Tenere conto della granularità della porzione di software che è sottoposta al testing	Unit testing, Integration testing, System testing

Tabella 2.5: Classificazione contesti gamification.

Dunque, grazie alla gamification, si punta a migliorare la qualità e la pratica dei test per infine migliorare la qualità del software. Di seguito vengono presentati alcuni esempi.

In [23, Garcìa et al.] viene proposto **GOAL** (Gamification On Application Lifecycle), un framework con lo scopo di guidare il processo che supporta l'introduzione della gammification in qualsiasi fase dello sviluppo software, quindi anche in quella di testing. Gli elementi scelti sono punti, livelli, badges, classifica, challenge e social graph. Lo Unit Testing è stato menzionato come l'unico utilizzato per applicarvi elementi ludici.

Un esempio di software, propriamente definito gioco, dove ci si focalizza sul testing, è [20, **Code Defenders**], il quale è basato sull'idea del *mutation testing* (vedi paragrafo 2.2). Al centro del gioco troviamo un programma sotto test in cui i giocatori prendono parte con ruoli di attaccante (figura 2.5), che cerca di

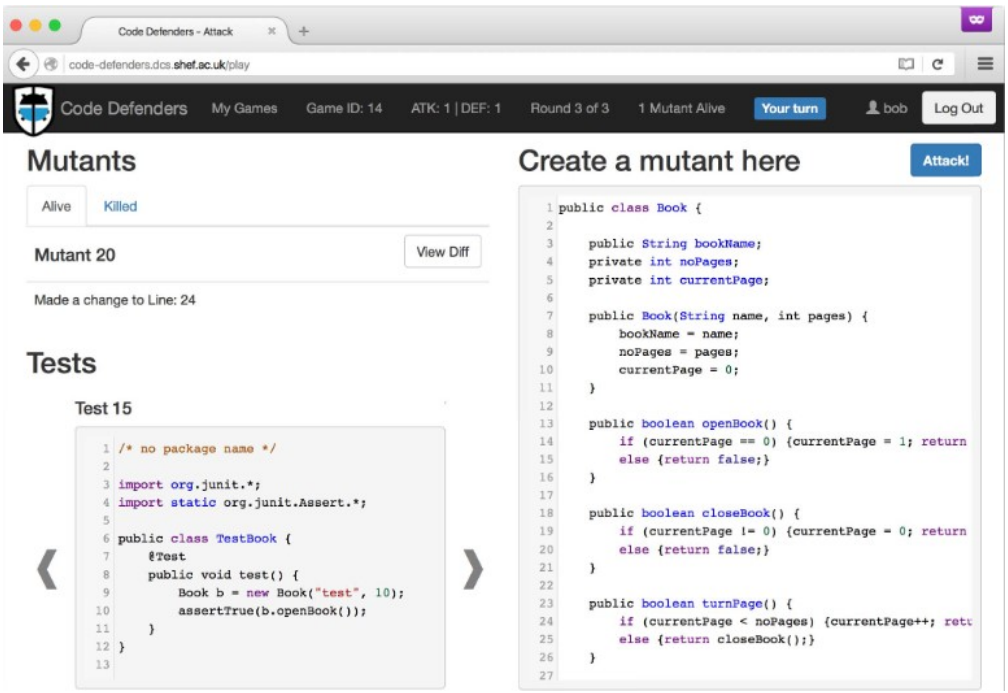


Figura 2.5: Vista dell'attaccante.

creare una versione mutante del codice del programma, e di difensore, che scrive i casi di test per il programma cercando di scoprire quanti più mutants possibili, che l'attaccante ha precedentemente inserito. Il sistema può essere usato come un framework di valutazione da parte degli educatori in ambito accademico ad esempio. Gli obiettivi di gamification comuni risultano essere la crescita della motivazione, di engagement e di divertimento nell'imparare il mutation testing. Chiaramente con conseguenze nel miglioramente delle skills da parte dei partecipanti a capire

e sviluppare i test adeguati. Gli elementi qui scelti sono punti, duelli e livelli, la classifica, ad esempio, è stata usato solo per alcune analisi. L'ipotesi degli autori era letteralmente "attraverso l'uso di un gioco sulla mutation testing, gli studenti saranno in grado di comprendere tutti i concetti rilevanti di questo tipo di testing mentre giocano e si divertono, e, alla fine, diventeranno sia migliori sviluppatori che tester, il che aiuta a produrre software di qualità superiore".

La piattaforma [25, **CodeSignal**, precedentemente CodeFights] è una delle alternative per il recruiting di sviluppatori, dove le aziende possono pagare per mettersi in contatto con i giocatori migliori. Il team del "gioco" assicura che i partecipanti possano sviluppare le loro abilità nel test design grazie al framework a disposizione, calibrato con dati di esempio robusti e cin il supporto di oltre 70 linguaggi di programmazione.

In [17, **Learn2Mine**], un ambiente di apprendimento basato su cloud, vengono forniti agli educatori gli strumenti per insegnare nei corsi di data science e agli studenti il sistema ludicizzato. Sebbene il software testing non fosse il focus della piattaforma e del corso, i problemi presentati comprendono concetti base di unit testing, per cui, gli studenti finiscono con l'imparare e applicare anche queste nozioni nel loro codice ai fini di guadagnare le relative ricompense.

In [19, **WEeSTT-CyLE**], acronimo per "*Web-Based Repository of Software Testing Tutorials - a Cyberlearning Environment*", viene usata la gamification come strategia per motivare gli studenti ad imparare le tecniche di software testing e migliorarne la conoscenza. Gli elementi ludici usati sono punti, badges e classifiche, oltre alla possibilità di giocare in team.

In [24, **Laurent et al.**] viene proposto un sistema per il crowdsourcing con la gamification inserita, allo scopo di etichettare mutants e valutare i parametri coinvolti nel processo (figura 2.6). In pratica, vengono mostrati solo i mutants che rispecchiano le abilità del giocatore, il sistema ludicizzato è focalizzato sulle attività di testing e nello specifico in quelle tecniche basate sui fallimenti. Viene indicato che il sistema può essere usato sia a scopi didattici che aziendali. Più in generale, sebbene il crowdsourcing testing sia una pratica ormai comune nelle aziende, il focus è principalmente orientato alle applicazioni web e mobile e non usa propriamente la gamification se non addirittura non presente.

La Gamification è stata analizzata in parecchi studi e analisi, come abbiamo potuto vedere negli esempi, per supportare il software testing, per lo più applicando elementi base come i punti, i livelli e le classifiche, e con lo scopo di migliorare motivazione e abilità nei tester senza lasciare intendere la vera tecinca di testing utilizzata. Come detto, il testing è essenziale per capire e assicurare se si stia creando un software di qualità, per cui inserire questi elementi ludici ne può solo aumentare l'efficacia.

È chiaro, talvolta, che bisogna analizzare alcuni limiti dell'applicabilità, non

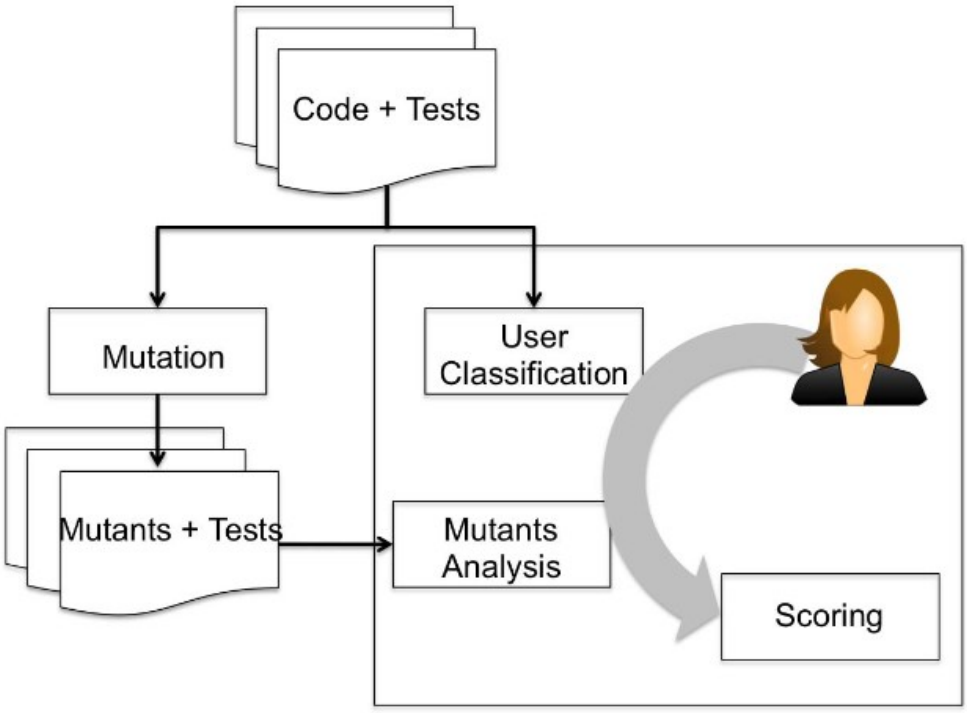


Figura 2.6: Schema del sistema di crowdsourcing in Laurent et al.

esistono solo vantaggi, come in molte situazioni: ad esempio, in [21, Dal Sasso et al.] viene specificata la *pointsification*, ovvero la ricerca da parte dell'utente di totalizzare più punti che può trascurando lo scopo primario, ovvero completare i tasks. Il giocatore finisce così per trascurare i compiti più difficili in favore di quelli più semplici da portare a termine. O ancora, per completare i tasks, il giocatore può voler "imbrogliare", possibilmente anche riuscendoci, così da ottenere una più elevata soddisfazione finale nella competizione con gli altri, vanificando magari lo scopo reale per il quale quel task era stato pensato. Per ultimo ma non meno importante, inserire la gamification nei processi di sviluppo di un software comporta un più alto costo sia in termini di tempo che di risorse economiche.

Tra gli studi citati non abbiamo incontrato, ad ora, alcuna ricerca che affrontasse l'utilizzo della *Gamification* nell'ambito del *Mutation Injection* del *Software Testing*. Il lavoro di questa tesi procede quindi in questa direzione, come vedremo nel prossimo capitolo, per provare ad unificare per la prima volta queste discipline.

Capitolo 3

Architettura del tool

Per raggiungere gli obiettivi preposti, si è cercato di implementare una soluzione adeguata ed il più vantaggiosa possibile considerato anche l'utilizzo della già presente architettura di *Scout* (vedi 2.1.2). Il tutto, come però vedremo, presenta alcuni limiti, ad esempio, riguardanti le prestazioni nell'utilizzo di Scout oppure il riconoscimento di alcuni widget durante e dopo l'applicazione della mutazione all'interno del tool (vedi 5.1).

L'idea di base è quella di utilizzare gli elementi del DOM, i *widgets*, come punto di riferimento per la creazione dei *mutants*. In particolare, attraverso la manipolazione del codice sorgente (HTML) della pagina web in esame. Il tool, qui sviluppato [41], utilizza [1, Selenium] (vedi 2.1.1) per facilitare la manipolazione del DOM della pagina web tramite browser, il che ha permesso la creazione e la modifica dei *widgets*, a partire da quelli già esistenti, rendendoli disponibili all'utente non appena vengono iniettati nel codice sorgente e non appena viene ricaricata la GUI.

La soluzione cerca di automatizzare quanto più possibile la creazione casuale di questi *mutants*. In generale, si cerca anche di perfezionare il modo in cui vengono inseriti e generati, per renderli il più possibile riconoscibili da parte del tester così da creare una più consapevole partecipazione alla sessione di test stessa. Il successo nell'aver riconosciuto queste mutazioni, verrà premiato, tramite il plugin della Gamification interno a Scout, e le statistiche relative gli verranno mostrate durante e al termine della sessione.

3.1 Framework di Mutant Injection e strumenti

Riprendendo quanto visto in precedenza (paragrafo 2.2.2), la *mutant injection* applicata qui riguarda la manipolazione della parte del DOM relativa ai widgets scelti. Modificando il sorgente HTML della pagina, attraverso opportune operazioni

Parametro	**Valore**
Depth	> 2
OffsetParent	Object, not NULL
Excluded Category	localName not in [*script, br, hr, style, label, img*]

Tabella 3.1: Filtri di ricerca per il widget vittima.

sulle stringhe, è possibile ottenere effetti visibili nella GUI, così da rendere tangibile a primo impatto il mutant che ne si origina, relativo a quel widget. Come vedremo, sono state utilizzate parte delle mutazioni in tabella 2.3 come punto di riferimento per l'applicabilità del tool. Le tipologie che non sono state considerate riguardano l'interazione con dipendenze esterne alla pagina HTML o con il codice *javascript* e quindi non manipolabile tramite le stringhe all'interno del sorgente, limitazioni che comunque verranno considerate per miglioramenti futuri del tool.

Per rendere più dinamico il processo di selezione del widget, il tool è stato automatizzato in modo da cercare, all'interno della pagina HTML, un widget che rispettasse i parametri seguenti, tabella 3.1, all'interno del DOM:

- **depth**, considerando i widgets come in un albero gerarchico che parta dall'elemento *body* come radice, viene scelta un profondità minima dal quale cercare l'elemento così da evitare di dover andare ad esaminare un elemento più esterno che possa andare ad occupare proporzioni significative della pagina poiché contenitore di elementi più piccoli, che, ad esempio, nel caso di mutants quali la rimozione del widget, comporterebbe la cancellazione di una quasi totalità degli elementi minori, che alla modifica del sorgente non saranno più disponibili

- **offsetParent**, all'interno del DOM gli elementi sono considerati come oggetti astratti, i quali contengono quindi vari attributi che permettono di identificarne le proprietà, in figura 3.1 ne vediamo alcuni i quali verranno usati nel tool e descritti nel seguito. In particolare, *offsetParent* permette di identificare, secondo la documentazione, se il widget relativo è visibile o meno nella GUI, così facendo si può evitare di applicare una mutazione a questi tipi di elementi poiché non ne si ricaverebbe alcun impatto visivo

- **element category**, secondo quanto riportato nella documentazione di [32, Mozilla], gli elementi del DOM sono suddivisibili in categorie, ciò è risultato utile, come vedremo più avanti, per la creazione di alcuni mutants. Per la ricerca del widget invece è stato reso necessario escludere alcuni di questi elementi, identificabili mediante *localName* (fig. 3.1), poiché non comporterebbero alcuna utile modifica visiva al sorgente della pagina ma talvolta ne

potrebbero causare la "rottura" e il mal funzionamento. I tag degli elementi che sono stati presi in considerazione sono *script, br, hr, style, label, img*

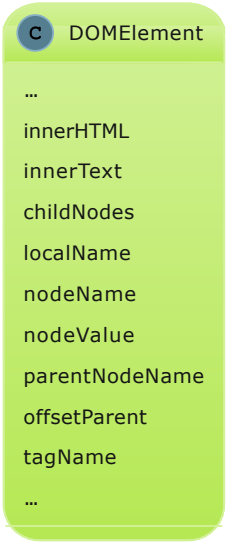


Figura 3.1: Classe astratta degli elementi del DOM con alcuni attributi.

Successivamente, nel paragrafo 3.2, vedremo nel dettaglio come i *widgets*, opportunamente selezionati come sopra, sono stati usati e manipolati per creare i *mutants*.

3.1.1 Funzionalità Selenium

Per interagire con il DOM, all'interno del browser in esecuzione, è stato usato Selenium (vedi 2.1.1), in particolare, le API del *WebDriver* (disponibili in vari linguaggi, scelte in *Java* per essere più facilmente compatibili e adattabili in *Scout*), per permettere, come detto, una maggiore facilità nella manipolazione della pagina web. Nello specifico, per il tool, sono risultati utili

- *pageSource*, che ha permesso di accedere al sorgente HTML della pagina dove cercare i widgets
- *executeScript*, che ha permesso di eseguire del codice Javascript, utile in primis per ottenere l'albero degli elementi della pagina e poi per iniettare nel browser il nuovo sorgente, dopo la modifica con il relativo mutant applicato, mediante `document.write()`, utilizzato in casi di testing come questo

Un evidente svantaggio nell'utilizzo del solo codice sorgente, ad esempio, è dovuto alla mancata possibilità di interazione con gli eventi dinamici, i *listeners*, del

Javascript presente all'interno di una applicazione web. Al contrario, viene permesso con una più relativa semplicità di gestire il sorgente anche offline, ovvero in modo indipendente dall'applicazione web in esecuzione, dato viene effettuata una pura manipolazione delle stringhe di testo all'interno del codice HTML.

3.1.2 Augmented testing

Attraverso l'uso di Scout (vedi paragrafo 2.1.2), il quale fa anch'esso uso di Selenium, è possibile interagire con il SUT mediante l'AT offerto dal sistema. Come possiamo vedere dallo schema in figura 3.2 (moduli già presenti non colorati, moduli modificati in arancione, moduli aggiunti in verde, vedi paragrafo 3.2), il prototipo modella al suo interno uno stato che mantiene per tutta l'esecuzione della sessione di testing, memorizzando parametri e informazioni riguardanti le azioni effettuate dal tester che poi devono essere replicate nella GUI della pagina web tramite il WebDriver di Selenium. Le azioni eseguite dal tester vengono trattate, nello specifico, nelle

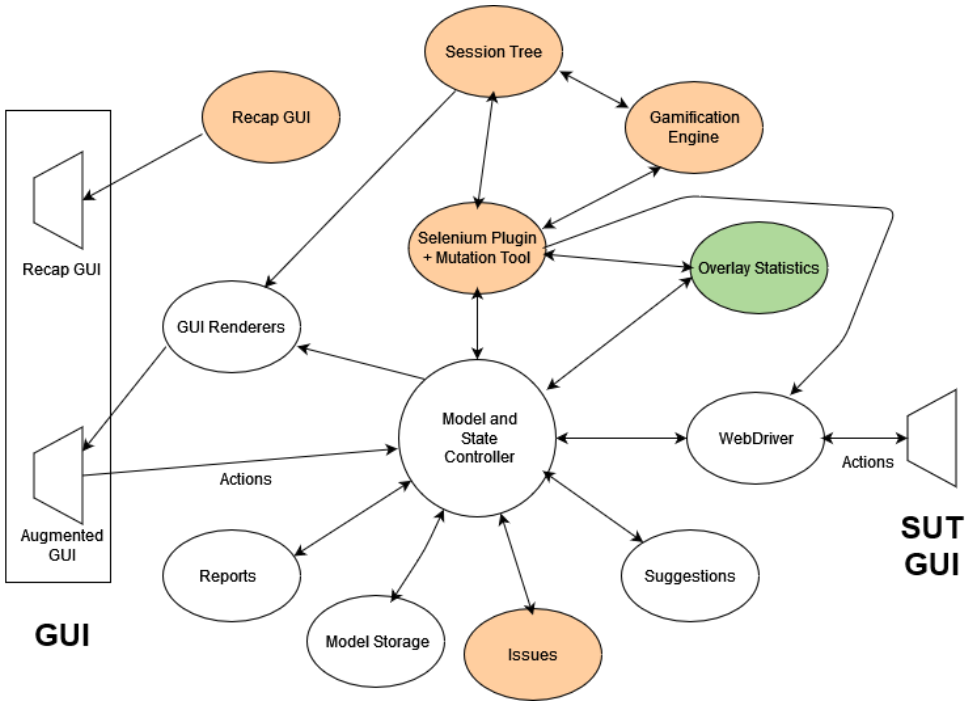


Figura 3.2: Scout overview.

seguenti tipologie:

- *check*, è stato richiesto un controllo del contenuto del widget
- *issue*, viene notato un comportamento diverso, come un bug, nel widget

- *click*, agisce come navigazione verso un'altra destinazione del sito web
- *type*, una casella di testo che necessita di essere riempita con l'input digitato dall'utente
- *select*, agisce come selettore di una scelta effettuata su una lista di tipo "drop-down"

La replicazione delle azioni si basa sui widgets con i quali si è avuto interazione e in Scout vengono generati a partire da uno script che, attraverso il driver, va a prelevare gli elementi del DOM nella pagina e ne va a salvare le informazioni nello stato corrente. Per essere mostrati in Scout, devono essere posizionati esattamente negli stessi punti e con le stesse dimensioni, dato l'Augmented Layer altro non è che uno screenshot della pagina mostrata, ricavato periodicamente grazie al WebDriver di Selenium, usato nel metodo `createCaptureAndReplaceWidgets()` di Scout.

Tra le informazioni salvate per ogni widget troviamo l'*xPath* (XML Path Language), un tipo di sintassi che fornisce una maniera flessibile per accedere alle diverse parti di un documento XML, HTML o SVG, tramite un percorso, come fosse un indirizzo URL, e navigare nella struttura gerarchica, in questo caso della pagina web, più facilmente. Come vedremo successivamente, è una proprietà risultata molto utile nello sviluppo del tool.

3.1.3 Sessione di testing in Scout

Il modulo *Session Tree*, visto in figura 3.2, rappresenta il modo in cui la sessione di testing è strutturata all'interno di Scout. La singola sessione è interpretata dalla classe **Session**, che modella una struttura ad albero non binario in cui ogni nodo è un altro oggetto, classe **Node**, che altro non è che la rappresentazione delle pagine web, oggetto **Page**. Nell'astrazione di più alto livello che viene interpretata da *Session* troviamo alcune informazioni utili per tenere traccia delle pagine web visitate e dei widgets con in quali si è avuto un'interazione in tutta la sessione. In *Page* troviamo gli attributi e i metodi utili per mantenere riferimenti ai widgets, alle interazioni e allo stato dello *State Controller* relativi alla solo pagina del nodo corrente, le quali vengono identificaate mediante un ID, corrispondente alla URL del sito. Per i dettagli, viene mostrato un class diagram in figura 3.3 che contiene anche le informazioni aggiunte e descritte successivamente nel paragrafo 3.2. È inoltre presente un meccanismo per tenere traccia del tempo trascorso all'interno di ogni pagina, tramite l'uso della classe **Timing**, utile ai fini dei calcoli dei punteggi e delle statistiche per la Gamification.

Per quanto riguarda quest'ultima, vengono utilizzate classi a supporto dell'elaborazione delle metriche basate sia sulla sessione in corso che su quelle precedenti, ovvero quelle delle sessioni precedenti rese disponibili in modo persistente alla fine

del test e utilizzabili come punto di riferimento per il tester/giocatore per accedere e visualizzare i risultati conseguiti nel tempo. Si utilizzano le classi **Stats**, **StatsComputer** e **GamificationUtils**. Anche qui si lascia ad uno schema dettagliato, figura 3.4, sul contenuto di esse, che contiene le modifiche per l'adattamento del tool dei mutants, che saranno descritte nel prossimo paragrafo.

3.2 Architettura

La fase di sviluppo è stata per lo più composta da due parti, una riguarda la creazione vera e propria del tool[1] a se stante (funzionante indipendentemente da Scout), l'altra riguardante l'inserimento di esso all'interno di Scout con l'integrazione di alcuni elementi di Gamification (vedi paragrafo 2.3) per agevolare l'esperienza di testing da parte dell'utente.

Per quanto concerne la prima fase, entrando nello specifico dell'architettura utilizzata per la creazione, il tool è stato sviluppato come fosse una API, così da potere essere versatile e facilmente utilizzabile all'interno di Scout ma anche indipendentemente da esso. In particolare, dato il prototipo di Scout è stato scritto in Java, si è preferito mantenere una maggiore compatibilità usufruendo sempre del JVM, macchina virtuale che esegue, interpreta e traduce il codice in quello nativo della macchina in esecuzione. Ciò è stato possibile anche attraverso Kotlin [6], evoluzione di Java, molto usato negli ultimi anni, ad esempio in ambienti di sviluppo Android o in quelli di backend. La scelta ha riguardato principalmente l'elevata versatilità, rispetto al Java, su alcune sue limitazioni, come le *extension functions*, che hanno capacità di estendere una classe con nuove funzionalità senza avere la possibilità di accedere al codice sorgente, le *espressioni lambda*, funzioni che vengono passate direttamente all'interno di un'espressione senza essere dichiarate, le *inline functions*, usate per compattare in una linea il corpo delle funzioni, o le *data classes*, per compattare l'esplicita dichiarazione dei getter e dei setter di una classe. Questi sono esempi di ciò che invece in Kotlin è disponibile. Dunque, il tutto tradotto in una minore quantità di *LOC*, linee di codice, che in fase di sviluppo portano sempre il loro vantaggio.

All'interno del tool [41] troviamo una classe principale ***SeleniumAPI*** delegata delle funzioni principali alla creazione del mutant all'interno di una pagina web grazie al supporto, come detto, del WebDriver di Selenium (per una visione d'insieme sulla struttura si veda la figura 3.5). Il metodo per eseguire tutte queste operazioni, come una sorta di wrapper, viene chiamato `scrapingWebsite(website: String)`, che utilizza l'istanza del WebDriver, ricevuta dal costruttore della classe, per interagire

[1]Per "tool", qui e nei riferimenti successivi, tranne dove diversamente specificato, ci si riferisce allo strumento di generazione automatica di mutants all'interno di un'applicazione web.

con il browser, e una stringa contenente la URL della pagina dell'applicazione web da modificare. In particolare, mediante il driver viene prelevato il sorgente HTML della pagina, *pageSource*, e viene salvato localmente in file per poter procedere alla successiva analisi dei widgets. Parallelamente al codice sorgente (non ci si riferisce qui ad operazioni asincrone, tutte i metodi invocati sono puramente sincroni), viene utilizzato un altro metodo del driver, *executeScript*, che ha permesso di ottenere, come detto nel paragrafo 3.1.1, l'albero dei widgets, rappresentazione del DOM della pagina, mediante una opportuna ricorsione. In particolare vengono prelevate solo le proprietà utili al tool, quali:

- ***attributes***, lista degli attributi che definiscono le caratteristiche di un widget come l'*id*, la *classe* o lo *stile* css
- ***localName*** o *tagName*, il nome del tipo di widget
- ***nodeName*** e ***nodeValue***, anche essi usati per identificare il tipo di widget, anche le sole stringhe di testo, o i caratteri di *newline*
- ***innerHTML***, il contenuto HTML interno al widget corrente
- ***parentNodeName***, il nome del tipo di widget dell'elemento padre a quello corrente
- ***offsetParent***, indica l'eventuale non visibilità del widget (valore `null`)
- ***xPath***, ottenuto mediante lo stesso metodo già presente ed utilizzato nel *SeleniumPlugin* di Scout, utile per localizzare gerarchicamente il widget
- ***depth***, ottenuto dalla ricorsione sui widgets, utile per escludere widget "poco profondi" nell'albero
- ***xPos***, ***yPos***, ***width*** e ***height***, ottenuti mediante un metodo disponibile per gli elementi del DOM, `getBoundingClientRect()`, e usati successivamente, nell'integrazione in Scout

Ottenuto l'albero dei widgets con gli attributi sopra elencati, viene parsificato sotto forma di array per una più rapida gestione nella ricerca del widget da scegliere come candidato all'applicazione del mutant, questo sempre grazie alle capacità di Kotlin nella manipolazione delle strutture dati.

Come detto sopra e nel paragrafo 3.1, la proprietà *depth* è necessaria per evitare di eliminare parti di albero significative che porterebbero con sé molti nodi figlio. Per scegliere opportunamente il valore limite è stata effettuata una breve analisi statistica sul comportamento, in particolare quantitativamente, degli elementi `<div>` all'interno del DOM di alcune pagine web, poiché spesso contenitori primari di tutti gli altri elementi. Si veda tabella 3.2 (ottenuta dallo script *depthStats.js*

Sito	tot. divs	# d1	avg. d1	# d2	avg. d2	# d3	avg. d3	# d (max)	avg. d
wikipedia.org	86	7	299	14	124	20	74	13 (6)	19
tesla.com	57	3	344	1	56	2	330	14 (12)	11
polito.it	143	2	634	1	1138	8	152	4 (14)	1
google.com	46	2	212	7	58	6	31	3 (9)	2
bgg	470	1	508	2	2197	4	125	50 (23)	11
twitter.com	50	1	168	1	167	1	166	3 (15)	1
stackoverflow	193	3	462	3	701	3	97	48 (11)	2
github.com	323	5	600	2	151	5	649	19 (18)	5
plantuml	84	5	97	7	67	15	29	1 (10)	0
microsoft.com	98	2	489	7	220	8	190	2 (13)	5

Tabella 3.2: Analisi profondità widgets nel DOM.

presente nel repo [41], branch *master*). In ordine le colonne rappresentano il numero totale dei widgets di tipo *div* presenti nella pagina, il numero di *div* al livello di profondità 1, la media di widgets interni (figli) che ognuno di questi possiede, ripetuto per i livelli 2, 3 e max, ovvero l'ultimo livello in cui è possibile trovare elementi *div*. Possiamo notare, per le 10 pagine analizzate, come un ridotto numero di contenitori *div* porti con sé un numero elevato di widgets interni per i primi due livelli e come dal terzo in poi (vengono qui omessi i livelli intermedi) fino all'ultimo livello gli elementi figli decrescano, segno indicativo del fatto che se viene scelto uno di quei *div* dei primi 2 livelli è molto probabile che il mutant ad esso applicato vada a modificare, o talvolta rimuovere, gran parte della GUI della pagina: perciò viene scelto il livello 3 (> 2) come limite inferiore da considerare nella selezione del widget.

Procedendo nella ricerca del widget candidato a diventare mutant, viene applicato un filtro anche per i tipi di elementi del DOM da escludere, dato non contengono informazioni visive utili ai fini del GUI testing o non ne permettono la corretta modifica, come il tag `<img>` che conterrebbe i metadati dell'immagine relativa e non facilmente modificabile con una semplice manipolazione delle stringhe nel codice HTML. I widgets che vengono esclusi sono i seguenti: *script, br, hr, style, label, img*.

Nel paragrafo 3.2.1 vengono descritti i tipi di mutants nella loro implementazione pratica, scelti a partire da un sottoinsieme di 10 mutazioni in tabella 2.3 per i motivi descritti nel paragrafo 3.1. La tabella 3.3 mostra le mutazioni implementate al widget selezionato, per meglio rendere, visivamente, nella fase di testing.

Al termine dell'applicazione del mutant al widget "estratto" viene usato nuovamente *executeScript* per iniettare all'interno del browser in esecuzione l'intero

#	Mutant
1	Rimozione completa
2	Invisibilità
3	Duplicazione
4	Aggiunzione simile
5	Aggiunzione differente
6	Cambio posizione
7	Cambio posizione per renderlo sovrapposto ad un altro
8	Ridimensionamento
9	Cambio aspetto (es. colore)
10	Cambio tipo di widget (ad es. pulsante cambiato in campo di testo in input)

Tabella 3.3: Tipologie GUI mutants implementati.

codice HTML della pagine, mediante `document.write()`. Attualmente il browser in esecuzione, gestito dal WebDriver di Selenium, è *Microsoft Edge* per alcune limitazioni, che verranno considerate nel paragrafo 5.1. Ad esempio è stata riscontrata l'impossibilità, a causa di regole di sicurezza impostate di default in altri browser come *Firefox*, di utilizzare il comando *document.write()*.

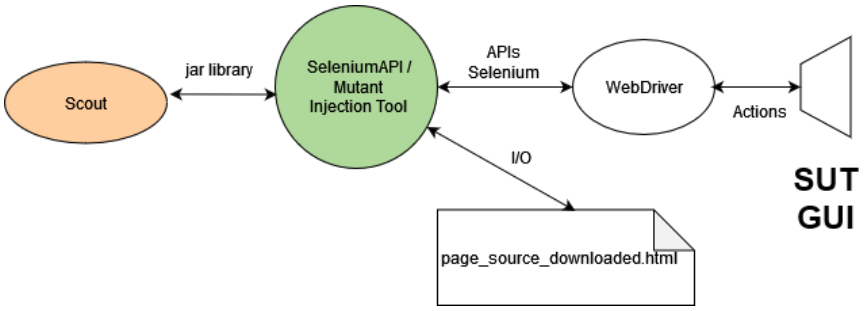


Figura 3.5: Panoramica del tool di Mutants Injection.

Il processo di estrazione dei widgets, come detto, avviene tramite il metodo *executeScript* offerto dal WebDriver, che permette di ottenere un oggetto *JSON*, contenente tutte le informazioni prima descritte e facilmente parsificato mediante [36, Gson] per essere utilizzato all'interno delle strutture dati del tool. Di seguito, in figura 3.6, viene mostrato un class diagram rappresentativo delle strutture interne del tool, completo anche del necessario per l'integrazione successiva in Scout.

Per una panoramica del funzionamento si veda lo schema architetturale in figura 3.5, dove si nota la, precedentemente descritta, interazione con il WebDriver di Selenium per ottenere il codice sorgente della pagina dove verrà applicato un mutant

ad un widget "vittima" con conseguente iniezione nel browser del nuovo sorgente. Il tutto, come vedremo nel paragrafo 3.2.2, viene inserito e utilizzato in Scout per la fase successiva.

3.2.1 Funzionalità

Facendo riferimento alla tabella 3.3, vengono descritti di seguito i metodi di implementazione dei mutants, generati casualmente a partire dal widget, scelto anch'esso casualmente e con i criteri visti in precedenza. Tutte le modifiche al widget partono da due proprietà di *WidgetInfo*, `queryWidget()` e `innerHTML`. La prima permette di ottenere la stringa identificativa dell'elemento del DOM con i suoi attributi, ad esempio, in `<a href="/">Link!</a>` viene considerato dal primo < al primo >, ovvero l'elemento *<a>* con il suo, unico in questo caso, attributo *href*. Gli elementi interni vengono presi con *innerHTML*, proprietà del DOM adatta ad ottenere la stringa completa degli elementi figli del corrente. Era stato fatto un tentativo con un altro metodo ad hoc *queryChildren()* che però causava spesso eccezioni per cui la scelta è ricaduta sul già presente attributo prima citato. I metodi per la creazione dei mutants, nel dettaglio, ricevono tutti il *pageSource* corrente e senza modifiche per generare il nuovo sorgente con la modifica applicata, se è possibile farlo:

1. **Rimozione completa**: la rimozione, va semplicemente a cercare nel sorgente il widget corrente per sostituirlo con una stringa vuota, così all'inserimento del nuovo codice nel browser non sarà più presente;

2. **Invisibilità**: per rendere un widget invisibile si va a cercare tra gli attributi l'eventuale presenta di *style*, inserendo opportunamente la proprietà CSS di *visibility: hidden*. Nonostante è possibile che alcune proprietà di stile siano presenti in un file CSS separato, l'aggiunzione tra gli attributi nell'HTML, della stessa, ha la priorità e rende così possibile l'operazione;

3. **Duplicazione**: per quanto riguarda la duplicazione, si va a cercare il widget corrente nel sorgente sostituendolo con la concatenazione di due copie di sé stesso;

4. **Aggiunzione Simile**: per aggiungere un widget simile è stata considerata la categorizzazione fatta in [32, Mozilla] che prevede la suddivisione in 6 sottoinsiemi principali degli elementi del DOM, a cui ne è stato aggiunto uno manualmente che potesse gestire gli elementi delle liste come *li*. A supporto di ciò, è presente l'oggetto *DOMCategories*, figura 3.6, che contiene queste informazioni e i metodi per ricavare il widget coerente alla ricerca da effettuare. In particolare, per la ricerca del "simile", mediante *getSimilar()*, viene estratto casualmente un elemento dello stesso insieme di partenza del widget "vittima"

per poi essere sostituito nel codice sorgente mantenendo sempre gli stessi elementi interni;

5. **Aggiunzione Differente**: per aggiungere un widget differente, analogamente a quello simile, viene, mediante *getDifferent()*, estratto casualmente un elemento di un insieme differente, ovvero categoria distinta, rispetto quello di partenza. L'operazione di sostituzione avviene come sopra;

6. **Cambio Posizione**: il cambio di posizione di un widget prevede per prima una rimozione del widget dalla sua posizione attuale, come al punto 1, poi l'inserimento di esso in una nuova posizione che risulta nel cambio di relazione gerarchica tra l'elemento corrente e il suo nodo genitore. Per ricercare la nuova posizione viene considerata la sola parte del *body* nel sorgente, in cui è possibile scegliere casualmente la posizione del tag di chiusura di uno dei *div* presenti all'interno, ovvero `</div>`. Così facendo il mutant diventa un nodo "fratello" del *div* scelto;

7. **Cambio Posizione per renderlo sovrapposto ad un altro**: il cambio di posizione per sovrapposizione sfrutta la proprietà CSS *position* per andare ad inserire nel widget l'attributo *absolute* o *fixed* che, al contrario di *relative* o del valore di default *static*, permettono di mantenere l'elemento sempre in una posizione tale che, se anche se la pagina viene "scrollata", possano sovrapporsi agli altri elementi, eccetto alcuni casi per *absolute*, quando l'elemento corrente non ha predecessori con una posizione relativa al layout;

8. **Ridimensionamento**: per ridimensionare un widget, analogamente come al punto 2, è stato usato l'attributo *style* per aggiungere le proprietà CSS che modificassero la larghezza, *width*, e l'altezza, *height*, del widget ad un valora fisso di 420px per entrambi;

9. **Cambio Aspetto**: anche per cambiare aspetto al widget, si intende nello stile risultante nella GUI, è stato usato l'attributo *style*, in particolare impostando il colore di sfondo *background-color* e lo spessore, lo stile e il colore del contorno dell'elemento;

10. **Cambio Tipo di widget**: come ai punti 4 e 5, il cambio di tipo di widget prevede l'utilizzo delle categorie del DOM per trovare un sostituto dell'elemento corrente. Nello specifico, è analogo al punto 5, si cerca quindi un widget differente e lo si inserisce al posto del precedente, senza però essere concatenato al preesistente questa volta e quindi duplicato come avviene per l'aggiunzione.

In figura 3.7 vengono mostrati in modo schematico e d'insieme gli esempi di applicazione dei 10 mutants, sopra descritti, ad un widget pre-selezionato della pagina principale di Wikipedia, presente all'URL ***wikipedia.org***.

Per ottenere una più generale affidabilità, il tool è stato sottoposto a dei test quantitativi che mostrassero il corretto funzionamento e l'avvenuta applicazione dei mutants visti sopra. Sono state usate le pagine di *wikipedia.org*, *tesla.com* e *polito.it*, in ognuna delle quali 42 widgets casuali sono stati sottoposti ad una mutazione, scelta in modo sequenziale dalla tabella 3.3, applicata sequenzialmente sul sorgente della pagina aggiornato eventualmente dalla precedente mutazione. I risultati si sono basati su due attributi della classe ***MutationWidget***, *found* e *mutationApplied* che hanno permesso di capire rispettivamente, per ogni *mutationName*, se il widget venisse correttamente individuato all'interno del sorgente HTML e se quest'ultimo, dopo la mutazione, fosse stato realmente modificato. Su ognuna delle 3 pagine, il tool è stato eseguito 2 volte in modo indipendente, così da creare una statistica finale su 84 tentativi di mutazione per ogni pagina. In figura 3.8 sono presenti i tre grafici corrispondenti, ognuno riportante i seguenti valori, suddivisi per tipo di mutant e per tipo di widget:

- *Valor Medio di "found"* (in blu), **found** vale **1** se il widget estratto è stato trovato nel sorgente;
- *Valor Medio di "mutationApplied"* (in arancione), **mutationApplied** vale **1** se il mutant è stato inserito correttamente nel sorgente;
- *Totale Mutazioni tentate* (in grigio).

Possiamo notare come per *wikipedia.org* e *polito.it* tutti i widgets correttamente estratti siano stati altrettanto mutati, tranne nei pochi casi in cui non sono stati trovati all'interno della pagina, dovuti probabilmente alla modifica cumulativa del codice sorgente. Quest'ultima situazione è contrassegnata con una "x" nella tabella per indicare che nessuna mutazione è stata tentata, in questo caso il numero totale del conteggio (in grigio) è indicativo solamente dei widgets estratti. Mentre per *tesla.com*, oltre ad essere presente un numero maggiore di widgets non trovati, troviamo tre casi, mutants *1, 4 e 5*, in cui l'applicazione della mutazione al sorgente risulta fallita nonostante il widget fosse stato correttamente identificato: per il mutant *1* si può trattare di un errore nella modifica della stringa del sorgente, caso isolato quindi ignorato; per i mutants *4 e 5* si tratta invece dei limiti previsti sulle categorie non presenti in [32, Mozilla], in questo caso ***source***, errori quindi anche qui considerati ignorabili ma comunque discussi nel paragrafo 5.1 e tra i miglioramenti futuri del tool.

Non considerando l'aspetto qualitativo[2], ovvero se effettivamente nella GUI venisse percepito un cambiamento, a volte anche questo causato dall'utilizzo di

[2]Un esempio lo possiamo comunque vedere in figura 3.9, che mostra due esecuzioni differenti dell'applicazione massiva dei 42 mutants sulla pagina principale di Wikipedia.

categorie differenti nei casi dei mutants *4, 5 e 10*, i risultati sono comunque stati ritenuti attendibili e quindi l'applicazione dei mutants accettabile per l'inserimento in Scout.

3.2.2 Adattamento del tool in Scout

La seconda fase di sviluppo ha riguardato l'inserimento del tool, descritto sopra, all'interno di Scout, con l'integrazione di alcuni elementi di Gamification, [40]. A partire dal *SeleniumPlugin*, come abbiamo visto nello schema in figura 3.2, è stato possibile utilizzare la classe *SeleniumAPI*[3] mediante il jar appositamente creato alla fine della fase precedente. L'istanza viene generata all'interno di `startSession()` immediatamente dopo l'inizializzazione del *WebDriver*, necessario, come visto, per il funzionamento del tool. La vera e propria iniezione del mutant, tramite `scrapingWebsite()`, avviene in parallelo alla creazione dell'*EasterEgg*, all'interno di `verifyAndReplaceWidgets()`, per permettere un'unica modifica ad un widget di una pagina non precedentemente visitata.

Dal momento che Scout, tramite l'*Augmented GUI*, permette di interagire con i widgets della pagina attraverso l'uso di azioni, descritte nel paragrafo 3.1.2, da parte dell'utente, sono stati inseriti i metodi per controllare l'eventuale interazione con i mutants, ovvero per controllare se il widget sotto test, possibilmente modificato dal tool, venisse scoperto e quindi, il bug, trovato. In particolare, con le azioni di *issue e click* è possibile interagire con il tool per tenere traccia dei mutants, come vedremo successivamente mediante alcuni elementi di Gamification:

- *issue*, azione da eseguire nel caso in cui si voglia evidenziare una problematica riguardante il posizionamento o l'aspetto di un widget (testo, immagine, icona, destinazione di un link, ecc. errati o anche non presenti). Per segnalare un issue sono possibili due alternative:

  1. si esegue un *check* sul widget, digitando il testo della segnalazione
  2. si digita il testo della segnalazione direttamente su un widget di tipo *click*. Questa possibilità non era presente nel modello di Scout di partenza, è stata perciò aggiunta per avere la possibilità di interagire con il riconoscimento del mutant anche per un widget provvisto di un collegamento verso pagine esterne a quella corrente, essendo un tipo di elemento molto comune all'interno di una qualunque applicazione web e quindi previsto per essere modificato dal tool di mutant injection

[3]Non è stato specificato in precedenza ma il nome della classe, derivante dal tool sviluppato in Kotlin, potrebbe risultare non rilevante e creare confusione tra le altre varie strutture già presenti. Una nomenclatura più corretta sarebbe stata, ad esempio, *MutantInjectionAPI*.

- *click*, azione da eseguire quando si vuole cliccare un collegamento per raggiungere un'altra pagina o un'altra sezione della pagina corrente. Oltre a quanto detto prima, se il mutant generato è corrispondente ad un widget non più visibile nella pagina perché nascosto o rimosso (mutant *1* o *2* della tabella 3.3), l'azione "click" permette di esplorare le zone vuote della pagina dove si pensa ci fosse precedentemente quel widget

Il fulcro per svolgere i controlli di quanto detto sopra, per la ricerca del mutant, è il metodo `checkMutantInteraction()`, che per funzionare correttamente necessita di importanti informazioni tra quelle della struttura dati del mutant, che sono state aggiunte al tool in Kotlin proprio per quest'esigenza, nell'oggetto *MutationWidget*:

- *xPath*, successivamente all'inserimento del mutant nel sorgente della pagina, questa proprietà può essere soggetta a cambiamento. In particolare, per quanto riguarda i mutants *1* o *2* il path risulta in una stringa vuota per permettere la conseguente ricerca mediante "click", non essendo visivamente individuabile. Per i mutants *4, 5 e 10*, il nuovo path corrisponde gerarchicamente a quello del widget di partenza a meno del tag (*localName*) dell'elemento differente, anche per i casi di aggiunta dove a cambiare gerarchia è il widget originale, ad esempio, viene aggiunto un pulsante **body[1]/div[1]/button[1]** accanto un link testuale **body[1]/div[1]/a[1]**, entrambi sono figli di un div ma solo quello originario, il link in questo caso, diventa il secondo figlio, quindi nel caso del mutant *3*, dove il widget viene duplicato, il path sarà quello originario perché il widget di partenza diventerà il figlio **[2]** se consideriamo l'esempio precedente. Per le altre mutazioni resta valido il medesimo xPath di partenza, essendoci dei cambiamenti solamente all'interno delle proprietà CSS dell'elemento, a parte il mutant *6*, che necessita di una ricreazione dell'albero dei widgets per fare rigenerare al browser il giusto percorso nel DOM dopo il cambio di posizione avvenuto. Ad ogni creazione di issue si va quindi a controllare se il path del widget identificato, disponibile all'interno dei relativi metadati, sia corrispondete a quello del mutant generato

- *posizione* e *dimensioni*, usati per permettere la conseguente ricerca mediante "click" per le mutazioni *1 e 2*, non essendo visivamente individuabili. Al click in una zona "vuota" da widgets viene controllata l'eventuale precedente presenza dell'elemento grazie alla proprietà dell'AT di Scout, *MoveAction*, che permette di ricavare il punto di coordinate (x,y) del click del mouse all'interno della pagina. Il tutto rispetto alla posizione del mutant e alla sua dimensione, considerando anche un intorno di 32 pixels, trattati come tolleranza

Per tenere traccia dei mutants creati e trovati si è interagito con l'albero della sessione, figura 3.3, in particolare creando i riferimenti al totale dei mutants generati (e correttamente applicati) per la *Session* di Scout, dall'inizio della prima

navigazione sino all'ultima pagina navigata prima della terminazione, e al totale di quelli correttamente trovati. Così come per l'astrazione di ogni pagina, oggetto *Page*, che mantiene il riferimento all'oggetto associato ***MutationWidget mutant*** e un flag indicatore dell'eventuale scoperta da parte del tester, opportunamente assegnato nel metodo descritto sopra.

Quanto aggiunto è risultato utile ai fini della gamification, poiché ha permesso di inserire ulteriori statistiche su quanto fatto dal tester durante la sessione. Tramite le strutture viste nel paragrafo 3.1.3, figura 3.4, è stato possibile inserire i dati relativi al numero totale di mutants trovati e non, sia per quelli della sessione in corso che per quelli provenienti dalle sessioni precedenti, grazie ai file fungenti da database, che rendono persistenti le informazioni sulle metriche elaborate. Quest'utlime vengono mostrate nella schermata di riepilogo *RecapGUI*, in particolare, quella riguardo al calcolo del punteggio totale, ottenuto durante la sessione di testing, viene calcolata partendo dalla combinazione delle componenti considerate base e di quelle considerate come bonus, secondo la seguente formula:

$$\mathbf{P = a \cdot C + b \cdot EX + c \cdot EF + [\ d \cdot T\ ] + [\ e \cdot PR\ ]}$$

Le prime 3 componenti rappresentano il punteggio base, ognuna pesata del proprio coefficiente mentre quelle tra parentesi quadre, considerate come opzionali, sono quelle bonus. Per il dettaglio si rimanda a [33, Fulcini]. Il tester ha dunque la possibilità di accumulare punti bonus, fino ad un massimo del 50% sul totale base ottenuto, mediante:

$$\mathbf{[\ PB\ ] = d \cdot T + e \cdot PR}$$

dove **T** rappresenta la componente temporale legata alla durata della sessione e **PR**, quella legata alle problematiche trovate dal tester e quindi, in questo caso, anche l'interazione avvenuta con i mutants all'interno della pagina web, calcolata come la somma tra **bug, issue e easteregg** presenti tra le metriche finali della sessione. Per ognuna di esse è presente un coefficiente di saturazione, ovvero un valore massimo che possono assumere per evitare di ottenere un punteggio finale sul bonus troppo elevato. Oltre alla saturazione è presente anche il coefficiente peso che ad ogni problematica viene assegnato, nel caso della metrica sul mutant (qui indicato come il contatore dei bug) viene posto uguale a 1 per evidenziare unicamente il punteggio ottenuto alla fine di ogni navigazione su di una pagina che dipende, a sua volta, dalla combinazione lineare delle componenti *m1, m2, m3*:

- **m1**, rappresenta un peso fisso associato ad ogni tipologia di mutant, aggiunto nella creazione dell'oggetto *MutationWidget* ai fini della validazione del tool, come viene descritto nel capitolo successivo. Viene considerato solo se *m3* è presente
- **m2**, rappresenta la ***bugSeverityScore***, ovvero il punteggio che il tester deve assegnare alla percezione visiva del bug generato sulla GUI dal tool.

Score	Description of Severity
0	I did not notice any fault
1	I noticed a fault, but I would return to this website again
2	I noticed a fault, but I would probably return to this website again
3	I noticed a fault, and I would not return to this website again
4	I noticed a fault, and I would file a complaint

Tabella 3.4: Bug Severity basata sulla percezione dell'utente.

Se il punteggio su quel mutant non viene assegnato in fase di testing, la componente *m2* conterà come nulla nella combinazione con le altre. Anche questo attributo è presente nell'oggetto *MutationWidget* ed è stato aggiunto ai fini della validazione successiva

- **m3**, è semplicemente un indicatore binario sull'eventuale scoperta del mutant durante la navigazione all'interno della pagina

L'idea di dare un score al mutant, specialmente per *m2*, la troviamo in [39, Yandrapally et al.], dove viene specificata la severità con il quale considerare visivamente un mutant, si veda la tabella 3.4. Per assegnare il valore ***bugSeverityScore*** è stato implementato un altro elemento di gamification, il *voting*, attraverso un plugin appositamente dedicato, ***DisplayStatsOverlay***. Il plugin, inoltre, permette di visualizzare, se attivato, le statistiche in tempo reale sul numero di mutants presenti e quelli scoperti nella sessione in corso. Il voto alla severità del mutant può essere assegnato una volta sola per pagina esplorata, se questo è stato correttamente inserito all'interno di essa e risulta quindi individuabile (anche se non visibile, come abbiamo visto in precedenza). L'assegnazione avviene tramite tastiera, utilizzando i numeri da 1 a 5, perciò è stato reso necessario "disattivare" la possibilità di inserire questi 5 caratteri durante il typing per le issues nel *SeleniumPlugin*.

L'interazione con i mutants, riguardo quanto detto fin'ora, viene registrata tramite il metodo `saveMutantInteraction()` richiamato sia ad ogni cambio di pagina dovuto alla navigazione interna tra link, sia in *stopSession()* per considerare anche l'ultima pagina visitata della sessione. Viene generato anche un file[4] contenente una struttura dati JSON per permettere la successiva analisi, come vedremo, di quanto effettuato dal tester, formata nel modo seguente:

- pageId, il nome della pagina web visitata, identificato dall'indirizzo URL
- isMutFound, l'attributo che identifica l'eventuale scoperta del mutant

[4]chiamato "log_MutantsData.json" nello schema di figura 3.4 ma suddiviso in realtà in più file associati al nome del tester e al timestamp della creazione

- mutationScore, il punteggio fisso di base associato alla tipologia del mutant
- mutationName, il tipo di mutant applicato alla pagina
- xPath, per permettere eventualmente di rintracciare nuovamente il relativo mutant nel codice sorgente della pagina

L'analisi dei suddetti dati relativi alle sessioni di testing svolte verranno affrontate nel capitolo successivo insieme con i dettagli di quanto emerso dal sondaggio statistico finale fornito ai tester volontari.

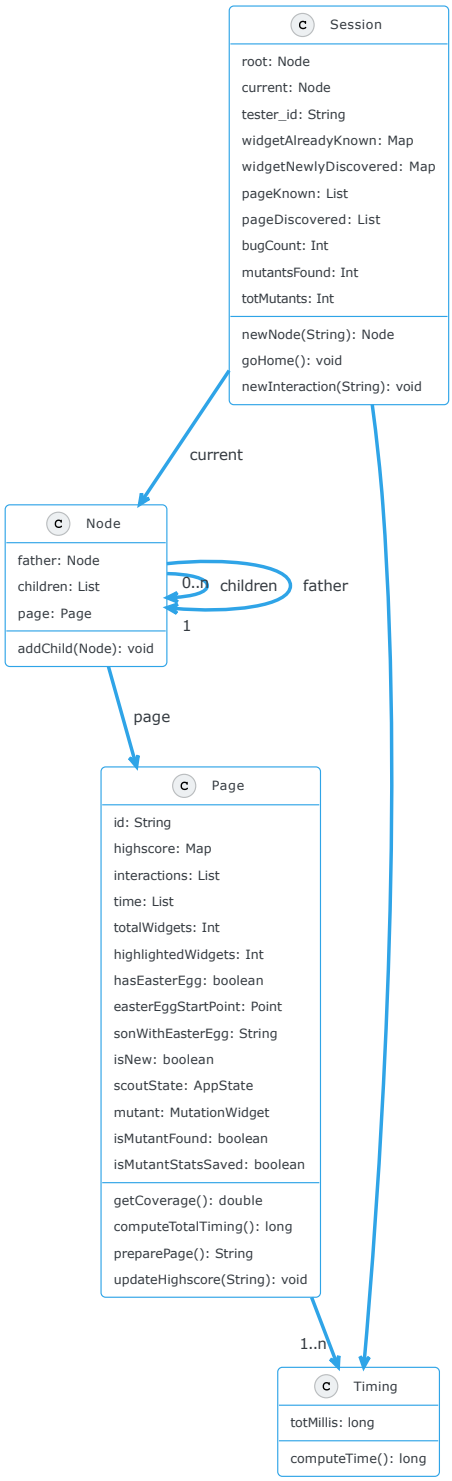


Figura 3.3: Classi rappresentati la sessione in Scout.

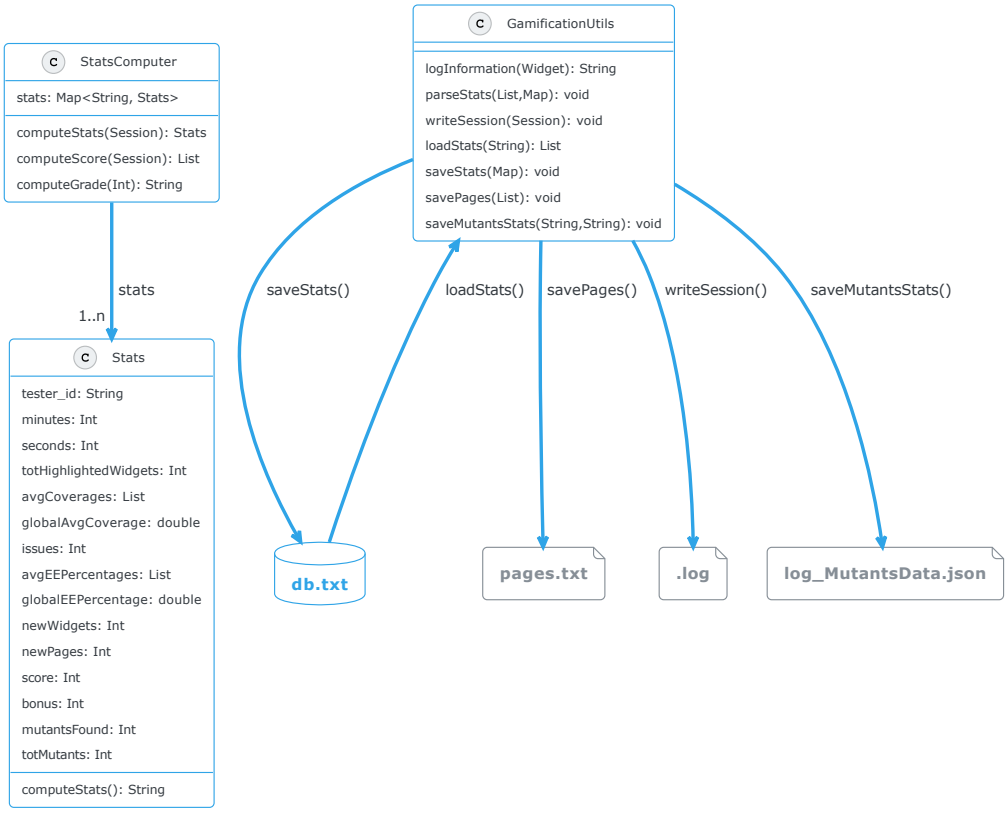


Figura 3.4: Gamification engine.

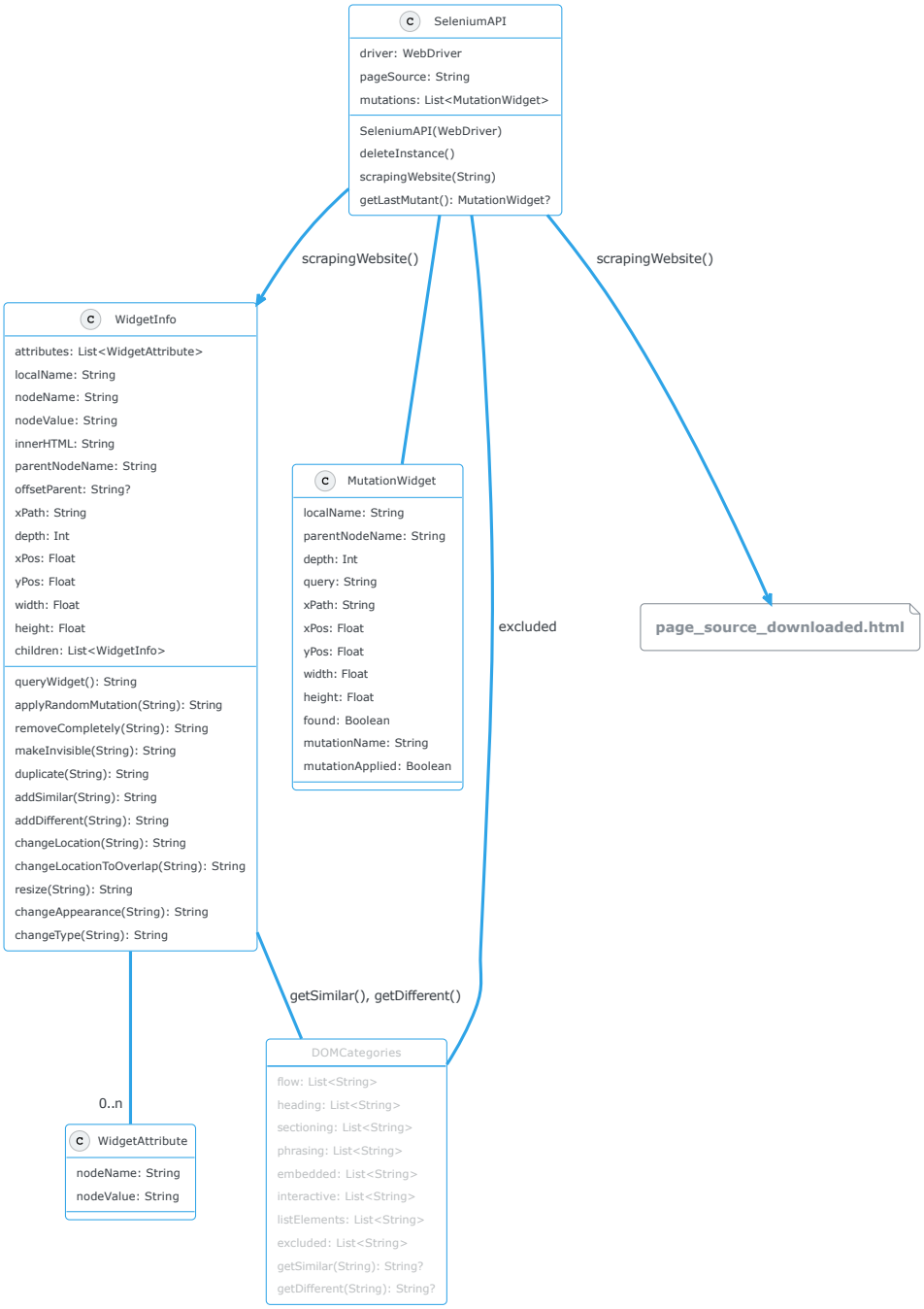


Figura 3.6: Class diagram Mutants Injection tool.

(a) Widget senza mutant

(b) Mutants 1, 2

(c) Mutant 3

(d) Mutant 4

(e) Mutant 5

(f) Mutant 6

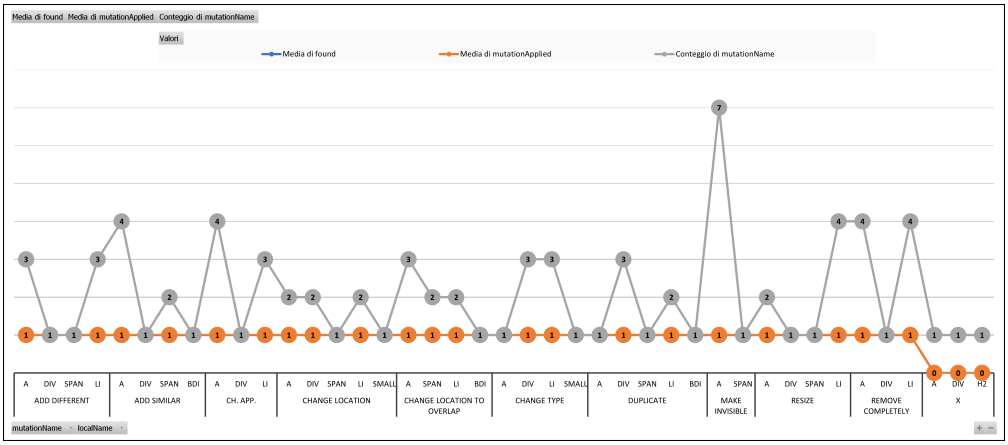
(g) Mutant 7

(h) Mutant 8

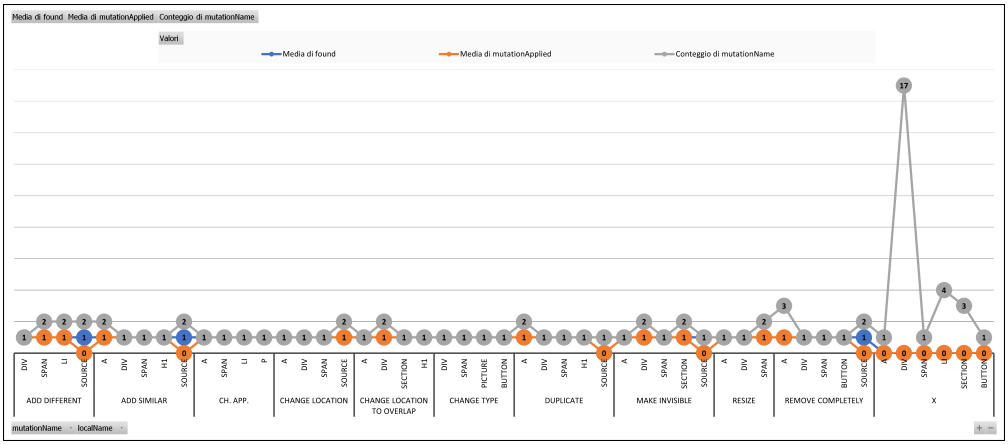
(i) Mutant 9

(j) Mutant 10

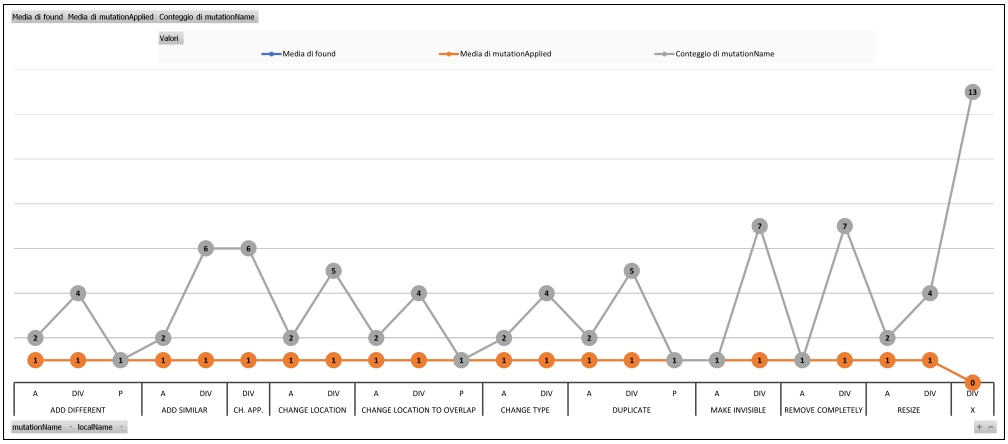
Figura 3.7: Esempi di mutants.



(a) wikipedia.org

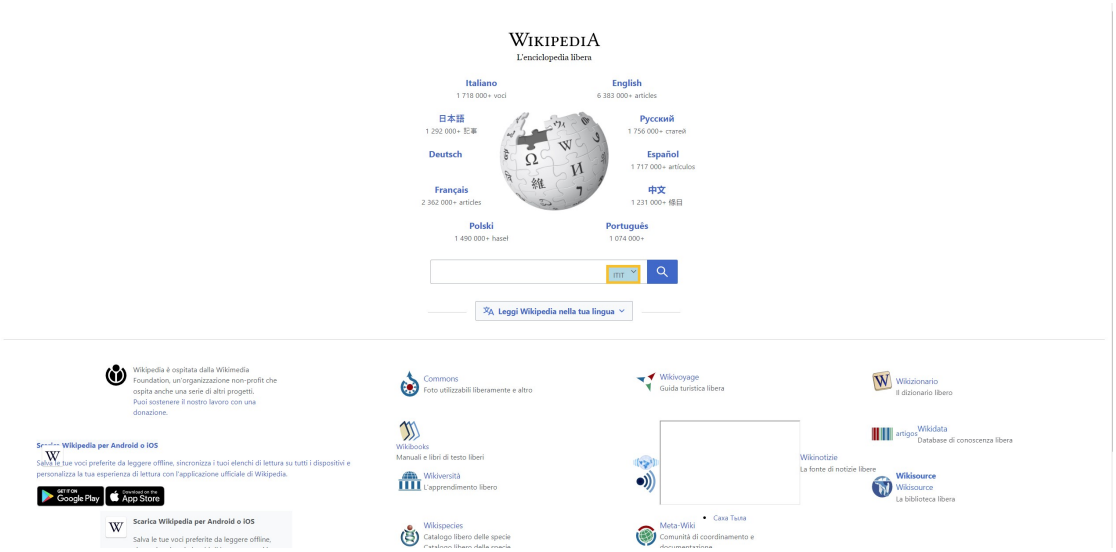


(b) tesla.com



(c) polito.it

Figura 3.8: Grafici quantitativi sull'andamento delle mutazioni nelle 3 applicazioni web testate.



(a) Esecuzione 1: mostra la pagina con il menù primario chiuso

(b) Esecuzione 2: mostra la pagina con il menù primario aperto

Figura 3.9: Esempio di applicazione massiva di 42 mutazioni in *wikipedia.org*.

Capitolo 4

Validazione del tool

Per validare l'utilizzo del tool, sviluppato e descritto nel capitolo precedente, è stata effettuata una fase di sperimentazione mediante l'uso di Scout con le modifiche viste per inserire la mutant injection. Queste sessioni di validazione, vere e proprie sessioni di testing, sono state effettuate a distanza e in modo indipendente, ovvero non in un ambiente controllato. I tester sono stati un campione di 6 persone, come vedremo, scelte secondo un adeguato criterio.

L'idea di base della sperimentazione è stata quella di effettuare delle fasi di ***Exploratory Testing*** all'interno delle applicazioni web rese disponibili e navigabili tramite Scout. Questa tecnica di testing prevede che non venga seguita alcuna specifica istruzione[1] da parte dell'utente, quindi non venga seguito alcuno "script" (a parte, come vedremo, delle indicazioni sul funzionamento e sulle limitazioni del tool) bensì venga data la possibilità di esplorare liberamente il software, la pagina web in questo caso, per trovare i bug sul momento, ovvero i mutants inseriti nella GUI dal tool di generazione automatica. Il tester ha perciò la possibilità di apprendere, contemporaneamente all'esplorazione, il funzionamento di Scout, anche grazie alle istruzioni fornite, e, di conseguenza, di interagire in modo autonomo con gli elementi di Gamification presenti, in modo da adattare il proprio punteggio finale senza la dipendenza da predeterminate azioni da eseguire.

[1] Ad esempio, effettuare necessariamente il click su una specifica parte della pagina o inserire uno specifico dato in uno specifico form.

4.1 Metodo

L'esperimento è stato sottoposto al campione di persone fornendo loro le istruzioni[2] da seguire, con incluso un minimo di background per far apprendere la modalità di test e la tipologia di test da svolgere. In particolare, ciascun soggetto sperimentale ha dovuto eseguire 3 fasi di *Exploratory Testing* su tre istanze successive di Scout, per ognuna delle quali si è testata una specifica applicazione web fornita tra le opzioni del menu iniziale di Scout. Le applicazioni da testare sono state le seguenti, tutte a partire dalla home page fornita:

1. **Wikipedia**, da *wikipedia.org*;
2. **Portale della Didattica - PoliTO**, da *didattica.polito.it*;
3. **Board Game Geek**, da *boardgamegeek.com*.

Le 3 fasi di testing si sono terminate al raggiungimento di almeno **10 pagine esplorate** o di circa 15 minuti trascorsi, a discrezione comunque del tester. La motivazione delle 10 pagine ha riguardato la creazione dei mutant: poiché si sono usate le mutazioni della tabella 3.3, per garantire che venissero usate almeno una volta tutte quante e quindi per rendere comparabili le sedute di test tra i vari volontari, si sono dovute apportare alcune modifiche al tool nel suo complesso. Nello specifico, come abbiamo visto, la generazione dei mutants è stata pensata per essere totalmente casuale e la mutazione pensata per essere applicata ad un widget per pagina, estratto anch'esso in modo casuale. Qui, invece, il tool è stato adattato in modo da generare i 10 tipi di mutants ciclicamente ed in ordine progressivo, sempre su widgets estratti casualmente: un contatore segue la creazione delle nuove pagine e genera il mutant corrispondente, dalla lista dei 10 previsti, all'indice dato dal contatore, l'eventuale undicesima pagina genererà nuovamente il primo mutant e così via seguendo l'ordine della tabella 3.3. Un'altra modifica è stata apportata al filtro di selezione del widget casuale all'interno del DOM, in particolare limitando l'altezza massima della pagina a cui trovare un determinato elemento, ovvero la coordinata ***yPos*** di *WidgetInfo*, impostata quindi a *1200.0*, per evitare di estrarre un widget non visibile nella parte della pagina mostrata dalla finestra del browser poiché più in basso e quindi visibile solo dopo conseguente scroll manuale: la motivazione della modifica risiede proprio in quest'ultima azione, che rimane uno dei limiti di Scout, come vedremo nuovamente nel capitolo successivo.

Come accennato al termine del capitolo precedente, il partecipante all'esperimento ha avuto la possibilità di utilizzare l'elemento di Gamification del "voting",

[2]All'interno del repo in [40] è presente il file *script.pdf* contenente le istruzioni fornite al tester con relativi link al materiale da scaricare e al sondaggio di valutazione finale.

Score	Descrizione
0	Non riesco a trovarlo
1	Noto un leggero bug, potrei dover analizzare meglio la pagina
2	Noto un leggero bug, dovrei analizzare meglio la pagina
3	Noto un leggero bug, non ho bisogno di analizzare meglio la pagina
4	Noto un evidente bug, non ho bisogno di analizzare meglio la pagina
5	Noto un grave ed evidente bug

Tabella 4.1: Bug Severity basata sulla percezione dell'utente usata nel plugin DisplayStatsOverlay.

inserito tramite il plugin *DisplayStatsOverlay*, per assegnare un *bugSeverityScore*. La tabella 3.4 presa da [39, Yandrapally et al.] è stata qui adattata per una migliore interpretazione da parte del tester, si veda tabella 4.1. Nello specifico la scala dei valori è stata ampliata negli ultimi due punti ai fini di applicare una diversa valutazione nel punteggio finale assegnato al tester, si può notare quindi come la scala dei valori abbia 6 punteggi, il punteggio "0" è assegnato in questo caso se nessuna interazione col plugin è stata effettuata, come specificato nelle istruzioni.

Altro elemento aggiunto, che ha permesso di pesare l'eventuale scoperta di un mutant da parte del tester, è stato il ***mutationScore***, utilizzato, come visto, dal parametro *m1* nel calcolo del punteggio bonus all'interno di una determinata pagina. La scala di valori, tabella 4.2, si basa sull'analisi fatta in precedenza, figura 3.8, prendendo come riferimento una percezione visiva di base indicativa della difficoltà con la quale uno specifico tipo di mutant venga a primo impatto visualizzato sulla pagina. Le mutazioni *3, 8 e 9* sono state considerate le più semplici, mentre quelle riguardanti il cambio di posizione, *6 e 7*, le più complesse da identificare. L'unione del *mutationScore* e del *bugSeverityScore* ha permesso di creare, come vedremo nei risultati in 4.3, una statistica più qualitativa sull'impatto dei mutants disponibili nel tool.

Prima delle istruzioni sul funzionamento del tool sono stati forniti i passi da seguire per impostare l'ambiente di testing, considerando le limitazioni di Scout. Per prima cosa i seguenti requisiti si sono resi necessari:

- S.O. Microsoft Windows 10 (oppure 11);
- Microsoft Edge, installato e aggiornato correttamente sulla macchina, per essere compatibile con il WebDriver di Selenium;
- Java correttamente installato (versione >= 8), necessario per eseguire il tool in tutte le sue parti;

#	Nome Mutant	Score
1	removeCompletely	2
2	makeInvisible	2
3	duplicate	1
4	addSimilar	3
5	addDifferent	3
6	changeLocation	4
7	changeLocationToOverlap	4
8	resize	1
9	changeAppearance	1
10	changeType	3

Tabella 4.2: Mutation Score usata per indicare la difficoltà nella ricerca di un mutant.

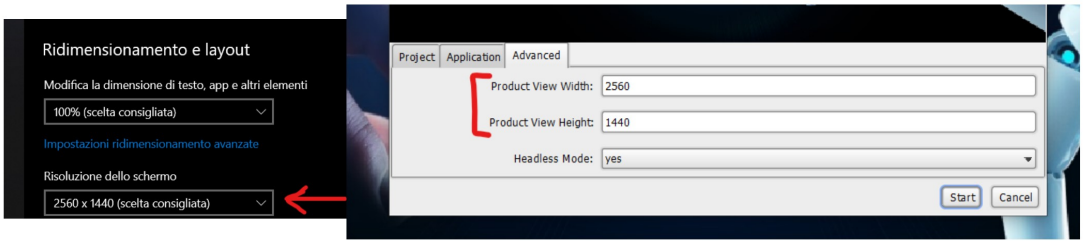


Figura 4.1: Impostazioni layout dello schermo nel S.O. per l'utilizzo di Scout.

- Connessione ad internet funzionante e stabile, necessaria per effettuare la navigazione online delle applicazioni web.

In generale si è preferito non fornire una macchina virtuale con l'ambiente già preimpostato poiché non si è voluto rendere più "pesante" l'esecuzione della sessione di testing, tenendo presente il possibile sovra-utilizzo delle risorse da parte di Scout. Le altre limitazioni, sulle impostazioni di quest'ultimo, riguardano alcuni dettagli sulla risoluzione dello schermo utilizzata dal S.O. che, per far funzionare il tool, deve essere la medesima di quella impostata dentro le opzioni disponibili in Scout e soprattutto non deve essere modificata nelle sue proporzioni di ridimensionamento e zoom, e quindi mantenersi al 100% (figura 4.1), poiché in alcune prove è risultata una proporzione delle posizioni non corretta per quanto riguarda i widgets all'interno dell'Augmented Layer che invece risultavano corretti con il 100% delle dimensioni.

Nella fase conclusiva è stato chiesto ai tester di compilare un questionario utile, come vedremo successivamente, all'analisi dei risultati, insieme con la sottomissione dei dati sulla sessione, creati all'interno del tool.

Per quanto riguarda i tester, il campione delle persone è stato selezionato considerando che lo strumento, Scout in questo caso, nel quale è stato inserito il tool di generazione automatica dei mutants, fosse specificatamente professionale ed orientato all'utilizzo da parte di persone con certo background sul SE ma anche possibilmente sul Software Testing in generale. Per questo motivo i partecipanti all'esperimento sono stati scelti da un ambiente molto simile al target reale: sono stati studenti ed ex studenti del corso di laurea magistrale in Ingegneria Informatica presso il Politecnico di Torino. Alcuni dettagli sul background generale dei tester sono stati ottenuti, anch'essi, dal questionario, tabella 4.3.

Tabella 4.3: Domande del questionario finale, prima parte.

ID	Domanda (Tipo)
1.1	Età (Scelta Multipla)
1.2	Hai mai avuto esperienze professionali con Java, come studente o lavoratore? (Scelta Multipla)
1.3	Quanti anni di esperienza hai con la programmazione in Java? (Scelta Multipla)
1.4	Quanti anni di esperienza hai con la programmazione in Web? (Scelta Multipla)
1.5	Quali tra questi linguaggi hai usato per la programmazione Web? (Checkbox)
1.6	Hai mai avuto esperienze di testing di applicazioni Java? (Scelta Multipla)
1.7	Quali tool hai utilizzato per fare testing di applicazioni Java? (Aperta)
1.8	Hai mai avuto esperienze di testing di applicazioni Web? (Scelta Multipla)
1.9	Quali tool hai utilizzato per fare testing di applicazioni Web? (Aperta)
1.10	Hai mai avuto esperienze di Mutation testing di software? (Scelta Multipla)
1.11	Quali tool hai utilizzato per fare Mutation testing? (Aperta)

L'età è stata compresa tra i 22 e i 30 anni, il che influisce sulla possibile esperienza di background acquisita. Come vediamo dalle figure 4.2, 4.3 e 4.4 (che mostrano rispettivamente le risposte ottenute dalle domande 1.3, 1.4, 1.5), 5 dei partecipanti hanno avuto almeno 1 anno di esperienza sia nella programmazione Java che in quella web e mediamente tutti a conoscenza dei principali linguaggi di sviluppo usati nelle applicazioni web, principalmente in PHP/HTML/CSS, che, dato la diretta interazione che dà con la GUI di una pagina, può avere aiutato a riconoscere più facilmente alcuni elementi del DOM durante il testing. In figura 4.5 e 4.6 osserviamo l'esperienza dei partecipanti nei confronti del testing: sebbene, come detto, sarebbe importante, in questo caso, la mancanza non è un elemento

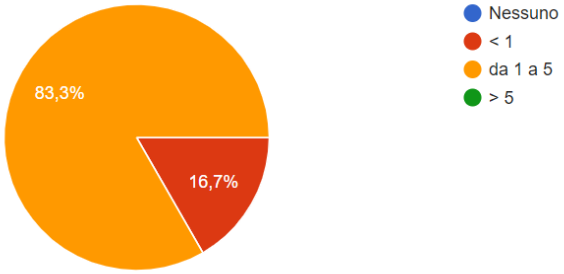


Figura 4.2: Sondaggio finale: esperienza in Java.

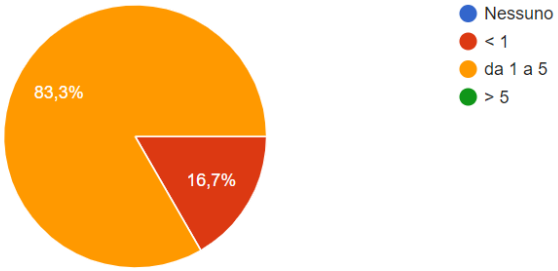


Figura 4.3: Sondaggio finale: esperienza in programmazione web.

particolarmente influente a causa dell'esplorazione meramente manuale e intuitiva dell'interfaccia grafica delle pagine web in Scout, essendo simile all'interazione che farebbe qualsiasi utente. Tuttavia, chi ha utilizzato alcuni strumenti ha evidenziato la conoscenza di *JUnit* o di *Jest*. Nessuno dei partecipanti ha avuto esperienze con il Mutation Testing, lasciando intendere, dalla secca risposta negativa, che l'argomento non fosse nemmeno familiare. Per cui, possibilmente, l'esperienza di questo tool ne è stata il primo approccio.

4.2 Esempio di Sessione

Prima dell'analisi dei dati derivanti dalle sessioni di testing dei volontari, viene mostrato qui un esempio di sessione del tool nella sua interezza, utilizzato sull'applicazione web di *Wikipedia*. Vengono mostrati in sequenza gli screen per ogni pagina navigata in modo da evidenziare i mutants generati, con una panoramica sul punteggio finale ottenuto. Ricordando l'ordine della tabella 3.3, la navigazione di esempio tramite Scout porta in ordine quanto segue:

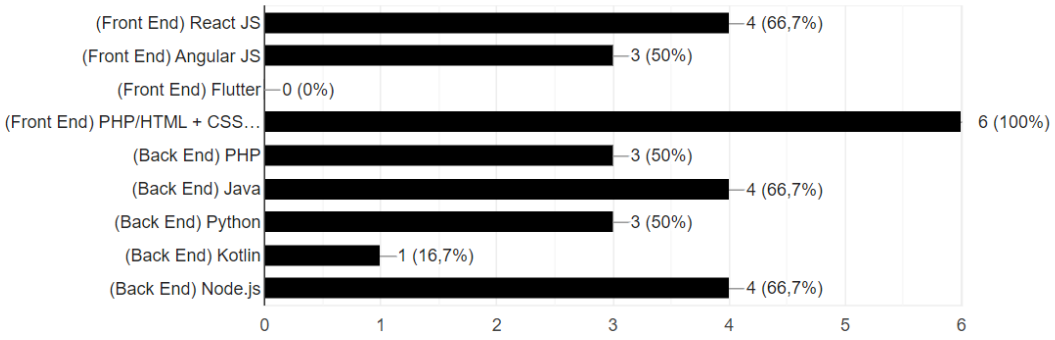


Figura 4.4: Sondaggio finale: linguaggi conosciuti per la programmazione web.

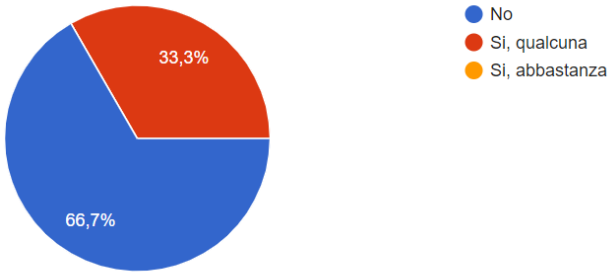


Figura 4.5: Sondaggio finale: esperienza in testing di applicazioni Java.

1. Si parte dalla home page di Wikipedia, in cui viene generato il primo mutant (figura 4.7). Come vediamo, la rimozione è visibile nella zona inferiore della pagina, senza accedere a nessun menù di opzioni in particolare. Come previsto si deve interagire con il "click" sulla zona dove era presente precedentemente il widget per rendere effettiva la scoperta del mutant. Si può inoltre votare il "severity score", in questo caso un 5 potrebbe essere adeguato alla situazione creatasi;

2. Cliccando su link "Italiano 1718000+ voci", si arriva nella pagina in figura 4.8. In questo caso, dato l'elevato numero di widgets contenenti puramente testo, è più complesso trovare il secondo mutant (invisibilità). Dopo una breve esplorazione, nel caso non si sia trovato nulla, si può perciò lasciare la pagina per proseguire la navigazione. Il voto alla percezione può essere qui ignorato, non avendo trovato nulla di concreto;

3. Cliccando sul link che porta alla pagina "it.wikipedia.org/wiki/Enciclopedia"

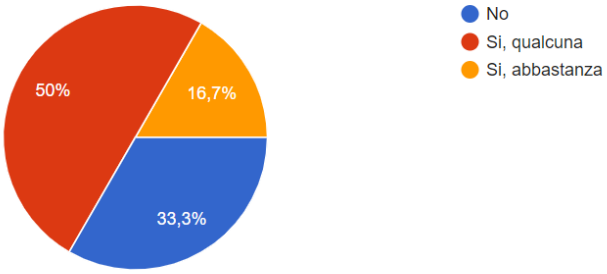


Figura 4.6: Sondaggio finale: esperienza in testing di applicazioni web.

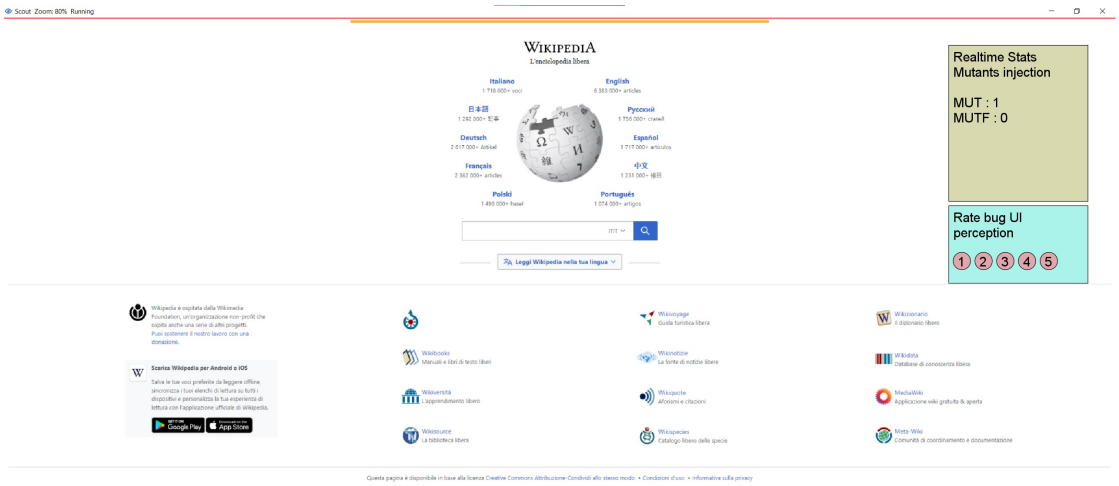


Figura 4.7: Esempio Sessione: pagina iniziale - mutant #1.

viene generato il mutant della duplicazione (figura 4.9). Notiamo come nel menù di navigazione sulla sinistra vengano sovrapposte alcune voci, dovute alla duplicazione di qualcuno dei widgets che ne compongono le parti. Effettuando le dovute azioni per assegnare "issues" si può cercare il mutant. La percezione in questo caso può dipendere dal successo della scoperta, non essendo scontato che il mutant venga correttamente conteggiato. Possiamo assegnare "2" ad esempio se non siamo riusciti a trovare il widget che effettivamente indicava la mutazione e che avrebbe richiesto una più approfondita esplorazione;

4. Cliccando sul link che porta alla pagina "it.wikipedia.org/wiki/Conoscenza" (figura 4.10), notiamo una situazione simile alla seconda mutazione. Data la difficoltà nella ricerca del mutant, si continua la navigazione senza effettuare la votazione sullo score;

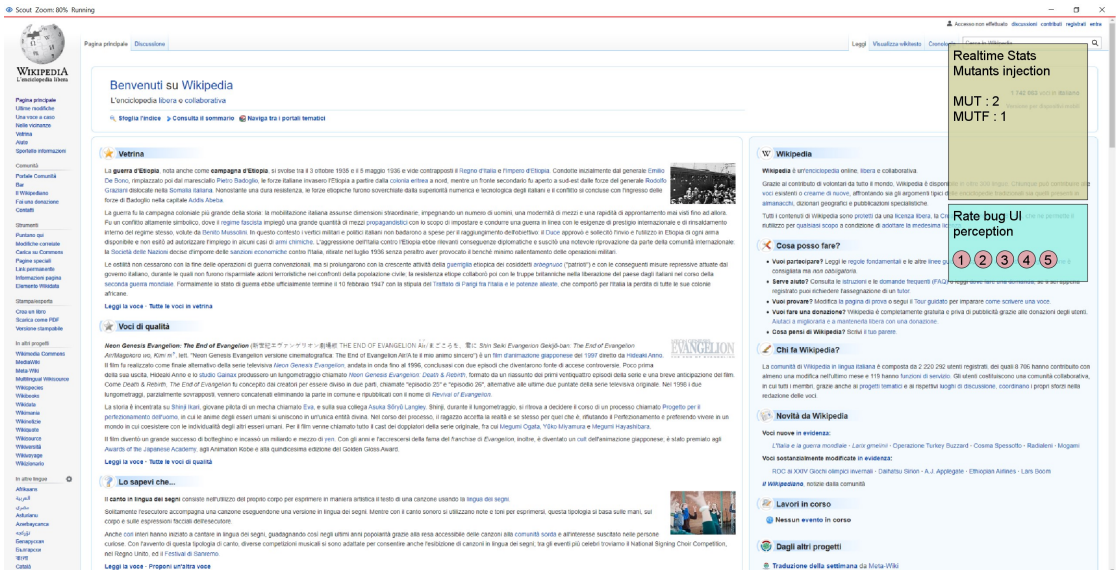


Figura 4.8: Esempio Sessione: mutant #2.

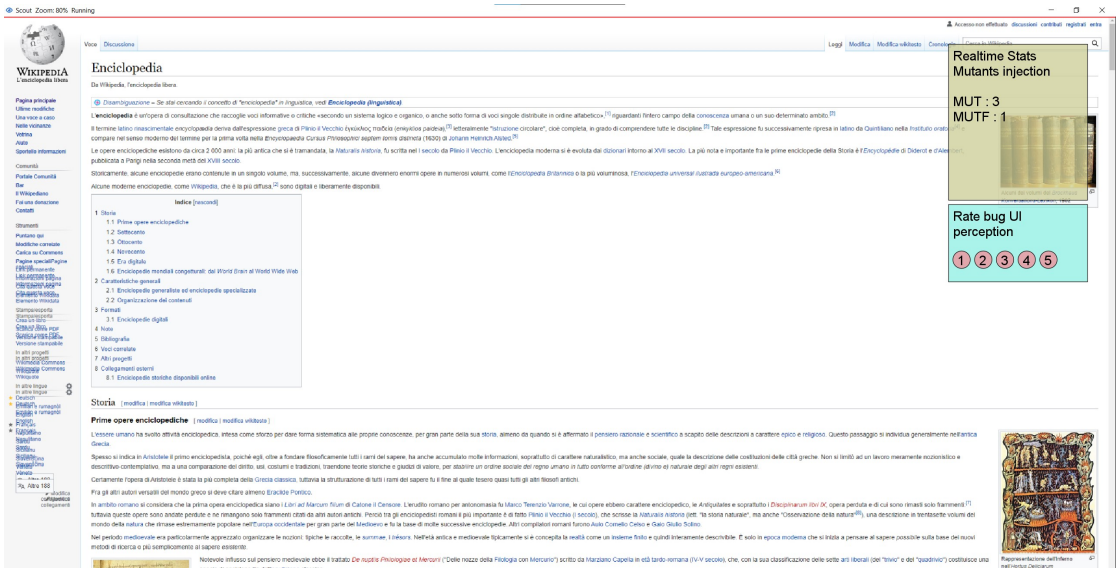


Figura 4.9: Esempio Sessione: mutant #3.

5. Cliccando sul link che porta alla pagina "it.wikipedia.org/wiki/Informazione", si genera l'aggiunzione di un widget simile. In figura 4.11 vediamo il dettaglio del mutant, i due riquadri gialli indicano le assegnazioni di "issues". In questo caso si può assegnare uno score di "4" alla severity;

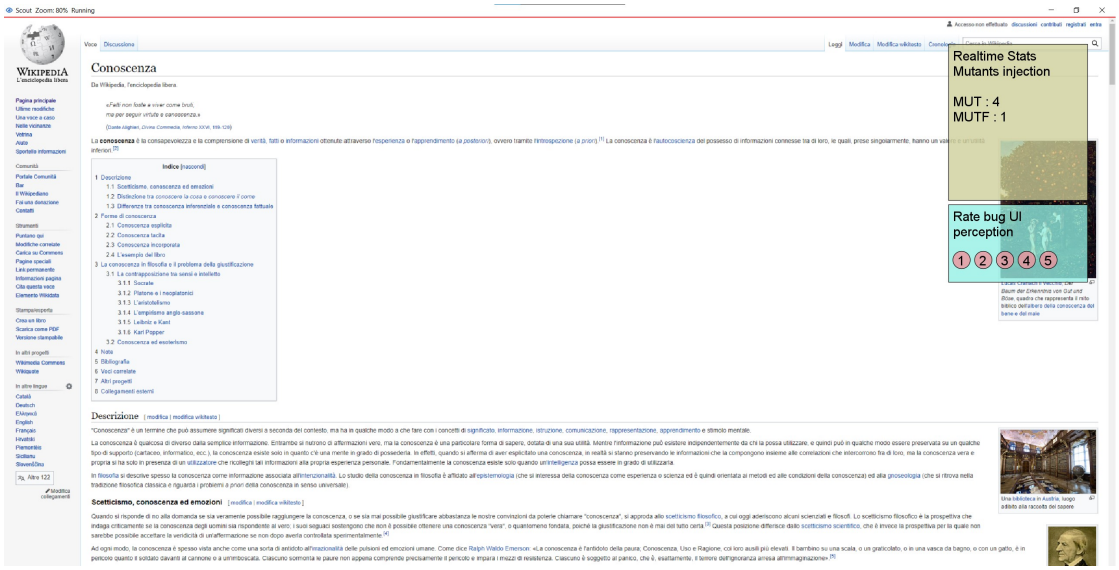


Figura 4.10: Esempio Sessione: mutant #4.

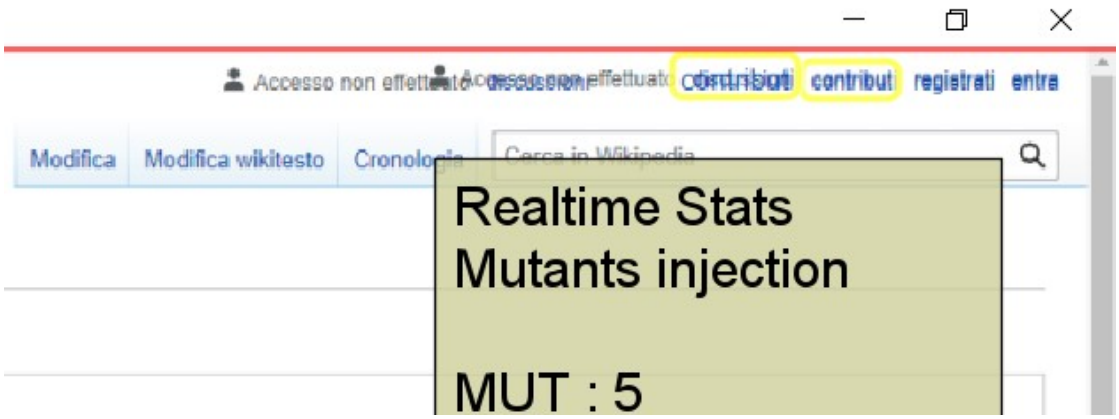


Figura 4.11: Esempio Sessione: mutant #5.

6. Cliccando sul link che porta alla pagina "it.wikipedia.org/wiki/Ingegneria_-dell%27informazione", avviene il cambio di posizione, in questo caso, di un elemento della lista del menù di navigazione sulla sinistra GUI. In figura 4.12 vediamo dove, l'elemento indicato dalla freccia, viene riposizionato. Essendo oltre lo scroll originario della pagina, lo screen è stato preso dal browser e non direttamente da Scout, perciò impossibile da trovare dentro l'Augmented GUI a causa del limite di Scout nell'effettuare lo scroll della pagina. Lo score sulla percezione viene anche qui ignorato per continuare con l'indicazione che non è stato trovato alcun mutant;

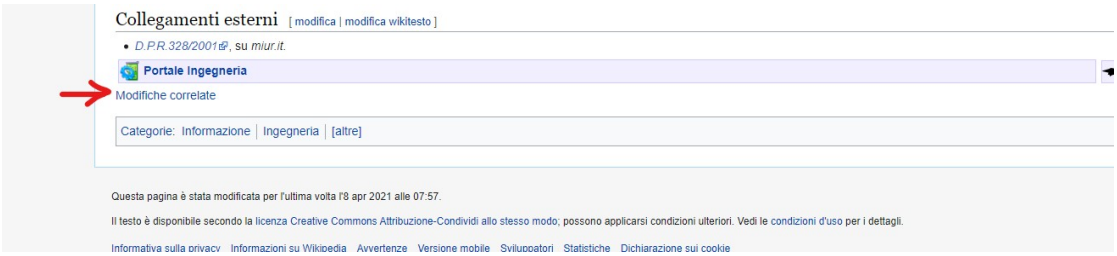


Figura 4.12: Esempio Sessione: mutant #6.

7. Cliccando sul link che porta alla pagina "it.wikipedia.org/wiki/Politecnico_di_Torino", avviene l'altro tipo di cambio di posizione. In figura 4.13 vediamo nel dettaglio il mutant dopo l'assegnazione dell'issue e l'avvenuta scoperta. In questo caso si può assegnare un "5" alla severity;

Figura 4.13: Esempio Sessione: mutant #7.

8. Cliccando sul link che porta alla pagina "it.wikipedia.org/wiki/Castello_del_Valentino", è possibile notare il ridimensionamento di uno dei widgets all'interno della sezione "Indice", in figura 4.14 il dettaglio. In questo caso viene cambiata la dimensione dell'elemento di lista "Esterno" che però mantiene il testo interno allineato in alto a sinistra del contenitore quindi presenta un grande spazio vuoto che fa da separatore con l'elemento successivo. Anche qui registrabile tramite apposita issue, lo score sulla severity di questo mutant può ad esempio essere "4";

Figura 4.14: Esempio Sessione: mutant #8.

9. Cliccando sul link che porta alla pagina "it.wikipedia.org/wiki/Torino", si nota il cambio di aspetto in uno degli elementi di tipo link della prima riga del contenitore di testo principale. In figura 4.15 vediamo il dettaglio. Anche qui registrabile tramite apposita issue, lo score sulla severity di questo mutant può ad esempio essere "3", essendo non grave come bug;

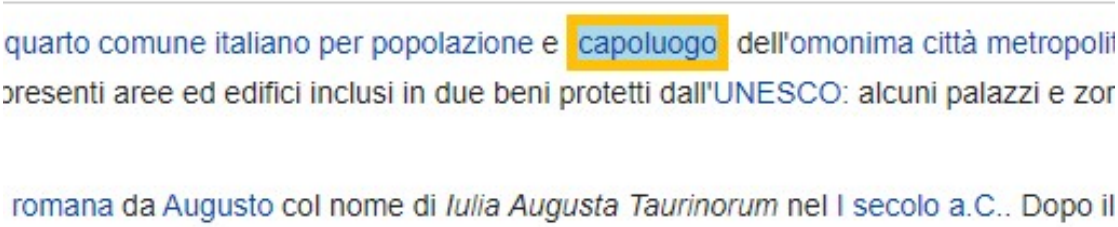


Figura 4.15: Esempio Sessione: mutant #9.

10. Cliccando sul link che porta alla pagina "it.wikipedia.org/wiki/Italia", viene cambiato il tipo di widget di uno degli elementi del menù di navigazione sulla sinistra della GUI. In figura 4.16 è segnato in rosso. Lo score sulla severity può essere ad esempio di "4".

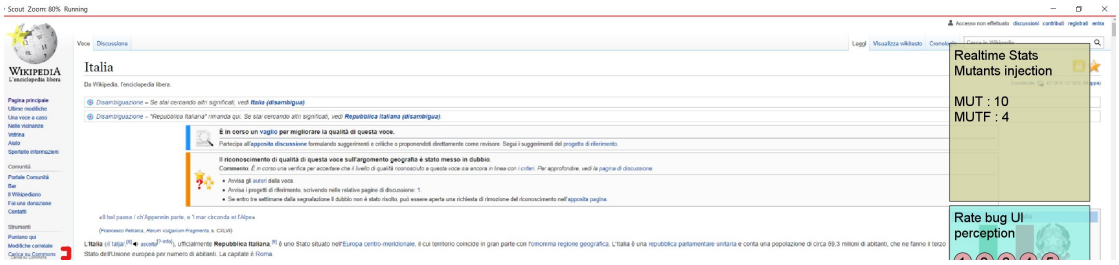


Figura 4.16: Esempio Sessione: mutant #10.

Al termine della sessione viene mostrato il riepilogo, con indicato, tra le altre cose, il numero di mutants trovati, figura 4.17. I dettagli di quanto raccolto, tramite i

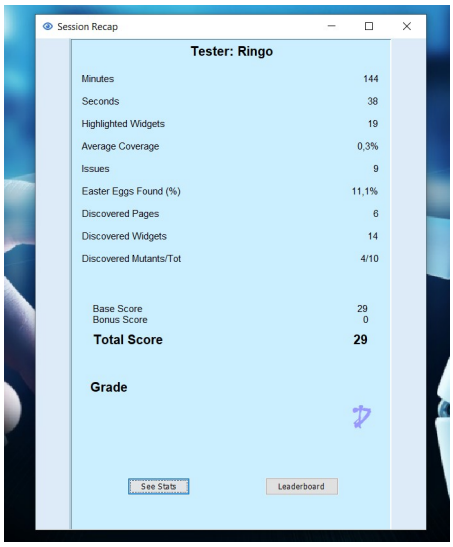


Figura 4.17: Esempio Sessione: schermata di riepilogo.

dati della sessione, dalle scoperte delle mutazioni e dai relativi score alla percezione della "severity" li possiamo analizzare in figura 4.18, dove, quest'ultimi, vengono raffigurati con il valore *m2* (barre rosse), mentre i mutants segnalati correttamente (barre blu) vengono rappresentati tramite il corrispondente valore *m1*. Si nota la contrapposizione tra le quattro segnalazioni corrette (indicate dal valore *true*) e le corrispettive valutazioni manuali che indicano l'effettiva percezione del bug rispetto a quanto era lo score (*mutationScore*) di base. Inoltre per i valori positivi di m2 contrassegnati come "non trovati" (indicati dal valore *false*), in questo caso i mutants *3, 5 e 10* (*duplicate, addDifferent e changeType*), si ricava che il tester ha possibilmente riscontrato errori nel segnalare correttamente il bug con il tool ma comunque ha percepito una variazione della GUI significativa tanto da portarlo ad

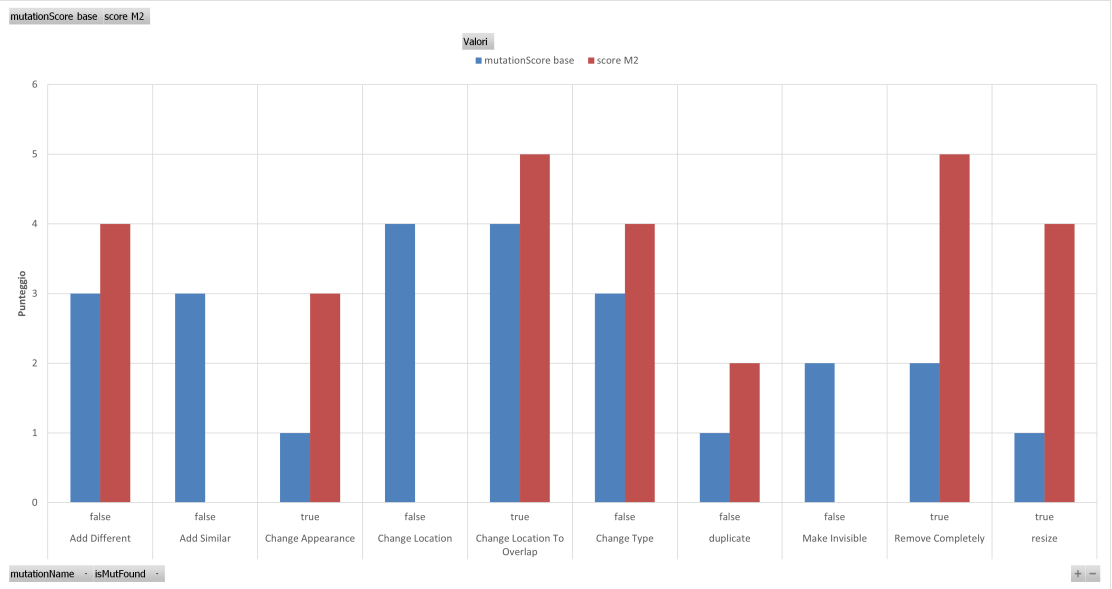


Figura 4.18: Esempio Sessione: statistiche sulle interazioni con i mutants.

assegnare uno score sulla severity. Per i restanti mutants non trovati, è presente il solo punteggio di base, senza quindi il voto assegnato, a causa della totale percezione negativa avuta nel notare e riscontrare un cambiamento significativo.

4.3 Risultati

Per valutare l'utilità del tool si è fatta un'analisi qualitativa sul corretto funzionamento delle mutazioni create nelle pagine delle applicazioni proposte, per questo motivo è stato chiesto al tester di esplorarle tutte e 3, cercando di interagire con più mutant possibili al fine di scoprirne un numero maggiore rispetto agli altri partecipanti e quindi dare un maggiore contributo alla validazione del lavoro svolto. In tabella 4.4 troviamo le domande del sondaggio riguardanti le interazioni con il tool.

Tabella 4.4: Domande del questionario finale, seconda parte.

ID	Domanda (Tipo)
2.1	Hai compreso il modo di utilizzare Scout e il suo contesto di utilizzo? (Likert)
2.2	Hai compreso l'utilità di inserire i mutants in contesto di GUI testing? (Likert)

Domande del questionario finale, seconda parte.

ID	Domanda (Tipo)
2.3	Hai trovato la Barra di Progresso utile a mostrare i tuoi progressi nel testing di una pagina? (Likert)
2.4	Conoscere l'esistenza di un Punteggio finale ti ha spinto a migliorare la tua performance? (Likert)
2.5	Quali tra i seguenti elementi di Gamification erano a te familiari prima dello svolgimento dell'esperimento? (Checkbox)
2.6	Quali tra i seguenti elementi di Gamification conoscevi ed hai utilizzato con consapevolezza? (Checkbox)
2.7	Quali tra i seguenti elementi di Gamification riterresti essenziali in una sessione di testing? (Checkbox)
2.8	Nel complesso, sei riuscito a trovare le sessioni di testing stimolanti relativamente al sito web testato? (Likert)
2.9	Nel complesso, sei riuscito a trovare i mutants funzionanti relativamente al sito web testato? (Likert)
2.10	Secondo te, quanti mutants sei riuscito a trovare relativamente al sito web testato? (Likert)
2.11	Nel complesso, quanto è stato facile trovare i mutants relativamente al sito web testato? (Likert)
2.12	Secondo te, quali mutants (tabella a pagina 2 dello script) sei riuscito a trovare con più facilità, relativamente al sito web testato? (Checkbox)
2.13	Quanto è stata utile l'interfaccia di "Overlay Stats" per tenere traccia dei mutants? (Likert)
2.14	È stata facilmente utilizzabile l'interfaccia di "Overlay Stats" per dare il voto alla visibilità dei mutants? (Likert)
2.15	Hai riscontrato dei problemi durante lo svolgimento dell'esperimento? Se sì, descrivili brevemente (Aperta)
2.16	Hai dei suggerimenti per migliorare il tool proposto? (Aperta)

Oltre alle domande a risposta multipla e quelle aperte, la maggior parte è stata impostata nella forma della scala Likert: il partecipante ha avuto la possibilità di selezionare una risposta da 1 ("poco o nulla", risposta fortemente negativa) a 5 ("molto", risposta fortemente positiva).

Inizialmente è stato chiesto il livello della comprensione dell'utilizzo di Scout e nell'utilità della GUI mutant injection. Dalle figure, 4.19 e 4.20, notiamo una situazione intermedia, dovuta probabilmente alla poca esperienza nel web testing ed in particolare nel GUI testing con la tecnica di AT prevista in Scout.

Successivamente, sono stati raccolti dei giudizi riguardo gli elementi di gamification presenti nel tool: una rilevante importanza ha avuto la barra di progresso, per

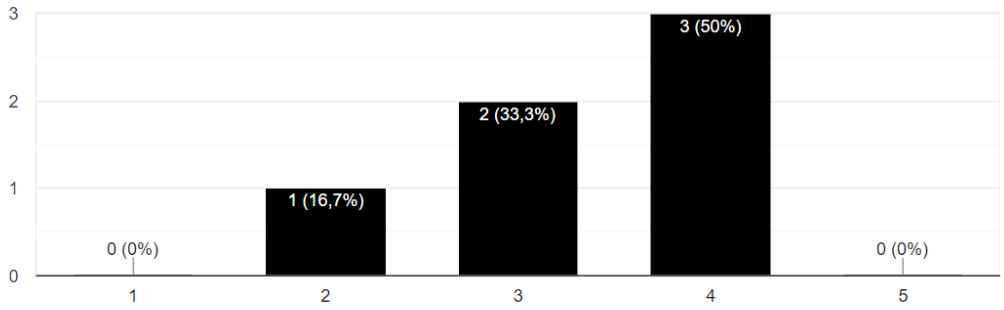


Figura 4.19: Sondaggio finale: comprensione dell'utilizzo di Scout.

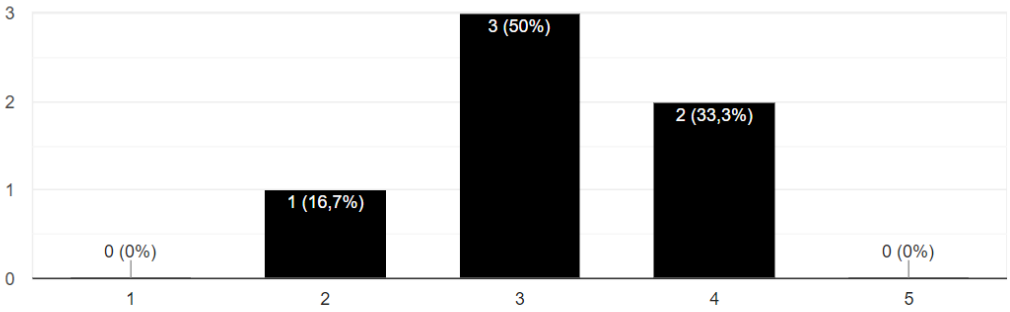


Figura 4.20: Sondaggio finale: comprensione dell'utilità dei mutants nel GUI testing.

tracciare le interazioni con i widgets, ma anche e soprattutto il punteggio finale, ciò che, come atteso, è il principale mezzo con il quale il partecipante è spinto a fare del suo meglio al fine di migliorare il posizionamento nella classifica finale. Inoltre, 3 tester hanno segnalato l'essenzialità del voto, usufruibile tramite il plugin dedicato, per segnalare i mutants, essendo a conoscenza dell'ulteriore aggiunta di punteggio ad avvenuta azione. Quanto detto è, ad esempio, osservabile dal grafico ottenuto dalle risposte alla domanda 2.7, figura 4.21. In generale, la presenza dell'interfaccia "Overlay Stats" è stata ritenuta utile, risposte alle domande 2.13 e 2.14, sia per assegnare il voto *bugSeverityScore* che per tenere traccia del numero delle mutazioni trovate e generate.

Analizzando invece i mutants, è stato inizialmente chiesto se fosse stato notato complessivamente funzionante l'inserimento, relativamente all'applicazione testata,

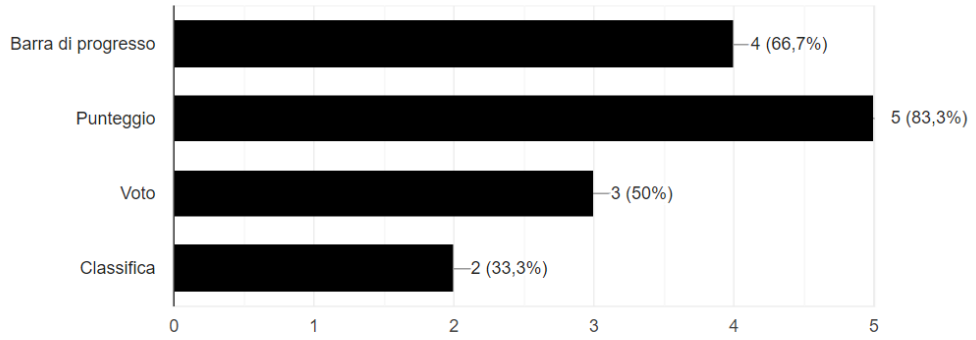


Figura 4.21: Sondaggio finale: elementi di gamification essenziali per il testing.

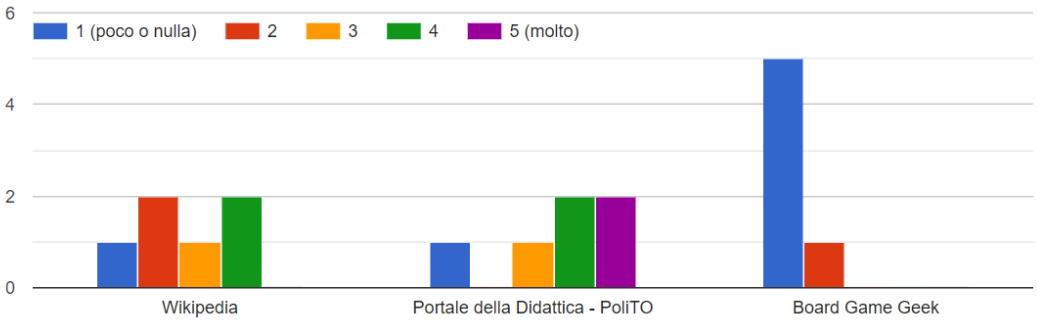


Figura 4.22: Sondaggio finale: facilità nel trovare i mutants.

delle mutazioni all'interno delle pagine visitate. Dalle risposte si può notare come l'applicazione web selezioanta come SUT abbia influito nell'esperimento, sia per quanto riguarda il reale funzionamento del mutant sia per la capacità del tester di trovarlo all'interno della pagina, variabile a causa della complessità e della quantità degli elementi del DOM. Tra i 3 siti provati, *Board Game Geek* è stato quello in cui i tester hanno avuto più difficoltà nell'individuare i mutants. Un ulteriore dato che evidenzia la complessità del compito di trovare i mutants generati si può riscontrare dalle risposte alla domanda 2.11, figura 4.22. Ancora una volta notiamo come in *Board Game Geek* la ricerca dei mutant non sia stata per niente semplice, mentre per gli altri due domini si nota una situazione migliore, soprattutto in *PoliTO* dove quattro tester ha dato una valutazione tra 4 e 5. Un ultimo dato ha permesso di valutare la percezione del partecipante nei confronti della tipologia di mutants

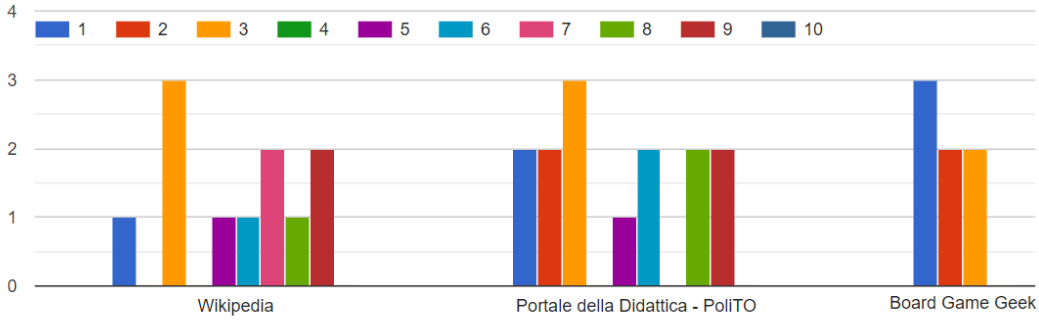


Figura 4.23: Sondaggio finale: tipologia di mutants trovati, secondo il tester.

trovati, riferiti a quelli in tabella 3.3. In figura 4.23 vediamo un confronto tra le tre diverse applicazioni web. In generale, la "duplicazione" è stata la mutazione percepita con più facilità. Rispecchiando quanto visto in precedenza, i siti di *Wikipedia* e *PoliTO* sono stati quelli dove una maggiore varietà di mutazioni è stata notata dal tester.

Nel paragrafo successivo si mettono a confronto, ad esempio, il numero dei mutants trovati, secondo il tester, complessivamente per tutte le applicazioni (domanda 2.10), con i dati reali, derivanti dai log delle sessioni.

4.3.1 Statistiche dalle sessioni

Il partecipante doveva, alla fine del test, oltre a rispondere al sondaggio, consegnare i files con i dati sulle statistiche relative alle interazioni avute con i mutants all'interno di ogni pagina navigata.

Un confronto immediato si può fare analizzando, per ogni tester

- il totale delle mutazioni generate dal tool con le quali si ha interagito,
- il totale delle mutazioni segnalate attraverso il plugin per assegnare il *bugSeverityScore*,
- il totale delle mutazioni segnalate riconosciute dal sistema come correttamente individuate,
- il totale delle mutazioni trovate secondo l'opinione del tester, quest'ultimo dato corrisponde per ogni tester alla risposta alla domanda 2.10 del sondaggio.

Per la rappresentazione di tali dati sono stati aggregati i complessivi delle tre applicazioni testate. Per primo, in figura 4.24 vediamo il confronto tra i mutants

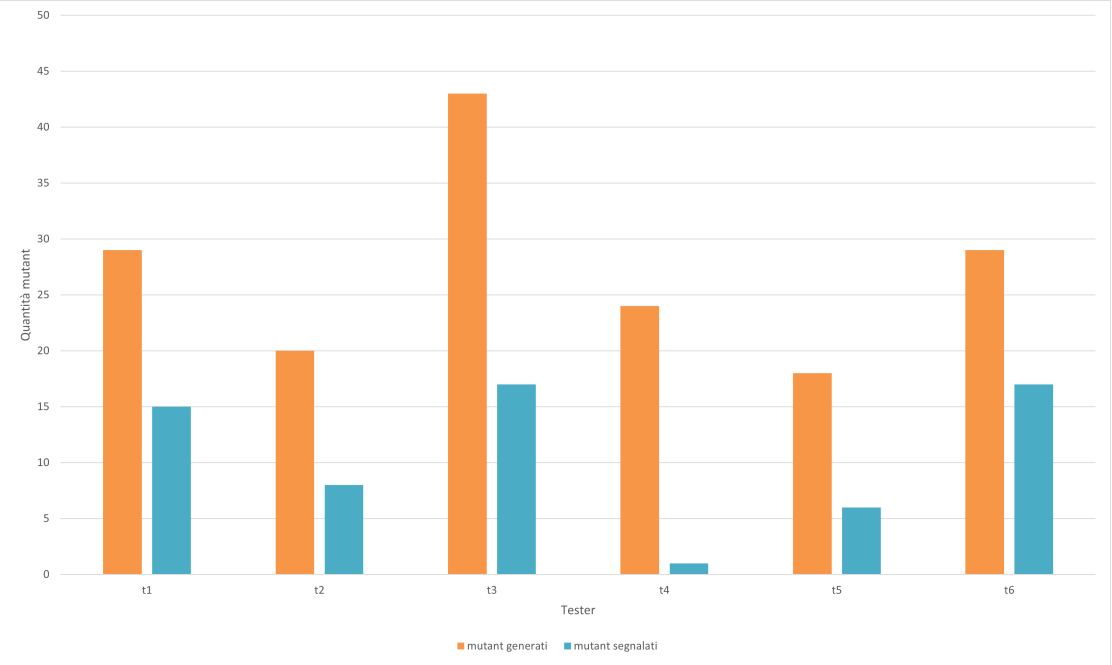


Figura 4.24: Dati delle sessioni sui mutants, confronto tra tester: generati vs segnalati.

generati (barre arancioni) e quelli segnalati tramite il voto (barre azzurre). Si nota come, in media, le percezioni segnalate siano state la metà dei bug effettivamente presenti, possibilmente sia per la difficoltà nel trovarli che per la reale mancanza di qualcosa di visivamente diverso nelle pagine visitate. Un tester (t4) si discosta da questi valori, ad indicare una maggiore difficoltà di interazione con le mutazioni ma probabilmente anche con il plugin del voto che, a causa dei rallentamenti di Scout, può non aver fornito un corretto funzionamento.

In figura 4.25 vediamo invece il confronto tra le precedenti segnalazioni (barre azzurre) e quelle indicate come corrette dal tool (barre viola), quindi un vero e proprio sottoinsieme del precedente. Ricordando che le segnalazioni corrette sono quelle dove è stata assegnata la issue al widget corrispondente al mutant oppure, per le mutazioni che nascondono/rimuovono l'elemento, i click del mouse sono stati eseguiti nella zona dove fosse presente in precedenza il widget, notiamo come circa la metà delle mutazioni o poco più, in media tra i partecipanti, sia stata scoperta in modo corretto. Per il tester t4 l'unico mutant segnalato corrisponde, in questo caso, a quello effettivamente trovato.

In figura 4.26 troviamo l'ultimo confronto, tra i precedenti mutants correttamente segnalati (barre viola) e quelli indicati dal sondaggio (barre marroni). La domanda del sondaggio (2.10) prevedeva di indicare per ogni applicazione web

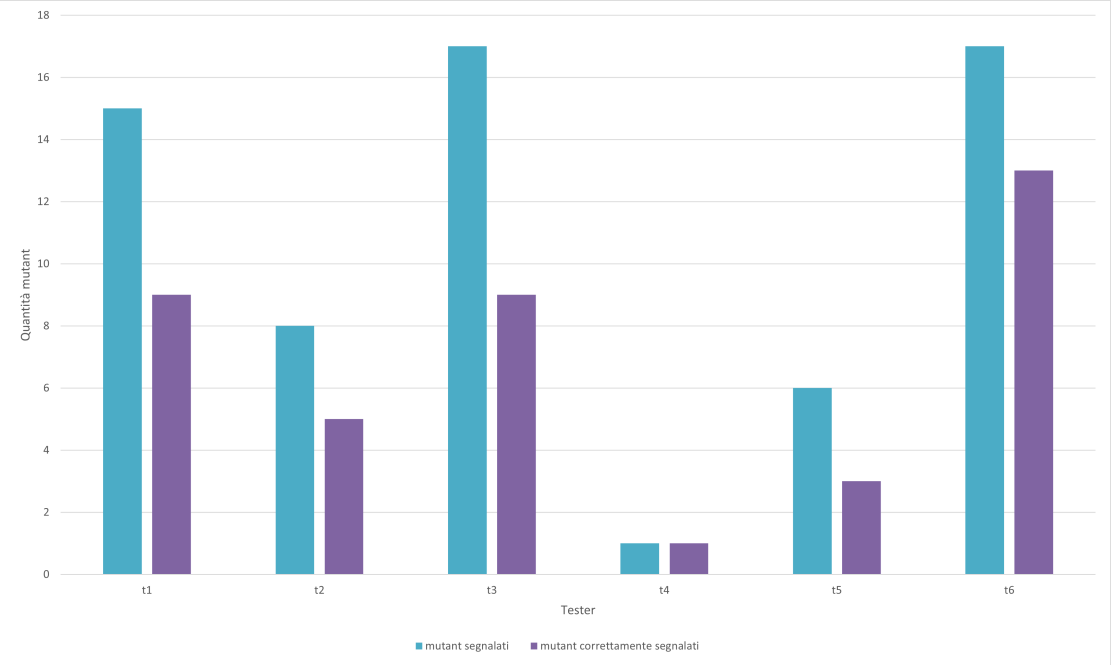


Figura 4.25: Dati delle sessioni sui mutants, confronto tra tester: segnalati vs corretti.

quante mutazioni il partecipante credesse di aver trovato, qui vengono raggruppate. Rispetto ai precedenti confronti, questo dato è indipendente, essendo le risposte non legate direttamente ai dati raccolti dalle sessioni di testing. Vediamo come, in generale, il tester abbia indicato di aver trovato più mutants di quelli effettivamente registrati come tali dal tool ma in numero comunque inferiore a quelli segnalati in totale. Per un tester (t6) quelli indicati sono corrisposti a ciò che era stato segnalato correttamente dentro Scout. Tra tutti, il dato più significativo ai fini della validazione è sicuramente il totale del numero di mutants correttamente trovati, che considerando una percentuale, tra tutti i tester, rispetto a quanto segnalato, risulta del **63%**. Un riepilogo di queste statistiche si trova in tabella 4.5.

Di seguito si analizzano invece, suddivisi per applicazione web, gli impatti che hanno avuto le diverse tipologie di mutants, analogamente al caso di esempio in figura 4.18. Ricordando il significato delle metriche, *m1* rappresenta il punteggio di base associato alla difficoltà prevista (indicato "come mutationScore di base") mentre *m2* il punteggio di *bugSeverityScore* assegnato dal tester. Un importante indicatore è la suddivisione dei seguenti grafici secondo il valore di *true/false* assegnato per ogni tipologia di mutazione ad indicarne l'eventuale scoperta: per ogni applicazione web troviamo quindi due schemi, uno per le tipologie di mutants trovate e l'altro per quelle non trovate, per ognuna delle quali viene confrontato lo

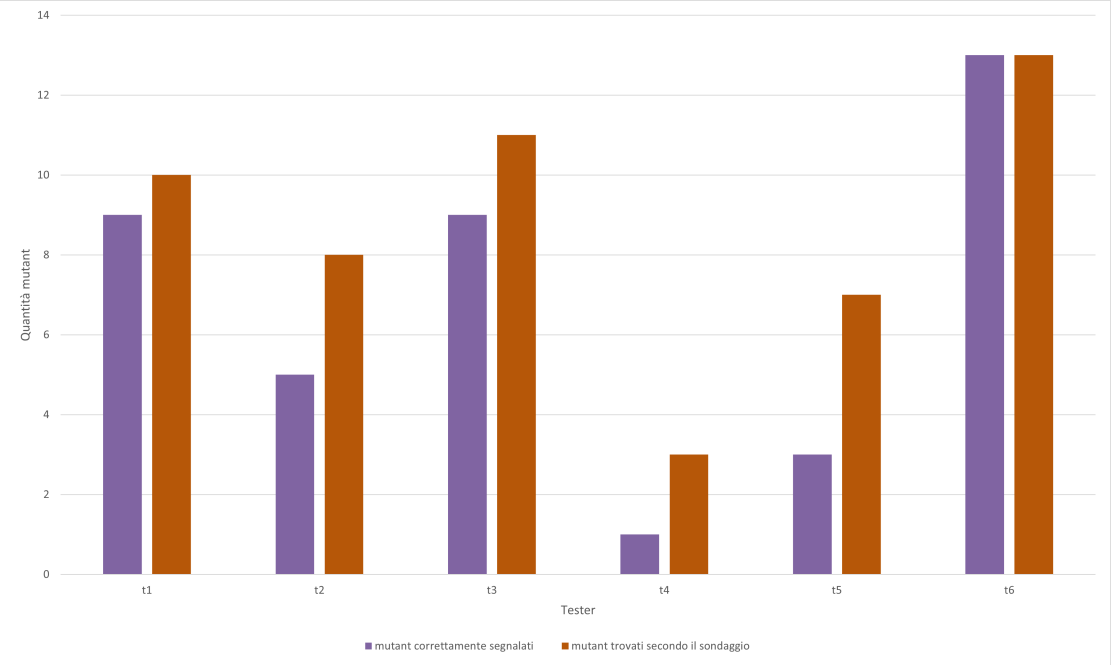


Figura 4.26: Dati delle sessioni sui mutants, confronto tra tester: corretti vs sondaggio.

Tester	Mutant Generati	Mutant Segnalati	Mutant Correttamente Segnalati	Mutant Trovati secondo il Sondaggio
t1	29	15	9	10
t2	20	8	5	8
t3	43	17	9	11
t4	24	1	1	3
t5	18	6	3	7
t6	29	17	13	13
Totale	163	64	40	52

Tabella 4.5: Statistiche complessive delle tre applicazioni web, divise per tester

score di base con quello assegnato manualmente dai partecipanti.

Cominciando da *Wikipedia*, in figura 4.27, notiamo subito un alto numero di tipi diversi di mutants scoperti, sono rimasti esclusi 3 di essi, *addSimilar*, *addDifferent* e *changeType*, figura 4.28, correlati dalle categorie del DOM e non casualmente non riconosciuti, probabilmente a causa dell'alto grado di complessità che presenta

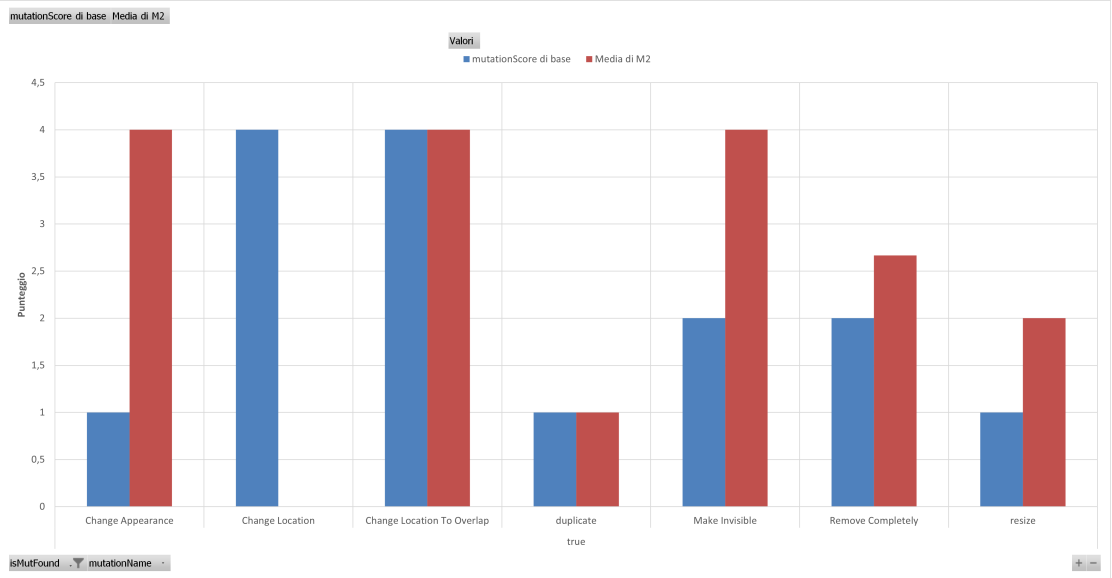


Figura 4.27: Dati delle sessioni sui mutants trovati: Wikipedia.

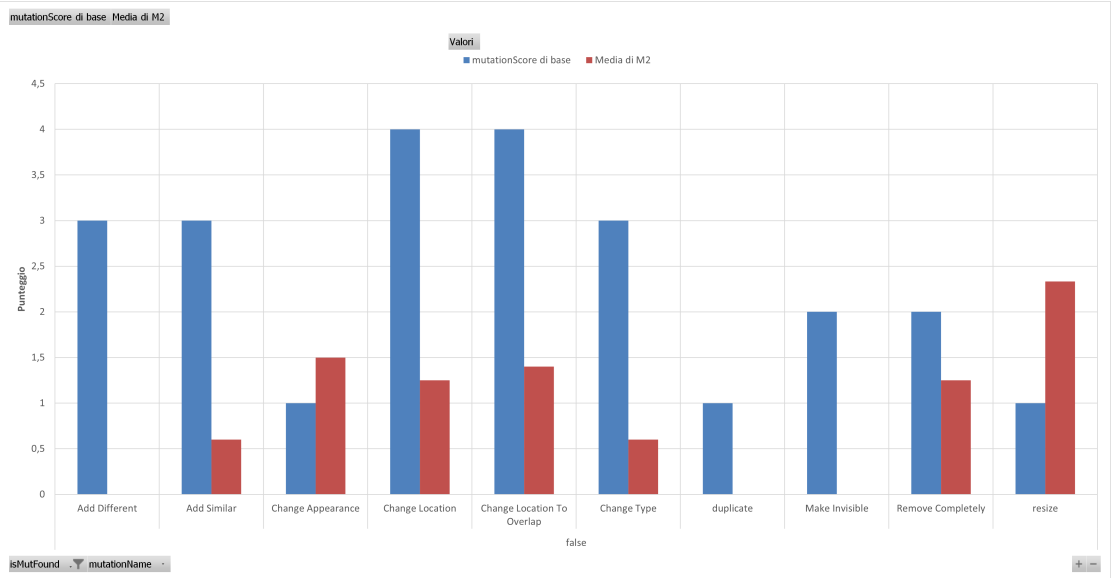


Figura 4.28: Dati delle sessioni sui mutants NON trovati: Wikipedia.

questa applicazione web nelle sue pagine interne, dense di widgets puramente testuali che possono fuorviare e complicare la ricerca del mutant. Per i mutants non trovati, comprendenti anche i tre mai scoperti, notiamo anche che il valore di *m2* è talvolta presente, il che indica che comunque qualche cambiamento nella pagina, seppur non evidente, sia stato percepito.

Per gli altri 7 mutants abbiamo una situazione diversa:

- *changeAppearance*, c'è stato chi l'ha correttamente trovato, segnalando la percezione di un evidente bug. Chi non l'ha direttamente trovato, ha comunque segnalato la presenza di qualcosa, dovuto magari all'impossibilità di segnalazione durante la navigazione;
- *changeLocation*, nonostante fosse complesso da trovare, c'è stato chi l'ha trovato dimenticandosi però, probabilmente, di segnalare il voto sulla percezione, essendo nulli i valori. Chi non l'ha trovato ha invece segnalato la presenza di qualcosa, dovuto sempre alla probabile impossibilità di segnalazione durante la navigazione;
- *changeLocationToOverlap*, qui tutti i partecipanti che l'hanno trovato hanno anche valutato come evidente il bug generato nella pagina, corrispondentemente a quanto previsto dallo score di base. Chi non l'ha trovato ha comunque segnalato la presenza di qualcosa di non molto chiaro;
- *duplicate*, qui chi non l'ha trovato non ne ha nemmeno segnalato la percezione, indice possibilmente della difficoltà nella ricerca. Chi l'ha trovato ha anche votato coerentemente a quanto previsto, come bug non grave;
- *makeInvisible*, anche qui non è stato segnalato nulla da chi non l'ha trovato. Chi l'ha trovato ha segnalato l'evidente presenza di qualcosa che non andasse nella pagina;
- *removeCompletely*, qui siamo in una via di mezzo, considerando che questo mutant viene generato nella homepage, essendo il primo, le considerazioni sulla complessità testuale valgono in parte rispetto alle altre pagine. Troviamo una situazione intermedia, dove sia chi l'ha trovato che non ha segnalato di aver percepito qualcosa che non andasse, in questo caso, un widget mancante;
- *resize*, anche qui troviamo una situazione intermedia simile alla precedente, dove però chi non l'ha trovato ha segnalato di aver percepito qualcosa di poco più grave.

Continuando con *PoliTO*, figura 4.29, notiamo che di fatto solo un mutant non è stato mai trovato, *changeType*, anche se ne ritroviamo una minima valutazione sulla severità, figura 4.30, fornita possibilmente anche erroneamente ma indicativa della volontà di approfondire l'analisi e l'esplorazione da parte del tester. Per gli altri 9 mutants troviamo la seguente situazione:

- *addDifferent* e *addSimilar*, per entrambi una situazione simile, chi non l'ha trovato ha percepito poco o nulla mentre gli altri hanno segnalato un leggero bug;

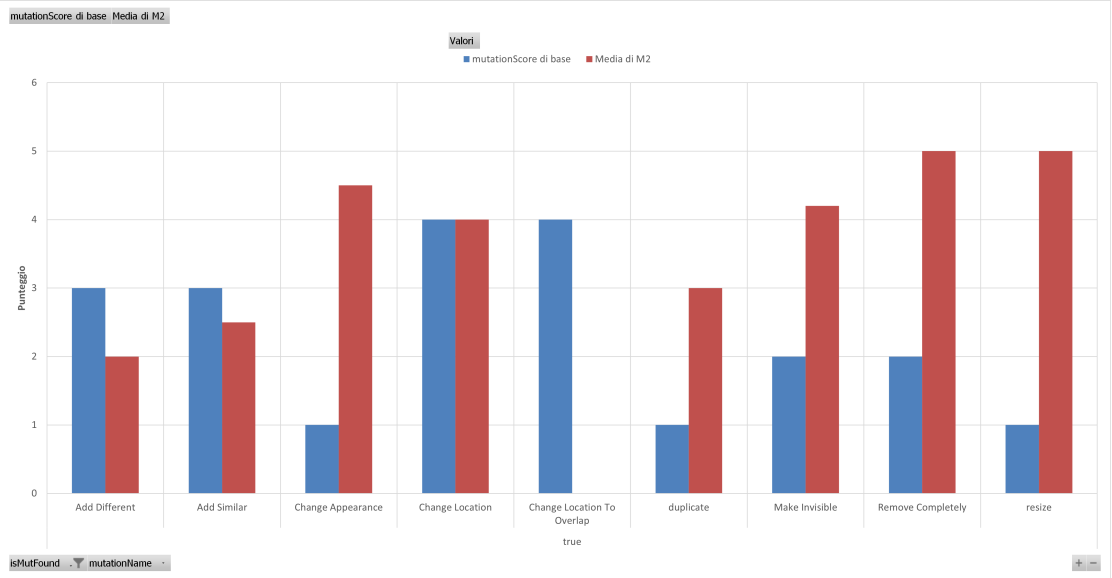


Figura 4.29: Dati delle sessioni sui mutants trovati: PoliTO.

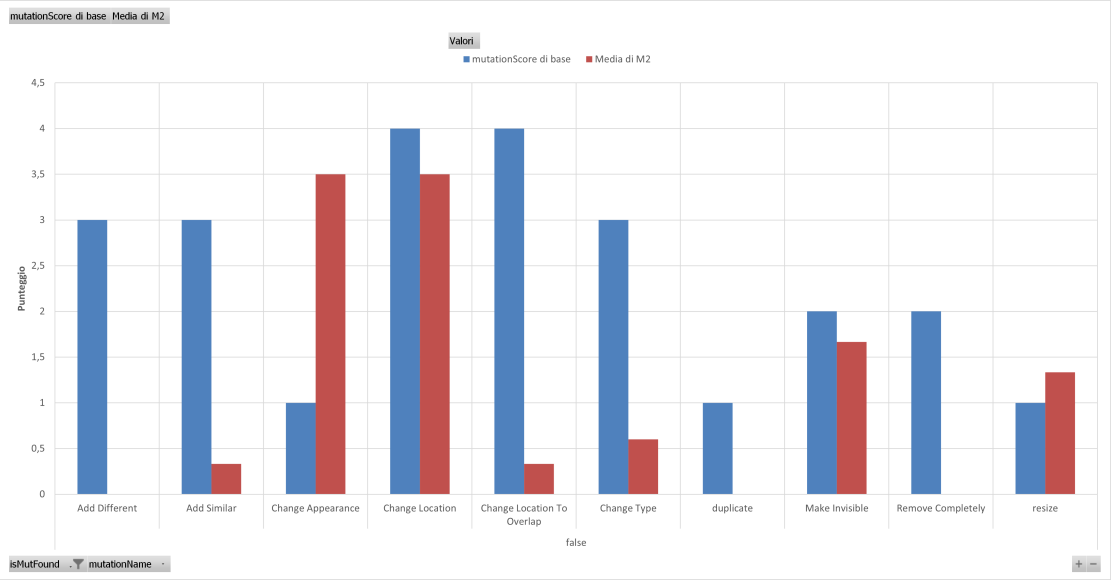


Figura 4.30: Dati delle sessioni sui mutants NON trovati: PoliTO.

- *changeAppearance*, qui, sia per chi l'ha trovato che non, è stato percepito un più grave, rispetto alla media, impatto del bug nella GUI;

- *changeLocation*, anche qui per tutti i tester è stato percepito qualcosa di abbastanza evidente nella pagina. Chi ha trovato il mutant ha votato coerentemente

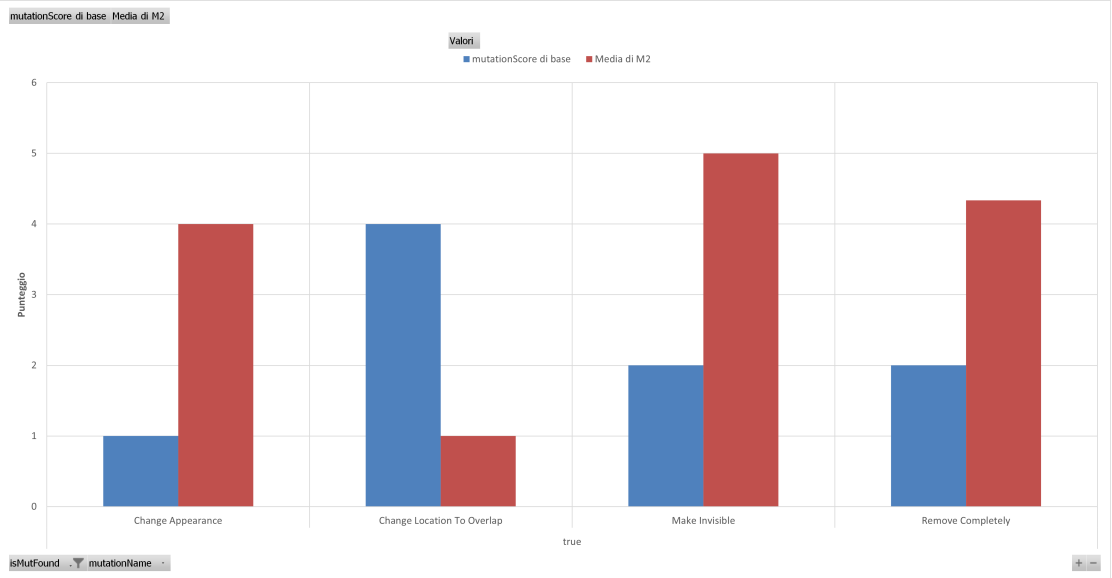


Figura 4.31: Dati delle sessioni sui mutants trovati: Board Game Geek.

allo score di base;

- *changeLocationToOverlap*, in parte è stato trovato ma comunque non segnalato da parte del tester. Risulta percepito minimamente da chi non l'ha trovato;
- *duplicate*, chi ha trovato questo mutant ha percepito un deciso cambiamento nella pagina. Chi no, ha coerentemente deciso di non votare per alcuna severità;
- *makeInvisible*, è stato segnalato come abbastanza evidente da chi l'ha trovato e appena percepito da chi non l'ha trovato correttamente;
- *removeCompletely*, questo mutant è stato percepito come molto grave ed evidente da tutti quelli che l'hanno correttamente trovato mentre non segnalato per gli altri;
- *resize*, anche questo è stato percepito come molto grave ed evidente da tutti quelli che l'hanno correttamente trovato. Gli altri hanno segnalato giusto un minimo cambiamento nella pagina.

Infine, in *BoardGameGeek*, che ricordiamo essere quello sommariamente più complesso col quale si è interagito, nelle figure 4.31 e 4.32, troviamo una situazione piuttosto diversa rispetto ai precedenti siti. Non sono mai stati trovati 6 sui 10 dei totali mutants generati, a causa probabilmente della maggiore complessità dei widgets in termini di disposizione e tipologia oltre che solo in quantità come

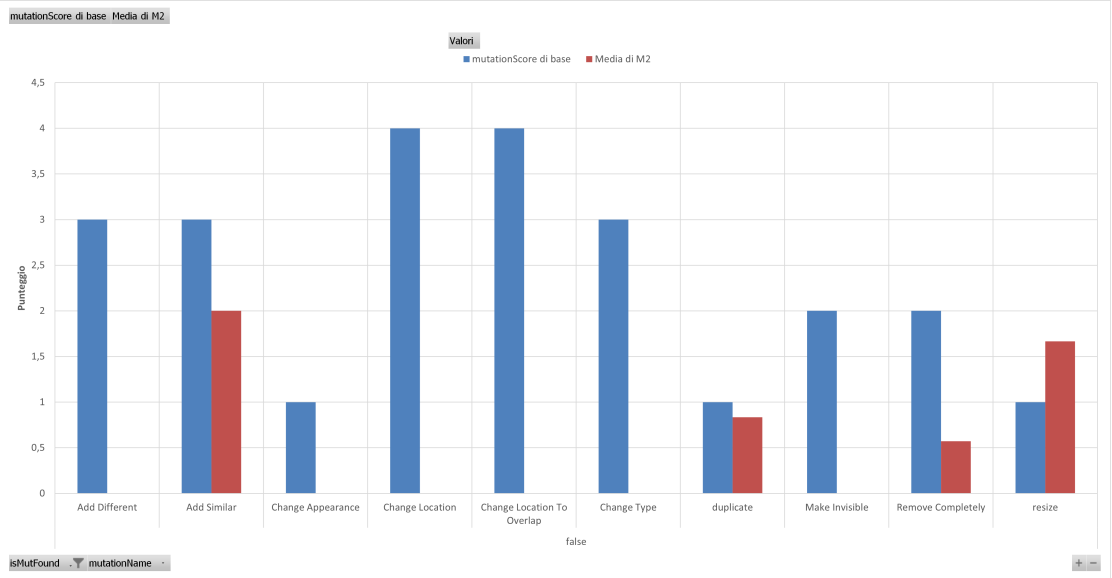


Figura 4.32: Dati delle sessioni sui mutants NON trovati: Board Game Geek.

per *Wikipedia*. In particolare, non sono stati trovati *addSimilar*, *addDifferent*, *changeLocation*, *changeType*, *duplicate* e *resize*. Tuttavia, per *addSimilar*, *duplicate* e *resize*, i tester hanno percepito un leggero bug nella pagina che avrebbero voluto ispezionare ed esplorare meglio. Per gli altri 4 troviamo la seguente situazione:

- *changeAppearance*, i tester che l'hanno trovato hanno votato per una percezione del bug abbastanza elevata malgrado non lo fosse di base. Gli altri non hanno percepito nulla;
- *changeLocationToOverlap*, chi l'ha trovato ha segnalato di aver percepito poco o nulla nella pagina, tale da approfondirne meglio l'esplorazione. Gli altri non hanno percepito nulla;
- *makeInvisible*, quelli che l'hanno trovato hanno percepito un grave ed evidente bug forse dovuto all'impossibilità di interagire con dei widgets molto importanti nella pagina. Gli altri non hanno percepito nulla;
- *removeCompletely*, è stato percepito come abbastanza evidente per chi l'ha trovato, probabilmente per le stesse ragioni precedenti. Gli altri hanno percepito qualcosa di non tanto grave da essere considerato.

Per una panoramica generale, analizziamo nel complesso i risultati di tutti e tre i dati precedenti, raccolti nei grafici di figura 4.33 e figura 4.34. Dai singoli grafici precedenti notiamo che nessun mutant è stato sempre trovato, in ogni occasione in cui è stato correttamente generato e inserito dal tool. Dal grafico generale

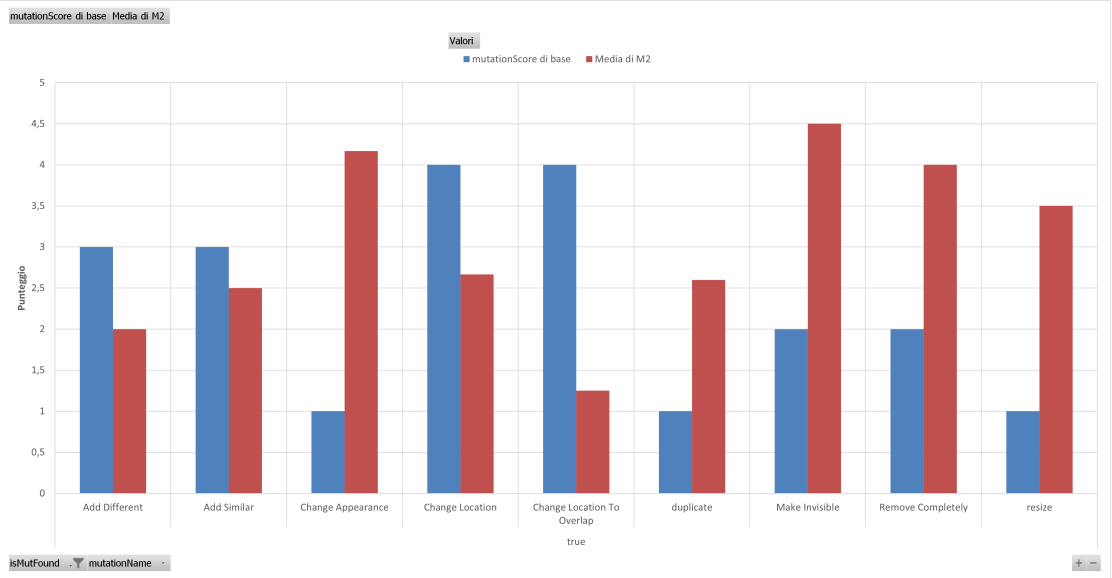


Figura 4.33: Dati delle sessioni sui mutants trovati: panoramica generale.

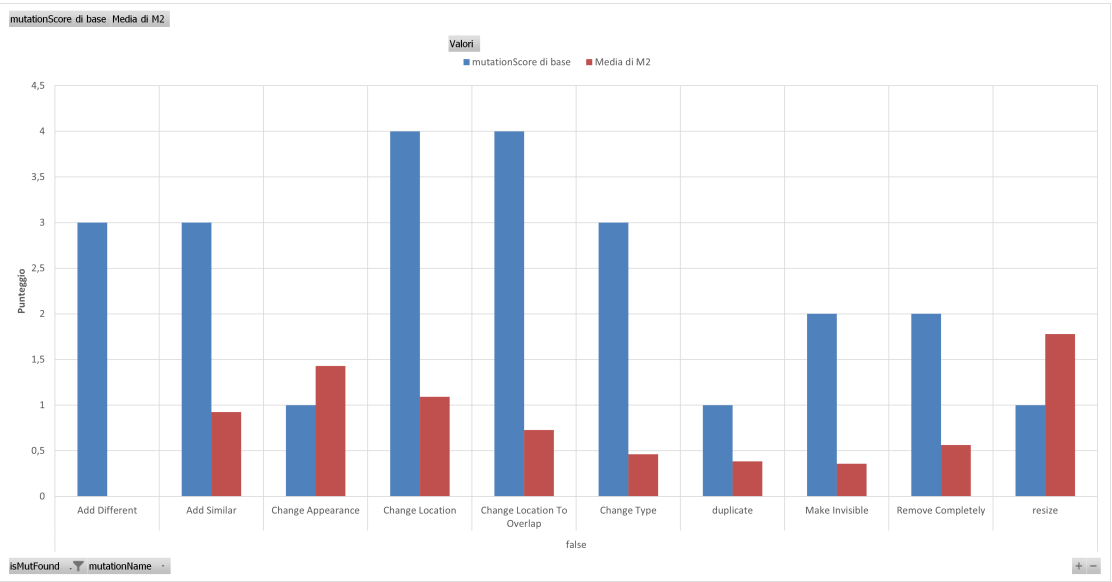


Figura 4.34: Dati delle sessioni sui mutants NON trovati: panoramica generale.

notiamo che l'unico mutant mai trovato è stato *changeType*, anche se minimamente percepito e segnalato tramite il voto della severità. Per gli altri, se analizziamo i mutants trovati, c'è stata un'alta percezione da parte del tester che ha votato con un elevato punteggio maggiormente quelli che in comune prevedevano la modifica dell'attributo *style* del widget, in particolare *changeAppearance*, *makeInvisible*,

removeCompletely e *resize*. Per quelli non trovati la segnalazione sulla percezione si è mantenuta bassa, ad indicare di aver notato qualcosa di non grave nella pagina ma comunque la volontà di esplorarla più a fondo o, probabilmente, è stata indicativa del fatto che non sono stati in grado di assegnare correttamente la issue anche a causa di segnalazioni sui widgets annidati che non ne hanno permesso l'esplorazione nell'Augmented Layer (vedi limiti nel paragrafo 5.1) o per altri bug di cui può essere affetto il tool.

4.3.2 Considerazioni

Nel complesso, in riferimento anche alle risposte ottenute dalla domanda 2.8, le sessioni sono risultate stimolanti in base all'applicazione web testata a causa del suo contenuto in termini di complessità che ha portato il partecipante ad interagire, in modo differente, con una diversa quantità di mutazioni. In particolare, *PoliTO* è stata quella più stimolante probabilmente per la ridotta presenza di widgets che ha permesso una più facile ricerca del mutant, mentre *BoardGameGeek* è stata la meno stimolante a causa dell'elevata complessità degli elementi presenti, a volte non statici come per la maggior parte di quelli presenti nelle altre due applicazioni web.

Le ultime due domande del sondaggio fornito ai volontari prevedevano, tramite risposta aperta, di descrivere gli eventuali problemi riscontrati durante le sessioni di testing e di suggerire dei miglioramenti generali per il tool. Di seguito vengono mostrati i più rilevanti, riformulati non testualmente:

- *(problema)*, qualche volta i mutants consapevolmente individuati non sono stati segnalati come trovati dal sistema. Scout spesso si bloccava non caricando le nuove pagine;

- *(problema)*, l'assegnazione di issue sui div che contengono link non sempre funziona;

- *(problema)*, alcune pagine visitate non mostravano correttamente i mutants e non veniva aggiornato il plugin di Overlay Stats, non mostrando neanche il riquadro per il voto. Scout lento in generale;

- *(problema)*, sul sito di PoliTO c'erano molti link che portavano all'apertura di una nuova scheda del browser, rendendo impossibile la navigazione alla nuova pagina. Non poter scorrere la pagina ha fatto dubitare talvolta che i mutants si trovassero nella parte nascosta della pagina. Alcuni mutants erano difficili da segnalare, come la duplicazione ad esempio di una lista opzioni che portava alla sovrapposizione dei due widgets e che lasciava il dubbio di dover assegnare una issue a tutti gli elementi duplicati o all'intera lista, non segnalabile però;

- *(problema)*, talvolta le pagine hanno presentato elementi non corretti nonostante non fossero considerati mutants. Durante la terza sessione Scout si è bloccato non permettendone la corretta terminazione;
- *(suggerimento)*, l'assegnazione del bugSeverityScore dovrebbe essere bloccante per la prosecuzione del test, con un qualche feedback, in modo che l'utente possa avere consapevolezza dell'azione svolta;
- *(suggerimento)*, aggiungere la possibilità di rimuovere un issue assegnato in precedenza per permettere la segnalazione di elementi internamente annidati. Migliorare le prestazioni generali del tool;
- *(suggerimento)*, sarebbe più utile far testare all'utente dei siti web con i quali ha già dimestichezza, altrimenti non sapendo come è fatta l'interfaccia grafica risulta molto difficile trovare delle modifiche, soprattutto quelle che prevedono un cambiamento minimo nella pagina. Aiuterebbe avere a disposizione degli esempi di mutants per avere idea di cosa cercare durante il test, accessibili magari da un plugin aggiuntivo.

In generale, sono state riscontrate delle difficoltà generali nell'utilizzo del tool che hanno riguardato principalmente i limiti nelle prestazioni di Scout ma anche alcune interazioni con i mutants offerte dall'*Augmented Layer*. Le maggiori limitazioni verranno discusse nel capitolo successivo insieme con i suggerimenti e altre proposte per i miglioramenti da applicare in sviluppi futuri.

Capitolo 5

Conclusioni

Nonostante il campione ristretto di partecipanti al quale è stata sottoposta la sperimentazione, il lavoro svolto ha prodotto dei risultati interessanti dal punto di vista dei feedback ottenuti durante le sessioni di testing, sia per quanto riguarda i sondaggi sia per i dati realmente raccolti da Scout. Sono stati evidenziati i vantaggi dell'inclusione degli elementi di Gamification ma anche gli svantaggi e le limitazioni sofferte dal tool, ad esempio per quanto riguarda la generazione di alcune mutazioni. In generale, la scoperta dei mutants inseriti ha portato degli esiti che sono stati positivi ed incoraggianti.

5.1 Limitazioni

Quanto emerso, durante lo sviluppo del tool, ha portato ad identificare alcuni limiti di applicabilità presenti sia tra le implementazioni delle varie mutazioni sia nel funzionamento dell'AT di Scout.

Per quanto riguarda i mutants, la scelta delle tipologie, tabella 3.3, è stata un sottoinsieme di quelle prese da [16, Alegroth et al.] e [18, Oliveira et al.], tabella 2.3, per un motivo ben preciso: le tipologie che non sono state considerate riguardano l'interazione con dipendenze esterne al solo sorgente HTML della pagina oppure l'interazione con le logiche del codice *javascript*, cose, quindi, non manipolabili direttamente tramite le stringhe all'interno del sorgente. Ad esempio, con le modifiche ai *listeners* il tester avrebbe potuto interagire diversamente con pulsanti o collegamenti nelle pagine e percepire diversamente il bug, in questo caso non puramente visuale.

Un altro limite lo troviamo nella scelta delle categorie del DOM, prese da [32, Mozilla], che ha influenzato il funzionamento di 3 dei 10 mutants implementati (*addSimilar*, *addDifferent*, *changeType*). Ciò che è successo è che talvolta non è stato propriamente identificato correttamente il widget appartenente all'insieme

scelto, o perché non particolarmente differente dal widget originario o perché non riconoscibile dall'Augmented Layer di Scout. Considerando anche che la scelta della categoria dell'elemento fosse puramente casuale tra tante possibilità, è spesso capitato uno dei casi precedenti, ad esempio ** che vengono cambiati in *<p>*, che non porta ad una evidente modifica in un widget, specialmente in una pagina con molti elementi. Inoltre, mancavano tra le categorie, elementi come *<li>*, importanti per applicazioni come Wikipedia, dove la presenza di liste è molto elevata: è stato necessario perciò aggiungere un altro insieme che rappresentasse widgets usati tipicamente nelle liste o nelle tabelle per elencare del testo.

L'iniezione del mutant generato nella pagina è avvenuto mediante il WebDriver di Selenium che ha permesso di eseguire un comando di script per generare da zero il sorgente HTML sovrascrivendolo a quello in esecuzione nel browser, *document.write()*. Ciò ha portato all'utilizzo di Microsoft Edge in sfavore di Firefox poiché è un'operazione non permessa per problemi di sicurezza in quest'ultimo e di conseguenza ha portato in Scout ad utilizzare proprio Edge che, a sua volta, non permette di far funzionare correttamente la modalità *headless*, ovvero la possibilità di nascondere la finestra del browser automatizzato dal WebDriver per lasciare solo la GUI della finestra di Scout, cosa che con Firefox invece è permessa. Nelle istruzioni fornite ai partecipanti è servito specificare questo dettaglio per non compromette l'esperienza di testing. Il comando per sovrascrivere il sorgente ha talvolta portato ulteriori modifiche, non previste e ancora ignote e non indagate, che sono state giustamente percepite come mutazioni nella pagina: ad esempio, in Wikipedia, il corpo delle informazioni centrali viene a volte ripetuto immediatamente dopo la pagina originaria, come fosse "appeso" al codice precedente, facendo terminare il mutant nella parte inferiore della pagina web, raggiungibile tramite scorrimento. Quest'ultima azione in Scout è un limite noto, poiché non è possibile effettuare lo scroll della GUI, il che porta a dover interagire necessariamente con i widgets visibili nella parte non nascosta dallo scroll: il mutant, seguendo questo principio viene generato solo per i tutti i widgets in questa parte iniziale di pagina, il che limita le possibilità di funzionamento del tool oltre che quelle di esplorazione del tester, come è stato segnalato da alcuni attraverso il dubbio di ricercare la mutazione nella parte non raggiungibile. In realtà, un mutant in particolare, *changeLocation*, può far verificare questa situazione, essendo anche qui casuale il modo in cui viene selezionata una nuova posizione per il widget scelto, per cui se dovesse essere riposizionato verso il fondo della pagina, se non raggiungibile, il mutant sarebbe impossibile da segnalare.

Dal sondaggio di valutazione finale è stato segnalato anche un limite di prestazioni di Scout che spesso ha reso impossibile, o comunque molto complicato, portare correttamente a termine la sessione di testing, in relazione al tipo di pagina navigata, al suo contenuto più o meno complesso. Altra considerazione è arrivata per la navigazione di alcuni tipi di link, in particolare quelli che prevedono l'apertura di

una nuova scheda del browser: in queste situazioni, Scout genera in modo corretto la mutazione e tutte gli altri metadati relativi ai widgets dell'intera pagina navigata, solo che il focus rimane sulla scheda d'origine, il che rende impossibile continuare la corretta interazione con gli elementi.

Per quanto concerne l'esperienza di Gamification, un elemento caratterizzante è stato il punteggio e la relativa classifica finale: non aver avuto la possibilità di interfacciarsi col punteggio ottenuto dagli altri partecipanti, per il tester, è stato diverso in termini di stimoli. In una segnalazione al riguardo è stato proposto un database centralizzato che permettesse di vedere, anche in sessioni di testing indipendenti come quelle qui svolte, i risultati ottenuti dai profili degli altri tester, per avere un riferimento tangibile su quanto personalmente raggiunto rispetto agli altri.

5.2 Lavori Futuri

Il tool di Mutant Injection applicato in un contesto ludicizzato di visual GUI testing di applicazioni web è, come detto, uno dei primi lavori in questo senso, per cui, come abbiamo anche visto dalle limitazioni, saranno necessari perfezionamenti generali per renderne ancora migliore l'usabilità e la generalizzabilità attraverso, inoltre, delle più adeguate sessioni di sperimentazione.

Di seguito vengono elencate alcune proposte di miglioramento, ispirate anche in parte dai suggerimenti dei partecipanti all'esperimento di testing:

- Implementare i mutants riguardanti la modifica al listener di un widget, analizzando una soluzione che riesca quanto fatto in parallelo con il codice sorgente HTML, utilizzando l'*executeScript* messo a disposizione dal WebDriver di Selenium;

- Trovare un modo più efficiente per categorizzare un widget, rispetto a [32, Mozilla], che tenga conto dell'effettivo impatto visivo della tipologia di elemento del DOM all'interno della pagina;

- Modificare l'*Augment State* in Scout per generare un adeguato feedback nella fase di scoperta dei mutants, ad esempio riquadro del widget di un colore diverso

- Il feedback può essere anche sotto forma di popup che permette così al tester di percepire immediatamente quanto avvenuto, anche per quanto riguarda la votazione sulla severità del bug assegnata manualmente;

- Il voto al *bugSeverityScore* non sempre è stato utilizzato correttamente, bisognerebbe trovare altri modi per accentuarne la funzionalità. Inoltre, si può

implementare l'assegnazione tramite click del mouse sulla GUI del plugin, oltre al già presente input da tastiera, che, come ricordiamo, rimuove la possibilità di usare i numeri da 1 a 5 nell'assegnazione di un issue;

- Ai fini della gamification si potrebbe perfezionare il calcolo del punteggio ottenuto dal tester, adattandolo in particolare sull'interazione con le mutazioni durante la navigazione
- Rendere la modalità *headless* di Scout compatibile anche per Microsoft Edge;
- Risolvere le incongruenze ottenute dopo l'iniezione nel browser del codice sorgente di alcune pagine;
- Trovare una soluzione per abilitare lo scroll in Scout;
- Gestire l'apertura dei link in altre schede che non permettono la corretta interazione con l'AT;
- Utilizzo di una database centralizzato che gestisca la classifica dei punteggi dei tester per migliorare l'esperienza ludicizzata;
- Possibilità di rimuovere l'issue assegnato in precedenza per permettere la segnalazione di elementi internamente annidati;
- Mettere a disposizione del tester degli esempi della mutazione corrente da cercare, possibilmente tramite plugin dedicato in Scout, per facilitare la consapevolezza delle azioni da compiere nell'AT.

Un ulteriore elemento da perfezionare sarebbe la responsività del tool che tuttavia non era il principale obiettivo di quest'attività di ricerca perciò, oltre a considerare questo miglioramento ci si potrà concentrare sul perfezionamento nell'inserimento dei mutants sviluppati nel sorgente HTML, ma anche sulla creazione di altri tipi di mutazioni che vadano ad interagire anche con gli eventi dinamici dell'applicazione web, tenendo in considerazione un opportuno adattamento in Scout, se si vuole usare quest'ultimo come strumento per effettuare GUI testing.

Bibliografia

[1] Software Freedom Conservancy. *Selenium*. 2004. URL: https://www.selenium.dev/ (cit. alle pp. 8, 24).

[2] GitHub. *GitHub*. 2007. URL: https://github.com/ (cit. a p. 18).

[3] MS Research. *Yogi project*. 2007. URL: http://research.microsoft.com/en-us/projects/Yogi (cit. a p. 5).

[4] StackOverflow. *StackOverflow*. 2008. URL: https://stackoverflow.com/ (cit. a p. 18).

[5] Sebastian Deterding, Dan Dixon, Rilla Khaled e Lennart Nacke. «From Game Design Elements to Gamefulness: Defining Gamification». In: vol. 11. Set. 2011, pp. 9–15. DOI: 10.1145/2181037.2181040 (cit. a p. 14).

[6] Developed by JetBrains e Open-source Contributors. *Kotlin*. 2011. URL: https://kotlinlang.org/ (cit. a p. 29).

[7] Upsorn Praphamontripong. «Web mutation testing». In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE. 2012, pp. 495–498 (cit. a p. 12).

[8] Non specificato. *Di che cosa parliamo quando parliamo di gamification*. 2012. URL: https://learninglab.sdabocconi.it/files/sito/Gamification.pdf (cit. a p. 15).

[9] Emil Alegroth, Michel Nass e Helena Olsson. «JAutomate: A Tool for System- and Acceptance-test Automation». In: mar. 2013, pp. 439–446. ISBN: 978-1-4673-5961-0. DOI: 10.1109/ICST.2013.61 (cit. a p. 7).

[10] LaShanda Dukes, Xiaohong Yuan e Francis Akowuah. «A case study on web application security testing with tools and manual testing». In: *2013 Proceedings of IEEE Southeastcon*. IEEE. 2013, pp. 1–6 (cit. a p. 7).

[11] Vahid Garousi, Ali Mesbah, Aysu Betin-Can e Shabnam Mirshokraie. «A systematic mapping study of web application testing». In: *Information and Software Technology* 55.8 (2013), pp. 1374–1396 (cit. a p. 7).

[12] Maurizio Leotta, Diego Clerissi, Filippo Ricca e Paolo Tonella. «Capture-replay vs. programmable web testing: An empirical assessment during test case evolution». In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE. 2013, pp. 272–281 (cit. alle pp. 7, 9).

[13] Pourya Nikfard. «Testing on web applications». In: *Gorgan, Iran, May* (2014) (cit. a p. 7).

[14] Alessandro Orso e Gregg Rothermel. «Software testing: a research travelogue (2000–2014)». In: *Future of Software Engineering Proceedings*. 2014, pp. 117–132 (cit. alle pp. 4, 5).

[15] Oscar Pedreira, Felix Garcia, Nieves Brisaboa e Mario Piattini. «Gamification in software engineering – A systematic mapping». In: *Information and Software Technology* 57 (gen. 2014). DOI: 10.1016/j.infsof.2014.08.007 (cit. a p. 18).

[16] Emil Alégroth, Zebao Gao, Rafael Oliveira e Atif Memon. «Conceptualization and evaluation of component-based testing unified with visual gui testing: an empirical study». In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2015, pp. 1–10 (cit. alle pp. 12, 76).

[17] Paul E Anderson, Thomas Nash e Renée McCauley. «Facilitating programming success in data science courses through gamified scaffolding and learn2mine». In: *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. 2015, pp. 99–104 (cit. a p. 22).

[18] Rafael AP Oliveira, Emil Alégroth, Zebao Gao e Atif Memon. «Definition and evaluation of mutation operators for GUI-level mutation analysis». In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2015, pp. 1–10 (cit. alle pp. 12, 76).

[19] Yujian Fu e Peter J Clarke. «Gamification-based cyber-enabled learning environment of software testing». In: *2016 ASEE Annual Conference and Exposition*. 2016 (cit. a p. 22).

[20] Josè Miguel Rojas e Gordon Fraser. «Code defenders: a mutation testing game». In: *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2016, pp. 162–167 (cit. a p. 21).

[21] Tommaso Dal Sasso, Andrea Mocci, Michele Lanza e Ebrisa Mastrodicasa. «How to gamify software engineering». In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2017, pp. 261–271 (cit. a p. 23).

[22] Gordon Fraser. «Gamification of software testing». In: *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)*. IEEE. 2017, pp. 2–7 (cit. a p. 19).

[23] Fèlix Garcìa, Oscar Pedreira, Mario Piattini, Ana Cerdeira-Pena e Miguel Penabad. «A framework for gamification in software engineering». In: *Journal of Systems and Software* 132 (2017), pp. 21–40 (cit. a p. 21).

[24] Thomas Laurent, Laura Guillot, Motomichi Toyama, Ross Smith, Dan Bean e Anthony Ventresque. «Towards a Gamified Equivalent Mutants Detection Platform». In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2017, pp. 382–384 (cit. a p. 22).

[25] Inc. CodeSignal. *Coding Tests and Assessments for Interviews - CodeSignal*. 2018. URL: `https://codefights.com` (cit. a p. 22).

[26] Gabriela Martins de Jesus, Fabiano Cutigi Ferrari, Daniel de Paula Porto e Sandra Camargo Pinto Ferraz Fabbri. «Gamification in software testing: A characterization study». In: *Proceedings of the III Brazilian Symposium on Systematic and Automated Software Testing*. 2018, pp. 39–48 (cit. a p. 20).

[27] Socios. *Socios, powered by Chiliz*. 2018. URL: `https://www.socios.com/` (cit. a p. 19).

[28] Yu-kai Chou. *Actionable gamification: Beyond points, badges, and leaderboards*. Packt Publishing Ltd, 2019 (cit. a p. 15).

[29] Michel Nass, Emil Alegroth e Robert Feldt. «Augmented testing: Industry feedback to shape a new testing technology». In: *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2019, pp. 176–183 (cit. alle pp. 5, 10).

[30] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon e Mark Harman. «Mutation testing advances: an analysis and survey». In: *Advances in Computers*. Vol. 112. Elsevier, 2019, pp. 275–378 (cit. a p. 11).

[31] Michel Nass, Emil Alégroth e Robert Feldt. «On the industrial applicability of augmented testing: An empirical study». In: *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2020, pp. 364–371 (cit. alle pp. 5, 9, 10).

[32] MDN contributors. *Content categories*. 2021. URL: `https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Content_categories` (cit. alle pp. 25, 33, 35, 76, 78).

[33] Tommaso Fulcini. *Gamification Applicata al GUI Testing di Applicazioni Mobile*. 2021. URL: `https://webthesis.biblio.polito.it/18074/` (cit. alle pp. 15, 38).

[34] Davide Gallotti. *Ideazione e prototipazione di un sistema di gamification per il testing di applicazione web*. 2021. URL: https://webthesis.biblio.polito.it/19104/ (cit. a p. 4).

[35] Thomas Hamilton. *Mutation Testing in Software Testing: Mutant Score & Analysis Example*. 2021. URL: https://www.guru99.com/mutation-testing.html (cit. a p. 11).

[36] Copyright 2008 Google Inc. *Gson 2.8.8*. 2021. URL: https://github.com/google/gson (cit. a p. 32).

[37] Lukas Moldon, Markus Strohmaier e Johannes Wachs. «How Gamification Affects Software Developers: Cautionary Evidence from a Natural Experiment on GitHub». In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021, pp. 549–561. DOI: 10.1109/ICSE43902.2021.00058 (cit. a p. 18).

[38] The JUnit Team. *JUnit Testing Framework*. 2021. URL: https://junit.org/junit5/ (cit. a p. 5).

[39] Rahulkrishna Yandrapally e Ali Mesbah. «Mutation Analysis for Assessing End-to-End Web Tests». In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2021, pp. 183–194 (cit. alle pp. 13, 39, 49).

[40] Riccardo Gabellone. *Scout with Mutants Injection*. 2022. URL: https://git-softeng.polito.it/d023270/gamification-mutation-plugin (cit. alle pp. 36, 48).

[41] Riccardo Gabellone. *Web GUI Mutants Injection Tool*. 2022. URL: https://git-softeng.polito.it/s256793/mutants_injection_tool (cit. alle pp. 24, 29, 31).

[42] Klaas-Jan Stol, Mario Schaarschmidt e Shelly Goldblit. «Gamification in software engineering: the mediating role of developer engagement and job satisfaction». In: *Empirical Software Engineering* 27.2 (2022), pp. 1–34 (cit. a p. 19).