UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

tesi di laurea magistrale in Ingegneria del Software II

Un processo di sviluppo supportato da strumenti per la riduzione dei test breakage in ambito di applicazioni web template-based

Anno Accademico 2018/2019

relatore
Ch.mo prof. Porfirio Tramontana

candidato
Pierantonio Cangianiello
matr. M63/158

Alla mia famiglia

Indice

1 Introduzione **1**

1.1 Cenni storici 1

1.2 Definizioni 3

1.3 Problemi affrontati 4

1.4 Obiettivi 5

1.5 Organizzazione del lavoro di tesi 5

2 Panoramica e ambito d'interesse **7**

2.1 Strumenti per la creazione degli script di GUI Testing 7

2.1.1 Locatori 9

2.2 Evoluzione delle GUI 10

2.2.1 Esempi di evoluzione delle GUI 12

2.3 L'evoluzione degli script di GUI Testing 14

2.4 Cause dei GUI Test breakage 16

2.5 GUI Test Repair: stato dell'arte 19

2.5.1 Approccio basato su Event Flow Graph 20

2.5.2 Approccio basato su algoritmi genetici 21

2.5.3 Approccio incrementale 22

2.5.4 Approccio White-box 23

2.5.5 Approccio basato su locatori robusti 24

2.6 Sistemi di templating per il web 25

2.6.1 Evoluzione dei template engine 25

2.6.2 Definizione di “template” 27
2.6.3 Motivazioni per il templating 27
2.6.4 Esempio di utilizzo dei template 29
2.6.5 Considerazioni sui template engine 32

3 Rilevazione e riduzione dei test breakage 33

3.1 Introduzione 33
3.2 Scenario di test breakage 33
3.2.1 Esempio di test breakage su applicazioni web 35
3.2.2 Considerazioni 37
3.3 Processo di rilevazione e riparazione 39
3.4 Strumenti automatici di supporto al processo 42
3.4.1 Premessa 42
3.4.2 Descrizione degli algoritmi proposti 42
3.4.3 Caso base: template singolo 47
3.4.4 Caso generale: template multipli 49
3.5 Esempio reale di processo di sviluppo e manutenzione 52

4 Implementazione prototipale 61

4.1 Background tecnologico 61
4.2 Architettura 62
4.3 Descrizione dell’implementazione 66
4.3.1 Command Dispatcher 67
4.3.2 Processor Façade 68
4.3.3 TextMate Template File Processor 73
4.3.4 Hook Locator Builder 76

5 Esempi **79**

5.1 Installazione del prototipo 79

5.2 Esempio su architettura MVVM: AngularJS 80

5.2.1 Preparazione ambiente 80

5.2.2 Fase 1: Hook Injection 80

5.2.3 Fase 2: Test Capture 81

5.2.4 Fase 3: Rimozione hook inutilizzati 82

5.2.5 Fase 4: Modifica della vista 83

5.2.6 Fase 5: Controllo di coerenza tra test e template 83

5.2.7 Fase 6: Test repair manuale 85

5.3 Esempio su architettura MVC: Spring & FreeMarker 86

5.3.1 Preparazione ambiente 86

5.3.2 Fase 1: Hook Injection 89

5.3.3 Fase 2: Test capture 90

5.3.4 Fase 3: Rimozione hook inutilizzati 91

5.3.5 Fase 4: Modifica della vista 92

5.3.6 Fase 5: Controllo di coerenza tra test e template 92

5.3.7 Fase 6: Test repair manuale 94

6 Sperimentazione **96**

6.1 Premessa 96

6.1.1 Caratterizzazione locatori XPath Selenium 97

6.1.2 Caratterizzazione delle operazioni di manutenzione 99

6.2 Descrizione del processo 101

6.2.1 Research questions 101

6.2.2 Objects 101

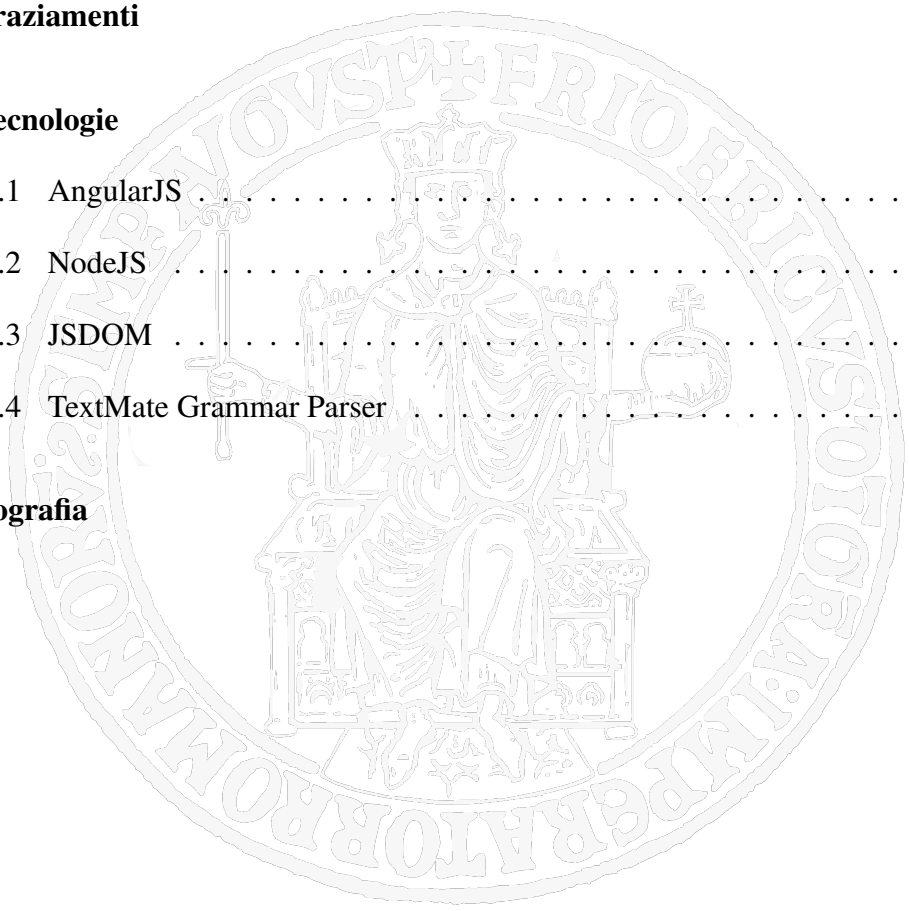
6.2.3 Factors 101
6.2.4 Independent Variables 102
6.2.5 Dependent Variables 102
6.2.6 Experimental Procedure 102
6.3 Risultati e analisi quantitativa 102
6.4 Analisi qualitativa 104
6.5 Problemi riscontrati 108
6.6 Threats to validity 109
6.6.1 Internal validity 109
6.6.2 External validity 110

Conclusioni e sviluppi futuri **111**

Ringraziamenti **113**

A Tecnologie **114**
A.1 AngularJS 114
A.2 NodeJS 116
A.3 JSDOM 120
A.4 TextMate Grammar Parser 121

Bibliografia **124**



Elenco delle figure

2.1.1 Katalon Recorder: esempio di un caso di test 8

2.1.2 Pagina di gestione utenti 9

2.1.3 GUI Test automation workflow 9

2.2.1 Evoluzione della GUI: sostituzione di componenti 12

2.2.2 Ristrutturazione grafica di una funzionalità 13

2.6.1 Risultato visivo del rendering 31

3.2.1 Test breakage durante il processo di sviluppo 34

3.2.2 Test breakage: tempo di rilevazione e riparazione 35

3.2.3 DOM: Registrazione script di test 36

3.2.4 Modifica del DOM 37

3.3.1 Processo per il rilevamento e il repair dei test breakage 41

3.4.1 Prima Hook Injection 47

3.4.2 Hook-based capture 48

3.4.3 Modifica e controllo consistenza tra test e template 48

3.4.4 Successiva Hook Injection 50

3.4.5 Hook Injection: template multipli 51

3.5.1 Interfaccia di Gestione Task 54

4.2.1 Component diagram del prototipo 63

4.2.2 Riorganizzazione dei componenti 64

4.2.3 Interazione tra i componenti 65

4.3.1 Diagramma delle classi 69

4.3.2 Raggruppamento token 75

4.3.3 Albero DOM ricavato dal template 76

5.2.1 Registrazione caso di test con Selenium su template AngularJS 82

5.2.2 Caricamento script di estensione per Selenium 83

5.2.3 Registrazione test con Selenium su template AngularJS (hook-based) 84

5.2.4 Hook rimossi: differenze su template AngularJS 85

5.2.5 Modifica del template AngularJS 86

5.2.6 Test non eseguibile su applicazione AngularJS 87

5.2.7 Ripristino degli hook su template AngularJS 88

5.3.1 Registrazione caso di test con Selenium 90

5.3.2 Registrazione test con Selenium su template FreeMarker (hook-based) . . . 91

5.3.3 Hook rimossi: differenze su template FreeMarker 92

5.3.4 Modifica del template FreeMarker 93

5.3.5 Test non eseguibile su applicazione FreeMarker 94

5.3.6 Ripristino degli hook su template FreeMarker 95

6.3.1 Diagramma di dispersione 103

A.2.1 Architettura di NodeJS 118

A.2.2 Runtime di NodeJS 118

Elenco dei listati

2.6.1 Utilizzo del template engine Smarty 29
2.6.2 Dati di esempio passati al template engine 30
2.6.3 Template Smarty 30
2.6.4 Risultato del rendering: codice HTML 31
3.2.1 Esempio di test script XML 36
3.5.1 Template: Gestione Task 53
3.5.2 Hook injection 55
3.5.3 Registrazione test di aggiunta task 56
3.5.4 Rimozione hook inutilizzati 57
3.5.5 Refactoring del template 58
4.3.1 Template: Yargs fluent API 68
4.3.2 Test Case in formato HTML 71
4.3.3 Controllo di coerenza tra test e template 72
4.3.4 TextMate Template File Processor Entry Point 74
4.3.5 Locator builder Hook-based 77
5.2.1 Hook injection su template AngularJS 81
5.3.1 Hook injection su template Freemarker 89
A.4.1 Esempio di template Twig e HTML 122
A.4.2 TextMate Scope Tree 122

Capitolo 1

Introduzione

1.1 Cenni storici

In diversi campi di interesse delle discipline ingegneristiche il testing è un tema trattato da lungo tempo. A seguito dei due conflitti mondiali, lo sviluppo dei calcolatori elettronici era orientato principalmente all'utilizzo nelle grandi aziende, negli sforzi bellici e nello sviluppo di primitive forme di intelligenza artificiale. Probabilmente, i più famosi esempi di test furono quelli pensati da Alan Turing, come il "Test di Turing", un metodo ideato per poter valutare l'intelligenza di un elaboratore: se un essere umano non fosse stato in grado di distinguere l'interazione con un computer da quella con un altro essere umano, la macchina avrebbe superato il test. Lo stesso Turing ha contribuito con un rapporto tecnico [19] introducendo idee innovative per testare la correttezza dei programmi per calcolatori mediante l'uso della logica matematica. Risultava evidente che il testing e il controllo qualità sarebbero diventati dei passaggi insostituibili nei processi di sviluppo.

Durante questo periodo, i governi realizzarono l'enorme potenziale di guadagno che l'apertura dei mercati e gli scambi globali avrebbero portato, e ciò per le aziende si sarebbe tradotto nella necessità di iniziare a strutturare i processi per poter garantire prodotti di qualità più alta in modo da poter competere con le realtà concorrenti straniere. La storia è simile per quanto riguarda il testing del software. Agli albori dello sviluppo software, dato che gli elaboratori e la conoscenza necessaria per poter creare software era a disposizione solo per pochi, esisteva solo un limitato numero di linguaggi di programmazione. Inoltre, la mancanza di linguaggi che potessero essere sfruttati su piattaforme diverse costituiva un ulteriore ostacolo alla

diffusione della conoscenza.

Dopo la rivoluzione che IBM ha portato nell'industria tecnologica con la diffusione su larga scala dei primi personal computer, iniziò ad aumentare la richiesta di software che potesse funzionare correttamente su piattaforme diverse, man mano che diversi competitor stavano rilasciando in contemporanea i loro prodotti sul mercato. Nel contempo, gli utenti finali iniziavano a percepire la necessità di programmi che potessero effettivamente essere eseguiti su piattaforme eterogenee, pertanto iniziarono ad emergere le problematiche correlate al testing di numerose condizioni e combinazioni, prima che un prodotto software potesse essere commercializzato. Arrivando infine ad oggi, gruppi di ricerca, sviluppatori ed aziende continuano negli sforzi per lo sviluppo di software che possa essere eseguito correttamente su diverse piattaforme e dispositivi, avendo addirittura a disposizione le potenzialità di Internet "in tasca" grazie a dispositivi come gli smartphone.

Per una chiara introduzione agli argomenti di interesse, è utile in questa sede suddividere concettualmente il software come oggi lo conosciamo in due macro aree: la prima è rappresentata da applicazioni che richiedono in qualche misura un'interazione con utenti mediante dispositivi di input e interfacce grafiche, come ad esempio le app per smartphone, i programmi CAD, sistemi di intrattenimento multimediali, e così via; la seconda è rappresentata da tutti i programmi che non prevedono interazione diretta con gli utenti, come ad es. sistemi di backup automatici, software per l'assistenza di frenata dei veicoli, software per il controllo di sensori e attuatori sulle linee produttive industriali, ecc.

Per ciascuna di queste due categorie si pone il problema del testing: in particolare, in questo lavoro di tesi verranno affrontati diversi temi riguardanti il testing automatizzato di interfacce utente (GUI) e le problematiche riguardanti la manutenzione dei test automatici nel corso dello sviluppo software.

1.2 Definizioni

Il testing rientra nella categoria delle tecniche dinamiche adottate nel processo di verifica e convalida del software: a differenza delle tecniche statiche, un test ha bisogno di un programma in esecuzione per poter esaminare i parametri di interesse come gli output, i tempi di esecuzione e così via. Oltre a verificare il rispetto dei requisiti, uno degli obiettivi principali del testing è quello di rilevare difetti nel software; in seguito, l'attività di bug fixing consente di trovare e correggere tali difetti in modo da incrementare la robustezza del software. Il sistema oggetto del testing viene poi continuamente sottoposto a testing, sia a seguito di bug fixing, sia a seguito di nuovi sviluppi: in tal caso si parla di regression testing, utile a verificare che non siano stati introdotti nuovi difetti durante le attività di manutenzione.

Un aspetto critico della progettazione ed esecuzione del testing è il grado di automazione. Diverse tipologie di test come unit tests, performance test ed altri, possono essere completamente automatizzati: sfruttando dei framework o strumenti dedicati è possibile rendere il processo di testing veloce e facilmente ripetibile. Un evidente svantaggio però, è la necessità di dover manutenere le suite di test facendo attenzione a non introdurre bug nei test stessi. Per ovviare a ciò, spesso si adottano ulteriori framework di supporto, come per esempio per il controllo di condizioni (assertion frameworks): così facendo si riduce il codice necessario ad implementare i test e di conseguenza la probabilità che questi ultimi contengano dei bug. Per organizzare i test automatici si ricorre alla stesura di uno o più script, ovvero un insieme di file in cui sono contenuti i casi di test unitamente ai risultati attesi per ognuno di essi. Non è sempre utile o vantaggioso automatizzare i test: potrebbe accadere che alcune funzionalità cambino con una frequenza tale da rendere onerosa la manutenzione dei test associati, pertanto in tali casi è utile ricorrere al test manuale.

1.3 Problemi affrontati

Un problema rilevante nell'ambito dell'automazione dei test di interfacce grafiche è il numero elevato di test che potrebbero diventare inutilizzabili ogni volta che il software viene modificato [15]. Un caso di test per interfacce grafiche consiste di una sequenza di azioni utente (eventi) mediante componenti grafici, e dei punti di controllo che determinano se il programma è stato eseguito correttamente. Questa tipologia di test è strettamente legata alla struttura dell'interfaccia grafica, ovvero ciascun test ha riferimenti a specifici componenti sulla GUI e codifica sequenze di eventi su di essi. Durante il processo di manutenzione del software, potrebbe accadere che l'interfaccia utente viene modificata e alcuni test potrebbero risultare inutilizzabili per i seguenti motivi:

- La sequenza di eventi codificata nel test non è più consentita sull'interfaccia grafica modificata.
- Le asserzioni non riescono più a controllare correttamente gli elementi grafici richiesti.

Bisogna sottolineare che questo genere di problema spesso non si pone per i test manuali: un collaudatore umano riesce ad eseguire un caso di test e a tollerare leggere modifiche su un interfaccia grafica [6], potendo dedurre (usando il buon senso) se un cambiamento è voluto in quanto parte dell'evoluzione del software o è assimilabile ad un errore (e quindi un bug introdotto durante la manutenzione). Al contrario, se un test automatico incontra un qualsiasi dettaglio che differisce da ciò che è atteso, semplicemente o fallisce o si blocca. Quindi, nonostante l'automazione dei test risulta vantaggiosa in quanto può velocizzare i processi di sviluppo e test, i benefici possono ben presto svanire a causa degli elevati costi di manutenzione nel momento in cui un numero elevato di test diventa inutilizzabile a causa di modifiche del software, richiedendo pertanto una riscrittura o una nuova registrazione (nel caso di tecnica capture/replay) di gran parte di essi.

1.4 Obiettivi

Gli obiettivi principali di questo lavoro di tesi sono riportati nei punti seguenti:

- Proporre un processo di sviluppo supportato da strumenti automatici, che possano segnalare gli eventuali impatti delle modifiche apportate alle viste sui casi di test, già durante gli interventi sul codice sorgente e senza dover eseguire i test.
- Fornire strumenti che diano indicazioni utili per la riparazione manuale dei test impattati dalle modifiche.
- Valutare qualitativamente il risparmio temporale durante la manutenzione degli script, e valutare la robustezza della soluzione.

1.5 Organizzazione del lavoro di tesi

Il capitolo 2 introduce i seguenti argomenti, che rappresentano il punto di partenza del lavoro di tesi:

- Gli strumenti utilizzati per il test capture/replay per le applicazioni web.
- L'evoluzione delle interfacce grafiche e il problema dei test breakage che ne conseguono, ovvero varie formulazioni del problema dei guasti dei test dovuti a diversi tipi di modifiche del software.
- Qualche breve cenno sulle soluzioni proposte in letteratura per effettuare la riparazione dei test non funzionanti.
- Ambito di applicabilità del processo e degli strumenti presentati in questo lavoro di tesi per la riduzione dei test breakage: le applicazioni web basate su tecnologie di templating.

Il capitolo 3 introduce diverse nozioni propedeutiche alle parti successive:

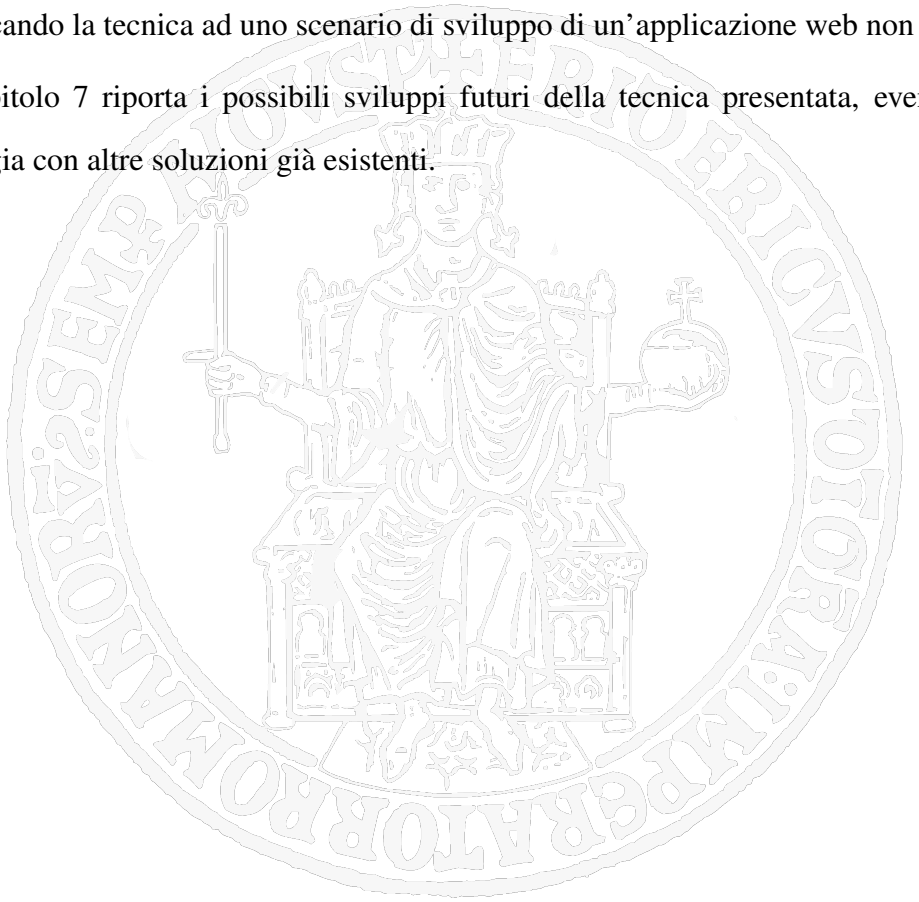
- Un esempio concreto di test breakage durante lo sviluppo di un'applicazione web seguendo un processo di sviluppo tradizionale.
- La descrizione del processo e degli strumenti implementati per ridurre il numero di potenziali test breakage, rilevando già durante lo sviluppo tali situazioni.
- Un esempio di conduzione del processo supportato dagli strumenti per la riduzione dei breakage.

Il capitolo 4 riporta nel dettaglio l'implementazione prototipale degli strumenti a supporto del processo di sviluppo che consentono di raggiungere gli obiettivi posti all'inizio.

Il capitolo 5 riporta alcuni esempi concreti di conduzione del processo e utilizzo degli strumenti, considerando diverse tecnologie di templating. L'obiettivo è mostrare il grado di indipendenza dalla specifica tecnologia.

Il capitolo 6 è dedicato alla valutazione del prototipo e alla discussione dei risultati ottenuti applicando la tecnica ad uno scenario di sviluppo di un'applicazione web non banale.

Il capitolo 7 riporta i possibili sviluppi futuri della tecnica presentata, eventualmente in sinergia con altre soluzioni già esistenti.



Capitolo 2

Panoramica e ambito d'interesse

Questo capitolo presenta alcune nozioni riguardanti l'automazione dei test di interfacce grafiche, il tema dell'evoluzione e manutenzione delle interfacce grafiche in sistemi software complessi e le conseguenze (in termini di guasti o *breakage*) che tali attività hanno sui test; è riportato poi un accenno ad alcune soluzioni proposte in ambito accademico per la riparazione degli script di test; infine è mostrato il contesto tecnologico nel quale è stata affrontata tale problematica.

2.1 Strumenti per la creazione degli script di GUI Testing

In questa sezione si introduce la tecnica di *capture/replay* (o equivalentemente, *record/replay*) per la generazione automatica degli script; gli strumenti che implementano tale tecnica consentono di catturare sequenze di azioni e di valori immessi dall'utente in un'interfaccia grafica e generare degli script che possono rieseguire in automatico tali sequenze sull'applicazione.

Nell'ambito delle applicazioni web è possibile generare script di test mediante il tool Selenium IDE[1]; in particolare, si farà riferimento al porting "Katalon Recorder"[2] per il browser Chrome. Per mostrare il processo di capture/replay, la figura 2.1.1 mostra un test creato da Katalon Recorder dalla sequenza di input e azioni sulla pagina di gestione degli utenti riportata in figura 2.1.2.

[1] https://www.seleniumhq.org/projects/ide/

[2] https://www.katalon.com/resources-center/blog/katalon-automation-recorder/

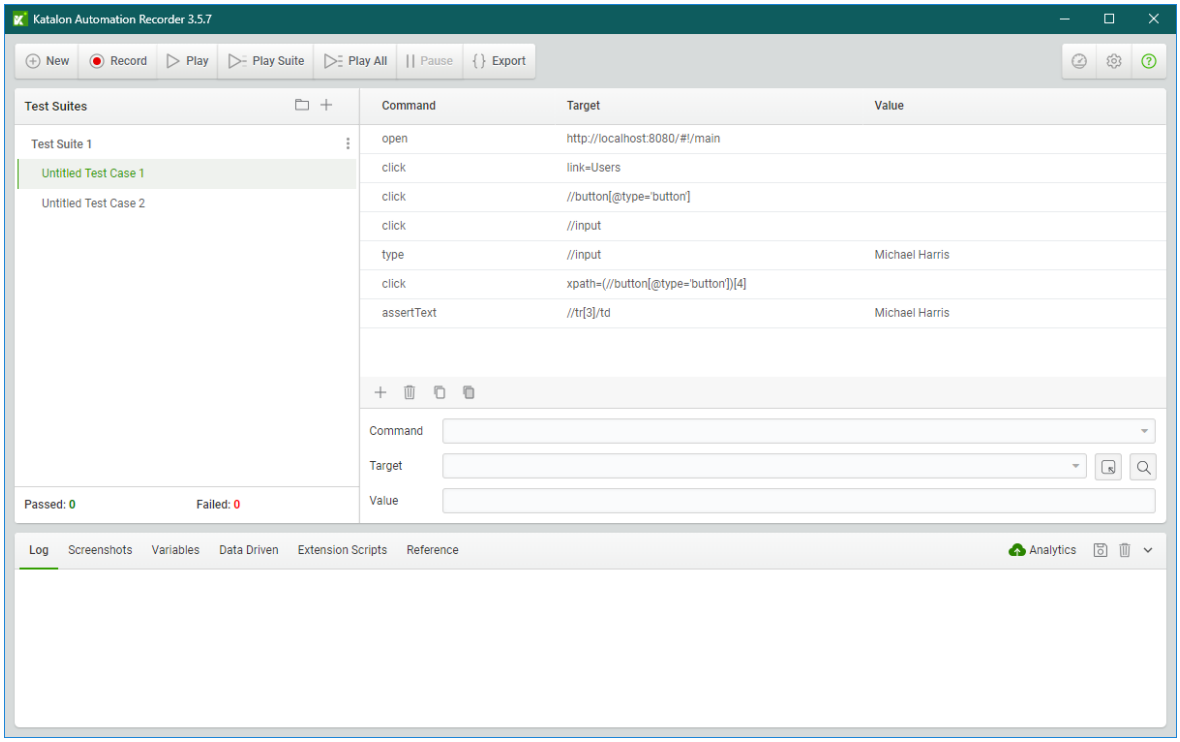


Figura 2.1.1: Katalon Recorder: esempio di un caso di test

Durante il processo di capture, ciascuna azione o input dell'utente genera un comando nello script di test; ciascun comando è una tupla [action, locator, value][3]. La parte "action" di un comando denota o un evento generato dall'utente (`click`, `type`) durante il processo di registrazione, o un elemento di controllo del processo di replay (come `assertText`). La parte "component" indica l'elemento dell'interfaccia grafica (link, button, input box) col quale l'utente ha interagito durante quel particolare step del processo di registrazione. La parte "value" si riferisce a qualsiasi input da parte dell'utente, come il valore selezionato da una lista o il valore immesso in un campo di testo, oppure al valore di controllo in fase di replay.

Nella figura 2.1.3 è riportato uno schema concettuale del processo di automazione dei test di interfacce grafiche mediante la tecnica capture-replay.

[3]Gli script di Selenium identificano la tupla come [command, target, value]; si è adottata invece un'altra terminologia per motivi di chiarezza.

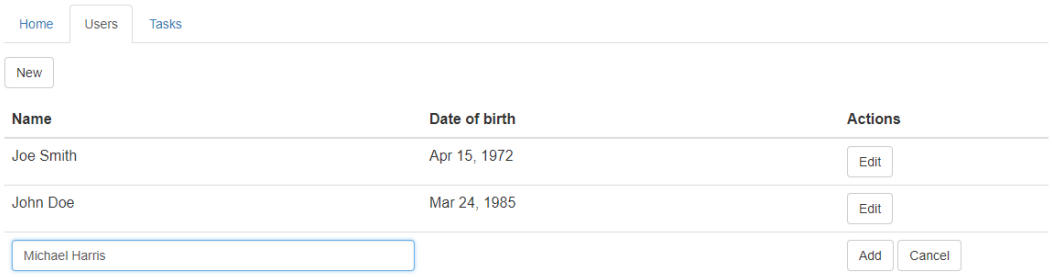


Figura 2.1.2: Pagina di gestione utenti

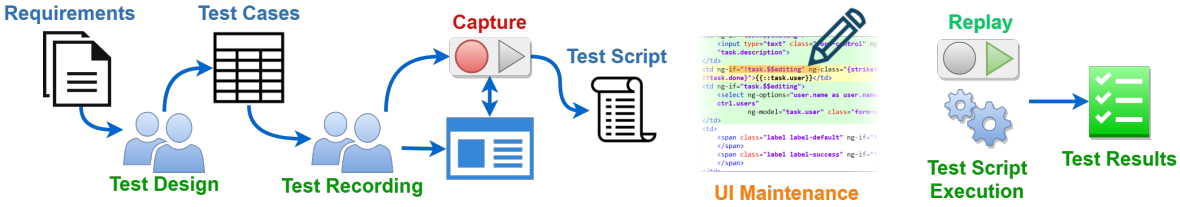


Figura 2.1.3: GUI Test automation workflow

2.1.1 Locatori

Per avere una panoramica completa sul funzionamento degli strumenti di capture/replay e sulle problematiche legate al breakage degli script di GUI testing, bisogna introdurre qualche concetto più specifico riguardante gli aspetti tecnologici delle GUI, in questo caso particolare verranno trattati quelli delle applicazioni web.

Le interfacce grafiche delle applicazioni web sono costruite mediante la composizione di elementi HTML come campi di testo, radio buttons, liste, collegamenti ipertestuali e così via. Ciascun elemento HTML (come avviene similmente per XML) è identificato da un tag e possiede opzionalmente uno o più attributi in forma di coppie nome/valore (ad es. `class="container"`). Gli attributi possono avere svariati impieghi: tra tutti, possono fornire un modo semplice per identificare un elemento mediante un nome, o indicare caratteristiche visuali come il colore, lo spessore dei bordi e altre. Inoltre, essi possono essere utilizzati per mantenere lo stato "dinamico" di un oggetto, come ad es. il valore di una casella di testo o la spunta di una checkbox. Infine, un elemento HTML può avere del contenuto testuale incluso tra i tag di apertura e chiusura.

I locatori possono essere utilizzati da JavaScript e dagli strumenti di capture/replay per identificare e manipolare gli elementi HTML. Esistono due categorie principali di locatori:

1. Locatori *attribute-based*: utilizzano gli attributi come id o name per identificare gli elementi.

2. Locatori *structure-based*: fanno riferimento alla struttura della pagina per identificare gli elementi. Possono essere suddivisi in due classi distinte:

   (a) Locatori *hierarchy-based*: identificano gli elementi in termini della loro posizione all'interno dell'albero del DOM[4] mediante strategie come XPath o selettori CSS.

   (b) Locatori *index-based*: sono impiegati in situazioni in cui alcuni elementi della pagina possiedono locatori identici nel caso si utilizzano le altre tipologie; in questo caso vengono impiegati degli indici per differenziarli.

2.2 Evoluzione delle GUI

Nei sistemi complessi, i cambiamenti evolutivi (oltre che correttivi) sono parte integrante del ciclo di vita del software e coprono spesso un arco temporale piuttosto lungo. La maggior parte dei sistemi software che prevedono interazione con gli utenti mediante l'uso di interfacce grafiche sono progettati ricorrendo agli strumenti di *rapid prototyping* [16], i quali consentono l'evoluzione delle interfacce grafiche al fine di migliorare l'esperienza complessiva degli utenti. La velocità con cui cambiano le interfacce grafiche può essere anche molto più alta della logica di business: per esempio, le applicazioni web più popolari come le webmail, i social media, o le applicazioni desktop come le suite per ufficio, aggiornano continuamente le interfacce, mentre le logiche sottostanti restano sostanzialmente invariate.
Di seguito sono elencati alcuni dei motivi principali per l'evoluzione delle interfacce grafiche:

[4] Acronimo per "Document Object Model"

- Cambio di stile complessivo: in questa categoria di modifiche ricadono il cambio dei font impiegati, i colori, il posizionamento e la dimensione dei componenti, l'aggiunta di elementi visivi per semplificare la localizzazione di funzionalità più utilizzate e così via.

- Cambio di tecnologia: in questo caso, il cambiamento apportato è più radicale e si riferisce all'introduzione (o alla sostituzione) del framework grafico/strutturale (come ad es. il passaggio da JQuery ad Angular) o addirittura al passaggio ad un nuovo linguaggio (come ad es. il passaggio da QT a JavaFX).

- Innovazione dell'esperienza utente: in quest'ambito rientrano tutti i cambiamenti dovuti all'adozione dei nuovi pattern per la User Experience, che possono coinvolgere anche alcuni o tutti gli altri punti elencati: l'esempio per eccellenza è il passaggio al *Material Design* a cui si è assistito di recente su larga scala sia nell'ambito mobile, sia nelle applicazioni web.

- Cambio del livello di astrazione: alcune funzionalità possono evolvere per diventare più generiche e pertanto alcuni elementi grafici potrebbero subire modifiche o essere sostituiti per adeguarsi di conseguenza. Un esempio è la sostituzione di una lista di elementi preimpostati con una lista costruita dinamicamente.

- Adeguamento agli standard di accessibilità: molte applicazioni, soprattutto nel settore pubblico/istituzionale, per legge hanno l'obbligo di rispettare dei parametri di accessibilità a tutela e supporto degli utenti con ridotte capacità sensoriali: questo tipo di modifiche si rende necessario nel momento in cui vengono aggiornate le leggi che regolano questi aspetti.

- Introduzione di nuove *gesture*: grazie all'evoluzione dell'hardware per l'interazione utente, sono state rese disponibili nuove modalità di approccio con le interfacce grafiche, prima impossibili: si pensi alle gesture multitouch come il *pinch-to-zoom*, lo

swipe, il *long-touch* e così via: anche in questo caso si rende necessario il refactor delle GUI per poter rispondere a queste nuove interazioni.

2.2.1 Esempi di evoluzione delle GUI

In questa sezione sono riportati alcuni esempi pratici di evoluzione delle GUI che risultano significativi nell'ambito della manutenzione degli script di test. In figura 2.2.1 si considera l'evoluzione di un'interfaccia grafica di un'applicazione web, in particolare del primo step di una procedura guidata per la creazione di un catalogo prodotti. In entrambe le versioni in figura, la maschera consente l'inserimento dei dati principali del catalogo prodotti: il nome e la tipologia di articoli che dovrà contenere.

Nella prima versione, la selezione della categoria avviene mediante una lista predeterminata di categorie, visualizzate come un gruppo di radio buttons; nella seconda versione, a causa dell'introduzione delle categorie dinamiche, i radio buttons sono stati sostituiti con una combo box, in modo da consentire un numero arbitrario di possibili valori.

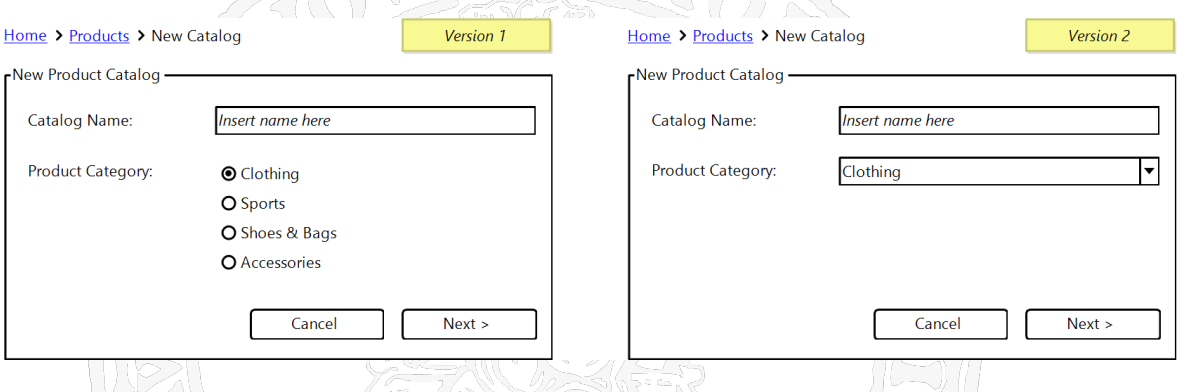


Figura 2.2.1: Evoluzione della GUI: sostituzione di componenti

In figura 2.2.2 è rappresentato un altro esempio relativo ad un caso più complesso, ovvero l'evoluzione di una funzionalità che consente la definizione di attributi dei prodotti e dei valori che essi possono assumere. In questo caso la funzionalità resta invariata, mentre la struttu-

ra delle schermate, la modalità di inserimento e la navigazione dell'applicazione risultano modificati.

Nella prima versione, l'inserimento, la modifica e la cancellazione degli attributi avvengono nella stessa schermata, mediante una tecnica di editing *inline* della tabella; l'aggiunta e la rimozione dei valori di ciascun attributo avvengono invece in una schermata diversa, raggiungibile mediante il menù di navigazione dell'applicazione. Per semplificare il flusso di lavoro e migliorare l'esperienza utente, nella versione 2 sono state apportate le seguenti modifiche: l'editing e la creazione degli attributi non avvengono più inline, ma sono stati spostati in una schermata di editing dedicata, assieme alla lista dei relativi valori. Questa modifica consente di rendere meno frammentata la funzionalità di manutenzione degli attributi e valori, riducendo di conseguenza il tempo di navigazione tra le diverse schermate coinvolte.

Figura 2.2.2: Ristrutturazione grafica di una funzionalità

A valle di questi esempi, prendendo come riferimento un'architettura *Model View Controller*, si può definire un GUI "refactoring" come un insieme di cambiamenti che impattano

soltanto la parte visuale (e di conseguenza il codice della vista e del controller che implementano l'interfaccia grafica) ma non il modo in cui essa si comporta (quindi non il modello sottostante) [2].

2.3 L'evoluzione degli script di GUI Testing

L'automazione dei test di interfacce grafiche gioca un ruolo chiave nel ridurre i costi elevati del testing manuale [4]. Per poter automatizzare il processo di testing è possibile scrivere programmi (utilizzando solitamente linguaggi di scripting) che imitano il comportamento degli utenti, eseguendo al loro posto le azioni sulle interfacce grafiche, grazie a dei framework di testing dedicati. La stesura degli script di test da zero è un processo che comporta un investimento significativo: è necessario implementare sofisticate logiche di testing, ovvero del codice che processa i dati in input, utilizza tali dati per valorizzare oggetti sulle schermate, agisce su di esse per avviare le computazioni desiderate e infine recupera i risultati, confrontandoli con gli oracoli[5] per determinare se l'applicazione si comporta correttamente. Il riuso della logica di testing nel tempo è lo scopo ultimo dell'automazione dei test.

Purtroppo, il rilascio di nuove versioni delle applicazioni contenenti modifiche alle interfacce grafiche comporta il guasto (*breakage*) dei corrispondenti script di test, annullando pertanto i benefici dell'automazione dei test. Anche soltanto alcune semplici modifiche come quella di figura 2.2.1 potrebbero invalidare diverse istruzioni all'interno degli script di test che riferiscono quei particolari oggetti dell'interfaccia grafica. Per far sì che i test scritti per tali interfacce grafiche possano essere riutilizzati, bisogna ripristinarne il funzionamento, e questo processo è laborioso e complesso; in uno scenario reale, data la complessità degli script, potrebbero essere impiegati anche diversi giorni per renderli nuovamente operativi in modo da poter essere utilizzati per il test delle versioni successive dell'applicazione. Inoltre, gli strumenti a supporto del testing sono in grado di rilevare le eccezioni sollevate dagli script

[5]Nell'ambito del testing, un "oracolo" è un meccanismo che determina se l'esecuzione del software è corretta per un particolare caso di test. Esso è composto da due parti: l'informazione dell'oracolo che rappresenta il risultato atteso e la procedura dell'oracolo che confronta l'informazione dell'oracolo con il risultato vero e proprio[12].

di test soltanto a runtime, pertanto è necessario eseguire gli script (con dispendio di tempo spesso quantificato in ore, a causa di possibili cicli presenti nei test) in modo da raggiungere quelle particolari istruzioni che riferiscono gli elementi dell'interfaccia grafica che sono stati modificati. A valle di queste considerazioni, spesso si preferisce scartare del tutto i test scritti in precedenza e scriverne di nuovi, nel momento in cui vengono rilasciate nuove versioni dell'applicazione. Di conseguenza, dato che spesso gli script non vengono consegnati ai clienti insieme all'applicazione, raramente ne viene controllata l'aderenza ai requisiti; in generale, gli script nuovi sono meno sofisticati e meno efficaci rispetto a quelli precedenti, che invece erano stati utilizzati e rifiniti per un diverso tempo.

Evoluzione basata su modelli

È possibile astrarre la struttura di un'interfaccia grafica in modelli che possano guidare il processo di testing basato sugli script, effettuando un controllo di coerenza tra operazioni contenute negli script e il modello ricavato dall'interfaccia grafica. Nello specifico, i test di regressione di interfacce grafiche implicano la creazione e il confronto tra modelli di alto livello dell'applicazione prima di utilizzare gli algoritmi che si occupano di creare i casi di test per le successive versioni dell'applicazione [20]. I modelli possono essere estratti dal codice sorgente dell'applicazione, ma bisogna mettere in conto due limitazioni di fondo [6]:

- Nel testing black box (e in particolare nel testing di interfacce grafiche), il codice sorgente dell'applicazione non è disponibile, inoltre spesso il testing viene affidato a terze parti che non possono avere accesso ai sorgenti: in questo caso, derivare i modelli dalla struttura dell'interfaccia grafica non è fattibile.
- Anche se il codice sorgente fosse disponibile, ci sono delle limitazioni che rendono inefficace l'estrazione dei modelli dal codice. Si consideri il caso in cui gli oggetti delle interfacce grafiche sono creati attraverso un'API[6], come ad esempio la funzione

[6] Acronimo di "Application Programming Interface", ovvero un'insieme di funzioni e procedure che consentono la creazione di applicazioni in grado di accedere ai dati o alle funzionalità di un particolare servizio o sistema.

`CreateWindow`. Questa specifica funzione richiede un certo numero di parametri, tra cui l'identificativo del tipo di componente che è un valore generato soltanto a runtime (tramite la chiamata `RegisterClass`): in questo caso è del tutto impossibile la creazione del modello dell'interfaccia grafica a partire dai sorgenti. Inoltre, derivare il modello a partire dal codice sorgente richiede: una conoscenza specifica delle API che creano e manipolano i componenti, la creazione di parser e analizzatori per i linguaggi usati per creare le interfacce grafiche, lo sviluppo di strumenti per gli IDE[7] in grado di estrarre i modelli a partire dal codice. All'aumentare degli ambienti di sviluppo, dei linguaggi e delle API, il numero di combinazioni di questi ultimi rende non fattibile quest'approccio.

In seguito verrà mostrato come sia possibile superare tali limiti nel caso di applicazioni template based, facendo opportune ipotesi sulla struttura delle interfacce grafiche in questo contesto.

2.4 Cause dei GUI Test breakage

In questa sezione è riportata una classificazione delle cause *prossimali*, ovvero quelle che sono direttamente legate a ciascun breakage del codice di test. Esistono altre tipologie di cause dette *distali*, come i cambiamenti apportati al codice dell'applicazione che comportano il breakage dei test. Per fare un esempio di causa *prossimale*, uno sviluppatore potrebbe modificare il codice di un'applicazione alla versione V, rimuovendo un particolare elemento da una pagina. Tale modifica potrebbe causare il breakage di un test nella successiva versione V' quando esso tenta di accedere a quel particolare elemento. Il fatto che il locatore corrispondente all'elemento rimosso sia non più valido, è la causa *prossimale* del breakage; il fatto che lo sviluppatore abbia modificato la pagina rimuovendo l'elemento, è la causa *distale*. Di seguito sono riportate le cause prossimali classificate in base alla loro tipologia [7]:

[7] Acronimo di "Integrated development environment", ovvero ambiente integrato per lo sviluppo.

- Locatori attribute-based: in questa categoria ricadono tutti i test breakage che sono correlati ai locatori che utilizzano gli attributi per identificare gli elementi dell'applicazione:
  - Attributo non trovato: si verifica in un test che tenta di trovare un elemento tramite un attributo che funzionava nella versione V, ma non nella successiva V'. Gli attributi generalmente coinvolti in questo caso sono: ID auto-generati, ID manuali, href, name, value, class, onClick. Un esempio di causa distale può essere la cancellazione o la modifica di uno degli attributi elencati.
  - Testo dell'elemento non trovato: si verifica in un test che tenta di localizzare un elemento utilizzando il testo contenuto tra il relativo tag di apertura e chiusura.
- Locatori hierarchy-based: si verifica in un test che tenta di accedere ad un elemento mediante un'espressione gerarchica che era valida nella versione V, ma non nella successiva V'. Le cause distali più frequenti in questo caso includono l'aggiunta o la rimozione di elementi intermedi nell'albero del DOM.
- Locatori index-based: si verifica in un test che tenta di accedere ad un elemento tramite un'espressione gerarchica contenente un indice esistente nella versione V, ma non nella successiva V'. La causa distali più frequente è l'aggiunta, la rimozione o il differente ordinamento all'interno di gruppi di elementi simili tra loro.
- Valori: le cause elencate in questa categoria sono correlate ai valori (dati) presenti nei test:
  - Valore di testo in input non valido: si verifica quando un valore di input usato da un test è accettato nella versione V, ma non nella successiva V'. Una causa distale, per esempio, è il cambiamento del vincolo di validità su un campo di testo, come il vincolo sulle maiuscole/minuscole in un campo password non presente nella versione V, ma presente nella successiva V'.

  - Valore mancante: si verifica quando un valore di input non incluso in un test non era richiesto nella versione V, ma presenti nella successiva V'. Una causa distale può essere l'aggiunta di campi obbligatori ad una form.
  - Valore assente da una lista: si verifica quando un valore che veniva selezionato in un test da una lista nella versione V, non è più presente nella successiva V'.
  - Valore dell'asserzione: le asserzioni nei test capture/replay confrontano i valori attesi coi valori reali e segnalano i casi in cui essi non corrispondono. Se il comportamento del test differisce tra la versione V e la successiva V', si verifica il breakage. La causa distale potrebbe essere ad esempio la modifica tra V e V' di un messaggio di errore rilevato dal test.

- Ricaricamento della pagina: le cause ricadenti in questa categoria riguardano gli eventi di ricaricamento della pagina web. Questi eventi possono essere generati sia dall'utente, sia richiesti automaticamente dall'applicazione in seguito a condizioni specifiche:
  - Ricaricamento necessario: si verifica quando un test eseguito sulla versione V non causa il ricaricamento della pagina, mentre lo causa nella versione successiva V'. Il breakage potrebbe verificarsi a causa dell'incompatibilità tra il ritardo introdotto nel tempo di caricamento e le successive azioni del test.
  - Ricaricamento non più necessario: si verifica quando un test eseguito sulla versione V causa il ricaricamento della pagina, mentre non lo causa nella versione successiva V'. In questo caso il test potrebbe restare in attesa di un ricaricamento della pagina e generare un errore dopo un timeout prefissato, senza poter eseguire le azioni successive.
- Tempo di sessione utente: molte applicazioni web utilizzano sessioni utente a scadenza, come ad esempio le applicazioni di home banking prevedono per motivi di sicurezza il logout automatico dell'utente dopo un breve periodo di inattività. In questa categoria ricadono i breakage legati appunto ai tempi di sessione:

- Tempo di sessione prolungato: si verifica quando un test eseguito sulla versione V è progettato per attendere un tempo di sessione t_1, ma quando eseguito sulla versione successiva V' dovrebbe attendere un tempo t_2, sufficientemente maggiore di t_1. In questo caso, il messaggio di scadenza della sessione atteso dal test non si presenta, causando il breakage.
- Tempo di sessione diminuito: è il caso duale del punto precedente. Il breakage potrebbe manifestarsi forzando il logout dell'utente prima del completamento del comando di attesa del test, invalidando tutte le azioni successive.

La maggior parte dei breakage osservati in [7] sono causati dai locatori. È chiaro che la risoluzione di questa classe di errori mediante tecniche di rilevazione dei breakage e repairing può avere un impatto significativo sulla riusabilità dei test tra rilasci successivi dell'applicazione.

2.5 GUI Test Repair: stato dell'arte

In letteratura sono stati presentati diversi approcci che affrontano il problema dei GUI test breakage e in diverse modalità mirano alla **riparazione** degli script di test. Tali soluzioni sono specifiche all'ambito dei test di interfacce grafiche, dato che le soluzioni proposte per il repairing degli unit test come ad es. JUnit non sono applicabili agli script prodotti dagli strumenti di capture/replay [7].

Pur essendovi alcune differenze con la soluzione proposta in questo lavoro di tesi, è utile valutare gli altri lavori di ricerca per avere un quadro chiaro dello stato dell'arte in questo ambito. Gli aspetti considerati sono sostanzialmente due:

- Rilevazione dei test breakage: è necessaria sia per individuare gli script da riparare che per fornire una metrica che consenta di valutare successivamente l'efficacia del processo di riparazione.

- Riparazione degli script: è il processo che mira a ripristinare il funzionamento degli script rilevati.

2.5.1 Approccio basato su Event Flow Graph

La tecnica basata su Event Flow Graph mira al repairing degli script mediante trasformazioni successive. Similmente ai modelli di *control-flow*, che rappresentano tutti i possibili percorsi di esecuzione in un programma, o ai modelli di *data-flow* che rappresentano tutte le possibili definizioni ed usi di una locazione di memoria, i modelli di *event-flow* rappresentano tutti le possibili sequenze di eventi che possono essere eseguiti su un'interfaccia grafica. Più nello specifico un'interfaccia grafica viene suddivisa in una gerarchia di finestre modali[8] (formando l'albero di integrazione); ciascuna finestra modale è rappresentata come un *event-flow graph* (EFG) che mostra tutti i possibili percorsi di esecuzione degli eventi per quella finestra; i singoli eventi sono rappresentati utilizzando le loro precondizioni ed effetti. La base di partenza poter eseguire il repair consiste nell'affidarsi ai modelli EFG dell'interfaccia grafica imperfetti ed inaccurati (ottenuti mediante reverse engineering tramite GUI *ripping* [13]), oltre che all'input manuale [14].

La tecnica proposta in [5] denominata SITAR (ScrIpT repAireR) ha lo scopo di elevare il livello di astrazione degli script inusabili (a causa di modifiche dell'interfaccia grafica) dal linguaggio di scripting ad un modello, al quale successivamente applicare delle trasformazioni di repairing per ottenere dei casi di test usabili, mediante sintesi di codice a partire dal modello corretto. Più in dettaglio, SITAR prende in input una applicazione da testare A_0 insieme all'event-flow graph G_0 associato (ottenuto mediante reverse engineering), la versione modificata dell'applicazione A_1 insieme all'event-graph G_1, e la suite di test TS_0 (inclusiva delle asserzioni e punti di controllo degli oggetti grafici) creata per A_0. SITAR identifica il sottoinsieme $ts_0 \subseteq TS_0$ dei casi di test che non sono più utilizzabili per A_1 e pertanto candidati al repair. SITAR usa infine delle trasformazioni per il repair e l'input manuale (come

[8] Una finestra modale (traduzione di "modal dialog") è una finestra che costringe l'utente ad interagire con essa prima che si possa tornare indietro ad usare la restante parte dell'applicazione di cui fa parte. Un esempio concreto può essere la finestra di apertura file.

le annotazioni) per riparare i casi di test in ts_0 in modo che possano essere eseguiti (e le cui asserzioni possano essere soddisfatte) in A_1.

2.5.2 Approccio basato su algoritmi genetici

I modelli di event-flow presentati nella sezione 2.5.1 sono efficaci quando bisogna generare brevi sequenze di eventi per il testing. Ad esempio, se una GUI ha cinque eventi, dove ciascun evento può essere eseguito dopo qualsiasi altro, ci sono soltanto $5^2 = 25$ sequenze di lunghezza 2 e solo 125 di lunghezza 3. Sequenze di eventi più lunghe possono far emergere guasti non rilevabili da quelle più brevi [21, 22]; inoltre esse forniscono più contesto e raggiungono codice più complesso nel programma. Bisogna tenere in conto comunque due limitazioni primarie delle sequenze test lunghe. La prima riguarda il numero di sequenze che cresce esponenzialmente con la lunghezza. In un'interfaccia con 5 eventi, se si generano sequenze di lunghezza 10 si hanno potenzialmente fino a $5^{10} = 9.765.625$ sequenze univoche e per una lunghezza di 20 si arriva a $9,5 \times 10^{13}$. Al fine di tenere sotto controllo la crescita esponenziale pur garantendo la stessa copertura, si può applicare alle sequenze una tecnica di campionamento derivata dal *combinatorial interaction testing* (CIT); ad ogni modo la tecnica soffre di una seconda limitazione, ovvero i casi di test *infattibili*. Un caso di test si definisce infattibile se almeno uno dei suoi eventi che dovrebbe essere disponibile durante l'esecuzione del test, nei fatti non lo è. In tale situazione, l'esecuzione del test potrebbe bloccarsi o fallire. L'infattibilità di un caso di test può essere dovuta a difetti nel programma o ad informazioni mancanti nel modello usato per ottenere il caso di test.

In [9] per superare questa seconda limitazione viene proposta una tecnica che prevede la riparazione di una suite di test CIT rimuovendo i casi di test infattibili ed inserendo nuovi casi fattibili per aumentare la copertura, attraverso un processo evolutivo. Il framework in una prima fase costruisce i campioni del CIT e li esegue; in una seconda fase, i test infattibili vengono scartati e viene impiegato un algoritmo genetico[9] per generare nuove sequenze che

[9] Un algoritmo genetico è un algoritmo euristico ispirato al principio della selezione naturale e all'evoluzione biologica.

migliorano la copertura del CIT.

2.5.3 Approccio incrementale

Le tecniche esaminate in altri lavori di ricerca si sono concentrate prevalentemente sul repair dei test considerando i breakage tra successivi rilasci dell'applicazione. L'approccio suggerito in [8] si basa sull'analisi dei cambiamenti a grana più fine che intercorrono durante il processo di evoluzione del software, ovvero sfruttando le potenzialità dei moderni repository di codice[10] (come SVN o Git) che permettono di valutare le differenze tra versioni intermedie di più basso livello (spesso chiamate "commit"). L'approccio WATERFALL consiste nell'applicare iterativamente i test repair tra queste versioni intermedie in modo da ottenere risultati migliori in termini di efficacia. Sebbene l'approccio proposto non implica la scelta di una particolare tecnica di test repair, è stata scelta la tecnica WATER (Web Application Test Repair) [1] che è specificamente pensata per il repair dei locatori.

Ad alto livello, WATER è una tecnica di testing differenziale utilizzata per confrontare le esecuzioni di un test t su due differenti rilasci R e R' dell'applicazione da testare, in cui t viene eseguito correttamente su R ma non su R'. WATER esegue il test t su entrambe le versioni R ed R' e raccoglie i dati di queste esecuzioni; successivamente, esamina le differenze tra R ed R' e, basandosi su tali differenze e sui dati raccolti durante l'esecuzione, utilizza delle euristiche per trovare un insieme di potenziali repair PR per t. Per ciascun potenziale repair $p \in PR$, WATER esegue t utilizzando p. Se t viene eseguito correttamente su R', p viene aggiunto alla lista dei repair suggeriti. Questo passaggio viene ripetuto finché non sono stati considerati tutti i potenziali repair in PR; infine, la lista dei possibili repair viene mostrata agli sviluppatori per essere valutata.

[10] Un repository di codice è un archivio di file (generalmente fruibile in rete) nel quale è conservato il codice sorgente al fine di tenere traccia delle modifiche nel tempo e consentire a più collaboratori di contribuire allo sviluppo.

2.5.4 Approccio White-box

L'approccio presentato in questa sezione è definito white-box perché vengono prese in considerazione le modifiche apportate al codice dell'interfaccia grafica al fine di apportare le dovute correzioni agli script di test. L'agenda di ricerca proposta in [2] prevede tre passaggi distinti per poter implementare il repair automatico dei test:

- **Studio dell'evoluzione dell'interfaccia grafica.** Questo passaggio ha lo scopo di identificare, mediante approccio empirico, i più comuni tipi di cambiamento delle GUI. Studi precedenti sull'evoluzione delle API per i componenti object oriented hanno evidenziato che più dell'80% dei cambiamenti sono refactor delle API che comportano modifiche ai nomi o al posizionamento degli elementi ma non ai comportamenti [3]. Similmente, ci si aspetta che la maggior parte dei cambiamenti alle GUI consistono in refactoring.

- **Supporto automatizzato al refactoring del codice delle GUI.** Lo sviluppo di tecniche e strumenti per gli sviluppatori può facilitare l'evoluzione delle interfacce grafiche; una possibilità concreta risiede nell'estensione degli attuali IDE per supportare i refactoring delle GUI. Per esempio, invece che cambiare manualmente il codice per sostituire dei radio button con una lista, uno sviluppatore potrebbe sfruttare un refactoring automatico per quest'operazione. Così come per tutti gli altri refactoring, l'IDE potrebbe eseguire un'analisi (potenzialmente sia statica che dinamica) al fine di identificare se il refactoring è ammissibile e applicare le dovute trasformazioni al codice.

- **Repairing automatizzato degli script di test.** L'obiettivo principale è semplificare l'evoluzione degli script di test: agevolare l'evoluzione del codice delle GUI tramite il punto precedente è un modo per ottenere delle informazioni necessarie al cambiamento degli script di test. Invece di provare a dedurre i cambiamenti apportati dallo sviluppatore, sarebbe possibile avere una dettagliata sequenza di modifiche apportate:

a questo punto risulterebbe semplice applicare cambiamenti corrispondenti agli script di test esistenti, nello specifico ai riferimenti degli oggetti delle interfacce grafiche coinvolti.

2.5.5 Approccio basato su locatori robusti

Gli strumenti più utilizzati per il testing delle applicazioni web come Selenium WebDriver offrono agli sviluppatori una ricca API per definire casi di test basati sul DOM delle pagine. Utilizzando questa API, gli sviluppatori possono ad esempio localizzare una casella di testo, inserire del contenuto al suo interno, localizzare un pulsante, generare un evento di click su di esso, localizzare il testo che mostra il risultato dell'elaborazione e controllare se esso coincide con il comportamento atteso. Tutti questi passaggi vengono codificati utilizzando un linguaggio di alto livello (come Java) in modo molto simile a quanto fatto con JUnit. I locatori degli elementi giocano un ruolo chiave nel testing; tra tutti, i locatori XPath sono notevolmente potenti e flessibili. Essi rappresentano la scelta più generale, dato che la maggior parte dei locatori può essere specificata delle opportune espressioni XPath. Nella pratica, definire manualmente un'espressione XPath si rivela essere un compito complesso, che può richiedere competenze ed esperienze specifiche; a tal proposito esistono strumenti (come FirePath, XPath Checker, XPath Helper ed altri) in grado di elaborare locatori XPath al posto dell'utente. Tali locatori possono essere espressioni XPath assolute o relative, le quali sfruttano tag e attributi individuati euristicamente per provare ad incrementare la resilienza ai cambiamenti.

Per locatore XPath *robusto* si intende un'espressione che riesce a selezionare l'elemento target anche se la pagina è stata modificata in una successiva versione dell'applicazione. In [11] viene proposto un nuovo algoritmo, chiamato ROBULA (ROBust Locator Algorithm), in grado di prevenire parzialmente e ridurre l'*aging*[11] dei casi di test di applicazioni web, generando automaticamente dei locatori robusti.

[11] Nell'accezione più generale l'aging (traducibile con "invecchiamento") si riferisce alla tendenza del software a fallire o a causare guasti in un sistema, col passare del tempo.

L'algoritmo segue un approccio top-down, iniziando dall'espressione XPath più generale (ad es. "`//*`", che intercetta tutti gli elementi contenuti nella pagina) e specializzandola mediante trasformazioni successive. L'algoritmo prende in input una pagina web e un XPath assoluto, selezionando l'elemento d'interesse della pagina web (ad es. un pulsante, un link, ecc.). Per tale elemento l'algoritmo restituisce un'espressione relativa XPath robusta (se esiste) in grado di selezionare univocamente l'elemento d'interesse. Se non esiste un'espressione XPath relativa, ROBULA restituisce un'espressione assoluta simile a quella ricevuta in input.

2.6 Sistemi di templating per il web

In questa sezione è riportata una panoramica del contesto di applicabilità della soluzione proposta in questo lavoro di tesi: le applicazioni web template-based.

Nel corso del tempo sono state sviluppate diverse tecnologie che permettessero di generare pagine web dinamicamente, come ad esempio i cataloghi di prodotti degli e-commerce o i titoli delle notizie sui quotidiani online, cercando nel contempo di rendere più semplice lo sviluppo di applicazioni web, migliorarne la flessibilità, ridurre i costi di manutenzione e consentire uno sviluppo rapido e possibilmente in parallelo tra la logica di presentazione e quella di business. I template engine hanno fornito un notevole contributo in questo senso, partendo da un'idea molto semplice: separare la specifica della logica di business dalla specifica di come una pagina web mostra tali informazioni. Grazie a questa separazione, i template engine promuovono il riuso dei componenti, l'intercambiabilità di diversi stili grafici, punti singoli di cambiamento per i componenti più comuni e una migliore chiarezza del sistema nel suo complesso.

2.6.1 Evoluzione dei template engine

Generare pagine web dinamiche ha come implicazione primaria che un web server non può semplicemente associare ad una URL un file HTML su disco; piuttosto, il server deve far

corrispondere alle varie URL pezzi di codice che producono in output blocchi di HTML contenenti sia dati che informazioni sullo stile di visualizzazione.

Per citare un esempio, Java ha iniziato a supportare lo sviluppo per il web lato server con le Servlet, le quali consistevano in metodi in grado di rispondere a comandi GET o POST generando in risposta codice HTML attraverso comandi di "print" sullo stream di output verso il client. Ad esempio, il codice seguente genera una semplice pagina HTML contenente la stringa "Hello" concatenata con il parametro dinamico `name` ricevuto nella richiesta HTTP:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
       throws ServletException, IOException {
   response.setContentType("text/html");
   PrintWriter out = response.getWriter();
   out.println("<html>");
   out.println("<body>");
   out.println("<h1>Greeting Servlet</h1>");
   String name = request.getParameter("name");
   out.println("Hello, " + name + ".");
   out.println("</body>");
   out.println("</html>");
}
```

Specificare il codice HTML con quest'approccio è noioso, soggetto ad errori, di difficile lettura e soprattutto non può essere scritto da un utente che si occupa solo di design grafico. Per mitigare il problema si potrebbe ricorrere ad un approccio object oriented per esprimere i concetti di HTML in classi come Table o OrderedList, ma ciò non eviterebbe comunque l'inclusione di HTML nel codice delle Servlet.

Il passaggio successivo dell'evoluzione fu l'introduzione delle Java Server Pages (JSP), che a prima vista sembravano essere un grosso passo in avanti; tuttavia, nella sostanza le JSP non sono altro che Servlet "al contrario", ovvero file HTML con frammenti di codice Java incorporato (le JSP vengono comunque tradotte dal container in Servlet prima di essere eseguite). Traducendo in JSP la stessa pagina dell'esempio precedente, si potrebbe ottenere un risultato simile al codice seguente:

```
<html>
<body>
<h1>Greeting Servlet</h1>
Hello, <%=request.getParameter("name")%>.
</body>
</html>
```

I file JSP possono nascere come file HTML con riferimenti puntuali e mirati ai dati, come nel codice precedente, ma in poco tempo possono nuovamente degenerare in codice HTML misto a Java, così come avviene nelle Servlet. Se ciò avviene (come è vero, nella pratica), i designer addetti alla grafica non possono più lavorare sulle JSP per modificarne solo l'aspetto.

Anche se le JSP non rappresentavano la risposta definitiva, sono state alla base di idee riguardo il templating che hanno iniziato a diffondersi tra gli sviluppatori. Un template si può vedere nella sostanza come un documento HTML in cui vi sono dei "buchi" che possono essere riempiti dai dati o dai risultati di semplici azioni.

2.6.2 Definizione di "template"

Pur effettuando una ricerca in letteratura, non emerge una definizione formale di template, e pertanto ciò viene elencato appunto come uno dei problemi del mercato relativo ai template engine [10]. Un template può definirsi come un documento contenente elementi ed azioni incorporate che il template engine è in grado di valutare quando processa il template per produrre l'output finale.

2.6.3 Motivazioni per il templating

L'obiettivo primario che spinge all'utilizzo dei template engine è sicuramente la separazione netta tra logica di business e la visualizzazione dei dati. Nel caso dello sviluppo web, ciò implica che non dovrebbe esserci codice nell'HTML e, viceversa, non dovrebbe esserci HTML nel codice. Di seguito sono elencati alcuni buoni motivi per i quali è desiderabile tale separazione [17]:

- **Incapsulamento**: l'aspetto di un sito web è completamente confinato nei template e, allo stesso modo, la logica di business è completamente confinata nel modello dati: entrambi sono entità complete ed indipendenti.

- **Chiarezza**: un template non è un *programma* il cui comportamento emergente è una pagina HTML; piuttosto, un template *è* un file HTML che un esperto di grafica o uno sviluppatore possono leggere in maniera diretta.

- **Separazione delle competenze**: un esperto di grafica può creare i template dell'applicazione in maniera completamente parallela rispetto allo sviluppo del codice. Ciò riduce il carico sugli sviluppatori, non solo per aver affidato la parte di template ad un'altra figura professionale, ma soprattutto per la riduzione dei costi di comunicazione: i designer possono affinare la parte HTML senza dover dialogare con gli sviluppatori.

- **Riuso dei componenti**: così come gli sviluppatori sono soliti suddividere metodi complessi in metodi più piccoli e semplici, anche i designer possono riorganizzare i template in vari sotto-template o componenti, come le caselle di ricerca, le tabelle dati, le barre di navigazione e così via.

- **Punti singoli di cambiamento**: essendo in grado di poter riorganizzare i template, i designer possono decidere il giusto grado di astrazione per i diversi elementi, sia per gli elementi più granulari come i collegamenti, i pulsanti e così via, che per quelli più complessi come quelli che mostrano ad esempio i dati anagrafici di un utente. Successivamente, cambiare ad esempio il modo in cui gli utenti appaiono nelle liste richiede semplicemente la modifica del solo template della visualizzazione dati utente; ciò evita anche gli errori introdotti quando bisogna intervenire su punti distinti per cambiare un singolo comportamento.

- **Manutenibilità**: cambiare l'aspetto di un'applicazione web deve richiedere un cambio di template e non di codice; d'altronde, cambiare il comportamento di un programma è sempre molto più rischioso di dover cambiare un template. Inoltre, le modifiche ai template normalmente non richiedono il riavvio di un'applicazione pubblicata su un server, mentre invece i cambiamenti al codice richiedono spesso ricompilazione e riavvio.

Spesso gli sviluppatori percepiscono tale separazione netta tra vista e logica come un costo aggiuntivo. Anche se ciò potrebbe sembrare vero nel caso di piccole applicazioni, in realtà i progetti più grandi riescono a progredire molto più velocemente nel lungo periodo, avvantaggiandosi inoltre di maggiore flessibilità e codice robusto grazie ai benefici appena elencati.

2.6.4 Esempio di utilizzo dei template

In generale, l'utilizzo di un template engine prevede tre fasi:

- Specifica del template da utilizzare.
- Assegnazione dei valori che verranno integrati come contenuto nel template.
- Riempimento del template con i valori assegnati per ottenere il markup HTML risultante.

Pur potendo variare le interfacce di programmazione per ciascun template engine, i passaggi elencati sono comunque sempre presenti. Prendendo in considerazione il template engine Smarty per PHP, nel listato 2.6.1 ne è riportato un esempio di utilizzo minimale. Alla riga 3 viene effettuata l'inclusione del file PHP contenente l'implementazione del template engine e alla riga 5 ne viene creata un'istanza. Le righe 7 e 8 caricano il contenuto di un file (listato 2.6.2) contenente i dati da mostrare nella pagina e la riga 11 passa tali dati all'istanza del template engine. Alla riga 14 viene invocato il metodo che effettua il rendering del template (listato 2.6.3) utilizzando i dati forniti in precedenza.

```
1  <?php
2
3  include('Smarty/libs/Smarty.class.php');
4
5  $smarty = new Smarty;
6
7  $jsonContent = file_get_contents("test.data.json");
8  $array = json_decode($jsonContent, true);
9  foreach ($array as $key => $value)
10 {
11     $smarty->assign($key, $value);
12 }
```

```
13
14 $smarty->display('sample.tpl');
15
16 ?>
```

Listato 2.6.1: Utilizzo del template engine Smarty

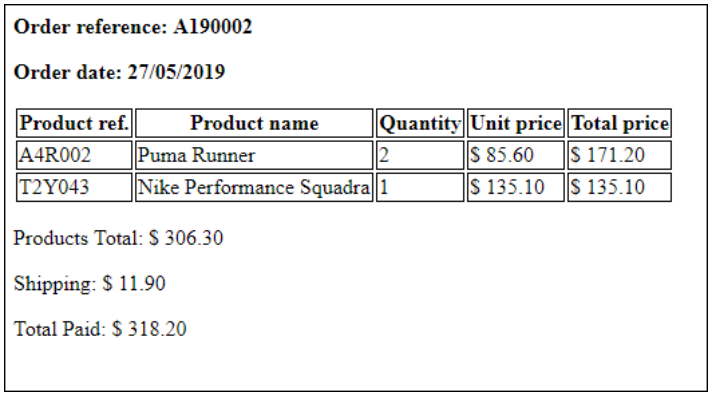
```
1  {
2      "order" :{
3          "data": {
4              "reference": "A190002", "order_date": "27/05/2019", "total_products_wt": "$ 306.30",
5              "total_shipping": "$ 11.90", "total_paid": "$ 318.20" },
6          "products": [
7              { "product_reference": "A4R002", "product_name": "Puma Runner",
8                "product_quantity": 2, "unit_price": "$ 85.60", "total_price": "$ 171.20" },
9              { "product_reference": "T2Y043", "product_name": "Nike Performance Squadra",
10               "product_quantity": 1, "unit_price": "$ 135.10", "total_price": "$ 135.10" }
11         ]
12     }
13 }
```

Listato 2.6.2: Dati di esempio passati al template engine

```
1  <html>
2  <body>
3  <div>
4      <p><strong>Order reference: {$order.data.reference}</strong></p>
5      <p><strong>Order date: {$order.data.order_date}</strong></p>
6
7      <table>
8
9          <tr>
10             <th>Product ref.</th>
11             <th>Product name</th>
12             <th>Quantity</th>
13             <th>Unit price</th>
14             <th>Total price</th>
15         </tr>
16
17         {foreach from=$order.products item=product}
18         <tr>
19             <td>{$product.product_reference}</td>
20             <td>{$product.product_name}</td>
21             <td>{$product.product_quantity}</td>
22             <td>{$product.unit_price}</td>
23             <td>{$product.total_price}</td>
24         </tr>
25         {/foreach}
26
27     </table>
28
29     <p>Products Total: {$order.data.total_products_wt}</p>
30     <p>Shipping: {$order.data.total_shipping}</p>
31     <p>Total Paid: {$order.data.total_paid}</p>
32 </body>
33 </html>
```

Listato 2.6.3: Template Smarty

Nella figura 2.6.1 è riportato il risultato del rendering grafico del template, mentre nel listato 2.6.4 è riportato il codice HTML risultante.



Order reference: A190002

Order date: 27/05/2019

Product ref.	Product name	Quantity	Unit price	Total price
A4R002	Puma Runner	2	$ 85.60	$ 171.20
T2Y043	Nike Performance Squadra	1	$ 135.10	$ 135.10

Products Total: $ 306.30

Shipping: $ 11.90

Total Paid: $ 318.20

Figura 2.6.1: Risultato visivo del rendering

```
<html>
<body>
<div>
    <p><strong>Order reference: A190002</strong></p>
    <p><strong>Order date: 27/05/2019</strong></p>

    <table>
        <tr>
            <th>Product ref.</th>
            <th>Product name</th>
            <th>Quantity</th>
            <th>Unit price</th>
            <th>Total price</th>
        </tr>

                <tr>
            <td>A4R002</td>
            <td>Puma Runner</td>
            <td>2</td>
            <td>$ 85.60</td>
            <td>$ 171.20</td>
        </tr>
                <tr>
            <td>T2Y043</td>
            <td>Nike Performance Squadra</td>
            <td>1</td>
            <td>$ 135.10</td>
            <td>$ 135.10</td>
        </tr>

    </table>

    <p>Products Total: $ 306.30</p>
    <p>Shipping: $ 11.90</p>
    <p>Total Paid: $ 318.20</p>
</body>
```

```
37 </html>
```

Listato 2.6.4: Risultato del rendering: codice HTML

Nell'esempio mostrato, l'utilizzo del template engine Smarty consiste essenzialmente nell'invocazione di due soli metodi: `assign()` e `display()`. Il metodo `assign()` viene invocato per assegnare valori ai parametri in input (in questo caso provenienti da un file di dati esterno), mentre il metodo `display()` viene invocato sia per specificare il template da utilizzare che per riempirlo coi valori passati. Nell'esempio mostrato nel listato 2.6.3, il parametro `$order` contiene due campi, `data` e `products`, che corrispondono ai dati strutturati del listato 2.6.2. Il campo `products` è un'array che viene iterato mediante la direttiva `foreach` di Smarty: in questo modo è possibile replicare parti del template per creare delle liste.

2.6.5 Considerazioni sui template engine

Anche se un linguaggio di templating è essenzialmente un linguaggio di programmazione, i template hanno senso soltanto nel caso in cui siano utilizzati con lo scopo di semplificare la specifica delle viste: se si incorpora logica in maniera eccessiva, tale semplificazione viene meno. Nella pratica, molti template engine come Smarty implementano linguaggi Turing-completi [18] e possono pertanto far perdere i vantaggi del loro uso se non si rispettano alcune regole basilari riguardo la separazione delle responsabilità.

Capitolo 3

Rilevazione e riduzione dei test breakage

3.1 Introduzione

In questo capitolo sono riportate le idee alla base del processo e degli strumenti a supporto dello sviluppo che consentono di rilevare e ridurre i breakage degli script di test per le interfacce grafiche. Preliminarmente viene descritto un processo di sviluppo durante il quale si introducono dei test breakage a causa di modifiche alle viste; viene quindi mostrato l'impatto in termini temporali che tale condizione ha sul processo di sviluppo.

3.2 Scenario di test breakage

In figura 3.2.1 è riportato uno schema semplificato di un processo di sviluppo; i test automatici (unit tests, integration tests, end to end tests, ecc.), se previsti, vengono lanciati immediatamente dopo gli sviluppi per verificare che non siano stati introdotti bug sulle feature preesistenti (regression tests). Per quanto riguarda i test capture/replay per le interfacce grafiche, possono essere registrati nuovi test per coprire le nuove feature implementate.

Durante le operazioni di manutenzione successive si può verificare che qualche modifica apportata alle viste produca guasti nei test che sono stati registrati. Un esempio di modifica potrebbe essere la reingegnerizzazione di un componente grafico per renderlo più generico, al fine di riutilizzarlo anche nelle nuove feature; si supponga quindi che tale cambiamento comporti il breakage di uno o più test che ne prevedevano l'utilizzo.

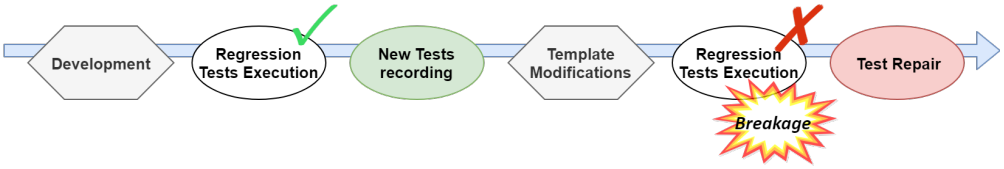


Figura 3.2.1: Test breakage durante il processo di sviluppo

Più in generale, durante tutto il processo di evoluzione del software, a seguito delle modifiche effettuate lo sviluppatore esegue tutti i test per verificare che non siano state prodotte regressioni; tra le varie problematiche che possono verificarsi, alcune modifiche possono comportare guasti ai test precedentemente registrati.

Su base intuitiva, si può ipotizzare che **il numero di breakage** b_C introdotti dalla singola modifica ad un template sia inferiore o al più uguale al numero di breakage b_F introdotti nel completamento dell'intero sviluppo (che può comprendere modifiche a template multipli). Verosimilmente, se la suite di test viene interrotta ad ogni fallimento, detto t_B **il tempo medio che intercorre tra l'inizio dell'esecuzione dei test e il verificarsi di un fallimento dovuto a breakage**, si ha che nel primo caso il tempo di rilevamento totale di tutti i breakage è $b_C \cdot t_B$, mentre nel secondo caso è $b_F \cdot t_B$.

A questo punto, si consideri che lo sviluppatore deve:

- preliminarmente indagare sul problema, per capire in ciascun caso se sono state prodotte regressioni; sia quindi t_I il **tempo medio di indagine** necessario per risalire al breakage come causa del fallimento;
- provvedere alla riparazione dei test; sia quindi t_R il **tempo medio di repair** per ciascun test;
- rieseguire nuovamente i test per verificare che ciascun repair non comporti ulteriori effetti collaterali (sia come regressioni, sia come ulteriori breakage); se l'intervento di repair è andato a buon fine, verrà rilevato il successivo fallimento dovuto a breakage.

Fatte queste considerazioni, nel caso peggiore si avrà che il tempo totale impiegato è almeno pari a $b_F \cdot (t_B + t_I + t_R)$. In figura 3.2.2 è riportato il processo di rilevazione e riparazione descritto con i tre contributi temporali.

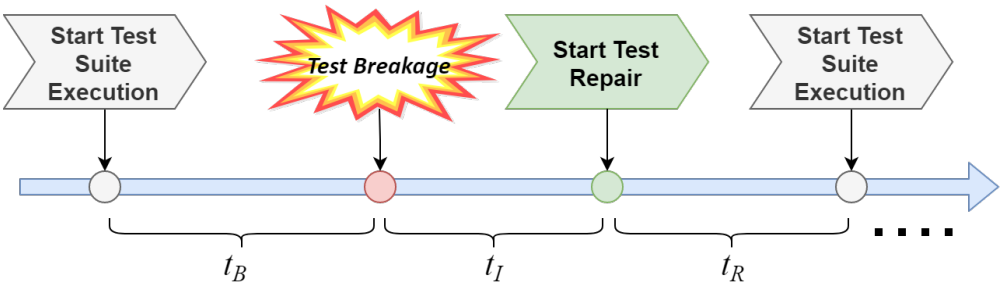


Figura 3.2.2: Test breakage: tempo di rilevazione e riparazione

3.2.1 Esempio di test breakage su applicazioni web

Quanto detto in precedenza può applicarsi a qualsiasi contesto architetturale e tecnologico; le differenze in termini temporali possono ovviamente essere non trascurabili tra i diversi ambiti che si riscontrano nella realtà. In questa sezione verrà illustrato un esempio di test breakage, esaminando il contesto specifico delle applicazioni web.

Si consideri la struttura di un'interfaccia grafica web in termini di albero del DOM: in figura 3.2.3, a sinistra è rappresentato un semplice DOM di esempio.

Durante la fase di registrazione (capture) per la creazione di un caso di test, l'utente ha modo di interagire con i componenti della pagina, generando alcuni eventi (i_1 e i_2 in figura), e di inserire dei punti di controllo nel test mediante asserzioni (a_1 in figura). Al termine della registrazione del test, lo strumento di capture produce uno script nel formato deciso dall'utente. Nella figura 3.2.3 è anche riportato lo pseudo-codice dello script di test: sono stati considerati eventi ed asserzioni, e i locatori necessari ad individuare gli elementi interessati dal caso di test, sfruttando la sintassi XPath. Lo script di esempio è stato tradotto in formato XML nel listato 3.2.1.

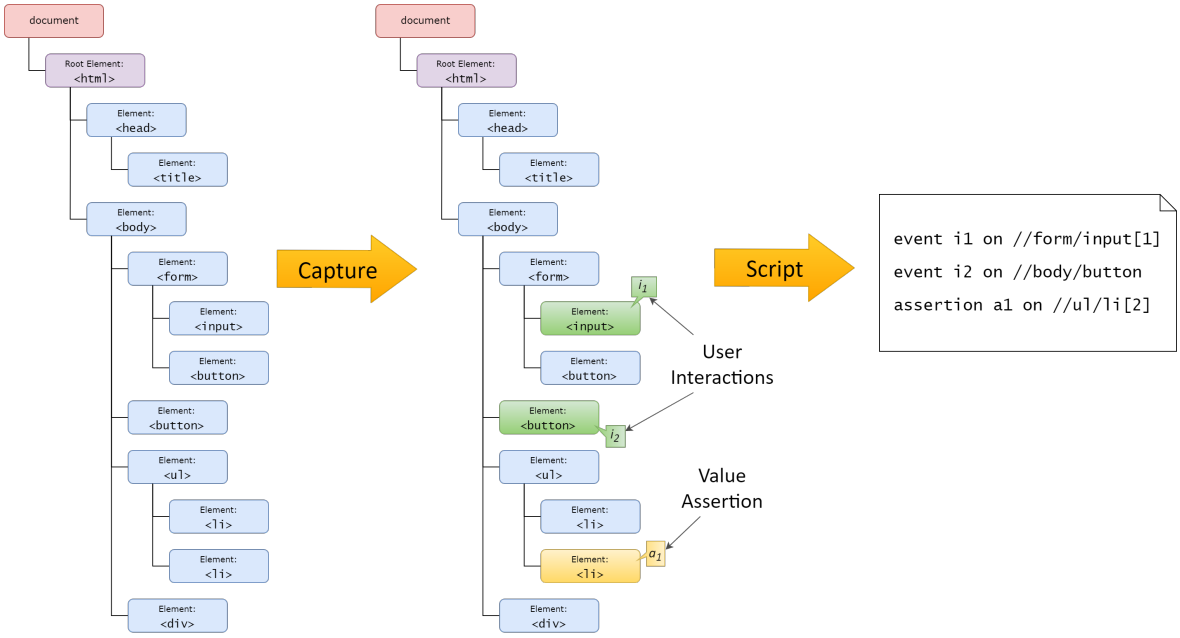


Figura 3.2.3: DOM: Registrazione script di test

```xml
<?xml version="1.0" encoding="UTF-8"?>
<TestCase>
<selenese>
    <command>type</command>
    <target>//form/input[0]</target>
    <value>test-user</value>
</selenese>
<selenese>
    <command>click</command>
    <target>//button</target>
    <value></value>
</selenese>
<selenese>
    <command>assertText</command>
    <target>//ul/li[1]</target>
    <value>test-data</value>
</selenese>
</TestCase>
```

Listato 3.2.1: Esempio di test script XML

Si supponga che a questo punto il layout della View subisca una modifica che comporta il riposizionamento e/o la sostituzione di qualche tag, come mostrato in figura 3.2.4.

Come evidenziato, tra la revisione R_i e la successiva R_{i+1}, le espressioni dei locatori XPath presenti nello script di test registrato in precedenza non sono più utilizzabili per identificare gli elementi, pertanto una successiva esecuzione dei test terminerebbe con l'interruzione di questo script.

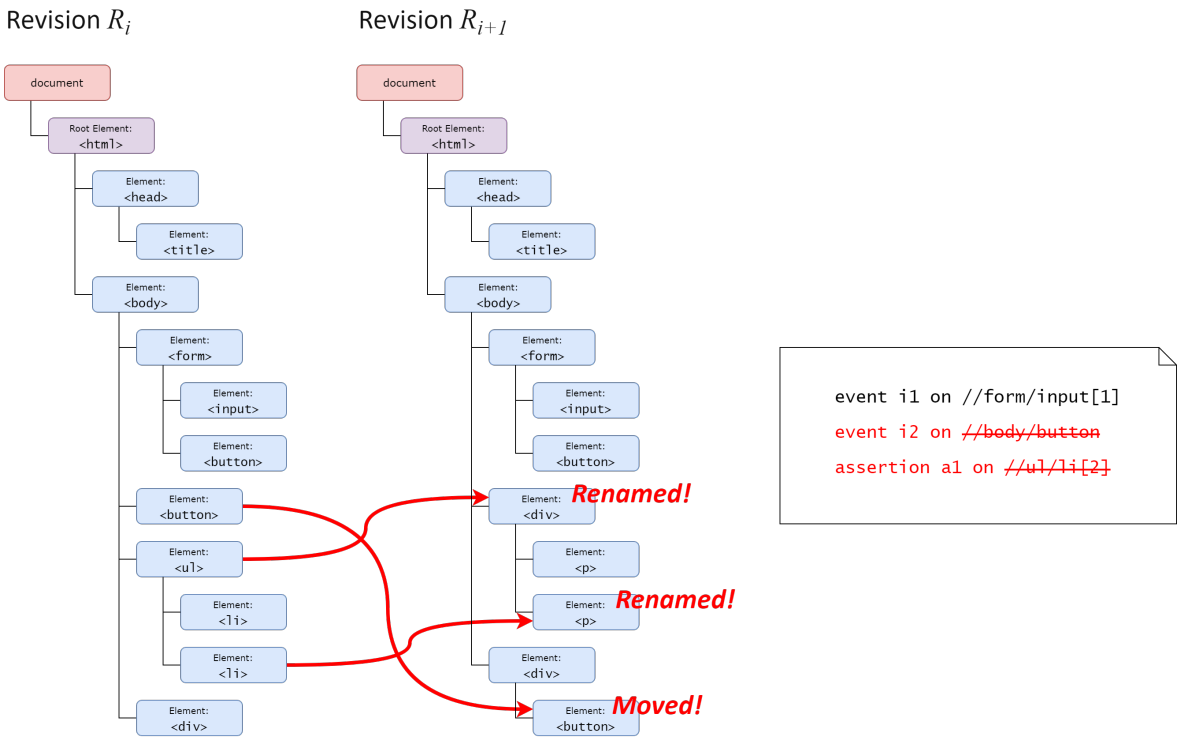


Figura 3.2.4: Modifica del DOM

3.2.2 Considerazioni

Partendo dall'esempio mostrato in figura 3.2.3 si possono effettuare alcune considerazioni con l'obiettivo di individuare in seguito un metodo di rilevazione precoce dei test breakage, riducendo conseguentemente la somma dei tempi t_B, t_I e t_R definiti nella sezione 3.2.

Modalità di implementazione del componente View

Seguendo l'approccio **programmatico**, gli elementi dell'interfaccia grafica possono essere creati, posizionati e stilizzati utilizzando direttamente un linguaggio di programmazione (spesso può essere lo stesso linguaggio utilizzato per scrivere la logica di business). Dato che non sempre è possibile imporre una separazione netta del codice che si occupa della costruzione delle viste dal resto dell'applicazione, può verificarsi un aumento dei tempi t_I e t_R, dato che **le differenze tra il codice su cui i test funzionano e quello successivo alle modifiche che causano breakage, non hanno corrispondenza uno-a-uno con le differenze**

che sussistono tra le viste a runtime. Pertanto, pur potendo individuare quale parte della pagina web ha subito il cambiamento che ha impatto sui test (mediante l'individuazione dei locatori non funzionanti e ispezione del DOM), non è detto che sia altrettanto immediato ricondurre tali modifiche al codice. Riassumendo, non esiste in generale un modo affidabile per individuare il frammento di codice modificato che ha reso i locatori del test non più validi, pertanto d'ora in avanti l'approccio programmatico non verrà considerato.

Seguendo l'approccio **dichiarativo**, su cui si fondano i template, si utilizza un linguaggio di markup dedicato (come HTML, XHTML, XML, YAML, ecc.) eventualmente in combinazione con altri linguaggi di stilizzazione (come CSS) per definire la struttura, il posizionamento e gli attributi dei componenti dell'interfaccia grafica. Se, come capita con gran parte dei template engine per il web, il codice HTML è alla base dei template, l'individuazione del locatore non funzionate **può più facilmente ricondursi ai frammenti di template modificati**. Anche in questo caso però, non esiste un metodo deterministico (automatico o manuale) per poter riparare i locatori non funzionanti, dato che alcuni elementi potrebbero aver subito modifiche o spostamenti tali da non consentire di rilevare e riparare facilmente il breakage (solo in alcuni casi particolari è possibile superare tale problema, come ad esempio in presenza di attributi "ID" o "name", a patto che siano non ambigui). Intuitivamente, anche se in misura minore rispetto all'approccio programmatico, quest'aspetto influisce comunque sui tempi t_I e t_R.

Ambiguità dovute all'innestamento dei template

Ai fini del riuso del codice e della suddivisione in componenti, è possibile utilizzare tecniche di inclusione o estensione (a seconda dell'implementazione del template engine) che consentono di comporre i template a partire da quelli più piccoli. Purtroppo, non avendo a runtime un metodo efficace per individuare i punti di innestamento tra i vari template, **a runtime possono verificarsi delle condizioni di ambiguità nei percorsi XPath dei locatori**, a valle di modifiche nella struttura. Anche quest'aspetto influisce sui tempi t_I e t_R.

Quando effettuare il lancio dei test

Per individuare rapidamente un caso di test breakage, avendo ipoteticamente a disposizione risorse di calcolo illimitate, basterebbe semplicemente lanciare tutti i test **a valle di ogni singola modifica**. Purtroppo, in condizioni di sviluppo reali su un'applicazione non banale, ciò non solo **non è praticabile**, ma sarebbe addirittura controproducente rispetto al processo "tradizionale", dato che il tempo t_B verrebbe moltiplicato per il numero di modifiche effettuate, andando di fatto ad azzerare completamente il tempo a disposizione per lo sviluppo. D'altro canto, se esistesse però un modo efficiente per avere indicazioni riguardo l'impatto delle modifiche nello stesso istante in cui vengono apportate, si potrebbero potenzialmente azzerare t_B e t_I, riducendo se possibile anche t_R.

L'obiettivo primario è la riduzione del tempo di indagine t_I, del tempo di rilevamento del breakage t_B e del tempo richiesto per il repair t_R.

3.3 Processo di rilevazione e riparazione

In questa sezione è presentato un processo di sviluppo e manutenzione che comprende alcuni step aggiuntivi, implementati in buona parte da strumenti automatici, tenendo in conto le considerazioni esposte nella sezione 3.2.2 e alcune idee già adottate in altri lavori presentati nella sezione 2.5. I passaggi aggiuntivi del processo si fondano sulle seguenti considerazioni:

- La difficoltà insita nella rilevazione dei test breakage è dovuta essenzialmente alla mancanza di "riferimenti" **espliciti** e **non ambigui** tra la struttura dei template e i locatori dei test. L'idea alla base della tecnica di rilevazione dei test breakage si focalizza primariamente sulla risoluzione di questo problema, creando appunto dei "riferimenti" persistenti tra i template e i test, in maniera del tutto automatica.

- Supponendo di avere a disposizione dei riferimenti bidirezionali tra i template e i test, in linea di principio è possibile stabilire in maniera deterministica **quali test possono**

essere potenzialmente impattati dalla modifica di uno specifico template. Questo concetto è mutuato dalle funzioni di "find usages" o "find references" (presenti negli IDE per i linguaggi come Java), che consentono di individuare in quali parti del codice è referenziato l'elemento analizzato. Anche questo passaggio è automatico.

- A valle di ciascuna modifica effettuata, grazie anche al punto precedente, è possibile individuare esattamente quali test diventerebbero non funzionanti, potendo ricostruire l'innestamento che verrebbe generato a runtime, ma **senza dover eseguire i test**. Anche questo passaggio è automatico.

- Una volta individuato il breakage prodotto dalla modifica, grazie alla presenza di riferimenti espliciti tra template e test, è possibile determinare più semplicemente qual è la correzione da apportare per rendere di nuovo i test funzionanti. Il repair, infine, è un passaggio manuale.

In figura 3.3.1 è riportata una formalizzazione del processo di sviluppo di nuovi template e di modifica di template esistenti, arricchito con step aggiuntivi, scaturiti dalle considerazioni fatte in precedenza e suddividendo i compiti manuali da attività svolte da strumenti automatici.

Di seguito il dettaglio per ogni step, seguendo la numerazione riportata nella figura:

1. Nel caso di creazione di nuovi template, lo sviluppatore procede con l'implementazione come da specifica.

2. Prima di registrare i test, in ciascun file di template tutti gli elementi vengono contrassegnati mediante un riferimento univoco, sfruttando quelle caratteristiche del linguaggio di markup che non hanno effetti sulla visualizzazione o sul funzionamento degli elementi grafici.

3. Si effettua la registrazione dei test, rendendo lo strumento di capture consapevole della presenza dei riferimenti creati al punto precedente, in modo che possa generare locatori basati sui riferimenti stessi.

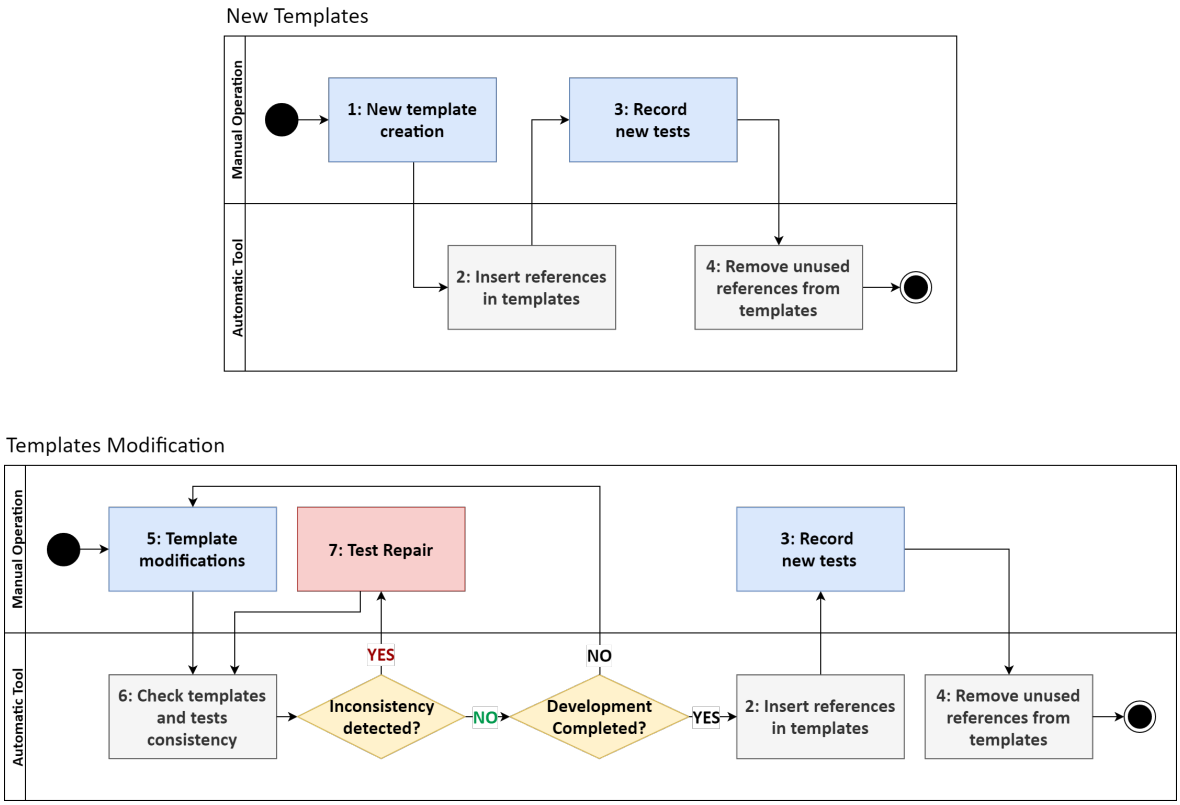


Figura 3.3.1: Processo per il rilevamento e il repair dei test breakage

4. Una volta raccolti tutti i riferimenti realmente utilizzati nei locatori dei test, si effettua la rimozione dai template di tutti quei riferimenti che non sono stati usati nei test.

5. Lo sviluppatore esegue attività di manutenzione, implementando nuove feature, risolvendo bug, e così via. In questa fase i template possono essere creati, rimossi o modificati.

6. A seguito di ogni modifica apportata al codice al punto precedente, si utilizza uno strumento automatico che controlla la consistenza tra i template e i test sfruttando i riferimenti presenti in entrambi.

7. Nel caso in cui vengono rilevati breakage, lo strumento segnala la condizione allo sviluppatore che provvede al repair del test.

Il processo viene reiterato finché non terminano le attività di sviluppo propedeutiche al rilascio, momento in cui verranno comunque rilanciati tutti i test.

3.4 Strumenti automatici di supporto al processo

A seguire verrà mostrata la descrizione algoritmica degli strumenti automatici richiesti e in seguito un esempio d'uso, prima in un caso "base" che contempla un solo template e successivamente in un caso più generale, nel quale si presenta anche la problematica di innestamento di template multipli, con o senza iterazioni.

Per la corretta esecuzione del processo esposto e degli strumenti automatici è essenziale che siano sempre fissate le precondizioni sui dati coinvolti nei test automatici.

3.4.1 Premessa

La possibilità di implementare strumenti automatici a supporto del processo di sviluppo così come proposto nelle sezioni precedenti, implica la necessità di intervenire sul test recorder per poter imporre la strategia di generazione dei locatori basata sui riferimenti inseriti nei template. Non in tutti i casi è possibile personalizzare lo strumento di capture/replay, ma nel caso dell'implementazione di Katalon Recorder, ciò è consentito grazie al caricamento di opportuni script di estensione; nel capitolo successivo verranno mostrati i dettagli tecnici.

3.4.2 Descrizione degli algoritmi proposti

Avendo presentato alcuni esempi di funzionamento della tecnica proposta, è riportata qui una descrizione degli algoritmi a supporto dei vari passaggi automatici del processo.

Injection degli hook nei template

Questo passaggio consiste nell'injection dei test hook e nella creazione della successiva release di test R_i':

1. Si effettua il calcolo dell'identificativo più alto utilizzato nei casi di test registrati precedentemente e presenti pertanto nella release R_i; ciò avviene visitando tutti i template T_i e tutti gli elementi $e_{ij} \in T_i$ individuando qual è il valore massimo degli identificativi tra tutti gli hook presenti. Tale valore è chiamato `maxHookId`.

2. Si inizializza il contatore degli identificativi degli hook con $K = \texttt{maxHookId} + 1$.

3. Si effettua il parsing di ciascun template T_i al fine di risalire alla struttura ad albero A_i dei nodi del template.

4. Per ciascun nodo e_{ij} dell'albero A_i, in maniera **ricorsiva**:

   (a) Viene aggiunto il **template hook** con identificativo K se non è già presente un altro hook e l'elemento e_{ij} è una radice del template.

   (b) Viene aggiunto il **test hook** con identificativo K se non è già presente un altro hook e l'elemento e_{ij} **non** è una radice del template.

   (c) Si effettua l'incremento $K \Leftarrow K + 1$.

Registrazione dei test con locatori hook-based

Questo passaggio consiste nella registrazione dei casi di test, aggiungendo al recorder di Selenium lo script di locator builder per la generazione di XPath basato sui test hook. Partendo dall'elemento foglia e su cui viene prodotto l'evento o l'asserzione:

1. Si inizializzano le variabili seguenti:

   (a) Path basato sugli hook: `pathLocator := ""`

   (b) Condizione di innestamento dei template: `includesChildTemplate := false`

2. Se l'elemento corrente e è contrassegnato con un **template hook** (quindi è una radice) si valorizza il flag `isTemplateRoot` a `true`, altrimenti a `false`.

3. Detto S_e l'insieme ordinato di tutti i nodi vicini ad e (ovvero che condividono lo stesso nodo genitore di e, escluso e stesso) nell'albero DOM, se $S_e \not\equiv \varnothing$ si assegna alla variabile `index` la posizione di e all'interno di S_e.

4. Si valutano le seguenti condizioni (per la minimizzazione del percorso basato sugli hook):

   (a) `pathLocator == ""`: è stato individuato il primo hook utile alla costruzione dell'espressione.

   (b) $S_e \not\equiv \varnothing$: esistono uno o più elementi vicini con lo stesso hook; questo caso si verifica quando vi sono liste di elementi ripetuti.

   (c) `isTemplateRoot == true`: il test hook corrente è posto su un elemento radice di un template.

   (d) `includesChildTemplate == true`: il test hook dell'iterazione precedente era un elemento radice di un template e pertanto è necessario esplicitare anche il test hook corrente nel `pathLocator`, dato che lo stesso template figlio potrebbe essere incluso in punti diversi del template padre.

5. Se **almeno una** delle condizioni elencate è vera si effettuano le operazioni seguenti:

   (a) Si genera il locatore `elemLocator` per l'elemento e, utilizzando l'hook presente sull'elemento e la variabile `index` se valorizzata.

   (b) Si effettua la concatenazione in testa `pathLocator = elemLocator + pathLocator`.

   (c) Si assegna alla variabile `includesChildTemplate` il valore corrente della variabile `isTemplateRoot`. Tale variabile sarà utilizzata nell'iterazione successiva.

6. Se l'elemento e non è la radice del DOM, si assegna ad e il suo nodo genitore; si riprende l'esecuzione dal punto 2.

7. Una volta raggiunta la radice, l'algoritmo termina e la variabile `pathLocator` contiene il percorso composto dai locatori basati su hook per l'elemento d'interesse.

Il test recorder, a questo punto, dopo aver invocato il locator builder per la costruzione dell'espressione XPath basata sugli hook (e tutti gli altri locator builder inclusi di default nella configurazione), procede a validarlo, ovvero effettua una verifica delle seguenti condizioni:

- Il locator ottenuto deve effettivamente reperire un elemento della vista corrente
- Il locator ottenuto non deve essere ambiguo, ovvero deve reperire non più di un elemento della vista corrente

Rimozione degli hook inutilizzati

Questo passaggio consiste nella rimozione di tutti i test hook (iniettati in precedenza) che non sono stati utilizzati in nessuno dei casi di test registrati. L'obiettivo di questa procedura è la minimizzazione dell'impatto che gli hook hanno sul codice sorgente delle viste: sarebbe perfettamente lecito lasciare tutti gli hook, ma ciò causerebbe inutili complicazioni durante la manutenzione dei template, dato che si desidera evidenziare mediante gli hook **soltanto quegli elementi effettivamente coinvolti nei casi di test** e non l'intera struttura della vista. L'algoritmo per effettuare la rimozione degli hook inutilizzati prevede come input i casi di test registrati; Detto H_t l'insieme di tutti gli hook coinvolti nei test e H_p l'insieme di tutti gli hook presenti nei template, per ogni template T_i e per ogni elemento $e_{ij} \in T_i$ vengono rimossi tutti quegli hook che fanno parte dell'insieme differenza $H_p \setminus H_t$. A valle di questo step è possibile creare la release R_i''.

Controllo di consistenza tra template e test

Questo passaggio si innesta durante la manutenzione del codice delle viste, utilizzando lo strumento di verifica di consistenza degli hook tra i test e il codice sorgente. La modalità proposta in questo lavoro di tesi è di tipo *offline*, ovvero deve essere **avviata manualmente dall'utente**. Durante questa fase l'utente può essere avvisato non appena si verifica una condizione di breakage dei test dovuta alla modifica del codice. Lo strumento di verifica implementa un algoritmo come quello riportato di seguito:

1. Si effettua il parsing di ciascun template T_i al fine di risalire alla struttura ad albero A_i dei nodi del template.

2. Esaminando tutti gli alberi A_i, detto H_p l'insieme di **tutti gli hook** presenti nei template, si costruisce la corrispondenza γ: $H_p \rightarrow H_p$ tra ciascun **template hook** r_i (radice di A_i) e i **test hook** h_{ix} sottoposti gerarchicamente ad r_i all'interno dell'albero.

3. Per ciascun caso di test, si estrae l'insieme di locatori utilizzati. In particolare si considera ciascun locatore L composto da una sequenza di k hook $[h_1, h_2 \ldots, h_k]$ (dove k è variabile per ciascun locatore) che individuano a runtime il percorso nell'albero DOM dell'elemento interessato dall'evento o dall'asserzione.

4. Per ogni locatore L, si itera ciascun hook in $h_l \in [h_1, h_2 \ldots, h_k]$ di L:

   (a) Se h_l è un **template hook**, si controlla se h_l appartiene al dominio della corrispondenza γ, ovvero se esiste un template contenente una radice contrassegnata con template hook h_l. Se ciò è verificato, si valorizza la variabile `currentRoot` con h_l, altrimenti viene **segnalato l'errore** di template hook non trovato.

   (b) Se h_l è un **test hook** si controlla se vi è una corrispondenza tra `currentRoot` e h_l mediante γ. In altre parole, si controlla che h_l sia gerarchicamente sottoposto (all'interno del template) all'ultima radice `currentRoot` trovata nel locatore L. Se tale corrispondenza non viene trovata, viene **segnalato l'errore** che indica sia il locatore L che la radice `currentRoot`, fornendo quindi un alto livello di dettaglio sia sul test impattato che sul template che causa il breakage per la mancanza dell'hook h_l.

Infine, a valle delle attività di manutenzione, avendo contestualmente corretto eventuali breakage, è possibile creare la release R_{i+1}.

3.4.3 Caso base: template singolo

In quest'esempio verrà considerata una pagina web ottenuta da un singolo template, riutilizzando l'albero DOM già mostrato in figura 3.2.3. Si supponga di poter marcare automaticamente tutti gli elementi mediante degli *hook* univoci, prima della registrazione degli script di test; in figura 3.4.1 è riportata la marcatura della revision R_i del DOM considerato in precedenza. Avendo ottenuto la revision R'_i a valle dell'operazione di *hook injection*, è possibile eseguire normalmente la registrazione dello script di test, istruendo opportunamente lo strumento di capture affinché sia sensibile agli hook aggiunti al DOM, per la creazione dei locatori del test al posto delle espressioni XPath; in figura 3.4.2 è mostrato il flusso di registrazione basato sugli hook.

A valle della registrazione del caso di test, gli hook che non sono interessati dal test possono essere rimossi dal codice sorgente della pagina, generando conseguentemente la revision R''_i.

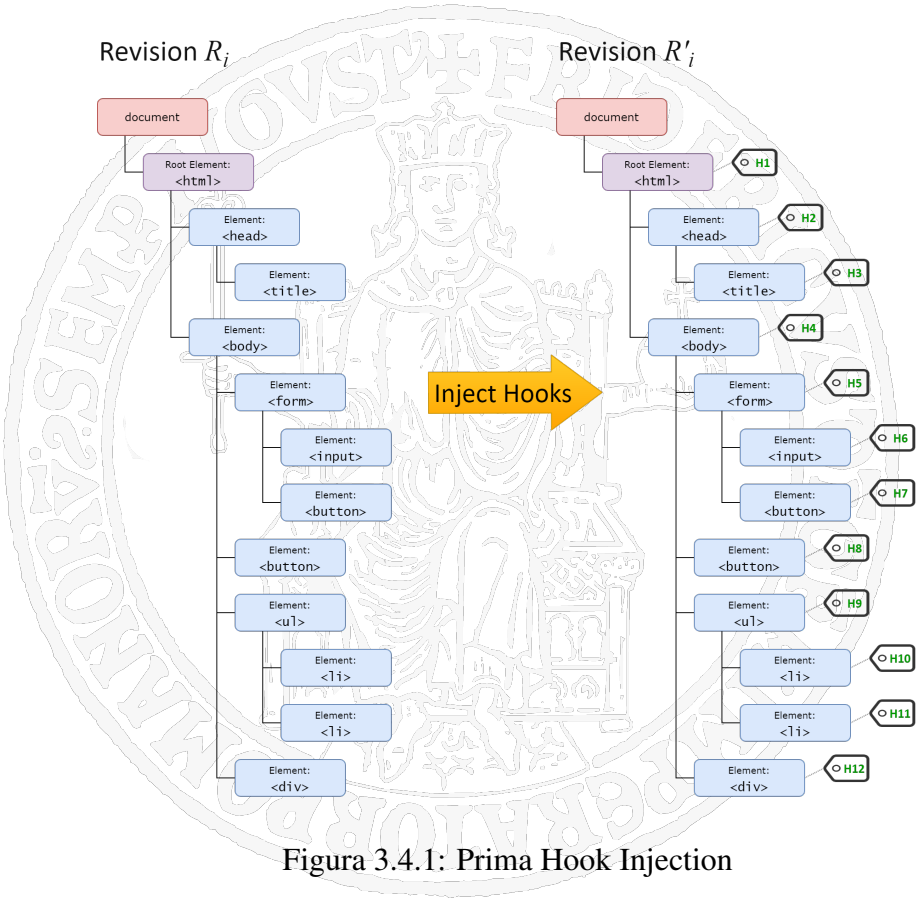


Figura 3.4.1: Prima Hook Injection

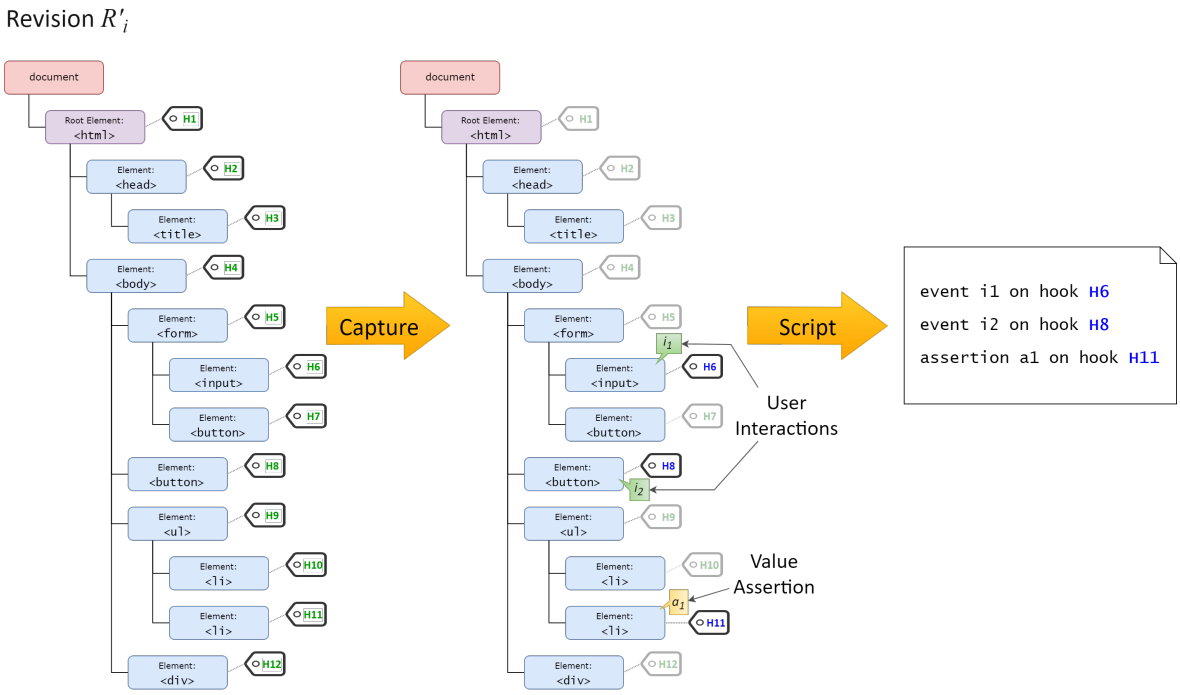


Figura 3.4.2: Hook-based capture

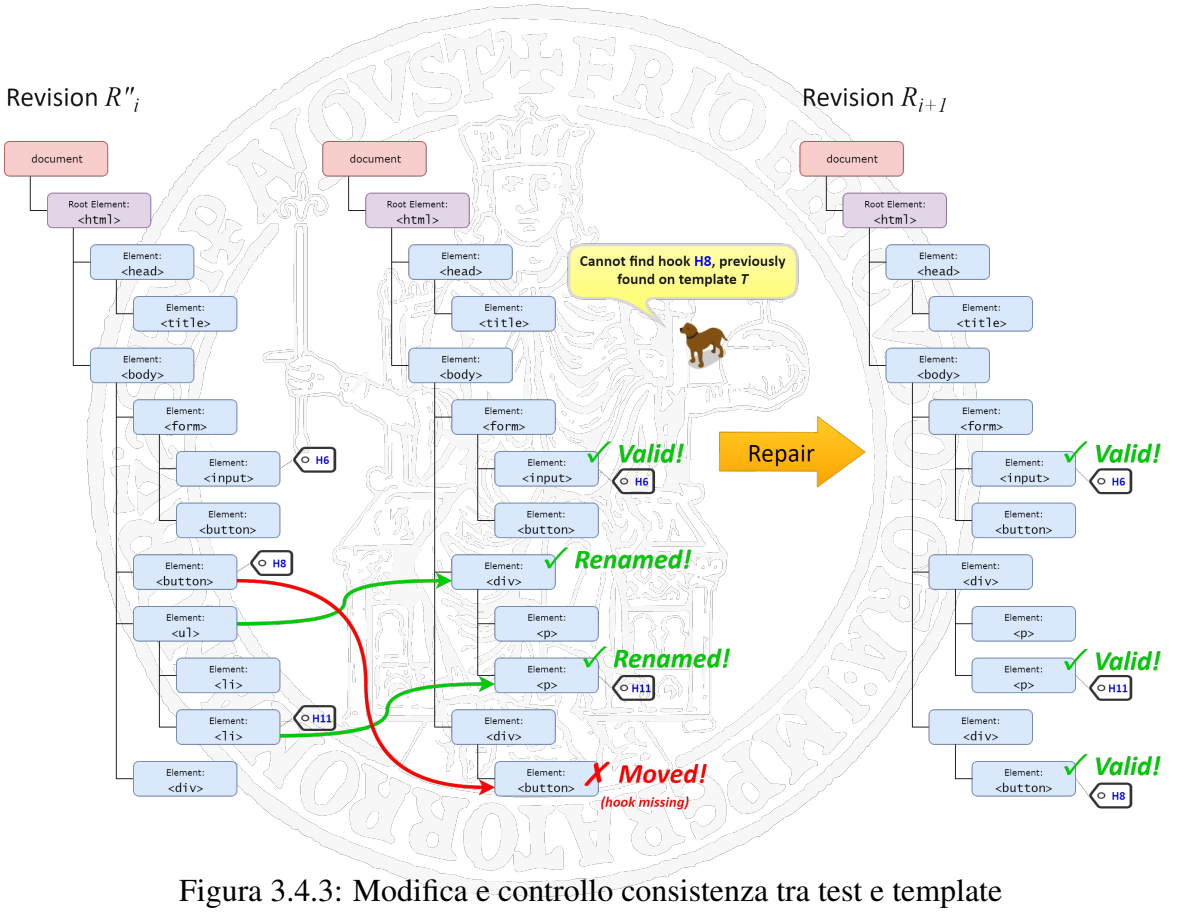


Figura 3.4.3: Modifica e controllo consistenza tra test e template

Durante la fase di manutenzione del codice, si supponga di effettuare un controllo automatico di consistenza, che tiene traccia di tutti gli hook utilizzati in ogni script di test e del codice sorgente HTML dei template dell'applicazione web. Dopo ogni modifica, esso può essere usato per verificare se vi è qualche test reso inutilizzabile e in tal caso notificare immediatamente lo sviluppatore, che può provvedere manualmente al reinserimento dell'hook rimosso durante le modifiche apportate, oppure alla correzione dello script test, ove necessario.

I passaggi essenziali della processo possono essere riassunti nei punti seguenti:

- Injection dei test hook nel codice della revision R_i, creazione della revision R'_i.
- Registrazione dei test utilizzando i locatori basati sugli hook.
- Rimozione degli hook non utilizzati dai test e creazione revision R''_i.
- Modifica delle viste sulla revision R''_i
- Controllo di consistenza automatico, seguito da eventuale repair.
- Rilascio nuova revision R_{i+1} (non contenente modifiche che comportano test breakage).

Nelle successive revisioni è possibile iterare nuovamente i passaggi precedenti, sotto le seguenti ipotesi:

- Persistenza: il meccanismo di hook injection non altera gli hook preesistenti
- Univocità: gli identificativi dei nuovi hook non collidono con quelli preesistenti

In figura 3.4.4 è mostrata la revision R_{i+1} e la successiva R'_{i+1} contenente i nuovi hook.

3.4.4 Caso generale: template multipli

Nella sezione precedente, sfruttando la conoscenza degli hook presenti nei test e confrontandoli con quelli nei template, è stata introdotta l'idea di base per poter rilevare in anticipo i

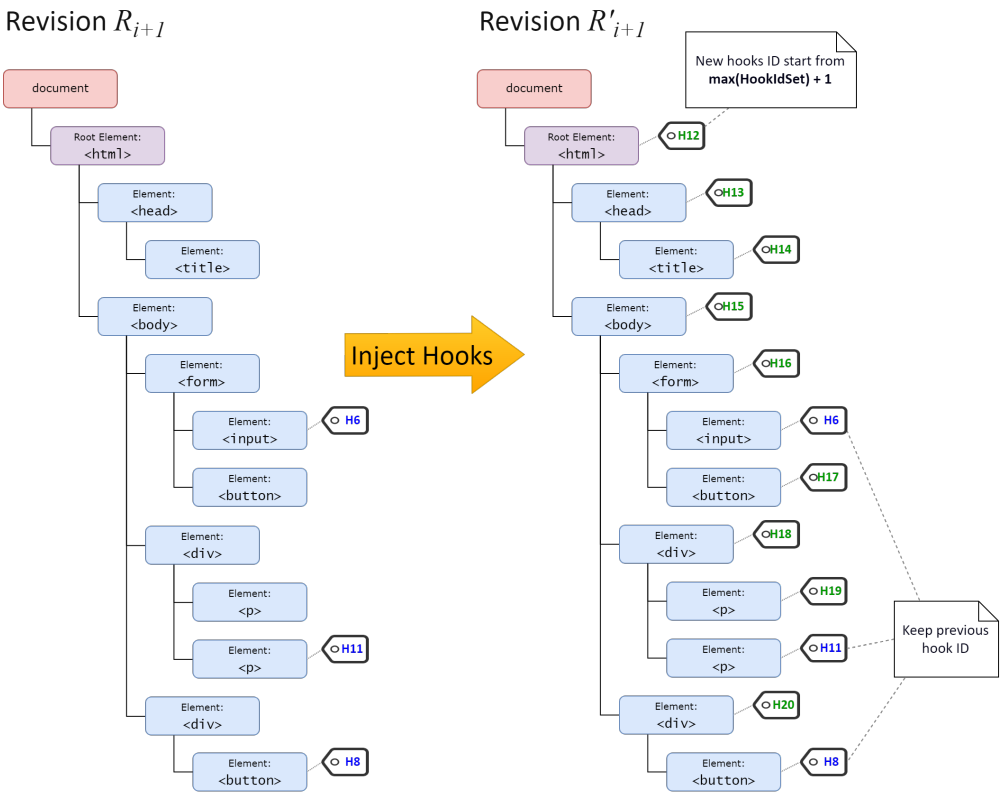


Figura 3.4.4: Successiva Hook Injection

breakage dei test per quelle viste strutturate in un unico template. Ciò purtroppo non costituisce il caso generale, dato che in diverse implementazioni dei template engine è possibile organizzare il codice in componenti riutilizzabili, dove ciascuno di essi ha un proprio template, o può essere assemblato a partire da altri componenti. Pertanto, pur inserendo degli hook globalmente univoci all'interno dei template, a tempo di esecuzione alcuni hook potrebbero essere istanziati più volte all'interno della pagina (o addirittura in maniera ricorsiva), rendendo ambigui i riferimenti presenti negli script di test (questo problema è già stato considerato nella sezione 3.2.2).

Per ovviare a questo problema è sufficiente adottare due semplici contromisure:

- Template hook: oltre a marcare tutti gli elementi dei template con i test hook, gli elementi "radice" di ciascun template devono essere marcati con uno speciale "template hook".

- I locatori hook-based utilizzati negli script di test devono tener conto nella gerarchia

anche dei template hook oltre all'hook di test dello specifico elemento di interesse, sfruttando dove necessario degli indici per identificare un'istanza specifica di un template se quest'ultimo è stato replicato più volte.

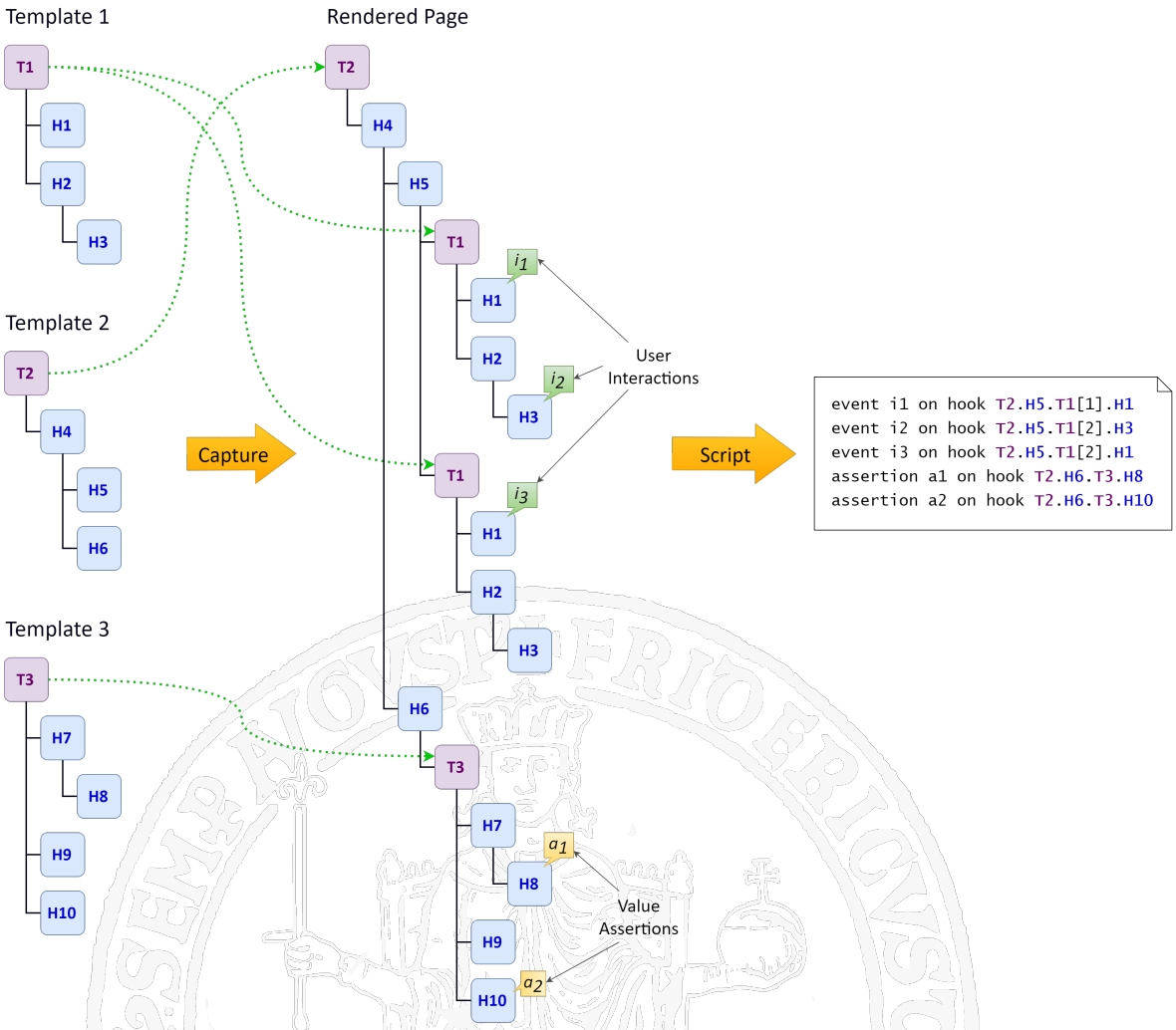


Figura 3.4.5: Hook Injection: template multipli

In figura 3.4.5 è mostrato il risultato della hook injection applicata a tre template distinti, ciascuno avente il proprio template hook radice (per brevità gli elementi sono stati privati del tag ed identificati soltanto dal proprio hook). A tempo di esecuzione, il template engine provvede ad istanziare e innestare opportunamente i vari frammenti, formando il layout finale della pagina; a questo punto si può avviare la registrazione del caso di test. Come

si può osservare in figura, al fine di evitare qualsiasi ambiguità riguardo le espressioni dei locatori hook-based in fase di esecuzione del test, si è utilizzata una sorta di *dot notation* per indicare univocamente ciascun elemento mediante gli hook; a tal proposito, si noti che non è necessario comporre l'intero percorso nella gerarchia, ma è **sufficiente per ciascun template l'hook radice ed il "punto di innesto"** sul template padre, proprio grazie all'univocità degli identificativi degli hook di test.

Per quanto riguarda le asserzioni è anche possibile utilizzare i quantificatori universali o esistenziali nel caso si voglia far riferimento ad elementi all'interno di template (o gerarchie di template) replicati. Ad esempio, se un'asserzione deve essere verificata su tutti gli elementi aventi l'hook `H2` nel layout finale della pagina, è possibile omettere l'indice associato al template `T1` ed utilizzare l'operatore $\forall$, ottenendo un'espressione del tipo: `assertion a3 forall T2.H5.T1.H2`.

3.5 Esempio reale di processo di sviluppo e manutenzione

Pur non avendo ancora mostrato un'implementazione degli strumenti automatici a supporto del processo, è possibile mostrare il funzionamento della tecnica di rilevazione dei breakage applicato ad una funzionalità di una semplice applicazione web basata sul framework MVVM AngularJS. Esistono diverse tecnologie che consentono di implementare una SPA[1] in maniera molto simile a quanto possibile con AngularJS, come ad esempio EmberJS, VueJS ed altre. Quanto mostrato in questa sezione si può riproporre facilmente su applicazioni sviluppate in altri contesti tecnologici simili.

La principale innovazione del framework AngularJS è la gestione del two-way binding tra la View e il ViewModel in maniera completamente trasparente allo sviluppatore, con la possibilità di personalizzare in alcuni casi particolari la direzione del binding per favorire le

[1] Acronimo di "Single Page Application", ovvero un'applicazione web che non richiede ricaricamenti per passare da una schermata all'altra, ma effettua dinamicamente sostituzioni di parti della pagina lato client, manipolando il DOM tramite script.

performance o per rispettare vincoli di design. Il listato 3.5.1 mostra il codice di uno stato[2] per la visualizzazione di una lista di attività (è stato omesso il codice del ViewModel per brevità) e dello stato di modifica e aggiunta di un'attività, mentre in figura 3.5.1 sono mostrate le relative interfacce utente.

```
<!-- TASK LIST STATE -->

<ui-view>
    <div class="form-group">
        <button type="button" class="btn btn-default" ui-sref=".edit">New</button>
    </div>
    <table class="table">
        <thead>
            <tr>
                <th style="width:35%">Task description</th>
                <th style="width:25%">Assigned to</th>
                <th style="width:10%">Status</th>
                <th style="width:30%">Actions</th>
            </tr>
        </thead>
        <tbody>
            <tr ng-repeat="task in ctrl.tasks">
                <td ng-class="{strikethrough : !!task.done}">{{::task.description}}</td>
                <td ng-class="{strikethrough : !!task.done}">{{::task.user}}</td>
                <td>
                    <span class="label label-default" ng-if="!task.done">To Do</span>
                    <span class="label label-success" ng-if="!!task.done">Done</span>
                </td>
                <td>
                    <div class="form-group">
                        <button type="button" class="btn btn-xs btn-success check-mark"
                                ng-if="!task.done"
                                ng-click="task.done = true">Mark as Done</button>
                        <button type="button" class="btn btn-xs btn-default pencil"
                                ng-if="!!task.done"
                                ng-click="task.done = false">Mark as To Do</button>
                        <button type="button" class="btn btn-xs btn-default pencil"
                                ng-if="!task.done"
                                ui-sref=".edit({taskIndex: $index})">Edit</button>
                        <button type="button" class="btn btn-xs btn-danger ballot-mark"
                                ng-click="ctrl.remove(task)">Remove</button>
                    </div>
                </td>
            </tr>
        </tbody>
    </table>
</ui-view>

<!-- TASK EDIT STATE -->

<div class="form-group">
    <form>
        <div class="form-group">
            <label>Task description:</label>
```

[2]Nell'ambito delle Single Page Application, si ci riferisce alle schermate come "stati", ovvero particolari aree dell'applicazione che incapsulano sia il layout che la logica di presentazione. Possono essere tra loro innestati mediante punti di ingresso espliciti, in modo da organizzare le funzionalità dell'applicazione in un albero.

```
50            <input type="text" class="form-control" ng-model="ctrl.task.description">
51        </div>
52        <div class="form-group">
53            <label>Assigned to:</label>
54            <select ng-options="user.name as user.name for user in ctrl.users"
55                    ng-model="ctrl.task.user" class="form-control"></select>
56        </div>
57        <button type="button" class="btn btn-default" ng-click="ctrl.save()">Save</button>
58        <button type="button" class="btn btn-default" ui-sref="^">Cancel</button>
59    </form>
60 </div>
```

Listato 3.5.1: Template: Gestione Task

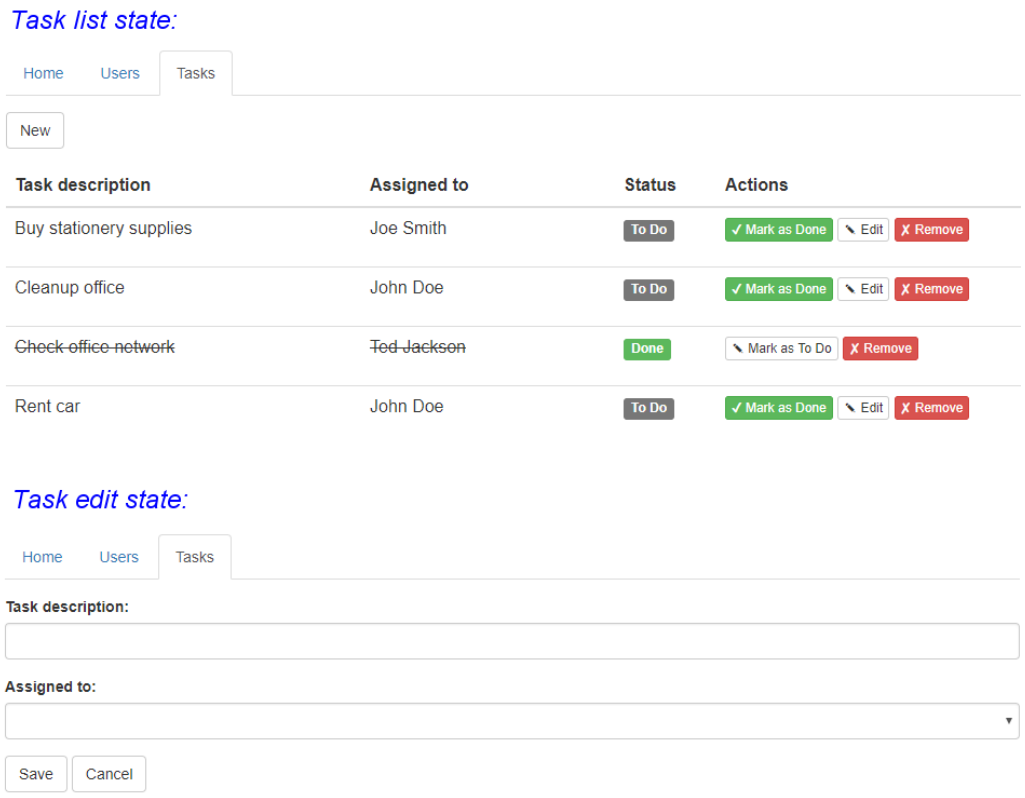


Figura 3.5.1: Interfaccia di Gestione Task

Hook Injection: In questa prima fase, il codice sorgente delle viste viene arricchito mediante l'inserimento dei test hook (prefisso `x-hook-*`) e degli hook radice dei template (prefisso `x-tpl-*`). Gli hook sono stati aggiunti come attributo `data-*` degli elementi HTML, dato che quest'operazione non ha alcun effetto collaterale sul comportamento o sull'aspetto

visivo dell'applicazione. A valle dell'operazione di injection si ottiene il codice riportato nel listato 3.5.2.

```
<!-- TASK LIST STATE -->

<ui-view data-x-tpl-40="">
    <div class="form-group" data-x-hook-41="">
        <button type="button" class="btn btn-default" ui-sref=".edit" data-x-hook-42="">New</button>
    </div>
    <table class="table" data-x-hook-43="">
        <thead data-x-hook-44="">
            <tr data-x-hook-45="">
                <th style="width:35%" data-x-hook-46="">Task description</th>
                <th style="width:25%" data-x-hook-47="">Assigned to</th>
                <th style="width:10%" data-x-hook-48="">Status</th>
                <th style="width:30%" data-x-hook-49="">Actions</th>
            </tr>
        </thead>
        <tbody data-x-hook-50="">
            <tr ng-repeat="task in ctrl.tasks" data-x-hook-51="">
                <td ng-class="{strikethrough : !!task.done}" data-x-hook-52="">{{::task.description}}</td>
                <td ng-class="{strikethrough : !!task.done}" data-x-hook-53="">{{::task.user}}</td>
                <td data-x-hook-54="">
                    <span class="label label-default" ng-if="!task.done" data-x-hook-55="">To Do</span>
                    <span class="label label-success" ng-if="!!task.done" data-x-hook-56="">Done</span>
                </td>
                <td data-x-hook-57="">
                    <div class="form-group" data-x-hook-58="">
                        <button type="button" class="btn btn-xs btn-success check-mark"
                                ng-if="!task.done"
                                ng-click="task.done = true" data-x-hook-59="">Mark as Done</button>
                        <button type="button" class="btn btn-xs btn-default pencil"
                                ng-if="!!task.done"
                                ng-click="task.done = false" data-x-hook-60="">Mark as To Do</button>
                        <button type="button" class="btn btn-xs btn-default pencil"
                                ng-if="!task.done"
                                ui-sref=".edit({taskIndex: $index})" data-x-hook-61="">Edit</button>
                        <button type="button" class="btn btn-xs btn-danger ballot-mark"
                                ng-click="ctrl.remove(task)" data-x-hook-62="">Remove</button>
                    </div>
                </td>
            </tr>
        </tbody>
    </table>
</ui-view>

<!-- TASK EDIT STATE -->

<div class="form-group" data-x-tpl-30="">
    <form data-x-hook-31="">
        <div class="form-group" data-x-hook-32="">
            <label data-x-hook-33="">Task description:</label>
            <input type="text" class="form-control" ng-model="ctrl.task.description" data-x-hook-34="">
        </div>
        <div class="form-group" data-x-hook-35="">
            <label data-x-hook-36="">Assigned to:</label>
            <select ng-options="user.name as user.name for user in ctrl.users"
                    ng-model="ctrl.task.user" class="form-control" data-x-hook-37=""></select>
        </div>
        <button type="button" class="btn btn-default" ng-click="ctrl.save()"
                data-x-hook-38="">Save</button>
        <button type="button" class="btn btn-default" ui-sref="^" data-x-hook-39="">Cancel</button>
```

```
60      </form>
61  </div>
```

Listato 3.5.2: Hook injection

Test Capture: A questo punto viene eseguita la registrazione del caso di test, configurando lo strumento di capture/replay affinché utilizzi gli hook per identificare gli elementi della pagina. Lo strumento Katalon Recorder a tal proposito consente di specificare un file Javascript contenente l'estensione necessaria per rilevare gli hook. Il listato 3.5.3 mostra il risultato dell'operazione di capture. Si noti l'asserzione descritta a partire dalla riga 43: trascurando gli stati superiori nella gerarchia, il percorso di selezione della cella contenente la descrizione del task è composto da soli due hook (`x-tpl-40` e `x-hook-55`). Se ci fosse stato bisogno di identificare una cella appartenente ad una riga specifica della tabella, sarebbe bastato specificare l'hook intermedio `x-hook-55` aggiungendo il relativo indice; bisogna sottolineare che quest'operazione viene comunque eseguita in automatico dallo strumento di capture in tutti i casi in cui vi siano elementi vicini (siblings) marcati con lo stesso hook a tempo di esecuzione.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <TestCase>
3  <selenese>
4      <command>open</command>
5      <target>http://localhost:8080/#!/main/tasks</target>
6      <value></value>
7  </selenese>
8  <selenese>
9      <command>click</command>
10     <target>//*[@x-tpl-18] //*[@x-hook-27] //*[@x-tpl-40] //*[@x-hook-42]</target>
11     <value></value>
12 </selenese>
13 <selenese>
14     <command>click</command>
15     <target>//*[@x-tpl-18] //*[@x-hook-27] //*[@x-tpl-40] //*[@x-tpl-30] //*[@x-hook-34]</target>
16     <value></value>
17 </selenese>
18 <selenese>
19     <command>type</command>
20     <target>//*[@x-tpl-18] //*[@x-hook-27] //*[@x-tpl-40] //*[@x-tpl-30] //*[@x-hook-34]</target>
21     <value>Test task</value>
22 </selenese>
23 <selenese>
24     <command>click</command>
25     <target>//*[@x-tpl-18] //*[@x-hook-27] //*[@x-tpl-40] //*[@x-tpl-30] //*[@x-hook-37]</target>
26     <value></value>
27 </selenese>
28 <selenese>
```

```
    <command>select</command>
    <target>//*[@x-tpl-18] //*[@x-hook-27] //*[@x-tpl-40] //*[@x-tpl-30] //*[@x-hook-37]</target>
    <value>label=Joe Smith</value>
</selenese>
<selenese>
    <command>click</command>
    <target>//*[@x-tpl-18] //*[@x-hook-27] //*[@x-tpl-40] //*[@x-tpl-30] //*[@x-hook-37]</target>
    <value></value>
</selenese>
<selenese>
    <command>click</command>
    <target>//*[@x-tpl-18] //*[@x-hook-27] //*[@x-tpl-40] //*[@x-tpl-30] //*[@x-hook-38]</target>
    <value></value>
</selenese>
<selenese>
    <command>verifyElementPresent</command>
    <target>//*[@x-tpl-18] //*[@x-hook-27] //*[@x-tpl-40] //*[@x-hook-52 and text() = 'Test task']</target>
    <value></value>
</selenese>
<selenese>
    <command>verifyElementPresent</command>
    <target>//*[@x-tpl-18] //*[@x-hook-27] //*[@x-tpl-40] //*[@x-hook-55 and text() = 'To Do']</target>
    <value></value>
</selenese>
</TestCase>
```

Listato 3.5.3: Registrazione test di aggiunta task

Rimozione hook inutilizzati: Al fine di minimizzare il livello di invasività sul codice sorgente, si possono rimuovere tutti gli hook non coinvolti nel caso di test registrato, ottenendo il codice del listato 3.5.4.

```
<!-- TASK LIST STATE -->

<ui-view data-x-tpl-40="">
    <div class="form-group">
        <button type="button" class="btn btn-default" ui-sref=".edit" data-x-hook-42="">New</button>
    </div>
    <table class="table">
        <thead>
            <tr>
                <th style="width:35%">Task description</th>
                <th style="width:25%">Assigned to</th>
                <th style="width:10%">Status</th>
                <th style="width:30%">Actions</th>
            </tr>
        </thead>
        <tbody>
            <tr ng-repeat="task in ctrl.tasks">
                <td ng-class="{strikethrough : !!task.done}" data-x-hook-52="">{{::task.description}}</td>
                <td ng-class="{strikethrough : !!task.done}">{{::task.user}}</td>
                <td>
                    <span class="label label-default" ng-if="!task.done" data-x-hook-55="">To Do</span>
                    <span class="label label-success" ng-if="!!task.done">Done</span>
                </td>
                <td>
                    <div class="form-group">
                        <button type="button" class="btn btn-xs btn-success check-mark"
```

```
                                ng-if="!task.done"
                                ng-click="task.done = true">Mark as Done</button>
                        <button type="button" class="btn btn-xs btn-default pencil"
                                ng-if="!!task.done"
                                ng-click="task.done = false">Mark as To Do</button>
                        <button type="button" class="btn btn-xs btn-default pencil"
                                ng-if="!task.done"
                                ui-sref=".edit({taskIndex: $index})">Edit</button>
                        <button type="button" class="btn btn-xs btn-danger ballot-mark"
                                ng-click="ctrl.remove(task)">Remove</button>
                    </div>
                </td>
            </tr>
        </tbody>
    </table>
</ui-view>

<!-- TASK EDIT STATE -->

<div class="form-group" data-x-tpl-30="">
    <form>
        <div class="form-group">
            <label>Task description:</label>
            <input type="text" class="form-control" ng-model="ctrl.task.description" data-x-hook-34="">
        </div>
        <div class="form-group">
            <label>Assigned to:</label>
            <select ng-options="user.name as user.name for user in ctrl.users"
                    ng-model="ctrl.task.user" class="form-control" data-x-hook-37=""></select>
        </div>
        <button type="button" class="btn btn-default" ng-click="ctrl.save()"
                data-x-hook-38="">Save</button>
        <button type="button" class="btn btn-default" ui-sref="^">Cancel</button>
    </form>
</div>
```

Listato 3.5.4: Rimozione hook inutilizzati

Modifica della vista: Si supponga di dover effettuare a questo punto un'operazione di manutenzione della vista, al fine di migliorare l'esperienza utente. Ad esempio, si voglia dare la possibilità di effettuare l'aggiunta e la modifica di elementi della tabella con la modalità *inline*, senza dover ricorrere ad una schermata dedicata. In questo caso, la modifica consiste quindi nello spostare i campi di input collocandoli opportunamente nel template della lista dei task ed eliminare il template di edit. Una volta eseguita tale operazione, si ottiene il codice riportato nel listato 3.5.5.

```
<!-- TASK LIST STATE -->

<ui-view data-x-tpl-40="">
    <div class="form-group">
        <button type="button" class="btn btn-default" ui-sref=".edit" data-x-hook-42="">New</button>
    </div>
```

```
7     <table class="table">
8         <thead>
9             <tr>
10                <th style="width:35%">Task description</th>
11                <th style="width:25%">Assigned to</th>
12                <th style="width:10%">Status</th>
13                <th style="width:30%">Actions</th>
14            </tr>
15        </thead>
16        <tbody>
17            <tr ng-repeat="task in ctrl.tasks">
18                <td ng-if="!task.$$editing" ng-class="{strikethrough : !!task.done}"
19                    data-x-hook-52="">{{::task.description}}</td>
20
21                <td ng-if="task.$$editing">
22                    <input type="text" class="form-control" ng-model="task.description" data-x-hook-34="">
23                </td>
24
25                <td ng-if="!task.$$editing" ng-class="{strikethrough : !!task.done}">{{::task.user}}</td>
26
27                <td ng-if="task.$$editing">
28                    <select ng-options="user.name as user.name for user in ctrl.users"
29                            ng-model="task.user" class="form-control" data-x-hook-37=""></select>
30                </td>
31
32                <td>
33                    <span class="label label-default" ng-if="!task.done" data-x-hook-55="">To Do</span>
34                    <span class="label label-success" ng-if="!!task.done">Done</span>
35                </td>
36
37                <td ng-if="!task.$$editing">
38                    <div class="form-group">
39                        <button type="button" class="btn btn-xs btn-success check-mark"
40                                ng-if="!task.done"
41                                ng-click="task.done = true">Mark as Done</button>
42                        <button type="button" class="btn btn-xs btn-default pencil"
43                                ng-if="!!task.done"
44                                ng-click="task.done = false">Mark as To Do</button>
45                        <button type="button" class="btn btn-xs btn-default pencil"
46                                ng-if="!task.done"
47                                ui-sref=".edit({taskIndex: $index})">Edit</button>
48                        <button type="button" class="btn btn-xs btn-danger ballot-mark"
49                                ng-click="ctrl.remove(task)">Remove</button>
50                    </div>
51                </td>
52
53                <td ng-if="task.$$editing">
54                    <div class="form-group">
55                        <button type="button" class="btn btn-default"
56                                ng-click="ctrl.save(task)" data-x-hook-38="">Save</button>
57                        <button type="button" class="btn btn-default"
58                                ng-click="ctrl.cancelEdit(task)">Cancel</button>
59                    </div>
60                </td>
61            </tr>
62        </tbody>
63    </table>
64 </ui-view>
```

Listato 3.5.5: Refactoring del template

Controllo di consistenza: All'atto del salvataggio, lo strumento di controllo consistenza segnala che i comandi dello script del listato 3.5.3 dalla riga 13 alla riga 42 non sono più validi, dato che l'hook `x-tpl-30` non esiste più a causa della rimozione del template di edit. Avendo però riportato tutti i componenti senza rimuovere i loro hook (`x-hook-34` alla riga 22, `x-hook-37` alla riga 29 e `x-hook-38` alla riga 56 del listato 3.5.5), è sufficiente cancellare dallo script di test i riferimenti all'hook mancante per riottenere il test funzionante.

Capitolo 4

Implementazione prototipale

La realizzazione di un prototipo che implementa gli algoritmi a supporto del processo presentato nel capitolo precedente ha attraversato diverse fasi. Preliminarmente si è deciso di adottare un approccio verticalizzato su una singola tecnologia di templating (e pertanto più semplice e veloce da progettare e implementare), che consentisse di validare rapidamente le idee di base. Successivamente, dopo aver verificato la bontà delle ipotesi fatte attraverso alcuni esperimenti ad-hoc, si è deciso di rendere il prototipo più generico, per poter effettuare infine una sperimentazione in condizioni realistiche.

4.1 Background tecnologico

Nei diversi passaggi analizzati nella sezione 3.4.2, è emersa più volte la necessità di dover manipolare la struttura dei template delle pagine web. Sia per la fase di hook injection che per la rimozione e il controllo di coerenza degli hook, si rende quindi indispensabile un supporto tecnologico per astrarre gli aspetti legati alla lettura, modifica e salvataggio dei template. Un altro passaggio fondamentale consiste nella lettura ed elaborazione degli script di test, dai quali estrarre i path basati sugli hook, al fine di effettuare i controlli di coerenza coi template. Allo scopo di velocizzare la realizzazione prototipale, avendo la possibilità di esportare gli script di test anche in formato XML o HTML, è possibile ricondurre la problematica del caricamento e lettura degli script a quella dei template, nel caso in cui entrambi sfruttino lo stesso formato.

La scelta del template engine sul quale provare il primo prototipo è ricaduta sul framework

AngularJS (ne è riportata una breve descrizione in A.1), dato che consente di implementare i template in markup HTML puro (avendolo inoltre già utilizzato estensivamente in ambito professionale).

Per l'implementazione del prototipo la scelta è ricaduta sulla piattaforma NodeJS (descritto in A.2.2), che consente di sviluppare applicazioni in linguaggio JavaScript senza aver bisogno del browser; è inoltre possibile reperire facilmente librerie e strumenti adatti all'elaborazione di template. Le versioni più recenti di JavaScript supportate da NodeJS lasciano piena libertà allo sviluppatore riguardo al paradigma da adottare, che può essere sia Object Oriented (grazie alla presenza del costrutto class e dell'ereditarietà prototipale) che funzionale (grazie ai costrutti lambda, le closures, le funzioni di ordine superiore, e così via), o un misto di entrambi. Ciò consente di esprimere con grande flessibilità i diversi componenti dell'applicazione, sfruttando inoltre le caratteristiche di tipizzazione dinamica del linguaggio.

La libreria JavaScript scelta per la lettura degli script di test in formato HTML è JSDOM (descritta in A.3), che espone un'API conforme allo standard DOM. Nella prima implementazione prototipale, questa libreria è stata utilizzata sia per elaborare i template AngularJS (trattandosi di HTML puro) che per la lettura degli script Selenium generati da Katalon Recorder. Nella seconda implementazione prototipale è stato utilizzato il parser FreeMate (basato sul formato TextMate, descritto in A.4) per elaborare template basati su altre tecnologie come Twig, Smarty e altri.

4.2 Architettura

L'architettura del prototipo è organizzata in modo da suddividere ciascun passaggio automatizzato mostrato nel processo in figura 3.3.1 in diversi componenti.

1. **Hook Injector**: è il componente che si occupa di iniettare gli hook in tutti gli elementi dei template (punto 2 del processo).

2. **Hook Cleaner**: è il componente che si occupa di rimuovere gli hook dai template che non sono utilizzati nei test (punto 4 del processo).

3. **Hook Checker**: è il componente che effettua il controllo di coerenza tra i template e i test (punto 6 del processo).

4. **Test Suite Crawler**: è il componente che effettua la lettura di tutti script di test per individuare gli hook effettivamente utilizzati (punto 6 del processo).

Oltre a questi componenti principali vi è il **Command Dispatcher** che ha il ruolo di esporre un'interfaccia unica (a riga di comando) per l'interazione con l'utente; il **TextMate Parser** e le relative grammatiche sono usati per interpretare i diversi linguaggi di templating; il componente **JSDOM** è utilizzato per il caricamento degli script di test in formato HTML. È previsto anche un componente separato (**Hook Locator Builder**) da caricare nello strumento di capture/replay, per consentire la generazione dei locatori basati sugli hook durante la registrazione dei test (il dettaglio è riportato nella sezione 4.3.4). In figura 4.2.1 è riportato un diagramma di alto livello dei componenti.

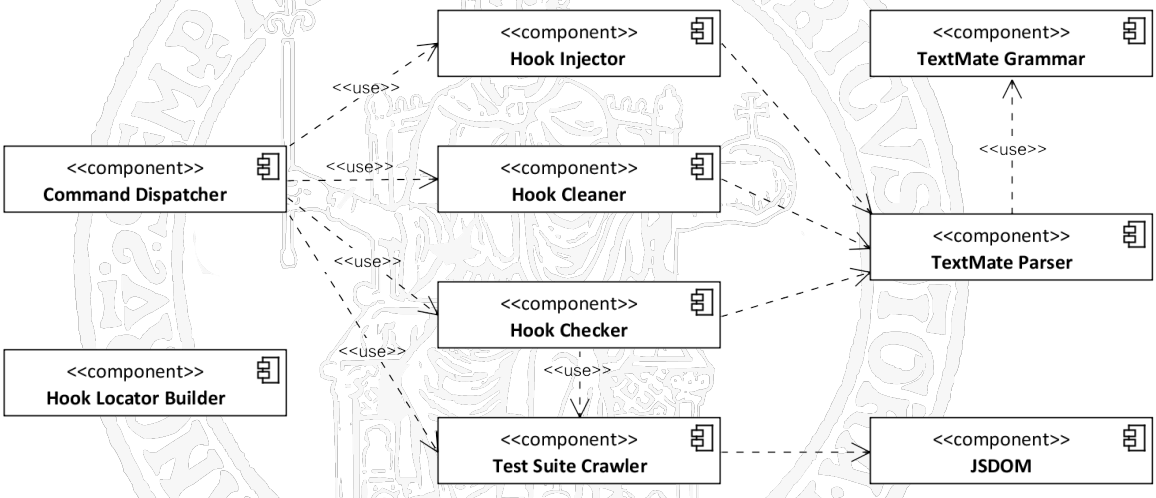


Figura 4.2.1: Component diagram del prototipo

I quattro componenti elencati all'inizio (Hook Injector, Cleaner, Checker e Test Suite Crawler), pur essendo utilizzati in momenti diversi durante il processo di sviluppo e pur avendo scopi diversi, presentano comunque diversi aspetti in comune:

- Ogni componente deve individuare i file di interesse (template o script di test) in maniera ricorsiva a partire da un percorso base (espresso come glob o cartella radice).
- Ogni componente deve leggere e/o scrivere il contenuto dei file di template o di test.
- Ogni componente prevede l'utilizzo di una libreria esterna che effettua parsing dei file (**JSDOM** o **TextMate**) per poter interpretare e/o manipolare il contenuto.

Alla luce di questi punti e ai fini di una corretta **organizzazione e riuso** del codice, le parti comuni sono state estrapolate da ciascun componente in modo da poter essere riutilizzate. Tralasciando momentaneamente la specificità di ciascuno dei quattro componenti principali, in figura 4.2.2 è mostrata questa suddivisione di dettaglio: il **Command Dispatcher**, tramite un **Façade**, attiva il **File Processor** (che si occupa di navigare il file system, leggere e scrivere i file) e quest'ultimo utilizza la particolare **strategia di processing**, specifica per ciascun componente (hook injection, removal, checking, ecc.), che tramite il **Parser** fornito è in grado di interpretare e/o manipolare il contenuto dei file.

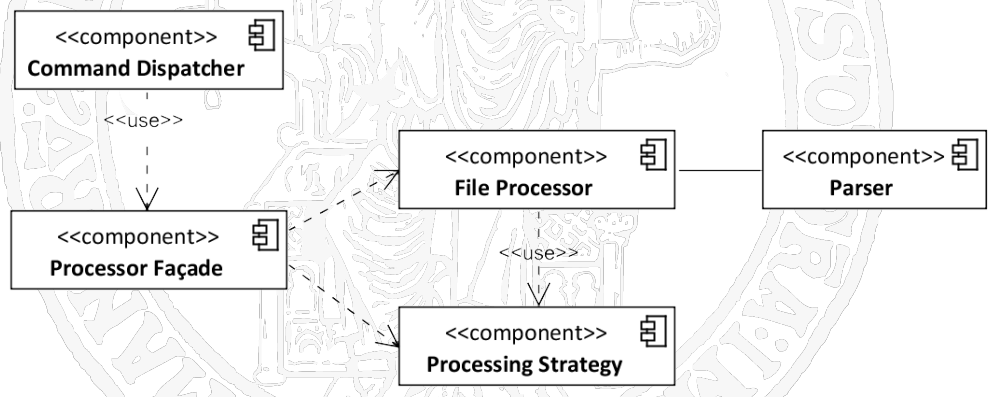


Figura 4.2.2: Riorganizzazione dei componenti

La figura 4.2.3 mostra un sequence diagram dettagliato che descrive le interazioni tra i vari componenti mostrati. Come anticipato, la funzionalità specifica di ciascun macro-componente (Hook Injector, Cleaner, Checker e Test Suite Crawler) è incapsulata nella **Processing Strategy**, mentre tutti gli altri componenti coinvolti sono trasversali. Gli step 7, 8 e 9 del diagramma rappresentano appunto la parte specifica per ciascuna strategia di processing, che **implementa gli algoritmi** riportati nella sezione 3.4.2.

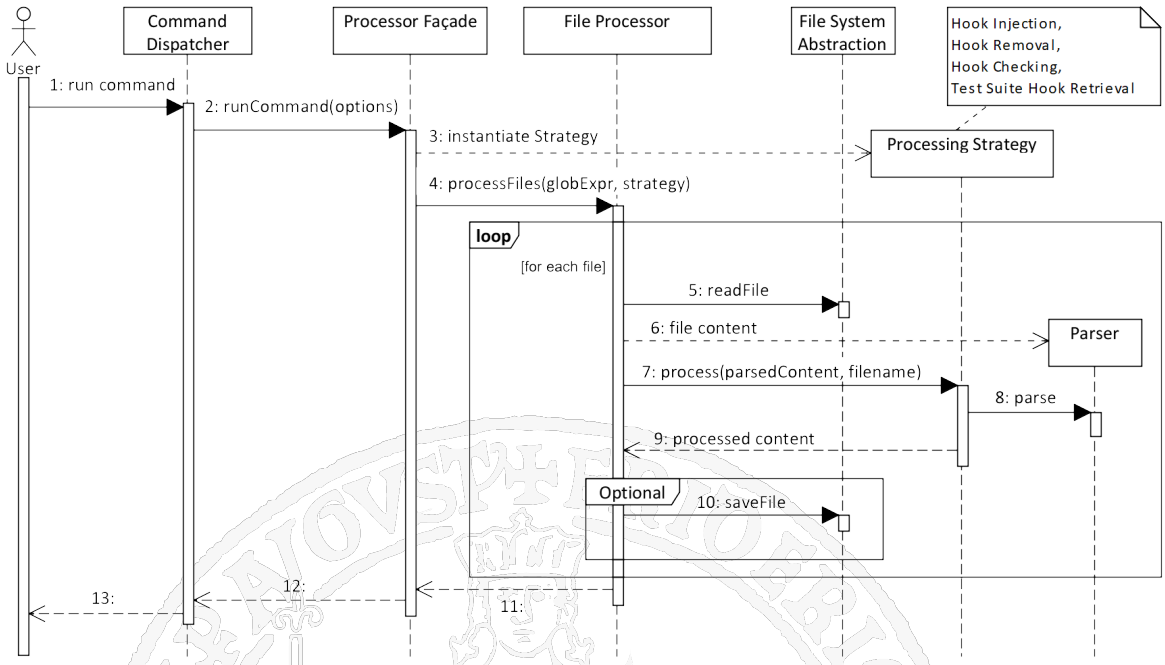


Figura 4.2.3: Interazione tra i componenti

Il funzionamento interno del prototipo, seguendo il sequence diagram, è descritto nei punti seguenti:

1. L'utente invoca il dispatcher passando il nome del comando e le opzioni desiderate.
2. In base al comando impartito al dispatcher, al Processor Façade viene delegata l'azione da compiere, passando contestualmente i parametri necessari per completare l'operazione.

3. A seconda del comando ricevuto, il Processor Façade provvede a selezionare ed istanziare la strategia appropriata per l'elaborazione dei template o dei test (Processing Strategy).

4. Il Processor Façade fornisce al File Processor il path di base dei template o dei test e l'istanza della strategia di processing selezionata.

5. Il File Processor effettua la lettura di un file alla volta.

6. Viene creata un'istanza del parser opportuno (JSDOM per le suite di test, TextMate per i template), passando il contenuto testuale del file caricato.

7. Il File Processor invoca la **strategia di processing**, fornendo l'istanza del parser preparata e il nome del file.

8. Per **interpretare** il contenuto del file ed **eseguire le operazioni** previste nella strategia, si invocano le API specifiche del parser.

9. Il contenuto processato viene restituito al File Processor.

10. Se previsto dal comando impartito, il File Processor effettua la scrittura su filesystem della versione processata del file.

4.3 Descrizione dell'implementazione

Il prototipo è utilizzabile dall'utente come un programma eseguibile a riga di comando. Esso prevede alcuni parametri in input per attivare le diverse modalità di funzionamento (aggiunta, rimozione e verifica degli hook) e i percorsi nei quali si trovano i template e gli script di Selenium in formato HTML.

Per quanto riguarda la registrazione dei test con Selenium, verrà descritta l'implementazione di uno script che estende il meccanismo di costruzione dei locatori mediante un builder aggiuntivo che è in grado di rilevare gli hook.

4.3.1 Command Dispatcher

Il punto d'ingresso del prototipo è stato implementato utilizzando la libreria `yargs` che espone un'API per creare applicazioni CLI (command line interface), definendo l'albero dei parametri, le opzioni, la guida contestuale, la validazione e i vincoli necessari per ciascuno di essi. Avviando il programma senza alcun parametro si ottiene l'output seguente:

```
$ node .\main.js
Usage: main.js <command> [glob] [options]

Commands:
  main.js inject-hooks     Injects test hooks in HTML files
  main.js verify-hooks     Runs consistency checks between templates and test
                           suites
  main.js remove-hooks     Remove unused test hooks from HTML files
  main.js show-used-hooks  Show used test hooks

Options:
  --help     Show help                                                 [boolean]
  --version  Show version number                                       [boolean]

Not enough non-option arguments: got 0, need at least 1
```

È possibile attivare la guida contestuale specificando l'opzione `--help`, come mostrato di seguito per il comando `verify-hooks`:

```
$ node .\main.js verify-hooks --help
main.js verify-hooks

Runs consistency checks between templates and test suites

Options:
  --help     Show help                                                 [boolean]
  --version  Show version number                                       [boolean]
  --grammar  Grammar name
     [string] [required] [choices: "angularjs", "html", "php", "smarty", "twig"]
  --suites   Path of Selenium test suites                   [string] [required]
```

A seconda del comando scelto, dopo aver validato i parametri, il dispatcher effettua un'invocazione ai metodi dei Processor Façade. A seconda della funzionalità richiesta, potrà essere invocato il Façade di processing per i template o quello per le suite di test, passando le opzioni scelte dall'utente. Nel listato 4.3.1 è riportata la sintassi dichiarativa basata sulla *fluent*[1]

[1] Un'interfaccia "fluent" (termine coniato inizialmente da Eric Evans e Martin Fowler) indica un metodo di progettazione per le API object oriented, basato estensivamente sulla tecnica del method-chaining con l'obiettivo di massimizzare la leggibilità del codice sorgente, incorporando a tale scopo un domain-specific language (DSL) nell'interfaccia di programmazione.

API di `yargs` per la definizione dei comandi, delle opzioni e degli handler per ciascuno di essi.

Gli handler passati alla funzione `asyncHandler` delegano il controllo ai vari metodi dei Processor Façade, raccogliendo eventuali errori e restituendone in output la descrizione.

```
yargs
    .detectLocale(false)
    .usage('Usage: $0 <command> [glob] [options]')
    .command({
        command: 'inject-hooks',
        desc: 'Injects test hooks in HTML files',
        builder: optionsBuilder(baseOptions),
        handler: asyncHandler(injectHooksHandler)
    })
    .command({
        command: 'verify-hooks',
        desc: 'Runs consistency checks between templates and test suites',
        builder: optionsBuilder(suitesOptions),
        handler: asyncHandler(verifyHooksHandler)
    })
    .command({
        command: 'remove-hooks',
        desc: 'Remove unused test hooks from HTML files',
        builder: optionsBuilder(suitesOptions),
        handler: asyncHandler(removeHooksHandler)
    })
    .command({
        command: 'show-used-hooks',
        desc: 'Show used test hooks',
        builder: optionsBuilder(suitesOptions),
        handler: asyncHandler(showUsedHooksHandler)
    })
    .demandCommand(1, 1)
    .strict(true)
    .argv;
```

Listato 4.3.1: Template: Yargs fluent API

4.3.2 Processor Façade

Il Command Dispatcher invoca i Processor Façade per richiamare le implementazioni degli algoritmi introdotti nella sezione 3.4.2. Facendo riferimento alla figura 4.3.1, si può notare che sia il Template Processor Façade che il Test Suite Crawler Façade sfruttano il pattern Strategy, che consente di utilizzare in maniera intercambiabile algoritmi diversi, senza dover apportare modifiche all'utilizzatore. In questo caso, le strategie sono le implementazioni dei diversi algoritmi presentati nella sezione 3.4.2; la strategia selezionata viene fornita al Template Processor, il quale può applicarla a ciascun template, a valle del caricamento e

del parsing. Ciascuna strategia è implementata come funzione ricorsiva che prende in input un nodo dell'albero DOM del template; ulteriori parametri variano poi a seconda della responsabilità specifica.

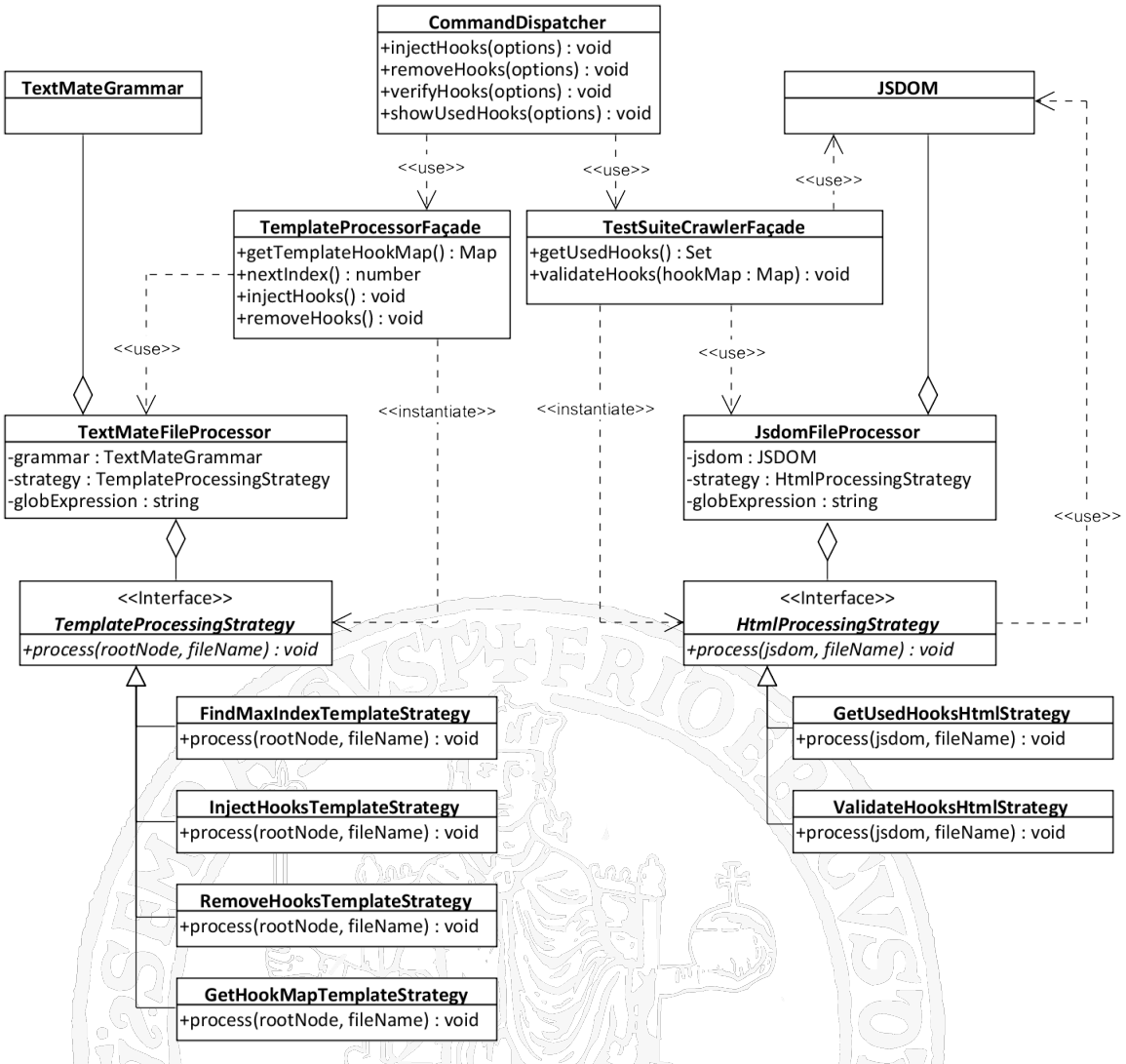


Figura 4.3.1: Diagramma delle classi

La differenza sostanziale tra le due implementazioni risiede nella tipologia di file supportati da ciascuno. L'implementazione Test Suite Crawler Façade, come riporta il diagramma delle classi, ha un livello di accoppiamento elevato con la tecnologia JSDOM, dato che il parsing dei template HTML produce un DOM completo di tutti i metodi per la manipolazione degli

elementi: ciò rende il file processor particolarmente semplice (esso non compie nessun tipo di trasformazione aggiuntiva dopo il parsing effettuato da JSDOM), ma di contro tutte le classi che interagiscono col DOM sono inevitabilmente legate alle API di JSDOM. L'implementazione Template Processor Façade è invece slegata dalla particolare tecnologia di parsing dei template, dato che il file processor che utilizza il motore TextMate si occupa anche della costruzione di un DOM semplificato (incompatibile a livello di API con JSDOM).

Template Processor Façade

Esso effettua il processing di tutti i template indicati dall'utente, utilizzando il componente `TextMateFileProcessor` (descritto nella sezione 4.3.3), il cui compito è l'esplorazione ricorsiva delle cartelle contenenti i template e il parsing di ognuno di essi come albero DOM. Per poter ricostruire il DOM a partire da un template, viene utilizzato un parser generico che è in grado di interpretare qualsiasi formato di template, a patto di fornire il descrittore della grammatica per ciascun linguaggio. La libreria FreeMate per NodeJS implementa tale parser sfruttando grammatiche TextMate in formato CSON (convertite dall'originale formato tmLanguage). Le strategie di processing sono le seguenti (si fa riferimento agli algoritmi presentati nella sezione 3.4.2):

- **FindMaxIndexTemplateStrategy**: implementa l'algoritmo che effettua il calcolo del massimo indice degli hook. È utilizzata prima dell'operazione di injection.
- **InjectHooksTemplateStrategy**: implementa l'algoritmo che inietta gli hook all'interno di ciascun template.
- **RemoveHooksTemplateStrategy**: implementa l'algoritmo di rimozione degli hook inutilizzati. Il Processor Façade fornisce in input l'insieme di hook effettivamente utilizzati negli script di test.
- **GetHooksMapTemplateStrategy**: estrapola da ciascun template tutti gli hook presenti, organizzandoli in una struttura dati indicizzata in base ai template hook. Il risultato di questa strategia di processing è un dizionario le cui chiavi sono i template

hook e i valori sono gli insiemi di test hook per ciascun template, utilizzato durante il controllo di coerenza tra i template e i test.

Test Suite Crawler Façade

L'implementazione del Crawler per i test di Selenium segue anch'essa il funzionamento presentato in figura 4.3.1, sfruttando il parser JSDOM, ma si serve di altre due strategie, una impiegata nella fase di rimozione degli hook inutilizzati, l'altra durante il controllo di coerenza tra i test e i template. Nel listato 4.3.2 è riportato un semplice esempio di script di test in formato HTML che può essere letto dal Selenium Test Crawler.

```
<html>
<head>
   <title>freemarker-demo-1</title>
</head>
<body>
   <table>
      <thead>
         <tr><td colspan="3">Test Case Freemarker</td></tr>
      </thead>
      <tbody>
         <tr>
            <td>open</td>
            <td>http://localhost:8080/freemarker-sample/index</td>
            <td></td>
         </tr>
         <tr>
            <td>click</td>
            <td>//*[@x-test-tpl-1]//*[@x-test-tpl-7]//*[@x-test-hook-15]</td>
            <td></td>
         </tr>
         <tr>
            <td>type</td>
            <td>//*[@x-test-tpl-1]//*[@x-test-tpl-7]//*[@x-test-tpl-16]//*[@x-test-hook-20]</td>
            <td>Test activity</td>
         </tr>
      </tbody>
   </table>
</body>
</html>
```

Listato 4.3.2: Test Case in formato HTML

La strategia **ValidateHooksHtmlStrategy** che si occupa del **controllo di coerenza** tra template e test sfrutta JSDOM per leggere il test della cella contenente l'espressione XPath del locatore basato sugli hook. Per fare ciò, effettua i passaggi seguenti:

- Lo script di test è rappresentato in formato HTML, contenente una tabella in cui vi sono gli step del test. Ciascun file HTML viene esplorato mediante l'espressione XPath seguente al fine di estrarre i locatori dalle righe della tabella:

```
/html/body//tbody/tr/td[2]/text()
```

- Mediante un'espressione regolare, da ciascun locatore viene ricavato un insieme ordinato di hook, ad esempio:

```
IN: '//*[@x-test-tpl-1]//*[@x-test-tpl-7]//*[@x-test-hook-15]'
OUT: ['x-test-tpl-1', 'x-test-tpl-7', 'x-test-hook-15']
```

- Il Command Dispatcher fornisce in input il dizionario ottenuto dall'invocazione di **GetHooksMapTemplateStrategy**. Utilizzando la corrispondenza ricavata dai template tra template hook (radice) e gli hook di test, si verifica per ciascun elemento dell'insieme:
  - Se l'elemento è un template hook, esso deve esistere nel dominio della corrispondenza (come chiave del dizionario).
  - Se l'elemento è un hook di test, deve esistere una corrispondenza tra il template hook trovato in precedenza e l'elemento corrente.
- Viene sollevato un errore nel caso in cui fallisca il controllo dello step precedente.

Per completezza, nel listato 4.3.3 è riportata l'implementazione della strategia che effettua il controllo. Il primo parametro in input è un locatore (sotto forma di insieme ordinato di hook) e il secondo parametro è la corrispondenza tra template hook e gli hook di test (sotto forma di dizionario).

```
function validateHookSetStrategy(hookSet, templateHookMap) {
    let templateHook;
    for (let hook of hookSet) {

        // templateHook may contain previous value
        templateHook = templateHookMap.get(hook) || templateHook;

        if (!templateHook) {
            throw new Error("Template hook not found: " + hook);
```

```
10          }
11
12          //not currently on template hook, check if child test hook is present
13          if (templateHookMap.get(hook) !== templateHook && !templateHook.hooks.has(hook)) {
14              throw new Error("Test hook " + hook + " not found for template: " +
15                  templateHook.fileName + "\nHook path:\n" + hookSet.join(' ->\n'));
16          }
17      }
18  }
```

Listato 4.3.3: Controllo di coerenza tra test e template

4.3.3 TextMate Template File Processor

L'implementazione del File Processor basato su JSDOM (**JsdomFileProcessor**) non effettua alcun tipo di elaborazione a valle del parsing e prima di invocare la strategia di processing, dato che le API del DOM offrono già tutti i metodi di manipolazione e trasformazione necessari. L'implementazione che sfrutta il parser TextMate (**TextMateFileProcessor**) effettua invece alcune elaborazioni preliminari prima di passare il controllo alla strategia di processing, al fine di ricostruire la struttura ad albero della vista, ignorando le specificità del particolare linguaggio di templating. Il listato 4.3.4 riporta il punto di ingresso del Template File Processor basato su TextMate, di seguito sono elencati i punti salienti:

- Alla riga 2 l'espressione glob[2] passata in input viene interpretata e tradotta in una lista di file da processare.
- Alla riga 7 ciascun file viene letto come stringa UTF-8.
- Alla riga 8 si utilizza l'oggetto corrispondente alla grammatica selezionata per effettuare il parsing del file. L'output di questo passaggio è una lista di righe, ciascuna contenente i token individuati secondo la grammatica.
- Le righe 9 e 11 effettuano un raggruppamento dei token e la costruzione della struttura ad albero del template.

[2]Un glob pattern è una sintassi attraverso la quale si rappresenta un insieme di stringhe. È la sintassi tradizionalmente usata nelle shell testuali dei sistemi Unix e Unix-like per effettuare l'espansione di nomi di file e directory ed essa riprende in piccola parte quella delle espressioni regolari.

- La riga 12 invoca la strategia di processing passata in input.
- Dalla riga 13 in poi si effettua la scrittura del risultato del processing, se previsto.

```
1  async function processFiles(globExpr, grammar, processor, writeOutput = false) {
2      const files = await glob(globExpr);
3      let file, fileName, content, lines, root, tree, stream = process.stdout;
4
5      for (file of files) {
6          fileName = path.resolve(file);
7          content = await readFile(fileName, 'utf-8');
8          lines = grammar.tokenizeLines(content);
9          root = groupTokens(iterableTokens(lines), [ TAG_SCOPE, TAG_NAME_SCOPE,
10             ATTRIBUTE_SCOPE, EMPTY_ATTRIBUTE_SCOPE]);
11         tree = buildTagTree(root);
12         processor(tree, fileName);
13         if (writeOutput) {
14             stream = fs.createWriteStream(fileName);
15             stream.write(tree.toString());
16             stream.end();
17         }
18     }
19 }
```

Listato 4.3.4: TextMate Template File Processor Entry Point

La funzione `groupTokens` alla riga 9, effettua un raggruppamento preliminare dei token in base ad una lista di oggetti *scope* contenenti le direttive per riconoscere l'inizio e la fine di ciascun elemento sintattico di interesse.

L'organizzazione in gerarchia dei gruppi di token in questa fase è fondamentale per individuare successivamente i tag e gli attributi della sintassi HTML. In figura 4.3.2 è riportato un esempio di raggruppamento dei token in base agli scope; il token di inizio e fine di ciascun gruppo è contrassegnato da una freccia, i rettangoli in rosso racchiudono i token che formano i tag HTML di apertura o di chiusura, quelli in blu racchiudono i token che formano gli attributi e i loro valori. È in questa fase che tutti gli altri token, specifici della particolare tecnologia di templating, vengono ignorati.

Il raggruppamento ottenuto non è però sufficiente per poter processare il template come albero DOM, dato che la gerarchia costruita non riguarda ancora gli elementi HTML ma soltanto i token che il parser ha inizialmente prodotto. Per innalzare ulteriormente il livello di astrazione, alla riga 11 del listato 4.3.4, la funzione `buildTagTree` si occupa appunto di costruire la gerarchia degli elementi del DOM, partendo dai gruppi di token prodotti in precedenza.

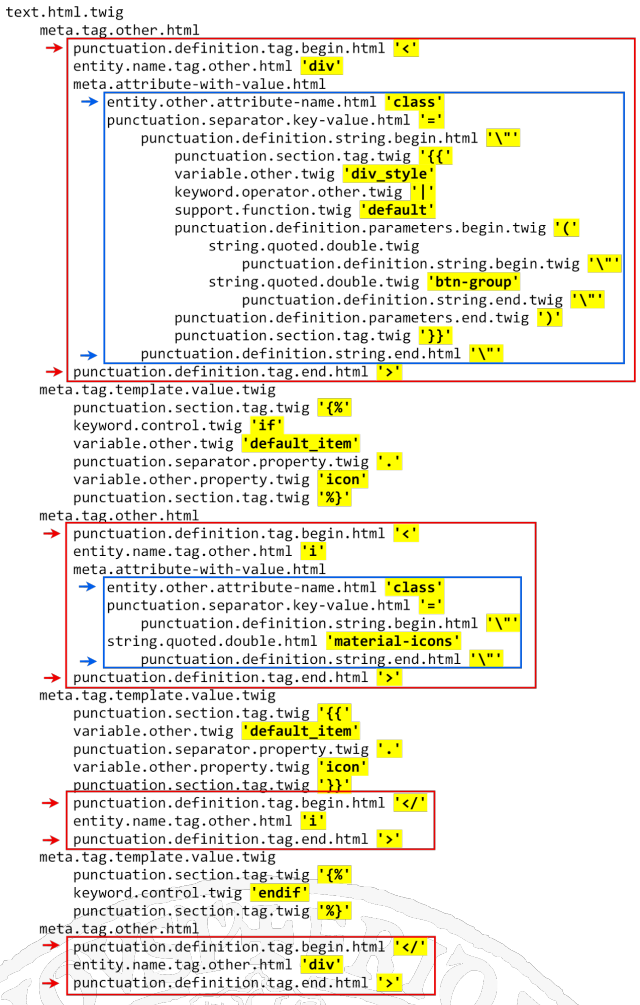


Figura 4.3.2: Raggruppamento token

L'algoritmo implementato è anche robusto rispetto ad eventuali "errori" o piccole violazioni delle regole di chiusura dei tag (che in HTML possono essere tollerate), senza però impattare sull'output prodotto in seguito a partire dal template processato. Questa è un'altra differenza che esiste col parser JSDOM, il quale effettua automaticamente modifiche correttive al DOM nei punti in cui vi erano alcune violazioni delle regole dei tag: purtroppo questo comportamento non sempre è desiderabile, e comunque non è compito di questo strumento apportare modifiche alla struttura del template. In figura 4.3.3 è riportato il risultato della costruzione dell'albero DOM a partire dai gruppi di token; i rettangoli in verde delimitano i nodi dell'albero individuati, in grigio è riportato il contenuto ignorato dal processor.

Figura 4.3.3: Albero DOM ricavato dal template

4.3.4 Hook Locator Builder

In questa sezione è riportata la descrizione del componente (in figura 4.2.1) che si occupa di creare i locatori basati sugli hook durante la fase di test capture (punto 3 nella figura 3.3.1). Lo strumento di capture/replay Katalon Recorder, basato su Selenium, prevede la possibilità di personalizzare il funzionamento interno mediante script di estensione che possono essere caricati direttamente dall'interfaccia grafica. Gli script di estensione hanno accesso ad alcuni oggetti e funzioni del recorder, in particolar modo delle funzioni che si occupano di creare i locatori durante la fase di capture, per ogni evento che l'utente produce sulla schermata. Queste funzioni, chiamate Locator Builders, ricevono in input l'elemento di interesse per l'evento o asserzione corrente e restituiscono una stringa che rappresenta il locatore. Il funzionamento è riassunto nei punti seguenti:

- Ciascun Locator Builder viene caricato ed attivato, assegnandogli un nome e una priorità.
- Durante la fase di capture, per ogni elemento di interesse vengono invocati tutti i Locator Builder, ottenendo un locatore da ciascuno.
- Tutti i locatori vengono immediatamente valutati, stabilendo così se riescono realmente a localizzare l'elemento di interesse in maniera univoca (non ambigua).
- Tra tutti i locatori che superano il controllo del punto precedente, viene selezionato il primo locatore (seguendo la priorità dei builder che li hanno prodotti) che è in grado di trovare l'elemento, inserendolo nel nuovo step registrato.

Per caricare ed attivare un Locator Builder aggiuntivo, il recorder mette a disposizione come variabile globale l'oggetto `LocatorBuilders` sul quale effettuare l'aggiunta mediante il metodo `add()`; il metodo `setPreferredOrder()` consente invece di modificare l'ordine di priorità dei Locator Builder attivi:

```
function myLocatorBuilder(elem) {
    // create locator here...
    return locatorString;
}

LocatorBuilders.add('myLocBuilder', myLocatorBuilder);
LocatorBuilders.setPreferredOrder(['myLocBuilder']); // myLocBuilder will be the first one.
```

Nel listato 4.3.5 è riportato il codice del Locator Builder che implementa l'algoritmo mostrato nella sezione 3.4.2.

```
// Hook-based Locator Builder
function hookBasedLocatorBuilder(elem) {
    let pathLocator = null;
    let includesChildTemplate = false;

    while (elem !== document) { // climb up to the document root

        let hookName = getHookFromAttributes(elem);
        if (hookName) {

            const isTemplateRoot = hookName.indexOf(templatePrefix) >= 0;

            const siblings = getAllSiblings(elem, e => e.hasAttribute(hookName));

            const index = siblings.length > 1 ? siblings.indexOf(elem) + 1 : 0;

            if (!pathLocator || siblings.length > 1 || isTemplateRoot || includesChildTemplate) {
                pathLocator = '//*[@' + hookName + ']' + // attribute-based XPath
                          (index > 0 ? '[' + index + ']' : '') + // use index when needed
                          (pathLocator || ''); // prepend new hook

                includesChildTemplate = !!isTemplateRoot;
            }
        }
        elem = elem.parentNode;
    }
    return pathLocator;
}

LocatorBuilders.add('hookBasedLocatorBuilder', hookBasedLocatorBuilder);
LocatorBuilders.setPreferredOrder(['hookBasedLocatorBuilder']);

// helper function: get all sibilings by predicate
function getAllSiblings(elem, predicate) {
    var siblings = [];
    elem = elem.parentNode.firstChild;
    do {
        if (elem.nodeType === 1 && predicate(elem)) {
            siblings.push(elem);
        }
```

```
41      } while ((elem = elem.nextSibling))
42      return siblings;
43  }
44
45  // helper function: extract hook value from DOM element
46  function getHookFromAttributes(elem) {
47      for (var i = 0, l = elem.attributes.length; i < l; ++i) {
48          let match = elem.attributes[i].name.match(hookRegex);
49          if (match) {
50              return match[0];
51          }
52      }
53  }
```

Listato 4.3.5: Locator builder Hook-based

Capitolo 5

Esempi

Questo capitolo riporta alcuni esempi di utilizzo degli strumenti implementati, illustrando le varie fasi del processo di sviluppo su tecnologie di templating diverse tra loro, con riferimento al paradigma architetturale supportato in ciascun caso.

5.1 Installazione del prototipo

Per avviare ed utilizzare il prototipo sono necessari alcuni passaggi preliminari di preparazione dell'ambiente, elencati di seguito[1]:

- Installare NodeJS e NPM utilizzando l'installer disponibile sul sito ufficiale[2], o in alternativa seguire le istruzioni per l'installazione con il package manager del sistema operativo in uso, sempre dal sito ufficiale. La versione di NodeJS richiesta è la 8 o successive.
- Copiare i file del prototipo in una cartella, ad esempio in `~/test-hooks-prototype`.
- Posizionarsi nella cartella del prototipo e lanciare il comando `npm install` per scaricare tutte le dipendenze necessarie (una descrizione di NPM è riportata in A.2).
- Installare l'applicazione da testare in una cartella, ad esempio in `~/test-webapp`; in `~/test-webapp/templates` saranno presenti i template (sempre a titolo di esempio).

[1] Le convenzioni utilizzate negli esempi fanno riferimento a sistemi Linux. Per Windows o altri sistemi, possono essere necessari alcuni adattamenti.

[2] https://nodejs.org/en/download/

- Posizionarsi nella cartella del prototipo e avviare il file `main.js` per verificare il corretto funzionamento, utilizzando il path dei template specificato in precedenza (si fa riferimento al template engine Twig per questo passaggio):

```
$ node main.js inject-hooks '../test-webapp/templates/**/*.twig' --grammar twig
```

- Verificare che i sorgenti dei template siano stati modificati con gli hook di test.

5.2 Esempio su architettura MVVM: AngularJS

Per verificare il funzionamento su un'architettura MVVM, è stato scelto il framework strutturale AngularJS che include un template engine basato su HTML puro.

5.2.1 Preparazione ambiente

Nell'esempio riportato in questa sezione verrà utilizzata l'applicazione "demo" fornita col prototipo e basata su AngularJS. Tale applicazione sarà installata in `~/angular-demo`. Per avviare l'applicazione è necessario eseguire i seguenti passaggi:

- Installare il server http mediante il comando `sudo npm i -g http-server`. (si ipotizza che NodeJS e NPM siano stati già installati come riportato nella sezione 5.1).
- Posizionarsi nella cartella dell'applicazione `~/angular-demo` e lanciare il comando `http-server`. A questo punto, è possibile visualizzare l'applicazione dal browser all'URL `http://localhost:8080`. Nel caso in cui la porta di default 8080 non sia disponibile, è possibile utilizzare il flag `--help` per una guida completa dei parametri di `http-server`, tra cui l'impostazione della porta desiderata.

5.2.2 Fase 1: Hook Injection

Dopo aver installato il prototipo come descritto, è possibile avviare la fase di injection degli hook di test lanciando il comando seguente dalla cartella del prototipo:

```
$ node main.js inject-hooks '../angular-demo/**/*.html' --grammar angularjs
```

Se l'operazione è andata a buon fine il programma non restituisce alcun output. È possibile verificare la presenza degli hook, ad esempio nel file `~/angular-demo/index.html`:

```
<html lang="en" dir="ltr" x-test-tpl-1>
<head x-test-hook-2>
    <meta charset="utf-8" x-test-hook-3>
    <title x-test-hook-4>Sample Project</title>
    <link rel="stylesheet" href="css/bootstrap.min.css" x-test-hook-5>
    <link rel="stylesheet" href="css/custom.css" x-test-hook-6>
</head>

<body ng-app="myApp" x-test-hook-7>

    <div class="container" x-test-hook-8>
        <ui-view x-test-hook-9></ui-view>
    </div>

    <script src="js/lib/angular.min.js" charset="utf-8" x-test-hook-10></script>
    <script src="js/lib/angular-ui-router.min.js" charset="utf-8" x-test-hook-11></script>
    <script src="js/config.js" charset="utf-8" x-test-hook-12></script>
    <script src="js/services.js" charset="utf-8" x-test-hook-13></script>
    <script src="js/controllers.js" charset="utf-8" x-test-hook-14></script>
    <script src="js/directives.js" charset="utf-8" x-test-hook-15></script>

</body>
</html>
```

Listato 5.2.1: Hook injection su template AngularJS

L'impatto visivo sull'applicazione è ovviamente nullo, trattandosi di attributi che non producono cambiamenti allo stile della pagina.

5.2.3 Fase 2: Test Capture

Si effettua la registrazione del test utilizzando i locatori standard di Selenium (in figura 5.2.1). Per utilizzare questa modalità è sufficiente avviare il recorder con le impostazioni di default. La versione utilizzata è la 3.7.0.

Per effettuare la registrazione utilizzando gli hook è possibile caricare dal recorder (in figura 5.2.2) lo script di estensione `attributeHooksLocators.js` fornito col prototipo (nella sottocartella `selenium`).

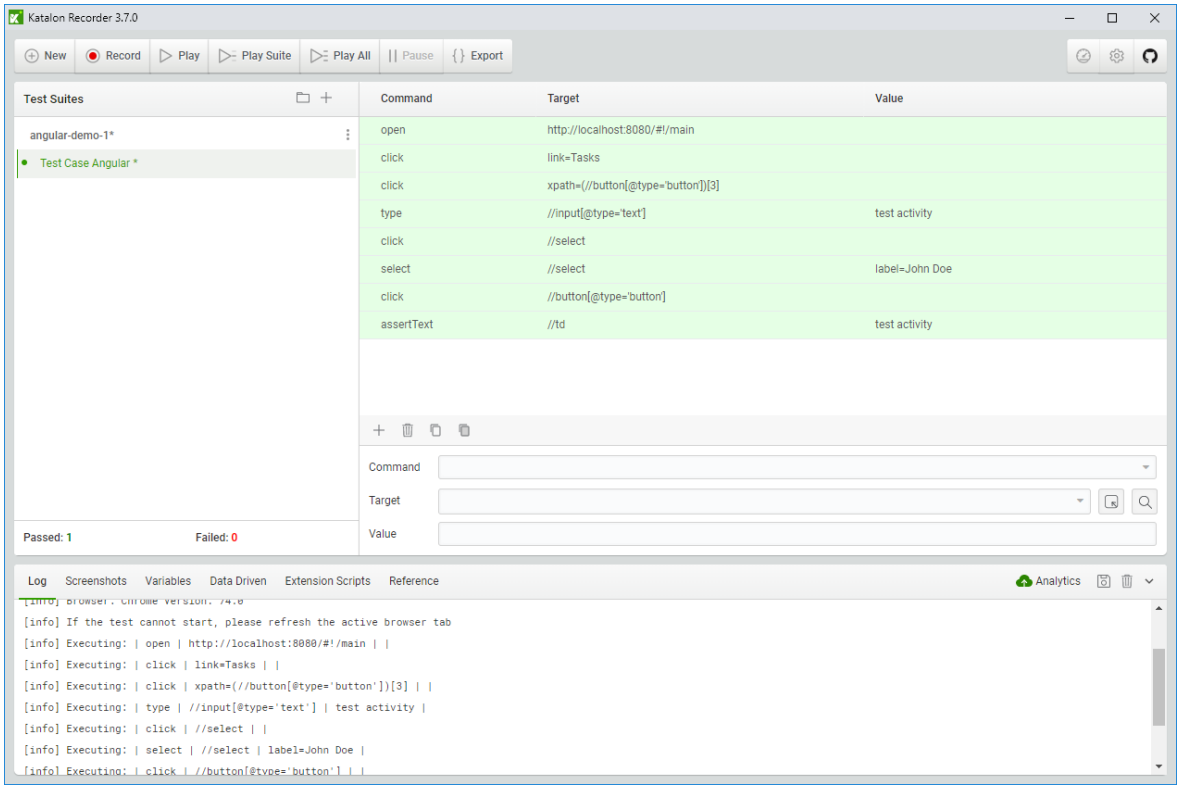


Figura 5.2.1: Registrazione caso di test con Selenium su template AngularJS

A questo punto si effettua la registrazione dello stesso test (ovvero, che utilizza gli stessi elementi grafici dell'applicazione), che però conterrà i locatori hook-based. Il risultato della registrazione del test è riportato in figura 5.2.3.

La suite di test ottenuta dalla seconda registrazione dovrà essere salvata come file HTML, ad esempio nella cartella `~/test-suites`.

5.2.4 Fase 3: Rimozione hook inutilizzati

È possibile effettuare la rimozione di tutti gli hook inutilizzati lanciando il comando seguente dalla cartella del prototipo:

```
1 $ node main.js remove-hooks '../angular-demo/**/*.html' --grammar angularjs --suites ../test-suites
```

Confrontando il template `taskEditState.html` dell'applicazione AngularJS prima e dopo la rimozione degli hook, si ottiene il risultato riportato in figura 5.2.4. Sono stati cerchiati gli hook che sono coinvolti nella suite di test salvata precedentemente nella cartel-

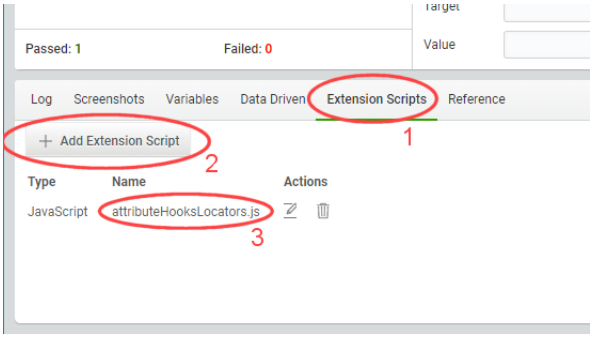


Figura 5.2.2: Caricamento script di estensione per Selenium

la `~/test-suites`, mentre tutti gli altri evidenziati in arancione, sono stati rimossi perché inutilizzati.

Per verificare che l'operazione di rimozione sia stata correttamente eseguita, si riesegue il test ottenendo di nuovo lo stesso risultato riportato in figura 5.2.3.

5.2.5 Fase 4: Modifica della vista

Si supponga di dover effettuare a questo punto una modifica che ha impatto sulla funzionalità di gestione dei task. In particolare, si consideri la rimozione della vista dedicata alla modifica dei task, dando invece la possibilità di effettuare l'aggiunta e la modifica di elementi della tabella con la modalità *inline*, senza dover ricorrere ad una schermata separata. A seguito dell'operazione di modifica del template, si mantengono tutti gli hook contenuti nel template `taskEditState.html`, ricollocandoli opportunamente dove necessario. In figura 5.2.5 è riportato un confronto tra la versione precedente e la successiva: gli elementi contenenti gli hook contrassegnati in verde non sono stati impattati da alcuna modifica, mentre quello contrassegnato in rosso è stato ricollocato sul tag corrispondente nella nuova versione.

5.2.6 Fase 5: Controllo di coerenza tra test e template

Risulta evidente (in questo caso banale, già prima di un controllo automatico) che gli hook relativi al template di edit dei task non sono stati riportati: ciò è dovuto al fatto che è stata effettuata una semplice operazione di spostamento di tag, senza controllare gli altri template.

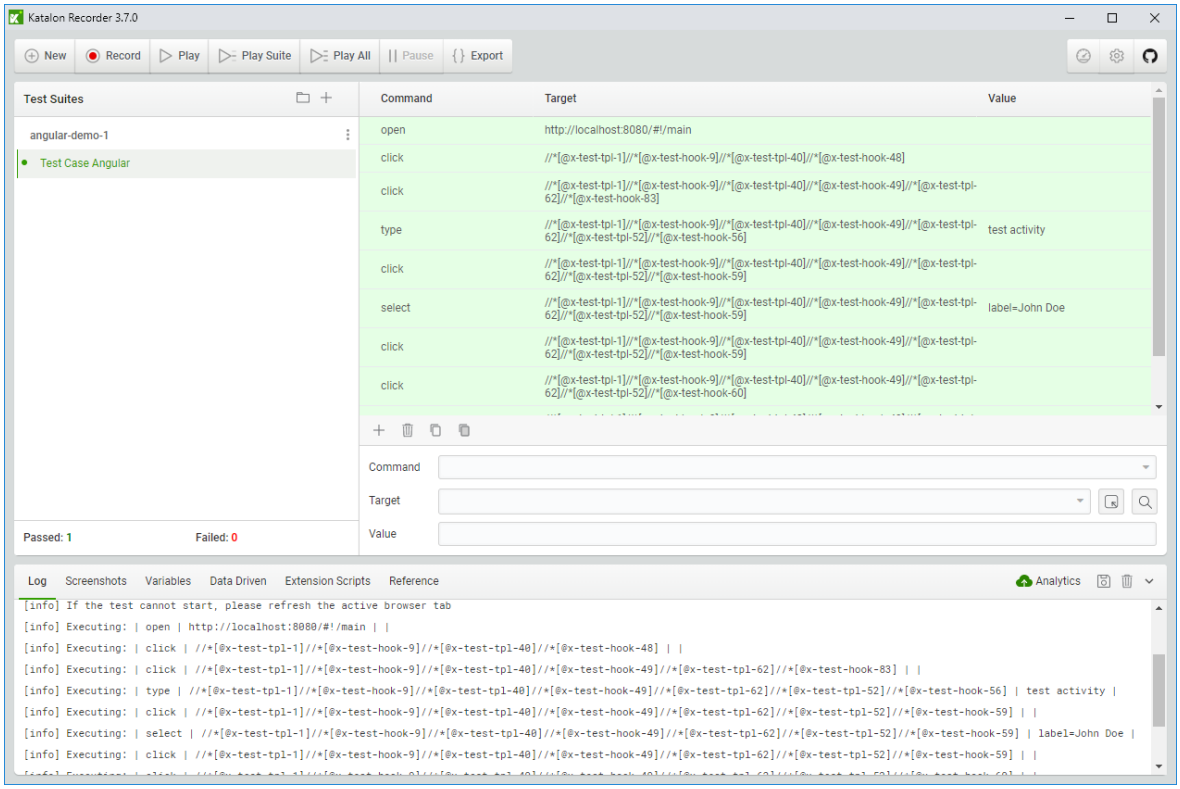


Figura 5.2.3: Registrazione test con Selenium su template AngularJS (hook-based)

Per rilevare tale anomalia è sufficiente lanciare il prototipo per la verifica di consistenza degli hook mediante il comando seguente:

```
$ node main.js verify-hooks '../angular-demo/**/*.html' --grammar angularjs --suites ../test-suites
```

L'output ottenuto è il seguente:

```
Error while processing ~/test-suites/angular-demo-1.html
Error: Test hook x-test-tpl-52 not found for template: ~/angular-demo/tasksState.html
Hook path:
    x-test-tpl-1 ->
    x-test-hook-9 ->
    x-test-tpl-40 ->
    x-test-hook-49 ->
    x-test-tpl-62 ->
    x-test-tpl-52 ->
    x-test-hook-56
```

Questo risultato indica che, partendo dal template `tasksState.html`, alcuni hook contenuti nella suite di test `angular-demo-1.html` non sono più raggiungibili; in particolare, il primo è `x-test-tpl-52` che era presente nella versione precedente nel template `taskEditState.html`, ora rimosso.

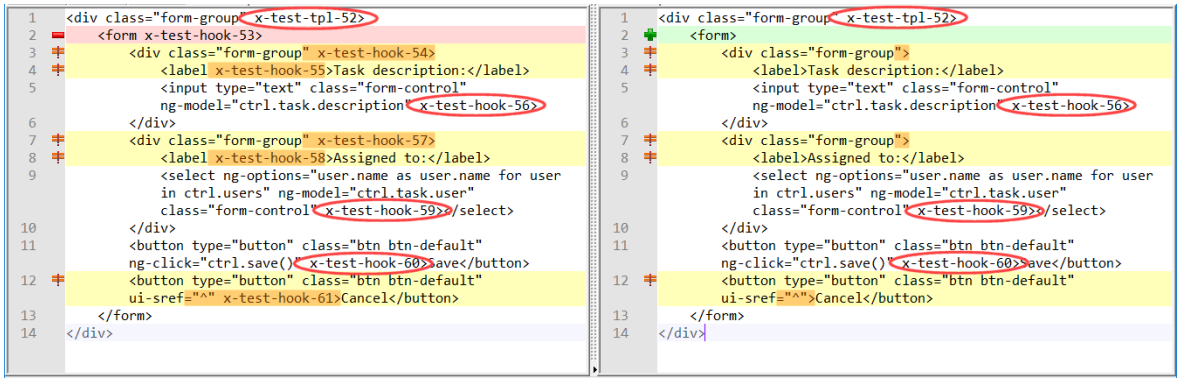


Figura 5.2.4: Hook rimossi: differenze su template AngularJS

A prescindere dalla condizione di errore, l'aspetto più importante riguarda il fatto che è stato possibile rilevare la condizione di incoerenza **senza dover lanciare i test**, con conseguente risparmio di tempo (ciascun blocco di Selenium dura diversi secondi a causa dei timeout). In un'applicazione con centinaia di viste (e relativi test), tale risparmio in termini temporali può essere determinante. In questo caso, lanciando comunque il test per avere conferma del problema, si ottiene il risultato riportato in figura 5.2.6, come è stato già previsto dall'operazione di controllo effettuata precedentemente.

5.2.7 Fase 6: Test repair manuale

Per rendere di nuovo i test utilizzabili è necessario effettuare l'operazione di repair, riportando gli hook mancanti sulla nuova vista. Il primo passaggio del repair consiste nel riportare gli hook dal template `taskEditState.html` della versione precedente al template `tasksState.html` della nuova versione, come riportato in figura 5.2.7.

Oltre al repair del template è necessario sostituire anche l'hook radice all'interno della suite di test, dato che il template `taskEditState.html` non esiste più, e i relativi hook sono stati spostati in un altro template già esistente. Quest'operazione si effettua con una semplice rimozione massiva della stringa `//*[@x-test-tpl-52]` dalla suite di test.

Rilanciando a questo punto il comando di verifica degli hook, i template risultano nuovamente coerenti coi test; ciò è confermato anche dall'esecuzione del test che è di nuovo funzionante.

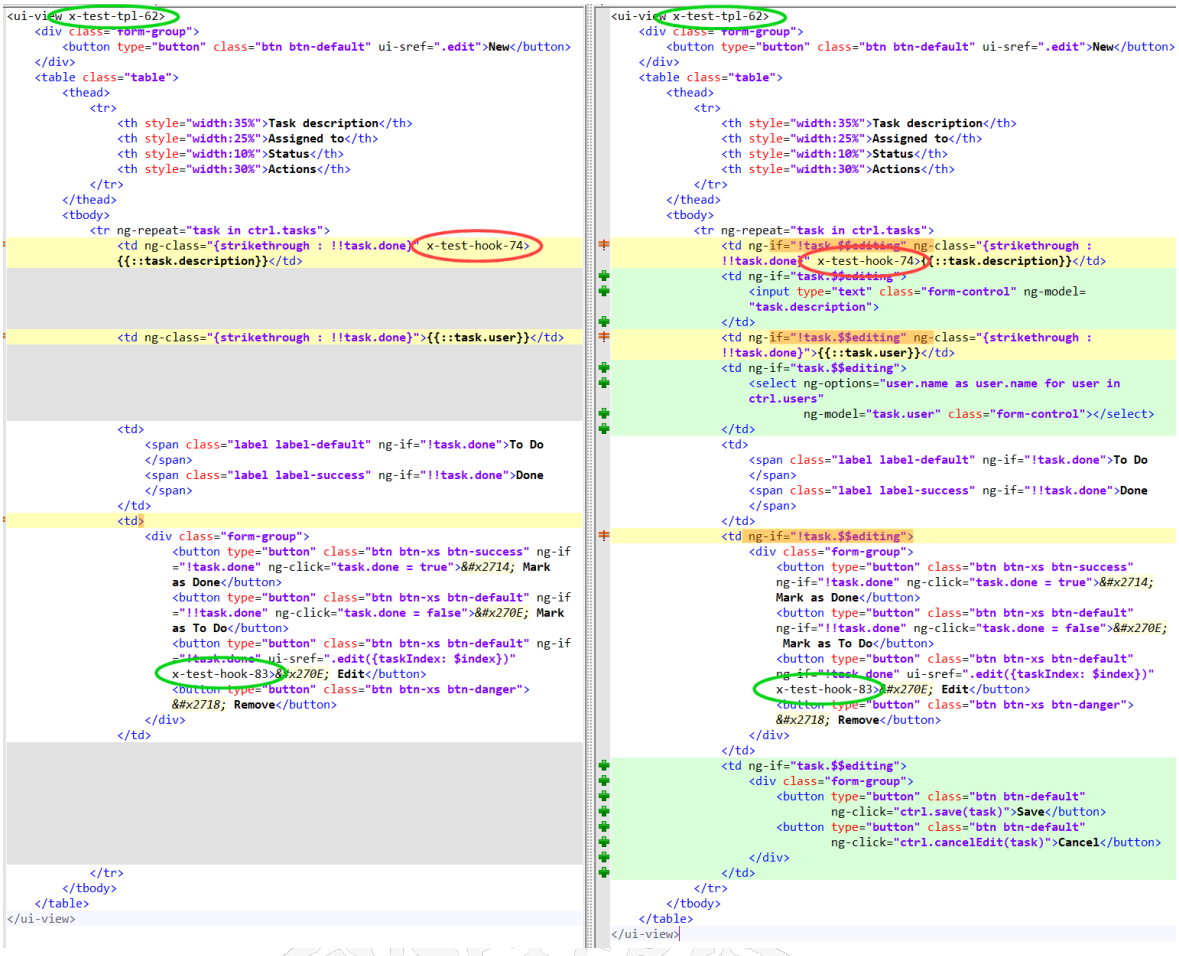


Figura 5.2.5: Modifica del template AngularJS

5.3 Esempio su architettura MVC: Spring & FreeMarker

La tecnologia scelta per l'applicazione del processo su un'architettura Web MVC è Spring Framework in abbinamento col template engine FreeMarker.

5.3.1 Preparazione ambiente

Di seguito sono elencati i passaggi preliminari per poter avviare il progetto di esempio basato su Spring MVC & FreeMarker:

- Installare il prototipo come descritto nella sezione 5.1.

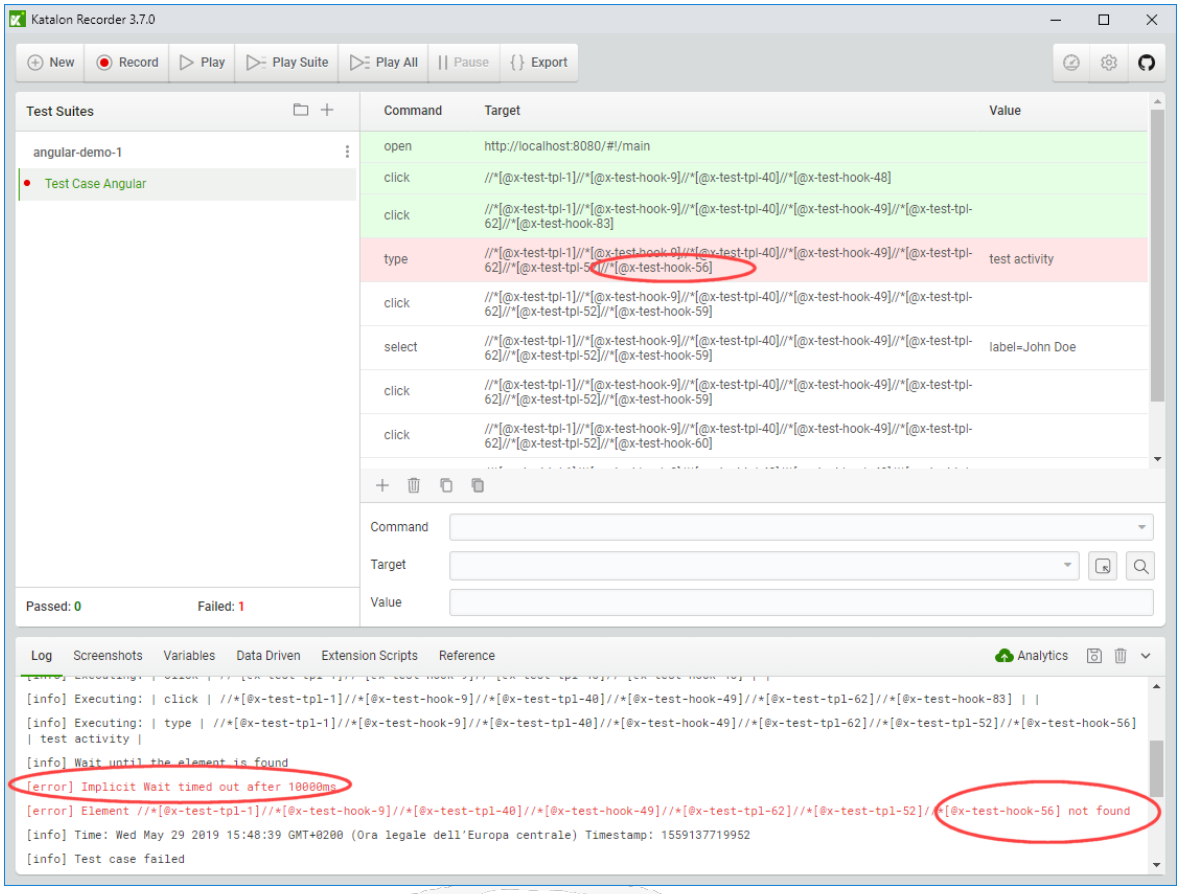


Figura 5.2.6: Test non eseguibile su applicazione AngularJS

- Installare Java Development Kit alla versione 8 o successive, utilizzando l'installer disponibile sul sito ufficiale[3], o in alternativa seguire le istruzioni per l'installazione con il package manager del sistema operativo in uso.
- Installare il Servlet container Tomcat[4] alla versione 7 o successive.
- Installare il sistema di gestione progetti Maven, disponibile in download dal sito ufficiale[5]. Questo strumento serve per eseguire la build del pacchetto WAR da avviare su Tomcat.
- Copiare i file dell'applicazione web in una cartella, ad esempio in `~/test-webapp`; in `~/test-webapp/templates` saranno presenti i template (sempre a titolo di esempio).

[3]https://www.oracle.com/technetwork/java/javase/downloads/index.html
[4]https://tomcat.apache.org/download-80.cgi
[5]https://maven.apache.org/install.html

Figura 5.2.7: Ripristino degli hook su template AngularJS

- Posizionarsi nella cartella del prototipo e avviare il file `main.js` per verificare il corretto funzionamento, utilizzando il path dei template specificato in precedenza:

```
$ node main.js inject-hooks '../test-webapp/src/main/webapp/**/*.ftl' --grammar freemarker
```

- Verificare che i sorgenti dei template siano stati modificati con gli hook di test.
- Posizionarsi nella cartella `~/test-webapp` e lanciare il comando `mvn clean install` che provvede a scaricare tutte le dipendenze necessarie e alla build del pacchetto WAR. L'esito di questo comando dovrebbe produrre un messaggio di "Build Success".
- Avviare il server Tomcat e copiare il pacchetto WAR prodotto dal comando precedente contenuto nella cartella `~/test-hooks-prototype/target` alla cartella `webapps` di

Tomcat. A questo punto, è possibile visualizzare l'applicazione dal browser all'URL `http://localhost:8080`.

5.3.2 Fase 1: Hook Injection

Lanciare il comando seguente per eseguire la hook injection sui template FreeMarker:

```
$ node main.js inject-hooks '../freemarker-demo/src/main/webapp/**/*.ftl' --grammar freemarker
```

Se l'operazione è andata a buon fine il programma non restituisce alcun output. È possibile verificare la presenza degli hook, ad esempio nel file `index.ftl` contenuto nella webapp:

```
<html lang="en" dir="ltr" x-test-tpl-1>
    <head>
        <meta charset="utf-8" x-test-tpl-3>
        <title x-test-tpl-4>Sample Project</title>
        <link rel="stylesheet" href="css/bootstrap.min.css" x-test-tpl-5>
        <link rel="stylesheet" href="css/custom.css" x-test-tpl-6>
    </head>

    <body>

        <div class="container" x-test-tpl-7>

            <div class="form-group" x-test-hook-8>
                <ul class="nav nav-tabs" x-test-hook-9>
                    <li class="${((model.route!'') == 'home')?then('active','') }" x-test-hook-10>
                        <a href="index" x-test-hook-11>Home</a>
                    </li>
                    <li x-test-hook-12><a href="" x-test-hook-13>Users</a></li>
                    <li class="${((model.route!'') == 'tasks')?then('active','') }" x-test-hook-14>
                        <a href="tasks" x-test-hook-15>Tasks</a>
                    </li>
                </ul>
            </div>
            <#include "route.ftl">
        </div>

    </body>
</html>
```

Listato 5.3.1: Hook injection su template Freemarker

Lanciare il comando `mvn clean install` per effettuare una build dell'applicazione; avviare il server Tomcat e copiare il pacchetto WAR appena generato nella cartella `webapps` di Tomcat.

5.3.3 Fase 2: Test capture

Si effettua la registrazione del test utilizzando i locatori standard di Selenium (in figura 5.3.1), avviando il recorder con le impostazioni di default.

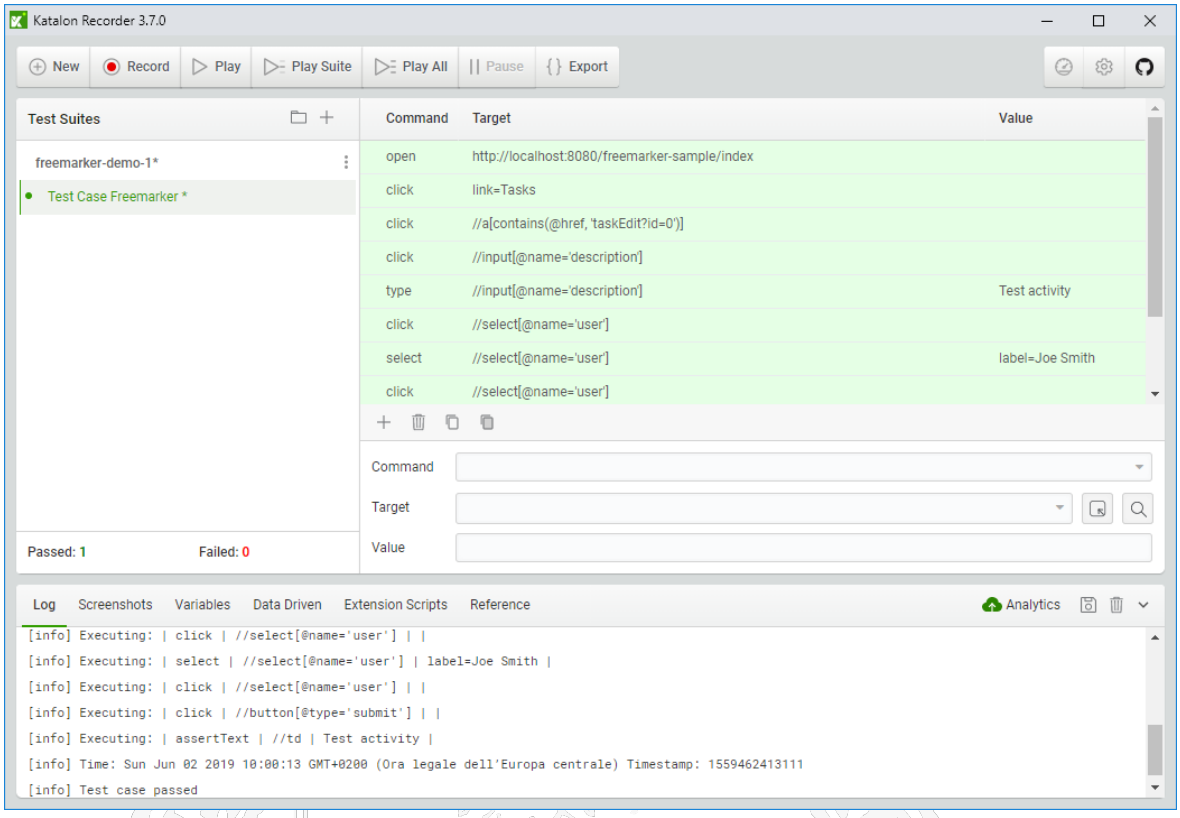


Figura 5.3.1: Registrazione caso di test con Selenium

Effettuare la registrazione caricando dal recorder lo script di estensione per Selenium, come mostrato nella sezione 5.2.3). Registrare a questo punto lo stesso test (ovvero, che utilizza gli stessi elementi grafici dell'applicazione), che però conterrà i locatori hook-based. Il risultato della registrazione del test è riportato in figura 5.3.2.

La suite di test ottenuta dalla seconda registrazione dovrà essere salvata come file HTML, ad esempio nella cartella `~/test-suites`.

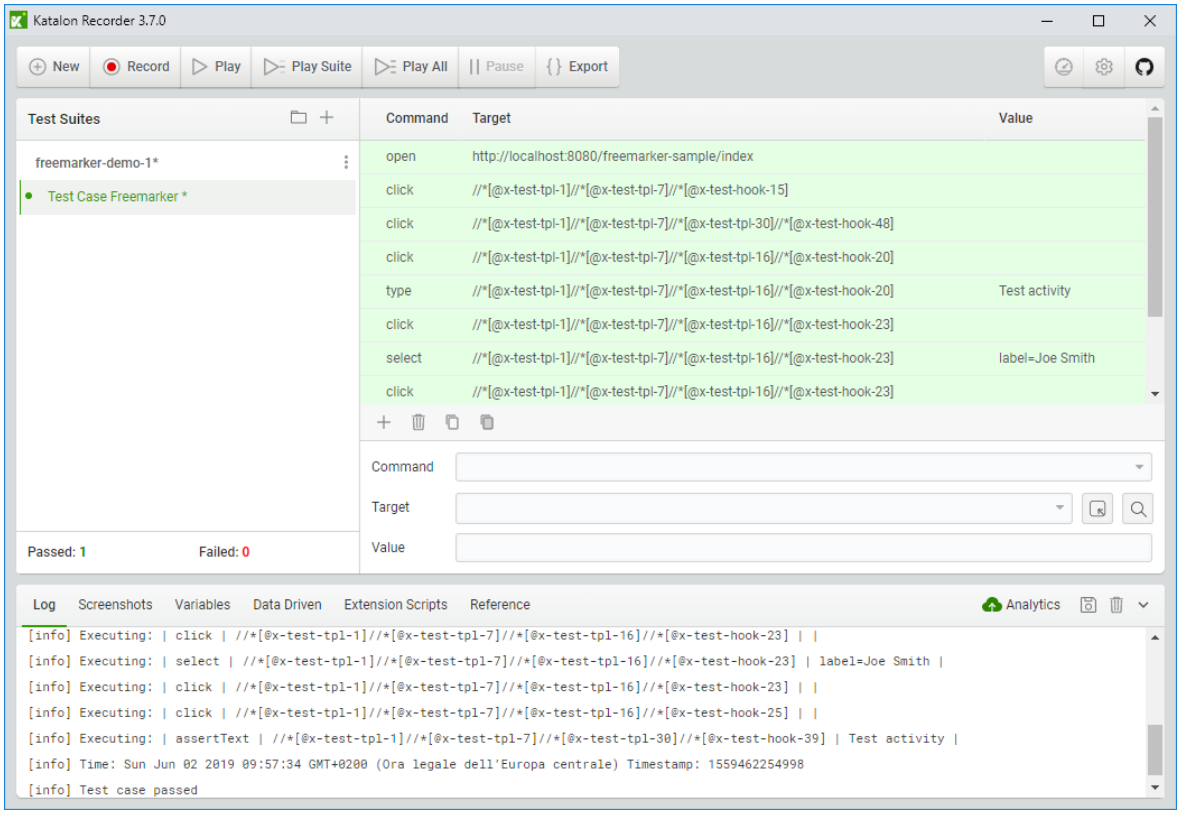


Figura 5.3.2: Registrazione test con Selenium su template FreeMarker (hook-based)

5.3.4 Fase 3: Rimozione hook inutilizzati

La rimozione di tutti gli hook inutilizzati si effettua lanciando il comando seguente dalla cartella del prototipo:

```
$ node main.js remove-hooks '../test-webapp/src/main/webapp/**/*.ftl' --grammar freemarker --suites
../test-suites
```

Confrontando il template `taskEdit.tpl` dell'applicazione AngularJS prima e dopo la rimozione degli hook, si ottiene il risultato riportato in figura 5.3.3. Sono stati cerchiati gli hook che sono coinvolti nella suite di test salvata precedentemente nella cartella `~/test-suites`, mentre tutti gli altri evidenziati in arancione, sono stati rimossi perché inutilizzati.

Per verificare che l'operazione di rimozione sia stata correttamente eseguita, si riesegue il test ottenendo di nuovo lo stesso risultato riportato in figura 5.3.2.

Figura 5.3.3: Hook rimossi: differenze su template FreeMarker

5.3.5 Fase 4: Modifica della vista

Così come fatto per l'applicazione basata su AngularJS, è stata modificata allo stesso modo la funzionalità di gestione dei task. Rimuovendo quindi la vista dedicata alla edit dei task, è stata effettuata l'operazione manuale di modifica, mantenendo tutti gli hook contenuti nel template `taskEdit.ftl`, ricollocandoli opportunamente dove necessario. In figura 5.3.4 è riportato un confronto tra la versione precedente e la successiva: gli elementi contenenti gli hook contrassegnati in verde non sono stati impattati da alcuna modifica, mentre quello contrassegnato in rosso è stato ricollocato sul tag corrispondente nella nuova versione.

5.3.6 Fase 5: Controllo di coerenza tra test e template

Anche in questo esempio, gli hook relativi al template di modifica dei task non sono stati riportati. Per rilevare tale anomalia è sufficiente lanciare il prototipo in modalità di verifica degli hook mediante il comando seguente:

```
$ node main.js verify-hooks '../test-webapp/src/main/webapp/**/*.ftl' --grammar freemarker --suites
../test-suites
```

L'output ottenuto è il seguente:

```
Error while processing ~/test-suites/freemarker-demo-1.html Error: Test hook x-test-tpl-16 not found
```

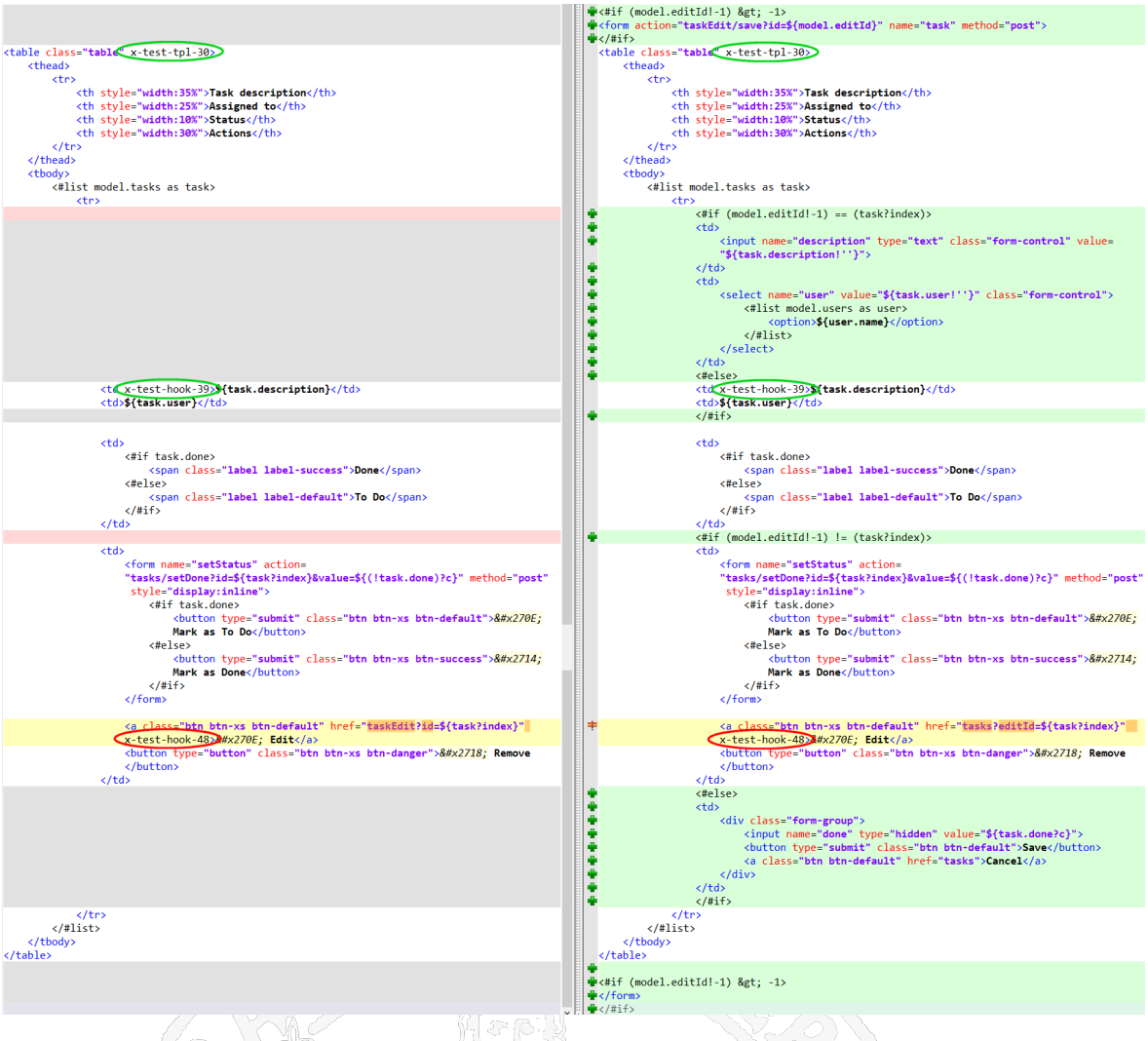


Figura 5.3.4: Modifica del template FreeMarker

```
for template:
~/test-webapp/src/main/webapp/WEB-INF/views/ftl/index.ftl
Hook path:
    x-test-tpl-1 ->
    x-test-tpl-7 ->
    x-test-tpl-16 ->
    x-test-hook-20
```

Questo risultato indica che, partendo dal template `index.ftl`, alcuni hook contenuti nella suite di test `freemarker-demo-1.html` non sono più raggiungibili; in particolare, il primo è `x-test-tpl-16` che era presente nella versione precedente nel template `taskEdit.ftl`, ora rimosso. Così come avvenuto per l'esempio AngularJS, anche in questo caso è stato possibile rilevare la condizione di incoerenza **senza dover lanciare i test**. Lanciando comunque il test per avere conferma del problema, si ottiene il risultato riportato in figura 5.3.5, come è stato

già previsto dall'operazione di controllo effettuata precedentemente.

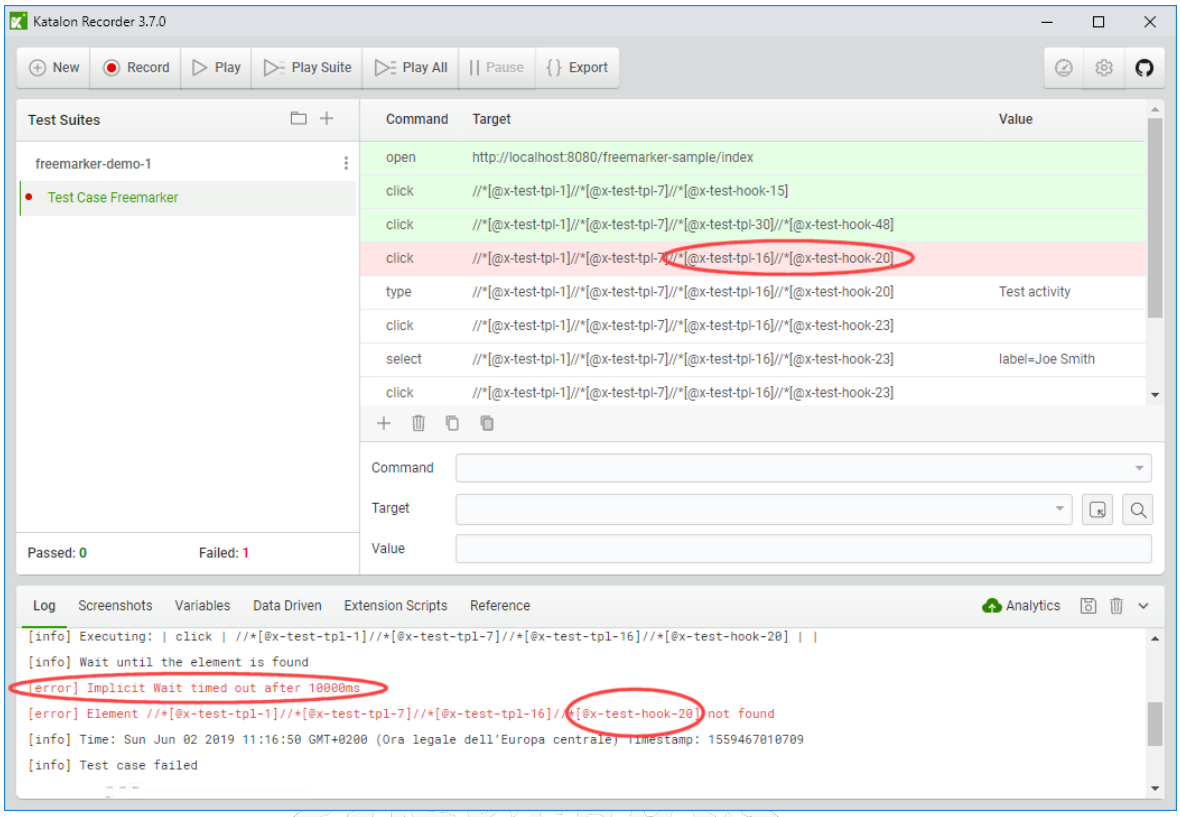


Figura 5.3.5: Test non eseguibile su applicazione FreeMarker

5.3.7 Fase 6: Test repair manuale

Per rendere di nuovo i test utilizzabili è necessario effettuare l'operazione di repair, riportando gli hook mancanti sulla nuova vista. Il primo passaggio del repair consiste nel riportare gli hook dal template `taskEdit.ftl` della versione precedente al template `tasks.ftl` della nuova versione, come riportato in figura 5.3.6.

Oltre al repair del template è necessario sostituire anche l'hook radice all'interno della suite di test, dato che il template `taskEdit.ftl` non esiste più, e i relativi hook sono stati spostati in un altro template già esistente. Quest'operazione si effettua con una semplice rimozione massiva della stringa `//*[@x-test-tpl-16]` dalla suite di test.

```
<tbody>
    <#list model.tasks as task>
        <tr>
            <#if (model.editId!-1) == (task?index)>
            <td>
                <input name="description" type="text" class="form-control" value=
                "${task.description!''}">
            </td>
            <td>
                <select name="user" value="${task.user!''}" class="form-control">
                    <#list model.users as user>
                        <option>${user.name}</option>
                    </#list>
                </select>
            </td>
            <#else>
            <td x-test-hook-39>${task.description}</td>
            <td>${task.user}</td>
            </#if>

            <td>
                <#if task.done>
                    <span class="label label-success">Done</span>
                <#else>
                    <span class="label label-default">To Do</span>
                </#if>
            </td>
            <#if (model.editId!-1) != (task?index)>
            <td>
                <form name="setStatus" action=
                "tasks/setDone?id=${task?index}&value=${(!task.done)?c}" method="post"
                 style="display:inline">
                    <#if task.done>
                        <button type="submit" class="btn btn-xs btn-default">&#x270E;
                        Mark as To Do</button>
                    <#else>
                        <button type="submit" class="btn btn-xs btn-success">&#x2714;
                        Mark as Done</button>
                    </#if>
                </form>

                <a class="btn btn-xs btn-default" href="tasks?editId=${task?index}"
                x-test-hook-48>&#x270E; Edit</a>
                <button type="button" class="btn btn-xs btn-danger">&#x2718; Remove
                </button>
            </td>
            <#else>
            <td>
                <div class="form-group">
                    <input name="done" type="hidden" value="${task.done?c}">
                    <button type="submit" class="btn btn-default">Save
                    </button>
                    <a class="btn btn-default" href="tasks">Cancel</a>
                </div>
            </td>
            </#if>
        </tr>
    </#list>
</tbody>
```

```
<tbody>
    <#list model.tasks as task>
        <tr>
            <#if (model.editId!-1) == (task?index)>
            <td>
                <input name="description" type="text" class="form-control" value=
                "${task.description!''}" x-test-hook-20>
            </td>
            <td>
                <select name="user" value="${task.user!''}" class="form-control"
                x-test-hook-23>
                    <#list model.users as user>
                        <option>${user.name}</option>
                    </#list>
                </select>
            </td>
            <#else>
            <td x-test-hook-39>${task.description}</td>
            <td>${task.user}</td>
            </#if>

            <td>
                <#if task.done>
                    <span class="label label-success">Done</span>
                <#else>
                    <span class="label label-default">To Do</span>
                </#if>
            </td>
            <#if (model.editId!-1) != (task?index)>
            <td>
                <form name="setStatus" action=
                "tasks/setDone?id=${task?index}&value=${(!task.done)?c}" method="post"
                 style="display:inline">
                    <#if task.done>
                        <button type="submit" class="btn btn-xs btn-default">&#x270E;
                        Mark as To Do</button>
                    <#else>
                        <button type="submit" class="btn btn-xs btn-success">&#x2714;
                        Mark as Done</button>
                    </#if>
                </form>

                <a class="btn btn-xs btn-default" href="tasks?editId=${task?index}"
                x-test-hook-48>&#x270E; Edit</a>
                <button type="button" class="btn btn-xs btn-danger">&#x2718; Remove
                </button>
            </td>
            <#else>
            <td>
                <div class="form-group">
                    <input name="done" type="hidden" value="${task.done?c}">
                    <button type="submit" class="btn btn-default" x-test-hook-25>Save
                    </button>
                    <a class="btn btn-default" href="tasks">Cancel</a>
                </div>
            </td>
            </#if>
        </tr>
    </#list>
```

Figura 5.3.6: Ripristino degli hook su template FreeMarker

Rilanciando a questo punto il comando di verifica degli hook, i template risultano nuovamente coerenti coi test; ciò è confermato anche dall'esecuzione del test che è di nuovo funzionante.

Capitolo 6

Sperimentazione

6.1 Premessa

Al fine di poter validare il prototipo in uno scenario quanto più realistico possibile, si è scelto di prendere come riferimento una nota applicazione open source per il settore degli e-commerce: PrestaShop. Sarebbe stato opportuno selezionare più di un'applicazione, ma a causa del lungo tempo necessario per preparare l'ambiente di sperimentazione ed eseguire materialmente i test, ciò non è stato praticabile.

Inoltre, la procedura sperimentale qui presentata si basa su una procedura che **si discosta sostanzialmente** dall'utilizzo che normalmente sarebbe previsto per la tecnica di capture/-replay e rilevazione dei breakage presentata nei capitoli precedenti. In questa sede, al fine di effettuare una valutazione quantitativa seguendo un metodo scientifico, è stato necessario effettuare operazioni di merge tra versioni diverse dei sorgenti di un'applicazione preesistente (non avendo il tempo di svilupparne una da zero) per tentare di **"simulare" l'operazione di manutenzione**, ma l'utilizzo previsto per la tecnica presentata è ***durante*** **il processo di sviluppo** (e non a posteriori), con lo scopo di favorire un'evoluzione delle viste sfruttando continuativamente il meccanismo di controllo di coerenza tra template e test, che aiuta a **prevenire i breakage *durante* gli sviluppi**, fornendo feedback sullo stato dei test allo sviluppatore. Di particolare interesse, in realtà, potrebbe essere un approfondimento sui tempi di sviluppo impiegati a correggere i test basati su XPath rispetto a quelli hook-based, prendendo in esame un team di diverse persone.

La procedura di sperimentazione è stata eseguita considerando 15 operazioni di manutenzio-

ne dei template, individuate nella commit history dell'applicazione; il numero di test e dei locatori prodotti è stato determinato dal tempo massimo a disposizione per eseguire i test.

6.1.1 Caratterizzazione locatori XPath Selenium

Per poter eseguire un confronto con i locatori hook based, è necessario caratterizzare il funzionamento dei locatori prodotti di default dallo strumento di capture/replay. Di seguito è elencata una lista di locator builder utilizzati internamente da Katalon Recorder durante la fase di capture e degli esempi di locatori prodotti da ciascuno:

- Locator Builder basato su attributi "identificatori" (`id`, `link`, `name`): utilizza il valore di uno specifico attributo dell'elemento, se presente. Esempio:

```
link=Product Catalog
```

- Locator Builder basato sui nodi vicini (`xpath:neighbor`): utilizza il testo contenuto in uno degli elementi vicini. Esempio:

```
xpath=(.//*[normalize-space(text()) and normalize-space(.)='demo_14'])[1]/preceding::a[1]
```

- Locator Builder basato sugli anchor (`xpath:link`): se l'elemento d'interesse è un anchor, ne utilizza il testo. Esempio:

```
//a[contains(text(),'Product Catalog')]
```

- Locator Builder basato sui valori dei link ipertestuali (`xpath:href`): se l'elemento d'interesse è un anchor, ne utilizza il valore dell'attributo `href`. Esempio:

```
xpath=(//a[contains(@href, '/sell/catalog/products/19#tab-step1')])[2]
```

- Locator Builder basato sul percorso XPath (`xpath:position`): utilizza un percorso XPath relativo per localizzare l'elemento di interesse. Esempio:

```
//form/div[2]/div/div/table/tbody/tr/td[4]/a
```

- Locator Builder basato su CSS (`css`): utilizza le informazioni di stile (CSS) per localizzare l'elemento di interesse. Esempio:

```
1 css=td.product-sav-quantity.text-center > a
```

Come riportato nella sezione 4.3.4, ad ogni evento lo strumento di capture/replay applica una strategia per selezionare il locatore appropriato:

- Per l'elemento di interesse vengono invocati tutti i Locator Builder, ottenendo un locatore da ciascuno.
- Tutti i locatori vengono immediatamente valutati, stabilendo così se riescono realmente a localizzare l'elemento in maniera univoca (non ambigua).
- Tra tutti i locatori che superano il controllo del punto precedente, viene selezionato il primo locatore (seguendo la priorità dei builder che li hanno prodotti) che è in grado di trovare l'elemento, inserendolo nel nuovo step registrato.

Trattandosi di una selezione automatica dello strumento di capture/replay (normalmente non controllabile dall'esterno), nella comparazione si considereranno tutte le tipologie di locatori come **equivalenti tra loro** (non verrà fatta un'analisi separata per ciascuna tipologia). Verrà pertanto considerata nel confronto **l'intera strategia di generazione dei locatori** descritta precedentemente, e non le singole tipologie di locatori. L'unica tipologia ad essere esclusa dall'analisi è stata quella che produce locatori basati sui contenuti testuali soggetti ad internazionalizzazione, dato che questi sono influenzati dai cambiamenti introdotti dai traduttori (processo che è fuori dal controllo degli sviluppatori e dei tester). Pertanto, rimanendo sullo stesso livello di genericità, sono stati considerati i locatori che fanno riferimento a tutti gli altri elementi della struttura delle pagine (questa scelta è comunque presa in considerazione nella sezione relativa alle threats to validity).

6.1.2 Caratterizzazione delle operazioni di manutenzione

La scelta delle operazioni di manutenzione sulle quali effettuare la sperimentazione ha seguito i criteri seguenti:

- Sono stati ordinati i file dei template sia in base al numero di commit nel corso del tempo che alla dimensione, in maniera decrescente. I template coinvolti in un maggior numero di commit hanno probabilmente subito un **numero maggiore** di modifiche rispetto agli altri; quelli di maggior dimensione hanno una complessità più elevata, per cui è più probabile che abbiano subito modifiche **impattanti** per i test.
- Seguendo l'ordine del punto precedente, sono stati individuati i commit nei quali i template sono stati modificati maggiormente, sia in termini di contenuti che di layout.

Per consentire l'esecuzione dell'applicazione nella versione precedente e successiva ad ogni modifica, è stato necessario apportare alcune correzioni anche al modello dati mock sottostante, facendo comunque in modo che **i dati "finali" presentati sulle pagine coinvolte dai test restassero invariati tra le due versioni**. Per fare un esempio, questa condizione si è verificata quando il modello dati legato alla vista è evoluto insieme a quest'ultima, a causa di un cambio di nomi ai campi degli oggetti contenenti i dati.

Di seguito è elencata la lista delle modifiche considerate sulle varie funzionalità, con i commit ID[1] di riferimento e una breve descrizione di ciascun intervento:

1. Order Detail (`4afd421e - 58b8d357`): modificata sezione dati corriere; cambiati link di fatturazione; cambiata sezione indirizzi di consegna.
2. Order Detail (`226b2193 - 5d963a9f`): cambiata la tabella del dettaglio prodotti restituiti; spostato il campo di identificativo dell'ordine; cambiata la tabella delle personalizzazioni prodotto in layout responsive; modificata la sezioni dei totali.

[1] È possibile visualizzare ciascun commit sostituendo il relativo id nell'URL seguente: `https://github.com/PrestaShop/PrestaShop/commit/[[Commit-Id]]`

3. Order Detail - no return (`226b2193 - 5d963a9f`): spostato il campo di identificativo dell'ordine; cambiata la tabella delle personalizzazioni prodotto in layout responsive; modificata la sezioni dei totali.

4. Stores (`4afd421e - ef66b3b8`): ridisegnata la sezione dei contatti e degli orari di apertura dei punti vendita. La maggior parte dei tag sono stati sostituiti.

5. Checkout Product (`0bf5e870 - 0282aa3b`): ridisegnata la sezione personalizzazioni di prodotto nell'ordine. Tutti i tag sono stati cambiati.

6. Checkout Cart Totals (`9f4c97cf - a95807b7`): spostata sezione tasse nel riepilogo dei totali dell'ordine.

7. Checkout Address Step (`6f03ad81 - 4afd421e`): spostato il template dello step di inserimento indirizzo dell'ordine; aggiunto pulsante di inserimento di nuovo indirizzo nella stessa posizione del submit precedente.

8. Newsletter Block (`1dedad8f - 14f429d8`): cambiata posizione messaggio di avviso all'interno del componente newsletter.

9. Currencies Block (`8a2a95c5 - 9f8838d0`): sostituito all'interno del componente per la selezione delle valute.

10. Blockcart Product Line (`4502529c - 2880997b`): modificati alcuni tag del componente riga articolo del carrello.

11. Product - Catalog (`cf39da64 - 0a1fff75`): modificati i nomi degli elementi contenenti i dati di dettaglio degli articoli; nelle sezione "caratteristiche articoli" sostituito tag unordered list e tag figli con tag description list e figli.

12. Checkout Delivery Step (`7bd2f53f - 6662a7e3`): cambiato il livello di innestamento di alcuni elementi contenenti i dati di consegna dell'ordine.

13. Order Messages (`226b2193 - 5d963a9f`): modificato componente dei messaggi dell'ordine. La tabella HTML è stata sostituita dal layout responsive.

14. Order Followup (`1644fd94 - 9acef1ff`): invertito l'ordine di alcune colonne della tabella dei resi; Modificato il layout della tabella di dettaglio dei prodotti di ciascun reso (diversi elementi rinominati).

15. Contact Form (`4502529c - 1181556a`): modificata la lista dinamica dei campi della form di contatto. Cambiati i nomi e le proprietà di diversi elementi.

6.2 Descrizione del processo

6.2.1 Research questions

Si vuole valutare di quanto i test basati su hook injection siano più robusti rispetto ai test basati su locatori basati su XPath prodotti da uno strumento come Selenium, considerando l'evoluzione di un'applicazione web mediante refactoring che potenzialmente causano breakage dei test. Dato che, in entrambi i casi, la robustezza del test dipende direttamente dalla robustezza dei locatori, le valutazioni verranno effettuate direttamente su questi ultimi.

L'obiettivo pertanto è verificare di quanto la tecnica hook-based riesce a ridurre il numero di locatori broken rispetto a XPath.

6.2.2 Objects

- Un'applicazione open source (Prestashop e-commerce), OP

6.2.3 Factors

- Operazioni di manutenzione V_x delle viste (template). Per ciascuna V_x è fissato un numero K_x di locatori presenti nel test che utilizza le viste modificate in V_x. Sono state considerate 15 operazioni di manutenzione (V_1, V_2, ... V_{15}).

6.2.4 Independent Variables

- Tipologia di locatori L (variabile categorica a due livelli). Valori possibili: T_S (Selenium XPath locators, prodotti dalla strategia di generazione di Selenium descritta in 6.1.1), T_H (Hook-based locators).

6.2.5 Dependent Variables

- Numero di locatori broken $B_x(L)$ (variabile numerica discreta, $B_x(L) \geq 0$) nella specifica versione V_x.

6.2.6 Experimental Procedure

- Per ciascuna V_x (avente K_x locatori), utilizzando il tipo di locatore L si esegue la registrazione del test prima della manutenzione e si valuta il numero di locatori broken $B_x(L)$ a seguito della manutenzione.
- Per la tipologia di locatori T_H (Hook-based locators) viene effettuato preliminarmente il merge dal commit precedente a quello successivo per propagare gli hook nel codice delle viste. In questa fase si risolvono manualmente eventuali conflitti dovuti al merge.
- Si calcola la differenza tra **il numero** di locatori broken per le due tecniche, ovvero tra $B_x(T_S)$ e $B_x(T_H)$. Questa quantità dipende dal numero totale di locatori K_x per la specifica V_x, dato che **questo valore K_x non è costante**, ma varia per ciascuna V_x. La differenza del numero di locatori broken si identifica con $R_x = B_x(T_S) - B_x(T_H)$.

6.3 Risultati e analisi quantitativa

Nella tabella 6.1 sono riportate le operazioni di manutenzione e il valore di R_x e K_x per ciascuna di esse. Si può notare che il numero di breakage dei test basati sui locatori hook-based è costantemente zero, grazie alle segnalazioni fornite durante le operazioni di merge

dallo strumento di verifica di consistenza tra template e test; ciò potrebbe non essere vero però in generale, dato che esistono altre potenziali cause dovute ad errore umano, come gli hook posizionati male durante la fase di merge, una duplicazione di hook, o altri problemi legati ai test e non agli hook (come problemi di sincronizzazione).

#	Operazione di manutenzione	$B_x(T_S)$	$B_x(T_H)$	R_x	K_x
1	Order Detail Template layout change	9	0	9	15
2	Order Detail Template layout change	8	0	8	15
3	Order Detail (no return) Template layout change	8	0	8	13
4	Stores Page Template layout change	4	0	4	11
5	Checkout Product Template layout change	2	0	2	12
6	Checkout Cart Totals Template layout change	3	0	3	7
7	Checkout Address Step Template layout change	1	0	1	8
8	Newsletter Block Component redesign	1	0	1	4
9	Currencies Block Component redesign	1	0	1	2
10	Blockcart Product Line Template layout change	2	0	2	5
11	Product (Catalog) Template layout improvements	3	0	3	14
12	Checkout Delivery Step Template redesign	2	0	2	10
13	Order Messages Component layout change	1	0	1	8
14	Order Followup Template layout change	1	0	1	5
15	Contact Form Template layout change	2	0	2	6

Tabella 6.1: Risultati sperimentazione

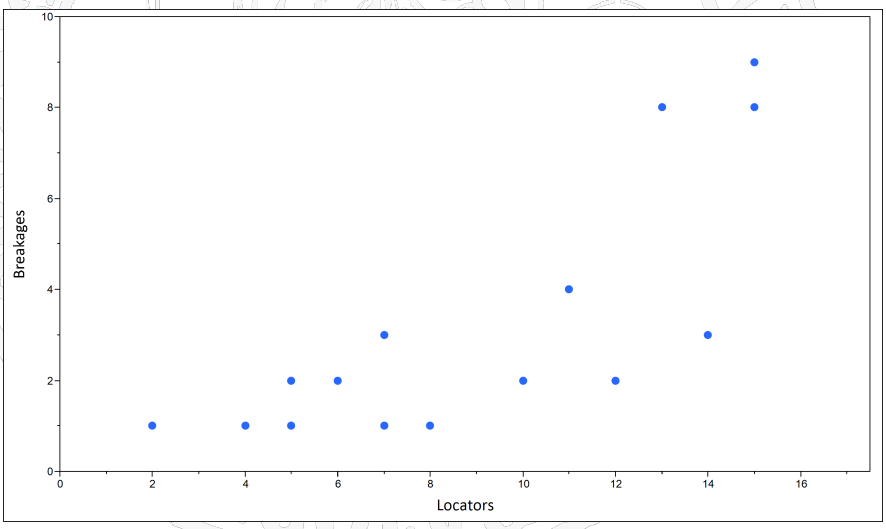


Figura 6.3.1: Diagramma di dispersione

In figura 6.3.1 è riportato un diagramma di dispersione che mostra il numero di breakage al variare del numero di locatori per test. A causa del criterio adottato nella scelta delle modifiche considerate (trattato nella sezione 6.1.2), la **frequenza reale** di test breakage che si riscontra in media utilizzando i locatori prodotti dalla strategia di Selenium è probabilmente più bassa di quella misurata.

6.4 Analisi qualitativa

In questa sezione verranno analizzate le principali cause di breakage rilevate durante la sperimentazione, con riferimento alla tassonomia riportata nella sezione 2.4. Le seguenti tipologie di cause **sono state escluse** dall'analisi per le ipotesi fatte in precedenza:

- Valori: per l'ipotesi fatta sulle precondizioni imposte sui dati (sezione 3.4), non è contemplato il breakage dei test dovuto ad un cambiamento dei dati tra le diverse esecuzioni dei test.
- Ricaricamento della pagina: avendo analizzato unicamente la validità dei **singoli locatori** presenti nei test e non l'intero flusso di esecuzione di ciascun test (nel quale possono essere previsti cambi o ricaricamenti di pagina), non è stato necessario valutare questa tipologia di cause.
- Tempo di sessione utente: è una tipologia che non è stata considerata in questo lavoro di tesi, avendo soltanto affrontato le problematiche relative agli impatti diretti che le **modifiche ai template** hanno sui test e non il cambio di condizioni al contorno quali la durata della sessione utente.

Di seguito sono riportate invece le cause di breakage che sono state effettivamente rilevate durante la sperimentazione, seguendo la tassonomia indicata precedentemente. Come già riportato nei risultati dell'analisi quantitativa, l'utilizzo dei locatori hook based **ha consentito di evitare gli scenari di breakage** descritti. Nei casi considerati, probabilmente il livello di complessità della modifica e la quantità di test non sono stati tali da far emergere condizioni

"al limite" per cui si verificassero breakage anche nel caso dei locatori hook based (ovvero, condizioni non rilevabili dallo strumento di controllo coerenza). Ciò nonostante, **non è possibile comunque affermare che non esistano in assoluto condizioni in cui i breakage non si verificano**, dato che non è stata prodotta una dimostrazione formale di questa ipotesi.

Locatori attribute-based

I breakage riguardanti i locatori basati su attributi che sono stati cambiati nell'operazione di manutenzione rappresentano solo una minima parte dei breakage rilevati. In particolare, nei test vi sono state due tipologie di breakage legati ai locatori attribute-based:

- Attributo non trovato: questa tipologia è stata rilevata in quei casi in cui la riorganizzazione del template ha provocato cambiamenti tali da non poter riapplicare più gli stessi attributi, o nei casi in cui i valori di alcuni attributi sono stati modificati volutamente dagli sviluppatori. Un esempio di locatore broken rilevato è il seguente:

```
1 //article[@id='invoice-address']/header/h1
```

  In questo esempio, essendo stata modificata la sezione relativa agli indirizzi degli ordini, non è stato riportato l'attributo `id='invoice-address'` sull'elemento presente nella versione successiva. Il corrispondente locatore basato su hook è invece il seguente:

```
1 //*[@x-test-tpl-766]//*[@x-test-hook-769]//*[@x-test-tpl-591]//*[@x-test-hook-594]
```

  L'attributo `x-test-tpl-591` fa riferimento alla **radice** del template in cui è contenuto l'elemento article, mentre `x-test-hook-594` fa riferimento all'elemento **foglia** `h1`, pertanto non è influenzato dal cambiamento riportato.

- Testo dell'elemento non trovato: questa tipologia è stata rilevata quando il locatore faceva riferimento ad un dato contenuto in un elemento HTML, non più presente (o spostato) nella versione successiva. Un esempio di locatore broken rilevato è il seguente:

```
xpath=(.//*[normalize-space(text()) and normalize-space(.)='Puma Runner'])[1]/following::td[1]
```

In questo esempio, il locatore non è più valido perché la posizione in cui è collocato il testo "Puma Runner" dopo la modifica è cambiato, e pertanto non è più in grado di localizzare l'elemento d'interesse. Il corrispondente locatore basato su hook è il seguente:

```
//*[@x-test-tpl-766]//*[@x-test-hook-769]//*[@x-test-tpl-372]//*[@x-test-hook-384][1]
//*[@x-test-hook-389]
```

L'elemento contenente l'attributo `x-test-hook-384` corrisponde a quello in cui è cambiata la collocazione del testo "Puma Runner". Essendo il locatore insensibile rispetto ai contenuti, è bastato riportare gli hook sugli elementi corrispondenti durante il merge. In questo esempio si può anche vedere l'applicazione del criterio di selezione basato sugli indici anche nel caso degli hook (condizione $S_e \not\equiv \varnothing$ di minimizzazione del percorso, in sezione 3.4.2), dato che `x-test-hook-384` fa riferimento ad una riga di una tabella ripetuta in base ai dati.

Locatori hierarchy-based

La quasi totalità dei breakage rilevati riguardano i locatori basati su espressioni XPath assolute o relative facenti riferimento ad elementi che sono stati cambiati nell'operazione di manutenzione. I casi che si sono verificati sono i seguenti:

- Cambiamento del nome del tag: questa tipologia si verifica quando cambia il tag per motivi di stilizzazione o di struttura del template. Un esempio di locatore broken rilevato è il seguente:

```
//table/tfoot/tr/td[2]
```

In questo esempio, la tabella è stata convertita in un grid layout responsive utilizzando i tag `div`, pertanto non è più possibile dopo la modifica utilizzare questo locatore

per intercettare l'elemento d'interesse. Il corrispondente locatore basato su hook è il seguente:

```
1 //*[@x-test-tpl-766]//*[@x-test-hook-769]//*[@x-test-tpl-372]//*[@x-test-hook-414]
```

Durante il merge, l'hook `x-test-hook-414` è stato spostato dal tag `td` al tag `div` corrispondente della nuova griglia responsive.

- Cambiamento dell'innestamento dei template: questa tipologia si verifica quando un template viene riorganizzato in diversi sotto-template, utilizzando punti di innesto diversi rispetto a quanto previsto dall'espressione XPath originale. Si sono verificati due casi:
  - Aggiunta di un livello di innestamento (suddivisione di un template in sotto-template): almeno nei casi considerati non ha comportato problemi, ma ciò potrebbe causare breakage dovuti ad ambiguità di innestamento; per risolvere il problema è necessario iniettare nuovamente gli hook e **correggere manualmente** i locatori nei test, aggiungendo gli hook relativi al nuovo punto di innestamento. Questa condizione **non è segnalata** dal controllo di coerenza.
  - Rimozione di un livello di innestamento (accorpamento di template "figli" nel template "genitore"): nei casi considerati è stato necessario **correggere manualmente** i locatori nei test, rimuovendo i templare hook (radice) non più esistenti (condizione segnalata dal controllo di coerenza).

Locatori index-based

Una parte dei breakage rilevati riguardano i locatori basati su espressioni XPath contenenti uno o più indici nel percorso. Un esempio di locatore broken in questo caso è il seguente:

```
1 //section[@id='content']/section[2]/form/footer/button
```

In questo esempio, pur essendo rimasta sostanzialmente invariata la struttura del template, l'elemento `section` considerato è stato spostato all'inizio, rendendo questo locatore

non più funzionante a causa del cambio di indice rispetto all'altro elemento omonimo. Il corrispondente locatore basato su hook è il seguente:

```
1 //*[@x-test-tpl-766]//*[@x-test-hook-769]//*[@x-test-tpl-447]//*[@x-test-hook-462]
```

In questo esempio, l'indice della sezione non è dovuto ad elementi prodotti dinamicamente dal template engine, ma si tratta di due **sezioni statiche**, che pertanto contengono elementi contrassegnati con **hook differenti**. Essendo l'attributo `x-test-tpl-447` presente sulla radice del template, è stato necessario soltanto ricollocare l'attributo `x-test-hook-462` sul pulsante contenuto all'interno della sezione spostata.

6.5 Problemi riscontrati

Per poter effettuare la sperimentazione, come primo step si è tentato di avviare l'applicazione effettuando una build di una versione risalente a gennaio 2016, periodo in cui vi sono stati diversi refactoring delle viste, dovute ad un restyling e a diversi improvement. Dopo diversi tentativi, purtroppo non è stato possibile lanciare una versione funzionante dell'applicazione così datata a causa di alcune dipendenze da componenti esterni (mediante il package manager Composer) che non sono più reperibili alle versioni corrette di allora (il descrittore del package manager aveva puntamenti sempre alle ultime versioni). Per questo motivo è stato avviato soltanto un subset di componenti infrastrutturali (il template engine, in particolare) in modo da poter effettuare comunque il rendering dinamico delle viste.

I modelli dati a cui fanno riferimento i template sono stati ricostruiti come dei mock mediante reverse engineering e non attingendo dalla base dati relazionale. In linea di principio ciò è anche più coerente con il principio di testing di unità: testando un componente specifico, se ci sono dipendenze da altri componenti complessi, è conveniente crearne dei mock per lo scenario d'interesse, in modo da limitare la complessità del test. Quest'aspetto è segnalato anche nelle threats to validity.

Per far funzionare quindi il capture di Selenium e il successivo replay dei test, è stato avviato

il template engine per servire al browser le pagine dell'applicazione Prestashop, utilizzando un modello di dati mock (che ovviamente è stato mantenuto identico, in termini di contenuto informativo, tra versioni successive).

6.6 Threats to validity

6.6.1 Internal validity

Selection:

- La selezione **manuale** delle coppie di versioni di una data applicazione potrebbe portare a biasing dei risultati. A questa minaccia concorre anche il particolare **criterio di selezione** adottato (seguendo l'ordinamento basato su numero di cambiamenti e dimensione dei template, come descritto nella sezione 6.1.2), accentuando le differenze di prestazioni tra le due soluzioni analizzate.
- Per evitare dipendenze dell'esperimento dal fattore internazionalizzazione, dei possibili locatori prodotti da Selenium sono stati usati per il confronto soltanto quelli che non dipendessero dai valori testuali localizzati (come ad esempio le label dei pulsanti). Dato che i locatori hook-based non fanno riferimento ai valori localizzati degli elementi statici della pagina, è stato necessario confrontare locatori che avessero lo stesso grado di genericità. Pur confrontando locatori Selenium dipendenti dalla localizzazione, questo punto comunque è da considerare come minaccia alla validità, dato che si effettuerebbe un confronto tra elementi disomogenei in quanto a contenuto informativo (gli uni dipendono dalla struttura e dalla localizzazione, gli altri solo dalla struttura).

Instrumentation:

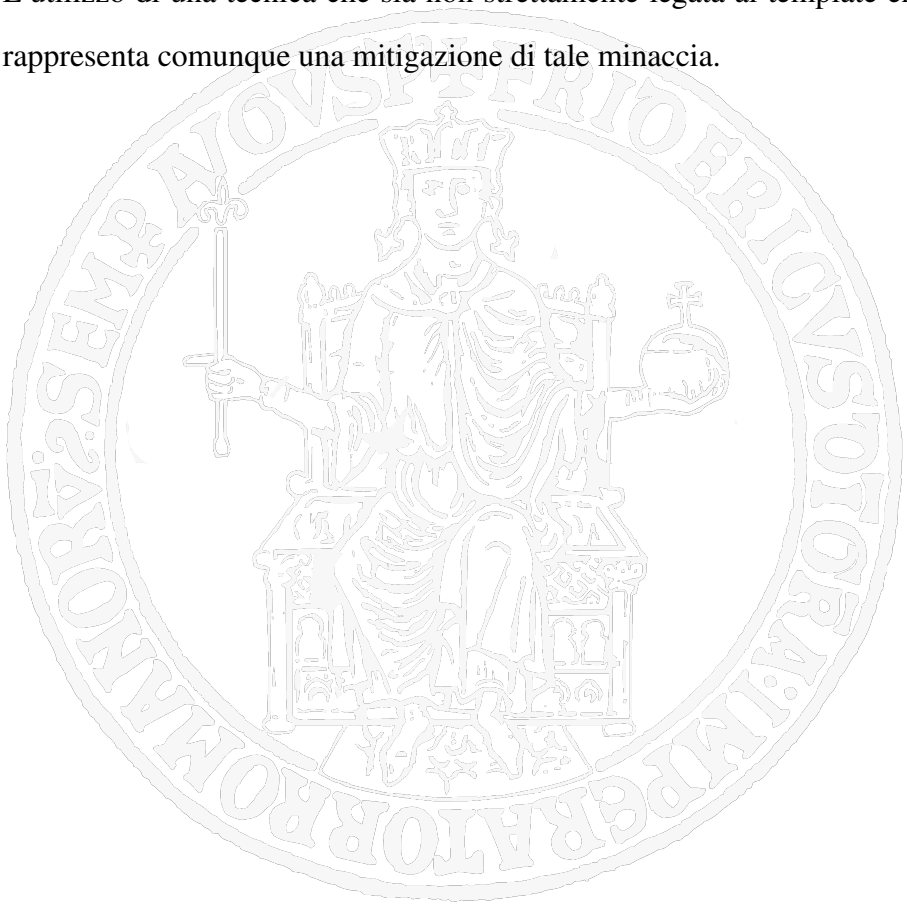
- Non essendo possibile eseguire l'applicazione nel suo contesto originale completo, a causa di dipendenze mancanti e non facilmente recuperabili alle versioni corrette, è

stato messo su un ambiente di "test harness" per tentare di eseguire comunque l'applicazione con componenti minimali. Ciò non dovrebbe comportare una minaccia significativa, perché la composizione del DOM non dipende dai componenti non caricati, ma solo dal binding col ViewModel. Ad ogni modo, essendo presente quest'anomalia è opportuno segnalarla come eventuale minaccia interna, dato che non si può escludere che una differente configurazione del test harness possa condurre a risultati diversi.

6.6.2 External validity

Population validity:

- A causa del lungo tempo necessario alla preparazione dell'ambiente sperimentale e alla difficoltà di reperimento di applicazioni open-source che potessero essere significative, è stata selezionata solo un'applicazione; ciò costituisce una minaccia alla generalità. L'utilizzo di una tecnica che sia non strettamente legata al template engine in esame rappresenta comunque una mitigazione di tale minaccia.



Conclusioni e sviluppi futuri

Il processo e gli strumenti esposti in questo lavoro di tesi costituiscono un ulteriore passo in avanti nel settore del testing automatizzato. Diverse applicazioni web open source, industriali e istituzionali sono sviluppate con tecnologie di templating e possono trarre beneficio dai test automatici, migliorando il processo di sviluppo mediante la tecnica proposta.

Diversi articoli accademici hanno contribuito alle idee alla base di questo progetto, in particolar modo l'approccio incrementale proposto in [8] e la strategia di generazione di locatori robusti proposta in [11]. Gli strumenti di supporto del processo di sviluppo mostrati in questo lavoro di tesi, sia per quanto riguarda la possibilità di adattamento a tecnologie di templating diverse, sia per le prestazioni quantitative e qualitative ottenute, hanno mostrato un livello di efficacia soddisfacente nel rilevare le condizioni di test breakage, che potenzialmente potrebbero avere impatti notevoli sui tempi di sviluppo e manutenzione del software. I margini di miglioramento in termini di robustezza, genericità e usabilità sono tuttavia da tenere in considerazione; nei punti seguenti sono riportati diversi spunti di approfondimento:

- Automatizzazione del controllo coerenza: attualmente implementato come procedura *on demand*, è possibile velocizzare ulteriormente il meccanismo di controllo coerenza tra test e template creando una versione *online* (modalità "watchdog") che possa segnalare le problematiche di breakage all'atto del salvataggio dei singoli file di template.

- Conversione di locatori degli script preesistenti: è possibile in linea di principio implementare un convertitore di script in grado di tradurre a runtime i locatori di default in quelli hook-based, durante la fase di riesecuzione dei test.

- Integrazione con gli IDE: gli automatismi implementati possono essere integrati con gli ambienti di sviluppo per dare indicazioni visuali già negli editor dei sorgenti, con-

sentendo di valutare in maniera immediata gli impatti delle modifiche (funzioni di "find references", highlighting, ecc.).

Oltre ai miglioramenti proposti, vi sono altre tematiche di ricerca di particolare interesse:

- Una sperimentazione per valutare tempi di sviluppo impiegati a correggere i test basati su XPath rispetto a quelli hook-based sarebbe utile per giustificare l'introduzione degli strumenti proposti nei processi di sviluppo, mostrando gli eventuali vantaggi in termini temporali.
- Si potrebbe valutare l'utilizzo misto hooks e XPath minimizzati (tecnica ROBULA o similari) per abbassare ulteriormente l'impatto sui sorgenti.
- Il porting della tecnica e degli strumenti su altre piattaforme (es. mobile, ecc) è un'ulteriore tematica da analizzare.

Ringraziamenti

Al termine di questo lavoro di tesi vorrei esprimere il mio ringraziamento a coloro che in un modo o nell'altro hanno accompagnato la mia vita negli anni di studi trascorsi. Anzitutto ringrazio il mio relatore, il prof. Porfirio Tramontana per la disponibilità, la competenza, i consigli e la pazienza nel sostenermi in tutto il periodo di svolgimento del lavoro di tesi. Un ringraziamento speciale va al mio datore di lavoro, il dott. Paolo Tamburrino per avermi dato la possibilità di completare gli studi in serenità, offrendomi un ambiente e un clima di lavoro meraviglioso, nel quale ho avuto l'occasione di crescere sia professionalmente che umanamente, sentendomi di fatto sempre *a casa*; ringrazio tutti i miei colleghi di lavoro che mi supportano (*e sopportano!*) quotidianamente, con i quali ho condiviso in questo tempo tante nuove sfide da affrontare e traguardi raggiunti con successo. Ringrazio tutti i miei amici che mi hanno incoraggiato a non mollare e a tenere d'occhio il traguardo. Ma il ringraziamento più importante va alla mia famiglia, a mio padre Luigi, a mia madre Pina e a mia sorella Adriana, per il loro grande affetto che posso sempre ritrovare, sia nei momenti belli che in quelli difficili. Un grazie di cuore.

Pierantonio

Appendice A

Tecnologie

A.1 AngularJS

AngularJS è un framework strutturale open source per le applicazioni web dinamiche, sviluppato in origine da Miško Hevery e Adam Abrons, attualmente supportato da Google. AngularJS consente l'utilizzo diretto di codice HTML standard come linguaggio di templating, estendendone la sintassi in modo da esprimere la struttura dell'applicazione in componenti, in maniera chiara e succinta. I meccanismi di data binding e dependency injection eliminano gran parte del boilerplate code che è normalmente necessario per scrivere un'applicazione web dinamica; inoltre, tutto il funzionamento di AngularJS avviene direttamente nel browser, rendendolo pertanto un ottimo complemento per qualsiasi tecnologia server side. Di seguito sono elencati alcuni aspetti peculiari di AngularJS:

- Fornisce agli sviluppatori l'opzione di scrivere applicazioni lato client utilizzando JavaScript utilizzando un modello MVC chiaro e ordinato.
- Le applicazioni scritte con questa tecnologia sono nativamente compatibili con tutti i browser. Le particolarità e le differenze implementative del DOM e di JavaScript per ciascun ambiente sono automaticamente gestite dal framework, sollevando l'utente dalla gestione dei problemi di compatibilità.
- È un framework open source (rilasciato sotto licenza MIT), completamente gratuito, utilizzato da milioni di sviluppatori (secondo una statistica pubblicata da SimilarTech[1]

[1] https://www.similartech.com/technologies/angular-js

si stima che, al 2019, oltre 410.000 siti web utilizzino tale tecnologia) e grazie all'enorme diffusione è possibile reperire facilmente online componenti e librerie aggiuntive sviluppate dagli utenti.

- Consente di strutturare applicazioni su larga scala, semplici da manutenere nel tempo e con ottime performance.

Le caratteristiche principali del framework AngularJS sono le seguenti:

- **Two-way data binding**: consiste nella sincronizzazione bidirezionale automatica dei dati tra il modello e i componenti della vista.
- **Scope**: sono oggetti referenziabili dal modello, rappresentano il punto di contatto tra il controller e la vista.
- **Controller**: sono composti da un'insieme di funzioni accessibili dalla vista mediante lo Scope e implementano la logica gestione degli eventi prodotti dall'utente.
- **Service**: sono oggetti singleton che implementano funzionalità di supporto specifiche, come ad esempio il servizio `$resource` utilizzato per astrarre il concetto di risorsa REST.
- **Filters**: utilizzati per processare liste di elementi, consentono di applicare delle logiche di selezione su insiemi di elementi, in maniera dichiarativa.
- **Directives**: rappresentano uno dei componenti fondamentali del framework e vengono richiamate all'interno del DOM mediante elementi, attributi, css ed altro. Possono essere utilizzate per creare tag personalizzati al fine di implementare nuovi componenti grafici o dinamizzare parti della vista.
- **Templates**: vengono renderizzati con l'aggiunta di informazioni provenienti dal modello, sotto la gestione del controller. Possono essere file singoli o composti da più parti mediante la direttiva di inclusione `ng-include`.

- **Routing**: consente di strutturare l'applicazione in sezioni secondo una logica ad albero e gestisce la navigazione.

A.2 NodeJS

NodeJS è una piattaforma per l'esecuzione di codice JavaScript server-side basata sull'interprete V8 di Google Chrome; è stato sviluppato inizialmente da Ryan Dahl nel 2009. È particolarmente indicato per le applicazioni che fanno uso intensivo di I/O, grazie all'approccio event-driven non bloccante, che lo rende un ambiente di esecuzione leggero ed efficiente.

Caratteristiche principali

Le caratteristiche principali di NodeJS sono elencate di seguito:

- **Ambiente asincrono ed event-driven**: tutte le API della libreria di NodeJS sono asincrone, ovvero non bloccanti: ciò significa essenzialmente che il codice eseguito su NodeJS non resta in attesa di risposta quando viene invocata una funzione dell'API, ma procede immediatamente con le istruzioni successive; grazie ad un meccanismo di notifica di eventi, la risposta rimasta in sospeso viene consegnata (quando pronta) tramite una callback.
- **Single threaded**: il codice eseguito in NodeJS utilizza un unico thread mediante event looping. Ciò rende superfluo l'utilizzo di primitive di sincronizzazione normalmente adottate in contesti multi-thread ed evita le problematiche legate a deadlock e starvation; oltre a ciò il context switch per la riattivazione dei thread viene completamente azzerato.
- **Scalability**: la scalabilità su sistemi multi-core o multi-processor si ottiene grazie alla funzionalità nativa di clustering che consente l'avvio simultaneo di più processi NodeJS, che comunicano mediante scambio di messaggi e callback.

- **Fullstack language**: utilizzando NodeJS per la parte server nello sviluppo di applicazioni web, è possibile sfruttare lo stesso linguaggio (e spesso anche buona parte del codice) sia per la parte frontend che per la parte backend.

Componenti

I componenti principali su cui si basa NodeJS sono i seguenti:

- **V8**: è il motore JavaScript di Google, scritto in C++ e impiegato nel browser Chrome. NodeJS fa affidamento sull'API di V8 per interpretare ed eseguire il codice Javascript.
- **libuv**: è una libreria di supporto multi-piattaforma che implementa le operazioni di I/O asincrono ed è scritta in C. È stata sviluppata principalmente per essere usata da NodeJS, ma ne esistono i binding anche per Lua, Julia, Python ed altri linguaggi. Questa libreria fornisce i meccanismi per gestire il file system, i DNS, la rete, i processi figli, la gestione dei segnali, il polling e lo streaming. Include anche un pool di thread (chiamato Worker Pool) per ripartire il carico di alcune attività non nativamente asincrone a livello di sistema operativo.
- **OpenSSL**: è una libreria general purpose per la crittografia. È utilizzata in gran parte dai moduli `crypto` e `tls` accessibili dal codice JavaScript.
- **zlib**: è una libreria per la compressione dati.
- **Binding C/C++**: consentono di accedere a librerie scritte in C o C++ basate su N-API, utilizzate per creare plugin nativi per NodeJS.

Nella figura A.2.1 è riportato uno schema semplificato dei livelli architetturali di NodeJS.

Runtime

Non appena viene avviato, subito dopo una fase di inizializzazione, l'esecuzione di NodeJS entra nell'Event Loop (anche definito Main Thread o Event Thread), che concettualmente

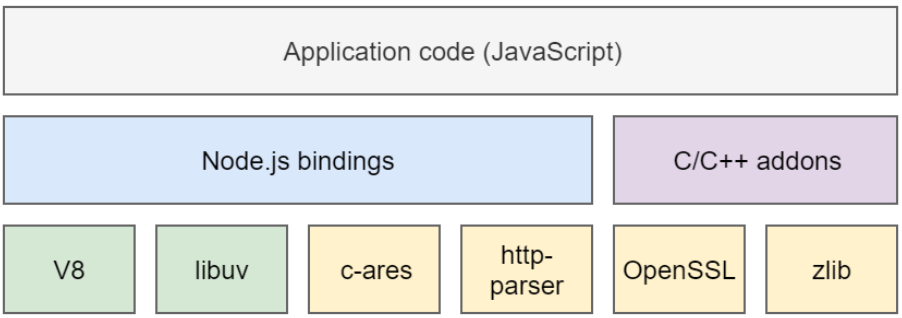


Figura A.2.1: Architettura di NodeJS

è pensato per rispondere alle richieste dei client eseguendo le callback opportune. Ciascuna callback JavaScript è eseguita in maniera sincrona e "atomica", ovvero non può essere interrotta da altre callback prima del suo completamento. Durante l'esecuzione di una callback è possibile utilizzare l'API di NodeJS per registrare ulteriori richieste asincrone, che continueranno l'esecuzione dopo il termine della callback corrente. Alcuni esempi di funzioni dell'API di NodeJS che effettuano richieste asincrone includono i timer (`setTimeout`, `setInterval`, ecc.), le funzioni dei moduli `fs` (per il filesystem) e `http` (per i servizi web) e molti altri.

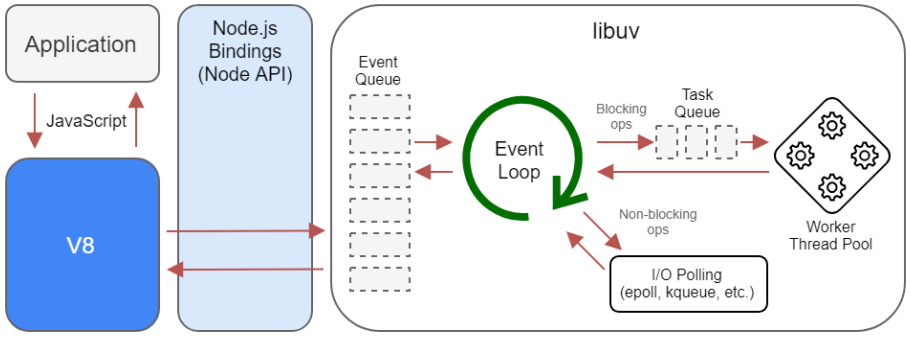


Figura A.2.2: Runtime di NodeJS

La figura A.2.2 rappresenta una versione semplificata del runtime di NodeJS. L'event loop è implementato mediante un singolo thread utilizzando le API di `libuv` e viene definito semi-

infinito, dato che si arresta nel momento in cui non ci sono più attività da eseguire in coda; dal punto di vista dello sviluppatore, è il momento nel quale il programma termina.

Il Worker Pool, che ha dimensione fissa, viene utilizzato per eseguire alcune operazioni CPU-intensive o che comunque non possono essere eseguite su tutti i sistemi operativi in modalità non bloccante (operazioni di crittografia, operazioni su filesystem, lookup DNS, ecc.). Inoltre, anche alcune librerie native di terze parti come `bcrypt` ripartiscono il carico computazionale sui thread del Worker Pool.

NPM

NPM è sia il nome del package manager che del principale repository di moduli pubblico per NodeJS. Esso consiste di tre componenti principali:

- **Il sito web**: per la ricerca di moduli in base a parole chiave.
- **La Command Line Interface (CLI)**: per eseguire operazioni di prelievo, aggiornamento, rimozione e manutenzione delle dipendenze.
- **Il registry**: è un database che ospita i pacchetti dei moduli per NodeJS, sia pubblici che privati (per aziende e organizzazioni).

La CLI di NPM utilizza il file `package.json` contenente il nome e la versione del progetto o libreria, le dipendenze da altri moduli, eventuali comandi per eseguire testing e altro. Le dipendenze recuperate mediante NPM e salvate nel file `package.json` possono essere richiamate dal codice JavaScript mediante la funzione `require()`, la stessa che viene adoperata per importare moduli JavaScript locali.

NPM, mediante l'opzione `-g` (global), consente inoltre l'installazione di comandi globali a supporto dello sviluppo o della build, come ad esempio i linter, i framework di testing e code coverage, i transpiler (come TypeScript, CoffeeScript) ed altri stumenti.

A.3 JSDOM

JSDOM è un'implementazione in JavaScript puro di diversi standard web, in particolare DOM e HTML, per l'uso in NodeJS. In generale, l'obiettivo del progetto è l'emulazione di buona parte delle funzionalità dei browser, con lo scopo di avere un ambiente che consenta di testare le applicazioni web ed effettuarne lo "scraping"[2] delle pagine.

Per simulare il funzionamento di un browser mediante JSDOM, è sufficiente crearne un'istanza passando il contenuto della pagina HTML al costruttore, ad esempio:

```
1 const dom = new JSDOM("<html> ... content here ... </html>");
```

L'oggetto `dom` così ottenuto ha le stesse proprietà dell'oggetto globale del browser, ovvero è possibile ottenere il nodo radice del documento mediante la proprietà `dom.window.document`. A questo punto si può effettuare ricorsivamente la navigazione dell'albero del DOM e manipolare gli attributi e gli elementi di interesse. Utilizzando il metodo `dom.serialize()` è possibile infine riottenere la versione testuale della pagina HTML contenente le modifiche apportate.

Purtroppo, l'utilizzo di JSDOM è limitato ai file HTML "puri" ovvero non contenenti frammenti di script eseguiti lato server. Per fare un esempio, il seguente frammento di template non è correttamente processabile mediante JSDOM, a causa dell'innestamento delle doppie virgolette:

```
1 <a href="{{ row.links("product") }}">
```

Per poter superare tale limite, è necessario utilizzare uno strumento che sia in grado di contemplare entrambi le sintassi e distinguere il contesto di ciascuna.

[2]Con il termine di web scraping si indicano diverse metodologie che consentono di estrarre e collezionare dati e informazioni da Internet. Generalmente, questa azione è compiuta attraverso software (bot) che simulano la navigazione nel web compiuta dagli utenti, andando a prelevare determinate informazioni dagli ipertesti disponibili sui siti internet.

A.4 TextMate Grammar Parser

TextMate è un editor di testo visuale open source per MacOS. Tra le diverse possibilità di personalizzazione è di particolare rilievo l'estensibilità del motore di evidenziazione della sintassi (syntax highlighting), per i diversi linguaggi di programmazione e di markup, mediante l'aggiunta di grammar files che contengono le definizioni per ciascun linguaggio. Oltre a TextMate, anche altri editor come Eclipse, Sublime, Atom e VisualStudio Code hanno nel tempo adottato il formato TextMate per i grammar files, consentendo il riuso di definizioni già disponibili per altri ambienti mediante semplici operazioni di importazione.

In generale, la syntax highlighting è il processo di stilizzazione grafica del testo contenuto in un documento, in base a determinate regole. L'approccio generalmente seguito consiste nel suddividere il testo in regioni e successivamente associare uno stile a ciascuna di esse. I grammar file di TextMate definiscono appunto una serie di regole che sfruttano le espressioni regolari per suddividere il testo, le quali vengono valutate una riga per volta. Oltre all'utilizzo negli editor per motivi di stilizzazione, i grammar files sono utili anche per effettuare il parsing di documenti al fine di estrarre ed elaborare il testo come sequenza di token che rappresentano gli elementi di interesse del linguaggio.

Per poter implementare in NodeJS un parser che sfrutti i grammar files di TextMate, è stata utilizzata la libreria FreeMate disponibile sul repository NPM. L'unica differenza sostanziale rispetto al formato TextMate (`tmLanguage`) è l'utilizzo di file CSON[3] per rappresentare le grammatiche; ciò non costituisce un problema, trattandosi di una semplice conversione di formato.

Esempio di funzionamento

Nel listato A.4.1 è riportato un frammento di template Twig & HTML di esempio, utilizzato per mostrare il funzionamento del parser FreeMate per NodeJS. Si noti in particolare l'innestamento di due stringhe alla riga 1: il valore dell'attributo `class`, delimitato dai doppi apici,

[3] CSON è l'acronimo di CoffeeScript Object Notation.

al suo interno contiene un'espressione Twig in cui è presente un'altra stringa, anch'essa delimitata da doppi apici; questo è proprio il caso "patologico" citato al termine della sezione A.3. Effettuando la lettura del documento mediante JSDOM o un qualunque altro parser per HTML, si produrrebbe un errore in corrispondenza del simbolo `btn-group`, dato che verrebbe interpretato come un attributo non avente però il carattere di uguale "=" a seguire.

```
<div class="{{ div_style|default("btn-group") }}">
    {% if default_item.icon %}
    <i class="material-icons">{{ default_item.icon }}</i>
    {% endif %}
</div>
```

Listato A.4.1: Esempio di template Twig e HTML

Per elaborare il documento con FreeMate, si effettua il caricamento del grammar file di Twig, si ottiene poi l'oggetto `grammar` che effettua l'operazione di *tokenization* del testo, come riportato di seguito:

```
const registry = new firstMate.GrammarRegistry(); // creazione del registro
await loadGrammarFiles(registry, grammarFiles); // caricamento grammatica
const grammar = registry.getGrammars().find(g => g.name === 'Twig');
const document = await readFile(documentFileName); // caricamento template
const lines = grammar.tokenizeLines(document); // tokenization
```

A questo punto, il documento è stata suddiviso in token e ad ognuno di essi è stato assegnato uno scope. Come riportato nel listato A.4.2, gli scope risultano essere organizzati in gerarchia per la corretta interpretazione del contesto di ciascun token. In particolare, è stata superata la condizione di ambiguità dovuta alle stringhe innestate, dato che il livello di attributo HTML (identificato dallo scope `entity.other.attribute-name.html`) è superiore nella gerarchia rispetto al livello di valore stringa Twig (identificato dallo scope `string.quoted.double.twig`). Dovendo processare unicamente i token relativi al markup HTML, è sufficiente ignorare tutti gli altri aventi scope Twig e procedere al trattamento dei soli tag e attributi HTML. Lo svantaggio nell'utilizzare direttamente i token risiede nel livello di astrazione inferiore rispetto al DOM.

```
text.html.twig
    meta.tag.other.html
        punctuation.definition.tag.begin.html '<'
        entity.name.tag.other.html 'div'
```

```
        meta.attribute-with-value.html
            entity.other.attribute-name.html 'class'
            punctuation.separator.key-value.html '='
                punctuation.definition.string.begin.html '\"'
                    punctuation.section.tag.twig '{{'
                    variable.other.twig 'div_style'
                    keyword.operator.other.twig '|'
                    support.function.twig 'default'
                    punctuation.definition.parameters.begin.twig '('
                        string.quoted.double.twig
                            punctuation.definition.string.begin.twig '\"'
                        string.quoted.double.twig 'btn-group'
                            punctuation.definition.string.end.twig '\"'
                    punctuation.definition.parameters.end.twig ')'
                    punctuation.section.tag.twig '}}'
                punctuation.definition.string.end.html '\"'
        punctuation.definition.tag.end.html '>'
    meta.tag.template.value.twig
        punctuation.section.tag.twig '{%'
        keyword.control.twig 'if'
        variable.other.twig 'default_item'
        punctuation.separator.property.twig '.'
        variable.other.property.twig 'icon'
        punctuation.section.tag.twig '%}'
    meta.tag.other.html
        punctuation.definition.tag.begin.html '<'
        entity.name.tag.other.html 'i'
        meta.attribute-with-value.html
            entity.other.attribute-name.html 'class'
            punctuation.separator.key-value.html '='
                punctuation.definition.string.begin.html '\"'
            string.quoted.double.html 'material-icons'
                punctuation.definition.string.end.html '\"'
        punctuation.definition.tag.end.html '>'
    meta.tag.template.value.twig
        punctuation.section.tag.twig '{{'
        variable.other.twig 'default_item'
        punctuation.separator.property.twig '.'
        variable.other.property.twig 'icon'
        punctuation.section.tag.twig '}}'
        punctuation.definition.tag.begin.html '</'
        entity.name.tag.other.html 'i'
        punctuation.definition.tag.end.html '>'
    meta.tag.template.value.twig
        punctuation.section.tag.twig '{%'
        keyword.control.twig 'endif'
        punctuation.section.tag.twig '%}'
    meta.tag.other.html
        punctuation.definition.tag.begin.html '</'
        entity.name.tag.other.html 'div'
        punctuation.definition.tag.end.html '>'
```

Listato A.4.2: TextMate Scope Tree

Bibliografia

[1] Shauvik Roy Choudhary, Dan Zhao, Husayn Versee, and Alessandro Orso. Water: Web application test repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, ETSE '11, pages 24–29, New York, NY, USA, 2011. ACM.

[2] Brett Daniel, Qingzhou Luo, Mehdi Mirzaaghaei, Danny Dig, Darko Marinov, and Mauro Pezzè. Automated gui refactoring and test script repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, ETSE '11, pages 38–41, New York, NY, USA, 2011. ACM.

[3] Danny Dig and Ralph Johnson. The role of refactorings in api evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Mark Fewster and Dorothy Graham. *Software Test Automation: Effective Use of Test Execution Tools*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[5] Zebao Gao, Zhenyu Chen, Yunxiao Zou, and Atif M. Memon. Sitar: Gui test script repair. *IEEE Trans. Softw. Eng.*, 42(2):170–186, February 2016.

[6] Mark Grechanik, Qing Xie, and Chen Fu. Maintaining and evolving gui-directed test scripts. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 408–418, Washington, DC, USA, 2009. IEEE Computer Society.

[7] M. Hammoudi, G. Rothermel, and P. Tonella. Why do record/replay tests of web applications break? In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, volume 00, pages 180–190, April 2016.

[8] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. Waterfall: An incremental approach for repairing record-replay tests of web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 751–762, New York, NY, USA, 2016. ACM.

[9] Si Huang, Myra B. Cohen, and Atif M. Memon. Repairing gui test suites using a genetic algorithm. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 245–254, Washington, DC, USA, 2010. IEEE Computer Society.

[10] Jason Hunter. The problems with jsp. 2000.

[11] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Reducing web test cases aging by means of robust xpath locators. In *Proceedings of the 2014 IEEE International Symposium on Software Reliability Engineering Workshops*, ISSREW '14, pages 449–454, Washington, DC, USA, 2014. IEEE Computer Society.

[12] A. Memon, I. Banerjee, and A. Nagarajan. What test oracle should i use for effective gui testing? In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 164–173, Oct 2003.

[13] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pages 260–, Washington, DC, USA, 2003. IEEE Computer Society.

[14] Atif M. Memon. An event-flow model of gui-based applications for testing: Research articles. *Softw. Test. Verif. Reliab.*, 17(3):137–157, September 2007.

[15] Atif M. Memon. Automatically repairing event sequence-based gui test suites for regression testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2):4:1–4:36, November 2008.

[16] Brad A. Myers. User interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 2(1):64–103, March 1995.

[17] Terence John Parr. Enforcing strict model-view separation in template engines. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 224–233, New York, NY, USA, 2004. ACM.

[18] Michiaki Tatsubori and Toyotaro Suzumura. Html templates that fly: A template engine approach to automated offloading from server to client. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 951–960, New York, NY, USA, 2009. ACM.

[19] A. Turing. The early british computer conferences. chapter Checking a Large Routine, pages 70–72. MIT Press, Cambridge, MA, USA, 1989.

[20] Qing Xie and Atif M. Memon. Model-based testing of community-driven open-source gui applications. In *Proceedings of the 22Nd IEEE International Conference on Software Maintenance*, ICSM '06, pages 145–154, Washington, DC, USA, 2006. IEEE Computer Society.

[21] Qing Xie and Atif M Memon. Using a pilot study to derive a gui model for automated testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2):7:1–7:35, November 2008.

[22] Xun Yuan and Atif M. Memon. Using gui run-time state as feedback to generate test cases. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 396–405, Washington, DC, USA, 2007. IEEE Computer Society.