

**POLITECNICO DI TORINO**  
**LAUREA MAGISTRALE in INGEGNERIA**  
**INFORMATICA**



**Politecnico  
di Torino**

**TESI di Laurea Magistrale**

**Connected wallbox: una soluzione di test  
automation completa, dal design  
all'implementazione**

**Supervisors**

**Prof. RICCARDO COPPOLA**

**Dott. ALESSANDRO TESSUTI**

**Candidate**

**ALESSIO MAGURNO**

**APRILE 2023**



# Abstract

Un test end-to-end può essere considerato un tipo di test funzionale in quanto mira a verificare che l'intera applicazione o il sistema funzioni correttamente, rispettando gli obiettivi funzionali specificati. Un test funzionale, in generale, è un tipo di test che viene utilizzato per verificare che una determinata funzionalità dell'applicazione o del sistema rispetti le specifiche funzionali. Un test end-to-end può essere visto come un'ulteriore estensione di questo tipo di test, in quanto non solo si verifica la funzionalità di una specifica funzione o componente, ma anche la sua integrazione e il corretto funzionamento dell'intero sistema. In molti casi, i test E2E vengono eseguiti manualmente, il che può essere un compito noioso e soggetto ad errori. L'automazione risolve in grossa parte questi problemi garantendo una copertura di test massima e riducendo la possibilità di errore. Questa tesi di laurea presenta un'esperienza condotta presso una società di consulenza italiana durante uno stage. Lo scopo dello stage era lo sviluppo di una soluzione di automazione dei test per un'azienda che si occupa di transizione alla mobilità elettrica. In particolare, il sistema in fase di test consiste in una wallbox connessa che consente agli utenti di ricaricare un'auto elettrica a casa e offre la possibilità di monitorare le sessioni di ricarica e il consumo di energia dai propri endpoint quali mobile app e sito web. Una wallbox è un dispositivo per la ricarica domestica in abitazioni private o condominiali, sviluppato in forma compatta e adatto da fissare al muro e permette di avere maggiore capacità di ricarica. La soluzione richiesta include una serie di test E2E che coprono i maggiori casi di utilizzo del sistema da parte di utente. I test sviluppati, quindi, devono essere in grado di automatizzare l'uso dei tre endpoint dell'intero sistema che sono: il sito web, la mobile app e la wallbox. Inoltre, il lavoro di tesi si è concluso con un'attività di ricerca sulle nuove tendenze dell'automazione dei test. In particolare, lo scopo era quello di capire come l'uso di intelligenza artificiale e apprendimento automatico

nell'automazione dei test potessero migliorare i test sviluppati. Gli approfondimenti svolti hanno riguardato sia tecniche di riconoscimento di immagini che tecniche di test associate alle intelligenze artificiali conversazionali come ChatGPT. La prima fase dello sviluppo è la parte di test design che consiste nella progettazione dei casi di test che coprono tutte le funzionalità da testare. Per iniziare, si svolgono i flussi richiesti manualmente per definire le operazioni da eseguire per completare i test. Inoltre, nella fase di progettazione, si è dovuta progettare una soluzione ad-hoc per automatizzare le operazioni da eseguire sulla wallbox. In particolare è stato fornito un simulatore di una automobile da ricaricare, che permette di simulare i tre stati in cui può trovarsi la wallbox: scollegata, pronta alla ricarica e in ricarica. Per cambiare tra i seguenti stati il simulatore presentava un potenziometro; l'obiettivo quindi è quello di utilizzare quel potenziometro tramite gli script di test in modo da automatizzare tutto il flusso di ricarica. Dopo aver completato la fase di progettazione, è iniziato lo sviluppo degli script di test. Gli script di test sono stati sviluppati nel framework proprietario dell'azienda. Questo framework si basa sull'uso di Selenium per i test delle pagine web e Appium per il test dell'applicazione mobile, tutti e due usati con il linguaggio Java. Inoltre, viene utilizzato Cucumber, che consente di descrivere come dovrebbe comportarsi il software in un testo strutturato nel semplice linguaggio Gherkin (Given - When - Then). L'attività di ricerca, che è stata svolta in parallelo allo sviluppo, propone un possibile miglioramento al framework attualmente in uso nell'azienda. Lo scopo principale, era quello di verificare se fosse possibile aggiungere l'uso di intelligenza artificiale e machine learning nei test da sviluppare per rendere i test ancora più consistenti. Nella prima parte è stata analizzata un'opzione open source che è facilmente integrabile con il framework Selenium-based, e questa analisi ha portato buoni risultati in termini di miglioramento nelle soluzioni di sviluppo e di test. Infatti, con questa soluzione, gli script di test risultano molto meno influenzati dai rapidi cambiamenti nel software in fase di test. Nella seconda parte dell'attività di ricerca, l'attenzione si è concentrata sullo sviluppo di una soluzione che integra tecniche di computer vision negli script di test automatizzati. Le tecniche di computer vision servono a valutare

l'output visibile di un'applicazione e lo confronta con i risultati attesi dal design. In questa fase sono state analizzate due soluzioni, la prima delle quali è basata su un plugin Appium che consente di effettuare il matching tra due immagini, ma dopo l'analisi effettuata quest soluzione risulta non particolarmente coerente; la seconda soluzione, invece, è basata sulla libreria OpenCV in Python e consente anche di effettuare il matching tra template, ma con risultati migliori rispetto alla soluzione precedente. Nell'ultima parte si è tentato di capire come l'utilizzo di intelligenze artificiali conversazionali, quali ChatGPT, cambia il modo con cui approcciarsi al test automation. A valle di questo lavoro di tesi si può considerare come il test automation sia destinato a cambiare il mondo del software testing, che è una parte fondamentale del ciclo di vita di un software. Allo stesso tempo si può notare come sia una tecnica di test che necessita ancora di un forte sviluppo per seguire i rapidi miglioramenti e sviluppi del software odierno. Partendo da questo lavoro e seguendo il trend di questo periodo storico , si potrebbe avere il focus sul migliorare e capire come stabilizzare le tecniche di test che usano l'artificial intelligence e il machine learning che rendono i test ancora più consistenti.



# Table of Contents

<b>List of Tables</b>	VIII
<b>List of Figures</b>	IX
<b>Acronyms</b>	XII
<b>1 Introduzione</b>	1
1.1 L'Azienda . . . . .	1
1.2 Test automation . . . . .	3
<b>2 Tecnologie Utilizzate</b>	10
2.1 Selenium . . . . .	12
2.2 Appium . . . . .	15
2.3 Cucumber . . . . .	19
<b>3 Sviluppo Progetto</b>	22
3.1 Sistema da testare . . . . .	22
3.2 Analisi As-is e to-Be . . . . .	24
3.2.1 AS IS . . . . .	26
3.2.2 TO BE . . . . .	26
3.3 Implementazione della soluzione di test . . . . .	28
3.3.1 Test Design . . . . .	28
3.3.2 Sviluppo test . . . . .	33
3.4 Deployment della soluzione . . . . .	41
<b>4 Attività di ricerca: Nuovi trend</b>	44
4.1 Artificial Intelligence Machine Learning Deep Learning	45

4.2	Intelligenza artificiale nell'automazione dei test . . . . .	48
4.3	Un nuovo approccio al testing: Self-Healing . . . . .	49
4.3.1	Healenium . . . . .	49
4.3.2	Esempio di utilizzo di Healenium . . . . .	51
4.3.3	Risultati Ottenuti: Self-Healing . . . . .	57
4.4	Visual testing . . . . .	59
4.4.1	OpenCv integrato in Appium . . . . .	59
4.4.2	OpenCv in Python e l'integrazione nel TAF . . . . .	67
4.5	Intelligenza artificiale conversazionale nel Test Automation	71
<b>5</b>	<b>Conclusioni e sviluppi futuri</b>	<b>77</b>
	<b>Bibliography</b>	<b>79</b>



# List of Tables

4.1	Differenze tra AI ML e DL . . . . .	47
-----	-------------------------------------	----

# List of Figures

1.1	Processo Software Testing . . . . .	6
1.2	Quadrante dello sviluppo test . . . . .	7
1.3	Grafico che mostra l'effort per numero di test preso da studio [9] . . . . .	8
1.4	Grafico che mostra la coverage per cambiamenti nei test case preso da studio [9] . . . . .	9
2.1	Localizzatori utilizzati da Selenium . . . . .	14
2.2	Schermata iniziale Appium Inspector . . . . .	17
2.3	Schermata Appium Inspector per trovare i localizzatori	18
2.4	Xpath suggerito automaticamente da Appium Inspector	18
3.1	Diagramma di contesto . . . . .	23
3.2	AS IS TO BE . . . . .	25
3.3	Simulatore di macchina elettrica con evidenziata la manopola che simula gli stati . . . . .	29
3.4	Caso di test fornito dall'azienda . . . . .	31
3.5	Caso di test dopo il test design . . . . .	31
3.6	Base per poggiare il telefono per associare la wallbox .	33
3.7	Base per poggiare il telefono per associare la wallbox .	34
3.8	Lavagna di avanzamento task . . . . .	34
3.9	Elemento HTML visionato da strumento "analizza" del browser . . . . .	35
3.10	Connettore stampato in 3D per fissare il servo di arduino al simulatore . . . . .	40
3.11	Setup completo per svolgere un test . . . . .	41

3.12	Deployment Diagram . . . . .	42
4.1	Descrizione di AI ML DL . . . . .	46
4.2	Diagramma a blocchi che sintetizza il funzionamento di Healenium . . . . .	51
4.3	Algoritmo confronto alberi . . . . .	52
4.4	Contenuto file Healenium.properties . . . . .	53
4.5	Modifica a runtime del locatore . . . . .	55
4.6	Log presi da un tentativo di healing . . . . .	55
4.7	Report Healenium . . . . .	56
4.8	Screenshot associato a report healenium . . . . .	56
4.9	Template per effettuare template matching . . . . .	64
4.10	Immagine da ricercare nello screenshot . . . . .	65
4.11	Tabella per riassumere i risultati dei test . . . . .	66
4.12	Tabella per riassumere i risultati dei test nel secondo caso	70
4.13	Domanda a ChatGPT sul test automation . . . . .	73
4.14	Domanda a ChatGPT sul test design . . . . .	74
4.15	Domanda a ChatGPT sullo sviluppo di codice per test automation . . . . .	75



# Acronyms

**AI**

Artificial intelligence

**ML**

Machine learning

**DL**

Deep learning

**DOM**

Document object model

**CI**

Continuous integration

**UI**

User interface

**OCR**

Optical character recognition

**LCS**

Longest common subsequence

**AUT**

Application under test

# Chapter 1

## Introduzione

### 1.1 L'Azienda

Reply nasce nel 1996 a Torino. Inizialmente il nome era Riplai, il cui significato era “Rizzante, Ignegnatti Pepino Lavorano Ancora Insieme”. I nomi ovviamente erano quelli dei “fondatori” di cui oggi restano Mario Rizzante come presidente e Oscar Pepino come co-ceo insieme a Tatiana Rizzante, figlia di Mario. Dal 1996 ad oggi, Reply ha conquistato il mercato nazionale e non solo; basti pensare che, da quando, nel 2006, la guida è passata a Tatiana Rizzante, l'azienda si è espansa in tutta Europa ed il fatturato è passato dai 33,3 milioni di euro nel 2000 a 1,48 miliardi di euro nel 2021 [1]. Reply, oltre ad essere la prima azienda ICT italiana a diventare un'azienda unicorno, [2] è anche una società quotata in borsa e le sue azioni sono cresciute anno dopo anno passando da un valore di circa 3 euro nel 2000 a 115 euro al 24 novembre 2022.

L'idea di Mario Rizzante è sempre stata quella di concentrarsi sulle aziende che puntavano ad affermarsi sul mercato tramite innovazione e nuovi modelli di business tramite i servizi digitali, l'IoT (Internet Of Things) e il cloud.

Reply nel gennaio del 2009 annuncia l'acquisizione del centro ricerche situato a Torino di Motorola, che aveva annunciato solo qualche mese prima la chiusura del polo di ricerca con conseguente licenziamento di oltre 300 dipendenti .

Reply oltre all'acquisizione, ha provveduto anche ad assumere 180

delle oltre 300 persone licenziate da Motorola. Da questa acquisizione nasce Reply Concept, Business Unit di Reply. Concept nasce come società specializzata nell'IoT (Internet Of Things), con l'obiettivo, quindi, di servire tutti quei clienti che hanno la necessità di sviluppare o testare dispositivi connessi. Con il passare degli anni, Concept ha sviluppato un "*Global Test Automation Centre*" capace di testare in maniera automatica 24 ore su 24 e 7 giorni su 7 gli applicativi o i dispositivi dei clienti, spaziando dai siti web, a centraline di veicoli, dai modem/router delle aziende di telecomunicazioni alle applicazioni sviluppate per smartphone. Sia durante la fase di sviluppo che dopo il lancio commerciale, essa permette ai clienti di tenere sotto controllo la qualità di prodotti e servizi. Concept ad oggi conta su un numero di risorse umane che supera le 300 persone, di cui circa il 35 per cento sono collaboratori esterni. Gli standard di selezione per diventare un dipendente Reply sono molto alti: occorre essere laureati con ottimi voti, si parla di voti sopra al 102 (Solo per alcune facoltà e università rinomate si scende al 98 come voto minimo) in una facoltà tra quelle incluse in una lista (che varia a seconda delle società facenti parti dell'intero gruppo Reply). Concept, in particolare, è una società che cerca di non far vedere la differenza tra dipendenti interni ed esterni e cerca di includere tutti nelle attività di team building e di formazione. Concept, quindi, è una società molto attenta al dipendente e cerca di spronare tutti a dare il meglio di sé stessi, offrendo diverse opportunità per migliorare e per imparare cose nuove ogni giorno. Questa è una cosa molto importante visto che uno degli obiettivi della società è quello di essere sempre all'avanguardia sulle tematiche IoT e il miglior modo, ovviamente è quello di offrire ai propri dipendenti la possibilità di frequentare corsi di aggiornamento. Concept inoltre dà ai propri dipendenti la possibilità di tenere dei webinar per ottenere delle "berries" utilizzabili poi per poter partecipare a conferenze o seguire particolari corsi non inclusi in quelli fruibili gratuitamente da tutti i dipendenti. Questi webinar consistono nel condividere, con chi vuole assistere, un progetto o un argomento approfondito dalla persona o dal gruppo che tiene il webinar. Alla fine, tutti i partecipanti potranno votare l'esposizione e la tematica affrontata. A seconda poi delle votazioni,

chi ha tenuto il webinar otterrà più o meno berries. La holding e di conseguenza tutte le società tendono ad essere molto dinamiche e quando una singola società raggiunge un numero abbastanza grande di persone si tende a dividere la società cercando di settorializzare il più possibile. In questo modo si cerca di avere tante società, ognuna specializzata in un determinato campo, ogni società poi può avere al suo interno delle Business Unit che possono essere viste come “mini società”. Ad esempio, Concept da poco tempo si è divisa in due società; Concept Quality (La BU in cui ho svolto questo lavoro di tesi) che si occupa principalmente di attività di test e validazione su vari sistemi e Concept Engineering business unit che si occupa della progettazione e dello sviluppo di sistemi hardware o software, come possono essere applicativi mobile o desktop, siti web, droni, robot e tanto altro. Concept Quality poi a sua volta è appunto divisa in più BU: la BU di automotive, che si occupa di temi inerenti all’automotive e dove vengono svolte attività di validazione sui sistemi di infotainment, sulle centraline telematiche, applicazioni mobile, e siti web sia per quanto riguarda la parte di front end e sia per quanto riguarda la parte di backend, attività che vengono svolte sui veicoli di varie case automobilistiche; la BU relativa al mondo Telco che si occupa di gestire i clienti del mondo delle telecomunicazioni, svolgendo attività di validazione su linee internet, telefoniche e apparati hardware come telefoni e modem/router.

Questa divisione di società e BU permette di facilitare l’individuazione di mission e vision per ogni società in quanto settorializzandosi, la mission viene quasi definita in modo autonomo[3].

## 1.2 Test automation

Il Software che non funziona correttamente può causare molti problemi, tra cui la perdita di denaro, di tempo, di reputazione aziendale e persino infortuni o morte [4]. Questa frase è presente nel capitolo introduttivo del *Syllabus “Foundation Level”* che è il libro di preparazione per una delle più importanti certificazioni di tester rilasciata da **ISTQB**. Questa frase in maniera implicita spiega al mondo d’oggi l’importanza del testing. Il testing del software è un modo per valutare la qualità



del software e per ridurre il rischio di failure nel software in produzione. Questa può essere una prima definizione di testing, questa però forse non dà la giusta importanza al ruolo che il testing ha nel ciclo di vita di un'architettura. Il test software, infatti, non sta solo nell'esecuzione dei test ma è un vero e proprio processo che è formato da diverse attività come la pianificazione, il design, l'analisi, la progettazione e l'implementazione dei test ed infine il report dei risultati con la conseguente valutazione della qualità del sistema da testare. Per tutto questo si fornisce una stima che va dal 30 al 60 per cento sull'intero ciclo di vita del software impiegato nel testing. Per ogni progetto gli obiettivi del testing possono includere:

- Prevenire difetti attraverso la valutazione di prodotti di lavoro come requisiti, user story, progettazione e codice.
- Verificare se tutti i requisiti specificati sono stati soddisfatti.
- Verificare se l'oggetto di test è completo e validare se funziona come gli utenti e altri stakeholder si aspettano.
- Aumentare la confidenza nel livello di qualità dell'oggetto di test.
- Trovare failure e difetti, in modo da ridurre il livello di rischio di una qualità del software inadeguata.
- Fornire informazioni sufficienti agli stakeholder, per consentire loro di prendere decisioni informate, soprattutto per quanto riguarda il livello di qualità dell'oggetto di test.
- Rispettare requisiti o standard contrattuali, legali o normativi, e/o verificare la conformità dell'oggetto di test a tali requisiti o standard [4].

Nel campo del software testing si è inserita da anni una tendenza che è l'automazione dei test. L'automazione dei test è il processo di utilizzo di strumenti software per eseguire casi di test ripetitivi e predefiniti rispetto a un'applicazione o un sistema. L'obiettivo principale dell'automazione dei test è aumentare l'efficienza e l'efficacia del processo di test riducendo

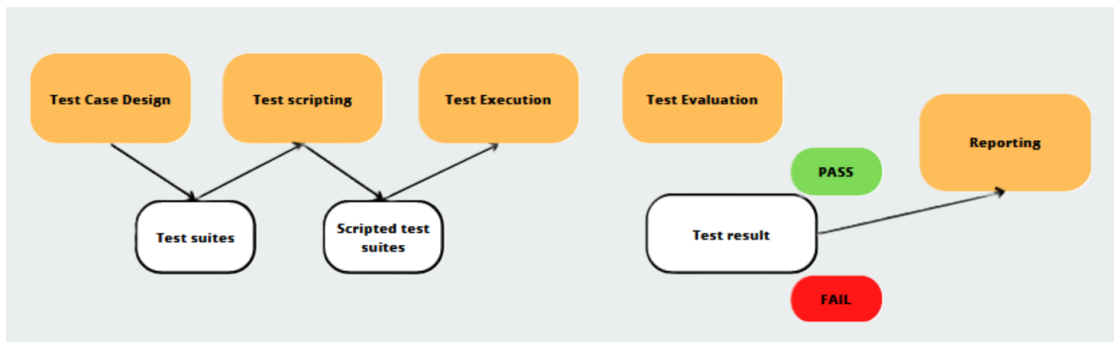
il coinvolgimento umano e riducendo al minimo il rischio di errore umano. Ci sono diversi vantaggi dell'automazione dei test, tra cui:

- **Test più rapidi:** l'automazione dei test consente un'esecuzione più rapida dei casi di test, il che può ridurre significativamente il ciclo di test e aiutare a identificare i problemi all'inizio del processo di sviluppo.
- **Maggiore *coverage*:** l'automazione dei test consente un grado più elevato di *coverage*, in quanto è possibile automatizzare attività ripetitive e dispendiose in termini di tempo che potrebbero non essere eseguibili manualmente [5].
- **Coerenza:** l'automazione dei test garantisce che gli stessi casi di test vengano eseguiti in modo coerente, con gli stessi input e output previsti, il che può aiutare a identificare e riprodurre i bug più facilmente.
- **Risparmi sui costi:** l'automazione dei test può portare a risparmi sui costi riducendo la quantità di test manuali richiesti e identificando e risolvendo i problemi all'inizio del processo di sviluppo.
- **Riutilizzabilità:** i casi di test automatizzati possono essere riutilizzati in diverse versioni di un'applicazione o di un sistema, il che può far risparmiare tempo e fatica a lungo termine.

Possiamo riassumere che il test automatizzato è più affidabile del testing manuale, è molto più veloce poiché eseguito da alcuni tool software ed è molto adatto per dei *test case* da eseguire ripetitivamente per un lungo periodo.

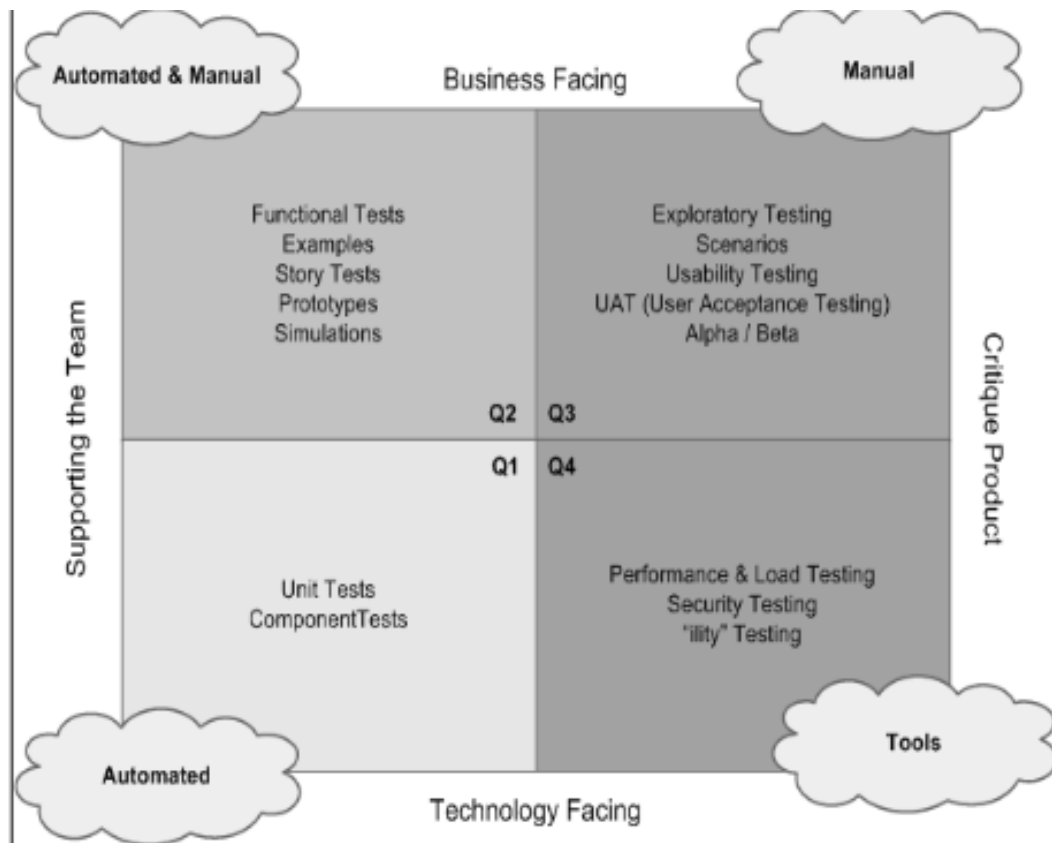
Per automazione dei test software si intende l'automatizzazione di tutto il software necessario a svolgere i test su uno o più applicativi partendo dall'esecuzione degli script di test fino alla verifica dei requisiti. Nella figura fig 1.1 viene riportato un breve schema dell'intero processo che riguarda il testing, partendo dal design fino al report dei risultati ottenuti. In questa catena di attività l'automazione entra in gioco in tutti i passi; le attività principali dove l'automazione è entrata a rivoluzionare le strategie di test sono sicuro quelle di *test execution*

e *test evaluation*. Per le prime attività che sono quelle di design e di scrittura vera e propria degli script di test, l'automazione è ancora un passo in dietro, in quanto la conoscenza umana nel generare i casi di test è ancora necessaria [6]. Tuttavia, l'automazione dei test non è adatta a tutti i tipi di test. Ad esempio, alcuni tipi di test, come i test di usabilità, sono più adatti al test manuale. Inoltre, l'automazione dei test richiede un investimento iniziale significativo in termini di tempo e risorse per configurare e mantenere l'infrastruttura di automazione dei test. Per spiegare meglio come i grandi vantaggi dei test automatizzati hanno cambiato molto nel mondo dello sviluppo dei test si può vedere la fig 1.2 presa da questo articolo [7].



**Figure 1.1:** Processo Software Testing

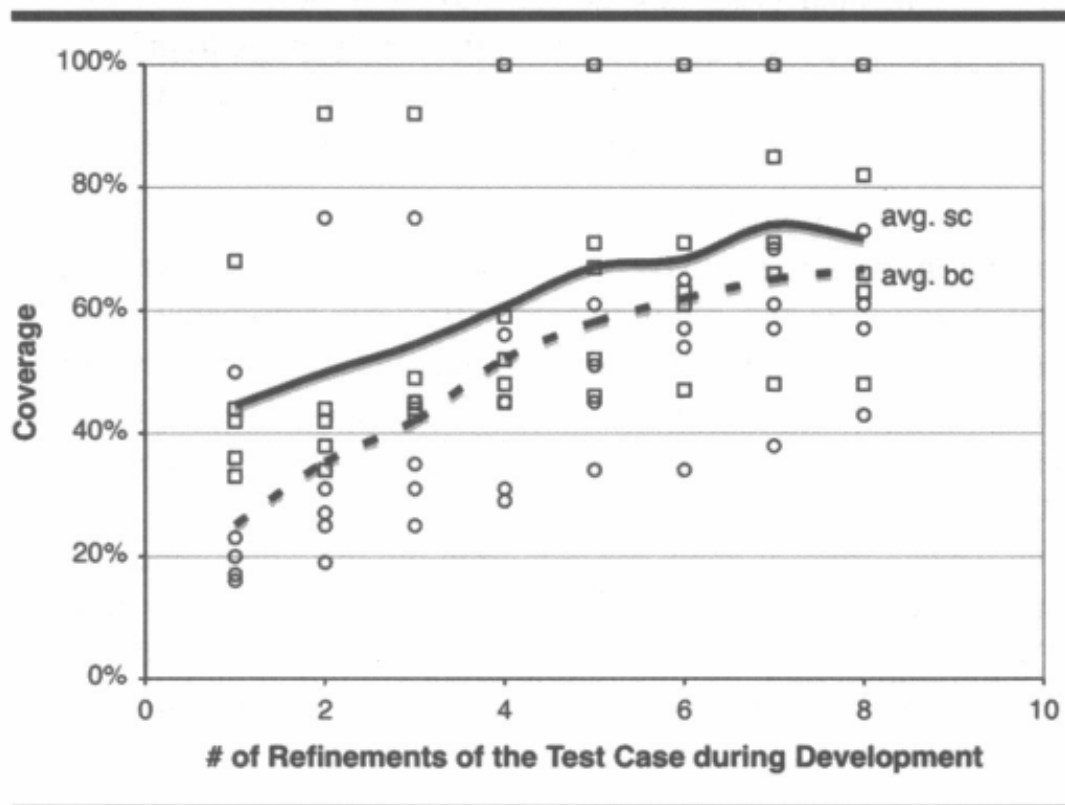
In [8], gli autori hanno condotto uno studio su un caso di test per una azienda. Prima dell'inserimento dei test automatizzati, tutta la lista dei test richiesti veniva portata a termine in 60 giorni riuscendo a svolgere in media un test al giorno, mentre con l'utilizzo dei test automatizzati l'intera lista dei test è stata sviluppata in 33 ore, quindi ben 16 volte più velocemente. Oltre al tempo sono stati analizzati altri aspetti; sullo stesso caso e con la stessa lista dei test non era stato trovato nessuno errore con i test manuali mentre sono stati trovati 3 bug con i test automatizzati. E' stato stimato inoltre il ROI. Il *return on investment* o ritorno sull'investimento o indice di redditività del capitale investito è un indice di bilancio che indica la redditività e l'efficienza economica della gestione caratteristica a prescindere dalle fonti utilizzate: esprime, cioè, quanto rende il capitale investito in quell'azienda [8]. Nel caso dei



**Figure 1.2:** Quadrante dello sviluppo test

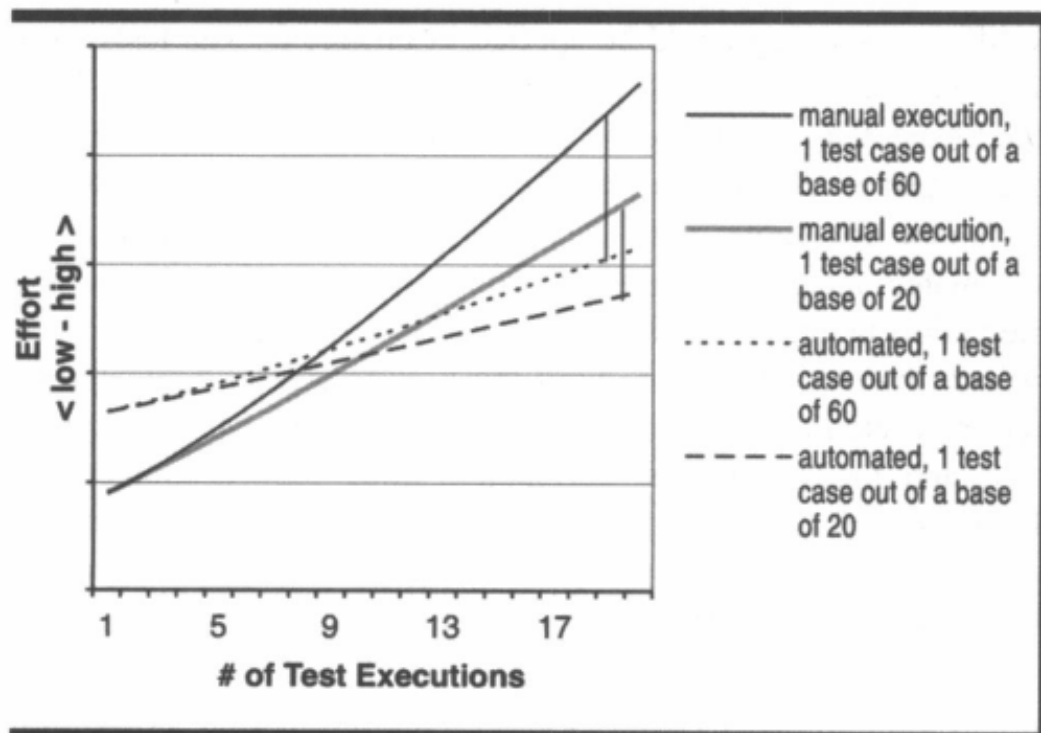
test manuali il costo è lineare con il crescere dei test mentre nel caso dei test automatizzati il costo rimane costante dopo la prima iterazione. Questa diminuzione è dovuta in primis al tempo di sviluppo dei test che fa sì che il ROI sia più alto. Questo è solo uno dei casi che ho trovato in letteratura per dimostrare anche a livello aziendale il miglioramento portato dai test automatizzati.

In figura fig 1.3 e fig 1.4 riporto due immagini prese da uno studio [9] che viene condotto riguardo la migrazione dello sviluppo dei test da manuali ad automatizzati. Da questo studio si possono evincere, anche a livello pratico, due dei maggiori vantaggi sopracitati del test automation: lo sforzo, che diminuisce al crescere del numero di esecuzioni di test e la coverage che invece, cresce in questo caso con il numero dei raffinamenti fatti sui test case durante lo sviluppo. Questo ramo del testing è il



**Figure 1.3:** Grafico che mostra l'effort per numero di test preso da studio [9]

ramo di cui si occupa nello specifico il team in cui ho svolto il mio lavoro di tesi ed è proprio il focus del mio progetto. Prima di entrare nello specifico a parlare del progetto volevo fare una breve introduzione delle tecnologie in uso in questo momento per quanto riguarda la *test automation* e nello specifico parlare dei tool in utilizzo nel framework aziendale che ho trovato.



**Figure 1.4:** Grafico che mostra la coverage per cambiamenti nei test case preso da studio [9]

## Chapter 2

# Tecnologie Utilizzate

Le applicazioni moderne presentano sempre di più una GUI complessa, che deve adattarsi a diversi device, diverse lingue e fare uso con l'avanzare del tempo sempre di più widget partendo da quelli nativi arrivando fino a quelli custom. Il modo con cui l'utente interagisce con queste è anch'esso in evoluzione, si parte da semplici click fino ad arrivare a *gesture* sempre più complicate. Questo avanzamento nella tecnologia e la velocità sempre più elevata di produzione di nuovo software rappresentano una vera e propria sfida per i sistemi di *test automation* sulla GUI. Ad aiutare gli sviluppatori di *test automation* ci sono alcuni tool che espongono dei metodi per controllare, da codice (approccio *full code*) o grazie a delle dashboard specializzate (approccio *low code*), tutti gli endpoint dei sistemi da testare.

In seguito sono descritti alcuni dei tool più popolari:

- **Selenium:** Selenium è uno dei tool di *test automation* più utilizzati e affidabili. È *open source* e supporta diversi linguaggi di programmazione, come Java, Python e Ruby. Il suo principale vantaggio è che è molto flessibile e può essere utilizzato per testare diverse piattaforme e browser. Tuttavia, può essere difficile da configurare e richiede una certa conoscenza tecnica per utilizzarlo al meglio.
- **Appium:** Appium è un altro tool di *test automation* popolare, specialmente per le applicazioni mobile. È *open source* e supporta sia i dispositivi Android che iOS. Ha una interfaccia *user-friendly* e

una buona documentazione, rendendolo facile da utilizzare per gli sviluppatori. Tuttavia, può essere più lento rispetto ad altri tool simili.

- **TestComplete:** TestComplete è un tool di *test automation* proprietario che offre un'interfaccia *user-friendly* e una vasta gamma di funzionalità avanzate, come la registrazione degli script e la creazione di report dettagliati. Tuttavia, è più costoso rispetto ad altri tool simili e richiede una licenza per l'utilizzo.

Ci sono diverse tipologie di tool per *test automation* che vanno dalla semplice registrazione e riproduzione di script, a soluzioni più complesse che offrono funzionalità avanzate come la gestione dei test, la generazione di report e la gestione degli errori. Ecco alcune delle tipologie più comuni di tool per *test automation*:

- ***Record and Play*** Registratori di script: questi strumenti registrano le azioni dell'utente e le trasformano in script che possono essere riprodotti in modo automatico. Questi tool sono molto semplici da usare, ma spesso non forniscono una copertura adeguata dei test e non sono molto flessibili.
- **Framework di automazione:** questi strumenti forniscono un'architettura solida per la creazione, l'esecuzione e la gestione dei test. Possono essere utilizzati per creare script complessi e offrono funzionalità avanzate come la gestione degli errori, la generazione di report e la gestione dei test.
- **Strumenti di test basati sul cloud:** questi strumenti utilizzano il cloud per eseguire i test su diversi sistemi operativi e dispositivi. Sono molto flessibili e possono essere utilizzati per testare software in ambienti complessi.
- **Strumenti di test di performance:** questi strumenti sono progettati per testare le prestazioni del software, come la velocità di risposta e la capacità di gestire grandi quantità di dati.



- **Strumenti di test mobile:** questi strumenti sono progettati per testare software su dispositivi mobili, come smartphone e tablet. Possono essere utilizzati per testare l'interfaccia utente, la compatibilità con diversi sistemi operativi e la funzionalità delle applicazioni mobili.

Nel framework di test in utilizzo nell'azienda, i tool utilizzati sono: Selenium per i test automatizzati di siti web e Appium per i test automatizzati su dispositivi portatili. Questi due tool vengono utilizzati nel linguaggio Java. Oltre a questo viene utilizzato Cucumber che, attraverso il linguaggio Gherkins, permette di nascondere l'implementazione delle funzioni mappando una scrittura più leggibile dall'uomo al codice che sta sotto.

## 2.1 Selenium

Automatizzare l'utilizzo di un sito web è forse la cosa più richiesta nello sviluppo di *test automation*. Il tool utilizzato, per riuscire a comandare attraverso codice Java il browser, è Selenium. Selenium è un tool open source per la gestione automatizzata dei browser, utilizzato come framework di testing. Selenium indica in realtà una suite, composta da diversi strumenti: Selenium IDE, Selenium Builder, Selenium Grid, Selenium WebDriver [10]. Quello utilizzato nel framework di test, che è quello utilizzato durante lo sviluppo del progetto, è Selenium WebDriver. Selenium WebDriver è un *object-oriented* API che gestisce in modo nativo un browser come farebbe un utente e supporta molti linguaggi di programmazione tra cui Java che è quello utilizzato nel framework in uso per il progetto di tesi. Selenium WebDriver mette a disposizione delle API per interagire con tutti gli elementi presenti nel DOM, performando azioni semplici come click oppure inserimento di testo e altre azioni più complicate come caricare dei file o fare il drag and drop di un elemento. Il funzionamento di Selenium WebDriver si basa sulla ricerca degli elementi nel DOM attraverso dei localizzatori per poi performare delle azioni sugli elementi proprio come farebbe un umano quando vuole interagire con un sito web. Per fare ciò esiste

un metodo chiamato `findElement` che ha come unico parametro il localizzatore del elemento e ritorna un oggetto di tipo `WebElement` che fa parte di Selenium e che ha tutti i metodi che permettono di interagire con esso sulla pagina web. Resta da spiegare a questo punto cosa sono i localizzatori. Come ben sappiamo le pagine web sono pagine HTML quindi ogni elemento visibile nella pagina ha un corrispondente elemento HTML con dei relativi attributi. I localizzatori utilizzano proprio gli attributi degli elementi HTML per identificare gli oggetti con cui interagire. Nella figura fig 2.1 si vedono i localizzatori messi a disposizione da Selenium. ***Id*** è quello migliore perché come norma di buona programmazione l'id di un elemento dovrebbe essere unico per tutti gli elementi del DOM. Quindi come buona norma durante lo sviluppo di *test automation* prevede di utilizzare, se disponibile, l'id come localizzatore. Gli altri localizzatori che si vedono in figura fanno riferimento tutti ad altri attributi HTML come:

- **ClassName**: utilizzo del valore dell'attributo `class` come localizzatore.
- **tagName**: utilizzo del nome del tag HTML come localizzatore.
- **linkText**: viene utilizzato per identificare i collegamenti ipertestuali su una pagina web.
- **partialLinkText**: accesso ai link utilizzando una parte del testo del collegamento.
- **name**: accede agli elementi utilizzando come localizzatore il valore dell'attributo `name`.

Tutti questi localizzatori vanno utilizzati sempre controllando prima che siano univoci nel DOM. Molte volte, infatti, questi attributi non si riferiscono ad un solo elemento. In questi casi ci sono gli ultimi due metodi presenti nella fig 2.1 che non sono stati ancora descritti; i due localizzatori sono *Xpath* e *cssSelector*. Non sempre possiamo utilizzare uno degli elementi messi a disposizione da Selenium per identificare un elemento nel DOM e per risolvere questi problemi esistono questi due selettori; i quali sono più articolati e permettono di utilizzare

più attributi per cercare un elemento. Questi due tipi di selettori permettono di cercare, ad esempio degli elementi con specifico tag e uno specifico attributo oppure degli elementi che si trovano all'interno di altri. Il più semplice è il `cssSelector` che come suggerisce il nome del metodo usa lo stesso modo che usa il linguaggio css per intercettare gli elementi nel DOM. Xpath, invece, è un percorso XML utilizzato per la navigazione attraverso la struttura HTML della pagina. È una sintassi o un linguaggio per trovare qualsiasi elemento su una pagina Web utilizzando l'espressione del percorso XML. XPath può essere utilizzato sia per documenti HTML che XML per trovare la posizione di qualsiasi elemento su una pagina Web utilizzando la struttura del DOM [11]. Quindi uno dei lavori più dispendiosi del programmatore di *test automation* sarà visitare l'HTML del sito web, con l'ausilio dello strumento ispeziona messo a disposizione da tutti i browser, e cercare i localizzatori migliori per intercettare l'elemento desiderato. Selenium è in grado di gestire un driver per i browser più conosciuti come Firefox e Chrome ed è molto semplice l'inizializzazione del driver che si effettua all'inizio di ogni test. Si può vedere nel frammento di codice 2.1 come si inizializza un driver Chrome con la possibilità di passare come parametro alcune opzioni come ad esempio l'apertura in incognito del driver.

```
20
21 driver.findElement(By.);
22
23 System.out.println
24 driver.quit();
25 }
26
27 public static void main
28 BaseTests test = ne
29 test.setUp();
30 }
31 }
32
```

By by

- `tagName(String tagName)`
- `xpath(String xpathExpression)`
- `className(String className)`
- `class`
- `id(String id)`
- `linkText(String linkText)`
- `cssSelector(String cssSelector)`
- `name(String name)`
- `partialLinkText(String partialLinkText)`
- `ByClassName (org.openqa.selenium.By)`
- `ByCssSelector (org.openqa.selenium.By)`
- `ById (org.openqa.selenium.By)`
- `ByLinkText (org.openqa.selenium.By)`
- `ByName (org.openqa.selenium.By)`
- `ByPartialLinkText (org.openqa.selenium.By)`
- `ByTagName (org.openqa.selenium.By)`
- `ByXPath (org.openqa.selenium.By)`

Press ⌘ to choose the selected (or first) suggestion and insert a dot afterwards

**Figure 2.1:** Localizzatori utilizzati da Selenium

**Listing 2.1:** Inizializzazione driver Selenium

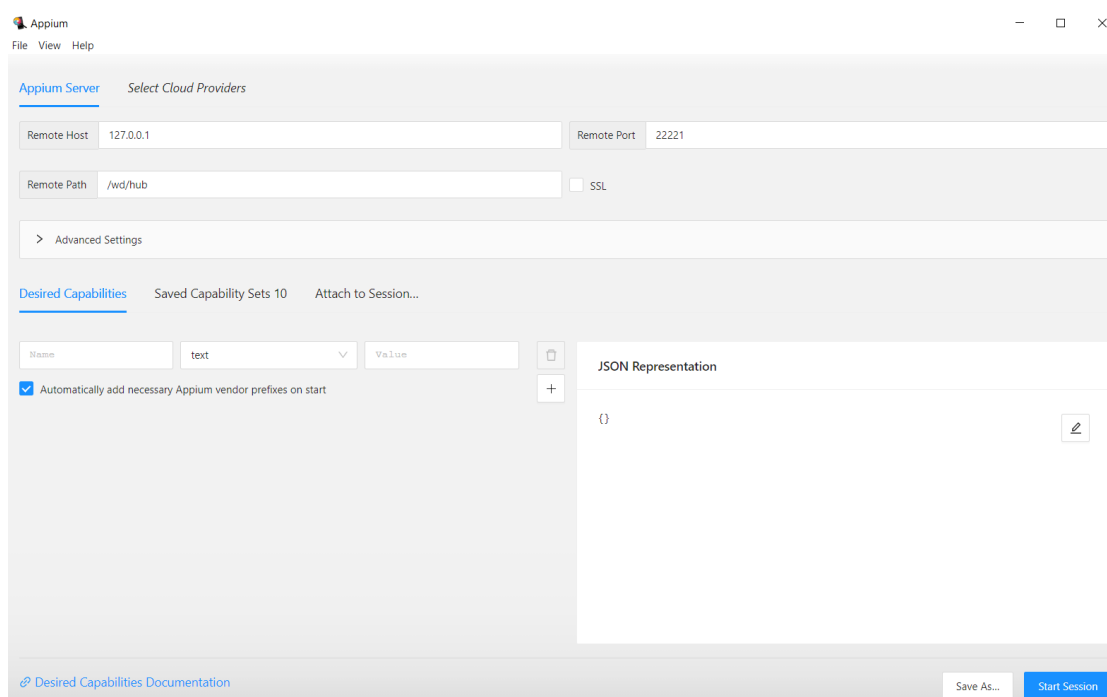
```
1 ChromeDriver driver;  
2     if (incognito) {  
3         options = new ChromeOptions();  
4         options.addArguments("--incognito");  
5     }  
6     driver = new ChromeDriver(options);  
7     wait = new WebDriverWait(driver, 120);  
8     return driver;
```

## 2.2 Appium

Alla pari dell'automazione dell'uso della GUI dei siti web, anche l'automazione dell'utilizzo di dispositivi mobili è ampiamente richiesta nelle soluzioni di *test automation*. Appium è un tool di automazione dei test *cross-platform*, *open source* per app native, ibride, Web mobile e desktop. Appium è fondamentalmente un server web che espone un'API REST. Un'API REST è un'API conforme ai principi di progettazione dello stile architetturale REST o representational state transfer. Per questo motivo, le API REST sono a volte indicate come API RESTful [12]. Riceve connessioni da un client, ascolta i comandi, li esegue su un dispositivo mobile e risponde con una risposta HTTP che rappresenta il risultato dell'esecuzione del comando. Il fatto di avere un'architettura client/server apre molte possibilità: possiamo scrivere il nostro codice di test in qualsiasi linguaggio che abbia un'API client HTTP, anche se rimane più semplice utilizzare una delle librerie client Appium. Possiamo mettere il server su una macchina diversa da quella su cui sono in esecuzione i nostri test. Possiamo scrivere codice di test e affidarci ai servizi cloud per ricevere e interpretare i comandi. L'automazione viene sempre eseguita nel contesto di una sessione. I client avviano una sessione con un server in modi specifici per ciascuna libreria, ma finiscono tutti per inviare una richiesta POST al server, con un oggetto JSON chiamato oggetto "capacità desiderate". A questo punto il server avvierà la sessione di automazione e risponderà con un ID di sessione utilizzato per inviare ulteriori comandi[12]. Inizialmente creato per

automatizzare le applicazioni mobili iOS e Android, Appium è diventato una piattaforma completa che offre possibilità di automazione basate su WebDriver per l'intero set di diverse piattaforme mobili e desktop. Questa piattaforma basata su WebDriver lo rende molto simile a Selenium, infatti anche Appium mette a disposizione delle API per identificare gli elementi con cui interagire nell'applicazione. La differenza sostanziale con Selenium sta in come il programmatore cerca i localizzatori. Nella programmazione web come ho spiegato nel capitolo 2.1 ogni browser mette a disposizione lo strumento per analizzare il codice HTML della pagina visualizzata. Se si vuole automatizzare un test su un dispositivo mobile non c'è questo aiuto, ma Appium ha pensato anche a questo mettendo a disposizione un tool che si chiama Appium Inspector. Appium Inspector è un tool che permette sostanzialmente di analizzare gli elementi mobile di qualunque schermata che si può vedere sul telefono. E questo ci permette di poter trovare dei localizzatori per intercettare gli elementi di cui si ha bisogno. Appium Inspector ha bisogno di un server Appium in esecuzione a cui possibile accedere, quindi se è in corso una sessione Appium allo stesso modo si può attivare una sessione su Appium Inspector. Nella figura fig 2.2 vediamo la schermata iniziale di Appium inspector che permette di startare una sessione. In basso sulla sinistra vediamo la parte dove inserire le *capabilities* del telefono con qui ci vogliamo collegare. Le *capabilities* obbligatorie da inserire sono il *platform name* che specifica il sistema operativo del telefono, il *deviceName* e *platformVersion*. Ci sono inoltre delle *capabilities* facoltative che aggiungono altre informazioni; la più importante è *appActivity* che permette di specificare l'app da aprire all'inizio della sessione. Una volta fatta partire la sessione quello che vedremo è nella figura fig 2.3. Alla sinistra vediamo uno screenshot di ciò che succede sul telefono. Al centro invece vediamo tutto il codice di ciò che vediamo sullo schermo. Possiamo selezionare un elemento dal codice e automaticamente sullo schermo a sinistra verrà evidenziato l'elemento. Una volta selezionato un elemento sulla destra verranno mostrate delle informazioni sull'elemento che potremo utilizzare come localizzatori. Se presente, come precedentemente spiegato per i test sul web, il localizzatore migliore è l'id, in quanto dovrebbe essere univoco

nella pagina. Si possono utilizzare inoltre gli altri attributi sempre dopo aver verificato la loro univocità nella pagina. Se nessuno degli attributi presenti riescono a selezionare univocamente l'elemento si ricorre all'utilizzo dell'XPath. L'XPath ha la stessa funzione spiegata prima nel capitolo 2.1, Appium Inspector genera in automatico l'XPath dell'elemento selezionato[fig 2.4]. L'uso dell'XPath è fortissimamente sconsigliato perché è un localizzatore che è fortemente dipendente dalla posizione degli elementi nello schermo quindi un aggiornamento dell'applicazione che modifica il layout degli elementi renderebbe il test non funzionante.



**Figure 2.2:** Schermata iniziale Appium Inspector

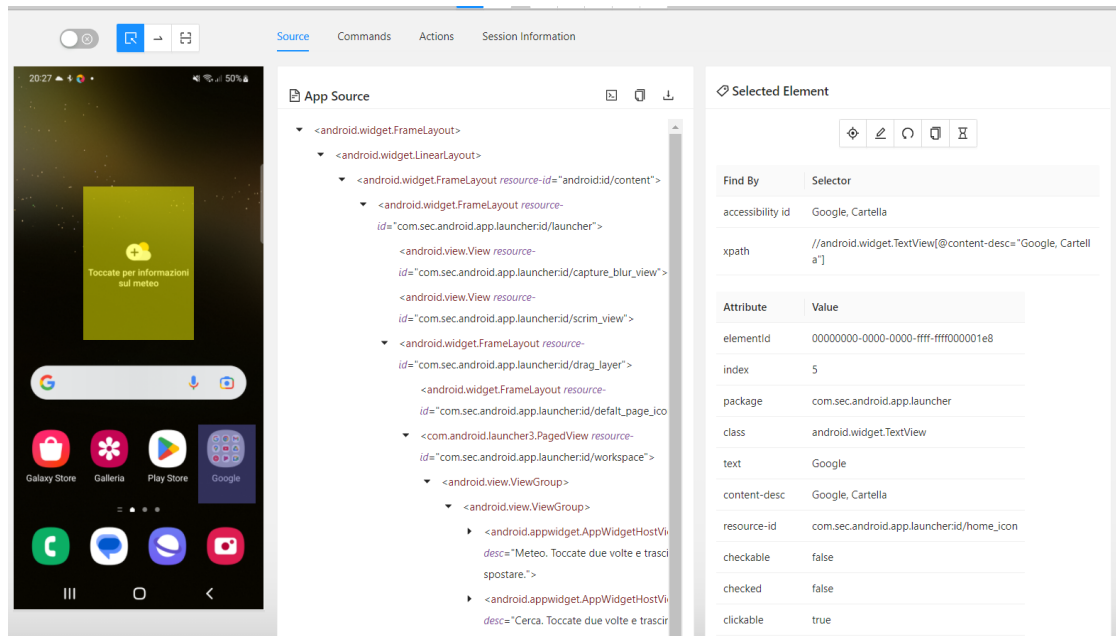


Figure 2.3: Schermata Appium Inspector per trovare i localizzatori

xpath	//android.widget.TextView[@content-desc="Google, Cartella"]
-------	---

Figure 2.4: Xpath suggerito automaticamente da Appium Inspector

## 2.3 Cucumber

Quello che si è delineato finora è un framework basato su codice Java che riesce ad automatizzare l'utilizzo degli endpoint più importanti dei sistemi odierni. Un lato importante dello sviluppo dei test è che spesso chi guarda i risultati dei test non è sempre una persona tecnica. Per questo nel framework di test aziendale è previsto l'utilizzo di Cucumber. Cucumber è uno strumento per eseguire test automatizzati scritti secondo lo stile di sviluppo basato sul comportamento (BDD). Cucumber è scritto nel linguaggio di programmazione Ruby. Inizialmente infatti Cucumber era usato solo per testare applicazioni Ruby, come complemento al framework RSpec. Poi si è esteso, e ora Cucumber supporta diversi linguaggi di programmazione attraverso varie implementazioni, come Java (Cucumber JVM), JavaScript, Ruby, .NET (SpecFlow). Sostanzialmente Cucumber è uno strumento per scrivere test leggibili dall'uomo mappati in codice: consente di descrivere come dovrebbe comportarsi il software in un testo strutturato nel semplice linguaggio Gherkin (Given - When - Then). Il testo è perfetto quindi per essere compreso anche da non sviluppatori, ma deve poi essere mappato in codice per poter essere operativo [13]. Gherkin usa un set di *keywords* che permettono di costruire una struttura al codice leggibile all'umano. Le *keywords* maggiormente usate sono : **Feature, Scenario, Given, Given, And, When**. La prima *keyword* *Feature*, permette di dare un nome di alto livello ad una *feature* software che si sta testando. Ogni file inizia con questa *keyword* seguita dai due punti e dal nome che si vuole assegnare al file. Poi si usa la *keyword* *Scenario* che permette di dividere in diversi scenari predefiniti una *feature*. Una delle cose più importanti di cucumber sono gli hooks che permettono di chiamare delle funzioni prima e dopo qualunque scenario ad esempio per inizializzare e chiudere il driver Selenium. Le altre *keywords* vengono utilizzate per descrivere gli *steps* eseguiti nel test. Le *keywords* degli *step* si usano in maniera differente a seconda del tipo di step da eseguire. I *Given steps* vengono utilizzati per descrivere il contesto iniziale del sistema. In genere è qualcosa che è successo in passato. Lo scopo degli *step Given* è di mettere il sistema in uno stato noto prima che l'utente (o il



sistema esterno) inizi a interagire con il sistema. I *When steps* vengono utilizzati per descrivere un evento o un'azione. Può trattarsi di una persona che interagisce con il sistema o può essere un evento attivato da un altro sistema. I *Then steps* vengono utilizzati per descrivere un risultato o un risultato previsto. La definizione di uno *step Then* dovrebbe utilizzare un'asserzione per confrontare il risultato effettivo con il risultato atteso. I *And steps* si usano quando ci sono più *steps* con la stessa *keyword* consecutivi. Ogni *step* viene eseguito uno alla volta in serie internamente ad ogni *step* Cucumber cercherà di trovare una corrispondenza tra il testo scritto in Gherkins e una funzione scritta nel codice e se non la troverà lancerà un'eccezione. Nel frammento di codice 2.2 possiamo vedere un esempio di *feature file* con alcuni *step* e nel frammento di codice 2.3 c'è una funzione che intercetta lo *step* scritto in Gherkin che permette di eseguire il codice associato allo *step*. Come si può vedere in figura un'altra importantissima funzione di Cucumber è poter passare dei parametri agli *step*. I parametri che si possono passare sono: **String**, **int** e anche intere tabelle.

**Listing 2.2:** Esempio di Feature Cucumber

```
1 Feature: MyFeature
2 @Test @Screenshots
3 Scenario: Example feature
4
5     Given I init the Selenium driver in incognito mode
6     Then I go to "www.google.com"
7     Then I click on the link with text "Profilo"
8     Then I click on the link with text "Cambia i dati
    personali"
```

**Listing 2.3:** Codice che intercetta step Cucumber

```
1  @Screenshot(disabled = true)
2  @Given("I init the Selenium driver in incognito mode")
3  public static void initIncognitoDriver() throws Exception
4  {
5      Utils.driverType = DriverTypes.WEB;
6      if (driver != null) {
7          driver.quit();
8      }
9      driver = Selenium.init(true);
10     driver.manage().timeouts().implicitlyWait(
11     TIMEOUT_TIME, TimeUnit.SECONDS);
12     CommonPage.loadDriver(DriverType.CHROME, driver);
13 }
14
15 // open website
16 @Then("I go to {string}")
17 public static void openWebSite(String url) throws
18 Exception {
19     Selenium.openWebsite(driver, url);
20 }
```

# Chapter 3

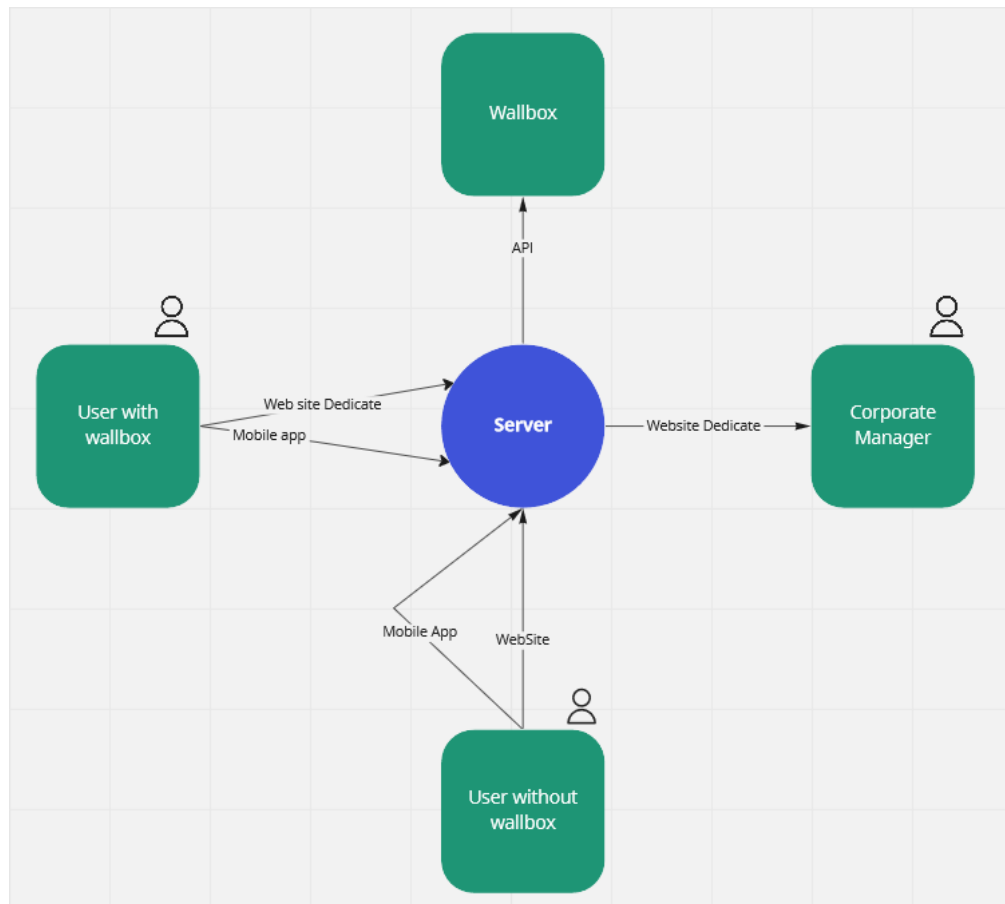
## Sviluppo Progetto

In questo capitolo, verrà illustrato il processo svolto per realizzare la soluzione di *test automation*. In primis verrà illustrato a grandi linee il sistema che verrà testato; questa parte non potrà essere molto specifica in quanto vige un contratto di riservatezza tra l'azienda che ha richiesto la soluzione di test e l'azienda per cui è stato svolto lo stage.

### 3.1 Sistema da testare

Il sistema che verrà testato è un sistema di un'azienda leader nella ricerca e nello sviluppo di soluzioni per la ricarica dei veicoli elettrici o con motore ibrido. Per cercare di spiegare meglio il sistema, sotto in figura fig3.1) viene riportato il diagramma di contesto per il sistema sotto test (SUT).

Un diagramma di contesto delinea il modo in cui le entità esterne interagiscono con un sistema software interno. Viene utilizzato principalmente per aiutare le aziende a capire l'ambito di un sistema. Di conseguenza, possono capire come progettare al meglio un nuovo sistema e i suoi requisiti o come migliorare un sistema esistente [14]. Quello che si evince dal diagramma di contesto sono gli utilizzatori del sistema e in che modo essi si interfacciano al sistema. Gli *endpoint* di questo sistema sono: la mobile app, il sito web che ha varie sezioni in base al tipo di utente che interagisce con il sistema e infine l'*endpoint* principale che è la wallbox di ricarica per le auto elettriche. Una wallbox è un



**Figure 3.1:** Diagramma di contesto

dispositivo per la ricarica domestica in abitazioni private o condominiali, sviluppato in forma compatta e adatto da fissare al muro e permette di avere maggiore capacità di ricarica (cioè una maggiore velocità) rispetto alle normali prese a muro. I livelli di potenza delle wallbox vanno da 3,5 a 22 kW in corrente alternata e la ricarica si può effettuare nell'arco di poche ore [15]. Nel SUT la wallbox può essere sia acquistata da un utente privato che essere acquistata da un'azienda, che la mette a disposizione per le ricariche di utenti che non la possiedono. L'utente che acquista la propria wallbox potrà attraverso un sito dedicato ai possessori di wallbox (CPMS) e attraverso la mobile app, associare la propria wallbox al suo account e programmare a proprio piacimento la ricarica della propria automobile. Ad esempio, può far partire una

ricarica, programmarne una per un orario prestabilito e vedere tutti i consumi associati ad ogni ricarica. L'utente che non possiede la wallbox invece, può comunque utilizzare il sistema utilizzando delle colonnine di ricarica messe a disposizione da alcune aziende. Le aziende possono acquistare delle wallbox e utilizzarle in modalità *"corporate"*. La modalità corporate viene controllata attraverso una dashboard personalizzata dal *"corporate manager"* che può associare le wallbox da mettere a disposizione e settare per ognuna di esse una tariffa da destinare agli utenti. L'utente che vuole utilizzare una wallbox non sua deve accedere alla mobile app e cercare attraverso la mappa una wallbox disponibile vicino a lui e sempre da lì può monitorare l'andamento della ricarica e tutti i consumi ad essa associati.

## 3.2 Analisi As-is e to-Be

Dopo la prima panoramica del SUT viene riportata di seguito un'analisi dei miglioramenti che la soluzione di test automation porterà al ciclo di sviluppo del software. Per fare ciò viene condotta un'analisi del processo di testing E2E del software prima di sviluppare i test e verrà definita la nuova situazione alla fine dello sviluppo. Questa analisi viene chiamata **AS-IS TO-BE**. L'analisi AS-IS, che deriva dall'inglese e significa *"così com'è"*, si basa sulla definizione, documentazione e misurazione di una situazione prima di un cambiamento programmato [16]. L'analisi AS-IS e TO-BE viene utilizzata nell'analisi dei processi quando si è in procinto di effettuare un cambiamento, per determinare lo stato attuale del processo (AS-IS) e determinare quali miglioramenti può portare il cambiamento che verrà fatto nel processo (TO-BE). Il miglioramento può essere catalogato in diversi termini, quello più importante è quello economico ma si ragiona anche a livello di qualità e affidabilità del processo. In figura (fig 3.2) viene riportato un riassunto dell'analisi condotta sottolineando gli aspetti di principale miglioramento del processo. Per l'accordo di riservatezza sopracitato non potranno essere riportate le esatte stime economiche di miglioramento dettate del processo ma verranno stimate in base ai costi medi ricercati in rete. Prima di descrivere l'analisi è bene fare una

	As is	To be
Dipendenti	La test-list richiesta veniva svolta in 10 giorni da due dipendenti e ogni aggiornamento software venivano ripetuti i test.	La test-list richiesta verrà completata da due dipendenti in 20 giorni ma ad ogni release del software basterà un giorno per la maintenance.
Processo	I test vengono svolti per due volte ogni release di software.	Una volta sviluppati i test possono essere svolti illimitatamente
Tecnologia	Dopo lo svolgimento dei test si generava manualmente un foglio con la reportistica legata ad un singolo test	Automaticament grazie al framework di test viene generata la reportistica che serve agli sviluppatori.

Figure 3.2: AS IS TO BE

precisazione su cosa sia il test *end-to-end*. Un test end-to-end può essere considerato un tipo di test funzionale in quanto mira a verificare che l'intera applicazione o il sistema funzioni correttamente, rispettando gli obiettivi funzionali specificati. Un test funzionale, in generale, è un tipo di test che viene utilizzato per verificare che una determinata funzionalità dell'applicazione o del sistema rispetti le specifiche funzionali. Un test end-to-end può essere visto come un'ulteriore estensione di questo tipo di test, in quanto non solo si verifica la funzionalità di una specifica funzione o componente, ma anche la sua integrazione e il corretto funzionamento dell'intero sistema. Con test *end-to-end* (*end-to-end testing*) si intende quell'attività di testing dell'interfaccia grafica come la vedono gli utenti del programma, dall'inizio alla fine. In altre parole rappresenta una metodologia utilizzata per verificare se il flusso di un'applicazione si sta comportando come progettato dall'inizio alla fine senza che vengano rilevati dei failure che andrebbero a inficiare sulla qualità dell'applicazione stessa [17].

### 3.2.1 AS IS

Allo stato attuale l'azienda che ha richiesto la soluzione di test automation per i test E2E, svolge i test E2E utilizzando le loro piattaforme manualmente. Questo vuol dire che prima di una nuova release del software i tester, utilizzano manualmente gli *endpoint* del sistema con lo scopo di testare i vari flussi che può svolgere l'utente, per assicurarsi che il sistema reagisce come ci si aspetta. Dai dati forniti dall'azienda i flussi E2E principali, che sono gli stessi da automatizzare, vengono svolti da due tester in maniera manuale, prima di ogni nuova release, in due settimane lavorative. Questo tempo introduce del ritardo nel processo di deployment di un nuovo aggiornamento del software. Facendo un'approssimazione sui costi, lo stipendio medio di un tester manuale per due settimane lavorative è di circa 800€ lordi, quindi l'azienda ad ogni release software spenderà all'incirca 3200€ ad ogni release da testare. Sempre partendo dai dati forniti dall'azienda questi flussi E2E vengono svolti per due volte prima di considerare il sistema affidabile e pronto al rilascio. Queste due volte naturalmente sono intese come due volte in cui il sistema risponde in maniera positiva al test. Dopo lo svolgimento dei test, i tester hanno un modello Excel da compilare manualmente che verrà inviato agli sviluppatori. Questo serve a riassumere i risultati dei test evidenziando eventuali problemi e cammini critici. Naturalmente le due settimane stimate sono una media dei casi in cui i flussi testati rispondono in maniera positiva e quando invece presentano alcune criticità. Questi ultimi casi allo stato attuale sono i più critici per il team di sviluppo poiché i test vanno poi ripetuti in maniera manuale. Dopo l'analisi dello stato attuale si passa all'analisi della nuova condizione in cui i flussi *end-to-end* verranno svolti in maniera automatizzata.

### 3.2.2 TO BE

Per tutti i flussi E2E proposti dal cliente sono stati stimati 15 giorni per il completamento. Questi 15 giorni includono il tempo di test design e quello dello sviluppo. Nelle stime fornite sono stati previsti due sviluppatori. Da aggiungere ai due sviluppatori c'è il *project*

*manager* che impatta a seconda dei rapporti con il cliente e dal suo coinvolgimento nel progetto. Il Project Manager è il responsabile di progetto, il suo core business è offrire consulenza ai committenti con l'obiettivo primo di difendere l'idea progettuale dei progettisti [18]. Questo tempo stimato è solo per il primo sviluppo dei test; dopo che i test sono stati sviluppati il tempo che viene dedicato ad ogni release è mediamente di due giornate per adattare i test a eventuali cambiamenti nell'UI. Quindi, facendo sempre una media con dei dati presenti in rete dello stipendio di uno sviluppatore di test automation, il costo di due sviluppatori è di circa 3500 € lordi per 15 giorni, questi costi però non vanno calcolati e aggiunti per ogni release del software ma vanno considerati solo per il primo sviluppo. Oltre a questo c'è da aggiungere solo le due giornate mediamente stimate in precedenza per eventuale *refactoring* dei test in modo da adattarlo ai cambiamenti delle nuove release. Seppur con cifre non precise, questo fa capire come ci sia un guadagno importante a livello economico a lungo termine e c'è anche un forte impatto sulle tempistiche di *deployment*. A livello economico per fare una stima più precisa ci sarebbe da conoscere inoltre il costo del framework proprietario dell'azienda che viene utilizzato per svolgere i test. Spostandoci dal lato economico al lato qualitativo del processo di test, la prima cosa da sottolineare è che una volta sviluppati i test possono essere svolti in un numero illimitato di volte. Questo è un enorme miglioramento rispetto allo svolgimento manuale dei flussi E2E di test. Per sottolineare ancora di più il guadagno a livello qualitativo c'è da considerare anche che il *framework* di test fornisce in tempo reale allo svolgimento dei test una reportistica associata al test svolto, che descrive in maniera dettagliata eventuali errori e problemi nel test. La reportistica è correlata agli screenshot dell'applicazione durante lo svolgimento del test; questo rende molto più facilitata la comprensione dei bug presenti. Concludendo l'analisi si evincono i netti miglioramenti dovuti dal processo di test automation sul ciclo di vita del software sia a livello economico ma soprattutto a livello di qualità dei test che di riflesso aumentano l'affidabilità del codice sviluppato. L'unica controindicazione può essere il livello di errore umano nello sviluppo dei test che è più alto nei test sviluppati su codice



che in quelli condotti manualmente. Questa però è un lato negativo che viene coperto da tutti gli altri vantaggi che ho sopracitato.

### 3.3 Implementazione della soluzione di test

Dopo aver analizzato il SUT si passa al lato implementativo del progetto. Il progetto richiesto riguarda l'automatizzazione dei flussi E2E più importanti di utilizzo del sistema precedentemente illustrato. Tra i flussi E2E ci sono quelli di iscrizione sul sito e sulla mobile app, di acquisto della wallbox, di pairing della wallbox e di utilizzo in ricarica della stessa. L'architettura che svolge i test quindi deve essere in grado di controllare tutti gli *endpoint* del sistema quali: mobile app, sito web e wallbox. Per controllare la mobile app e sito web ho già spiegato le tecnologie utilizzate nel capitolo (2), la sfida più grande è quella di controllare in maniera automatizzata la wallbox per iniziare e concludere le ricariche. Quello che è stato fornito è stato un simulatore dotato di un connettore da connettere alla wallbox su cui si possono simulare i tre stati che la wallbox può presentare che sono:

- **Disponibile:** La wallbox è scollegata dal veicolo.
- **Preparazione:** La wallbox è connessa al veicolo ma non passa corrente.
- **Ricarica:** La ricarica è in corso.

Viene riportato in figura fig 3.3 il simulatore con l'evidenza sulla manopola che girata a dovere permette di simulare gli stati della wallbox.

#### 3.3.1 Test Design

La prima fase del progetto è quella di *test design*. In questa fase si parte dalle specifiche del cliente per studiare come poter soddisfare le richieste. Il *test design* si riferisce al processo di creazione di un piano



**Figure 3.3:** Simulatore di macchina elettrica con evidenziata la manopola che simula gli stati

o di una strategia per l'esecuzione di test su un particolare sistema, applicazione software o prodotto. L'obiettivo della progettazione dei test è garantire che il processo di test sia efficace, efficiente e completo e che identifichi eventuali difetti o problemi del sistema o del prodotto da testare.

La progettazione dei test prevede in genere le seguenti fasi:

- **Definizione degli obiettivi del test:** Si tratta di identificare gli obiettivi e i requisiti del processo di test, come i tipi di difetti da

identificare, gli standard di prestazione da soddisfare o gli scenari utente da testare.

- **Identificazione degli scenari di test:** Gli scenari di test sono situazioni o condizioni specifiche che verranno testate, come diversi input, configurazioni o modelli di utilizzo.
- **Creazione di casi di test:** I casi di test sono istruzioni o script dettagliati per l'esecuzione di uno specifico scenario di test, compresi i passaggi da seguire, i risultati attesi e i dati o gli input necessari.
- **Determinazione dei dati di test:** I dati di test comprendono gli input, le condizioni e le variabili utilizzate nel processo di test, come ad esempio set di dati campione, profili utente o configurazioni di sistema.
- **Definizione delle procedure di test:** Le procedure di test delineano il processo di esecuzione dei casi di test, comprese le precondizioni, l'impostazione del test, le fasi di esecuzione e le attività successive al test.
- **Stabilire le metriche di test:** Le metriche di test sono utilizzate per misurare l'efficacia e l'efficienza del processo di test, come i tassi di difettosità, la copertura dei test o i tempi di esecuzione dei test.

La progettazione dei test è una parte essenziale del processo di collaudo del software, in quanto garantisce che lo sforzo di collaudo sia ben pianificato, strutturato ed efficace nell'identificare eventuali problemi del sistema o del prodotto da testare. Questa fase è svolta dagli sviluppatori uniti al project manager. Il tempo che si impiega per questa parte del processo non è trascurabile e dipende molto dal modo in cui sono state scritte le specifiche e dalla difficoltà dei test cases. Andando nello specifico nel progetto di tesi condotto, il test design si è sostanzialmente diviso in due parti: nella prima parte si sono svolti i test manualmente per definire gli scenari di test e capire come il sistema reagisce ai vari flussi. Il risultato di questa fase è passare dalle specifiche in figura fig 3.4 alle direttive di test step per step viste in figura fig 3.5.

Con l'attenzione sulle condizioni attese che devono essere quelle che andranno verificate nel corso dei test e inoltre alla colonna "Data" che rappresenta eventuali dati che servono per quello step.

### 3.2 Case: E2E charge of an Wallbox(Autostart, App)

Changes of statuses can be done using an Electric Vehicle Emulator with different switches that can simulate the car statuses.

- Switch On Wallbox (internet connection available)
- Wallbox is in Status A (Cable not connected, Not charging)
- Open the app
- Choose to login
- Associate the Wallbox using QR On App
- the Wallbox is in status = available
- change the Wallbox to Status B (Cable connected, Not charging)
- On App the Wallbox is in status = connected (to the car)
- change the Wallbox to Status C (Cable connected, charging)
- On App the Wallbox is in status = charging
- Stop the charge in App in app
- change the Wallbox to Status B (Cable connected, Not charging)
- On App the Wallbox is in status = finishing
- change the Wallbox to Status A (Cable not connected, Not charging)
- On App the Wallbox is in status = available

**Figure 3.4:** Caso di test fornito dall'azienda

STEP	EXPECTED RESULT	DATA
OPEN APP		
INSERT USERNAME		testreply1@mailinator.com
INSERT PASSWORD		Abc123!?
CLICK LOGIN BUTTON		
CLICK "RICARICA A CASA"		
CLICK "AGGIUNGI WALLBOX"	WALLBOX UNPAIRED	
CLICK "Le tue Wallbox"	CHECK TEXT "DISPONIBILE"	
SWITCH B STATUS	CHECK TEXT "PRONTO A RICARE"	
CLICK "START"	CHECK POPUP START RECHARGE	
SWITCH C STATUS	CHECK TEXT "STOP"	
CLICK "STOP"	CHECK POPUP END RECHARGE	
SWITCH B STATUS	CHECK TEXT "RICARICA TERMINATA"	
SWITCH A STATUS	CHECK TEXT "DISPONIBILE"	

**Figure 3.5:** Caso di test dopo il test design

La seconda parte invece è stata quella della scelta delle tecnologie da utilizzare. Per quanto riguarda la scelta delle tecnologie per testare la

mobile app e il sito web, prima del seguente progetto era stato scelto di utilizzare il framework che ho precedentemente illustrato con l'uso di Selenium, Appium e Cucumber. Per controllare il terzo *endpoint* del SUT, la wallbox, sono state trovate delle soluzioni ad-hoc. Il primo problema per l'automazione è quello dell'associazione della wallbox che permette di controllare le operazioni da svolgere sulla wallbox dal proprio telefonino. Per fare ciò l'utente deve scannerizzare un QR code che si trova sotto la wallbox. In ottica di automatizzare il flusso però si è dovuto studiare un metodo che permettesse di rendere automatica l'operazione di inquadrare il codice QR che si trova sotto la wallbox. In primis si è stampato in un foglio di carta il codice QR della wallbox per far sì che non si trovi sotto la wallbox, il che renderebbe impossibile il pairing automatizzato. Una volta avuto il foglio si è pensato ad un supporto su cui potesse andare il telefono con un buco alla fotocamera che inquadrasse sempre il foglio in modo tale che quando il test richiede di farlo, il tutto avviene in maniera automatizzata. In figura fig 3.6 e fig 3.7 il supporto con il foglio sotto da inquadrare. La seconda fase invece riguarda l'automatizzazione degli stati di ricarica della wallbox. Come spiegato in precedenza per fare ciò è stato messo a disposizione un simulatore (fig 3.3 ), il quale ha una manopola che permette di gestire i tre stati della wallbox:

- **Stato A:** corrisponde allo stato della wallbox “*Disponibile*”
- **Stato B:** rappresenta lo stato della wallbox “*Preparazione*”
- **Stato C:** rappresenta lo stato della wallbox “*Ricarica*”

L'obiettivo quindi, è quello di sviluppare una soluzione, da integrare con il framework che controlli gli altri *endpoint*, in grado di controllare la manopola del simulatore. Per fare ciò si è pensato ad una soluzione che prevede l'uso di arduino che comanda un servo collegato alla manopola del simulatore. L'obiettivo è perciò sviluppare del codice arduino e trovare una soluzione per collegare il servo alla manopola in maniera fissa.



**Figure 3.6:** Base per poggiare il telefono per associare la wallbox

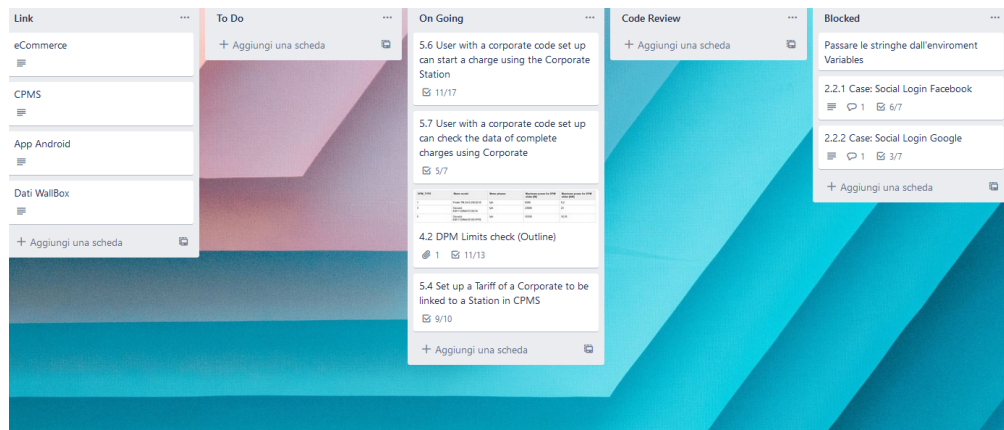
### 3.3.2 Sviluppo test

Dopo la fase di test design si passa al vero e proprio sviluppo del codice. E' molto importante per questa fase definire una strategia comune tra gli sviluppatori del codice e seguire le linee guida dettate dal project manager. Per tenere traccia degli avanzamenti oltre ai meeting di fine giornata è stato utilizzato un software gestionale tra i più utilizzati che è Trello. Qui in figura c'è un esempio di una pagina di lavoro di Trello fig 3.8. Grazie a questo software è molto facile tenere traccia dell'avanzamento dei test case, di tutti i dati accessori che servono per il test come ad esempio account di test o siti da testare e inoltre è presente anche la tabella "blocked" che permette di inserire dei test bloccati per eventuali problemi.

Si può dividere la fase di progettazione in tre parti che si differenziano in base agli *endpoint* da controllare. Per quanto riguarda mobile app e



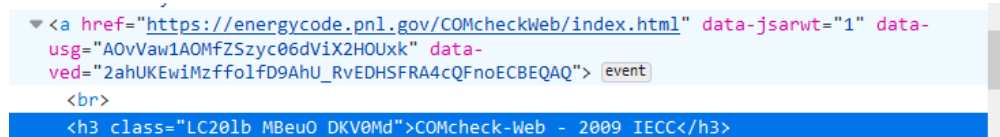
**Figure 3.7:** Base per poggiare il telefono per associare la wallbox



**Figure 3.8:** Lavagna di avanzamento task

sito web il flusso di sviluppo è abbastanza simile. Come ho spiegato nel capitolo delle tecnologie utilizzate per automatizzare l'utilizzo di un sito web e di una mobile app la prima cosa da fare è cercare i localizzatori

degli elementi con cui interagire. Non esiste un localizzatore univoco per un elemento e per questo la scelta del localizzatore è fondamentale per la qualità del codice scritto per testare. Per costruire il localizzatore in una pagina web si utilizza la funzione sviluppatore del browser che permette di vedere il codice html della pagina da testare. In figura fig 3.9 viene rappresentato un esempio di un elemento HTML e per ogni elemento va trovato un localizzatore univoco in tutto il DOM.



```
<a href="https://energycode.pnl.gov/COMcheckWeb/index.html" data-jsarwt="1" data-usg="AOvVaw1AOMfZSzyC06dViX2H0Uxk" data-ved="2ahUKewiMzfFolFD9AhU_RvEDHSFRA4cQFnoECBEQAQ"> event
<br>
<h3 class="LC201b MBeu0 DKV0Md">COMcheck-Web - 2009 IECC</h3>
```

**Figure 3.9:** Elemento HTML visionato da strumento "analizza" del browser

L'elemento sottolineato ha come unico attributo il nome della classe, ma quella classe è utilizzata per identificare più elementi del DOM quindi non si può utilizzare quello come localizzatore. In questi casi bisogna ricorrere ai localizzatori composti che sono l'xpath e il CssSelector. Questi due selettori permettono di scrivere dei localizzatori composti per intercettare gli elementi. Di default il browser costruisce un localizzatore che permette di intercettare uno specifico elemento. Questo è il localizzatore creato dal browser in maniera automatica `"/html/body/div[7]/div/div[11]/div[1]/div[2]/div[2]/div/div/div[1]/div[1]/div/a/h3"`. Questo selettore però è fortemente sconsigliato poiché è un selettore che parte dalla radice del DOM e cerca l'elemento scrivendo tutti gli elementi che stanno tra lui e la radice. Questo selettore quindi è fortemente dipendente dalla struttura del DOM quindi poco responsive ai cambiamenti dell'UI. Utilizzando questo selettore si risparmia del tempo nell'immediato ma un eventuale cambiamento di quella pagina dovuto ad un aggiornamento del SUT, renderebbe quel localizzatore inutilizzabile e con esso anche il test. Per costruire un selettore composto che comunque resti poco dipendente dal posizionamento degli elementi nel DOM la buona pratica è cercare degli elementi più vicini possibili all'elemento da noi cercato nell'albero della pagina html, e che siano localizzabili con degli attributi e da lì utilizzare le regole dell'XPath o



css selector per trovare gli elementi vicini nel DOM. Partendo da questo esempio si può scrivere un XPath che parte dall'elemento poco sopra che si vede in figura() e scendendo giù nell'albero del DOM si può trovare l'elemento da noi cercato. Questo è l'xpath che non dipende dall'UI `"a[href = 'https://energycode.pnl.gov/COMcheckWeb/index.html']/h3"`. Questo XPath trova l'elemento `"a"` che ha come attributo `"href"` il valore `"https://energycode.pnl.gov/COMcheckWeb/index.html"` e poi con lo slash si scende nella gerarchia del DOM a cercare un elemento `h3`.

Oltre alla ricerca dei localizzatori migliori durante lo sviluppo viene richiesto di scrivere delle funzioni più modulari possibili in modo da poterle riutilizzare per altri progetti. Per questo l'intento è quello di utilizzare i localizzatore più comuni come gli id o le classi che vengono utilizzati sempre negli elementi html. Nel blocco di codice viene riportato un esempio di uno step cucumber che chiama una funzione riutilizzabile.

**Listing 3.1:** Funzione Cucumber per eseguire click custom

```
1 And I click on the "Chargers": "a" element with attributes "
  href" with value "/chargers"
```

Nei blocchi di codice 3.2 e 3.3 vengono invece riportati il codice java che svolge la funzione e quello che intercetta lo step cucumber. Sfruttando i parametri che grazie a cucumber possono essere passati alle funzioni java, questa funzione in figura permette di clickare un elemento html div con l'attributo class pari al valore visto in figura. Si può notare che l'elemento, l'attributo e il valore dell'attributo sono delle stringhe inserite nello step, ciò vuol dire che con questa funzione si può clickare qualsiasi elemento html con qualsiasi attributo di un determinato valore, solo scrivendo lo step cucumber e cambiando i parametri passati. Questo significa che ogni funzione di click per attributo non dovrà più essere scritta ma basterà scrivere solo lo step cucumber. A volte però non è possibile scegliere un localizzatore in base agli attributi dell'elemento perché non tutti gli elementi possiedono attributi univoci nel DOM.

**Listing 3.2:** Funzione che intercetta il codice Gherkins

---

```
1 @Then("I click on the {string}: {string} element with  
   attribute {string} with value {string}")  
2     public static void clickCustomAttribute(String  
   description, String element, String attribute, String  
   value) {  
3         Selenium.clickCustomAttribute(driver, description,  
   element, attribute, value);  
4     }
```

**Listing 3.3:** Funzione che permette di clickare un elemento custom

```
1 public static void clickCustomAttribute(WebDriver driver ,
2     String description , String element , String attribute ,
3     String value) {
4     try {
5         WebElement e = driver.findElement(By.cssSelector(
6     element + "[" + attribute + "=\"" + value + "\""));
7         wait.until(ExpectedConditions.
8     elementToBeClickable(e)).click();
9     } catch (Exception e) {
10        Assert.fail("The " + description + " is not
11        present" , null);
12    }
13 }
```

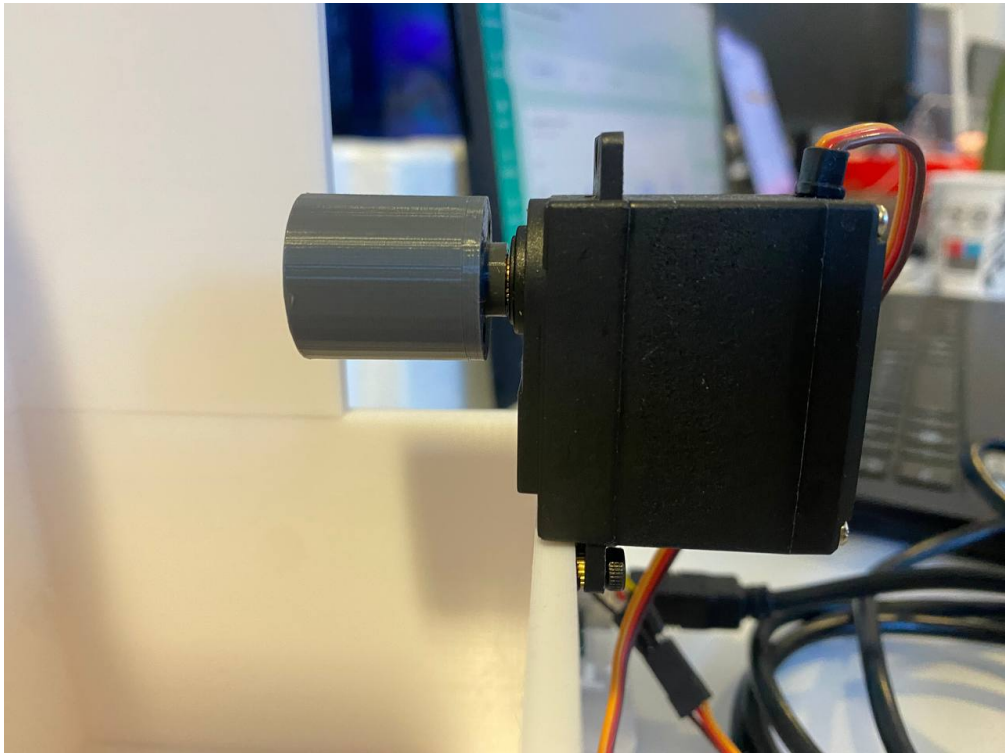
Analogo discorso si può fare per la progettazione dei test su mobile, dove però il codice non è più html e quindi non si può vedere dal browser ma si utilizza un plugin di appium che si chiama appium inspector che permette di vedere tutti gli elementi presenti nella pagina visualizzata su app.

Quindi il flusso base della progettazione è quello di:

- Individuare gli elementi con cui interagire.
- Costruire il miglior localizzatore che intercetti univocamente l'elemento.
- Scrivere lo step Cucumber per la funzione da eseguire.
- Scrivere il codice java che esegue la funzione.

Una volta scritto il codice java si passa alla fase di debug che consiste nell'eseguire i test e controllare i risultati ottenuti. Oltre al codice che serve per far eseguire le azioni che farebbe un umano allo script di test, la parte forse più importante è forse quella del controllo delle risposte del sistema in base ai risultati attesi. Nella fase di test design durante lo svolgimento manuale dei test sul sistema, si è preso nota della risposta ideale del sistema. Questo è importante per far sì di testare sia il lato frontend che il lato backend del sistema. Un'altra

parte importante dello sviluppo è il contatto con il cliente. Durante lo sviluppo dei test si entra a contatto con un nuovo sistema che è ancora in fase di sviluppo. Il ruolo del tester in questo caso è molto importante perché non deve solo scrivere e sviluppare gli script di test ma deve comunicare eventuali punti critici del sistema; punti poco chiari a livello di utilizzo o in cui il sistema risponde più lentamente. Quindi durante lo sviluppo si svolge anche un'analisi del sistema a livello di usabilità e facilità di comprensione per l'utente finale che aiuta molto gli sviluppatori del sistema. Vi è una comunicazione continua infatti tra i tester e i sviluppatori dell'app proprio per migliorare la qualità del codice indipendentemente dai flussi E2E da testare. La fase più difficile della parte implementativa è stata quella sviluppata per il controllo della wallbox. Come spiegato nel capitolo di test design si è scelto di utilizzare arduino che comanda un servo in grado di girare la manopola del simulatore. In questa fase i lati critici sono stati due: Il primo a livello di sviluppo è stato quello di collegare il framework scritto in java con il codice arduino che effettivamente gira il servo è il secondo problema è stato quello di modellare un pezzo fisico in grado di collegare il servo di arduino alla manopola del simulatore. Per quanto riguarda il secondo problema è stato modellato un pezzo 3D, partendo dalle misure della manopola e dalle scanalature della stessa, in grado di incastrarsi perfettamente sia con il servo Arduino che con la manopola del simulatore. Come prima cosa si è realizzato il modello a mano e poi spostato su Blender, che è un software di animazione 3D che ha permesso di costruire il pezzo da poter stampare. Infine si è stampato il pezzo e si è verificato con dei test entrasse e permettesse al servo(fig 3.10) di controllare il simulatore. Nel frattempo è stato sviluppato il codice Java che permette di comandare i pin di arduino. Per questo è stata utilizzata una libreria di terze parti che ha integrata la gestione di arduino e mette a disposizione dei metodi che gestiscono un servo collegato ad un pin. Qui nel blocco di codice 3.4 viene riportato il codice che permette di girare il servo comandato da Arduino.



**Figure 3.10:** Connettore stampato in 3D per fissare il servo di arduino al simulatore

**Listing 3.4:** Funzione Arduino per comandare il servo

```
1 public static void servoMove(Integer angle, Integer pin)
   throws IllegalArgumentException, IOException {
2     var myLED = currentBoard.getPin(9);
3     // myLED.setValue(0);
4     myLED.setServoMode(0, 180);
5     myLED.setValue(angle);
6     ReportFormatter.logInfo("Angle setted to " + angle);
7     try {
8         Thread.sleep(1000);
9     } catch (Exception ex) {
10        System.out.println("sleep error.");
11    }
```



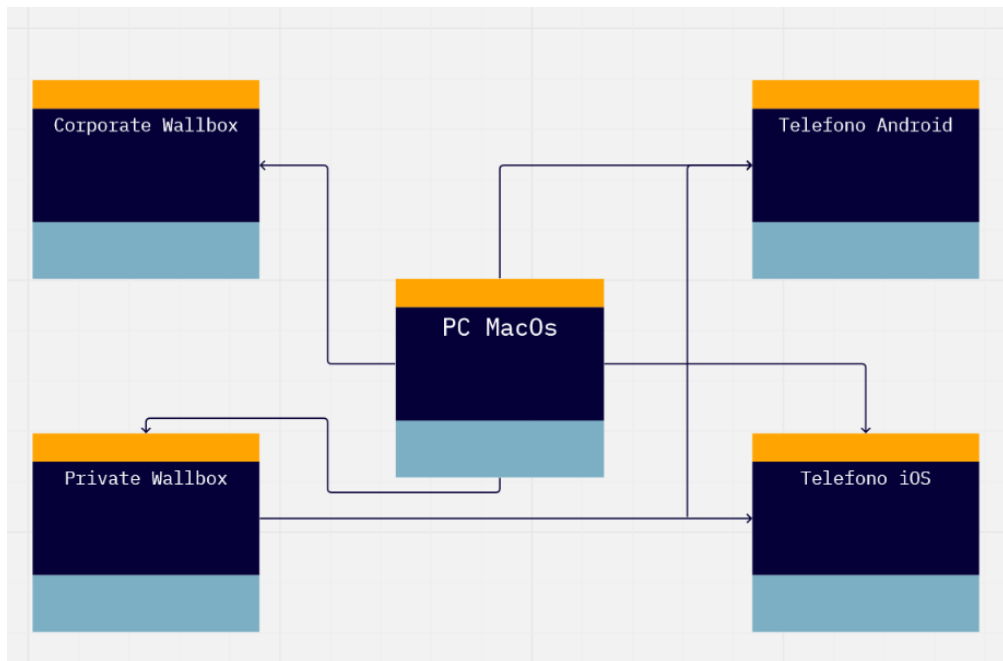
**Figure 3.11:** Setup completo per svolgere un test

### 3.4 Deployment della soluzione

Una volta sviluppati i casi di test l'ultima fase è quella del deployment dell'architettura completa per renderla disponibile al cliente per lanciarsi i test a suo piacimento. Poiché il framework di test è proprietario dell'azienda per cui è stato sviluppato il progetto non potrà essere analizzato tutto il framework ma verrà descritto come black box. Per rendere più semplice la spiegazione in figura [fig 3.12] viene riportato il deployment diagram dell'architettura che serve per testare.

Un deployment diagram è un tipo di diagramma UML che mostra la distribuzione fisica dei componenti software ai nodi hardware o agli ambienti di esecuzione. In altre parole, illustra come il sistema software viene distribuito su diverse macchine o dispositivi, tra cui server, workstation, router e così via.

Gli elementi principali di un diagramma di distribuzione sono:



**Figure 3.12:** Deployment Diagram

- **Nodi:** rappresentano i dispositivi o le macchine fisiche su cui possono essere distribuiti i componenti software. I nodi possono essere etichettati con le loro specifiche hardware o software.
- **Componenti:** rappresentano i moduli o i sottosistemi software che vengono distribuiti sui nodi. I componenti possono essere etichettati con il loro nome, la versione o altre proprietà.
- **Artefatti:** rappresentano i file o i dati utilizzati dai componenti, come eseguibili, script, file di configurazione o librerie. Gli artefatti possono essere memorizzati sui nodi o trasmessi tra di essi.
- **Percorsi di comunicazione:** rappresentano le connessioni o i collegamenti tra i nodi, come cavi di rete, collegamenti wireless o protocolli di comunicazione. I percorsi di comunicazione possono essere etichettati con il loro tipo, la larghezza di banda o la latenza.
- **Associazioni:** rappresentano le relazioni tra nodi, componenti e artefatti. Le associazioni possono essere utilizzate per mostrare

quali componenti sono distribuiti su quali nodi, quali artefatti sono utilizzati da quali componenti o quali percorsi di comunicazione sono utilizzati da quali nodi.

Utilizzando un diagramma di distribuzione, gli architetti e gli sviluppatori di software possono visualizzare la struttura fisica di un sistema software e capire come i suoi componenti sono distribuiti su diversi nodi. Questo può aiutarli a ottimizzare le prestazioni, la scalabilità, l'affidabilità e la sicurezza del sistema, identificando potenziali colli di bottiglia, singoli punti di guasto o rischi per la sicurezza.

Quello che avrà a disposizione il cliente sarà un nodo che è un computer MacOS da cui svolgere i test. La scelta del sistema operativo del computer nodo è stata obbligata dal fatto che sono stati richiesti i test mobile anche per telefoni iOS e questi ultimi possono essere comandati solo da un computer MacOS. Oltre a questo il cliente avrà un telefono iOS e uno Android che saranno quelli su cui si svolgeranno i test mobile. E per ultimo saranno presenti due wallbox che serviranno una per i test di ricarica di un utente privato e l'altra per i test degli utenti che non hanno la wallbox quindi ne devono utilizzare una di una azienda.



## Chapter 4

# Attività di ricerca: Nuovi trend

Dopo l'approfondimento dello sviluppo del progetto si può dire che per automatizzare dei test si utilizzano script predefiniti per eseguire un'azione specifica nel software e verificare se produce l'output atteso. Questo fa sì che il sistema e l'architettura costruita per l'automazione dei test deve essere cambiata con i cambiamenti del SUT. Questa resta una grande limitazione che non consente di avere dei sistemi capaci di fare test automatizzati all'aggiornarsi dell'architettura da testare. Per cercare di risolvere questo problema andrebbero utilizzate delle soluzioni che utilizzano l'intelligenza artificiale. L'intelligenza artificiale (AI) sta rivoluzionando numerosi campi, inclusi quelli del test e dell'automazione. Grazie alla sua capacità di apprendere dai dati e di adattarsi a nuove situazioni, l'AI può aiutare a migliorare la qualità e l'efficienza dei test automatizzati. In questo capitolo, esploreremo come l'AI sta influenzando il *test automation*, analizzando le sue applicazioni attuali e future. In parallelo allo sviluppo del progetto di tesi ho svolto un lavoro di ricerca per cercare di migliorare il framework in uso dal Team cercando di utilizzare tecniche che utilizzano l'AI. Nella prima parte della ricerca mi sono concentrato sul capire come l'AI può rivoluzionare il *test automation* e sull'utilizzo di alcuni tool che implementano il ML L'AI o il DL nel *test automation*. Nella seconda parte invece ciò che mi è stato chiesto è stato di trovare una soluzione

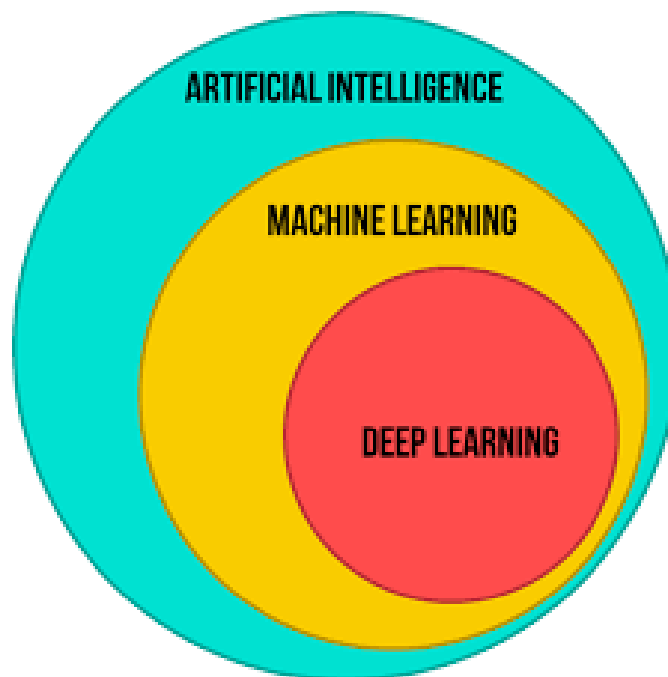
stabile per quanto riguarda il *Visual Testing* che consiste nella ricerca di elementi per immagine.

## 4.1 Artificial Intelligence Machine Learning Deep Learning

Spesso Artificial Intelligence, Machine Learning e Deep learning sono termini utilizzati in maniera simile ma in realtà sono diversi l'uno dall'altro.

- Il concetto di Artificial Intelligence è molto antico, infatti il termine è stato coniato nel 1956 in una conferenza chiamata Dartmouth Summer Research Project on Artificial Intelligence [19]. In quella conferenza [20] la proposta che John McCarthy, Marvin Minsky, Nathaniel Rochester e Claude Shannon fecero può essere riassunta dalla seguente frase che viene chiamata proposta di Dartmouth :” Lo studio procederà sulla base della congettura per cui, in linea di principio, ogni aspetto dell'apprendimento o una qualsiasi altra caratteristica dell'intelligenza possano essere descritte così precisamente da poter costruire una macchina che le simuli.” Ancora adesso dopo quasi 70 anni la definizione di Artificial Intelligence rimane sempre quella; per Artificial intelligence infatti intendiamo un vasto settore delle scienze informatiche che studia algoritmi e tecniche che permettono alle macchine di imitare comportamenti umani.
- Il Machine Learning, invece, è considerato come un ramo dell' AI che si concentra nell'utilizzo dei dati e di algoritmi per imitare come l'essere umano apprende. Alla base del Machine Learning c'è la raccolta e l'analisi dei dati che vengono processati da algoritmi per migliorare il funzionamento delle macchine. La più grande differenza quindi tra AI e ML è che il ML permette al sistema di imparare e migliorare automaticamente dalle esperienze attraverso i dati senza essere esplicitamente programmato [21].

- Il nuovo step di evoluzione del ML è il Deep Learning. Il Deep Learning infatti è un sotto insieme del ML che si ispira al modo di imparare dell'essere umano, infatti si basa su reti neurali artificiali che permettono ai computer di imparare dalle esperienze passate seguendo lo stesso schema di apprendimento dell'uomo. A differenza del ML il training time è molto più alto ed è computazionalmente più intenso, però il grande vantaggio è l'identificazione di nuove feature da parte del computer automaticamente senza che siano identificate manualmente.



**Figure 4.1:** Descrizione di AI ML DL

**Table 4.1:** Differenze tra AI ML e DL

<b>Artificial Intelligence</b>	<b>Machine Learning</b>	<b>Deep Learning</b>
Capacità di una macchina di sviluppare la propria intelligenza.	Capacità di una macchina di utilizzare algoritmi grazie ai quali può apprendere automaticamente dai dati e dall'esperienza.	Capacità di una macchina di apprendere tramite algoritmi che imitano il cervello umano.

## 4.2 Intelligenza artificiale nell'automazione dei test

Nel Test Automation le applicazioni più importanti dell'AI sono :

- **Self-healing tests:** gli strumenti di automazione dei test tradizionali hanno determinati identificatori per definire i componenti di un'applicazione. Quando l'applicazione viene aggiornata o modificata in qualsiasi modo, anche questi componenti possono cambiare. A causa delle definizioni fisse date a questi elementi, i test che funzionavano correttamente prima della modifica ora falliranno e forniranno un risultato falso negativo. I *self-healing* test applicano algoritmi di intelligenza artificiale per identificare automaticamente errori imprevedibili dovuti a proprietà dinamiche e per consigliare un'alternativa migliore o aggiornare automaticamente lo script. Questo tipo di test non solo impedisce ai test di fallire, ma consente anche di risparmiare il tempo che un tester avrebbe dovuto dedicare a cercare di trovare e risolvere il problema [22].
- **Visual Locators:** Nei test dell' UI basati sull'intelligenza artificiale, i localizzatori visivi sono ora in grado di trovare elementi, anche quando i loro localizzatori sono stati modificati su un'applicazione web, mediante immagini. Ciò elimina la necessità di hard coding con id o altri localizzatori simili. Inoltre, gli strumenti di automazione intelligente sono ora in grado di utilizzare l'OCR (riconoscimento ottico di caratteri) e altre tecniche di riconoscimento delle immagini per mappare visivamente l'applicazione e individuare regressioni visive o verificare gli elementi[22] .
- **AI Analytics of Test Automation Data:** I test generano abbondanti quantità di dati che devono essere analizzati per ricavarne un significato. L'applicazione dell'intelligenza artificiale a questo processo ne aumenta notevolmente l'efficienza. Gli algoritmi AI possono essere utilizzati per rilevare e classificare gli errori. Applicazioni più sofisticate di algoritmi AI sono anche in grado di

identificare falsi negativi e veri positivi nei casi di test. Questo può essere molto utile e ridurre notevolmente il carico di lavoro [22].

Le prime due applicazioni di quelle sopracitate saranno quelle di mio interesse e quelle che nei paragrafi successivi verranno analizzate più nel dettaglio.

## 4.3 Un nuovo approccio al testing: Self-Healing

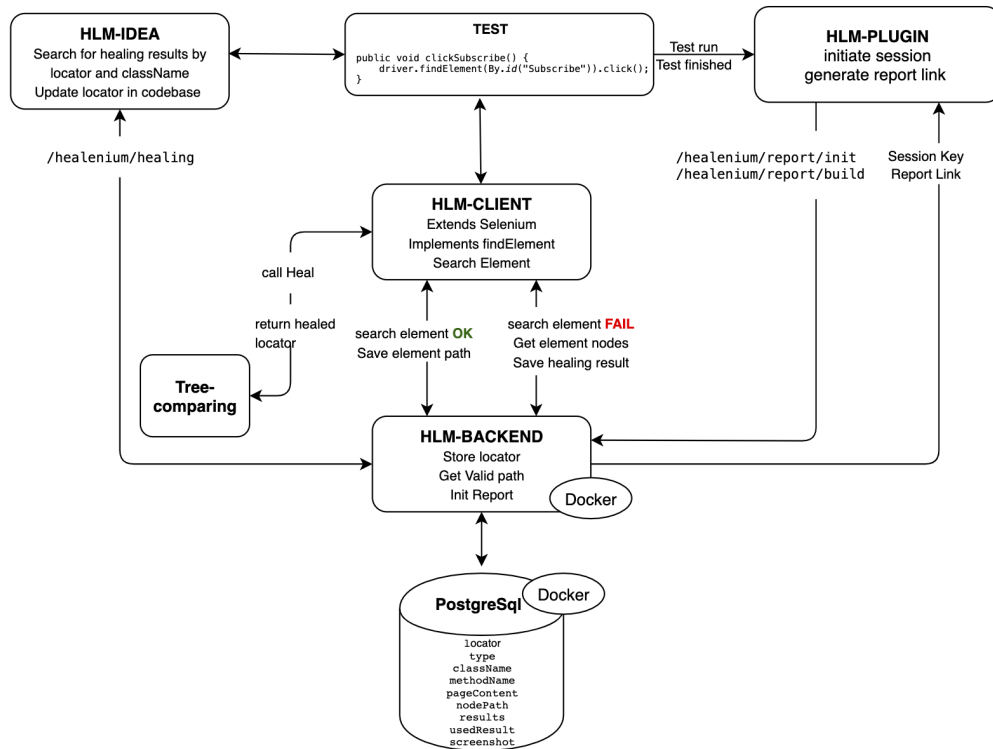
L'approccio di *self-healing* (tradotto autoguarigione) nel mondo del testing è un approccio di testing che sfrutta il ML per migliorare la *maintenance* associati ai test automatizzati. Come ho descritto prima nel capitolo 2.1 una delle strategie per compiere dei test automatizzati è quella di sfruttare dei localizzatori che referenziano gli elementi con cui interagire nelle pagine da testare. Nei vari aggiornamenti dei siti o delle applicazioni i localizzatori degli elementi possono cambiare e quindi questo rende i test difficili da mantenere. L'approccio di *self-healing* permette di riconoscere i cambiamenti dei locatori del SUT e adattare l'esecuzione dei test dinamicamente così da renderli più robusti. Dopo una breve analisi insieme anche al team in cui mi trovo questa è risultata l'applicazione dell'AI più vantaggiosa nell'immediato in quanto corregge l'instabilità dei test e li rende robusti ad eventuali aggiornamenti e cambiamenti del SUT. Nella ricerca dei tool che utilizzano l'approccio *self-healing* ho cercato di trovare quello che oltre ad essere più efficiente sia anche facilmente integrabile con il framework già in utilizzo nell'azienda, il quale è basato su Selenium con Java come spiegato nei capitoli precedenti. Per questi motivi le mie ricerche e i miei studi si sono concentrati in un tool chiamato Healenium.

### 4.3.1 Healenium

Healenium è un'estensione per framework di test che migliora la stabilità dei casi di test basati su Selenium, gestendo le modifiche degli elementi Web aggiornati. Le funzionalità di *self-healing* consentono di

sostituire il localizzatore che è cambiato in un nuovo aggiornamento del sistema, con un nuovo valore e correggere il test in fase di esecuzione. Un vantaggio notevole è che si tratta di un tool *open source* e disponibile in GitHub[23]. Oltre a questo un altro vantaggio da tenere in considerazione è che si tratta di un estensione di Selenium quindi è stato difficile implementarlo nel framework di test, perché anche questo tool utilizza i metodi precedentemente illustrati nel capitolo 2.1 per localizzare gli elementi nel DOM. Il funzionamento di Healenium si basa su un *backend* costituito da un database PostgreSQL che durante l'esecuzione salva alcune info relative agli elementi utilizzati durante il test, le quali serviranno nel momento in cui questi elementi dopo eventuali aggiornamenti del DOM non saranno più localizzati dagli stessi localizzatori. In figura fig 4.2 viene esplicitato con uno schema a blocchi il funzionamento di Healenium. Si può notare dallo schema come il funzionamento di base di esecuzione dei test rimanga invariato rispetto a quello dei framework *Selenium-based*. Viene inserita la comunicazione tra il *backend* e il *client* che svolge i test quando c'è un errore sulla ricerca degli elementi.

Se ciò accade nel normale funzionamento di Selenium si arriva all'eccezione *NoSuchElement* e il programmatore deve analizzare nuovamente la pagina per cercare un nuovo localizzatore in grado di localizzare l'elemento. Healenium, invece, utilizza il suo algoritmo di Machine Learning per il confronto degli alberi suggerendo un nuovo localizzatore e correggendo automaticamente con il nuovo elemento. L'Algoritmo di Machine Learning utilizzato è l'LCS Algorithm che è un algoritmo che confronta gli alberi, che in questo caso sono il DOM dei test precedenti e il DOM del test in esecuzione[fig 4.3]con dei pesi per specificare gli id, le classi e tutti gli attributi, per trovare la più lunga sottosequenza in comune per l'elemento da cercare. L'algoritmo in fase di healing genera un nuovo localizzatore con un relativo score che è un numero da 0 a 1 il quale indica l'affidabilità del nuovo localizzatore. In fase di setup di Healenium è possibile impostare una soglia di score sopra la quale effettuare l'healing[fig 4.4]. In questo modo, durante l'esecuzione stessa dell'automazione del test, se ci sono modifiche al localizzatore, viene corretto automaticamente e otterremo i risultati di esecuzione aspettati



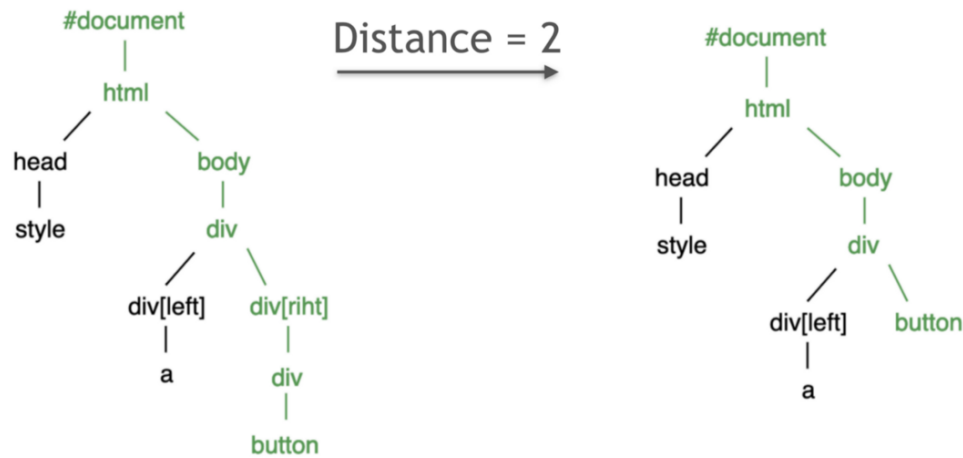
**Figure 4.2:** Diagramma a blocchi che sintetizza il funzionamento di Healenium

[23] . Alla fine del test in automatico è possibile visualizzare i report dell'elemento su cui è partito l'algoritmo di *self-healing* con relativo screenshot e con la possibilità, in caso di errore, di reportarlo così da salvare nel database e non riutilizzare il risultato precedentemente calcolato. Inoltre se si utilizza IntelliJ come IDE di sviluppo, Healenium mette anche a disposizione un plugin che automaticamente, se considerato corretto, corregge il localizzatore dell'elemento nel codice.

### 4.3.2 Esempio di utilizzo di Healenium

Naturalmente per implementare tutto ciò e testarlo in un progetto, avrei avuto bisogno di un sistema in aggiornamento per fare vere e proprie stime di guadagno nell'utilizzo di questo tool. Per questa ricerca ho fatto delle prove facendo un semplicissimo test che apre una





**Figure 4.3:** Algoritmo confronto alberi

pagina di chrome. Ho eseguito prima correttamente il test un paio di volte per consentire ad Heleanium di salvare i risultati relativi al test su database e poi ho simulato un cambiamento di un elemento nella pagina modificando manualmente l'HTML durante il test per testare le funzionalità di Heleanium. Prima di tutto verranno mostrati i cambiamenti fatti nel codice e il codice del test di prova. Per prima cosa dopo le varie installazioni delle dipendenze nel framework di test viene configurato in un container docker il database con il backend, che ha il compito di far partire l'algoritmo quando è necessario. E poi un file molto importante da configurare è il file `healenim.properties`[fig 4.4] I campi in questo file sono :

- **Recovery-tries:** Questo campo è il numero di prove che si provano a fare quando c'è la procedura di self-healing.
- **Score-cap:** Questo campo è lo score minimo accettato da heale-nium quando si effettua un'operazione di self-healing come ho precedentemente spiegato.
- **Heal-enabled:** Questo campo impostato a true attiva il self healing quando necessario; se si vuole disattivare il self-healing basterà mettere questo campo a false.

```
recovery-tries = 1
score-cap = 0.6
heal-enabled = true
hlm.server.url = http://localhost:7878
hlm.imitator.url = http://localhost:8000
```

**Figure 4.4:** Contenuto file Healenium.properties

- I due link sono link alle porte utilizzate da helenium per mettere il report e effettuare le procedure di self-healing

La struttura del codice del test rimane sempre la stessa spiegata precedentemente e utilizzata nel progetto. L'unica modifica fatta sostanziale è racchiudere il driver di selenium in un driver di Heleanium come ben si può vedere nella blocco di codice mostrato in basso[4.1].

**Listing 4.1:** Inizializzazione driver Healenium

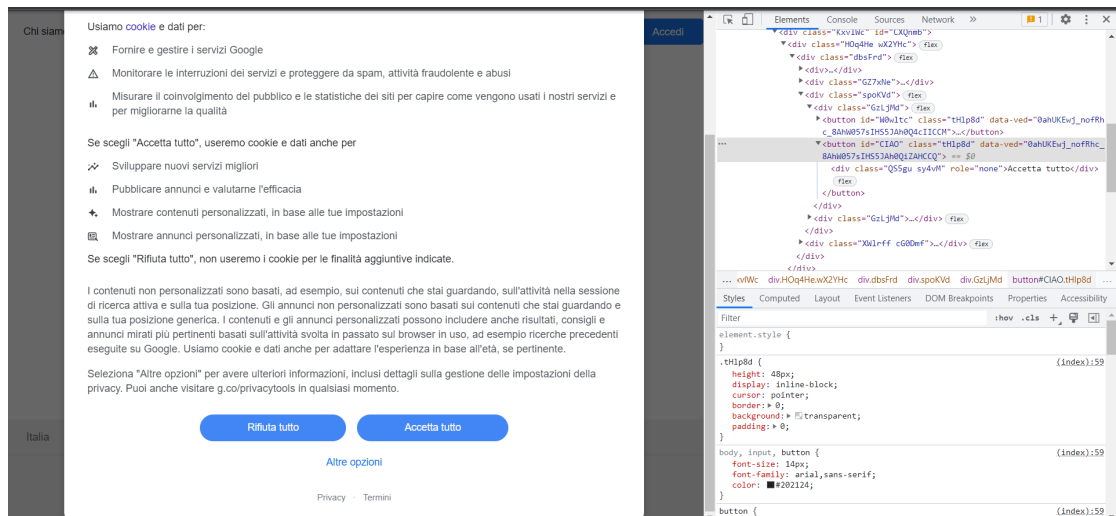
```

1  @Screenshot(disabled = true)
2  @Given("I init the Selenium driver in incognito mode")
3  public static void initIncognitoDriver() throws Exception
4  {
5      Utils.driverType = DriverTypes.WEB;
6      if (driver != null) {
7          driver.quit();
8      }
9      WebDriver delegate = Selenium.init(true);
10     driver = SelfHealingDriver.create(delegate); //CAMBIO
11     RISPETTO A INIZIALIZZAZIONE DI SELENIUM
12     driver.manage().timeouts().implicitlyWait(
13     TIMEOUT_TIME, TimeUnit.SECONDS);
14     CommonPage.loadDriver(DriverType.CHROME, driver);
15 }

```

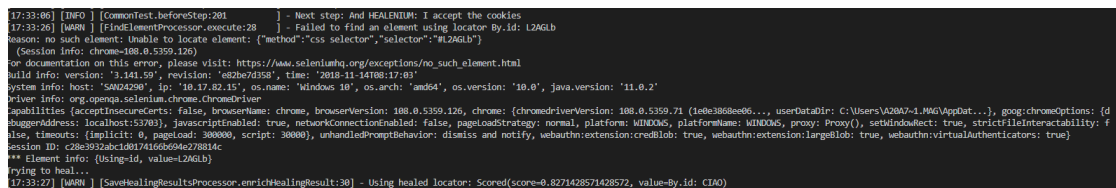
Per il resto del programma verrà utilizzato il driver allo stesso modo di come viene utilizzato in Selenium. Questo è un grosso vantaggio nell'integrazione di Healenium nel framework già in utilizzo nel team perché permette la migrazione di tutto il codice già in utilizzo solo con un cambiamento nell'inizializzazione del driver. Questo test deve semplicemente aprire il browser, andare sul sito *"www.google.com"* e cliccare il tasto per accettare i cookies. Durante le prime run dei test viene fatto completare correttamente e successivamente viene modificato manualmente nell'HTML l'ID del bottone che va premuto inserendo "CIAO" come valore come si può vedere in questo screenshot[fig 4.5]

A questo punto nel test quando scade il timeout si lancia l'eccezione *NoSuchElement* ed a questo punto entra in gioco Healenium che fa partire il meccanismo di *SelfHealing*. Viene mostrato in figura fig 4.6 uno screenshot dei log che sono stampati da Healenium durante la procedura di *self-healing*. La procedura parte con la dicitura *Trying to heal...* e successivamente viene stampato il valore trovato con il nuovo localizzatore con lo score relativo che dipende da quante modifiche ci sono nell'intera pagina rispetto ai precedenti test e quindi indica l'affidabilità della correzione. Essendo un test banale naturalmente il risultato è positivo ma è comunque interessante analizzare il report generato automaticamente da Healenium. Nel report[fig 4.7] possiamo



**Figure 4.5:** Modifica a runtime del locatore

vedere la funzione nella quale si è scatenata l'eccezione, il vecchio e il nuovo localizzatore e infine uno screenshot con l'elemento modificato riquadrato di rosso. Una funzione importante del report è il tasto *success* che può essere messo in verde, se la correzione è giusta, oppure in rosso se è sbagliata. Questo aiuta l'algoritmo a capire se ha preso la decisione giusta oppure no. Per completezza viene riportato anche il *feature file* scritto in Cucumber dove si può capire quali sono le azioni che il test deve compiere [4.2].



**Figure 4.6:** Log presi da un tentativo di healing

Healing Report 2023-01-17T16:39:24.52 Please, provide feedback on healing success to make algorithm stronger



Element	Screenshot	Success
<pre>test.steps.HealeniumSteps.clickAccetta()  failedLocatorValue = L2AGLb failedLocatorType = By.id healedLocatorValue = CIAO healedLocatorType = By.id</pre>		

Figure 4.7: Report Healenium

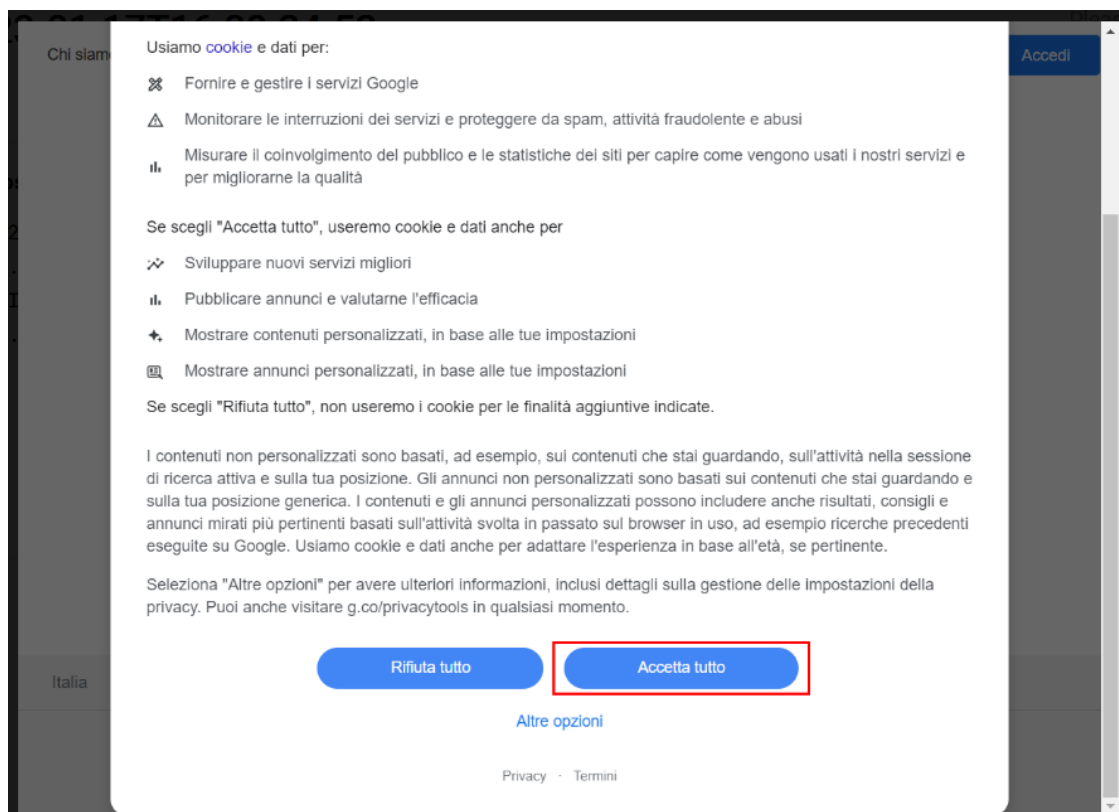


Figure 4.8: Screenshot associato a report healenium

**Listing 4.2:** Esempio Feature Healenium

```
1 Feature: MyFeature
2   @Test @Screenshots
3   Scenario: Example feature
4
5       Given HEALENIUM: I init the Healenium driver
6       And HEALENIUM: I go to "www.google.com"
7       Then HEALENIUM: I click on the link with text "
  Profilo "
8       Then HEALENIUM: I click on the link with text "Cambia
  i dati personali"
```

### 4.3.3 Risultati Ottenuti: Self-Healing

Purtroppo durante il periodo di ricerca non c'è stato il tempo per testare questo tool su un progetto vero perché nei progetti già sviluppati il tool non era utilizzato e quindi non era presente un database che contenesse i test lanciati in modo corretto che è alla base del funzionamento di Healenium. Per questo non si potranno avere delle stime esatte di risparmio del tempo. Allo stesso modo i risultati avuti pensandoli in ottica di un progetto vero e proprio, con una *test list* ampia, sono molto interessanti e qui vengono riassunti i vantaggi e i svantaggi di questo tool.

- **Vantaggi**

- Come primo vantaggio in assoluto sicuramente c'è l'abbattimento del tempo di *maintenance* dei test dovuto ad aggiornamenti software che riguardano l'UI. Infatti molti sistemi in sviluppo hanno frequenti aggiornamenti nell'interfaccia utente e questo rende i test automatizzati, che si basano sulla ricerca di elementi nel DOM, inutilizzabili. L'approccio *self-healing* permette di mantenere i test consistenti anche durante gli aggiornamenti del sistema.
- Il primo vantaggio ha come stretta conseguenza un altro vantaggio che deriva dal vantaggio di tempo. Questo vantaggio di tempo infatti, aiuta i team a concentrarsi sulla scrittura di

nuovi test invece di mantenere e correggere vecchi test che si sono interrotti a causa di problemi non legati al prodotto.

- I test forniscono un accurato feedback di qualità e la pipeline CI fallirà solo in caso di problemi relativi al prodotto.
- Un altro vantaggio da non sottovalutare è la facilità di integrazione con dei framework Selenium-based. In questo caso ha facilitato molto il setup di questo nuovo tool in un framework aziendale già precedentemente in uso.

- **Svantaggi**

- Questo approccio presenta oltre ai vantaggi detti alcuni svantaggi non trascurabili. Il primo è che per quanto sia facile l'integrazione con un framework *Selenium based* c'è un problema sull'eccezione che fa partire l'algoritmo di self-healing . Quando un elemento non viene trovato durante l'esecuzione di test per far partire l'algoritmo di self-healing c'è bisogno che venga lanciata l'eccezione di *noSuchElement*. Questa eccezione viene scatenata solo dai metodi *findElement*. Nel codice a volte c'è bisogno prima di ricercare un elemento con il metodo di *findElement* chiamare delle altre funzioni su questo Elemento; l'esempio più rilevante è quello dell'inserimento di un'attesa per far sì che l'elemento compaia a schermo. In questo caso non si usa solo il metodo *findElement* e quindi se l'elemento da cercare non è lo stesso non verrà scatenata l'eccezione *noSuchElement* ma un'altra eccezione che non permette all'algoritmo di *self-Healing* di partire.
- Il secondo svantaggio è legato ai tempi di esecuzione. L'eccezione che fa partire il *self-healing* verrà scatenata allo scadere di un timeout che porta con se il driver e solitamente viene impostato almeno a 120 secondi. Questo vuol dire che ogni elemento non trovato si aspetterà questo tempo. Questo comunque resta uno svantaggio relativo perché se l'algoritmo va a buon fine quei due minuti di attesa fanno risparmiare il tempo di correzione allo sviluppatore.

- Un altro svantaggio è l'obbligo che questo tool ha di mantenere in locale il database e il backend di Healenium. Questo può diventare un problema in un'ottica aziendale con tanti progetti che sono in esecuzione poiché questo potrebbe far esplodere la dimensione del database che contiene le informazioni sugli elementi utilizzati durante i test.

## **4.4 Visual testing**

Nell'ultima fase del periodo di ricerca, invece, è stato condotto uno studio di soluzioni che implementano il Visual Testing nel test su Mobile App. Lo studio si è incentrato su dispositivi mobili sia per esigenza aziendale in quanto ci sono già dei progetti in corso che, con l'uso di questi approcci su Mobile App verrebbero notevolmente semplificati. In secondo motivo durante lo sviluppo del progetto di tesi è venuto fuori che l'approccio basato su componenti è molto più affidabile nello sviluppo di test web. Quello che si pone l'approccio visuale al testing è quello di interagire con l'AUT nello stesso modo in cui ci interagisce l'utente a differenza dell'approccio basato su componenti che è un approccio legato alla ricerca del programmatore nel codice degli elementi con cui interagire. In seguito a numerosi studi [24], si può affermare che la scrittura dei casi di test diventa molto meno tediosa utilizzando l'approccio visuale, d'altro canto però, quest'ultimo presenta ancora molti casi di falso positivo o falso negativo nella ricerca degli elementi durante il test. Le soluzioni ricercate oltre ad essere affidabili, sono quelle di facile integrazione con il framework di test. In letteratura è presente una soluzione che soddisfa entrambi i requisiti sopracitati. Questa soluzione si basa sulla libreria OpenCv. Questa libreria ha un ampio sviluppo in Python però ultimamente viene utilizzata anche in Java insieme ad Appium.

### **4.4.1 OpenCv integrato in Appium**

Come spiegato nel capitolo 2 i tool utilizzati per lo sviluppo di test automatizzati su mobile app il tool utilizzato è Appium integrato nel



codice Java. Proprio partendo da questo la ricerca si è specializzata su una soluzione che permettesse di implementare l'approccio visuale con meno cambiamenti possibili nel framework. E la prima cosa trovata si trova nella documentazione di Appium [25], in cui si trova uno specchietto che descrive come da Appium 1.9 in poi è stato inserito un plugin che gestisce la ricerca di elementi per immagini. L'architettura usata da Appium si basa sulla libreria OpenCv per confrontare le immagini [26]. La tecnica che utilizza la libreria per confrontare le immagini è quella del **Template matching**. Il Template Matching è un metodo ampiamente utilizzato per il rilevamento di oggetti nelle immagini digitali, richiede una grande potenza di elaborazione poiché è un metodo esaustivo che confronta i livelli di intensità di un'immagine sorgente pixel per pixel con un'immagine modello che contiene l'oggetto da identificare[24]. Lo scopo quindi del template matching è quello di trovare un area nell'immagine template(ad esempio lo screenshot dell'AUT) che sia simile all'immagine che gli passiamo in input. Verrà spiegato in breve come effettua il template matching questa libreria. Per prima cosa c'è bisogno di due immagini:

- 1. **Immagine sorgente**: l'immagine che va ricercata nell'immagine Template.
- 2. **Immagine Template**: Immagine in cui verrà ricercata un area compatibile all'immagine sorgente.

Per effettuare il confronto la libreria effettua uno scorrimento da sinistra a destra e dall'alto verso il basso dell'immagine template pixel per pixel. Ad ogni scorrimento quindi si partirà dal pixel e si ritaglierà un area che parte da quel pixel e di grandezza pari all'immagine sorgente e solo allora verrà effettuata la comparazione tra le due aree. Questo confronto può essere fatto seguendo diversi approcci, la cosa fondamentale però è che si segua la stessa metrica così da avere per ogni pixel un risultato di quanto siano simili le due aree. Questo meccanismo è nascosto a livello di codice sotto i metodi che offre la libreria, in Appium infatti esistono due metodi che sono i più utilizzati; il primo è *findElementByImage* che usa come localizzatore un'immagine quindi mette insieme l'approccio componentBased con l'approccio Visuale,

il secondo invece si chiama *matchImage* che confronta le immagini e torna solo il risultato. Questi due metodi naturalmente hanno numerosi parametri che possono rendere più o meno accurato il confronto. Il parametro più importante in assoluto da impostare è la soglia di accuratezza. Questo parametro è indicato sottoforma di float e va da 0 a 1 e come ben si può immaginare indica all'algoritmo quale sia la soglia di accuratezza con cui considerare una comparazione positiva. Un altro parametro importante è un flag sul color. Come impostazione di default questi metodi prima di effettuare il *template matching* convertono l'immagine in scala di grigi per facilità sul confronto. Questo perché il confronto viene fatto dopo aver trasformato l'immagine in byte; un'immagine a colori però è rappresentata da tre byte uno per ogni colore a seconda della codifica utilizzata. Quella in scala di grigi invece ha come informazione un byte per ogni pixel e questo facilita di non poco l'algoritmo. Se si vuole però apprezzare il confronto anche sul colore con molta precisione la conversione dei colori in byte viene fatta con un altro metodo che tiene conto non solo di 255 colori come la scala di grigi ma associa ad un pixel più di un byte. Oltre a questi poi ci sono una serie di parametri che servono a fissare le dimensioni o a scalare le immagini qualora servisse. Qui sotto di seguito viene un piccolo esempio dell'uso di questa libreria con relativi frammenti di codice e in seguito analizzerò i risultati ottenuti. Questo esempio è preso da un problema avuto durante la progettazione della soluzione di test automation. In figura viene riportato uno screenshot[fig 4.9] di una schermata dell'applicazione mobile del sistema illustrato nel capitolo 3. In questa immagine si possono notare le Wallbox non private che l'utente può utilizzare. Nel test specifico era richiesto in modo specifico di selezionare la wallbox colorata di rosa. Questa wallbox è la wallbox che è associata all'account "*corporate*" legato al proprio account. Di seguito 4.3 viene riportato il codice utilizzato per svolgere la funzione che clicca sull'elemento che ha una specifica immagine.

**Listing 4.3:** Funzione per clickare un elemento trovato attraverso un immagine

```
1 public static void clickOnElementByImage(AndroidDriver<
  MobileElement> driver, String imageName) throws
  IOException {
2     try {
3
4         File img = new File("src/test/img/" + imageName);
5         String base64EncodedImageFile = Base64.getEncoder
  ().encodeToString(Files.readAllBytes(img.toPath()));
6         MobileElement element = driver.findElement(
  MobileBy.image(base64EncodedImageFile)); // or use the
  locator
7         element.click();
8
9     } catch (Exception e) {
10        Assert.fail("The image is not present", e);
11
12    }
13
14 }
```

Come spiegato in precedenza il *template matching* viene eseguito con un controllo byte per byte quindi il parametro da passare alla funzione è una stringa che contiene i byte dell'immagine da cercare[fig 4.10]. Per fare ciò serve inserire la seconda riga del codice [4.3] che permette di trasformare un immagine in byte. Si utilizza per questione di risparmio di spazio la codifica Base64. Base64 è un sistema di codifica che consente la traduzione di dati binari in stringhe di testo ASCII, rappresentando i dati sulla base di 64 caratteri ASCII diversi. Viene usato principalmente come codifica di dati binari nelle e-mail, per convertire i dati nel formato ASCII [BASE64]. Questo approccio non ha dei risultati molto stabili. Si dividono i risultati, però, in base al risultato aspettato dal test. In caso di ricerca di un elemento che effettivamente si trova nella pagina i risultati sono affidabili. Nel caso di ricerca di un elemento da immagine che non si trova nello schermo i risultati, invece, non sono confortanti. In figura[4.11] si può vedere una tabella che sintetizza i risultati dei test. Questa tabella prende spunto dalla Confusion Matrix che è uno strumento utilizzato nel Machine Learning per capire la bontà di un algoritmo di classificazione. Nella seguente tabella sono presenti 4 stati:

- **True Positive:** Questo risultato del test si ottiene quando il valore aspettato è true (elemento presente nello schermo) e effettivamente il codice riesce a rilevare l'elemento.
- **True Negative:** Questo risultato del test si ottiene quando il valore aspettato è false (elemento non presente nello schermo) e effettivamente il codice non rileva l'elemento.
- **False Positive:** Questo risultato del test si ottiene quando il valore aspettato è false (elemento non presente nello schermo) ma il codice rileva l'elemento.
- **False Negative:** Questo risultato del test si ottiene quando il valore aspettato è true (elemento non presente nello schermo) ma il codice non rileva l'elemento.

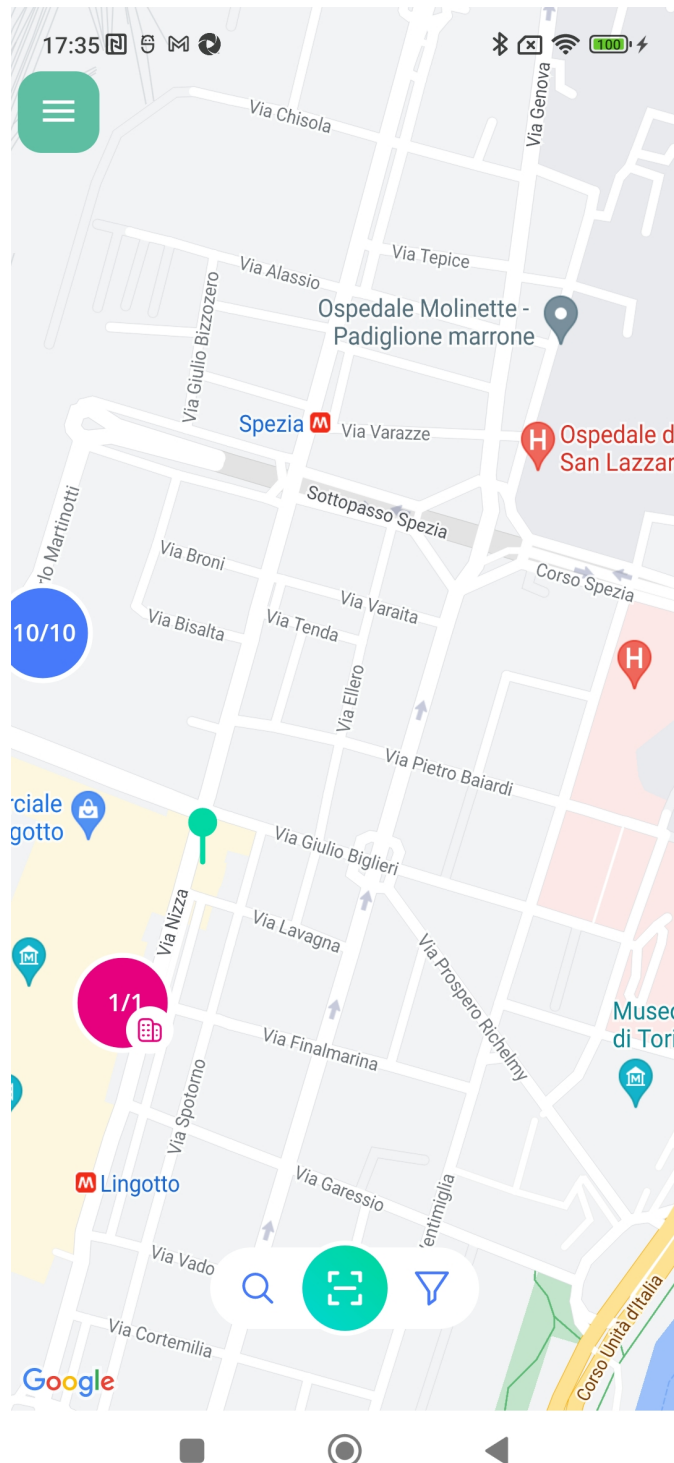
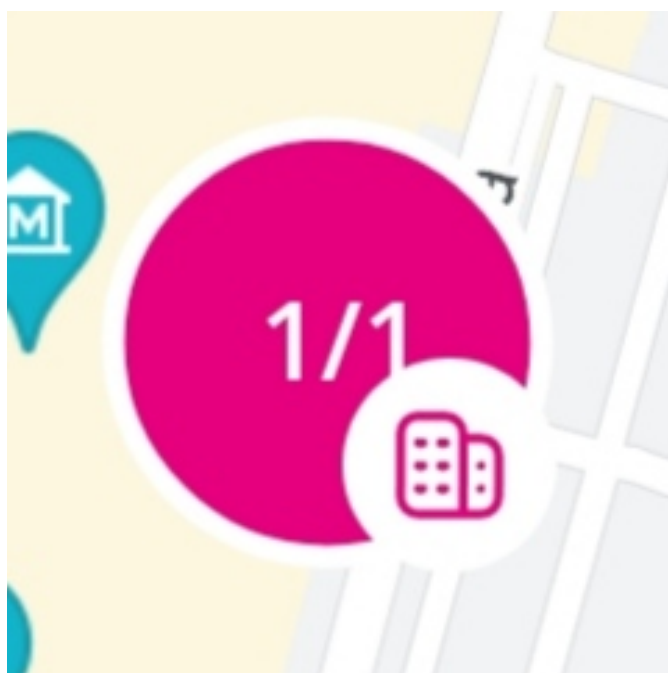


Figure 4.9: Template per effettuare template matching



**Figure 4.10:** Immagine da ricercare nello screenshot

		Value of Test	
		TRUE	FALSE
Expected Result	TRUE	48	2
	FALSE	22	28

**Figure 4.11:** Tabella per riassumere i risultati dei test

Quest'analisi è stata eseguita compiendo 50 test in cui l'immagine da ricercare era presente nello schermo e 50 per il caso opposto. Per analizzare il primo caso ci si concentra sulla prima riga che presenta i *true positive* e i *false negative*. Si può vedere dal numero presente nel grafico che è nettamente sbilanciato sui *true positive*. Questo vuol dire che il codice trova l'elemento 48 volte su 50 quando l'immagine è presente nello schermo. Questo numero molto positivo non si ottiene però per il secondo caso, cioè quello in cui il risultato atteso è false. Per vedere i risultati del secondo caso in analisi si guarda la seconda riga che presenta i *false positive* e i *true negative*. Nella seconda riga si vede che il numero di *true negative* è maggiore del numero dei *false positive*. Questa statistica però è molto più equilibrata rispetto al primo caso infatti su 50 test fatti solo 30 hanno dato il risultato atteso. Questo è un sintomo di poca affidabilità di questo algoritmo nella verifica di non presenza di un immagine. Per tutti e due i casi è importante sottolineare che la soglia di accuratezza utilizzata per performare i test è di 0.7/1. Questo numero è stato preso da una ricerca[] che afferma che è il numero minimo di threshold per considerare affidabile il *template matching*. Proprio per questi risultati poco affidabili in casi più difficili, di seguito si prova a capire se l'uso di OpenCv, che è la libreria più usata per compiere operazioni con le immagini, può portare dei risultati migliori. Tutta la potenza della libreria di openCV

infatti non si può utilizzare con Java che è in utilizzo nel framework aziendale ma bisogna spostarsi su Python; qui OpenCv offre dei metodi da utilizzare che possono effettuare, sempre con la tecnica del template matching, dei confronti più dettagliati. Nel prossimo capitolo viene illustrato i metodi che permettono di compiere template matching e come questi mi hanno permesso di arrivare a risultati migliori. Inoltre nell'ultimo punto viene illustrata l'integrazione tra questa parte scritta in Python e l'architettura già in utilizzo che è basata su codice Java.

#### 4.4.2 OpenCv in Python e l'integrazione nel TAF

Quello su cui mi sono concentrato per testare questa libreria e capire se utilizzabile in maniera affidabile è il semplice confronto di due immagini. La tecnica utilizzata è sempre la stessa utilizzata e illustrata prima(Template matching) però in questo caso si customizzare il confronto. Ad esempio OpenCv espone dei metodi che permettono di ritagliare le immagini da confrontare se ad esempio sappiamo già la parte in cui sarà il template nell'immagine sorgente. Da alcune ricerche ho poi appreso come la libreria fatta per python sia molta più sviluppata rispetto a quella usata da Appium internamente. Per accertarmi di ciò ho fatto un test con gli stessi screenshot usati in precedenza e confrontato l'accuratezza. Qui sotto in figura viene mostrato lo script in python [4.6] che permette di fare il confronto di due immagini.

**Listing 4.4:** Template Matching Python

```
1
2
3 img_bgr = cv2.imread( 'image.png', cv2.IMREAD_COLOR)
4 #img_gray = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2GRAY)
5 template = cv2.imread( 'triangoloWHITE.png', cv2.
    IMREAD_COLOR)
6
7 g, w ,h= template.shape[::-1]
8
9
10 # Apply template matching
11 res = cv2.matchTemplate(img_bgr, template, cv2.
    TM_CCOEFF_NORMED)
```



```
12 threshold = 0.87
13
14 loc = np.where( res >= threshold )
```

Come si può notare dal codice python il metodo *matchTemplate* oltre alle due immagini da confrontare ha un terzo attributo che viene chiamato il coefficiente di somiglianza. Nella *Template Matching*, i coefficienti vengono utilizzati per misurare la somiglianza tra un'immagine modello e un'immagine di ricerca. Esistono diversi tipi di coefficienti, ma i più comuni sono:

- **Sum of squared differences (SSD)**: Misura la somma delle differenze al quadrato tra il modello e i valori dei pixel corrispondenti nell'immagine di ricerca.
- **Cross-correlation coefficient (CC)**: Misura il prodotto normalizzato del punto tra il modello e i valori dei pixel corrispondenti nell'immagine di ricerca.
- **Normalized cross-correlation (NCC)**: È simile al coefficiente CC ma è normalizzato in modo da avere un valore compreso tra -1 e 1, con 1 che indica una corrispondenza perfetta e -1 che indica una perfetta non corrispondenza.
- **Correlation coefficient of phase-only matched filter (CPM)**: Si basa sulla trasformata di Fourier del modello e dell'immagine di ricerca e misura la correlazione tra le componenti di fase di queste trasformate.

La scelta del coefficiente dipende dalla natura delle immagini e dai requisiti dell'applicazione. Ad esempio, il coefficiente NCC è comunemente utilizzato quando le condizioni di illuminazione possono variare tra il modello e l'immagine di ricerca, mentre il coefficiente SSD è utilizzato quando le immagini hanno un basso contrasto o sono soggette a rumore. In figura [fig 4.12] c'è la stessa tabella utilizzata per l'analisi precedente, compilata con i risultati dei test con questo metodo. Si può ben notare come i risultati sono fortemente sbilanciati su *true positive*

e *true negative*. Questo significa che i risultati aspettati arrivano quasi a coincidere con i risultati ottenuti.

Visti gli ottimi risultati di questo metodo si è studiato un modo di implementare questo metodo nel framework aziendale. Per fare ciò si è partito da quello che già viene fatto dalla *Buisness Unit* della mia azienda che si occupa solo di *Test Automation*. Questo team ha già una parte che fa visual testing e utilizzano lo stesso framework. Per mettere in contatto i due servizi è stato sostanzialmente creato un piccolo servizio su un pc nella rete aziendale che espone delle API da chiamare per effettuare il confronto delle immagini. In figura il codice del codice relativo all'API che confronta due immagini[4.5]. Queste API avranno come parametri i byte delle immagini così da poterle ricostruire lato server e la threshold desiderata e ritorneranno in un JSON il risultato del confronto con dei dati accessori come ad'esempio le coordinate dell'immagine trovata che può servire in alcuni casi di test. Il blocco di codice [4.6] mostro il semplice metodo che permette dai byte di effettuare il decode e ricostruire l'immagine.

	Value of Test	
	TRUE	FALSE
Expected Result	TRUE	FALSE
TRUE	49	1
FALSE	4	46

**Figure 4.12:** Tabella per riassumere i risultati dei test nel secondo caso

#### Listing 4.5: API per Template Matching

```

1 @app.route('/check_elem', methods=['POST'])
2 @app.route('/visual/check_elem', methods=['POST'])
3 def check_element():
4     ip = title_API(f"bool element", request.remote_addr)
5     rqs = request.form
6
7
8     tmp = bool_elem(rqs, ip)
9     return jsonify(tmp)

```

#### Listing 4.6: Template Matching Python

```

1 decoded_data=base64.b64decode((encoded_data))
2 img_file = open('image.png', 'wb')
3 img_file.write(decoded_data)
4 img_file.close()

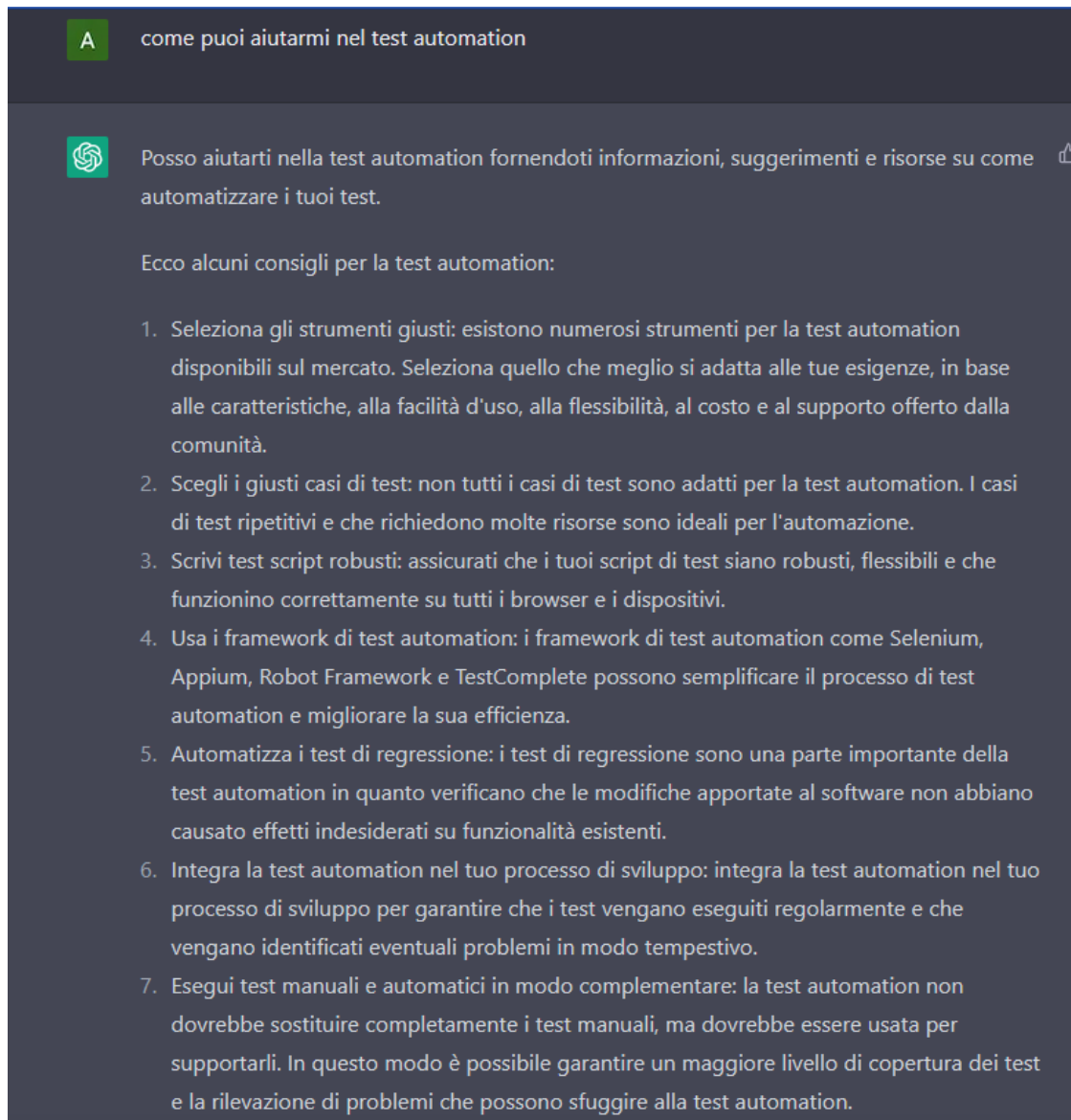
```

## 4.5 Intelligenza artificiale conversazionale nel Test Automation

Seguendo il trend odierno che spinge molto sulle intelligenze artificiali conversazionali, l'ultima parte del lavoro di ricerca è stato svolto su questa tematica. In particolare, il focus è stato quello di cercare di capire come questi sistemi di intelligenza artificiale possono dare una mano nell'ambito del test automation. La modellazione linguistica del codice è stata un argomento importante nella ricerca sull'ingegneria del software da quando la promessa di modellare codice è stata illustrata per la prima volta da Hindle et al. [27]. Con il miglioramento delle tecniche per la modellazione del linguaggio, i ricercatori hanno iniziato a utilizzare architetture di Deep Learning (DL) per apprendere rappresentazioni gerarchiche del codice che possono essere utilizzate per varie attività a valle [28]. Parallelamente, le comunità dell'apprendimento e di elaborazione del linguaggio naturale hanno iniziato a costruire modelli su larga scala incentrati su un tipo specifico di architettura neurale, il trasformatore. architettura neurale, il trasformatore [29]- [30]-[31], addestrato su enormi insiemi di dati di testo. Gli esperimenti hanno illustrato la potenza rappresentativa di questi modelli linguistici di grandi dimensioni (LLM), e di modelli linguistici adattati specificamente al codice [32]-[33]-[34]-[35]. Tuttavia, tali modelli erano in gran parte vincolati a specifiche impostazioni e non prevedevano forme naturali di interazione con gli utenti finali. Alla fine del 2022, OpenAI ha introdotto ChatGPT [36], uno strumento di AI strumento di intelligenza artificiale costruito sulla base degli LLM esistenti, che consentiva l'interazione attraverso un'interfaccia conversazionale. Per consentire questo tipo di interazione, OpenAI ha fatto uso dell'apprendimento dal feedback umano, perfezionando i metodi del precedente lavoro su InstructGPT [36], che ha addestrato i LLM con dati non supervisionati e con una supervisione sotto forma di sia con dati non supervisionati che con una supervisione sotto forma di istruzioni per il compito. In sostanza, il modello è stato inizialmente addestrato su conversazioni umane reali, conversazioni umane basate sul

testo, poi ha imparato a perfezionare le sue risposte sulla base del feedback di valutatori umani che hanno giudicato la qualità delle risposte in un contesto di apprendimento per rinforzo. Questo processo si è rivelato molto efficace nel creare un'interfaccia in cui gli utenti di accedere facilmente alla "conoscenza" latente dei LLM[37]. Per capire in che posto si colloca nello sviluppo di test automation ChatGPT in questo periodo di ricerca si sono provate a fare alcune domande a ChatGPT. Partendo da domande più generali e arrivando a domande più coincise. Sono racchiuse qui nelle figure fig 4.13 e fig 4.14 e fig 4.15, tre delle domande più significative che sono state fatte a ChatGPT nel periodo di ricerca. Le prime due domande riportate sono domande abbastanza generiche e riguardano in maniera generale il test automation e in maniera un po' più specifica sul test design. Le risposte date sono molto approfondite e sorprendenti. Il bot oltre a consigliare dei tool di sviluppo riesce anche a suggerire buone norme su come inserire il test automation nel ciclo di sviluppo software. Sulla fase di test design rimane un po' più carente in quanto non è possibile fornirgli tutte le informazioni riguardanti il SUT. Quindi anche sollecitato con domande coincise non riesce a fornire un output preciso di risposta. Basti pensare che alcuni SUT in fase di test non sono ancora rilasciati quindi non disponibili sul web. Per questo motivo ChatGPT non possiede le informazioni per effettuare tutte le azioni da compiere durante il test design. I risultati più eclatanti sono arrivati nella richiesta più puntuale riportata in fig . Questo blocco di codice è il codice con cui ChatGPT ha risposto alla domanda specifica fatta in fig 4.15 . Il seguente script 4.7 utilizza tutti i metodi che ho spiegato nel capitolo 2. L'unica pecca è la ricerca dei localizzatori degli elementi web; in questo, infatti, ChatGPT non riesce ad essere preciso. Nonostante questo può essere una grande mano a livello di sviluppo del codice agli sviluppatori di test automation. Si può sintetizzare quindi che ChatGPT, o una qualunque intelligenza artificiale conversazionale in questo momento, non può sostituire il ruolo dei tester che realizzano soluzioni di test automation. Molte volte infatti non è possibile fornire tutti i dati per arrivare ad una risposta già funzionante. Tuttavia può essere preso come un grande aiuto dal punto di vista di definizione di regole per inserire il test automation nel ciclo di sviluppo e soprattutto

una grossa mano a livello di sviluppo del codice necessario.



**Figure 4.13:** Domanda a ChatGPT sul test automation

puoi effettuare test design in test automation

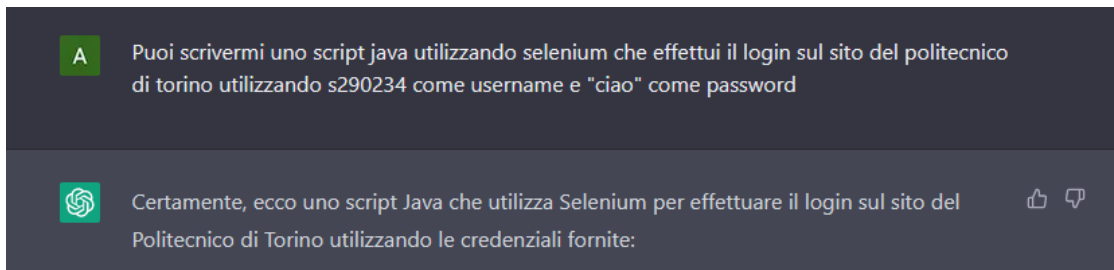
Sì, posso effettuare test design in test automation. Il test design è un'attività importante nella creazione di test automation efficaci. In particolare, il test design definisce il comportamento atteso del sistema sotto test (SUT), identifica le condizioni di test e stabilisce i passi necessari per eseguire il test.

Per effettuare il test design in test automation, posso utilizzare tecniche come la progettazione basata sui requisiti, la progettazione basata sui casi d'uso, la progettazione basata sui dati e la progettazione basata sui comportamenti. Inoltre, posso utilizzare strumenti come i diagrammi di flusso, le matrici di tracciabilità, le tabelle di decisione e i diagrammi di sequenza per aiutarmi nella progettazione dei test.

Una volta progettati i test, posso quindi automatizzarli utilizzando gli strumenti di test automation appropriati. La creazione di script di test automatizzati richiede competenze di programmazione e conoscenza degli strumenti di automazione. Posso utilizzare linguaggi di programmazione come Java, Python o Ruby per scrivere script di test, e strumenti di test automation come Selenium, Appium o TestComplete per eseguire i test.

In sintesi, posso progettare test efficaci e automatizzarli utilizzando una combinazione di tecniche di progettazione dei test e strumenti di automazione dei test.

**Figure 4.14:** Domanda a ChatGPT sul test design



**Figure 4.15:** Domanda a ChatGPT sullo sviluppo di codice per test automation

**Listing 4.7:** Codice fornito da ChatGPT

```
1
2 import org.openqa.selenium.By;
3 import org.openqa.selenium.WebDriver;
4 import org.openqa.selenium.WebElement;
5 import org.openqa.selenium.chrome.ChromeDriver;
6
7 public class PolitoLogin {
8     public static void main(String[] args) {
9         // Imposta il percorso del driver di Chrome
10        System.setProperty("webdriver.chrome.driver", "/
11        percorso/del/driver/chromedriver");
12
13        // Crea un'istanza del driver di Chrome
14        WebDriver driver = new ChromeDriver();
15
16        // Apri la pagina di login del Politecnico di Torino
17        driver.get("https://www.polito.it/session/new");
18
19        // Inserisci le credenziali di accesso
20        WebElement username = driver.findElement(By.id("
21        user_username"));
22        WebElement password = driver.findElement(By.id("
23        user_password"));
24        username.sendKeys("s290234");
25        password.sendKeys("ciao");
26
27        // Effettua il login
28        WebElement loginButton = driver.findElement(By.name("
29        commit"));
```



```
26         loginButton.click();
27
28         // Chiudi il browser
29         driver.quit();
30     }
31 }
```

## Chapter 5

# Conclusioni e sviluppi futuri

La soluzione di test automation presentata in questa tesi ha dimostrato di essere efficace nell'automatizzare i test funzionali e di regressione dei software. Questo lavoro di tesi vuole dimostrare come il test automation stia rivoluzionando l'approccio al software testing. Tuttavia esistono ancora dei limiti nel seguente approccio di testing. Alcuni di questi limiti possono essere eliminati grazie al lavoro di ricerca e grazie ad un utilizzo stabile di tecniche di AI nel ciclo di sviluppo dei test. Queste tecniche sono il trend del momento e questo lavoro di tesi ha dimostrato che sono applicabili nel test automation. Questo presuppone una voglia di cambiamento e di investimento nel campo del testing; cosa che non sempre viene fatta ma che è alla base di un software qualitativo. Come visto in questo lavoro di tesi infatti, tecniche sempre più sofisticate e precise di testing rendono lo sviluppo del codice più consistente e qualitativo. Proprio in questa ottica la soluzione di test automation potrebbe essere ulteriormente migliorata attraverso l'integrazione con strumenti di *continuous integration e continuous delivery* (CI/CD), per automatizzare il processo di esecuzione dei test e di rilascio del software. Infine, è importante sottolineare l'importanza della collaborazione tra gli sviluppatori e i tester nella definizione dei test e nella gestione dei cambiamenti nell'interfaccia utente. Solo attraverso un approccio collaborativo e integrato sarà possibile garantire una soluzione di test

automation efficace e sostenibile nel tempo. In sintesi, la soluzione di test automation presentata in questa tesi rappresenta un importante passo avanti nell'automazione dei test funzionali e di regressione dei software, ma esistono ancora opportunità per migliorare ulteriormente l'integrazione con strumenti di CI/CD.

# Bibliography

- [1] Reply. «L’assemblea degli azionisti. Bilancio 2021». In: (2021) (cit. on p. 1).
- [2] Edoardo Narduzzi. «La storia di Reply il primo unicorno Ict made in Italy». In: (2016) (cit. on p. 1).
- [3] F.Meloni. «Competitività e risorse: Il caso Concept Reply». Unpublished (cit. on p. 3).
- [4] *ISTQB Documentation*. <https://istqb.ita-stqb.org/docs/ITASTQB-FLSY-2018.pdf>. Accessed: 2022-11-19 (cit. on pp. 3, 4).
- [5] H. Zhi-gen, Y.Quan, and Z.Xi. «Research on Agile ProjectManagement with Scrum Method». In: (2009), pp. 26–29 (cit. on p. 5).
- [6] E. Dustin, J. Rashka, and J. Paul. «Automated software testing: introduction, management, and performance». In: *Boston: Addison-Wesley* 1 (1999), p. 5 (cit. on p. 6).
- [7] Eliane Collins, Arilo dias-Neto, and Vicente F.de Lucerna Jr. «Strategies for Agile Software Testing Automation: An Industrial Experience». In: (2012) (cit. on p. 6).
- [8] Emil Alegroth, Robert Feldt, and Helena H.Olsson. «Transitioning Manual System Test Suites to Automated Testing: An Industrial Case Study». In: () (cit. on p. 6).
- [9] Roland Weber Stefan Berner and Rudolf K. Keller. «Observations and Lessons Learned from Automated Testing». In: () (cit. on pp. 7–9).

- [10] *Documentation selenium*. <https://www.geekandjob.com/wiki/selenium>. Accessed: 2023-02-08 (cit. on p. 12).
- [11] *Xpath*. <https://www.guru99.com/xpath-selenium.html>. Accessed: 2023-02-06 (cit. on p. 14).
- [12] *Documentation Appium*. <https://appium.io/docs/en/about-appium/intro/>. Accessed: 2022-01-09 (cit. on p. 15).
- [13] *Documentation Cucumber*. <https://www.geekandjob.com/wiki/cucumber>. Accessed: 2023-02-04 (cit. on p. 19).
- [14] *Context Diagram*. <https://miro.com/blog/context-diagram/>. Accessed: 2023-02-04 (cit. on p. 22).
- [15] *Wallbox*. <https://blog.fratelligiacome1.it/wallbox>. Accessed: 2023-02-06 (cit. on p. 23).
- [16] *As is to be*. <https://www.qualitiamo.com/terminologia/analisi>. Accessed: 2023-02-08 (cit. on p. 24).
- [17] *Testing software end to end*. ]<https://vitolavecchia.altervista.org/tipologie-di-testing-software-il-test-end-to-end-end-to-end-testing/>. Accessed: 2023-02-10 (cit. on p. 25).
- [18] *Project Manager*. <https://www.adecco.it/il-lavoro-che-cambia/project-manager>. Accessed: 2023-02-23 (cit. on p. 27).
- [19] *Pagina wikipedia conferenza di Dartmouth*. [https://it.wikipedia.org/wiki/Conferenza\\_di\\_Dartmouth](https://it.wikipedia.org/wiki/Conferenza_di_Dartmouth). Accessed: 2022-12-04 (cit. on p. 45).
- [20] C.E Shannon, M.L Minsky, and J.McCarthy N.Rochster. «A proposal for the Darmouth summer project on artificial intelligence.» In: (1955) (cit. on p. 45).
- [21] *Differenza Ai, ML e DL*. <https://www.advancinganalytics.co.uk/blog/2021/12/15/understanding-the-difference-between-ai-ml-and-dl-using-an-incredibly-simple-example>. Accessed: 2022-12-06 (cit. on p. 45).
- [22] *AI in test automation*. <https://www.browserstack.com/guide/artificial-intelligence-in-test-automation>. Accessed: 2022-01-09 (cit. on pp. 48, 49).

- [23] *Documentazione Healenium*. <https://healenium.io/docs>. Accessed: 2022-12-16 (cit. on pp. 50, 51).
- [24] Artiom Basulto-Lantsova, Jose A. Padilla-Medina, Francisco J. Perez-Pinal, and Alejandro I. Barranco-Gutierrez. «Performance comparative of OpenCV Template Matching method on Jetson TX2 and Jetson Nano developer kits». In: (Jan. 2020) (cit. on p. 59).
- [25] *Template matching OpenCV*. [https://docs.opencv.org/3.4/de/da9/tutorial\\_template\\_matching.html](https://docs.opencv.org/3.4/de/da9/tutorial_template_matching.html). Accessed: 2022-01-15 (cit. on p. 60).
- [26] *APPIUM findByImage*. <https://appium.io/docs/en/advanced-concepts/image-elements/>. Accessed: 2022-01-11 (cit. on p. 60).
- [27] A. Hindle, E. T. Barr, Z. Su M. Gabel, and P. Devanbu. «On the Naturalness of Software». In: (2016) (cit. on p. 71).
- [28] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk. «Toward Deep Learning Software Repositories». In: (2015) (cit. on p. 71).
- [29] A. Vaswani, N. Shazeer, N. Parmar, L. Jones J. Uszkoreit, and A. N. Gomez. «Attention is All You Need». In: (2017) (cit. on p. 71).
- [30] T. Brown, B. Mann, N. Ryder, M. Subbiah, P. Dhariwal J. D. Kaplan, A. Neelakantan, P. Shyam, G. Sastry, and A. Askell. «Language Models are Few-Shot Learners». In: (2020) (cit. on p. 71).
- [31] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. «BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding.» In: (2018) (cit. on p. 71).
- [32] S. Lertbanjongngam, B. Chinthanet, T. Ishio, R. G. Kula, P. Leelaprute, B. Manaskasemsak, A. Rungsawang, and K. Matsumoto. «An Empirical Evaluation of Competitive Programming AI: A Case Study of Alpha-Code». In: (2022) (cit. on p. 71).
- [33] Z. Feng et al. «CodeBERT: A Pre-Trained Model for Programming and Natural Languages». In: (2020) (cit. on p. 71).

- [34] M. Allamanis, M. Brockschmidt, and M. Khademi. «Learning to Represent Programs with Graphs.» In: (2018) (cit. on p. 71).
- [35] M. Chen et al. «Evaluating Large Language Models Trained on Code». In: (2021) (cit. on p. 71).
- [36] *Site of ChatGPT*. <https://openai.com/blog/chatgpt>. Accessed: 2023-02-27 (cit. on p. 71).
- [37] Sajed Jalil, Suzzana Rafi-, Thomas D. LaToza, Kevin Moran, and Wing Lam. «ChatGPT and Software Testing Education:Promises Perils». In: (2022) (cit. on p. 72).