POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Gestionale

Percorso ICT e data analytics per il management

Test Automation e qualità del software nel settore bancario: analisi e risultati di un caso di studio presso Concept Quality Reply

Relatore:

Maurizio Galetto

Candidata:

Carlotta Spadaro

Anno Accademico 2023-2024

Sommario

Indice delle figure

Indice dei grafici

Indice delle tabelle

0. Abstract

Il lavoro di tesi mira a dimostrare come l'adozione di tecniche di Test Automation possa garantire un'elevata qualità del software, offrendo una valida soluzione alla crescente digitalizzazione dei servizi e ai continui cambiamenti. Attraverso un'analisi dettagliata dei vantaggi e delle sfide dell'automazione dei test, si dimostra come tale approccio possa ridurre tempi e costi di esecuzione, massimizzando la qualità del prodotto offerto al cliente. Si sono evidenziate anche le varie tipologie di test automatizzati, la loro applicazione in diversi ambiti quali bancario, automotive e retail e il contributo dell'intelligenza artificiale in tale contesto.

Osservando la realtà aziendale di Concept Quality Reply presso il team di Test Automation per clienti del settore bancario, è stato possibile definire dei contesti in cui l'automazione è conveniente rispetto all'esecuzione manuale dei test, valutando gli aspetti economici in ambedue i casi. Sono state effettuate delle stime riguardo i tempi di esecuzione e le risorse necessarie, analizzando due tipologie differenti di progetti: i test di System, utilizzati per valutare la qualità dell'applicazione prima di metterla in commercio e i testi di non regressione, che servono a garantire la non regressione dell'applicazione a seguito dell'introduzione di nuove funzionalità.

Inoltre, è stata descritta l'esecuzione di un progetto reale di Test Automation sviluppato presso l'azienda ospitante il tirocinio, dagli stadi preliminari di contrattazione con il cliente all'analisi dei risultati ottenuti. Sono state suggerite soluzioni per riorganizzare la suddivisione interna del lavoro e nuovi indicatori di performance per valutare i progetti eseguiti, sulla base delle difficoltà riscontrate durante le esecuzioni.

1. Introduzione ai sistemi di Testing

Il *testing* è una componente fondamentale nel ciclo di vita dello sviluppo di un applicativo software: si confronta il risultato ottenuto con quello atteso e, iterativamente, si apportano modifiche e migliorie al codice in modo da minimizzare l'errore. Il suo ruolo è cruciale nel garantire la qualità e l'affidabilità dell'applicativo poiché identifica difetti e problemi funzionali prima che esso venga distribuito agli utenti finali. Inoltre, ricorrere al *testing* anche in fase di post-produzione garantisce il mantenimento di elevati standard qualitativi e consente di intervenire nel caso di regressione dell'applicativo, a seguito di nuovi rilasci o modifiche.

L'IEEE (*Institute of Electrical and Electronics Engineers*) fornisce una definizione puntuale del *testing* come quel "processo di valutazione di un sistema attraverso strumenti manuali o automatici col fine di determinare se il sistema soddisfa i requisiti specificati oppure se il suo comportamento differisce da quello atteso". Da questa definizione si evince l'esistenza di due metodologie per il *testing*:

- Test manuali: sono eseguiti manualmente dai tester che interagiscono con il software;
- Test automatizzati: i tester si occupano della scrittura degli *script*[1], i quali verranno eseguiti sui dispositivi rendendo i test automatici.

In generale, obiettivi del *testing*, sia esso manuale o automatico, includono:

- Individuazione dei difetti (comunemente chiamati *defect*) e conseguente risoluzione del problema, per incrementare la qualità del software in questione;
- Raccolta di informazioni utili ai diversi attori della catena del valore, tra cui:
    - Sviluppatori: si avvalgono delle informazioni emerse per migliorare la qualità del codice;
    - Manager: ottimizzano la pianificazione dei test, stabilendo i periodi di manutenzione e concordando con il cliente la frequenza di esecuzione sulla base dell'andamento del progetto e degli interventi da effettuare;
    - Committente: riceve una visione chiara dello stato corrente dell'applicativo.
- Incremento della fiducia degli stakeholders nel prodotto e/o azienda in questione, poiché costituisce un ulteriore garanzia di qualità attraverso tecniche di automazione dei test in continua evoluzione.

[1] Con il termine script si intende il codice da scrivere e successivamente eseguire per rendere un test automatizzato.

Per ottimizzare il processo di *testing* deve essere stabilito il giusto *trade-off* fra ambito, tempo e costo[2]:

- Ambito (o *scope*): descrive la porzione di software da testare. Una riduzione dell'ambito comporta certamente un risparmio di tempo, ma può implicare la mancata verifica di alcune funzionalità cruciali per l'applicativo.
- Tempo: disporre di tempistiche maggiori consente di eseguire più test, verificando di conseguenza più sezioni e raggiungendo un livello qualitativo superiore, ma può comportare un incremento dei costi.
- Costo: esso è direttamente influenzato dall'ambito (maggiore è la porzione di software da testare, maggiore è il costo) e dal tempo di esecuzione (maggiore è il tempo necessario, maggiore è il costo). Tuttavia, ridurre ambito e tempo (con conseguente riduzione del costo) può implicare una diminuzione della qualità.

Sulla base di quanto appena esposto, è necessario individuare il giusto punto di equilibrio in relazione alle risorse lavorative, al budget a disposizione, agli standard qualitativi da raggiungere e ai tempi di esecuzione.

Oggigiorno si punta a una riduzione del *Time To Market* o TTM[3], minimizzando pertanto il periodo di tempo che intercorre tra l'ideazione di un prodotto e la sua effettiva distribuzione al cliente finale. Nel caso di un servizio software, è possibile implementare particolari tecniche di *testing* che vadano a ridurre il tempo necessario per il controllo qualità, agevolando il raggiungimento di elevati standard qualitativi.

In conclusione, la fase di "Controllo Qualità" all'interno del ciclo di vita di un applicativo software si realizza tramite l'esecuzione di test, mirati a verificare la corretta funzionalità dell'applicazione e raggiungere determinati standard qualitativi.

Dopo aver descritto brevemente gli obiettivi del *testing* e la sua importanza rispetto al ciclo di vita dello sviluppo software, le sezioni che seguono affronteranno un breve *excursus* storico sulla nascita della pratica di Test Automation (nonché l'automazione dei test, contrariamente all'esecuzione manuale), le sue applicazioni, la valutazione di vantaggi e sfide e, infine, le sue prospettive future.

[2] (Jose, 2021)
[3] La traduzione letterale di Time To Market è "Tempo di mercato"; esso indica il tempo necessario per l'ideazione, realizzazione e verifica di un servizio/prodotto, prima che venga commercializzato.

1.1 Nascita e storia del Test Automation

Il Test Automation è una pratica che prevede l'esecuzione automatica dei test, impiegando specifici strumenti per la scrittura del codice del test e la loro esecuzione in particolari ambienti. Essa si è sviluppata a seguito della transizione dal modello *Waterfall* al modello *Agile*, avvenuta intorno agli anni '60 del secolo scorso: tali metodi descrivono due metodologie completamente differenti per lo sviluppo di un progetto di realizzazione di un software. In particolare, il primo adotta un approccio sequenziale che consiste nell'esecuzione di una serie di fasi predefinite e ordinate tale che ogni fase deve essere completata prima che la successiva possa iniziare. Questo contesto è caratterizzato da un'ampia e dettagliata documentazione da redigere e rispettare, che funge da guida per le successive fasi e, inoltre, presenta una bassa flessibilità poiché i cambiamenti dei requisiti funzionali e non funzionali[4] del progetto sono difficili e costosi.[5]

Al contrario, il modello *Agile* adotta un approccio caratterizzato da continui cambiamenti ed evoluzione dei requisiti: il software viene sviluppato in modo iterativo e incrementale attraverso cicli di sviluppo brevi denominati *sprint*. In un tale contesto è centrale la collaborazione all'interno del team e il *feedback* continuo fornito dal cliente, per comprendere le sue necessità e inserirle all'interno del progetto. Difatti, si tratta di una metodologia flessibile dove i cambiamenti sono visti come un'opportunità e non come un costo, al fine di massimizzare la soddisfazione del cliente e adattare il prodotto alle sue esigenze.[6]

Durante i primi anni '60 si è diffusa la pratica del *Test-First Development* che prevede la progettazione dei test prima della scrittura del codice per le nuove funzionalità; l'obiettivo è la massimizzazione della qualità dell'applicativo, grazie alla possibilità di eseguire i test mentre il software viene realizzato e ciò permette di individuare e correggere gli errori già durante la fase di sviluppo. Si tratta di un primo importante traguardo nella storia del movimento *Agile*, poiché comincia a perdere valenza il modello rigoroso, sequenziale e scientifico a favore dell'idea secondo cui lo sviluppo del software è un'attività creativa e non meccanica, in continua evoluzione. Sebbene nello sviluppo informatico è centrale l'aspetto tecnico, sono anche fondamentali le *soft-skills* di ciascun individuo all'interno del team di sviluppo e *testing*. È da qui che ha origine la transizione da modello *Waterfall* ad *Agile*.[7]

[4] I requisiti funzionali riguardano le funzionalità che l'applicativo deve rispettare; i requisiti non funzionali riguardano altri aspetti quali le performance, l'affidabilità e così via.
[5] (Sommerville, 9th edition - 2011)
[6] (Beck, 2001)
[7] (Sommerville, 9th edition - 2011)

Intorno agli anni ’90 si verifica il cosiddetto «boom di Internet», che determina la necessità di una nuova ed efficace metodologia di sviluppo.[8] Tale bisogno è emerso a seguito delle valutazioni qualitative poco soddisfacenti dei software sviluppati sino a quel momento: si notò che essi non esaudivano pienamente le richieste e aspettative degli utenti finali e tali mancanze furono associate a errori commessi nelle fasi di realizzazione dell’applicativo, ovvero durante la stesura del codice e lo sviluppo dei test.

Le modalità e la frequenza di esecuzione di attività e processi caratterizzanti il ciclo di sviluppo di un software dipendono dall’approccio utilizzato. Tali attività e processi possono essere descritti come segue:

a) Definizione dei requisiti funzionali e non funzionali: si stabiliscono gli aspetti da implementare.
b) Pianificazione dello sviluppo: dopo aver analizzato e ideato il progetto, si elabora una schedulazione dettagliata per il suo sviluppo.
c) Implementazione del software: si esegue la codifica e, dunque, viene realizzato l’applicativo.
d) Collaudo: sono i cosiddetti cicli di *testing* in fase di pre-produzione, in cui si verifica che i requisiti funzionali stabiliti nel primo punto siano stati correttamente realizzati, prima del rilascio in produzione. In caso contrario, il codice viene opportunatamente modificato e migliorato.
e) Rilascio: l’applicativo viene messo in commercio e reso disponibile agli utenti finali.
f) Manutenzione: le funzionalità del software devono essere mantenute stabili nel tempo, anche in seguito ad aggiornamenti e implementazione di nuove funzionalità dell’applicativo. Si valuta pertanto la non regressione dell’applicazione, tramite cicli di test definiti come No Regression Test (NRT).

A seconda del modello di implementazione utilizzato, le fasi sopra descritte possono avere tempi e tipologie di esecuzioni differenti. Di seguito analizzeremo i principali modelli impiegati nel corso del tempo.

[8] (Filippetti, 2017)

Prima degli anni ‘90 veniva utilizzato principalmente il cosiddetto «modello a cascata», nonché *Waterfall*; esso prevede l’esecuzione di una serie di fasi consecutive prestabilite ma ciò determina il protrarsi dell’errore da una fase a tutte le successive (Figura 1.1) e, pertanto, l’accumularsi di errori. Tale modello è adatto per realizzare piccoli progetti software, dove è possibile stabilire i requisiti e i processi da eseguire in maniera quasi definitiva già in fase iniziale, senza possibilità di apportare modifiche in corso d’opera.

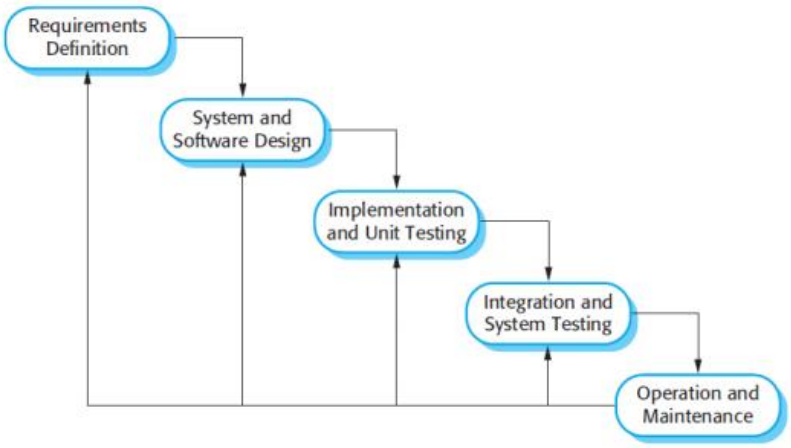


Figura 1.1 - Modello a cascata.
Fonte: (Pagano, 2020)

In Figura 1.1 sono riportate le fasi da eseguire e il rispettivo ordine. In un primo momento è necessario definire i requisiti per la realizzazione e lo sviluppo del progetto; successivamente, si realizza l'applicativo, sia da un punto di vista di grafica e design (nonché l’interfaccia con cui l’utente comunica, detto anche *front-end*) che di interazione (indicato come *back-end*). Una volta sviluppata l’applicazione, si prosegue con la fase di *testing*; essa prevede l’esecuzione di due principali tipologie di test:

- Test di unità: verificano la corretta funzionalità di singole unità sviluppate, dunque solo alcune parti di codice specifiche.
- Test di integrazione: verificano il corretto funzionamento dell’intera applicazione, pertanto come i diversi moduli comunicano fra loro.

Infine, l’ultima fase prevede la commercializzazione dell’applicativo, che lo rende disponibile al cliente finale. Solo in un secondo momento, a seguito di segnalazioni da parte dell’utente oppure di nuovi rilasci del software, verrà eseguita la manutenzione dell’applicazione per ottimizzare la qualità e valutare la presenza di regressioni a seguito di nuove modifiche.

Una piccola evoluzione del modello a cascata è costituita dal modello a prototipi, che continua a seguire un approccio sequenziale (*Waterfall*) ma aggiunge il concetto di prototipo, nonché software semi completo che si evolve in modo incrementale siano ad arrivare alla versione completa dell'applicativo.

Maturata l'esigenza di un approccio non sequenziale, si diffonde il *modello a spirale*: in questo caso le diverse fasi del progetto vengono svolte parallelamente in modo ciclico. Al termine di ciascun ciclo, è necessario che il team si allinei e, solo dopo, è possibile iniziare con un ulteriore ciclo. In tal modo si minimizza l'errore a ogni ciclo, convergendo alla fine del progetto a una soluzione ottimale.

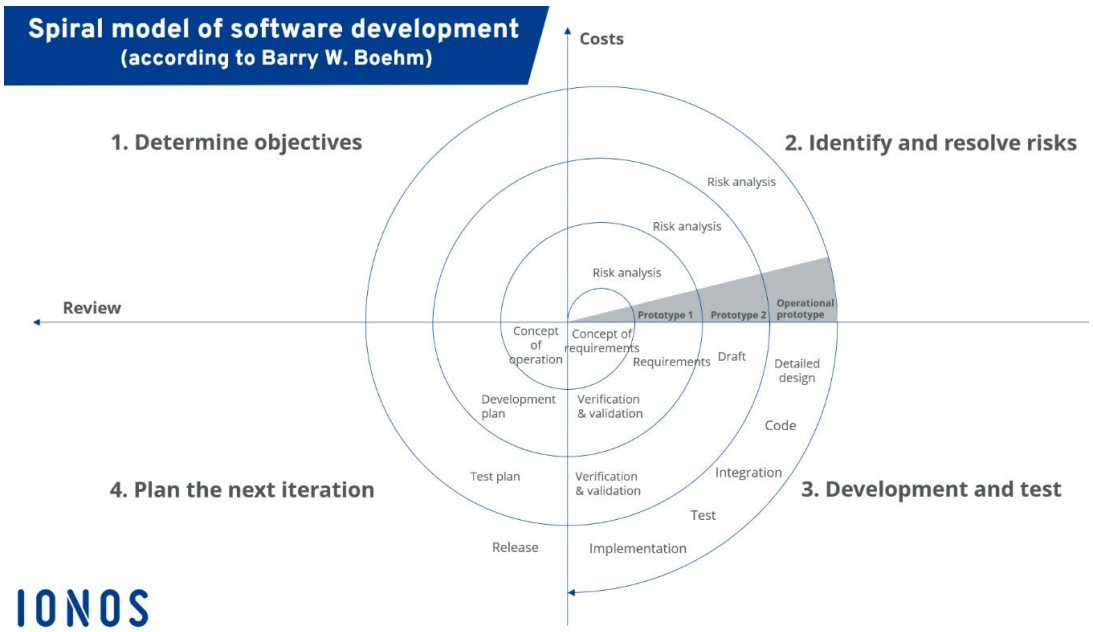


Figura 1.2 - Modello a spirale.
Fonte: (IONOS, 2023)

Dalla Figura 1.2 si evidenzia la natura ciclica di tale approccio: ciascun ciclo è costituito da 4 fasi, al termine del quale vengono realizzati dei prototipi che fungeranno da input al ciclo successivo. La prima fase prevede la definizione degli obiettivi di progetto, per quel momento specifico e con quel prototipo, come ad esempio la definizione dei requisiti o la pianificazione delle attività. La seconda fase prevede l'identificazione, l'analisi e la risoluzione dei rischi: si sviluppa pertanto una strategia che minimizza sia costi che rischi. Successivamente si passa alla progettazione dettagliata, alla scrittura del codice e all'esecuzione dei test: in questa fase si procede alla verifica e validazione continua di ciascun componente del software per verificare il suo corretto funzionamento e la sua corretta integrazione con il resto dell'applicazione. Infine, il team pianifica l'iterazione successiva

sulla base dei *feedback* ricevuti durante le fasi precedenti, degli esiti dei test e del risultato dell'analisi dei rischi.[9]

Dagli anni '90 in poi cominciò a essere utilizzato un approccio *Agile*: esso prevede l'esecuzione dei progetti tramite processi snelli e in continua evoluzione, una buona collaborazione all'interno del team e promuove la comunicazione diretta con il cliente.[10] Differisce dal modello a spirale poiché basato sulla collaborazione e comunicazione, priva di eccessivi formalismi e di schedulazioni predefinite da seguire a priori. Secondo Crispin e Gregory, gli autori di "*Agile Testing: A Practical Guide for Testers and Agile Teams*", *testing Agile* significa testare un applicativo seguendo un piano approssimativo per comprenderne a pieno le sue caratteristiche e lasciare che i *feedback* dei clienti guidino il processo di *testing*. In particolare, seguire un approccio di *testing Agile* implica rispettare le seguenti fasi:

a) Osservare e studiare il quadro generico;
b) Collaborare con il cliente;
c) Costruire delle basi solide per la metodologia *Agile*, basata sulla flessibilità;
d) Fornire e ottenere *feedback* dal cliente, per essere correttamente indirizzati;
e) Automatizzare e, dunque, ottimizzare i test;

I punti appena descritti costituiscono il nuovo approccio alla metodologia *Agile*.[11] [12]

L'autore Brian Marick, in "*Everyday Scripting with Ruby: For Teams, Testers, and You*"[13] mostra gli scopi differenti dei diversi test e la convenienza per la loro automazione o meno. In Figura 1.3 viene mostrato un quadro riassuntivo in cui i test vengono categorizzati in funzione di due assi: lo scopo dei test, mostrato in asse verticale, e la loro natura, riportata in asse orizzontale. Per quanto riguarda lo scopo, esso può essere «a supporto del team», nonché test che supportano la crescita del team verificando le corrette funzionalità dei codici sviluppati internamente, oppure «per criticare il prodotto», ovvero quei test che mirano a valutare la qualità del prodotto. Relativamente alla natura del test, esso può essere «orientato al business» e dunque serve a verificare che il software soddisfi i requisiti di business oppure «orientato alla tecnologia» che valuta gli aspetti tecnici. Inoltre, per ciascun test viene indicata la convenienza o meno per l'automazione. Di seguito verranno analizzati nel dettaglio i quadranti mostrati in Figura 1.3.

[9] (IONOS, 2023)
[10] (Muzzì & Caramellino, 2007)
[11] (Crispin. & Gregory, 2009)
[12] (Collins, Dias-Neto, & de Luc, 2012)
[13] (Marick, 2006)

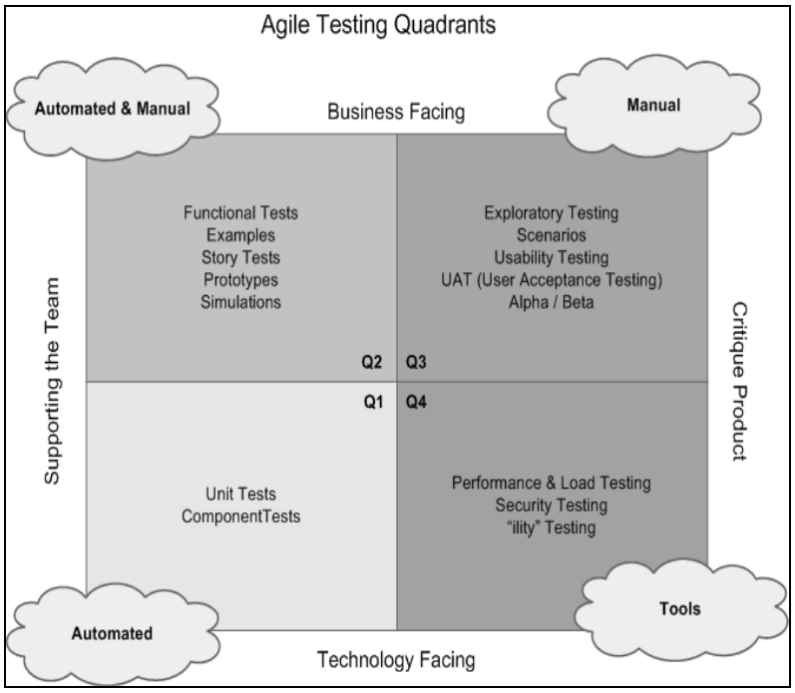


Figura 1.3 - Quadranti dell'Agile testing.
Fonte: (Marick, 2006)

- Q1: si tratta di test orientati alla tecnologia e a supporto del team, quali gli *unit test* oppure i *component test*. I primi verificano il funzionamento di una singola unità di codice, ad esempio una funzione, un metodo o una classe; i secondi hanno granularità maggiore rispetto agli *unit test* e verificano il corretto funzionamento di un intero componente. In generale, nel quadrante Q1 si collocano quei test che verificano che la sezione specifica testata svolga esattamente quanto voluto dal programmatore; è conveniente automatizzarli poiché spesso ripetitivi e frequenti.
- Q2: comprende i test orientati al business e a supporto del team, come i test funzionali, i prototipi oppure le simulazioni. Essi vengono identificati come i test dei requisiti aziendali, attraverso i quali si verifica che le specifiche di business delineate a inizio progetto siano state rispettate. È bene automatizzare questi test poiché spesso complessi e di lunga esecuzione, ma è fondamentale mantenere la loro esecuzione manuale se risulta necessaria la supervisione umana.
- Q3: si tratta di test orientati al business per criticare il prodotto, difatti vengono identificati come i test dei *defect* aziendali; esempi sono i test esplorativi, gli scenari di test di usabilità, i test di accettazione lato utente (UAT) e i test alpha o beta (i primi sono svolti in fase iniziale da parte dei tester o sviluppatori mentre i secondi coinvolgono un gruppo di utenti finali in condizioni reali di utilizzo del software). Data la natura di questi test, è necessario che siano svolti manualmente dai soggetti indicati.

- Q4: racchiude i test orientati alla tecnologia per criticare il prodotto, quali i test di sicurezza, di scalabilità oppure i test performance e carico; essi vengono identificati come i test dei difetti tecnici, sono solitamente automatizzati e prevedono l'utilizzo di un sottoinsieme di tool specifici del Test Automation.[14]

1.2 Applicazioni del Test Automation

Il Test Automation costituisce oramai una componente fondamentale per lo sviluppo del software per i diversi ambiti applicativi. Si tratta di uno strumento essenziale per garantire qualità elevata e affidabilità al cliente. La sua applicazione è differente fra i vari settori industriali, ognuno dei quali ha esigenze specifiche.

Per quanto riguarda le topologie di test automatizzati, esse sono varie e ognuna presenta funzionalità differenti. Di seguito ne vengono riportate alcune:

- *No Regression Test*:
  essi verificano che le nuove modifiche introdotte nel software non compromettano le funzionalità preesistenti. Si tratta di test ripetuti a cadenze regolari e, in quanto tali, è particolarmente conveniente la loro automazione al fine di ridurre tempi e costi per la loro esecuzione.
- *Smoke Test*:
  si tratta di test immediati e rapidi, il cui scopo consiste nel verificare le funzionalità basilari dell'applicativo, come ad esempio l'autenticazione, la navigazione fra le schermate principali o le operazioni basilari. Solitamente vengono svolti giornalmente poiché solo una volta superati questi test, è possibile proseguire con test più approfonditi. Ad esempio, se non è possibile accedere all'applicazione, allora non è possibile svolgere alcun test ulteriore; viceversa, verificato il corretto funzionamento dell'autenticazione allora sarà possibile procedere con altri test specifici che testano varie funzionalità.
- *API testing*:
  viene verificata la corretta funzionalità delle interfacce di programmazione delle applicazioni, nonché le API; esse permettono la comunicazione fra diversi software. Tali test analizzano i parametri standard di risposta (come ad esempio nel caso di un HTTP Rest, il codice 200 indica esito positivo) e il body stesso di risposta.

[14] (Collins, Dias-Neto, & de Luc, 2012)

- *System Test*:

  si tratta di test eseguiti in ambienti dedicati alla verifica e validazione del software prima del rilascio in produzione; tali ambienti simulano l'ambiente di produzione reale e sono solitamente isolati dall'ambiente di sviluppo, al fine di garantire l'esecuzione dei test senza interferenze. Di conseguenza, con questi test si vuole verificare il corretto funzionamento del sistema nella sua integrità, prima di rilasciarlo in produzione. Poiché solitamente viene effettuata una sotto selezione dei System Test per eseguirli come No Regression Test a seguito della messa in produzione dell'applicativo, è bene automatizzarli in modo tale da avere già a disposizione gli *script* e risparmiare tempo di esecuzione in fase di NRT.
- *Cross-Platform e Cross-Browser*:

  si tratta di test da svolgere su diverse piattaforme (ad esempio, Windows, macOS, Linux per desktop oppure iOS, Android per mobile) e diversi browser (ad esempio, Chrome, Safari, etc.). L'obiettivo è garantire il corretto funzionamento dell'applicativo, indipendentemente dalla piattaforma o dal browser utilizzato. In tal modo si verifica che il software testato sia accessibile e funzionale alla maggior parte degli utenti e non solamente a coloro che utilizzano determinati dispositivi o browser. Poiché viene richiesta l'esecuzione dei medesimi test su un numero elevato di piattaforme e/o browser, sarebbe ottimale automatizzare i test per parallelizzare le esecuzioni e ridurre costi e tempi.
- *Test di carico e performance*:

  sono test mirati a verificare che l'applicativo funzioni correttamente se sottoposto a determinate condizioni di carico. Si valutano diversi aspetti come, ad esempio, la scalabilità del sistema, cioè come esso risponde al crescere degli utenti collegati in contemporanea, oppure i tempi di velocità di elaborazione e risposta del sistema in presenza di elevato carico.
- *Test di sicurezza*:

  lo scopo è identificare e valutare le vulnerabilità del sistema informatico, per poi risolverle opportunamente. Vengono simulati attacchi oppure si verificano la conformità del sistema alle politiche di sicurezza.

L'implementazione delle diverse tipologie di test sopra citate varia in base al settore di applicazione poiché ciascuno di essi ha esigenze e priorità differenti. Di seguito verranno analizzati tre dei principali ambiti applicativi del Test Automation, nonché *automotive*, *retail* e *finance*.

Nel settore *automotive* la qualità e la sicurezza del software sono cruciali poiché influiscono indirettamente sulla sicurezza di conducenti e passeggeri. Pertanto, è fondamentale svolgere in maniera consistente e adeguata i testi di sicurezza che potrebbero prevenire attacchi informatici

tali da invalidare il software e, dunque, compromettere la sicurezza del veicolo. L'aspetto della sicurezza deve essere verificato anche nel caso di modifiche apportate al software; pertanto, è opportuno eseguire Test di Non Regressione per garantire che modifiche successive non influenzino il corretto funzionamento dell'applicativo. Infine, è fondamentale accertarsi che le API funzionino correttamente per garantire che i diversi componenti del software comunichino adeguatamente (ad esempio, i sistemi di *infotainment*[15] e l'unità di controllo del motore[16]).

Altro ambito applicativo è il settore *retail* ed *eCommerce*, in cui il Test Automation viene utilizzato per garantire che i siti web o le applicazioni funzionino correttamente in presenza di elevato carico. Difatti, i test di carico e di performance garantiscono che l'eCommerce sia in grado di mantenere elevate prestazioni anche in presenza di picchi di traffico in occasione di particolari eventi o momenti della giornata. Inoltre, bisogna verificare anche la corretta interazione fra sistemi di pagamento, inventario e logistica e ciò viene garantito con lo svolgimento dei test API. Infine, per garantire un'esperienza utente ottimale è fondamentale svolgere i test Cross-Platform e Cross-Browser, al fine di massimizzare l'esperienza di acquisto del cliente e, pertanto, la sua soddisfazione qualunque sia il dispositivo utilizzato.

L'ultimo ambito analizzato in cui il Test Automation viene implementato per garantire la qualità del software è quello *finance*, in cui ci si focalizza sugli istituti bancari e le rispettive applicazioni. Anzitutto, è bene garantire tramite *smoke test* che le funzionalità base dell'applicativo siano disponibili, per poi proseguire con test più approfonditi e specifici. Inoltre, l'esecuzione dei test di sicurezza è fondamentale per proteggere i dati sensibili degli utenti e prevenire attacchi al sistema, che potrebbero agevolare le frodi bancarie. Bisogna anche preservare la corretta funzionalità dell'applicativo ogni qualvolta venga introdotta una nuova funzionalità e ciò è reso possibile dai No Regression Test.

1.3 Vantaggi e sfide del Test Automation

Come già discusso nei capitoli precedenti, al giorno d'oggi il Test Automation costituisce una componente essenziale per il ciclo di vita e manutenzione del software, garantendo elevata qualità e affidabilità dell'applicativo, soprattutto in un contesto di continui cambiamenti e rapidi rilasci. Tuttavia, come ogni strumento, l'automazione dei test presenta dei vantaggi ma anche delle

[15] Si tratta di sistemi che integrano l'informazione con l'intrattenimento; un esempio pratico riguarda gli schermi nei veicoli.
[16] Si tratta di una serie di sensori e di attuatori che garantiscono il funzionamento di vari aspetti del veicolo.

limitazioni. Comprendere sia gli aspetti positivi che negativi è fondamentale per massimizzare i benefici di tale strumento e per implementarlo adeguatamente.

Come accennato in precedenza, l'obiettivo principale del Test Automation è il risparmio in termini di tempo e costi per l'esecuzione di test che, se svolti manualmente, necessiterebbero di un numero elevato di risorse e, di conseguenza, tempistiche e costi maggiori. Realizzarli automaticamente permetterebbe la loro esecuzione parallela, pertanto lo svolgimento di un numero maggiore di test a parità di tempo e una copertura più completa del codice e, dunque, del software.

Il *World Quality Report* 2020-2021 pubblicato da Capgemini, in collaborazione con Sogeti e Micro Focus fornisce dati significativi in termini di risparmio di tempi e costi a seguito dell'implementazione di tecniche di Test Automation: si ha una riduzione che varia dal 30% al 50% per quanto riguarda i tempi di esecuzione dei test e un risparmio del 25-30% nei costi operativi associati ai test. La riduzione dei tempi è di immediata comprensione: è più veloce svolgere un test automatico piuttosto che farlo manualmente, qualora sia presente già il codice del test. Il risparmio in termini di costi, invece, è dovuto a diversi aspetti quali minori ore di lavoro da retribuire, una conseguente riduzione del personale oppure la possibilità di individuare preventivamente un numero maggiore di errori, minimizzando gli interventi successivi.[17]

Un aspetto da non sottovalutare è la complessità dei casi che l'automazione può gestire: basti pensare ai No Regression Test, che dovranno essere eseguiti ad intervalli regolari rispettando le medesime condizioni. In un tale contesto, reperire i dati necessari per l'esecuzione del test (come, ad esempio, i dati correttamente censiti per effettuare un bonifico nel caso di test per un'applicazione bancaria) oppure il flusso da seguire potrebbe essere non immediato. Con l'automazione, invece, rimane tutto riportato nel codice del test e il flusso viene eseguito automaticamente. Ciò implica anche l'eliminazione del rischio di errore umano, garantendo che i test siano compiuti allo stesso modo anche a distanza di tempo. In tal senso, si garantisce un'esecuzione coerente dei test, mantenendo le stesse condizioni indipendentemente dai diversi momenti di attuazione.

Una soluzione particolarmente vantaggiosa permessa dai test automatizzati, e non replicabile con i test manuali senza costi aggiuntivi, consiste nella possibilità di eseguirli in maniera automatica e sequenziale durante le ore notturne. È evidente come ciò garantisce un notevole vantaggio in termini di tempo risparmiato, poiché la mera esecuzione dei test avverrebbe al di fuori delle ore lavorative, lasciando queste ultime interamente dedicate alla stesura del codice o alla correzione di

[17] (Capgemini, Sogeti, & Micro Focus, World Quality Report, 2020-2021)

difetti. Tale pratica porta ad un incremento del numero giornaliero di test scritti ed eseguiti. Nel caso in cui l'esecuzione dei test sia interna all'azienda e non commissionata ad aziende terze, l'esecuzione notturna risulterebbe più semplice da implementare. Nel caso di esecuzione affidata ad altre aziende, tale circostanza non è sempre permessa poiché a causa di regolamentazioni e politiche interne si preferisce autorizzare l'accesso agli ambienti di test solamente in orario lavorativo.

Dopo aver analizzato i vantaggi che l'automazione dei test garantisce, è fondamentale considerare anche le complessità per la sua implementazione. Essa non è immediata poiché richiede adeguate conoscenze e preparazione; in primo luogo, è necessario predisporre un ambiente di *testing* appropriato. Successivamente, è necessario avere a disposizione i dati necessari all'esecuzione dei test, forniti da un apposito team che si occupa per l'appunto di *Data Preparation* (DP). Oltre a questi aspetti implementativi e pratici, è necessario disporre anche di tester con elevata competenza per la creazione di *script* adeguati ai casi di test e consistenti nel tempo.

Un altro aspetto da valutare riguarda gli strumenti e i tools utilizzati per realizzare gli *script*, poiché a seconda delle diverse tipologie è possibile eseguire controlli differenti e non tutte le circostanze sono sempre automatizzabili. Ad esempio, con la maggior parte dei tools oggigiorno utilizzati non è possibile testare aspetti grafici come i colori del *front-end*. Questa tipologia di controlli, pertanto, dovrà essere eseguita manualmente dal tester.

Di conseguenza, l'implementazione di test automatizzati richiede investimenti iniziali elevati, sia per la formazione dei tester che per la predisposizione dell'ambiente di *testing* oppure l'acquisto dei dispositivi necessari.

Grazie all'esperienza maturata in azienda, ritengo sia fondamentale che la suddivisione del lavoro e dei progetti all'interno di un team di Test Automation debba essere ben studiata: nella realtà, soprattutto nel caso di aziende di consulenza in cui la mole di lavoro e di progetti è consistente, si tende ad assegnare il lavoro in funzione della disponibilità delle risorse in quel determinato momento. A mio avviso, sarebbe più efficiente optare per un'assegnazione che guarda alle competenze e all'interesse di ciascun tester, al fine di avere soggetti mirati che analizzano e valutano il prodotto in modo obiettivo, approfondito e con uno spiccato interesse.

Infine, trattandosi di test effettuati su software in continua evoluzione e soggetti a frequenti rilasci, è necessario effettuare manutenzione dei test per diverse ragioni: il flusso di esecuzione potrebbe cambiare a causa di modifiche nel layout o nelle logiche dell'applicativo oppure, a seguito del rilascio di nuove versioni (*build*), gli elementi mappati potrebbero cambiare il loro xPath, ossia l'identificativo di riferimento per il test automatico, causando l'incapacità durante l'esecuzione di

interagire con gli elementi mappati.[18] Per illustrare questo concetto più chiaramente supponiamo l'esistenza di un semplice test su una qualsiasi applicazione, che prevede l'analisi della sezione del profilo personale dell'utente. Ipotizzando che uno dei punti di aggancio di tale sezione segua il flusso mostrato in Figura 1.4:

Figura 1.4 – Flusso sezione profilo personale.

La sezione "Altro" potrebbe essere identificata come un *button* e, dunque, avere un xPath del tipo:

//XCUIElementTypeButton[@name="Altro"]

oppure potrebbe essere mappata sottoforma di icona o immagine come segue:

//XCUIElementTypeImage[@name="Altro"]

Tramite la seguente istruzione:

And I click NomeClasse.nomeAttributo

Il test automatizzato cercherà l'elemento definito nella classe "*NomeClasse*" univocamente identificato come "*nomeAttributo*" e lo cliccherà nella schermata dell'applicazione; se l'xPath cambia, allora non si troverà più quell'elemento nella pagina e, dunque, il test fallirà. Nell'esempio in questione, supponendo che l'elemento "Altro" sia definito come un *button* nella classe "AltroView" con il nome "altro", l'istruzione sarà la seguente:

And I click AltroView.altro

Fin quando la versione su cui viene lanciato il test identifica "Altro" come un pulsante, allora il test avrà successo; nel caso di una nuova versione che considera "Altro" come un'icona, allora il test fallirà perché *AltroView.altro* è definito come un *button* e non come un'immagine.

In conclusione, sebbene quest'ultima limitazione rappresenti una sfida significativa per l'applicazione del Test Automation è importante sottolineare che esistono ricerche innovative basate sull'intelligenza artificiale, mirate a superare tali problematiche. Nel capitolo successivo tale argomento verrà trattato in maniera più approfondita.

[18] (Mohammad, 2015)

Una soluzione immediata al problema appena descritto può essere la seguente: qualora si voglia indentificare un oggetto che ha come nome “Altro”, qualunque sia la sua natura (bottone, icona o altri elementi) è possibile scrivere l’xPath come segue:

//[@name=”Altro”]*

Dove con il simbolo di asterisco si intende il caso generico; tuttavia, questa soluzione non è sempre efficiente poiché può accadere che nella stessa pagina esistano più elementi con l’attributo *name* valorizzato come “Altro”. Nel caso del pulsante, è possibile che la struttura ad albero identificativa sia la seguente, mostrata in Figura 1.5.

Figura 1.5 – Possibile rappresentazione gerarchica di un button.

La Figura 1.5 mostra una possibile struttura gerarchica di un *button* all’interno di una schermata: vi è l’elemento generico appartenente alla classe dei pulsanti, a sua volta costituito sia da un testo che da un’icona. Tutti e tre gli elementi descritti avranno l’attributo *name* pari ad “Altro” e ciò che li differenzia è solamente la classe di appartenenza. Pertanto, se l’xPath identificativo di “Altro” nella classe “AltroView” è definito in modo generico con l’asterisco, allora durante l’esecuzione del test verranno trovati ben 3 elementi, di conseguenza il test fallirà poiché nella pagina quell’elemento non è univocamente identificato.

Un’ulteriore conferma riguardo le sfide del Test Automation fin ora descritte si ha analizzando i risultati del sondaggio mostrati nel Grafico 1.1, riportato nel *World Quality Report* 2023-2024. Esso riporta le percentuali di risposta alla domanda “Quali sono le maggiori sfide affrontate dagli ingegneri della qualità nei progetti Agile?”; tale quesito è stato posto a 315 individui provenienti da diverse località, con prevalenza di USA, Francia, Germania e UK. Le possibili risposte sono state le seguenti:

- Mancanza di tempo per testare o automatizzare;
- Mancanza di competenze sul codice;

- Mancanza di supporto nei processi di *testing*;
- Limitata carriera per i tester;
- Mancanza di conoscenza riguardo le tecniche *Agile*.

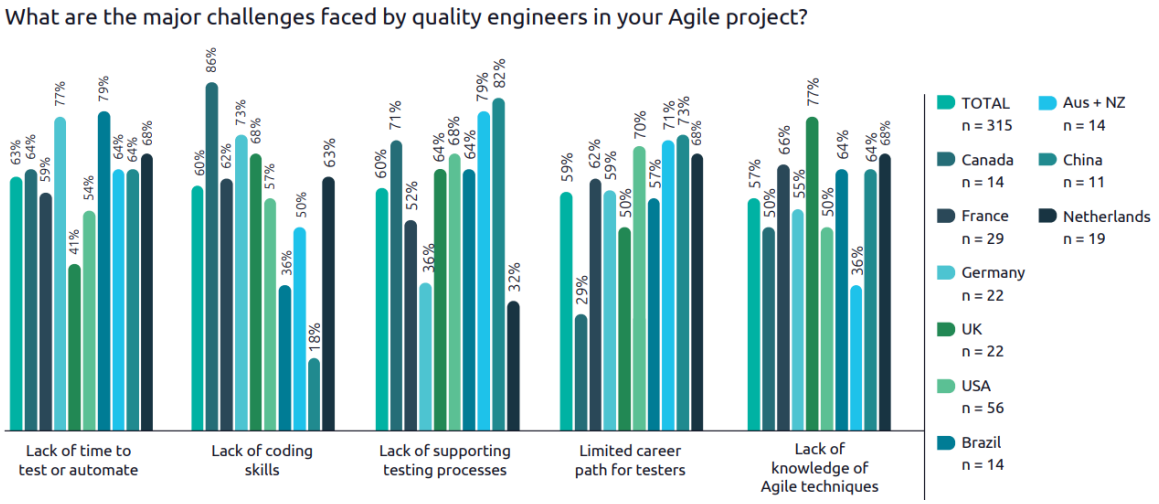


Grafico 1.1 - Risultato sondaggio riguardo le sfide del TA.
Fonte: (Capgemini, Sogeti, & OpenText, World Quality Report, 2023-2024)

Dai risultati ottenuti è possibile notare che, nonostante le differenze territoriali, le sfide sulla Quality Assurance tramite Test Automation siano comuni. La mancanza di tempo e la necessità di elevate competenze tecniche rappresentano i principali ostacoli per garantire elevati standard qualitativi del software tramite l'esecuzione di test automatizzati. A contrastare e mitigare questi problemi aiutano i numerosi investimenti nella formazione e nello sviluppo delle competenze dei tester, oltre all'adozione di processi di supporto più robusti.[19]

1.4 Prospettive future del Test Automation e il contributo dell'AI

Al giorno d'oggi il Test Automation è una pratica sempre più utilizzata per garantire qualità ed efficienza dell'applicativo. Poiché si assiste a continui e rapidi cambiamenti, l'implementazione di tali tecniche richiederebbe una costante manutenzione e aggiornamento dei codici dei test; per tali ragioni il contributo dell'intelligenza artificiale nello sviluppo e nel *testing* del software costituisce un oggetto di ricerca fondamentale.

Come accennato, la sfida maggiore dell'automazione dei test riguarda l'ambito della manutenzione: se il software aumenta la sua complessità e aggiunge nuove funzionalità allora sarà necessario scrivere nuovi test ed effettuare manutenzione sui precedenti. Studi recenti affermano che circa il

[19] (Capgemini, Sogeti, & OpenText, World Quality Report, 2023-2024)

40% del tempo dedicato dai tester per il processo di esecuzione dei test è impiegato per la loro manutenzione.[20]

Gli studi sull'intelligenza artificiale stanno cambiando le dinamiche del *testing* del software, sia manuale che automatico. Invece di eseguire i test manualmente, ci si sta orientando verso l'automazione tramite robot, i quali eseguiranno i test al posto delle persone. Sicuramente il machine learning per i robot richiederà il contributo umano, ma esso sarà minimo. In un mondo ideale, dove il *testing* sarà quasi interamente automatizzato, le nuove tecnologie forniranno risultati migliori rispetto a quanto sviluppato dai team di *testing* esistenti al giorno d'oggi.[21]

Di seguito verranno analizzati dettagliatamente i vantaggi riguardo l'utilizzo dell'intelligenza artificiale nel *testing* del software.

Figura 1.6 - Benefici dell'intelligenza artificiale nel testing del software.
Fonte: (Battina, 2019)

La Figura 1.6 racchiude alcuni dei vantaggi nell'automatizzare i test, già in parte delineati in precedenza, fra cui: validazione, incremento dell'accuratezza, maggiore copertura dei test, risparmio di tempo, *effort* e denaro, riduzione del *Time To Market* e, infine, riduzione degli errori.

Di seguito verranno approfonditi alcuni aspetti e come l'intelligenza artificiale può contribuire:

- La raccolta dei dati necessari per l'esecuzione dei test è un aspetto da non sottovalutare, che richiede notevole tempo da parte dei tester per trovare gli input adeguati con cui effettuare il test. Ad esempio, supponiamo il caso in cui il test richieda la verifica delle funzionalità dell'applicazione per un utente con saldo negativo: è necessario effettuare l'autenticazione e svolgere il test utilizzando l'utenza (o User ID) che presenta tale condizione di saldo. Poiché gli ID sono condivisi tra i vari team di *testing* e sviluppo

[20] (Jordan, Maurer, Lowenberg, & Provost, 2019)
[21] (Battina, 2019)

dell'azienda, le loro caratteristiche possono subire variazioni. Ad esempio, se un altro tester utilizza l'ID con saldo negativo per verificare l'incremento del residuo a seguito di un bonifico ricevuto, quell'utenza non avrà più saldo negativo e, pertanto, non potrà più essere utilizzato per il test sulle funzionalità di un utente con saldo negativo. Questa variabilità nelle caratteristiche possedute da ciascun ID evidenzia l'importanza di un sistema basato su intelligenza artificiale, in grado di identificare gli ID appropriati per ciascun test, tenendo traccia delle modifiche subite e consentendo un significativo risparmio di tempo per i tester.

- Gli sviluppatori potrebbero pensare di eseguire test automatizzati ogni qualvolta effettuano una modifica del codice sorgente e, con il contributo di sistemi di intelligenza artificiale, si potrebbero avere segnalazioni e spiegazioni dettagliate riguardo le cause di fallimento del test. Ad esempio, se a seguito della modifica del codice dell'applicazione un test fallisce poiché non trova più la corrispondenza di un xPath all'interno della pagina, tale informazione potrebbe essere direttamente trasmessa ai tester i quali, in modo mirato e preciso, effettuano la manutenzione del test. Pertanto, ciò garantirebbe un'individuazione immediata degli errori, risparmiando tempo e rendendo più sicuro il sistema. In aggiunta, oltre all'identificazione degli errori, si potrebbe integrare un sistema di AI che corregga automaticamente gli errori e sovrascriva lo *script* preesistente.
- Con i test automatizzati, è possibile eseguire un numero maggiore di test rispetto al caso manuale, a parità di tempo. L'intelligenza artificiale potrebbe svolgere anche un ruolo di ottimizzazione, migliorando ed efficientando gli *script* oppure prioritizzando l'esecuzione dei test, selezionando i casi più rilevanti da eseguire in modo prioritario rispetto agli altri.[22]

L'automazione del *testing* del software sta avendo un impatto rilevante in particolar modo negli Stati Uniti, promettendo crescite elevate per l'industria del software. Secondo un'analisi condotta da InfoWorld, l'88% delle imprese implementa l'automazione per più del 50% dei propri test, il che si traduce in cicli di *testing* più rapidi, un aumento del 71% nella copertura dei test e un miglioramento del 68% nella capacità di rilevamento dei problemi.[23] [24]

App Developer Magazine riporta che, negli ultimi due anni, l'automazione dei test è cresciuta dell'85%. Questo incremento è stato significativamente facilitato dalla disponibilità di tecnologie software Open Source presenti sul mercato.

[22] (Battina, 2019)
[23] (Polo, Reales, Piattini, & Ebert, 2013)
[24] (Wiklund, Eldh, Sundmark, & Lundqvist, 2017)

L'obiettivo dell'intelligenza artificiale e del machine learning nel contesto del *testing* del software è quello di automatizzare e affinare le operazioni di *testing*, migliorando significativamente l'efficacia del processo e la risoluzione degli errori, minimizzando l'intervento umano. Ciò consentirebbe una riduzione dei tempi di esecuzione e libererebbe risorse dai team di *testing*, permettendo loro di focalizzarsi su attività più complesse.[25]

L'avanzamento dell'intelligenza artificiale e il suo contributo nel *testing* del software, determinerà un significativo incremento dei benefici economici; aziende quali Apple, stanno intensificando gli investimenti in materia per ottimizzare le esecuzioni in settori quali salute, veicoli autonomi oppure motori di ricerca. L'integrazione dell'intelligenza artificiale nello sviluppo e *testing* del software è destinata a essere una delle tecnologie in più rapida espansione nei prossimi anni. Tali tecnologie miglioreranno i processi di *Quality Assurance* (o assicurazione della qualità) automatizzando compiti manuali e accelerando i cicli di sviluppo all'interno del Software Development Life Cycle (SDLC).[26]

Passando a un aspetto più pratico, di seguito verranno analizzati i tool maggiormente utilizzati di Test Automation integrati con l'AI, descritti da Sakshi Pandey nel suo articolo denominato "*AI Automation and Testing*"[27].

- *Self-healing tests*

  Come accennato precedentemente, a seguito di aggiornamenti o modifiche dell'applicativo, gli identificatori (nonché xPath) degli elementi possono cambiare determinando un test falso negativo (ovvero che fallirà ma che nella realtà dovrebbe risultare in *passed*). I *self-healing* sono test integrati con l'AI al fine di identificare automaticamente gli errori dovuti alla dinamicità dell'applicativo, consigliando alternative più efficienti o direttamente aggiornando il codice. Ciò minimizza il rischio di test falsi negativi e permette un risparmio in termini di tempo ai tester che avrebbero dovuto trovare l'errore e aggiornare il test.
- *Visual Locators*

  Si tratta di strumenti che permettono di identificare e trovare gli elementi dell'interfaccia utente dell'applicativo operando attraverso un'analisi visiva, senza allora la necessità di ricorrere a identificatori come l'ID, xPath e così via. Inoltre, gli strumenti di automazione dei test integrati con AI oggigiorno sono in grado di utilizzare l'OCR (*Optical Character*

[25] (Wiklund, Eldh, Sundmark, & Lundqvist, 2017)
[26] (Battina, 2019)
[27] (Pandey, 2023)

Recognition – Riconoscimento Ottico dei Caratteri) che consente la lettura dei caratteri all'interno dell'immagine per identificare regressioni visive dell'applicazione.

- *AI Analytics of Test Automation Data*

  Durante l'esecuzione dei test automatizzati, vengono raccolti una grande mole di dati (ad esempio, riguardanti le prestazioni del software, il tempo di esecuzione, i risultati dei test e così via). È necessario analizzarli per poterne estrarre conoscenza e valore ed esistono specifici algoritmi di AI e *machine learning* che permettono l'efficientamento di queste analisi. Ad esempio, alcuni sono in grado di identificare e classificare gli errori mentre altri individuano i test falsi negativi o falsi positivi.

2. Descrizione dei sistemi di Test Automation

In questo capitolo si esamineranno in dettaglio i sistemi impiegati per il Test Automation, analizzando le differenze che sussistono fra test automatici e manuali, le tecnologie e le metodologie comunemente utilizzate per automatizzare i test e, infine, l'impatto effettivo che l'automazione ha nel miglioramento della qualità del software.

2.1 Differenza fra test automatici e test manuali

Come già accennato nei paragrafi precedenti, esistono due tipologie di test del software: test automatici e test manuali. Tale differenza, oltre a essere dettata da questioni pratiche quali un utilizzo differente degli strumenti implementativi, è guidata da altri fattori, come descrive l'*ISO/IEC/IEEE 29119 – 1 seconda edizione 2022 – 01*, standard internazionale che fornisce ai professionisti del *testing* delle linee guida che coprono tutti gli aspetti del ciclo di vita di un software. Tali fattori sono:

- Numero di volte in cui il caso di test è probabile che venga rieseguito. Ad esempio, una regola empirica comunemente utilizzata è che, se un test deve essere eseguito cinque o più volte per progetti di No Regression Test, allora è generalmente conveniente automatizzarlo;
- Competenze dei tester dello specifico progetto;
- Budget e tempo a disposizione per il progetto.[28]

[28] (ISO/IEC/IEEE 29119-1, seconda edizione 2022-01)

In definitiva, è possibile definire il Test Automation come una pratica che prevede l'esecuzione di test automatizzati su applicazioni software il cui obiettivo principale è la minimizzazione del coinvolgimento umano e la massimizzazione dell'efficienza dei test. Si contrappone al test manuale, in cui è prevista l'esecuzione del caso di test da parte di un soggetto senza l'ausilio di strumenti automatizzati. Lo standard "*ISO/IEC/IEEE 29119-1 seconda edizione 2022-01*" definisce i concetti sopra riportati come segue:

> << *Test manuali: esseri umani che eseguono test inserendo informazioni in un elemento del test e verificando i risultati.*
> *Nota 1: I test automatizzati utilizzano strumenti, robot e altri motori di esecuzione dei test per eseguire i test. Il test manuale non utilizza questi elementi.* >>[29]

Pertanto, a differenziare i test automatizzati da quelli manuali è l'ausilio di tools, robot *(*definiti come un *meccanismo attuato programmato con un grado di autonomia, che si muove all'interno del suo ambiente, per eseguire i compiti previsti*[30]) e altri motori di esecuzione dei test.

Come riportato nel Capitolo 1.3, non è sempre conveniente ricorrere all'automazione dei test poiché tale strumento presenta una serie di vantaggi e svantaggi; allo stesso modo i test manuali sono indicati in determinati contesti, mentre risultano poco efficienti in altri. Di seguito verranno analizzati alcuni vantaggi e svantaggi dei test manuali.

Sicuramente i test manuali sono indispensabili se l'obiettivo del test stesso è la valutazione dell'usabilità dell'applicativo (la cosiddetta *User Experience*), poiché questa tipologia di test intrinsecamente necessita di una valutazione umana e soggettiva. Inoltre, si necessita del contributo umano e, dunque, di esecuzioni manuali, anche nel caso di test esplorativi[31] poiché richiedono elevata conoscenza dell'applicativo e creatività. Infine, è utile non ricorrere all'automazione dei test ed eseguirli manualmente nel caso di progetti a breve termine e non ripetibili, poiché l'investimento iniziale non sarebbe coperto dal risparmio apportato dall'automazione dei test.

Al contrario, impiegare il *testing* manuale nel caso di progetti di scala medio-grande e ripetuti nel tempo sarebbe una grossa perdita poiché implicherebbe un elevato investimento in termini di risorse umane, soprattutto nel caso di un numero elevato di test da eseguire e per l'esecuzione di

[29] (ISO/IEC/IEEE 29119-1, seconda edizione 2022-01)
[30] (ISO/IEC/IEEE 29119-1, seconda edizione 2022-01)
[31] I test esplorativi, al contrario di quelli tradizionali, non mirano a verificare funzionalità specifiche dell'applicativo ma esplorano globalmente l'applicazione con l'obiettivo di rilevare difetti in una qualsiasi sezione o flusso.

No Regression Test. Infine, l'accuratezza potrebbe essere minore a causa della maggiore probabilità di commettere errori da parte di un tester manuale, a differenza dei test automatizzati.

2.2 Tecnologie e metodologie utilizzate

La complessità delle applicazioni create oggigiorno è sempre crescente: ci si deve adattare a diversi dispositivi, personalizzare il più possibile le pagine e renderle accessibili a un numero sempre maggiore di utenti. A cambiare è anche l'interazione utente-sistema, non più basata su semplici *click* ma con nuove azioni come quella di *leave-behind*, *scroll* o doppio *tap*. Tutto ciò, seguito dalla crescente esigenza di rapidità nei cicli di sviluppo software, determina la necessità di tools sempre più efficienti per il Test Automation.

I tools maggiormente utilizzati nel panorama attuale vengono descritti di seguito.

- Selenium: è uno degli strumenti più diffusi per l'automazione del *testing* in piattaforme web. Supporta test cross-browser e multipiattaforma e permette la scrittura di test in diversi linguaggi di programmazione quali Java, C# oppure Python.
- Appium: si posiziona come uno strumento essenziale per il *testing* delle applicazioni mobili; è compatibile sia con dispositivi Android che iOS. È immediato da utilizzare e permette l'interazione utente sui dispositivi mobili.
- JUnit e TestNG: si tratta di framework per il linguaggio Java, noti per la loro semplicità e facilità d'uso. Forniscono un ambiente standardizzato per testare il codice. Sono integrati con strumenti di build come Maven, i quali gestiscono la compilazione del codice sorgente, la gestione delle dipendenze (come librerie esterne) e altre attività necessarie per l'esecuzione dei test.
- GitLab CI/CD: è una potente piattaforma di Continuous Integration (CI) e Continuous Deployment (CD) che si integra con GitLab. Continuous Integration è una pratica in cui il codice viene integrato in un repository condiviso fra i tester: a ogni modifica del codice il sistema avvia automaticamente dei processi per garantire la corretta compilazione ed evitare il propagarsi di eventuali errori. Una volta superate le verifiche nella fase di CI, allora è possibile il Continuous Deployment nonché la distribuzione automatica nei vari ambienti.
- IntelliJ IDEA: si tratta di un ambiente di sviluppo integrato con framework di test quali JUnit e consente di scrivere, eseguire e debuggare i test con facilità; è inoltre compatibile con strumenti di build quali Maven.

Di seguito verranno analizzati con maggiore accuratezza Appium e IntelliJ IDEA in quanto strumenti utilizzati nel corso dello stage formativo oggetto della tesi.

2.2.1 Appium

Secondo la documentazione ufficiale, Appium si può definire come "un progetto *open-source* e un ecosistema di software correlati, progettato per facilitare l'automazione dell'interfaccia utente su molteplici piattaforme di app".[32] Esso mette a disposizione un tool denominato Appium Inspector, che permette di ricavare gli identificativi (xPath) degli elementi presenti nelle schermate di un dispositivo mobile.

Di seguito, in Figura 2.1, viene riportata la schermata iniziale di Appium Inspector:

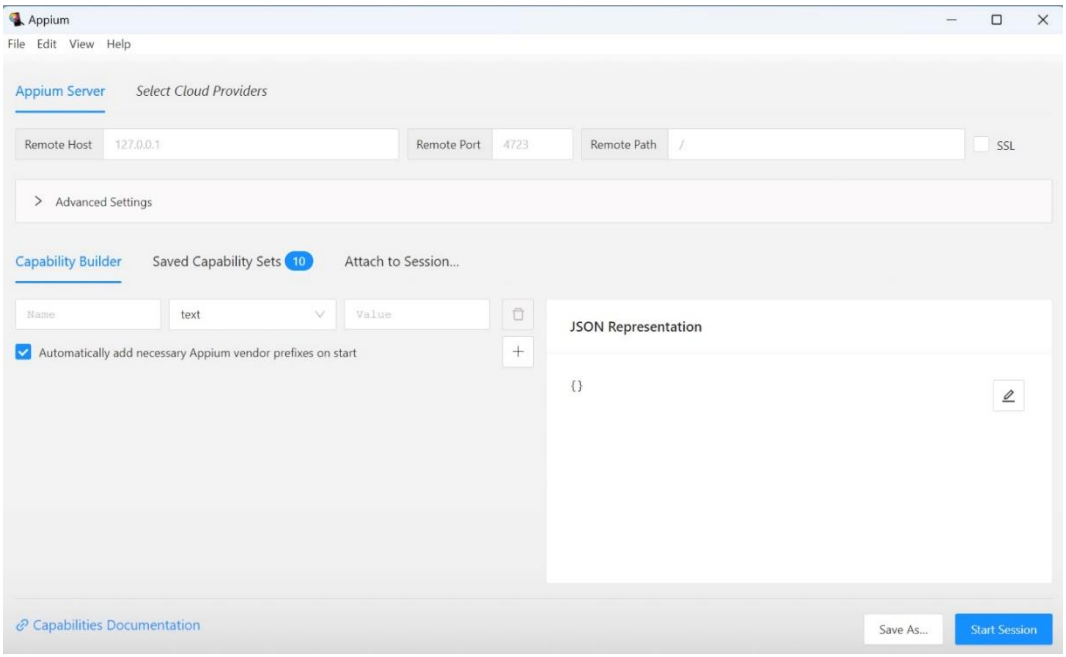


Figura 2.1 - Schermata principale Appium.

[32] (Appium Documentation, 2024)

Innanzitutto, è necessario selezionare il dispositivo da voler visualizzare al *click* del pulsante “Start Session”; i dispositivi sono univocamente identificati da una rappresentazione JSON, da inserire nel riquadro in basso a destra in Figura 2.1. Inoltre, è possibile assegnare un nome e memorizzare l’elenco dei dispositivi in “Saved Capability Sets”. Selezionato il dispositivo di cui si vuole analizzare la schermata è possibile avviare la sessione; la schermata che si visualizza è mostrata in Figura 2.2.

La schermata si divide verticalmente in 3 parti principali: a sinistra si visualizza lo schermo del dispositivo selezionato nel momento in cui è stata avviata la sessione, al centro viene mostrato il codice di quanto visualizzato e, infine, a destra viene mostrato il dettaglio dell’elemento selezionato a seguito dell’interazione con la schermata disposta sulla sinistra. Pertanto, è possibile interagire con l’elemento di interesse tappandolo sullo schermo virtuale oppure selezionandolo dalla struttura ad albero del codice, visualizzando così il dettaglio dell’elemento, tra cui il suo xPath suggerito da Appium stesso oppure i valori booleani di alcuni attributi come “cliccabile”, “selezionato” e così via. Questi ultimi possono risultare particolarmente utili nei controlli delle logiche di business; per chiarire meglio questo concetto consideriamo un esempio. Supponiamo di dover testare il flusso per l’esecuzione di un bonifico in un’applicazione di *mobile banking*; vi sono dei campi obbligatori come, ad esempio, il destinatario del bonifico oppure il conto o la carta del mittente. Il pulsante “Continua” che consente di proseguire con il flusso dovrà essere disabilitato e pertanto dovrà essere valida la condizione “Clickable = False” sino a quando i campi obbligatori

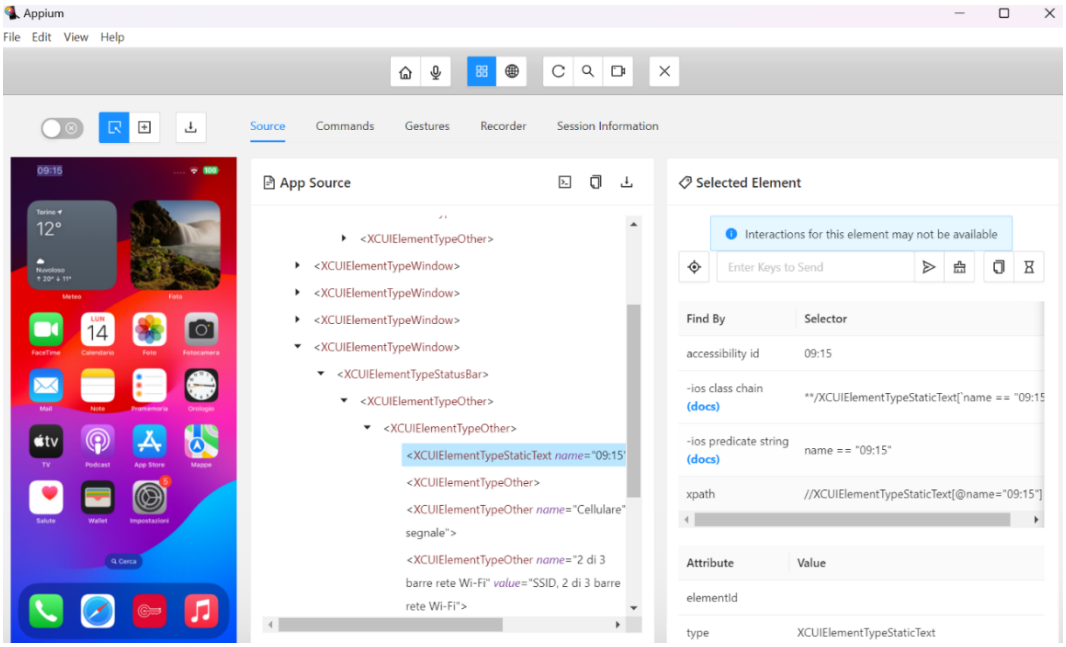


Figura 2.2 - Schermata dispositivo Appium.

non saranno correttamente compilati. Se tale logica non viene rispettata, non è possibile proseguire con l’esecuzione del bonifico. Dunque, nel test automatizzato che verifica la corretta funzionalità del bonifico, sarà necessario inserire il controllo sullo stato del pulsante “Continua”.

Non sempre l'xPath fornito da Appium è corretto da utilizzare o, per meglio dire, efficiente. Supponiamo che, per proseguire in un determinato flusso di test, sia necessario tappare sul nome dell'utente loggato in quella sessione: per identificare l'elemento su cui effettuare il *click*, Appium fornirebbe un xPath associato al nome presente in quel momento, ma ciò non sarebbe efficiente. Infatti, se il test venisse eseguito su un dispositivo connesso con un altro account, quel nome, e quindi l'elemento corrispondente, non verrebbe trovato, causando il fallimento del test. Per questo motivo, è necessario ricorrere alla struttura ad albero e, attraverso relazioni gerarchiche con i valori statici, quali le intestazioni della pagina o le barre di navigazione, è possibile risalire al valore dinamico. Questo aspetto è fondamentale e non deve essere sottovalutato, poiché influisce direttamente sulla durabilità e affidabilità dei test nel tempo.

Inoltre, è possibile interagire con gli oggetti della pagina individuando la loro posizione in relazione all'altezza e alla larghezza dello schermo. Questa funzionalità è utile quando non si desidera interagire con un elemento specifico, ma si intende eseguire un'azione in una determinata area della pagina. Tale concetto è di facile comprensione se si considera la funzione di "*Scroll*", ad esempio, in un carosello di carte o conti nella sezione dedicata di un'applicazione di *mobile banking*: per scorrere tra le diverse carte non è necessario interagire con un elemento specifico, ma è importante simulare il trascinamento del dito in una particolare fascia dello schermo e questo è reso possibile individuando le coordinate di inizio e fine di tale azione.

Infine, un'ulteriore funzionalità di Appium Inspector consente di interagire direttamente con la schermata e proseguire con il flusso in modalità virtuale: pertanto, è possibile selezionare un elemento e tapparlo, proseguendo con le successive schermate che saranno visualizzate nella sezione a sinistra della schermata riportata in Figura 2.2.

2.2.2 Intellij IDEA

IntelliJ IDEA è un ambiente di sviluppo integrato (IDE) compatibile con Java, il cui obiettivo è l'ottimizzazione della produttività degli sviluppatori. Difatti, automatizza le attività ripetitive e di routine, offrendo funzionalità come il completamento intelligente del codice, l'individuazione degli errori di scrittura o strutturali dell'ambiente oppure opzioni di debug. Include anche un sistema di comunicazione tramite chat integrato con l'intelligenza artificiale, consentendo agli sviluppatori di effettuare delle richieste e ricevendo opportune risposte. [33]

[33] (IntelliJ Documentation, 2024)

In Figura 2.3 viene mostrata la schermata di IntelliJ IDEA:

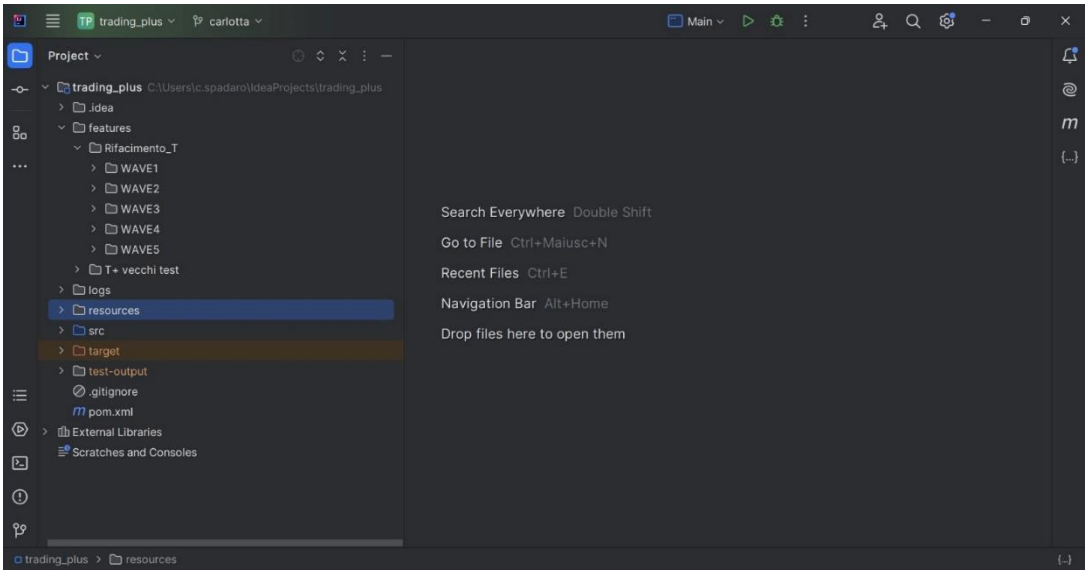


Figura 2.3 - Schermata IntelliJ IDEA.

Ciascun tester ha la possibilità di caricare in locale, cioè sul proprio computer personale, i progetti salvati su piattaforme remote quali GitLab, GitHub oppure BitBucket. Ciascun progetto è univocamente identificato da un *repository*, che può essere definito come un archivio digitale contenente il codice del progetto e la cronologia delle modifiche.

Generalmente ciascun tester lavora nel proprio *branch*, nonché una versione parallela del codice in cui è possibile apportare modifiche senza influenzare il *branch* principale. Una volta completato il lavoro nel proprio *branch*, è possibile integrarlo (cioè fare *merge*) con quello principale, permettendo così al resto del team di accedere alle modifiche e, a loro volta, di effettuare un *merge* nel proprio *branch*. Questa procedura assicura che tutti i membri del team siano allineati e abbiano accesso all'ultima versione, facilitando il lavoro parallelo su uno stesso progetto.

IntelliJ IDEA è in grado di rilevare eventuali conflitti durante il processo di *merge*. In caso di modifiche o eliminazioni nel *branch* personale rispetto al principale, il programma segnala tali conflitti, permettendo ai tester di conservare il codice originale se si tratta di un errore oppure, se volontario, di sostituirlo con le modifiche apportate.

Come descritto sin ad ora, IntelliJ IDEA è un ambiente di sviluppo che consente l'esecuzione di test automatizzati su vari dispositivi, la cui selezione avviene tramite un file di configurazione. Quest'ultimo presenta l'elenco dei dispositivi memorizzati su cui eseguire un determinato test; dato un unico accesso, è possibile la selezione contemporanea di un solo dispositivo, pertanto se l'obiettivo è parallelizzare l'esecuzione, è necessario ricorrere ad altri strumenti.

Inoltre, è possibile effettuare il debug dei test, monitorando in dettaglio le righe di codice eseguite e il risultato parziale ottenuto. In caso di fallimento del test, IntelliJ IDEA fornisce una segnalazione dettagliata della tipologia di errore riscontrato.

IntelliJ IDEA supporta Cucumber e Gherkin: il primo è uno strumento utilizzato per l'implementazione dei test automatici tramite il linguaggio definito di Gherkin. Quest'ultimo ha l'obiettivo di ridurre il divario fra il linguaggio tecnico utilizzato per la stesura del codice e la comprensione da parte di stakeholder non tecnici. Difatti, utilizza una serie di parole chiave per dare una struttura e un significato facilmente interpretabile alle righe di codice.[34] Di seguito viene riportato un esempio a puro scopo esplicativo, descritto nella documentazione ufficiale di Cucumber:

Scenario: Maker starts a game

When the Maker starts a game

Then the Maker waits for a Breaker to join

Definito lo scenario, risulta semplice la comprensione delle righe di codice, introdotte dalle parole chiave "When" e "Then". Esse richiamano a funzioni scritte in linguaggio Java, che effettuano quanto descritto in linguaggio naturale, definito da Gherkin: ciò permette di comprendere cosa le righe di codice fanno, senza addentrarsi nei tecnicismi dei linguaggi quali Java.

Tali strumenti supportano il cosiddetto *Behaviour-Driven Development* (*BDD*), definito come "un modo per i team di sviluppo e *testing* del software di ridurre il gap fra stakeholder e personale tecnico".[35] BDD è strettamente correlato alla metodologia *Agile* e prevede l'esecuzione di un processo iterativo, costituito da 3 fasi descritte di seguito e rappresentate in Figura 2.4:

1. Si esamina la User Story[36] da realizzare, discutendo sulla sua realizzare e su ciò che il cliente si aspetta.
2. Si integra nel software la nuova funzionalità.

[34] (Cucumber Documentation, 2024)
[35] (Cucumber Documentation - Behaviour-Driven Development, 2024)
[36] Il termine User Story viene tipicamente utilizzato nella metodologia *Agile* e, per il caso in questione, indica la nuova funzionalità da integrare nel software, richiesta dal cliente.

3. Si sviluppa il test automatizzato.

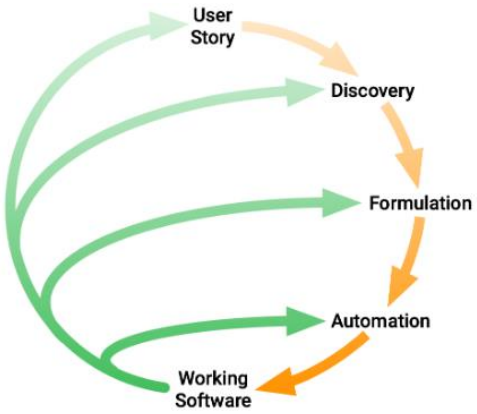


Figura 2.4 - Discovery, Formulation e Automation.
Fonte: (Cucumber Documentation - Behaviour-Driven Development, 2024)

Tali pratiche vengono definite come "*Discovery, Formulation e Automation*" e, difatti, indicano le tre fasi da eseguire ogni qualvolta sia necessario un piccolo cambiamento. Inoltre, si promuove l'introduzione di piccole funzionalità in modo frequente, da sviluppare secondo il paradigma *Discovery, Formulation e Automation*, piuttosto che un'unica, generica e grande funzionalità introdotta in via eccezionale. Pertanto, BDD incoraggia a intervenire tramite iterazioni rapide seguite da *feedback* dei clienti, da analizzare e utilizzare per l'iterazione successiva.

2.3 Impatto del Test Automation nella qualità del software

Sin ora è stata ampiamente discussa la stretta correlazione fra *Quality Assurance* e *Test Automation*. Il presente capitolo sarà dedicato all'analisi di studi che confermano quanto descritto e discusso fin ora.

2.3.1 Caso I: transizione da test manuali ad automatici

Analizziamo un primo studio denominato "*Transitioning Manual System Test Suites to Automated Testing: An Industrial Case Study*"[37]: esso analizza gli effetti dell'automazione dei test attraverso la tecnica *Visual GUI Testing* (VGT) in un contesto industriale. Tale tecnica utilizza il riconoscimento delle immagini per interagire con il sistema e verificare il suo comportamento tramite l'interfaccia utente (*GUI – Graphical User Interface*).[38]

[37] (Alegroth, Feldt, & Olsson, 2013)
[38] (Springer, 2014)

Per un confronto più diretto, analizziamo le condizioni nelle fasi pre-transazione, durante e post-transazione dai test manuali ai test automatizzati.

- *Pre–transizione*

  Nel caso dell'azienda in questione, il passaggio da test manuali a test automatizzati era necessario poiché a ogni rilascio del software da testare (circa un paio di volte l'anno), si richiedeva un'ampia sessione di No Regression Test. I casi di test da svolgere erano raccolti in 40 suite, ciascuna denominata "Descrizione di test di accettazione" (ATD); ciascuna ATD comprendeva circa 100 casi d'uso collegati insieme in catene di test che definiscono diversi scenari. Quest'ultima circostanza viene mostrata in Figura 2.5.

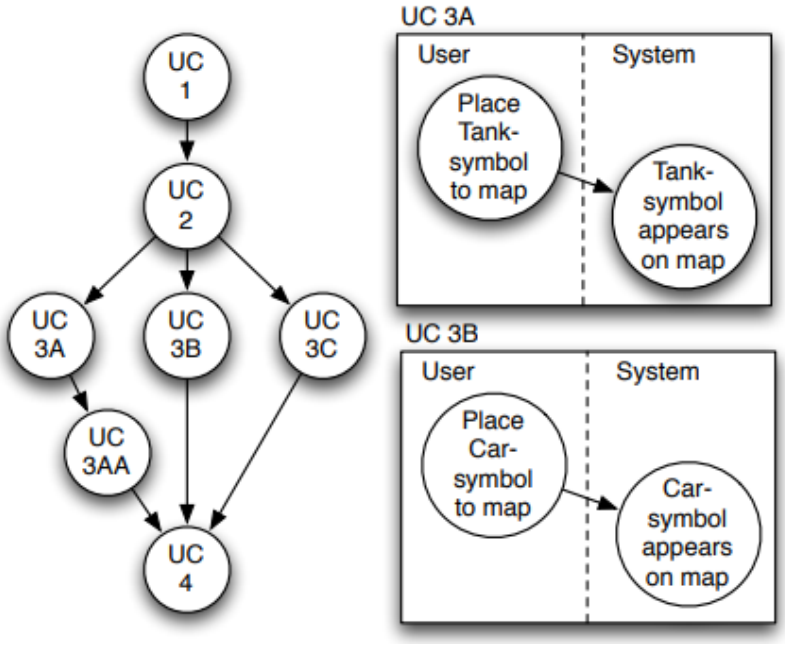


Figura 2.5 - Catena di test.
Fonte: (Alegroth, Feldt, & Olsson, 2013)

La Figura 2.5 presenta due sezioni principali: a sinistra viene raffigurata una catena di test, costituita da una serie di casi d'uso (raffigurati a destra) tutti correlati fra loro. Come si evince dalla rappresentazione, un caso di test è definito come un percorso lineare o ramificato, dipendente dai collegamenti. Questo, alle volte, risulta essere problematico perché a causa dei colli di bottiglia si rallenta l'intero processo; prendendo come esempio i collegamenti in Figura 2.5, è possibile notare che per svolgere il caso d'uso 4, è necessario percorrere interamente i tre rami paralleli 3A, 3B, 3C e, qualora uno di essi dovesse essere particolarmente duraturo, bloccherebbe l'intero caso.

A seguito di vari approfondimenti del caso studio in questione, si è constato che il tempo necessario per lo svolgimento di una sotto selezione dei No Regression Test manuali aveva una durata variabile da 6 a 10 settimane. Pertanto, la transizione ai test automatizzati avrebbe significato un grosso guadagno per l'azienda, in termini di tempo, costi e qualità.[39]

- *Durante la transizione*

  L'implementazione di tale metodologia ha richiesto circa 4 mesi. Circa 3 mesi dopo l'inizio del progetto, il sistema da testare ha subito un grande cambiamento, il quale ha causato il fallimento di circa l'85-90% dei test automatizzati. È stata necessaria una manutenzione dei test che ha comportato la modifica del 5-30% di ogni *script* e ha impiegato circa tre settimane, facendo sì che il 25,8% del tempo di sviluppo è stato utilizzato per effettuare manutenzione. Tale circostanza conferma ancora una volta l'importanza e la necessità di effettuare manutenzione nel caso di automazione dei test; si tratta di un aspetto da non sottovalutare e necessariamente da pianificare ad intervalli regolari per garantire la consistenza e affidabilità di un progetto di Test Automation.

  Successivamente è stata effettuata un'analisi sul *Return On Investment* (ROI), indicatore che valuta la redditività dell'investimento effettuato, che in questo caso coincide con l'impiego di tecniche di automazione dei test. Con i test manuali il costo cresce in maniera proporzionale al numero di test da eseguire, poiché le risorse impiegate per il primo ciclo di esecuzione saranno ugualmente necessarie per i cicli successivi. Invece, con i test automatizzati il costo rimane costante dopo la prima iterazione: in particolare, si ha un *effort* iniziale dovuto alla scrittura del codice dei test, ma alle successive iterazioni lo sforzo sarà minimo e costante poiché sarà semplicemente necessario eseguire i test già scriptati e valutare il loro esito. Il ROI è definito come segue:

$$ROI = \frac{Profitto\ netto}{Costo\ dell'investimento} * 100$$

  se il costo dell'investimento continua a crescere, allora il ROI diminuisce, così come nel caso dei test manuali. Invece, se il costo dell'investimento rimane costante, allora il ROI aumenta. Questo dimostra che nel lungo termine il test automatizzato permetterà un ritorno sull'investimento maggiore.

  Durante il periodo di transizione sono sorti sia problemi tecnici relativi ai tools utilizzati, di cui non discuteremo, sia problemi relativi alla transizione di per sé, riassunti in Tabella 2.1, con la rispettiva soluzione adottata.

[39] (Alegroth, Feldt, & Olsson, 2013)

Problema	Soluzione
VNC risulta problematico nell'individuazione delle grafiche GUI (interfaccia utente)	1. Minimizzazione dell'utilizzo di VNC 2. Utilizzo di VNC ad alta qualità
Necessità di manutenzione degli script e comprensione di quanto fatto da altri soggetti del team	Imposizione di standard di codifica per incrementare la comprensibilità e leggibilità degli script
Non è sempre possibile una mappatura 1-1 fra script manuali e automatici	Applicare una mappatura 1-1 solo se non inficia la qualità dei test

Tabella 2.1 - Problemi e soluzioni durante la transizione da test manuali ad automatici.

I problemi descritti in Tabella 2.1 sono comuni nel caso di transizione da test manuali a test automatici. Una casistica molto comune riguarda la poca conoscenza dello strumento utilizzato e, pertanto, può succedere che esso non performi in modo ottimale. In tali circostanze è bene intervenire in maniera opportuna: si può optare per un utilizzo ridotto e graduale del nuovo strumento oppure per un suo potenziamento. Tale scelta dipende dalle disponibilità economiche dell'azienda e dalla necessità di implementazione del nuovo strumento. Inoltre, in un progetto di Test Automation inevitabilmente collaborano più persone, pertanto è fondamentale stabilire uno standard unico e noto all'intero team in modo da rendere possibile a tutti la comprensione di quanto realizzato. Infine, l'ultimo aspetto riguarda la difficoltà nel trasformare interamente i test manuali in automatici: non tutti i processi e i controlli sono automatizzabili, di conseguenza è fondamentale comprendere per quali test l'automazione è vantaggiosa e, il resto, eseguirlo ancora manualmente.

- *Post–transizione*

  Terminate le procedure necessarie per automatizzare i test, sono stati confrontati i tempi di esecuzione rispetto al caso manuale. Dall'analisi è emersa una velocità di esecuzione circa 16 volte maggiore rispetto al metodo manuale. Pertanto, l'automazione ha determinato un notevole miglioramento in termini di tempo di esecuzione, senza inficiare negativamente sulla capacità di individuazione dei bug del sistema: difatti, tutti i bug identificati con i testi manuali, sono stati individuati con l'implementazione dei test automatizzati.

  Per concludere l'analisi, sono state poste delle domande ai tester, le cui risposte sono riportate in Tabella 2.2:

Domanda	Risposta

VGT funziona? Perché?	Sì, è la tecnica più capace per automatizzare i test
VGT può essere considerata un'alternativa al *testing* manuale o solo un suo complemento?	Un complemento, perché permette di individuare *defect* relativi solamente a quanto eseguito dal codice e nulla di più
Qual è il maggior problema di VGT?	La volatilità del tool e la scarsa qualità nel riconoscimento delle immagini
Quali sono le modifiche da apportare?	Il supporto per il *testing* di sistemi distribuiti

Tabella 2.2 - Domande e risposte dei tester post transizione.

In definitiva, i tester ritengono valida e capace tale tecnica di automazione; tuttavia, essa incorre in diversi problemi quali la volatilità del tool e le prestazioni non del tutto ottimali ma, d'altronde come un qualsiasi altro nuovo strumento appena implementato, esso richiede del tempo per stabilizzarsi e risolvere i principali problemi. I tester, infine, evidenziano la mancanza di un adeguato supporto per il *testing* di sistemi distribuiti, probabilmente perché le tecnologie impiegate sono difficili da configurare e utilizzare. Basti pensare alla configurazione degli ambienti di *testing*, i quali richiedono settaggi precisi e procedure lunghe.

In conclusione, è possibile affermare che, l'automazione dei test dipende dal contesto di applicazione e non in tutti i casi risulta vantaggiosa poiché richiede grossi investimenti iniziali e ulteriori costi di manutenzione. Tuttavia, nel momento in cui viene implementata correttamente e in contesti adeguati determina un risparmio in termini di tempi, costi e rilevazione dei *defect*. L'unica limitazione riguarda il fatto che, non essendo supportata da una supervisione umana, riesce a verificare unicamente quanto presente nel codice senza considerare il resto.

2.3.2 Caso II: benefici del Test Automation

Di seguito verrà analizzato un caso studio che esamina gli effetti del Test Automation su diverse tipologie di progetto: "Observations and Lessons Learned from Automated Testing" redatto da Stefan Berner, Roland Weber e Rudolf K. Keller[40].

Di seguito verranno descritti brevemente i cinque progetti su cui si baseranno le osservazioni e considerazioni di tale studio:

[40] (Berner, Weber, & Keller, 2005)

- *Sistema per la gestione della distribuzione degli asset*: si vuole testare e garantire la qualità di un sistema di gestione di asset hardware, tenendo traccia degli hardware distribuiti e della loro configurazione ai fini della fatturazione.
- *Piattaforma applicativa basata su Java*: si vuole valutare la qualità di alcuni componenti del framework realizzato.
- *Sistema distribuito di punto vendita per assicurazioni vita (POS)*: prevede la realizzazione e il *testing* dell'architettura software per la vendita di assicurazioni vita da parte di una grande banca che vuole entrare in altri mercati.
- *Supporto alle vendite per impianti industriali*: poiché l'obiettivo del sistema di vendite è quello di creare offerte personalizzate in tempi brevi, si vuole testare la flessibilità del sistema sulla base dei requisiti in evoluzione e la coerenza con le modifiche provenienti dai diversi reparti di vendita nazionale. Difatti, il Test Automation viene impiegato per verificare che i continui aggiornamenti non alternino le funzionalità preesistenti.
- *Automazione dei test per il sistema di controllo*: l'obiettivo era automatizzare degli *smoke test* preesistenti, che dovevano essere eseguiti ogni settimana su diversi hardware per verificare la stabilità della build di base.

Il processo di automazione dei test non prevede unicamente la realizzazione degli *script*, ma necessita la formulazione di una vera e propria strategia di automazione. Alcuni errori individuati nel caso studio in questione e comuni nella definizione della strategia sono i seguenti:

- *Test mal posizionati*: il test automatizzato deve essere consistente nel tempo e cioè il codice deve funzionare sia a distanza di tempo, supponendo che i flussi non cambiano, oppure anche sotto condizioni differenti quali l'utenza di accesso. Ad esempio, se è necessario verificare la logica del programma allora non ha senso effettuare un test tramite l'interfaccia utente poiché quest'ultima può cambiare.
- *Aspettative sbagliate*: si creano aspettative amplificate riguardo gli effetti del Test Automation. Spesso a seguito del passaggio da test manuali ad automatizzati, ci si aspetta un incremento del ROI molto rapido. Tuttavia, non sempre questo si verifica poiché ci sono aspetti positivi introdotti dall'automazione che il ROI non considera, anche se di grande potenziale. Essi sono i cicli di rilascio più brevi, un risparmio in termini di costo per la correzione dei bug (poiché questi vengono individuati in tempo e corretti) oppure una maggiore qualità del software poiché i tester si occupano di casistiche più complicate e peculiari, automatizzando i test più noiosi e ripetitivi. Pertanto, in determinati casi ci si aspetta unicamente un aumento del ROI, sottovalutando una serie di altri benefici apportati.

- *Mancanza di diversificazione*: molte organizzazioni che adottano l'automazione pensano di automatizzare tutto ciò che precedentemente veniva eseguito manualmente; tuttavia, questo non sempre è conveniente poiché alcuni test risulterebbero poco funzionali, non riuscendo magari a incorporare tutte le casistiche da verificare.
- *Utilizzo degli strumenti di automazione limitato all'esecuzione dei test*: spesso si pensa che l'automazione debba unicamente far riferimento ai test, quando invece esistono diverse aree in cui automatizzare garantirebbe un grosso vantaggio. Esempi sono le procedure di installazione e configurazione dei tools o sistemi da utilizzare.

Gli errori commessi per la definizione delle strategie sono stati i seguenti:

- per il caso del sistema di vendita delle assicurazioni vita c'erano aspettative molto più elevate riguardo la rilevazione dei *defect* del sistema a seguito dell'automazione dei test, stimando l'individuazione un numero nettamente superiore di problemi nel sistema.
- per il sistema di controllo, l'automazione non ha permesso di raggiungere risultati sorprendenti poiché non è stato possibile implementarla al di fuori dei test e, inoltre, non c'è stato un buon posizionamento e una buona diversificazione dei test, dato che l'unica interazione possibile era a livello di interfaccia. Tuttavia, questa problematica è stata risolta successivamente permettendo l'interazione con i componenti e potendo simulare un numero maggiore di scenari.
- per quanto riguarda la strategia attuata dal sistema di supporto per impianti industriali, essa è risultata come la più completa: l'automazione non era limitata unicamente all'interfaccia utente e c'è stata una buona diversificazione dei test, continuando ad eseguire manualmente quei test la cui automazione non risultava vantaggiosa.

Passando a un aspetto implementativo, ci si aspetta che, qualora un test debba essere eseguito per un numero maggiore di cinque volte, allora risulta conveniente la sua automazione. Nel caso dei progetti in analisi, tutti i casi di test rispettano tale soglia e circa il 25% dei test viene eseguito più di venti volte; pertanto, automatizzarli risulta essere una scelta conveniente.

Inoltre, gli autori hanno constato che l'*overhead* medio per creare dei test automatizzati, a partire dai manuali, è poco inferiore al fattore 2; tuttavia, si registra una varianza molto elevata poiché è possibile osservare valori di *overhead* pari a 30. Quindi è bene scegliere con accuratezza una buona strategia per evitare eccessivi sovraccarichi: solitamente è bene iniziare con l'automazione dei test che garantiscono un ROI maggiore (ad esempio, gli *smoke test* oppure i test di integrazione) per poi proseguire con il resto dei test.

A riguardo, gli autori sostengono che sia più appropriato seguire un approccio *bottom-up* e cioè inizialmente eseguire in maniera automatizzata solo una sottosezione di test per poi incrementare in maniera progressiva, piuttosto che automatizzare a priori tutti i casi di test. Ad esempio, per il progetto del sistema per la gestione degli asset si aveva a disposizione un tempo ridotto che non ha permesso l'automazione dei casi di test; inoltre, si prevedeva che il numero di esecuzione di quei test fosse ridotto (circa 3-5 esecuzioni), di conseguenza l'automazione non era la scelta migliore.

Il Grafico 2.1 mostra l'*effort* in funzione del numero di esecuzioni, distinguendo l'esecuzione manuale da quella automatizzata sia per un progetto di 20 test che per un altro di 60.

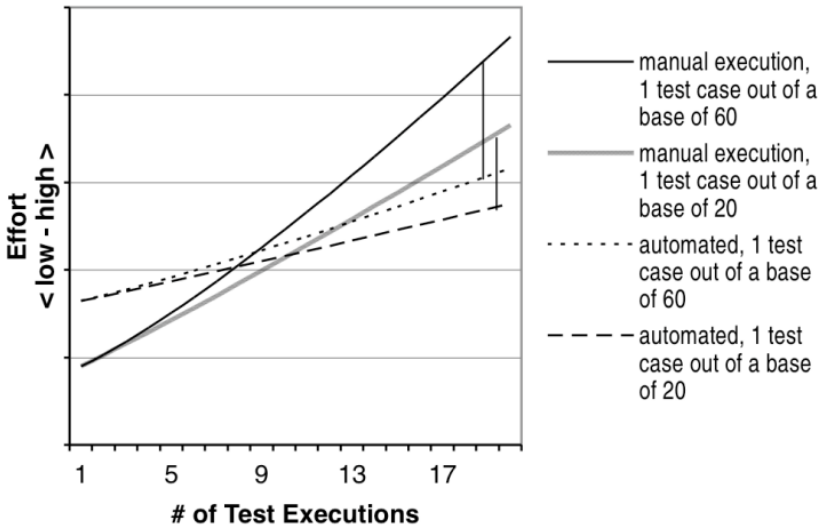
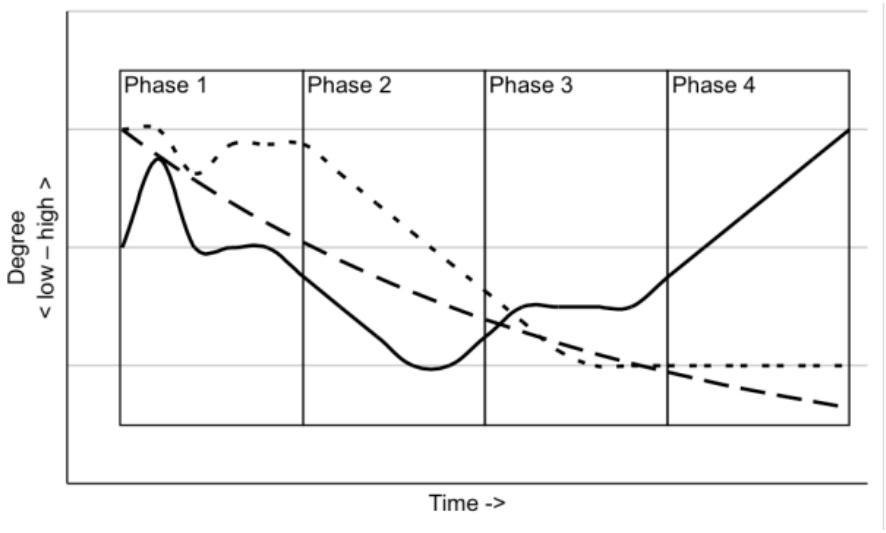


Grafico 2.1 - Effort dei test automatizzati e manuali.
Fonte: (Berner, Weber, & Keller, 2005)

Per entrambi i progetti è possibile notare che, al crescere del numero dei test eseguiti, lo sforzo richiesto per l'esecuzione manuale è maggiore rispetto al caso automatico. Difatti, la pendenza della prima retta è maggiore rispetto alla seconda. In entrambi i casi è richiesto uno sforzo da parte dei tester, ma per l'esecuzione manuale esso è maggiore perché si deve eseguire l'intero caso di test manualmente, al contrario di quello automatizzato dove è necessario avviarlo e valutare il suo risultato. Inoltre, è immediato comprendere che l'esecuzione di 60 test richiede un *effort* maggiore rispetto a eseguirne 20; tuttavia, tale incremento è maggiore nel caso dei test manuali rispetto a quelli automatici.

Gli autori pongono l'attenzione su un ulteriore aspetto: è necessario eseguire ed effettuare manutenzione dei test con frequenza e regolarità, altrimenti essi perdono validità e diventano poco comprensibili, causando il fallimento delle esecuzioni. Il Grafico 2.2 mostra il grado di necessità, integrità e comprensione dei test durante le diverse fasi del ciclo di vita di un software.

Di seguito vengono illustrate nel dettaglio le 5 fasi:



- Fase 1: è la fase di progettazione e primissima implementazione del progetto. La necessità dei test automatizzati è elevata poiché si vuole individuare il maggior numero di bug. Inoltre, dato il numero ridotto di test automatizzati sulle funzionalità basilari dell'applicativo, la loro comprensione risulta immediata e l'integrità è elevata.
- Fase 2: in questa fase le funzionalità del software aumentano e parallelamente i tester hanno acquisito confidenza con gli strumenti; date le nuove funzionalità, si preferisce creare nuovi casi di test e per tale motivo la manutenzione e correzione dei vecchi test passa in secondo piano. Ciò determina una minore comprensione e integrità dei test. Riguardo la necessità dei test, si è constatato che gli errori vengono principalmente individuati nella fase precedente, di conseguenza la necessità dei test automatizzati decresce in questa fase.
- Fase 3: il software è stabile e bisogna mantenerlo integro e funzionante, nonostante l'aggiunta e modifica di alcune funzionalità. Per tale motivo l'importanza dei test cresce nuovamente, tuttavia la comprensione e l'integrità continuano a decrescere, salvo manutenzione dei test preesistenti.
- Fase 4: in questa fase sarebbe utile eseguire test automatizzati di non regressione; pertanto, la necessità aumenta a picco. Tuttavia, la loro comprensione continua a

decrescere mentre l'integrità è stabile rispetto alla fase precedente. Questo scenario si verifica solo nel caso in cui non è stata pianificata alcuna attività di manutenzione.

Nel progetto del sistema distribuito per la vendita delle polizze a vita, è stata presa la decisione di investire nella manutenzione dei test automatizzati, migliorando la comprensione e l'integrità dei test. Di conseguenza, nella fase 4 del progetto è stato possibile eseguire i test di non regressione automatizzati.

Come già accennato precedentemente, la maggior parte dei bug viene individuato nella fase di sviluppo iniziale del progetto, durante la quale vengono sviluppati i System Test. Tuttavia, anche i test sono in continua evoluzione, difatti esiste il ciclo di sviluppo dei test che prevede diversi passaggi di affinamento a seguito dei quali risulta migliorata la qualità dei test stessi. A tal proposito, il Grafico 2.3 mostra la percentuale media di copertura delle istruzioni (indicata con "avg sc") e dei rami ("avg bc"), in funzione del numero di affinamenti dei test. La prima fa riferimento alla percentuale di codice eseguito durante i test; invece, la copertura dei rami misura la percentuale di flussi eseguiti. Per fare un esempio pratico, supponiamo di dover testare l'esecuzione di un bonifico in un'applicazione di un istituto bancario; il bonifico può avere esito positivo oppure negativo. Si verifica il 100% di copertura delle istruzioni nel momento in cui il codice relativo all'esecuzione del bonifico viene testato (qualunque sia l'esito); invece, si ha il 100% di copertura dei rami se e solo se viene testato sia il caso di esito positivo sia il caso di esito negativo.

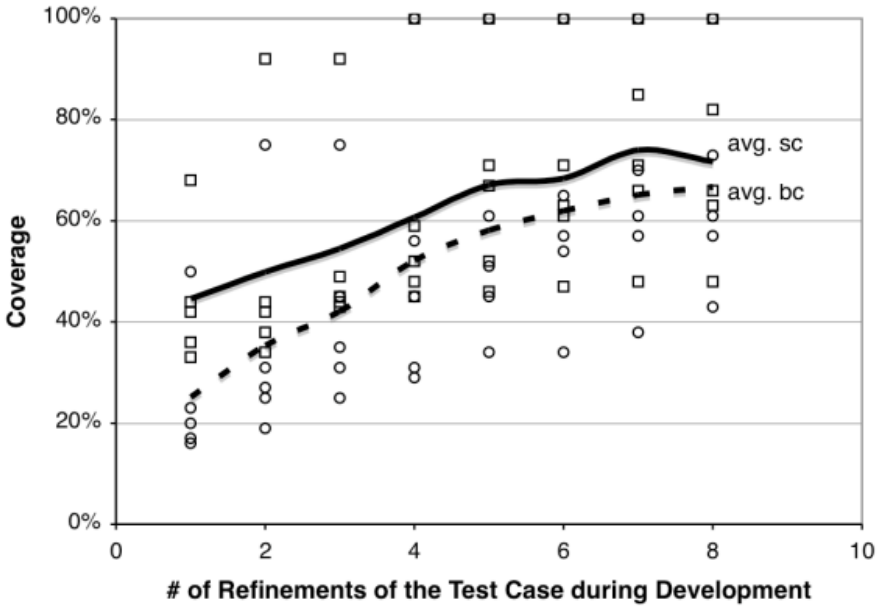


Grafico 2.3 - Copertura dei test.
Fonte: (Berner, Weber, & Keller, 2005)

Dalle curve mostrate nel Grafico 2.3 è possibile notare che nessun caso di test raggiunge più del 50% di copertura delle istruzioni e del 30% di copertura dei rami al primo raffinamento. Questo

dimostra che è molto difficile fornire un buon test sin dal primo momento; è necessario rivedere il test più volte e adattarlo ai cambiamenti prima di raggiungere buoni livelli di copertura.[41]

[41] (Berner, Weber, & Keller, 2005)

3. Test Automation nel settore bancario

L'automazione dei test per le applicazioni del settore bancario è una pratica sempre più diffusa, poiché gli istituti bancari cercano di migliorare l'efficienza operativa, ridurre i costi e massimizzare la qualità del servizio offerto, rispettando le norme in vigore e garantendo la sicurezza dei dati dei clienti. Tuttavia, essendo le applicazioni bancarie dei sistemi complessi, guidate da precise logiche di business nel rispetto di regole in materia di sicurezza, l'automazione dei test non è sempre immediata; in più, le normative bancarie cambiano frequentemente e ciò implica un continuo aggiornamento dei test o la creazione di nuovi.

Il settore bancario, in particolar modo, ha come obiettivo principale la creazione di un rapporto di fiducia con i propri clienti, garantendo in prima battuta la loro privacy e sicurezza. In un tale contesto, l'adozione di nuove tecnologie o strumenti quali il Test Automation, può incontrare una certa resistenza dovuta a vari fattori quali la poca conoscenza dello strumento, l'incertezza sulla loro affidabilità oppure la sicurezza nei processi e metodologie preesistenti, che disincentivano al cambiamento.

3.1 Regolamentazioni e requisiti

Gli istituti bancari devono rispettare normative e requisiti specifici che regolano l'adozione e l'implementazione di soluzioni automatizzate per il *testing*.

Di seguito verranno analizzate alcune normative in vigore per il settore bancario:

- *Payment Card Industry Data Security Standard* (*PCI DSS*): presenta diversi requisiti di sicurezza per proteggere i dati delle carte di pagamento, sviluppato dal *PCI Security Standards Council* nonché un'organizzazione formata dai principali circuiti di pagamento quali Visa, American Express, Mastercard e così via. Viene richiesta l'implementazione di varie misure di sicurezza, al fine di ridurre il rischio di frodi e offrire un'ulteriore garanzia di sicurezza ai clienti. Questo standard deve essere rispettato da qualsiasi istituzione che elabora, archivia oppure trasmette informazioni relative alle carte di pagamento; pertanto, banche e istituti finanziari devono rispettarlo.

  Analizzando il documento ufficiale "*Payment Card Industry Data Security Standard - Requirements and Testing Procedures*"[42] nella sua ultima versione 4.0.1 che risale al mese di Giugno 2024, è possibile visualizzare i 12 requisiti principali da rispettare, ognuno dei

[42] (Payment Card Industry Data Security Standard: Requirements and Testing Procedures, 2024)

quali copre uno specifico aspetto della sicurezza. La Tabella 3.1 riporta i requisiti e il rispettivo ambito a cui fanno riferimento:

PCI Data Security Standard – High Level Overview	
Build and Maintain a Secure Network and Systems	**1.** Install and Maintain Network Security Controls. **2.** Apply Secure Configurations to All System Components.
Protect Account Data	**3.** Protect Stored Account Data. **4.** Protect Cardholder Data with Strong Cryptography During Transmission Over Open, Public Networks.
Maintain a Vulnerability Management Program	**5.** Protect All Systems and Networks from Malicious Software. **6.** Develop and Maintain Secure Systems and Software.
Implement Strong Access Control Measures	**7.** Restrict Access to System Components and Cardholder Data by Business Need to Know. **8.** Identify Users and Authenticate Access to System Components. **9.** Restrict Physical Access to Cardholder Data.
Regularly Monitor and Test Networks	**10.** Log and Monitor All Access to System Components and Cardholder Data. **11.** Test Security of Systems and Networks Regularly.
Maintain an Information Security Policy	**12.** Support Information Security with Organizational Policies and Programs.

Tabella 3.1 - 12 requisiti del PCI DSS.
Fonte: (Payment Card Industry Data Security Standard: Requirements and Testing Procedures, 2024)

Per costruire e mantenere un sistema e una rete entrambi sicuri, bisogna installare e mantenere dei controlli di sicurezza di rete (ad esempio, *firewall* oppure sistemi di prevenzione) e applicare delle configurazioni sicure a tutti i componenti del sistema.

Risulta fondamentale anche proteggere i dati degli account, sia nel processo di archiviazione (attraverso crittografia o altre tecniche per ridurre il rischio di esposizione nel caso di violazione della sicurezza e accesso ai dati memorizzati) ma anche durante la loro trasmissione su rete pubblica.

Bisogna mantenere e seguire un programma di gestione delle vulnerabilità, occupandosi in prima battuta della protezione sia di sistemi che di reti da parte di software dannosi quali *malware* oppure virus; inoltre, ci si deve assicurare che l'ambiente sviluppato sia mantenuto sicuro.

Gli istituti bancari devono inoltre accertarsi di disporre di adeguate e rigorose misure di controllo per quanto riguarda la visualizzazione dei dati sensibili, limitando l'accesso solamente a coloro che ne necessitano per il normale svolgimento dell'attività lavorativa; inoltre, ciascun utente prima di accedere ai sistemi deve essere identificato e autenticato. Oltre alla protezione digitale, deve essere implementata una protezione fisica dei dati sensibili, riducendo l'accesso fisico a server o altri dispositivi che processano e/o memorizzano i dati delle carte.

Infine, gli ultimi aspetti da considerare riguardano il monitoraggio degli accessi tramite registri per individuare eventuali attività sospette o non autorizzate, l'esecuzione di test

regolari che verifichino il rispetto delle pratiche di sicurezza nel tempo e il supporto alla sicurezza tramite mirate politiche aziendali mirate o programmi di formazione.[43]

- *General Data Protection Regulation* (*GDPR*): è una normativa europea relativa alla protezione dei dati personali, entrata in vigore nel 2018. Impone obblighi rigorosi nel rispetto e nella tutela della privacy dei clienti e garantisce che i dati personali siano trattati in modo sicuro e conforme. La violazione del GDPR può comportare significative sanzioni, con ripercussioni non solo finanziarie ma anche reputazionali.[44]

  L'automazione gioca un ruolo fondamentale, permettendo l'esecuzione di test che simulano attacchi informatici e testano in momenti differenti la vulnerabilità del sistema. È anche possibile svolgere test automatizzati che verificano la crittografia o la robustezza dell'autenticazione durante l'accesso.

- *Certificazione ISO/IEC 27001*: è uno degli standard internazionali più importanti riguardante la gestione della sicurezza delle informazioni. Definisce i requisiti necessari per realizzare, mantenere e operare un miglioramento continuo dei Sistemi di Gestione della Sicurezza delle Informazioni (SGSI), che protegge i dati sensibili da minacce informatiche come attacchi o accessi non autorizzati.

  La norma, oltre a richiedere controlli di sicurezza mirati per la protezione di informazioni sensibili quali finanziarie, personali o aziendali, richiede la presenza di un processo di gestione degli incidenti, la loro risoluzione e un piano di miglioramento continuo. Garantisce pertanto che l'organizzazione sia sempre pronta a fronteggiare nuove minacce. Infine, tale certificazione può determinare un vantaggio competitivo per l'azienda che la possiede e permette di stabilire un rapporto di fiducia con il cliente.

  Il Test Automation in questo ambito potrebbe implementare dei test di controllo per mitigare i rischi, quali gestione degli accessi non autorizzati oppure valutazione della vulnerabilità del sistema ad attacchi esterni. Ciò permetterebbe agli istituti bancari di adottare una strategia come garanzia di sicurezza, mantenendo elevati livelli di protezione e implementando un miglioramento continuo ed efficiente.[45] Quest'ultimo viene definito come *continuous compliance*: la conformità alle regolamentazioni non è garantita solamente in un determinato momento ma viene costantemente monitorata tramite l'esecuzione di test ripetuti e frequenti.

[43] (Payment Card Industry Data Security Standard: Requirements and Testing Procedures, 2024)
[44] (Regolamento (UE) 2016/679 del parlamento europeo e del consiglio, 2016)
[45] (Certificazione ISO/IEC 27001, 2013)

Le normative finanziarie sono state e continueranno ad essere soggette a numerosi cambiamenti; queste trasformazioni influenzano indirettamente vari aspetti dell'organizzazione, la quale deve adattarsi a esse. Pertanto, a seguito di un cambiamento normativo, è fondamentale che tutti i sistemi si adeguino per mantenere la loro conformità ed evitare sanzioni. L'evoluzione degli studi sulla realizzazione di metodi automatizzati per tali scopi apporterebbe molti benefici, tuttavia presenta una serie di sfide. Innanzitutto, date nuove modifiche alle regolamentazioni, spesso è necessaria la comprensione e interpretazione umana; in secondo luogo, le variazioni hanno impatti indiretti, poco immediati ma necessari da individuare.[46] Data allora l'evidente necessità di una supervisione umana, sarebbe interessante lo sviluppo di sistemi integrati con l'intelligenza artificiale che, a ogni cambio normativa, modifichino in modo automatico quanto possibile. Ad esempio, un ambiente per lo sviluppo di sistemi automatici potrebbe essere integrato con un sistema di intelligenza artificiale, il quale riceve in input la nuova normativa, la confronta con la precedente e, sulla base delle differenze, modifica il codice del rispettivo test di verifica.

Inoltre, "*Toward Automated Change Impact Analysis of Financial Regulations*"[47] riporta un piano per affrontare l'impatto dei cambiamenti normativi nell'ambito finanziario, individuando 4 attività:

1. Identificazione del tipo di cambiamento normativo: l'individuazione dei cambiamenti deve essere su più livelli, dai cambiamenti testuali alla loro interpretazione legale.
2. Rilevazione e classificazione dei cambiamenti normativi: l'obiettivo è lo sviluppo di metodologie semi-automatiche, tali da assistere gli analisti nella valutazione dei cambiamenti normativi.
3. Tracciabilità dei requisiti: sarebbe utile disporre di un collegamento automatico e diretto fra diposizioni legali e requisiti del software, con l'obiettivo di individuare facilmente i requisiti intaccati dal cambiamento normativo.
4. Analisi dell'impatto del cambiamento: si analizzeranno sia gli impatti diretti della normativa, che quelli indiretti necessari a evitare incoerenze.

Pertanto, lo studio evidenzia l'efficacia di sviluppare strumenti automatizzati che assistano gli analisti a seguito di cambiamenti di normative e conseguenti impatti sui sistemi software.

[46] (Abualhaija, Ceci, & Sannier, 2024)
[47] (Abualhaija, Ceci, & Sannier, 2024)

3.2 Benefici del Test Automation nel settore bancario: caso pratico

Come ampiamente discusso fin ora, l'automazione dei test permette l'ottimizzazione dei processi aziendali, la riduzione dei costi operativi e una maggiore velocità nel rilascio di prodotti e/o servizi.

Nel seguente capitolo analizzeremo i vantaggi offerti dal Test Automation all'interno delle banche, con un focus su due casi pratici. Tale analisi fornirà un quadro concreto di come l'implementazione di test automatizzati possa costituire un vantaggio competitivo per l'ente bancario che lo adotta, migliorando la qualità e la sicurezza delle applicazioni, riducendo gli errori umani e garantendo un monitoraggio continuo e preciso delle operazioni.

Verranno analizzate due differenti tipologie di progetto: il primo riguarderà lo svolgimento di System Test automatici per un'applicazione bancaria, i quali verificano il funzionamento dell'applicazione nel suo complesso, mentre il secondo sarà incentrato sullo svolgimento periodico di No Regression Test (NRT).

3.2.1 Caso I: System Test

Di seguito verrà analizzato lo svolgimento di System Test per una nuova applicazione di trading di un ente bancario.

I test sono da svolgere su due dispositivi: uno con sistema operativo Android e l'altro con sistema operativo iOS. I test concordati con il cliente, per semplicità, sono stati suddivisi in *wave*, ciascuna delle quali presenta sottosezioni specifiche dell'applicazione. Ad esempio, la *wave 1* presenta tre sottosezioni: *Impostazioni*, *Login* e *Menu Aiuto*. Ciascuna di esse presenta un numero variabile di test che verificano il corretto funzionamento delle rispettive sezioni.

Il progetto è stato eseguito in circa 60 giorni lavorativi e sono stati eseguiti un totale di 1144 test, nonché 572 per dispositivo. Ciò implica che in media sono stati eseguiti circa 20 test al giorno. Si tratta di un dato importante e da non sottovalutare se si considera che le attività di un tester non si limitano esclusivamente all'esecuzione di test; difatti, è essenziale gestire l'interazione con gli sviluppatori, aggiornare il sistema gestionale che tiene traccia dei difetti (*defect*) riscontrati nel sistema testato o, ancora, preparare la documentazione da condividere con il cliente per tenerlo aggiornato su eventuali problematiche sorte e sull'avanzamento del progetto.

Per comprendere i benefici dei test automatici rispetto a quelli manuali, verrà effettuata una stima dei tempi di esecuzione del progetto in questione nel caso di test automatizzati.

Per semplicità, si suppone che l'esecuzione delle *wave* sia lineare e priva di sovrallocazione con altri progetti, tale da rendere possibile l'attività da parte di un singolo tester: la giornata lavorativa ha una durata di 8 ore e, come anticipato, non tutte queste ore sono dedicate esclusivamente all'esecuzione dei test. In base all'esperienza maturata in azienda, si stima che circa due ore della giornata lavorativa sono dedicate alle attività collaterali di gestione sopra menzionate; approssimativamente un'ora viene impiegata per attività di allineamento con il team, la gestione di eventuali problematiche o l'assistenza ai colleghi, e infine circa un'ora è riservata alla pausa e al team building. Pertanto, l'esecuzione dei test ha una durata pari a circa 4 ore.

Nel caso dei test automatici, avendo empiricamente rilevato che si eseguono circa 20 test al giorno è immediato effettuare una stima riguardo il tempo di scrittura ed esecuzione di un singolo test, come segue:

$$\frac{20\ test}{4\ ore} = 5\ test/ora$$

$$\frac{60\ minuti}{5\ test} = 12\ minuti\ per\ ciascun\ test\ automatico$$

Sulla base dell'esperienza maturata in azienda, è possibile concludere che un singolo test manuale viene eseguito in media in un arco di tempo inferiore. Questo dato potrebbe apparentemente contraddire la tesi secondo cui l'automazione offre vantaggi significativi, poiché, limitandosi a considerare solo questi dati, l'automazione dei test non converrebbe. Tuttavia, tale conclusione è errata, in quanto è necessario analizzare i processi complessivi coinvolti in ciascuna delle due metodologie per comprendere appieno le differenze e i benefici dell'automazione.

Il *testing* automatico prevede una fase iniziale dedicata alla stesura del codice, seguita dalla sua esecuzione, che viene ripetuta iterativamente fino a quando il test è corretto e non fallisce. Tale fase di scrittura non è presente nei test manuali. Tuttavia, per ciascun test manuale è necessario eseguire l'iter specifico per accedere alla sezione di interesse, che può prevedere procedure lunghe o tempi di caricamento estesi. Questo non è richiesto nei test automatici: una volta scritto il codice per il primo test relativo all'accesso a una determinata sezione, è possibile replicare facilmente il codice per gli altri test relativi alla medesima sezione copiando il frammento che descrive l'accesso a quell'area. Inoltre, dopo aver mappato gli elementi, che spesso si ripetono come le barre di navigazione presenti su tutte le pagine, è possibile scrivere il codice in modo più immediato. Pertanto, la durata dell'attività di scripting diminuisce in modo lineare all'aumentare del numero di test eseguiti per una medesima sezione.

Per chiarire meglio questo concetto, di seguito viene presentato un caso pratico relativo all'applicazione di trading. Consideriamo i test relativi alla sezione di impostazione delle preferenze da parte dell'utente. Inizialmente, questi si concentreranno sulla verifica del layout della pagina in diverse circostanze, come ad esempio nel caso di un profilo utente che possiede un solo *dossier* o di un profilo con più di uno. In particolare, nel primo caso potrebbe essere possibile modificare le preferenze per un solo *dossier* oppure per entrambi, mentre nel secondo caso tale scelta non è disponibile poiché si fa riferimento all'unico *dossier* presente. Ciò determina un layout diverso da verificare. Una volta completati i test di layout e in tale occasione mappati la maggior parte degli elementi, si procederà con la verifica delle diverse funzionalità della sezione. Ad esempio, uno dei test prevede il corretto indirizzamento alla pagina "Aiuto" quando si tappa sul relativo pulsante o, ancora, l'esecuzione del flusso e il corretto messaggio di errore nel caso di modifiche delle preferenze. Pertanto, il codice di test iniziale, nonché quello di accesso alla pagina di impostazioni, rimarrà invariato e, per ciascun test, sarà aggiunto il codice specifico relativo al flusso da verificare, sfruttando gli elementi già mappati.

Se volessimo tracciare l'andamento della percentuale di codice ripetuto nei diversi test relativi alla medesima sezione otterremmo approssimativamente il seguente grafico:

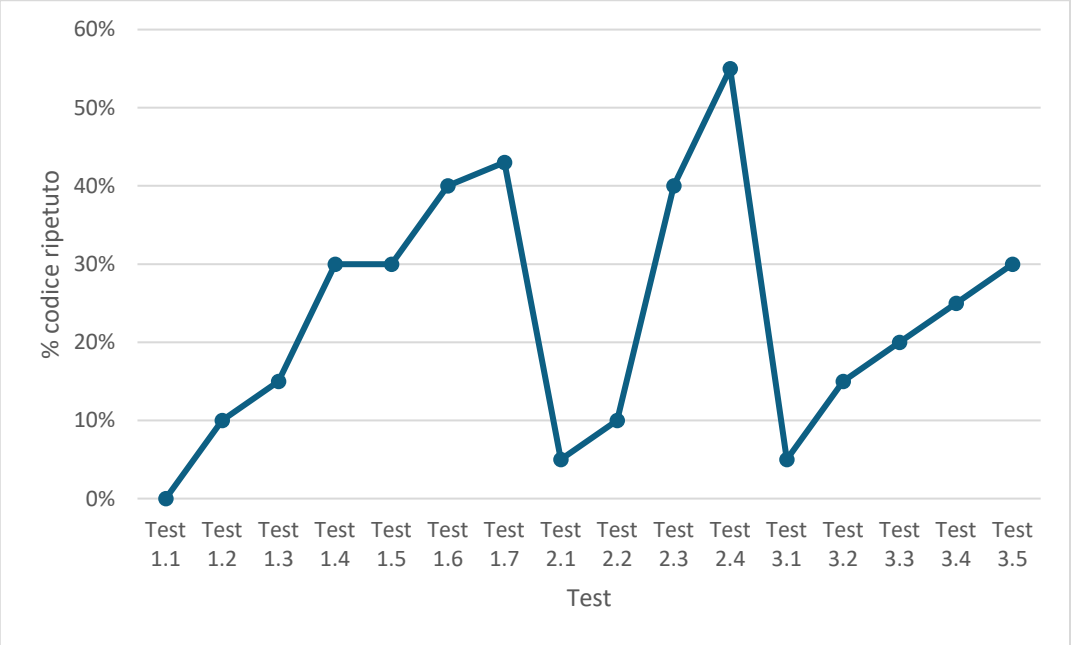


Grafico 3.1 - Andamento % Codice Ripetuto in funzione del Numero di Test.

Il grafico sopra riportato ha scopo puramente esplicativo e mira a facilitare la comprensione della struttura dei test relativi al progetto in questione. Supponendo che i test approfondiscano in modo sequenziale le diverse pagine all'interno della stessa sezione, è evidente che i punti di svolta del grafico corrispondono a nuove pagine. Ad esempio, fino al primo picco i test si concentrano sulla pagina principale di impostazione delle preferenze, includendo verifiche come il layout, la corretta

funzionalità dei pulsanti e così via. Successivamente si testa un'altra pagina, pertanto la percentuale di codice ripetuto inizialmente crolla per poi aumentare nuovamente: in questo caso specifico, dal Test 2.1 si passa ad analizzare il layout e le diverse funzionalità della pagina di conferma delle modifiche effettuate.

Questo esempio illustra come il Test Automation consenta di automatizzare e replicare le operazioni ripetitive tra i vari test, garantendo così un significativo risparmio in termini di tempo e risorse operative. Tale vantaggio si presenta anche per la stesura dei test per i diversi sistemi operativi: il flusso di esecuzione è identico per iOS e Android e l'unica differenza consiste nella mappatura degli oggetti. Pertanto, è possibile utilizzare lo stesso codice per il flusso di test, prestando attenzione a mappare gli elementi specifici per ciascun sistema operativo.

Un ulteriore aspetto da considerare riguarda la possibilità di parallelizzare, almeno in parte, l'attività di scripting con quella di esecuzione. Ad esempio, nel caso di un test con un flusso particolarmente lungo e/o operazioni che richiedono molto tempo, sarebbe ottimale procedere alla scrittura del test successivo parallelamente all'esecuzione del precedente. Tale approccio richiede sicuramente una buona padronanza degli strumenti e una certa esperienza, ma come ribadito più volte finora, è fondamentale avvalersi di tester competenti e preparati.

Infine, l'esecuzione dei test, sia manuali che automatici, è solitamente possibile solo se l'ambiente di test è pienamente funzionante. Tuttavia, può accadere che, a causa di nuovi rilasci dell'applicazione, aggiornamenti o altri fattori esterni ai tester, l'ambiente non funzioni correttamente. In tali circostanze, potrebbe non essere possibile effettuare l'accesso nell'applicazione, navigare al suo interno o accedere alle sezioni da testare. Questa situazione blocca completamente l'esecuzione dei test manuali, ma solo parzialmente quella dei test automatici. Infatti, è comunque possibile scrivere quelle linee di codice ripetute rispetto ai test precedenti o procedere con la stesura di quei test che rimangono inalterati tra le diverse sezioni dell'applicazione. Per chiarire meglio questo concetto, definiamo la tipologia di test uguali tra sezioni diverse. Ad esempio, si consideri la schermata di compravendita di titoli azionari dell'applicazione di trading; la compravendita può avere esito positivo oppure negativo, sulla base degli input forniti. Se i valori di input sono validi, allora l'esito sarà positivo; viceversa, si visualizzerà la schermata con esito negativo. Il flusso per proseguire alla compravendita è il seguente mostrato in Figura 3.1:

Figura 3.1 - Flusso schermata compravendita.

Sia nella schermata di esito positivo che negativo della compravendita, la barra di navigazione superiore è la medesima, e presenta il pulsante "Aiuto" per accedere all'omonima sezione. Si supponga di voler testare la funzionalità di tale pulsante in entrambe le circostanze e che, dopo aver scritto il codice per ottenere un esito positivo l'ambiente non sia più funzionante. In questo caso, sarebbe comunque possibile procedere con lo *script* del test che verifica l'esito negativo e non sarebbe necessario che l'ambiente di test sia perfettamente funzionante, poiché una volta inseriti input non validi per la compravendita (modificando semplicemente i valori forniti), la schermata risulterà quasi identica e il pulsante "Aiuto", essendo già stato mappato, può essere incluso nello *script* senza ricorrere ad un ulteriore mappatura. Pertanto, si tratterebbe di modificare lo *script* del test precedente, modificando solamente i valori di input. Certamente, una volta che le funzionalità dell'ambiente sono state ripristinate sarà necessario proseguire all'esecuzione del test.

In conclusione, gli esempi pratici forniti permettono di comprendere alcune funzionalità e vantaggi dei test automatici e la loro flessibilità.

3.2.2 Caso II: No Regression Test

Di seguito verrà analizzata l'esecuzione di un progetto di No Regression Test (NRT) per l'applicazione di *mobile banking* di un ente bancario.

Come descritto in precedenza, i No Regression Test vengono svolti periodicamente per garantire che l'applicazione continui a funzionare correttamente nel tempo. A differenza dei system test, gli *script* sono già stati realizzati in passato, e l'attività si limita alla loro esecuzione e, se necessario, alla manutenzione.

L'automazione dei test raggiunge la sua massima utilità proprio nei No Regression Test: è sufficiente eseguire gli script e, in caso di fallimento, verificare la causa e gestirla come un *defect* da sottoporre agli sviluppatori. Per queste ragioni, l'esecuzione dei test può essere parallelizzata su più dispositivi, intervenendo solo in caso di fallimento.

Il progetto in questione prevede l'esecuzione di circa 170 NRT in due giorni lavorativi.

Le condizioni operative di un progetto NRT differiscono significativamente da quelle dei System Test. Mentre per questi ultimi erano state stimate circa 4 ore fra *scripting* ed esecuzione, per gli NRT le attività di allineamento con il team e le attività collaterali di gestione risultano essere notevolmente ridotte. In questa tipologia di progetto, allineamenti frequenti o assistenza ai colleghi sono rari, in quanto lo scripting è già stato completato in precedenza e l'attività si limita all'esecuzione dei test. Inoltre, si presuppone che i principali bug dell'applicazione siano stati già individuati nelle fasi precedenti, riducendo notevolmente il tempo dedicato alla gestione dei *defect* e la comunicazione con gli sviluppatori.

In base all'esperienza maturata in azienda, si può stimare che le ore effettive dedicate all'esecuzione dei test in un progetto di questo tipo siano circa 6, all'interno di una giornata lavorativa di 8 ore. Si considera infatti, analogamente al caso precedente, un'ora per la pausa e/o per attività di team building e un'altra ora per attività collaterali di gestione e allineamento con i colleghi.

È immediato allora ricavare il numero di test eseguiti da un singolo tester in un'ora come segue:

$$\frac{170\ test}{2\ giorni * 6\ ore} \cong 14\ test/ora$$

Si tratta di un'efficienza circa tre volte superiore rispetto ai System Test, un dato assolutamente conforme e coerente con la realtà poiché, in questo contesto, si evita la fase di scripting, che impegna buona parte del tempo dei tester, e si prevede la gestione di un numero ridotto di *defect* nell'applicazione. Ciò garantisce una maggiore fluidità nell'esecuzione del progetto, direttamente proporzionale alla manutenzione effettuata sui test e alla robustezza dell'applicazione.

In conclusione, un aspetto di fondamentale importanza, che non può essere trascurato sia nel caso di esecuzione automatica che manuale, riguarda la ripetitività del lavoro e la monotonia che può derivarne per i tester, disincentivandoli a svolgere le attività in modo proattivo e critico.

Attività quali lo scripting dei test, la loro esecuzione oppure l'individuazione degli errori che causano il fallimento di un test sono stimolanti rispetto alla mera esecuzione manuale dei test. Infatti, questi ultimi richiedono una ripetizione costante delle stesse procedure, che può facilmente portare a distrazioni o perdita di interesse, determinando una maggiore probabilità di commettere errori. Al contrario, l'attività di scrittura del codice tramite appositi strumenti permette di ridurre la componente ripetitiva e noiosa, offrendo ai tester la possibilità di creare qualcosa in concreto, mantenendo alta la loro motivazione.

3.3 Legame tra qualità dell'applicazione bancaria e soddisfazione del cliente

Oggigiorno si assiste ad un livello di digitalizzazione elevato nella stragrande maggioranza dei settori, fra cui quello bancario. Esso ha subito una trasformazione notevole, per via dell'incremento dei servizi digitali che garantiscono accesso immediato e continuo alle operazioni bancarie. Difatti, con il crescente impiego della tecnologia mobile, i clienti possono gestire i loro conti bancari, effettuare bonifici oppure visualizzare lo storico delle transazioni direttamente dal proprio cellulare personale. Le app di *mobile banking* hanno raggiunto un livello di innovazione tale da consentire la nascita di istituti bancari completamente digitalizzati, privi di filiali fisiche, ma che offrono un'affidabilità e una sicurezza comparabili a quelle delle banche tradizionali.

Tuttavia, una gestione remota dei servizi può presentare delle criticità in termini di sicurezza e affidabilità da parte del cliente. Se quest'ultimo non considera il servizio offerto come sicuro, affidabile e facile da utilizzare, sarà meno incline a utilizzarlo, preferendo ricorrere ai metodi tradizionali soprattutto per questioni delicate quali affidare i propri risparmi ad un istituto bancario. Per questo motivo, garantire un'elevata qualità delle applicazioni di *mobile banking* è di fondamentale importanza.

Lo studio denominato "*The Influence of Mobile Banking Service ualityon Customer Satisfaction at Bank Rakyat Indonesia Tanjung Karang Branch Office*" dimostra l'esistenza di un'elevata correlazione fra qualità del servizio offerto da un'applicazione mobile di un istituto bancario e la soddisfazione del cliente.[48] La ricerca si basa sulla qualità del servizio offerto dall'applicazione di *mobile banking* della Filiale di Tanjung Karang della BRI (Bank Rakyat Indonesia) e sulla soddisfazione dei rispettivi clienti.

L'idea alla base di tale studio considera la qualità del servizio come un "pilastro strategico" dell'organizzazione per creare un'immagine positiva, fidelizzare i clienti e incrementare la loro fiducia. Si ritiene raggiunta la soddisfazione del cliente nel momento in cui il servizio offerto oltrepassa le sue aspettative e i suoi bisogni; di conseguenza, per fare ciò è necessaria una profonda comprensione del cliente e di quanto l'azienda stessa può offrire.

Il concetto di qualità in un tale contesto bancario comprende vari aspetti, fra cui l'affidabilità del servizio, la reattività e prontezza nel rispondere alle esigenze dei clienti, la sicurezza nella protezione dei dati e nelle transazioni, la facilità d'uso del servizio offerto e l'empatia con i clienti.

[48] (Sucandra & Rinnova, 2024)

La ricerca in oggetto utilizza una metodologia quantitativa, basata su dati raccolti da un campione specifico appositamente selezionato; la raccolta dati avviene tramite strumenti standardizzati quali questionari, moduli oppure osservazioni e l'analisi viene condotta con l'ausilio di strumenti statistici come la regressione, l'analisi della varianza o l'analisi di correlazione. Tale approccio consente di ottenere risultati oggettivi e accurati riguardo il fenomeno in questione, ma trascura aspetti soggettivi quali le percezioni degli intervistati o le loro esperienze.

Il campione selezionato per condurre gli studi statistici ha le seguenti caratteristiche:

- Genere: 62% femminile; il restante maschile.
- Età: la maggioranza, nonché l'83,3% ha un'età compresa fra i 20 e i 30 anni; il 12,5% ha un'età superiore ai 30 anni ma inferiore ai 40. Infine, il restante 4,2% supera i 40 anni.
- Lavoro: il campione è stato suddiviso in 4 sottocategorie per quanto riguarda la tipologia di lavoro, come segue:
    - Studenti universitari: 41,7%
    - Dipendenti pubblici: 3,1%
    - Dipendenti privati: 37,5%
    - Lavoratori autonomi: 17,7%

Il totale del campione ammonta a 96 individui.

È stato condotto il test di normalità, al termine del quale non è possibile rifiutare l'ipotesi nulla che i dati siano distribuiti normalmente con un livello di confidenza del 95%. Pertanto, è possibile proseguire senza particolari limitazioni con l'applicazione dei diversi strumenti statistici.

I fattori analizzati sono la qualità del servizio e la soddisfazione del cliente; dopo essere stati raccolti e analizzati i dati, è stata effettuata un'analisi di correlazione fra i due fattori i cui risultati sono ripotati in Tabella 3.2.

Correlations

		Service Quality	Customer Satisfaction
Service Quality	Pearson Correlation	1	,588**
	Sig. (2-tailed)		,000
	N	96	96
Customer Satisfaction	Pearson Correlation	,588**	1
	Sig. (2-tailed)	,000	
	N	96	96

**. Correlation is significant at the 0.01 level (2-tailed).

Tabella 3.2 - Risultati Analisi di Correlazione.
Fonte: (Sucandra & Rinnova, 2024)

Il valore di correlazione ottenuto fra qualità del servizio e soddisfazione dei clienti è pari a 0,588, il che indica che con un livello di confidenza pari al 99% la qualità del servizio ha un'influenza positiva sulla soddisfazione dei clienti.

Inoltre, è stata effettuata un'analisi di regressione lineare con "Soddisfazione dei clienti" come variabile dipendente. I risultati ottenuti sono riportati in Tabella 3.3.

Coefficients[a]					
Model	Unstandardized Coefficients		Standardized Coefficients	T	Sig.
	B	Std. Error	Beta		
(Constant)	7,398	2,635		2,808	,006
Service Quality	,410	,058	,588	7,050	,000
a. Dependent Variable: Customer Satisfaction					

Tabella 3.3 - Risultati Analisi di Regressione Lineare.
Fonte: (Sucandra & Rinnova, 2024)

Dalla Tabella 3.3 è facilmente ricavabile la seguente equazione:

$$Soddisfazione\ dei\ clienti = 7{,}398 + 0{,}410 * Qualità\ del\ servizio$$

L'intercetta è pari a 7,398 e rappresenta il livello di soddisfazione del cliente quando la qualità del servizio è nulla; per il problema in questione si tratta di un valore privo di significato perché la qualità del servizio effettivamente non può essere pari a zero. Significativo è invece il coefficiente della qualità del servizio, ovvero 0,410: segnala che per ogni unità di aumento nella qualità del servizio allora la soddisfazione dei clienti aumenta di 0,410 unità.

Inoltre, è stato calcolato il valore dell'R^2, che indica la porzione di varianza spiegata dal modello. Si è ottenuto un valore pari a 0,346: ciò significa che il 34,6% della variabilità nella soddisfazione dei clienti è correttamente spiegata dal modello, che possiede come unica variabile indipendente la qualità del servizio. Un'ulteriore considerazione può essere effettuata considerando l'$R^2\ adjusted$, ovvero una versione modificata dell'R^2, che tiene conto del fatto che, all'aumentare delle variabili indipendenti nel modello, anche se non significative, l'R^2 aumenta comunque e questo può essere fuorviante. L'$R^2\ adjusted$ per il caso in questione ha un valore pari a 0,339, molto simile al valore di R^2: ciò significa che il modello è adeguato e, in questo caso, l'unica variabile indipendente del modello è significativa per il problema.

Infine, è stato effettuato il test statistico t-Student con ipotesi nulla che il coefficiente β della variabile indipendente sia nullo, mentre ipotesi alternativa che β sia diverso da 0. Confrontando il T calcolato pari a:

$$T_{calc} = \frac{\beta}{std\ err} = \frac{0{,}410}{0{,}058} = 7{,}050$$

con il valore della T in tabella, corrispondente ad una significatività del 95%, si ottiene:

$$7{,}050 > 1{,}98$$

Ovvero:

$$T_{calc} > T_{tabella}$$

Pertanto, si rifiuta l'ipotesi nulla secondo cui il coefficiente della variabile indipendente "Qualità del servizio" sia nullo e, ancora una volta, si conferma che tale variabile influenza la soddisfazione dei clienti. Si può arrivare a tale conclusione confrontando anche il valore del livello di significatività del p-value, calcolato in Tabella 3.3:

$$0{,}000 < 0{,}05$$

In conclusione, lo studio riportato dimostra l'importanza della qualità del servizio offerto per le applicazioni di *mobile banking* nel garantire una buona soddisfazione dei clienti del settore finanziario; per tale motivo tecniche quali il Test Automation possono contribuire a garantire standard qualitativi elevati. Pertanto, investire in Test Automation per valutare la qualità delle applicazioni nel settore finanziario può apportare un numero elevato di benefici.

4. Analisi di un progetto di Test Automation presso Concept Quality Reply

4.1 Introduzione all'azienda

Reply S.p.A. è una società italiana di consulenza in sistemi informatici, fondata a Torino nel 1996 da Oscar Pepino e Mario Rizzante. Quest'ultimo è attualmente il presidente della società e in contemporanea amministratore delegato, in congiunta con la figlia Tatania Rizzante. Fornisce consulenza ai propri clienti, ideando, sviluppando e gestendo modelli di business che sfruttano i vantaggi introdotti dai nuovi paradigmi tecnologici quali Big Data, Intelligenza Artificiale e così via.

Dal 2006, l'azienda ha ampliato significativamente il proprio raggio di azione espandendosi oltre i confini italiani, registrando una notevole crescita: secondo il Bilancio Annuale del 2023 approvato dall'Assemblea degli Azionisti, l'azienda registra un fatturato di 2.118,0 milioni di Euro, registrando un incremento di circa 12 punti percentuali.[49] Oggigiorno il mondo della consulenza è in continua evoluzione e le grandi aziende si affidano sempre più a tali figure per affrontare le ultime sfide. In particolare, assumono un ruolo sempre più rilevante le società di consulenza specializzate in tecnologie digitali e sostenibilità ambientale. Le grandi aziende, infatti, preferiscono affidarsi a queste realtà specifiche per garantire la propria competitività sul mercato e mantenere un miglioramento continuo in tali ambiti, senza però distogliere l'attenzione dal proprio *core business*.

Reply S.p.A. è una holding che controlla società più piccole, ciascuna specializzata nel proprio settore e ambito, ma stabilisce valori e obiettivi univoci per tutte le sotto società. Ciascuna sotto società a sua volta è costituita da *Business Unit* (BU) specializzate o in un determinato settore o in uno specifico cliente o in entrambi. Ad esempio, la società Concept Reply offre un portafoglio completo per lo sviluppo IoT, a partire dallo sviluppo di software *embedded* fino alle soluzioni *front-end* e *back-end*. È costituita da BU specializzate nel Test Automation, alcune per un unico cliente, ad esempio nel settore bancario, altre specializzate nel Test Automation per tutti i clienti, ad esempio nel settore *retail*. La specializzazione delle Business Unit dipende dalle tipologie dei clienti e dalla mole di attività richieste in ciascun progetto.

Il lavoro di stage e studio dei sistemi automatizzati è stato svolto presso Concept Quality Reply, appartenente a Concept Reply. Quest'ultima si è da tempo divisa in due sezioni: Concept Quality e Concept Engineering. La prima si occupa di svolgere attività di *testing* automatizzati e di validazione

[49] (Reply, 2023)

dei sistemi, mentre la seconda si focalizza sulla progettazione e sviluppo di sistemi hardware o software come, ad esempio, applicazioni mobile o desktop, siti web e così via.

Di seguito verranno analizzati gli strumenti gestionali utilizzati dall'azienda.

4.2 Sistemi di gestione

4.2.1 TAF

Il TAF è una piattaforma sviluppata internamente da Concept Reply, che consente di gestire l'esecuzione dei test su diverse piattaforme e consultarne report e statistiche generati automaticamente. Tendenzialmente l'accesso è limitato al personale Reply poiché esiste un'altra apposita piattaforma dedicata al cliente, descritta nel Capitolo 4.2.2; tuttavia, se il cliente lo richiede, viene consentito loro l'accesso per lanciare in autonomia i test oppure, nel periodo di esecuzione di un progetto, visualizzare l'andamento dei test eseguiti giornalmente.

Tramite un'interfaccia grafica facile ed intuitiva, è possibile pianificare i cicli di test da eseguire da remoto, anche da utenti non strettamente tecnici, selezionando i test di interesse e i dispositivi da utilizzare. Tale framework consente di eseguire test di qualsiasi genere e, pertanto, web, mobile, automotive, API e altro ancora.

Nella dashboard principale della piattaforma è possibile visualizzare le statistiche di esecuzione dei test: ad esempio, è possibile visualizzare il numero di test eseguiti, falliti e passati in un determinato arco temporale. Pertanto, si seleziona un periodo di interesse e si visualizzano le statistiche sopra riportate. È possibile visualizzarle anche in funzione di altri fattori quali la piattaforma utilizzata, se Web o Mobile, oppure il Device (Android oppure iOS). Si tratta di informazioni utili maggiormente al management che ai tester, poiché consentono loro di capire le condizioni di lavoro all'interno del team. Ad esempio, se l'avanzamento giornaliero di un progetto risulta rallentato, è fondamentale analizzare le cause di tale rallentamento. È necessario determinare se il ritardo sia attribuibile a fattori esogeni, come la natura del progetto stesso, o a fattori endogeni, quali la carenza di personale.

Di seguito, in Figura 4.1, vengono riportate le statistiche e i grafici calcolati automaticamente, selezionando l'arco temporale di cui si vuole vedere l'andamento dei test.

Figura 4.1 - Statistiche sezione "Daily Chart".

In Figura 4.1 vengono riportati i grafici della sezione "*Daily Chart*" nella Dashboard principale. Il grafico a sinistra mostra il numero di test eseguiti, passati e falliti in una specifica ora dei giorni selezionati. Il grafico a torta sulla destra mostra le medesime informazioni ma con una granularità maggiore, relativa all'arco di tempo selezionato.

Successivamente si individuano due sezioni fondamentali per il TAF, ovvero il *Test Repository* e il *Text Execution*, strettamente correlati poiché il primo definisce la configurazione dei test, eseguibili tramite la pagina di *Text Execution*. Segue una descrizione dettagliata di tale funzionamento.

L'esecuzione dei test è possibile previo caricamento dei codici elaborati dai tester. Come indicato nel Capitolo 2.2.2, la stesura degli *script* viene effettuata utilizzando il programma IntelliJ IDEA. Una volta configurato il caricamento dei test nella sezione *Test Repository* del TAF, è possibile caricare tramite macchina virtuale un file ZIP contenente gli *script* dei test, estratti da IntelliJ IDEA. Questa procedura presenta una significativa limitazione, comportando un notevole dispendio di tempo poiché, anche una modifica minima al codice deve essere effettuata su IntelliJ IDEA; successivamente, per riportare le modifiche anche sul TAF, è necessario estrarre il nuovo file ZIP aggiornato con le modifiche e caricarlo sulla piattaforma. Tale procedura consente di sovrascrivere la vecchia versione con la nuova, in modo tale che sul TAF sia presente lo script aggiornato.

Si tratta di un processo macchinoso, che richiede una serie di passaggi quali: l'accesso alla macchina virtuale, le operazioni di configurazione del *package* da estrarre da IntelliJ IDEA affinché il file zip sia conforme ai settaggi del TAF, l'estrazione stessa del file e, infine, la riscrittura del nuovo file in sostituzione al vecchio. Sarebbe interessante ottimizzare tale processo oppure sviluppare una sezione integrata, che permetta di modificare il codice direttamente dal TAF.

In conclusione, il *Test Repository* ha lo scopo di definire i test da implementare e configurare le impostazioni necessarie quali il nome del test, la descrizione e altri aspetti prettamente più tecnici. Inoltre, è possibile importare i test tramite apposito file Excel strutturato oppure esportare nel medesimo formato l'elenco dei test inseriti e le rispettive caratteristiche. Di seguito, in Figura 4.2, viene mostrata la schermata del *Test Repository*:

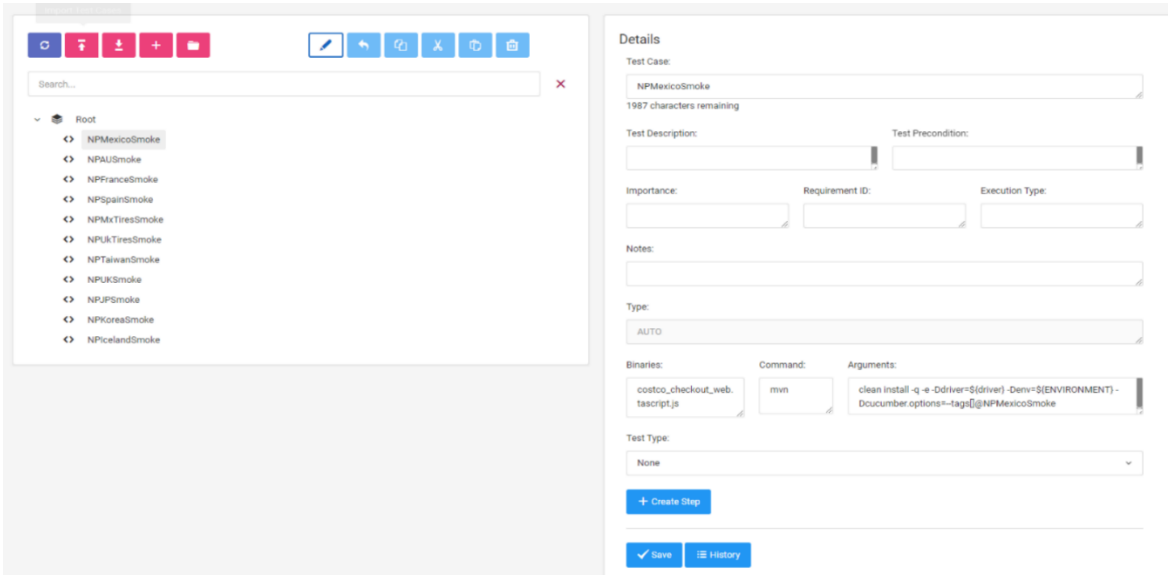


Figura 4.2 - Schermata del Test Repository.

In particolare, i comandi possibili da eseguire su ciascun test vengono riassunti in Tabella 4.1.

	Crea una nuova cartella in cui è possibile inserire i test
	Crea un nuovo test caso di test da inserire
	Scarica sottoforma di file Excel l'elenco dei test e le rispettive caratteristiche, relativi alla cartella selezionata
	Carica i casi di test nella cartella selezionata, a partire dal file Excel
	Abilita la modifica dell'elenco sottostante, permettendo ad esempio di cancellare cartelle o casi di test
	Si tratta dei comandi di copia, ritaglia e incolla
	Elimina l'elemento selezionato

Tabella 4.1 - Comandi del Test Repository.

Per quanto riguarda la selezione e l'esecuzione dei test, queste avvengono tramite la pagina *Test Execution*. In questa sezione è possibile creare un *testset*, nonché un insieme di test da eseguire. La schermata è suddivisa in due sezioni: a sinistra viene visualizzata la struttura ad albero delle cartelle configurate nel Test Repository, mentre a destra sono mostrati le statistiche di esecuzione per ciascuna sottocartella selezionata (ad esempio, numero di test lanciati, passati e falliti).

La struttura dei test segue un'organizzazione ben precisa. Ad esempio, i progetti svolti in un determinato periodo si troveranno all'interno di una stessa cartella. Ogni progetto avrà delle sottocartelle: per un progetto NRT, vi saranno sottocartelle relative alla data di esecuzione dei Test di Non Regressione, mentre per i progetti di System Test, le sottocartelle saranno organizzate in *Wave* o macro-sezioni dei test da eseguire. Le foglie dell'albero rappresentano i test, suddivisi in

base al sistema operativo, Android oppure iOS. La struttura appena descritta, ai fini di una compresione visiva immediata, viene mostrata in Figura 4.3.

Ad esempio, il Progetto 1 prevede l'esecuzione di No Regression Test nel corso del tempo; i test da

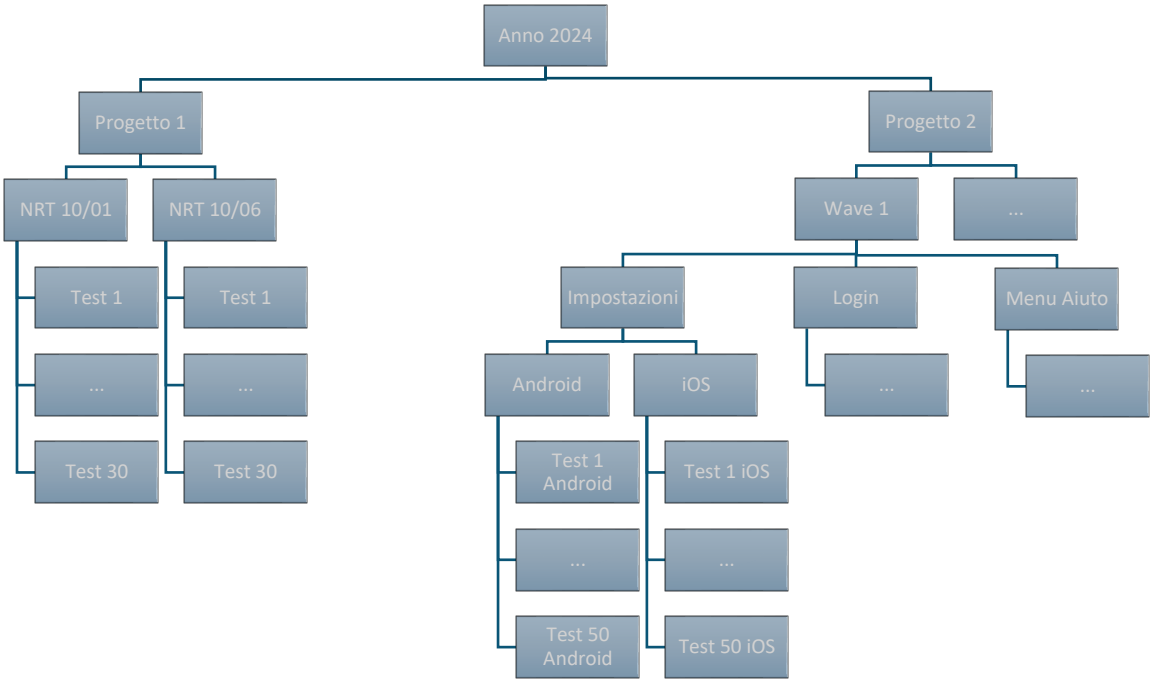


Figura 4.3 - Struttura ad albero dei Test.

eseguire e i rispettivi esiti nel corso delle esecuzioni saranno racchiusi in apposite cartelle. Invece, il Progetto 2 comprende diverse *Wave*, ciascuna delle quali si focalizza su specifiche sezioni da analizzare e verificare.

Selezionate le sottocartelle, è possibile visualizzare il totale dei test e il loro stato e cioè quanti eseguiti, non eseguiti, falliti, non completati oppure N/A ("Not Applicable").

I test non completati sono quei test effettivamente selezionati per l'esecuzione, ma non completamente eseguiti a causa di problemi strutturali del TAF, come mancanza di connessione o simili, o del test stesso (ad esempio, un test vuoto in cui il codice non è ancora stato realizzato). Invece, per quanto riguarda gli N/A si tratta casistiche non testabili a causa di una mancanza di dati. Un caso di esempio di test concluso con esito N/A prevedeva di verificare la corretta visualizzazione di dati in *delay* (ovvero dati non mostrati in tempo reale ma con un certo ritardo) in una specifica sezione di un'applicazione di trading. Nonostante le condizioni al contorno cambiassero, tale sezione non mostrava mai i dati desiderati e, in seguito ad approfondimenti, si è constatato che per quella sezione era stata prevista solamente la ricezione di dati in tempo reale. Si tratta allora di una carenza di dati, tale per cui il test non può essere eseguito; pertanto, il caso di test è stato contrassegnato come N/A. La casistica appena descritta è molto rara; casi decisamente più

frequenti riguardano la mancanza di utenze specifiche con cui eseguire il test, per via di ritardi del team di *data preparation*.

Infine, dalla schermata del TAF selezionando le foglie dell'albero è possibile proseguire con l'esecuzione dei test, indicando il dispositivo su cui eseguirlo. La schermata di selezione del dispositivo viene mostrata in Figura 4.4.

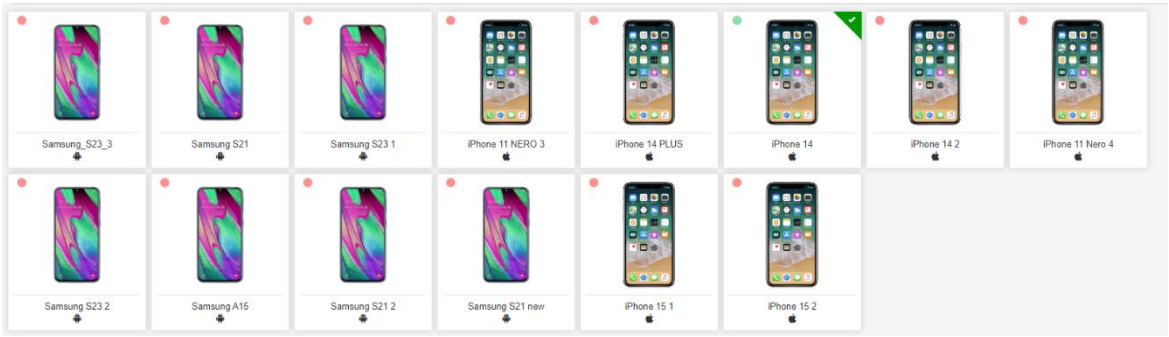


Figura 4.4 - Schermata selezione dispositivo.

Il dispositivo selezionato in quel momento viene contrassegnato con la spunta verde; invece, la disponibilità dei dispositivi viene indicata con il pallino in alto a sinistra: se verde, il dispositivo è libero. Altrimenti, il test verrà messo in coda dato che momentaneamente il dispositivo non è disponibile a causa di problemi tecnici oppure poiché è già in esecuzione un altro test su quel cellulare.

La sezione core della piattaforma TAF è costituita dalla pagina di *Execution Progress*, che consente di monitorare in tempo reale tutti i test in esecuzione in quel momento e visualizzarne l'avanzamento. Vengono mostrate in un elenco le esecuzioni in corso; in ciascuna riga sono riassunte le seguenti informazioni:

- *Request Time*: mostra l'istante in cui è stata inviata la richiesta di esecuzione del test;
- *Start Time*: indica l'istante in cui ha inizio l'esecuzione del test;
- *Test Name*: indica il nome del test che si sta eseguendo;
- *Device*: mostra il nome dispositivo su cui è eseguito il test;
- *Capability*: indica il tipo di dispositivo utilizzato e la rispettiva versione;
- *Started by*: indica il nome del tester che ha avviato l'esecuzione;
- *Cycle Name*: è il nome del Test Cycle, nonché un raggruppamento logico dei test;
- *Status*: indica lo stato di esecuzione del test, che può assumere tre valori: superato, fallito oppure in esecuzione;
- *Progress*: riporta la frazione di step eseguiti sino a quel momento, sul totale di step;
- *Action*: riporta le righe di esecuzione del test.

Durante l'esecuzione del test, è possibile visualizzare le linee di codice eseguite in tempo reale; se il test viene interamente eseguito con successo allora lo stato del test sarà *passed*; al contrario, in caso di fallimento, viene mostrata la tipologia di errore.

Inoltre, è possibile selezionare un test già eseguito e visualizzare lo storico delle diverse esecuzioni, con i rispettivi esiti e il report associato. Quest'ultimo è un file che mostra la linea di codice eseguita e la schermata dell'applicazione al momento di quella specifica riga di codice. Si tratta di uno strumento molto utile in caso di fallimento del test poiché permette di individuare la natura del fallimento; supponiamo che un test fallisca a causa di un elemento non trovato all'interno della pagina (come, ad esempio, il pulsante "Aiuto"). Se nella schermata riportata nel report esso è presente, allora si tratta di un errore nel codice del test poiché effettivamente il pulsante è presente ma non viene trovato nella pagina probabilmente a causa dell'xPath errato o dell'istruzione implementata male. Invece, se il pulsante non è presente, si tratta di un *defect* dell'applicazione da segnalare agli sviluppatori.

In conclusione, il TAF rappresenta uno strumento di gestione interna che consente l'esecuzione dei test, l'elaborazione della relativa reportistica e l'allineamento dei membri del team riguardo lo stato dei diversi progetti.

Per la gestione esterna e la comunicazione, sia con i clienti sia con altre Business Unit appartenenti alla stessa azienda (come ad esempio gli sviluppatori), ma che cooperano ugualmente al progetto, viene invece utilizzato ALM. Quest'ultimo sarà descritto nel capitolo seguente.

4.2.2 ALM – Application Lifecycle Management

Application Lifecycle Management (ALM) è il sistema gestionale sviluppato da Concept Reply, utilizzato per la trasmissione formale di aggiornamenti ai clienti e per la segnalazione dei difetti (*defect*) agli sviluppatori. Il sistema consente l'accesso alle cartelle specifiche dei progetti, ognuna delle quali contiene i test, organizzati secondo una struttura ad albero che segue la stessa logica rappresentata in Figura 4.1. ALM, non essendo un ambiente di sviluppo ed esecuzione dei test, non contiene i codici dei test, ma mantiene l'informazione relativa all'esito del test e agli eventuali difetti ad esso associati. Di seguito viene illustrato un esempio pratico che mostra come ALM viene utilizzato nel caso di apertura e risoluzione dei *defect*.

Si consideri che il Test 1 di un progetto relativo a un'applicazione di mobile banking richiede di verificare la corretta impostazione della pagina di *overview* iniziale. In tale pagina, è previsto un carosello scrollabile che mostra le carte possedute dall'utente, insieme ad altre informazioni

visualizzabili scorrendo la pagina. Il test prevede di verificare la presenza di un pulsante per ogni carta, al cui tap si accede al dettaglio delle transazioni effettuate con quella specifica carta. Questa verifica deve essere effettuata sia per il sistema operativo iOS sia per Android.

Si supponga che nell'applicazione per Android il pulsante non sia presente, mentre nella versione per iOS il pulsante sia presente ma non cliccabile. Questi costituiscono due *defect* da segnalare agli sviluppatori, entrambi associati al Test 1 del progetto, poiché si tratta di errori che invalidano il Test 1 iOS e il Test 1 Android. È importante sottolineare che, ai fini della tracciabilità e della gestione degli errori, è necessario aprire *defect* distinti per ciascun sistema operativo. Nel caso in esame, è evidente la necessità di questa distinzione, poiché si tratta di due errori differenti; tuttavia, anche nel caso in cui l'errore fosse identico sia per iOS che per Android, saranno comunque aperti due *defect* separati, anche se gli step da eseguire e l'errore segnalato coincidono.

Come menzionato in precedenza, è possibile accedere tramite ALM all'esito dei test, separatamente per le versioni iOS e Android. Pertanto, al Test 1 sia per iOS che per Android sarà associato un esito pari a "*Failed*" e sarà correlato un *defect* da inoltrare agli sviluppatori. Quest'ultimo seguirà una struttura ben definita, come di seguito riportato:

- Utenza del tester che sta segnalando l'errore, accompagnato dalle iniziali di nome e cognome;
- Utenza fittizia, denominata anche come User ID, utilizzata per accedere all'applicazione;
- Nome dell'app, accompagnato dalla versione (iOS oppure Android);
- Versione dell'applicazione;
- Tipo di dispositivo utilizzato, ad esempio iPhone 14 con versione iOS 15 oppure Samsung s23 con versione 13;
- Data e ora di apertura del *defect*;
- Step eseguiti;
- Risultato atteso, ovvero ciò che dovrebbe risultare secondo l'analisi funzionale;
- Errore riscontrato.

Infine, ogni *defect* è associato a un ID univoco generato automaticamente dal sistema per facilitarne l'individuazione, e un titolo che semplifica lo smistamento dei *defect* da parte degli sviluppatori. Questo perché, verosimilmente, ci saranno sviluppatori dedicati esclusivamente al sistema iOS e altri ad Android; di conseguenza, il titolo consente di identificare immediatamente a chi assegnare la gestione del *defect* o di comprendere la sua natura, distinguendo tra un problema di *front-end* (ad esempio, l'assenza di un pulsante) oppure un problema che richiede ulteriori approfondimenti (come la mancata ricezione di alcuni dati).

Le informazioni inserite nel corpo del *defect* sono necessarie non solo per tracciare gli errori individuati e la loro successiva risoluzione, ma anche per supportare e guidare gli sviluppatori nel processo di verifica e correzione del problema. In particolare, una volta inoltrato il *defect*, gli sviluppatori procederanno ad un'attenta analisi. Di conseguenza, accederanno all'applicazione utilizzando lo User ID specificato dal tester, seguiranno gli step descritti e verificheranno la natura dell'errore.

La gestione dei *defect* da parte degli sviluppatori può variare in base alle circostanze. Di seguito vengono analizzate alcune di queste situazioni:

- È possibile che gli sviluppatori si rendano conto che il rilascio di una build abbia compromesso alcune sezioni dell'applicazione e provvedano a rilasciare una nuova build corretta. Se i *defect* riguardanti quelle sezioni vengono gestiti dagli sviluppatori dopo il rilascio della nuova *build*, questi contrassegneranno il *defect* come "Respinto Al Tester", indicando in un'apposita sezione dei commenti che il problema non sussiste più e di testarlo nuovamente con la nuova versione. In generale, qualora il problema segnalato non venga riscontrato dagli sviluppatori, o perché si presenta la casistica appena descritta oppure perché effettivamente quell'errore non esiste, il *defect* verrà respinto al tester.
- È possibile che il problema segnalato non rientri nelle competenze degli sviluppatori. Ad esempio, se il tester deve verificare che nella pagina di riepilogo di un bonifico siano riportate correttamente tutte le informazioni indicate nell'analisi funzionale e alcune informazioni risultano mancanti, verrà aperto un *defect*. Gli sviluppatori, in questo caso, verificheranno che il codice sorgente relativo alla pagina sia corretto. Pertanto, se l'informazione mancante è dinamica e proviene da un database, sarà loro compito assicurarsi che la query utilizzata per richiedere tali informazioni sia corretta. Nel caso in cui il problema non dipenda dal codice ma dal database privo di valori, il *defect* non sarà più di competenza degli sviluppatori, ma verrà inoltrato a chi si occupa del *back-end* dell'applicazione, ossia della gestione delle logiche e l'invio dei dati.
- Infine, la casistica più comune riguarda l'individuazione dell'errore da parte dello sviluppatore e la sua conseguente risoluzione, nel caso in cui il problema sia di sua competenza. In tal caso, lo sviluppatore cambierà lo stato del *defect* in "Corretto" e lo inoltrerà al tester. Quest'ultimo, a sua volta, effettuerà una verifica del test utilizzando la versione aggiornata dell'applicazione. Se il problema risulta risolto, procederà con la chiusura del *defect* e aggiornerà lo stato del test associato da "*Failed*" a "*Passed*".

L'esito dei test e i *defect* sono visualizzabili dal cliente, il quale rimane aggiornato sulla tipologia di errori e sull'avanzamento.

Come descritto nel Capitolo 4.2.1, l'ambiente di *testing* utilizzato è il TAF, in cui è possibile caricare ed eseguire gli script dei test su diversi dispositivi. Se il test viene eseguito correttamente, il suo stato sarà automaticamente impostato su "*Passed*"; in caso di fallimento, lo stato sarà "*Failed*". Tali esiti possono essere trasferiti dal TAF ad ALM seguendo una specifica procedura, in modo tale da rendere visibile al cliente, attraverso ALM, lo stato di avanzamento del progetto.

4.2.3 Report giornalieri

Considerando l'ambiente e la metodologia *Agile* adottati da Reply, l'azienda promuove una comunicazione diretta e non troppo formale con il cliente, al fine di comprendere appieno le sue esigenze e ridurre i tempi di comunicazione. Per questo motivo, al termine di ogni giornata lavorativa, i tester inviano al cliente un report sotto forma di file Excel che mostra l'avanzamento del progetto. Viene riportato l'elenco dei test da eseguire, con il rispettivo stato che può assumere tre valori: "*Passed*", "*Failed*" e "*No Run*". Tendenzialmente, ai test in "*Failed*" è associato il *defect* ancora in corso di risoluzione, che determina l'esito negativo di quel test. Un foglio Excel mostra l'elenco dei *defect* aperti per quel progetto e il rispettivo stato, che può essere "Chiuso" e pertanto risolto oppure "In Lavorazione".

Per fornire una visione immediata dello stato di avanzamento, viene riportato un diagramma a torta che illustra la percentuale di test superati, falliti o ancora non eseguiti; tale analisi viene effettuata separatamente per il sistema operativo Android e iOS. La schermata riepilogativa viene riportata in Figura 4.5.

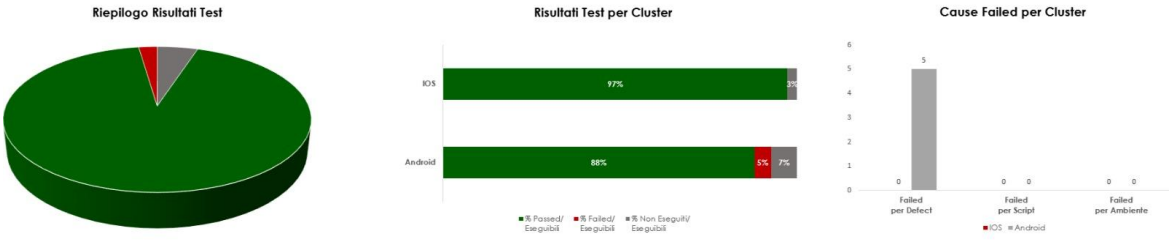


Figura 4.5 - Diagramma a torta riassuntivo del report giornaliero.

Dalla Figura 4.5 emerge chiaramente l'andamento dell'esecuzione in corso: per il caso in questione, 5 test su Android risultano falliti a causa di difetti ancora non risolti, mentre per il sistema operativo iOS non vi sono difetti in corso. Inoltre, vengono evidenziati i test che falliscono a causa di script o di inaccessibilità agli ambienti, sebbene in questo caso entrambe le casistiche non si presentino. La

prima tipologia di fallimento è dovuta ad anomalie nel comportamento dello script automatico rispetto a quanto previsto dal test manuale validato, ma si tratta di una casistica generalmente rara.

Infine, in delle tabelle riassuntive vengono riportate le informazioni numeriche come segue:

- Tabella di riepilogo.

  Separatamente per Android e iOS riporta le seguenti informazioni:

  - Numero di test totali;
  - Numero di test eseguibili, pari alla differenza fra il totale dei test e i test in N/A.
  - Numero dei test eseguiti;
  - Percentuale degli eseguiti;
  - Percentuale dei non eseguiti o eseguibili;
  - Numero di test in *passed*;
  - Percentuale dei test eseguibili in *passed*;
  - Percentuale dei test eseguiti in *passed*;
  - Numero di test in *failed*;
  - Percentuale dei test eseguibili in *failed*;
  - Percentuale dei test eseguiti in *failed*;
  - Numero di test *failed* per script;
  - Numero di *failed* per ambiente;
  - Numero di test in N/A.
- Tabella per il sistema operativo iOS con le numeriche sopra riportate, in funzione delle sezioni da analizzare. Ad esempio, se per un'applicazione bancaria di trading deve essere analizzata la sezione per la Compravendita e quella degli Ordini, allora separatamente per queste due macroaree verranno individuate le metriche sui rispettivi test iOS.
- Tabella per il sistema operativo Android, speculare a quella descritta per iOS.

Di seguito in Figura 4.6 viene mostrato un esempio delle tabelle sopra descritte:

Riepilogo

Status	Cluster	Esecuzione					Risultati									
	Descrizione	Test	Eseguibili	Eseguiti	% Eseguiti/ Eseguibili	% Non Eseguiti/ Eseguibili	Totale Passed	% Passed/ Eseguibili	% Passed/ Eseguiti	Totale Failed	% Failed/ Eseguibili	% Failed/ Eseguiti	Failed per Defect	Failed per Script	Failed per Ambiente	N/A
	iOS	109	108	106	98%	2%	106	98%	100%	0	0%	0%	0	0	0	1
	Android	109	108	106	98%	2%	106	98%	100%	0	0%	0%	0	0	0	1
	TOTALE	218	216	212	98%	2%	212	98%	100%	0	0%	0%	0	0	0	2

IOS

Status	Device	Esecuzione					Risultati									
	Modello	Test	Eseguibili	Eseguiti	% Eseguiti/ Eseguibili	% Non Eseguiti/ Eseguibili	Totale Passed	% Passed/ Eseguibili	% Passed/ Eseguiti	Totale Failed	% Failed/ Eseguibili	% Failed/ Eseguiti	Failed per Defect	Failed per Script	Failed per Ambiente	N/A
	Compravendita	48	48	46	96%	4%	46	96%	100%	0	0%	0%	0	0	0	0
	Ordini	61	60	60	100%	0%	60	100%	100%	0	0%	0%	0	0	0	1
	TOTALE	109	108	106	98%	2%	106	98%	100%	0	0%	0%	0	0	0	1

ANDROID

Status	Device	Esecuzione					Risultati									
	Modello	Test	Eseguibili	Eseguiti	% Eseguiti/ Eseguibili	% Non Eseguiti/ Eseguibili	Totale Passed	% Passed/ Eseguibili	% Passed/ Eseguiti	Totale Failed	% Failed/ Eseguibili	% Failed/ Eseguiti	Failed per Defect	Failed per Script	Failed per Ambiente	N/A
	Compravendita	48	48	46	96%	4%	46	96%	100%	0	0%	0%	0	0	0	0
	Ordini	61	60	60	100%	0%	60	100%	100%	0	0%	0%	0	0	0	1
	TOTALE	109	108	106	98%	2%	106	98%	100%	0	0%	0%	0	0	0	1

Figura 4.6 - Tabelle riassuntive del report giornaliero.

In conclusione, le informazioni fornite nei report giornalieri consentono non solo di monitorare gli avanzamenti e i difetti riscontrati, ma anche di informare il cliente in maniera più immediata e diretta, offrendo una visione completa senza la necessità di consultare il sistema gestionale ALM, descritto nel Capitolo 4.2.2.

4.2.4 Migliorie al sistema di gestione

Le procedure appena descritte rappresentano l'aspetto gestionale e meno tecnico dell'esecuzione di un progetto di Test Automation. Ciò dimostra come non si tratti semplicemente di scrivere ed eseguire i test, ma di gestire una serie di aspetti collaterali. Attualmente, nel caso dell'azienda e della specifica Business Unit presso cui è stato eseguito il tirocinio, tali aspetti sono gestiti dal tester. Implementare tale tipologia di organizzazione e gestione del lavoro presenta certamente dei pro e dei contro, che verranno analizzati di seguito.

Come illustrato nei capitoli precedenti, per garantire l'affidabilità e la consistenza di un progetto di Test Automation è necessario disporre di tester qualificati, a cui fornire una continua formazione e specializzazione, sulla base delle esigenze specifiche della Business Unit presso cui lavorano. Pertanto, richiedere ad un tester di specializzarsi dal punto di vista tecnico e affidagli anche gli aspetti collaterali di gestione del progetto potrebbe determinare un sovraccarico. Pertanto, si avrebbe una carenza o nella gestione del progetto oppure nella parte tecnica. In un tale contesto, sarebbe opportuno difatti separare le due attività di gestione e di esecuzione dei test, introducendo figure specializzate.

Considerando la frequente sovrapposizione di progetti, per evitare confusione e sovraccarico di lavoro, sarebbe ideale disporre di un gestore per ciascun progetto, supportato da circa due o tre tester, a seconda dell'entità del progetto. I tester si concentrerebbero esclusivamente sulla scrittura e ottimizzazione degli script. In caso di individuazione di *defect*, essi comunicherebbero l'errore al gestore del progetto, il quale si occuperebbe della comunicazione con il cliente e sviluppatori, gestendo l'inoltro dei *defect*, l'aggiornamento dei report e le altre attività correlate.

Tuttavia, per la maggior parte dei difetti, salvo per quelli banali e immediati di *front-end*, è necessario conoscere l'applicazione, il suo funzionamento, i flussi da eseguire e, pertanto, sarebbe più immediato ed efficace far gestire quello specifico problema al tester che l'ha riscontrato.

Ecco che, sulla base della tipologia dei progetti assegnati alla Business Unit e sulla disponibilità delle risorse, si opta per l'una o l'altra organizzazione.

Un ulteriore aspetto da valutare riguarda l'assegnazione delle risorse ai progetti; durante il percorso di tirocinio affrontato, si procedeva tendenzialmente con l'assegnazione dei tester ai progetti sulla base delle disponibilità. Ma lavorare a lungo su uno stesso progetto permette di conoscerlo più approfonditamente, di instaurare un rapporto di fiducia con il cliente e comprendere maggiormente le sue esigenze. Pertanto, sarebbe ottimale procedere con un'assegnazione fissa dei progetti al fine di promuovere i vantaggi appena descritti ma, per evitare l'effetto opposto, ovvero la demotivazione dei tester dovuta alla ripetitività del progetto, si potrebbe pensare periodicamente di cambiare le assegnazioni.

A causa della limitata disponibilità di risorse, non è possibile assegnare un individuo esclusivamente a un singolo progetto; anzi, ciò determinerebbe una sotto allocazione delle risorse. Per questo motivo, è necessario che una stessa risorsa lavori contemporaneamente su più progetti, il cui numero varia in base all'importanza e al carico di lavoro di ciascun progetto. Sarebbe pertanto utile valutare un'integrazione del gestionale interno, nonché del TAF, che mostri il carico di lavoro assegnato a ciascuna risorsa all'interno della Business Unit, in modo da guidare consapevolmente le decisioni del *management* nell'accettare nuovi progetti e nel distribuire equamente il carico di lavoro tra le risorse.

Di seguito verranno analizzate le principali sezioni della nuova ipotetica sezione da integrare all'interno del TAF:

- Sezione 1: viene riportato l'elenco dei componenti della specifica Business Unit e per ciascuno di essi vengono mostrati in dettaglio i progetti assegnati e la percentuale svolta in ciascuno. In Figura 4.7 è illustrata una possibile schermata di dettaglio.

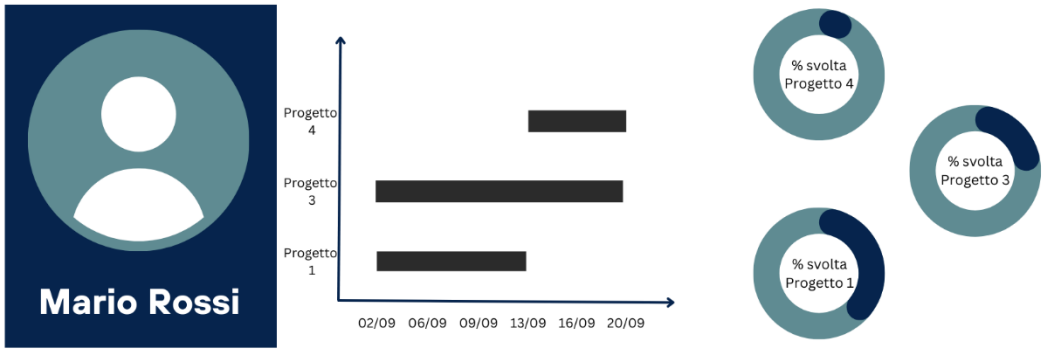


Figura 4.7 - Mockup "Dettaglio componente".

La schermata mostrata in Figura 4.3 evidenzia il carico di lavoro assegnato al tester Mario Rossi e la percentuale di lavoro svolto per ciascun progetto. Questa informazione è utile sotto diversi aspetti. In primo luogo, serve per gli alti livelli di *management* per valutare l'efficienza di quel soggetto e, di conseguenza, orientarsi sulla tipologia di progetti da

assegnargli. Inoltre, può essere visto anche come una sorta di incentivo per il tester, dato che i risultati da lui conseguiti sono trasparenti e visibili a tutti.

La percentuale di completamento raffigurata nei grafici può essere calcolata come segue:

$$\frac{Numero\ test\ svolti\ da\ Mario\ Rossi}{Numero\ test\ totale\ del\ progetto} * 100$$

Il numero di test svolti può essere facilmente ricavato dal TAF poiché, come descritto nel Capitolo 4.2.1, viene tenuta traccia del tester che esegue un determinato test. Allo stesso modo, il numero totale di test di un progetto può essere ricavato contando i test nella cartella di riferimento del progetto.

- Sezione 2: mostra graficamente il lavoro dell'intera Business Unit; pertanto, potrebbe essere utile raffigurare un Gantt in funzione del tempo come mostrato in Figura 4.8:

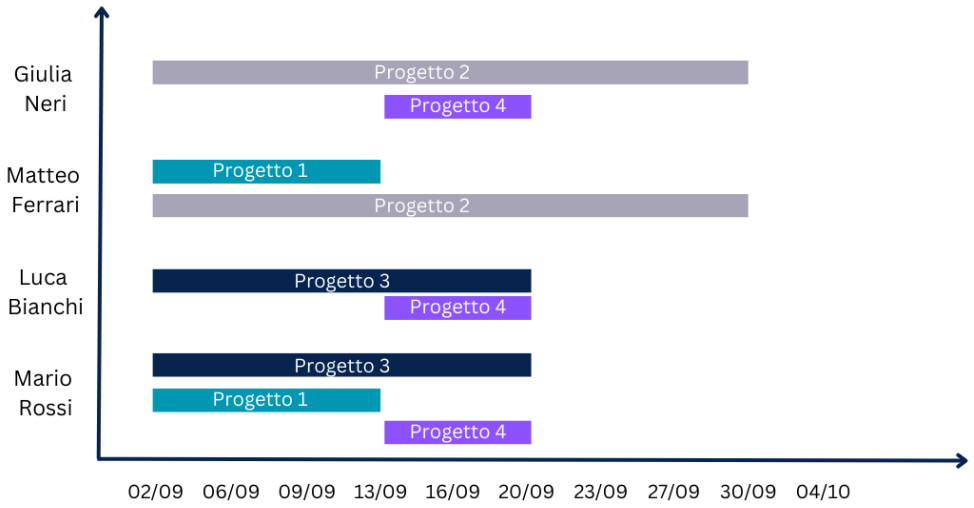


Figura 4.8 - Esempio Gantt.

In questo modo il *management* ha la possibilità di tenere traccia dei progetti in corso, chi lavora su di essi e di visualizzare eventuali criticità nella Business Unit come ritardi nei progetti oppure sovra-allocazione delle risorse. Tuttavia, si tratta di uno strumento particolarmente utile anche per il team stesso, soprattutto se di grandi dimensioni, il quale rimane aggiornato sulle attività presenti e future, e su come le risorse sono state assegnate.

Inoltre, analizzando i problemi sorti durante l'esecuzione dei diversi progetti presso Concept Quality Reply, si suggerisce di seguito una soluzione che però andrebbe a rivoluzionare l'attuale organizzazione e, pertanto, anche non immediata da implementare. Il principale problema durante l'esecuzione dei test, qualsiasi sia l'applicazione di riferimento, riguarda la mancanza di comunicazione e trasparenza fra i diversi attori della catena del valore. Come accennato in precedenza, per una corretta esecuzione dei test è necessario disporre di apposite utenze, create

dal team di *Data Preparation* (*DP*). Essendo il team di *testing* e quello di *Data Preparation* completamente scorrelati, diventa difficile comprendere le necessità reciproche. In particolare, una volta che gli analisti funzionali realizzano gli *scart*[50], stipulano anche un elenco con le utenze necessarie per eseguire i test e le rispettive caratteristiche. Tuttavia, le specifiche non vengono sempre comprese e si realizzano utenze non adatte. Risulterebbe molto più immediato e semplice fornire alla DP i test che dovranno essere eseguiti, in modo tale da comprendere la natura delle utenze da utilizzare. Pertanto, la soluzione proposta prevede la realizzazione di team completi per l'esecuzione di un progetto e non di team orientati alle funzioni da eseguire. In questo modo, vi sarebbe una comunicazione diretta fra DP, tester e sviluppatori, facilitando la comprensione delle rispettive mansioni e un efficientamento del lavoro da eseguire.

Per quanto riguarda, invece, il report giornaliero, per rendere più trasparente l'informazione da trasmettere al cliente sarebbe interessante integrarlo con quanto previsto da schedulazione originaria al fine di rilevare immediatamente eventuali ritardi o anticipi per il termine del lavoro. Pertanto, potrebbe essere utile confrontare il progresso pianificato con quello effettivo e, qualora tale scostamento sia eccessivo e prolungato nel tempo, intervenire opportunamente.

Dall'esperienza maturata presso l'azienda, è emerso che l'andamento del progresso giornaliero varia significativamente in base alla tipologia di progetto. Ad esempio, nel caso di un progetto di No Regression Test (NRT), è plausibile ipotizzare un avanzamento lineare, poiché il codice dei test è già pronto e essi devono solo essere eseguiti. In un tale contesto, supporre che il numero di test eseguiti quotidianamente sia costante e raggiunga valori elevati, risulta verosimile.

Diversamente, in un progetto di System Test, l'esecuzione dei test è necessariamente preceduta da una fase di scrittura. Poiché il software è sottoposto a test per la prima volta, è ragionevole aspettarsi un numero maggiore di *defect* rilevati rispetto ai progetti di NRT. Di conseguenza, nella fase iniziale, l'attenzione si concentra prevalentemente sulla rilevazione e gestione dei *defect*, con un avanzamento limitato dei test, poiché un problema non risolto può bloccare l'esecuzione di diversi test. Man mano che i *defect* vengono risolti e i test sbloccati, si osserva un avanzamento esponenziale, poiché gli script dei test, una volta corretti in relazione ai problemi riscontrati, richiedono solo minime modifiche per essere eseguiti correttamente.

Infine, è fondamentale considerare la natura dei progetti, i quali dipendono strettamente dalla disponibilità dell'ambiente o dell'applicativo da testare. Pertanto, un giorno di malfunzionamenti o

[50] Gli *scart* sono documenti che riportano quanto scritto in analisi funzionale, in step da eseguire per lo svolgimento dei test. Si tratta, pertanto, di una sorta di scaletta con step sequenziali che i tester dovranno eseguire.

di inaccessibilità agli ambienti può causare ritardi nel progetto che risultano difficilmente riducibili o addirittura non recuperabili. Questo rappresenta un rischio significativo che deve essere tenuto in considerazione nell'ambito di tali progetti.

4.3 Sviluppo di un progetto reale

Nel presente capitolo sarà esaminato un caso pratico relativo allo sviluppo di un progetto di System Test per un'applicazione di trading gestita da un ente bancario.

L'applicazione, non ancora resa disponibile agli utenti finali, deve essere verificata con test preliminari prima di essere messa in commercio. In generale, il ciclo di vita di un applicativo può essere descritto come segue:

1. Il cliente fornisce agli sviluppatori l'analisi funzionale dell'applicativo da realizzare.
2. Gli sviluppatori realizzano l'applicazione e conducono gli Unit Test per verificare il corretto funzionamento del codice in ambiente di sviluppo.
3. I tester eseguono i System Test in ambiente di sviluppo, con l'obiettivo di rilevare il maggior numero possibile di difetti, al fine di fornire al cliente il software con un numero ridotto di problematiche.
4. Il cliente esegue gli *User Acceptance Testing* (UAT), ovvero test condotti sempre in ambiente di sviluppo da chi ha commissionato la specifica funzionalità, ossia il cliente stesso. Ad esempio, se l'ente bancario ha commissionato l'ideazione di diverse sezioni dell'applicazione di trading a team diversi, allora coloro che si sono occupati della parte di "Compravendita" effettueranno gli UAT relativamente a quella sezione, così come chi ha ideato la pagina degli "Ordini" eseguirà su di essa i test utente. Poiché si tratta di team appartenenti all'azienda cliente, il lavoro svolto dai tester è cruciale per consegnare un prodotto con il minor numero di difetti.
5. Facoltativamente, possono essere effettuati i cosiddetti test di preproduzione, in cui si testa l'applicazione in un ambiente reale, anziché in quello di sviluppo.
6. L'applicazione viene rilasciata in produzione e resa disponibile all'utente finale.
7. Periodicamente vengono eseguiti i No Regression Test (NRT), per verificare che l'applicazione non subisca regressioni, a seguito di nuove funzionalità introdotte e build rilasciate.

Si noti come per lo sviluppo di un software esistono svariate tipologie di test da eseguire, ciascuno con obiettivi differenti. L'attività di tirocinio è stata svolta presso il team di Test Automation di Concept Quality Reply, il quale si occupa dell'esecuzione di System Test e NRT automatici per le applicazioni dei propri clienti.

Di seguito verrà analizzato l'iter per lo sviluppo e l'esecuzione di un progetto di Test Automation.

In prima battuta, il cliente fornisce l'analisi funzionale e un team addetto in Concept Reply si occupa di analizzarla, al fine di stimare nel dettaglio il numero di test da eseguire, il titolo e il flusso di esecuzione del test. Solitamente, il cliente fornisce in modo indicativo il numero di test che lui vorrebbe, sulla base del *budget* di cui dispone e, pertanto, si cerca di attenersi a quel valore. Salvo casi estremi, si trova un accordo con il cliente e, una volta stabilito il numero di test e il contenuto, un apposito team procede con la realizzazione dello *scart*. Di seguito viene riportato un esempio di quest'ultimo:

 # 0.0 - Effettuare la login con un BT che soddisfi i requisiti del test, accedere al Dettaglio Titolo e tappare su CTA Compravendita in fondo alla pagina

 # 1.0 Verificare che si atterri correttamente in schermata di "Conferma presa visione"

 # 2.0 - Analizzare la navbar
 # Verificare che sia così composta:
 # - Indietro
 # - Titolo "Conferma presa visione"
 # - CTA "Aiuto"

 # 2.0 - Tappare su Aiuto
 # Verificare che si atterri sull'omonima sezione

 # 3.0 - Riatterrare in pagina "Conferma presa visione" e tappare su Indietro
 # Verificare che si atterri sulla schermata precedente

Si tratta di un flusso di operazioni da seguire, tramite le quali si accede a specifiche sezioni da testare. Se il test è automatizzato, a ciascuno step corrisponde una serie di linee di codice che consentono di ottenere il risultato desiderato; ad esempio, per lo step 0 è necessario inserire la riga di codice relativa alla funzione di *LogIn*, che gestisce l'autenticazione e l'accesso all'applicazione, per poi proseguire con il flusso indicato.

Un possibile sviluppo potrebbe riguardare la creazione di *scart* mediante l'uso dell'intelligenza artificiale. Sarebbe opportuno sviluppare un tool che, partendo dall'analisi funzionale, sia in grado di stimare il numero di test necessari, generare i rispettivi titoli e produrre gli *scart*. Sebbene possa

sembrare una sostituzione del lavoro svolto da un analista funzionale, la supervisione e il controllo umano rimangono indispensabili per garantire il soddisfacimento delle richieste del cliente.

Ma prima di procedere con l'esecuzione dei test, sia di System che di Non Regressione, è necessario disporre degli User ID, nonché le utenze con cui accedere alle applicazioni. Si tratta di una fase preliminare all'esecuzione del progetto, senza la quale questo non può essere avviato, e prende il nome di *data preparation*. In Concept Reply vi è un team dedicato alla creazione di utenze con specifiche caratteristiche per l'esecuzione di test mirati.

Analizzando nel dettaglio i System Test da eseguire per l'applicazione di trading, le sezioni da testare sono state suddivise in *wave*, da svolgersi in maniera sequenziale con una parziale sovrapposizione fra la fine di una e l'inizio dell'altra. La struttura dei test concordata è stata la seguente, riportata in Figura 4.9.

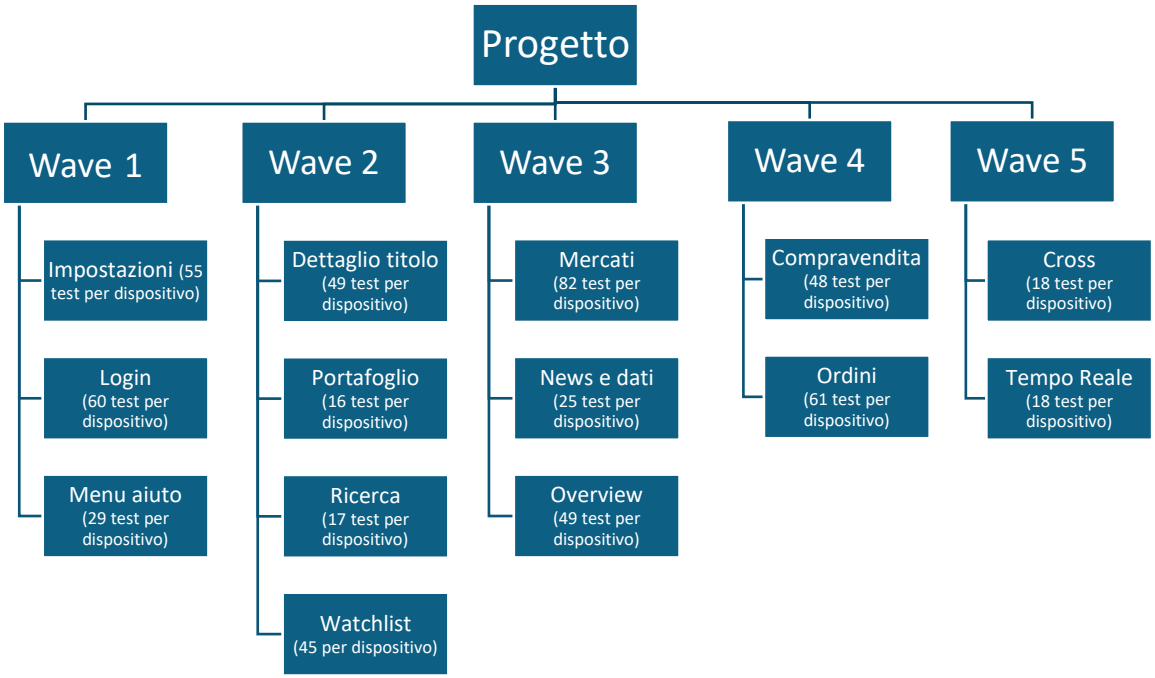


Figura 4.9 - Struttura dei test.

La schedulazione stimata ad inizio progetto viene riportata in Figura 4.10:

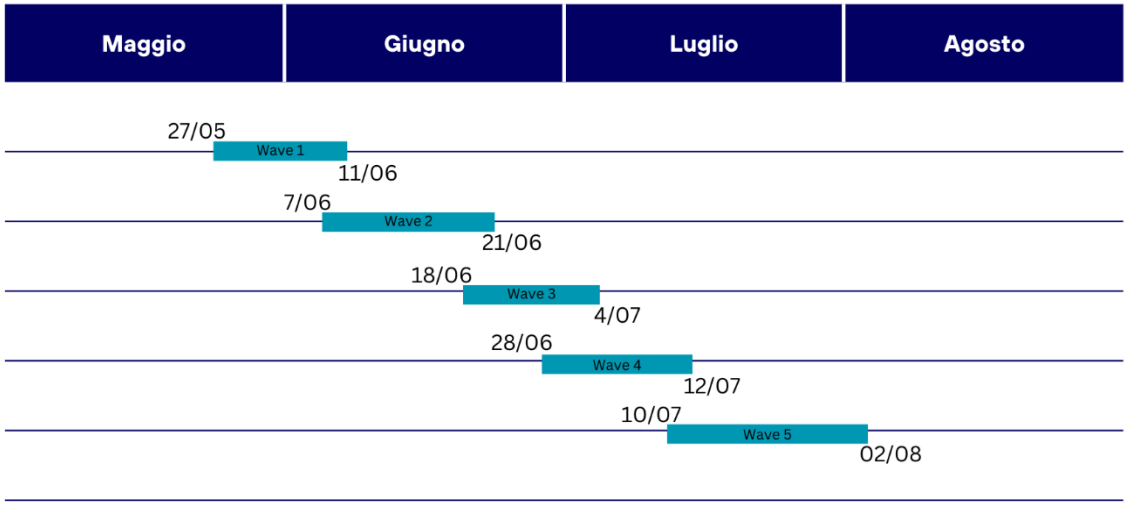


Figura 4.10 - Schedulazione iniziale.

L'accordo raggiunto con il cliente riguardo le scadenze e il numero di test da eseguire è risultato compatibile con il carico di lavoro del team in quel periodo. Considerando la presenza di altri due progetti in contemporanea, non è stato possibile assegnare una risorsa interamente dedicata al progetto dell'applicazione di trading. Per tale motivo, è stata stimata l'allocazione di due tester al progetto, i quali contemporaneamente si occupavano di un'altra attività.

In particolare, l'organizzazione interna del team e la suddivisione del lavoro sono state pianificate come segue: dopo aver effettuato un'analisi preliminare dei test, al termine della quale non sono state rilevate situazioni particolarmente critiche o prioritarie per l'esecuzione di specifici test, è stato calcolato il numero di test da scrivere ed eseguire giornalmente per assicurare il rispetto delle tempistiche del progetto. Di conseguenza, è stato deciso il numero di risorse da allocare, tenendo conto della compatibilità con altri progetti in parallelo, come mostra la Tabella 4.2.

	Test giornalieri	Numero risorse necessarie
Wave 1	$\frac{(55+60+29)*2\ test}{12\ giorni} = 24\ test\ al\ giorno$	1 interamente dedicata al progetto oppure 2 parzialmente
Wave 2	$\frac{(49+16+17+45)*2\ test}{11\ giorni} = 23\ test\ al\ giorno$	1 interamente dedicata al progetto oppure 2 parzialmente
Wave 3	$\frac{(82+25+49)*2\ test}{13\ giorni} = 24\ test\ al\ giorno$	1 interamente dedicata al progetto oppure 2 parzialmente

Wave 4	$\frac{(48+61)*2\ test}{12\ giorni} = 18\ test\ al\ giorno$	1 interamente dedicata al progetto oppure 2 parzialmente
Wave 5	$\frac{(18+18)*2\ test}{18\ giorni} = 4\ test\ al\ giorno$	1 parzialmente

Tabella 4.2 - Allocazione risorse.

Le stime sopra riportate riguardo al numero necessario di risorse, sulla base dei test giornalieri da eseguire, sono state concordate con il *management* in azienda ma sono perfettamente compatibili con le stime effettuate nel Capitolo 3.2.1, secondo le quali un tester è in grado di eseguire in media 20 System Test giornalmente.

Considerando che, una risorsa parzialmente dedicata al progetto contribuisca circa la metà rispetto ad una risorsa interamente dedicata, è immediato comprendere l'equivalenza fra una persona interamente allocata al progetto oppure due parzialmente dedicate. Pertanto, per il progetto in questione si è optato per l'allocazione di due persone sul progetto, in contemporanea dedicate anche ad altri progetti.

Di seguito verrà effettuato un confronto tra quanto schedulato e quanto effettivamente eseguito, con particolare attenzione alle *wave 3*, *4* e *5*, nonché le attività direttamente seguite durante il percorso di tirocinio.

La *wave 1* è iniziata e terminata correttamente, secondo quando previsto da schedulazione. Sono stati individuati 11 *defect*, tutti correttamente risolti.

La *wave 2* è iniziata il 7/06, come preventivato, e al 21/06 presentava uno stato di avanzamento pari al 99%, a causa di un *defect* ancora da risolvere che bloccava l'esecuzione di un test per dispositivo, per un totale di 2 test falliti su 254.

In particolare, l'analisi funzionale prevedeva che le notizie non ancora lette dovessero essere visualizzate con il carattere in grassetto. Tuttavia, dopo vari scambi interni tra gli sviluppatori e il cliente, tale *defect* è stato chiuso poiché si è trattato di una "*Change Request*" (CR), ovvero una richiesta del cliente per apportare modifiche a quanto concordato nella fase iniziale. Nel caso in questione, l'azienda cliente ha ritenuto non necessaria la visualizzazione in grassetto delle notizie non lette; di conseguenza, la *wave 2* è stata completata con successo, sebbene con un ritardo di 9 giorni, e con un totale di 6 *defect* correttamente gestiti e risolti.

Come si evince dalla situazione appena descritta, i ritardi non dipendono sempre da un'inefficiente allocazione delle risorse, ma spesso sono dovuti a cambiamenti nelle richieste durante il processo,

che richiedono tempo per essere approvati internamente dall'azienda cliente e per ristabilire un accordo contrattuale che sia vantaggioso per entrambe le parti, azienda di consulenza e cliente, senza impattare significativamente su quanto stabilito inizialmente.

Parallelamente era in corso la *wave 3*, il cui andamento viene mostrato nel Grafico 4.1. Esso riporta l'andamento della percentuale di test eseguiti e il numero di difetti in corso nel tempo: la linea spezzata indica il progresso giornaliero, mentre l'istogramma mostra il numero di *defect* ancora in corso e non risolti al termine di ciascuna giornata lavorativa.

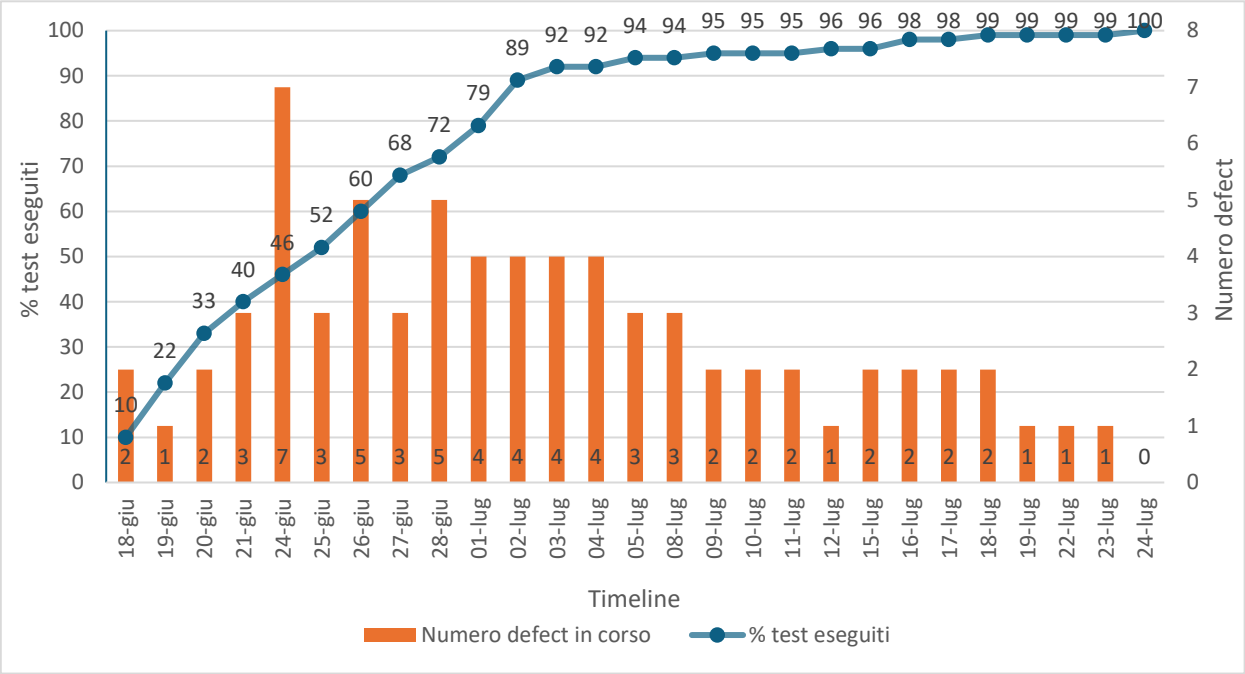


Grafico 4.1 - Andamento wave 3.

Alla data prefissata per il termine della *wave* in oggetto, ossia il 4 Luglio, si è raggiunto il 92% dei test eseguiti. È importante notare come la situazione sia rimasta invariata rispetto al giorno precedente, poiché i 4 *defect* in corso non hanno registrato progressi. Questi *defect* erano di competenza del *back-end*, gestito da un'entità esterna all'azienda Concept Quality Reply per il progetto in questione, e hanno richiesto diversi giorni per essere risolti.

Un *defect* particolarmente problematico riguardava la visualizzazione della schermata contenente l'elenco di strumenti finanziari, nello specifico le opzioni: il flusso per accedervi veniva correttamente eseguito solo nel caso di opzioni "americane", mentre per quelle "europee" compariva un messaggio di errore, dovuto al fatto che le informazioni relative a queste ultime non venivano correttamente inviate. A questo punto, è stato necessario capire come gestire la problematica e trovare un accordo fra responsabile dello sviluppo *back-end* e cliente; alla fine è

stato deciso di chiudere il *defect* a seguito di una *Change Request* (CR) e svolgere i test rimanenti utilizzando le opzioni americane.

Per quanto riguarda tutti gli altri *defect*, questi sono stati correttamente chiusi e risolti, nonostante la lentezza nel processo, probabilmente dovuta a un sovraccarico di lavoro dell'azienda responsabile del *back-end* e al conseguente rallentamento per lo smaltimento delle richieste.

Pertanto, la *wave* 3 termina con un notevole ritardo, non attribuibile ai tester dei System Test, con un totale di 20 *defect* rilevati.

Alla luce della situazione descritta, la *wave* 4 ha subito uno slittamento: anziché iniziare il 28/06, è stata avviata il 05/07, non appena la situazione della *wave* precedente si è stabilizzata. È bene precisare che le modifiche alle scadenze sono concordate con il cliente, il quale è parte integrante del processo di esecuzione dei test e riceve aggiornamenti giornalieri sull'avanzamento dei test o su eventuali problemi.

Per il progetto in questione, oltre agli aggiornamenti forniti tramite report giornalieri, descritti nel Capitolo 4.2.3, erano previste due chiamate infrasettimanali tra tester, sviluppatori e cliente per discutere di eventuali punti critici e concordare insieme come risolverli, in conformità con i principi della metodologia *Agile*.

L'andamento della *wave* 4 è stato il seguente, descritto nel Grafico 4.2:

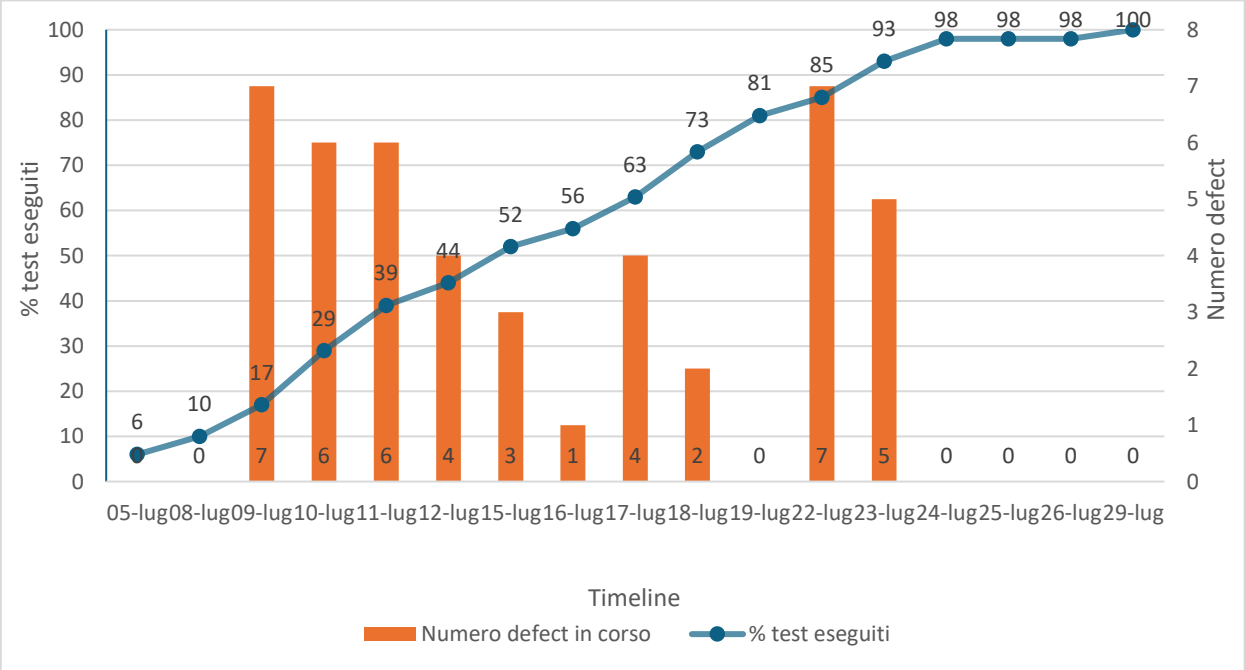


Grafico 4.2 - Andamento wave 4.

Il ritardo nell'inizio dell'esecuzione ha inevitabilmente causato uno slittamento nella data di conclusione: con un totale di 27 difetti riscontrati, la *wave* 4 si è conclusa in ritardo il 29 luglio.

Come evidenziato dall'andamento riportato nel Grafico 4.2, gli ultimi quattro giorni sono stati piuttosto stabili, senza alcun progresso e senza ulteriori difetti rilevati. Il 2% dei rimanenti test riguardavano una sezione dell'applicativo non ancora sviluppata e, di conseguenza, non testabile. È stato quindi necessario attendere il completamento dello sviluppo di tale sezione per poter scrivere ed eseguire i relativi test.

Una particolare circostanza si è verificata durante un test riguardante la sezione ordini, in cui era richiesto di analizzare gli ordini storici dei titoli e, in particolare, di verificare il corretto flusso per la modifica di un ordine storico. Un ordine è considerato storico quando risulta concluso con uno dei seguenti stati: rifiutato, ineseguito o eseguito; pertanto, la modifica di un ordine concluso non risulta logicamente corretta. Inoltre, nell'applicazione di System il pulsante "Modifica" relativo a un ordine storico non era mai presente. Per tali ragioni, durante una chiamata infrasettimanale, il problema è stato esposto al cliente, il quale, dopo consultazioni interne, ha valutato che il test fosse invalido e, pertanto, si è proseguito con la sua eliminazione. A seguito di questa scelta l'analisi funzionale è stata modificata, eliminando la parte riguardante la modifica di un ordine storico.

Infine, parallelamente alla *wave 4*, il 17 luglio ha avuto inizio la successiva e ultima fase. Come evidenziato dai calcoli riportati in Tabella 4.1, il tempo stimato per il completamento di questa fase risultava significativamente superiore rispetto al carico di lavoro effettivamente previsto. Ciò era dovuto al fatto che, in fase di pianificazione, era stato considerato anche lo svolgimento di test su una sezione aggiuntiva, poi annullato. Di conseguenza, nonostante i ritardi accumulati nel corso del progetto, la *wave 5* si è conclusa in anticipo il 29 luglio, senza particolari criticità e con l'individuazione di soli 3 difetti.

In conclusione, nonostante le *milestone* intermedie del progetto abbiano subito significativi ritardi, il progetto è terminato in tempo senza particolari criticità. Sono stati individuati complessivamente 67 difetti, un dato rilevante soprattutto in relazione al tempo a disposizione; infatti, la possibilità di eseguire un numero così elevato di test, individuando un numero considerevole di difetti, è reso possibile dall'adozione di tecniche di Test Automation. Eseguendo i test manualmente, sarebbe stato più probabile incorrere in errori umani, i quali avrebbero potuto impedire l'individuazione di alcuni *defect*, seppur banali ma fondamentali per aumentare il livello qualitativo dell'applicazione.

Riguardo il tema degli errori nel caso di esecuzioni manuali, di seguito viene riportato un esempio che dimostra l'efficacia dei test automatizzati per la riduzione degli errori umani: un test richiedeva di verificare il corretto popolamento di un menù a tendina per l'ordinamento dei risultati, i cui valori da visualizzare dovevano essere "Ordine crescente" e "Ordine decrescente". Dopo la segnalazione agli sviluppatori e la loro correzione, visivamente sembrava tutto corretto, tuttavia, il test

automatico falliva. Andando a visualizzare l'errore fornito da Intellij IDEA, programma utilizzato per la stesura ed esecuzione dei test descritto nel Capitolo 2.2.2, non veniva correttamente individuata la dicitura "Ordine decrescente"; effettuando un controllo visivo più accurato, effettivamente si è individuato un errore di battitura: al posto di "Ordine decrescente" nell'applicazione veniva riportato "Ordine decescente". Questo esempio dimostra come l'implementazione di tecniche di Test Automation consenta di ridurre gli errori umani, garantendo una maggiore precisione nell'individuazione dei difetti.

Infine, conclusa l'esecuzione dei System Test, il cliente ha richiesto di effettuare un re-test di alcune sezioni che avevano subito dei cambiamenti, al fine di verificare che le funzionalità fossero rimaste invariate. In particolare, sono stati eseguiti 4 test per ciascun sistema operativo, per un totale di 8 test e, in media, si impiega circa 4 minuti per l'esecuzione di ciascuno. Tuttavia, l'esecuzione è stata completata in circa 15 minuti grazie alla parallelizzazione di esecuzione tra i due dispositivi, considerando pertanto che in 4 minuti sono stati eseguiti 2 test (uno per iOS e uno per Android).

Per effettuare un confronto, il re-test è stato anche eseguito manualmente; i problemi emersi sono stati i seguenti:

- Difficoltà nel comprendere il flusso: in particolare, si trattava di test appartenenti alla *wave 1*, eseguiti circa 3 mesi prima, il cui *scart* non era molto dettagliato. Ad esempio, per testare la pagina di dettaglio di un titolo, lo *scart* indicava semplicemente di "accedere alla pagina di dettaglio titolo, tramite uno dei punti di aggancio" poiché esistono diversi modi per accedervi, spiegati dettagliatamente nell'analisi funzionale.
  Nel caso del re-test tramite automazione, il flusso viene eseguito automaticamente e, pertanto, non è necessario cercarlo in analisi funzionale; al contrario, nel caso di re-test eseguito manualmente e dopo diverso tempo, è necessario consultare nuovamente l'analisi funzionale per individuare il punto di aggancio per poi procedere con l'esecuzione.
- Impossibilità di parallelizzare i test: se il re-test viene eseguito manualmente da un unico tester, i test devono essere condotti in modo sequenziale e separatamente su ciascun dispositivo, senza possibilità di parallelizzazione.

Considerando il tempo necessario per consultare l'analisi funzionale e individuare un punto di aggancio, oltre all'esecuzione di ciascun test, il tempo medio impiegato per il re-test manuale è stato il seguente:

$$4\ minuti * 8\ test = 35\ minuti$$

a cui è necessario sommare il tempo necessario per consultare l'analisi funzionale e individuare i flussi. Pertanto, il tempo richiesto per l'esecuzione manuale è stato più del doppio rispetto a quello necessario per i test automatizzati. Inoltre, va considerato che la risorsa impiegata nell'esecuzione dei test manuali non può svolgere in contemporanea altre attività, a differenza dei test automatizzati che, parallelamente alla loro esecuzione, consentono al tester di dedicarsi ad un'altra mansione poiché è necessario il suo intervento solo in caso di fallimento del test. Ecco che, con l'esecuzione di test manuali si ha un dispendio di tempo e si perde in efficienza operativa.

4.3.1 Valutazione interna del progetto e best practice

Come precedentemente accennato, l'obiettivo principale nella fase di System Test è identificare il maggior numero possibile di difetti, poiché successivamente il prodotto verrà consegnato al cliente, il quale effettuerà ulteriori test per verificare l'usabilità del software. Per massimizzare l'efficienza, è fondamentale che l'organizzazione interna sia ben strutturata e costantemente monitorata, per comprendere gli aspetti chiave che hanno portato al successo o meno di un progetto. Pertanto, è opportuno che al termine di ogni progetto si effettui una valutazione interna, per trarne insegnamenti utili al futuro, migliorando così la qualità interna e, conseguentemente, il servizio offerto ai clienti.

Una prima valutazione interna da effettuare al termine di un progetto riguarda l'esito degli User Acceptance Testing (UAT): se durante gli UAT si riscontra una bassa difettosità dell'applicativo, allora ciò implica che i System Test sono stati eseguiti correttamente e che quanto consegnato al cliente era già in buona parte privo di difetti; al contrario, se gli UAT evidenziano numerose problematiche, significa che i System Test non sono stati abbastanza efficaci. In casi estremi, potrebbe accadere che il cliente richieda delle spiegazioni riguardo il fallimento di test, in fase UAT, riguardo sezioni che precedentemente erano state dichiarate come funzionanti durante la fase di System Test. Questa discrepanza può essere spiegata da diversi fattori:

- Esecuzione inesatta dei System Test: è possibile che i test siano stati mal eseguiti. In particolare, se manuali, l'errore potrebbe derivare da disattenzioni del tester; se automatici l'errore potrebbe essere stato causato dall'omissione di alcuni controlli da inserire nel codice. Si tratta di una circostanza che accade raramente, ma non impossibile.
- Modifiche nel codice o nelle funzionalità dell'applicazione: una casistica decisamente più frequente riguarda il caso in cui ci siano stati nuovi sviluppi oppure *Change Request* a seguito dell'esecuzione dei test che riguardavano quelle specifiche sezioni. In una tale

circostanza, i test eseguono controlli che, a seguito dei cambiamenti introdotti, non risultano più validi.

- Interferenze con l'ambiente esterno: l'ambiente di System Test è controllato e predisposto per rispettare determinate circostanze, mentre per gli UAT si integra il sistema simulato con quello reale (ad esempio, vengono utilizzati i dati reali) e tale integrazione potrebbe causare dei problemi che nella fase di System Test non si verificavano.

In queste situazioni è bene analizzare dettagliatamente il problema e individuarne la causa, valutando se il problema è causato da una scarsa copertura e inefficienza dei System Test oppure per problemi esterni non imputabili ai tester.

Analizzando l'andamento degli UAT nel caso reale del progetto dell'applicazione di trading, è possibile affermare che i System Test sono stati ben eseguiti. Nella Tabella 4.3 sottostante sono riepilogati i risultati ottenuti per ciascun sistema operativo:

	iOS	Android
Indice di difettosità globale	4,17%	8,79%
Indice di difettosità applicativa	2,91%	3,21%
Indice di riciclo medio	1	1,03

Tabella 4.3 - Risultati UAT.

La valutazione viene effettuate tramite diversi indici: difettosità globale, difettosità applicativa e indice di riciclo medio. Il primo fa riferimento al numero di test falliti sul totale, a causa di *defect*: ad esempio, se un test fallisce poiché il titolo della pagina risulta essere "Compravendite" anziché "Compravendita", questo viene conteggiato nella difettosità globale. Tuttavia, non rientra nella difettosità applicativa, la quale si riferisce ai difetti riguardanti le funzionalità specifiche dell'applicazione. Un esempio di difetto applicativo si verifica ad esempio quando, nel processo di compravendita, il pulsante "Continua" si attiva anche se non tutti i campi obbligatori sono stati compilati: si tratta di un difetto funzionale che non soddisfa le logiche di business. Difetti funzionali più gravi possono includere, ad esempio, il mancato inoltro di ordini o problemi nella visualizzazione del portafoglio azionario. Infine, viene calcolato l'indice di riciclo medio, il quale misura quanti cicli sono necessari, in media, per correggere un difetto. Un valore pari a 1 indica che, mediamente, ogni difetto è stato risolto al primo ciclo di test. Al contrario, un valore superiore a 1 indica che la risoluzione di ogni difetto ha richiesto più di un ciclo. Ad esempio, se un difetto viene identificato e, una volta risolto, il test corrispondente non fallisce al primo ciclo di re-test, l'indice sarà pari a 1. Se invece il difetto viene risolto dopo 2 cicli, l'indice sarà superiore a 1.

Analizzando adesso le numeriche, è evidente che i risultati prodotti dagli UAT sono positivi. Considerando che i test UAT sono stati 455 per dispositivo è immediato ricavare il numero di test falliti:

$$iOS: 455\ test * 4{,}17\% = 19\ test$$

$$Android: 455\ test * 5{,}01\% = 23\ test$$

Di questi, quelli relativi a funzionalità dell'applicazione sono:

$$iOS: 455\ test * 2{,}19\% = 10\ test$$

$$Android: 455\ test * 2{,}63\% = 12\ test$$

Sebbene il numero di test, preso come valore assoluto, possa apparire elevato, è importante considerare che il totale dei test effettuati è consistente. Pertanto, valutando i risultati esclusivamente in termini percentuali, si può affermare che siano stati soddisfacenti. Come accennato in precedenza, questi test sono condotti su un'applicazione in continua evoluzione, che può subire modifiche o problemi derivanti dall'integrazione con sistemi esterni, pertanto rilevare 0 difetti negli UAT è utopico anche se è l'obiettivo a cui ambire.

Grazie all'esperienza maturata in azienda, si propongono una serie di indicatori aggiuntivi per valutare più approfonditamente i progetti e sviluppare *best practice* che consentano di trarre insegnamenti utili per la gestione del lavoro futuro, ottimizzando i processi interni e, pertanto, la qualità offerta al cliente.

Soprattutto nel caso di progetti di No Regression Test, è opportuno considerare un indice di stabilità dell'applicazione, che tiene conto dell'indisponibilità dei servizi o di accesso dell'applicazione sottoposta a test. Infatti, può verificarsi che l'applicazione non sia accessibile per diverse ore, bloccando l'esecuzione dei test e causando ritardi rispetto ai tempi previsti. Quando si verificano i periodi di indisponibilità, si procede con la comunicazione al cliente di tale disservizio. Esso viene anche tracciato giornalmente nei report, poiché se l'avanzamento giornaliero è ridotto a causa dei servizi, questo deve essere reso noto.

Per tale ragione, al termine del progetto, si propone di valutare la disponibilità del sistema attraverso un indicatore che esprima l'instabilità del sistema, calcolato come segue:

$$Indice\ di\ instabilità = \frac{Ore\ totali\ di\ indisponibilità}{Ore\ totali\ del\ progetto}$$

Se questo valore risultasse elevato, sarebbe utile discuterne con il cliente per proporre un prolungamento dei tempi di esecuzione nel ciclo successivo di NRT. Questa valutazione non solo

garantisce un margine di sicurezza aggiuntivo per i tester, ma offre al cliente una visione sulla stabilità dell'applicazione, consentendogli eventualmente di intervenire con misure correttive. In questo modo, oltre a ottimizzare l'organizzazione interna dei test, si fornisce al cliente un'ulteriore valutazione qualitativa.

Per gli stessi obiettivi sopra indicati, sarebbe utile valutare anche un indice di risoluzione dei difetti relativi a un determinato progetto. I difetti riscontrati durante i test vengono comunicati e risolti dagli sviluppatori, che variano di progetto in progetto. Come evidenziato dai risultati descritti nel Capitolo 4.3, si possono verificare dei periodi in cui il lavoro dei tester si interrompe, in attesa della risoluzione dei difetti rimanenti, per poi procedere al re-test e concludere l'esecuzione. Se questa situazione dovesse verificarsi frequentemente, è importante tenerne conto tramite un indicatore e segnalarla al cliente. L'indice di risoluzione dei difetti potrebbe essere calcolato direttamente dalla piattaforma utilizzata per la gestione dei *defect*, ovvero ALM descritta nel Capitolo 4.2.2, poiché tiene traccia delle tempistiche di apertura e di chiusura dei difetti. In particolare, l'indice di risoluzione media dei difetti potrebbe essere calcolato come segue:

$$Tempo\ medio\ di\ risoluzione\ dei\ difetti = \\ = \frac{\sum_{i=1}^{Numero\ Totale\ Difetti}(Data\ risoluzione - Data\ apertura)}{Numero\ Totale\ Difetti}$$

Se tale indice risultasse particolarmente elevato, potrebbe essere opportuno discuterne con il cliente: in alcuni casi, un valore elevato potrebbe dipendere dalle criticità dei difetti riscontrati, mentre in altri dalla lentezza dei tempi di risoluzione da parte del team di sviluppatori. Nel primo caso, potrebbe essere opportuno intervenire sull'applicativo mentre nel secondo, una soluzione potrebbe riguardare l'efficientamento delle risorse di sviluppo.

Infine, un'ulteriore analisi da condurre riguarda la regressione dei difetti, un fenomeno frequente durante i cicli di No Regression Test (NRT). Spesso accade che alcuni bug vengano risolti in un ciclo di test per poi ripresentarsi nei successivi. Nei casi reali riscontrati in azienda, si è notato che problemi relativi alla corretta esecuzione dei bonifici e il rispettivo esito positivo si ripresentavano nonostante fossero stati correttamente risolti in precedenti cicli di NRT. Questo rende evidente la necessità di identificare e monitorare in modo accurato i difetti che tendono a riemergere, in modo da migliorare l'organizzazione dei test da eseguire e segnalare la casistica agli sviluppatori. Un metodo risolutivo che si propone con lo scopo di efficientare le esecuzioni, consiste nel focalizzare i test iniziali sui bug ripetuti. Per esempio, nel caso di difetti riguardanti i bonifici, si potrebbero eseguire questi test tra i primi, così da poter valutare immediatamente se il problema persiste. Se tali bug si ripresentano, essi devono essere sottoposti a test ricorrenti anche a seguito

di rilasci dell'applicativo che risolvono altre funzionalità, non direttamente collegate al bonifico. Ciò è dovuto al fatto che i difetti che si ripetono, possono dipendere da fattori esterni e indiretti, e la loro risoluzione potrebbe avvenire anche a seguito di modifiche su altre parti del sistema; in alternativa, si potrebbe trattare semplicemente di un'indisponibilità momentanea che dopo poco non si ripresenta. Questo approccio consente di risparmiare tempo, evitando la gestione formale dell'apertura di nuovi *defect* e la rispettiva gestione. Certamente, se il problema continua a persistere allora si ricorre all'apertura dei *defect*.

Per monitorare in modo efficace i difetti ricorrenti, si potrebbe considerare l'integrazione di una sezione dedicata nel sistema gestionale ALM (descritto nel Capitolo 4.2.2), che permetta di inserire come informazione aggiuntiva durante l'apertura di un *defect*, l'area dell'applicazione e l'operazione specifica interessata dal bug in oggetto. Questo consentirebbe di effettuare un conteggio dei difetti riferiti ad una specifica sezione e/o funzionalità, evidenziando eventuali criticità e facilitando ai tester l'organizzazione per l'esecuzione dei cicli di test. L'introduzione di un indice di regressione dei difetti potrebbe inoltre fornire un quadro più completo della stabilità dell'applicazione nel tempo e aiutare il cliente e gli sviluppatori a comprendere meglio le aree più a rischio.

4.3.2 Caso implementativo di scrittura di un codice

Il seguente capitolo mostrerà l'esecuzione pratica di scrittura di un caso di test reale.

Si desidera verificare che, nella schermata di dettaglio di un titolo, sia possibile accedere correttamente alla sezione "Dettaglio liquidità", la quale fornisce informazioni relative alla facilità di acquisto o vendita del titolo nel mercato in cui esso è negoziato. Lo *scart* del test è il seguente:

"# 0.0 - Effettuare la login con un BT che soddisfi i requisiti del test, prendere visione della documentazione ed accedere in schermata di compravendita

 # 1.0 - Verificare che si atterri correttamente sulla schermata di compravendita

 # 2.0 - Tappare su Textlink "Visualizza Dettagli liquidita"

 # 3.0 - Verificare che al tap si apra il bottomsheet di dettaglio liquidita.

Il codice relativo ai diversi step è il seguente:

- Step 0.0:
  And I login
  And I click MercatiView.btnMercati
  And I click MercatiView.azioniMercati

And I click DettaglioTitoloView.primoTitolo

And I wait 3 seconds

And I click DettaglioTitoloView.btnCompravendita

And MercatiView.confermaPresaVisione should be present

And I scroll "DOWN" until i find the element CompravenditaView.checkBox1

And I click CompravenditaView.checkBox1

And I click CompravenditaView.checkBox2

And I scroll "DOWN" for 2 times

And I click CompravenditaView.checkBox3

And I click CompravenditaView.checkBox4

And I click AllarmiView.btnConferma

La funzione “And I login” effettua l’accesso e autenticazione nell’app, utilizzando lo User ID indicato nel test.

La funzione “*And I click NomeClasse.nomeAttributo*” si occupa di cercare l’elemento indicato nella classe “NomeClasse” e univocamente identificato con il nome “nomeAttributo”. Nel caso in questione la classe “MercatiView” possiede un elemento univocamente identificato come “btnMercati”, il cui codice viene riportato di seguito:

```
@IOSBy(xpath = "//XCUIElementTypeButton[@name=\"Mercati\"])
@Element(xpath = "//android.widget.TextView[@content-desc=\"Mercati\"]")
 public static PageElement btnMercati;
```

La prima riga si riferisce all'identificativo utilizzato nel caso di sistema operativo iOS; di conseguenza, sarà quello l'elemento ricercato se il test in esecuzione riguarda il sistema operativo iOS. Al contrario, nel caso in cui il test sia eseguito su Android, l'elemento ricercato sarà identificato tramite l'xPath riportato nella seconda riga, identificata da “@Element”. Per il caso di esempio, l'elemento in iOS viene identificato come un bottone, mentre su Android viene riconosciuto come un testo.

La funzione “*And I wait 3 seconds*” stabilisce un'attesa di 3 secondi prima che la successiva istruzione venga eseguita; viene spesso utilizzata nei casi in cui alcune sezioni non si caricano immediatamente, evitando così che il codice successivo venga eseguito mentre è ancora presente la schermata di caricamento. Questo accorgimento riduce la probabilità di falsi negativi, ossia test che falliscono nonostante siano corretti. Naturalmente, l'input relativo al numero di secondi di attesa è modificabile in base alle esigenze.

Per quanto riguarda l’istruzione “*And NomeClasse.nomeAttributo should be present*” questa verifica che l’elemento fornito come input sia presente all’interno della pagina;

tuttavia, affinché tale funzione non scateni un'eccezione[51] e faccia fallire il test, è necessario che l'xPath dell'elemento fornito sia univoco all'interno della pagina. Nell'esempio in questione, se l'elemento identificato con "MercatiView.confermaPresaVisione", che corrisponde ad un semplice testo, è presente due o più volte all'interno della pagina, allora il test fallirà. Per identificare l'elemento in maniera univoca e non far fallire il test sarà necessario indicare nell'xPath l'indice a cui si riferisce, nel seguente modo:

"*(//android.widget.TextView[@content-desc=\"Mercati\"])[1]*"

tale dicitura sta ad indicare in maniera univoca il primo elemento di quel tipo che si trova scorrendo la pagina dall'alto verso il basso. Ovviamente se ci si svuole riferire all'n-esimo elemento si dovrà inserire il numero *n*.

Questo concetto risulta facilmente comprensibile osservando le righe di codice che fanno riferimento alle *checkbox*. La pagina in questione richiede la selezione di 4 *checkbox*, attraverso le quali si accettano le condizioni e si conferma di aver preso visione dei documenti forniti. Per questo motivo, le *checkbox* sono identificate con un numero cardinale, che indica a quale elemento fare riferimento. Pertanto, le istruzioni dalla riga 9 alla 13 servono a selezionare in maniera sequenziale, dalla prima alla quarta, le *checkbox* presenti all'interno della pagina.

La funzione "*And I scroll "DOWN" until i find the element NomeClasse.nomeAttributo*" è particolarmente utile per pagine dinamiche, il cui contenuto può variare. Questa funzione scorre la pagina fino a individuare l'elemento fornito come input. In alternativa, si potrebbe utilizzare la funzione meno efficiente "*And I scroll "DOWN" for n times*", che simula lo scorrimento della pagina per un numero di volte pari a *n* (dove *n* è un valore di input). Tuttavia, nel caso di una pagina dinamica, lo scorrimento per *n* volte potrebbe non essere sufficiente per raggiungere la sezione desiderata, in quanto potrebbero essere apparse nuove informazioni nelle sezioni precedenti, a causa del contenuto variabile della pagina. Per questo motivo, l'uso della prima funzione risulta più efficiente.

- Step 1.0:

  And CompravenditaView. paginaCompravendita should be present

  Tale istruzione effettua un controllo, ovvero verifica che, dopo aver premuto il pulsante "Conferma", si atterri nella sezione desiderata, cioè quella relativa alla compravendita.

[51] In informatica un'eccezione viene scatenata nel momento in cui il blocco di istruzioni in esecuzione si interrompe poiché, a causa di un errore, non può più proseguire. Per il caso in questione, se l'elemento definito in NomeClasse.nomeAttributo non viene ritrovato, il metodo scatena l'eccezione poiché non può proseguire.

Quest'ultima è univocamente identificata dalla presenza dell'elemento specificato in *CompravenditaView.paginaCompravendita*.

- Step 2.0:

  And I click CompravenditaView.visualizzaDettagliLiquidita

  Tappa sull'elemento con il testo "visualizza dettagli liquidità", così come richiede il flusso.

- Step 3.0:

  And MercatiView.dettaglioLiquidità should be present

  Infine, si verifica che il titolo relativo alla nuova sezione sia corretto e, pertanto, che si venga indirizzati alla pagina corretta. Non vengono effettuati ulteriori controlli sulla correttezza delle informazioni trasmesse nella pagina di "Dettaglio liquidità" poiché si tratta di una sezione testata nelle *wave* precedenti.

Le funzionalità sopra riportate descrivono l'interazione automatica con l'applicazione tramite *click* oppure azioni di scorrimento. Tuttavia, è possibile realizzare anche delle funzioni che permettono di eseguire del codice con condizioni di *if-else* o cicli ripetuti[52].

Di seguito verrà mostrato un altro frammento di codice, sviluppato per verificare il corretto flusso di modifica del pin di accesso all'applicazione. Nel caso in cui il flusso venga completato con successo, il pin dell'utenza utilizzata per il test verrà modificato. Tuttavia, poiché le utenze vengono condivise da tester e sviluppatori all'interno di Concept Reply, eventuali terzi non a conoscenza della modifica potrebbero incontrare difficoltà di accesso a causa del PIN errato. Risulterebbe anche complesso tracciare tutti i PIN modificati e identificare quello valido in quel momento. Per tale ragione, è stato sviluppato il seguente codice che, oltre a modificare il PIN, prevede anche il suo ripristino al valore originario, comune a tutte le utenze.

And I login

And I change PIN with value "13577"

And I click LoginView.entra

And I wait 10 seconds

And I click LoginView.pin1

And I click LoginView.pin3

[52] In informatica, i costrutti di *if-else* rappresentano delle strutture condizionali in cui, qualora venga soddisfatta la condizione definita all'interno del blocco *if*, viene eseguito il codice corrispondente; in caso contrario, viene eseguito il codice presente nel blocco *else*. I cicli, invece, consentono di ripetere l'esecuzione di un blocco di codice o per un numero fisso di iterazioni, oppure fino a quando non viene soddisfatta una determinata condizione.

And I click LoginView.pin5

And I click LoginView.pin7

And I click LoginView.pin7

And I click AltroView.continua

And I wait 10 seconds

And I change PIN with value "13579"

La funzione che consente di modificare il pin è la seguente:

```
@And("I change PIN with value {string}")
   public void iChangePINWithValue(String value) throws Exception {
    Functions.clickIfPresent(MainView.altrobt);
    Functions.clickIfPresent(ProfiloView.sicurezza);
    Functions.clickIfPresent(ConnessioneSicuraView.tutteLeImpostazioni);
    MobileSteps.scrollForTimes("DOWN", 1);
    Functions.clickIfPresent(ProfiloView.modificaPin);
    if(value.equals("13579")){
     LoginView.insertOTS("13577");
    } else{
     insertPin();            }
    fillInputBox(value, "ProfiloView", "nuovoPIN");
    if(CommonPage.getDriverType().isIOS()){
     Functions.clickIfPresent(AltroView.FINE);
    } else{
     MobileSteps.closeKeyboard();     }
    fillInputBox(value, "ProfiloView", "nuovoPIN");
    if(CommonPage.getDriverType().isIOS()){
     Functions.clickIfPresent(AltroView.FINE);
    } else{
     MobileSteps.closeKeyboard();
```

```
    }
    Functions.clickIfPresent(AltroView.continua);
    GeneralSteps.waitUntilVisible("AltroView", "attendere");
    assertTrue(LoginView.pinModificatoCorrettamente.isPresent(),"esito negativo alla modifica del pin");
    Functions.clickIfPresent(chiudi);
  }
}
```

La prima parte della funzione indica il flusso per accedere alla sezione di modifica del pin, come descritto in Figura 4.11.

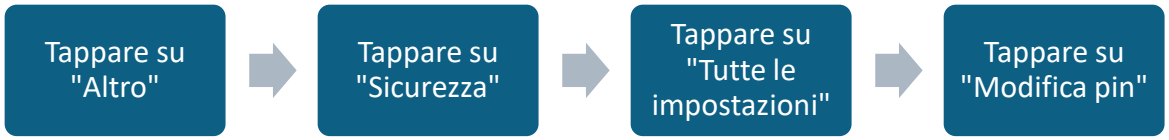


Figura 4.11 - Flusso per la modifica del PIN.

L'interazione con lo schermo avviene tramite *click* oppure *scroll*.

Supponiamo che il codice standard utilizzato per tutte le utenze sia 13579. Se il parametro di input della funzione, che indica il nuovo valore desiderato del pin, risulta diverso da tale sequenza, significa che il pin attualmente in uso è 13579. Diversamente, se il parametro passato corrisponde al codice standard, ci troviamo nella parte del flusso in cui si desidera ripristinare il valore originale e il valore del pin da inserire per accedere alla sezione sarà quello momentaneo. Poiché per accedere alla sezione di modifica è necessario inserire il pin valido al momento dell'accesso, il controllo descritto consente di determinare qual è il codice corretto in quel momento. L'inserimento viene eseguito tramite la funzione insertPin(), se si fa riferimento al valore standard; invece, per inserire un valore diverso sarà utilizzato LoginView.insertOTS("13577").

Tale controllo è fondamentale poiché consente di modificare il pin o reimpostarlo al valore standard, basandosi esclusivamente sull'analisi dell'input fornito. Ma nel caso in cui il pin in corso coincida con quello standard ma venga comunque eseguito il seguente comando:

And I change PIN with value "13579"

la procedura non avrà esito positivo e verrà visualizzato un messaggio di errore, in quanto il nuovo PIN deve necessariamente differire da quello già in uso.

Successivamente, il flusso prevede l'inserimento del nuovo codice per due volte, al fine di completare l'operazione. Viene inoltre effettuato un controllo relativo al sistema operativo: se il sistema è iOS, sarà necessario tappare su "Fine" per chiudere la tastiera; in caso contrario, per Android, la tastiera viene chiusa mediante l'esecuzione della funzione closeKeyboard().

Una volta completata correttamente l'operazione, le logiche di business prevedono l'atterraggio sulla schermata di *Pre Login*, ossia la schermata visualizzata prima dell'inserimento del codice di accesso all'applicazione. Per questo motivo, il codice continua con il click sul pulsante "Entra" e l'inserimento del nuovo codice; da qui in poi, verrà eseguito il codice che consente la modifica del PIN, riportandolo dal nuovo valore a quello standard.

Il codice appena descritto rappresenta una casistica molto comune, in cui si verifica la correttezza del flusso richiesto, mantenendo allo stesso tempo le caratteristiche originarie dell'utenza, per preservare la durabilità delle utenze e i dati con cui effettuare i test.

4.4 Valutazione economica

L'implementazione di un progetto tramite test automatici o manuali dipende da molteplici fattori, tra cui la disponibilità di risorse o strumenti e la sostenibilità economica del progetto stesso. Di seguito verrà eseguita una valutazione economica comparativa tra l'esecuzione di un progetto tramite Test Automation e il suo svolgimento tramite test manuali: saranno analizzati i costi e i ricavi relativi a ciascuna delle due opzioni, per determinarne l'impatto economico. Come riportato più volte in precedenza, il principale vantaggio dell'implementazione di test automatizzati consiste nell'esecuzione ciclica di test per verificare la non regressione del software, ottenendo un notevole vantaggio in termini di tempo di esecuzione. L'automazione risulta vantaggiosa anche nel caso in cui i test debbano essere eseguiti su diversi dispositivi, generalmente per testare la coerenza e la correttezza del *front-end* al variare dei dispositivi. Pertanto, se i test sono finalizzati unicamente a verificare la conformità dell'applicazione in quel momento, è conveniente procedere manualmente; al contrario, se i test dovranno essere replicati in futuro per scopi di non regressione o consistenza nei vari dispositivi, è consigliabile investire in un progetto di Test Automation.

Di conseguenza, per condurre la valutazione in oggetto si considera l'esecuzione di System Test su dispositivo mobile, da rieseguire successivamente come No Regression Test (NRT) con cadenza periodica. Per semplificare l'analisi, si assume che il controllo qualità venga eseguito dall'ente bancario stesso, al fine di considerare unicamente gli esborsi sostenuti dall'azienda per l'esecuzione dei test e i rispettivi benefici diretti e indiretti, senza l'intervento di società terze.

Di seguito verranno analizzati i costi necessari per l'implementazione di entrambe le soluzioni. In prima battuta analizziamo i costi diretti come segue:

- Test manuali:
    - Costi del personale, calcolabili per semplicità come la paga oraria moltiplicata per il numero di ore necessarie.
    - Risoluzione dei problemi post produzione, non emersi durante la fase di *testing*: essi richiederanno un costo di intervento maggiore rispetto alla loro risoluzione in fase di System Test.
- Test automatici:
    - Costi del personale, calcolabili per semplicità come la paga oraria moltiplicata per il numero di ore necessarie.
    - Risoluzione dei problemi post produzione, non emersi durante la fase di *testing*: essi richiederanno un costo di intervento maggiore rispetto alla loro risoluzione in fase di System Test.

I costi indiretti individuati nei due scenari, invece, sono i seguenti:

- Test manuali:
    - Costi dei dispositivi fisici; nel caso in questione sono i cellulari su cui testare.
- Test automatici:
    - Costi dei dispositivi fisici; nel caso in questione sono i cellulari su cui testare.
    - Costi dei tools per l'implementazione ed esecuzione dei test.
    - Manutenzione dei test.
    - Formazione del personale.

Di seguito saranno analizzate le diverse voci di ricavo:

- Test manuali:
    - Maggiore qualità dell'applicazione e, pertanto, incremento della quota di mercato.
    - Flessibilità nella valutazione del progetto, con la possibilità di rilevare difetti evidenti anche al di fuori degli obiettivi specifici del test eseguito poiché si ha la supervisione umana.
- Test automatici:
    - Maggiore qualità dell'applicazione e, pertanto, incremento della quota di mercato.
    - Riduzione del Time To Market (TTM), grazie al risparmio in termini di tempo che si ottiene testando in modo automatico e parallelamente su diversi dispositivi.

In Tabella 4.4 vengono riassunte le voci di costi e ricavi per avere una visione più chiara e immediata: le stime delle voci di costi e ricavi riportate sono state elaborate sulla base dell'esperienza maturata in azienda riguardo progetti del settore di riferimento, nonché quello bancario per il caso in questione.

<table>
<tr><td colspan="2"></td><td>Test manuali</td><td>Test automatici</td></tr>
<tr><td rowspan="5">Costi</td><td rowspan="2">Costi diretti</td><td>Personale</td><td>Personale</td></tr>
<tr><td>Risoluzione dei problemi post produzione</td><td>Risoluzione dei problemi post produzione</td></tr>
<tr><td rowspan="3">Costi indiretti</td><td rowspan="3">Dispositivi fisici</td><td>Dispositivi fisici</td></tr>
<tr><td>Tools</td></tr>
<tr><td>Manutenzione</td></tr>
<tr><td colspan="2" rowspan="2">Ricavi</td><td>Formazione personale</td><td></td></tr>
<tr><td>Maggiore qualità</td><td>Maggiore qualità</td></tr>
<tr><td></td><td>Flessibilità di valutazione</td><td>Riduzione TTM</td></tr>
</table>

Tabella 4.4 - Costi e ricavi per TA e Test Manuali.

Per quanto riguarda i costi del personale, è evidente che essi siano presenti sia per progetti di *testing* automatico che manuale: la differenza principale tra le due modalità riguarda il tempo necessario per l'esecuzione dei test, che influisce in modo diretto sul costo totale del personale poiché determina le ore-uomo da retribuire. Per semplicità di analisi, si suppone la presenza di un solo tester; come riportato nel Capitolo 3.2.1, la stima del tempo necessario per la scrittura e esecuzione di un test automatico è di 12 minuti. Pertanto, se si ipotizza l'esecuzione di α System Test (con $\alpha > 0$), il tempo complessivamente impiegato da un tester è pari a:

$$12\ minuti * \alpha\ test = (12\alpha)\ minuti$$

Invece, riguardo l'esecuzione di NRT, le stime riportate nel Capitolo 3.2.2, prevedono lo svolgimento da parte di un tester di 14 test in un'ora ovvero $\frac{60\ minuti}{14\ test} \cong 4{,}3\ minuti\ per\ ciascun\ test$.

Siano α i test da eseguire e β i cicli di NRT da realizzare, è immediato ricavare il tempo totale di esecuzione come segue:

$$(12\alpha + 4{,}3\alpha\beta)\ minuti$$

dove 12α sono i minuti necessari per la realizzazione del codice e la prima esecuzione, mentre $4{,}3\alpha\beta$ è il tempo necessario per svolgere gli α test β volte.

Proseguendo l'analisi per il caso di esecuzione di test manuali, si osserva un risparmio in termini di tempo per i System Test, poiché non è necessaria la scrittura del codice, ma si verifica un notevole dispendio di tempo per l'esecuzione degli NRT. Empiricamente, stimando una media di 6 minuti per l'esecuzione di ciascun test manuale, il tempo totale impiegato da un singolo tester per lo svolgimento di α test per β cicli di No Regression Test e uno di System Test, viene calcolato come segue:

$$6\,minuti * \alpha\ test * (1 + \beta\ cicli\ di\ NRT) = 6\alpha(1 + \beta\ )\ minuti$$

Supponendo che il valore medio della paga oraria di un tester sia il medesimo, sia che esso esegua test manuali che automatici, allora il costo del personale per i due progetti in esame dipende unicamente dal tempo impiegato, se calcolato come nel caso in questione in modo semplicistico come il prodotto fra paga oraria e monte ore. Effettuando un confronto fra il tempo impiegato per i test automatici e manuali si ottiene:

$$Tempo\ Test\ Manuali > Tempo\ Test\ Automatici$$

$$6\alpha(1 + \beta\ ) > (12\alpha + 4{,}3\alpha\beta)$$

$$6\alpha + 6\alpha\beta > 12\alpha + 4{,}3\alpha\beta$$

$$-6\alpha + 1{,}7\alpha\beta > 0$$

$$\alpha(-6 + 1{,}7\beta) > 0$$

La disequazione è soddisfatta e, pertanto, il tempo e il costo del personale per i test manuali è maggiore rispetto all'esecuzione automatica se:

$$\alpha > 0$$

$$\beta > \frac{6}{1{,}7} = 3{,}5$$

Ovvero se il numero di test da eseguire è maggiore di zero, ma tale condizione viene sempre soddisfatta, e se il numero di cicli di NRT è superiore o uguale a 4. Pertanto, limitatamente alle stime ricavate durante l'esperienza in azienda, è possibile concludere che eseguire un progetto automatizzando i test può essere vantaggioso se sono previsti almeno 4 cicli di NRT. Inoltre, per come è stata impostata l'equazione, si suppone che l'esecuzione sia per i test manuali che per gli automatici sia sequenziale, ovvero che gli α test durante i cicli di NRT siano svolti l'uno dopo l'altro. In un caso reale, è evidente che se si dispone di più dispositivi, è possibile lanciare contemporaneamente più test automatizzati, ciascuno in un dispositivo differente, ottenendo un notevole vantaggio. Tale fattore però non è considerato nel calcolo effettuato.

Se andassimo a considerare anche il numero di dispositivi γ su cui eseguire i test, ad esempio dispositivo con sistema operativo iOS e Android, esso risulterebbe un fattore moltiplicativo nel caso di test manuali, poiché gli α test dovranno essere eseguiti γ volte in modo sequenziale. Al contrario, per gli automatici l'esecuzione sui diversi dispositivi potrebbe assumersi parallela, senza pertanto modificare i tempi di esecuzione, direttamente collegati al costo del personale.

Un altro costo da valutare nell'esecuzione di un progetto sia nel caso di test manuali che di test automatizzati, riguarda la risoluzione dei *defect* emersi post-produzione. Si tratta di un rischio che non può essere completamente eliminato, nonostante l'attenzione e l'accuratezza dei test durante la fase di pre-produzione. Sulla base dei progetti seguiti in azienda e dei numerosi studi citati nella presente trattazione, è evidente che il numero di difetti individuati tramite test manuali è inferiore rispetto a quelli individuati con i test automatizzati. Di conseguenza, software testati manualmente, con elevata probabilità, dovranno subire costi maggiori in fase di post-produzione per la risoluzione dei difetti non rilevati durante la fase di pre-produzione. È possibile stimare tale gap con un'analisi empirica, analizzando il numero di *defect* rilevati nel caso di esecuzione di test manuali e automatici per uno stesso progetto. Tuttavia, per ottenere dei risultati robusti e statisticamente significativi è necessario analizzare un numero consistente di progetti e stimare il valor medio del differenziale di *defect* riscontrati con le due diverse metodologie. Oltre ad una mera valutazione del numero di difetti, bisognerebbe anche analizzare l'impatto che essi hanno: di norma, con il *testing* si rilevano i problemi funzionali dell'applicazione e invalidanti il rilascio in produzione. Quelli che emergono in fase di post produzione riguardano aspetti secondari dell'applicativo che ugualmente devono essere risolti e rilevarli in fase di pre-produzione consentirebbe di abbattere i costi, oltre che ad un guadagno reputazionale nel fornire al cliente finale un prodotto funzionante e con standard qualitativamente elevati.

Per quanto riguarda i costi dei dispositivi fisici, cioè i cellulari su cui effettuare i test per il caso in questione, l'investimento è il medesimo sia per l'esecuzione manuale che automatica. Per tale motivo, questa voce di costo non è da tenere in considerazione nella valutazione economica dei progetti poiché si tratta di una componente aggiuntiva che influisce in egual misura su entrambe le opzioni.

Diversamente, nel caso dei test automatizzati si ha un costo aggiuntivo relativo a tools e framework necessari per l'implementazione e l'esecuzione dei test, in parte descritti nel Capitolo 2.2. Per utilizzare tali strumenti l'azienda acquista annualmente le licenze; tuttavia, tale costo è da ammortizzare su tutti i progetti di Test Automation poiché si tratta di strumenti utilizzati trasversalmente dall'azienda, per i diversi ambiti di applicazione di tecniche di Test Automation. Per

il caso in questione, assumendo il punto di vista dell'azienda cliente che internalizza l'esecuzione dei test qualitativi, è corretto supporre che i tools e framework per l'automazione vengano utilizzati per le applicazioni mobile e web dell'*home banking*, per i gestionali interni utilizzati dagli operatori stessi oppure per eventuali applicazioni di trading di cui l'ente bancario dispone. Di conseguenza, si tratta sicuramente di un costo che l'azienda sostiene ma se ammortizzato per tutte le applicazioni diventa un valore quasi trascurabile.

Ai costi appena descritti, bisogna aggiungere i costi di manutenzione degli *script*: è necessario impiegare tempo e risorse per aggiornare il codice dei test, qualora cambiasse il flusso di esecuzione oppure gli xPath identificativi degli oggetti. Tuttavia, come descritto nel Capitolo 1.4, con l'integrazione di strumenti di intelligenza artificiale è possibile sviluppare un sistema di supporto che, in caso di cambiamenti, aggiorni automaticamente gli script. Si tratta pertanto di un problema momentaneo, che con l'evoluzione tecnologica si potrà superare.

Infine, l'ultima voce di costo analizzata riguarda la formazione del personale per l'esecuzione di test automatizzati. È necessario disporre di risorse adeguatamente formate dal punto di vista tecnico per la corretta stesura del codice, affinché esso sia consistente e duraturo nel tempo. Tale voce di costo potrebbe essere vista sia come l'incremento di paga da sottoporre ai tester automatici con adeguata formazione oppure come il costo di corsi di formazione da offrire per elevare le competenze del personale. Si tratta di un costo che l'azienda deve sostenere qualora decida di automatizzare, anche solo in parte, i test da eseguire; tuttavia, tale costo è da considerarsi in relazione ai benefici che in futuro si otterranno quali l'attendibilità dei test, la loro durabilità e, pertanto, il risparmio in termini di manutenzione futura.

Analizzando i ricavi, la voce che emerge sia nel caso di test automatizzati che manuali riguarda l'incremento della qualità del software realizzato, poiché è un aspetto intrinseco al concetto stesso di *testing*, inteso come controllo qualità. Tuttavia, qualora l'implementazione di tecniche di Test Automation per un progetto sia vantaggioso sia in termini di tempo che di esecuzioni, è evidente che la qualità raggiunta con l'automazione risulta superiore al caso manuale. Ciò dipende dagli aspetti discussi fin ora, quali la riduzione della probabilità di errore umano, la capacità di rilevare un numero maggiore di *defect* oppure la riduzione del *Time To Market* che garantisce una maggiore competitività nel mercato. È sicuramente difficile stimare in termini monetari il guadagno aggiuntivo derivante dall'incremento di qualità dovuto all'automazione dei test, tuttavia è considerabile come una voce di ricavo aggiuntiva da tenere in considerazione.

Per quanto riguarda la flessibilità dei test manuali, essa consente di individuare difetti non rilevabili tramite l'esecuzione di uno specifico test. Si riporta un esempio per comprendere meglio questo

concetto: se un test richiede di verificare la corretta esecuzione di un bonifico, l'esito derivante dall'esecuzione automatica sarà limitato alla riuscita oppure al fallimento del bonifico. Difatti, il codice sarà realizzato per compilare automaticamente i dati per l'esecuzione bonifico, proseguire con il flusso e, infine, analizzare la pagina di esito: se negativo, il test fallirà mentre se positivo il test risulterà in *passed*. Al contrario, se il test viene eseguito manualmente si avrà una maggiore flessibilità nell'individuazione degli errori: ad esempio, è possibile che il tester, al di là dell'esito del bonifico, visivamente si renda conto di un problema grafico quali un titolo errato nell'intestazione della pagina oppure la mancanza di qualche pulsante. Pertanto, c'è una maggiore probabilità di individuare malfunzionamenti o mancanze dell'applicazione, seppur quella verifica non è espressamente obiettivo del test.

Tuttavia, con l'evoluzione delle tecniche di Test Automation, tale problematica è in corso di risoluzione: difatti, come descritto nel Capitolo 1.4, si stanno diffondendo tecniche di Visual Locators che identificano gli elementi dell'interfaccia grafica attraverso un'analisi visiva. Pertanto, di volta in volta verrà analizzata l'intera schermata e non solamente l'elemento individuato tramite xPath. Dunque, quello che risulta essere un vantaggio aggiuntivo apportato dai test manuali, diventa irrilevante nel caso di implementazioni di tecniche di Test Automation all'avanguardia.

Infine, l'ultima voce di ricavi da considerare nei progetti di Test Automation riguarda la riduzione del *Time To Market* (TTM). L'accelerazione dei tempi e la possibilità di parallelizzare le esecuzioni su più dispositivi, consente di ridurre il tempo necessario ai rilasci in produzione, generando per l'azienda un vantaggio competitivo, e intervenire tempestivamente sui problemi riscontrati, rispondendo rapidamente alle esigenze del cliente.

In conclusione, valutate le voci di costi e di ricavi che riguardano l'implementazione di un progetto tramite test automatici o manuali e, a seguito dell'esperienza di tirocinio compiuta, è possibile effettuare delle considerazioni.

Innanzitutto, è opportuno valutare l'impatto economico determinato dall'implementazione del Test Automation in un progetto o, meglio, all'interno dell'azienda; sarebbe poco sensato effettuare gli investimenti fin ora descritti per eseguire un solo progetto tramite automazione dei test. Pertanto, l'azienda deve disporre di una serie di progetti validi da eseguire con test automatici. Qualora queste circostanze si verifichino, un ulteriore aspetto da valutare riguarda la formazione del personale: acquisire da zero le competenze per la corretta configurazione e implementazione dei tools da utilizzare per l'automazione può risultare complicato; è necessario allora un team di esperti in grado di formare il resto del personale. Seppur apparentemente banale, si tratta di un aspetto rilevante da considerare poiché redigere test automatizzati poco affidabili o duraturi,

comporta un dispendio di tempo e risorse per la risoluzione dei problemi ad ogni loro esecuzione e, pertanto, si rischierebbe di compromettere i benefici dell'automazione.

Stabilito allora che per l'azienda sia conveniente investire in tecniche di Test Automation, è necessario valutare se, dato un nuovo progetto, esso si adatti meglio ad un'esecuzione automatica oppure manuale. Se i test dovranno essere eseguiti in più cicli di No Regression Test oppure su più dispositivi, risulterà sicuramente conveniente implementare i test come automatizzati. Al contrario, se si tratta di test mirati a verificare la qualità dell'applicazione in quella fase di pre-produzione, senza necessità di procedere con futuri cicli di non regressione oppure *Cross-Platform Compatibility*, risulterà più immediato procedere tramite test manuali, risparmiando principalmente il tempo di stesura del codice e l'impiego di risorse specializzate.

Dovrà pertanto essere valutato il giusto trade-off fra l'investimento iniziale da fronteggiare e il guadagno che ne deriva. Le spese iniziali possono essere riassunte come segue:

- acquisto delle licenze per i tools e i framework di automazione;
- adeguata formazione del personale;
- manutenzione del codice.

Tali costi dovranno essere interamente coperti dai vantaggi che ne derivano, ovvero:

- risparmio in termini di tempo di esecuzione dei test;
- maggiore individuazione dei difetti dell'applicazione e, pertanto, incremento della qualità offerta;
- riduzione del Time To Market.

5. Conclusioni

In definitiva, è possibile affermare che il lavoro di tesi svolto ha come obiettivo principale quello di analizzare i processi coinvolti in un progetto eseguito tramite automazione dei test per valutare l'aspetto qualitativo di un'applicazione bancaria.

In un primo momento sono state esposte metodologie, vantaggi e svantaggi di tale strumento riportando diversi studi presenti in letteratura. Successivamente, sono stati analizzati dei casi pratici, eseguiti durante l'esperienza di tirocinio presso Concept Quality Reply, i cui risultati confermano quanto riportato dagli studi teorici. In particolare, si evidenzia l'efficacia del Test Automation sotto determinate condizioni quali la presenza di un team adeguatamente preparato

dal punto di vista tecnico, la disponibilità di risorse economiche da investire e, infine, la necessità di esecuzione dei test con cadenze periodiche.

Confrontando l'esecuzione manuale con quella automatica, si evidenzia come nel secondo caso sia presente una significativa riduzione dei tempi di esecuzione accompagnata da una maggiore capacità di individuazione dei difetti, a seguito di una minore probabilità di errore umano da parte del tester. Tuttavia, l'applicazione del Test Automation non è efficiente nel caso di test eseguiti una tantum, poiché i costi di investimento iniziali non sarebbero compensati dai benefici in termini di tempo di esecuzione e riduzione del *Time To Market*.

Con l'ausilio dei dati raccolti in azienda, si dimostra che l'esecuzione di progetti di *testing* tramite automazione risulta conveniente rispetto all'esecuzione manuale nel caso in cui i cicli di non regressione siano eseguiti per un numero maggiore di 4 volte. Tale stima risulta coerente con quanto riportato in diversi studi presenti in letteratura, con un errore minimo dovuto al fatto che i dati utilizzati per la stima sono relativi ad un singolo progetto.

Oltre all'aspetto tecnico, è stato analizzato anche quello gestionale: per quanto riguarda i sistemi sono state suggerite alcune implementazioni aggiuntive, sulla base delle criticità riscontrate durante il percorso. Ad esempio, si è ritenuto utile proporre una sezione relativa alle attività eseguite da ciascun tester, tramite un Gantt che evidenzia i progetti seguiti da ognuno e alcuni grafici che esprimono la percentuale di test svolti relativamente ai diversi progetti. Tale valutazione è fondamentale per comprendere l'efficienza del tester e allocarlo opportunamente sui diversi progetti. Inoltre, si suggerisce l'implementazione di un Gantt complessivo che permetta di valutare la situazione del team, allineando tutti i soggetti.

È stato analizzato anche l'aspetto prettamente organizzativo e di assegnazione dei progetti, suggerendo pertanto un'allocazione fissa al fine di conservare il *know-how* sviluppato.

Inoltre, è stata descritta l'evoluzione di un progetto, a partire dai primi contatti con i clienti, per proseguire con l'ideazione dei test, la loro esecuzione con i rispettivi risultati e, infine, la valutazione del progetto stesso. Dall'analisi è emerso che, coerentemente con la metodologia *Agile*, le richieste dei clienti sono in continua evoluzione, pertanto è necessario adattarsi, effettuando le opportune modifiche. Inoltre, si evidenzia come l'avanzamento del progetto non dipenda direttamente dal lavoro effettuato dai tester ma anche dalle richieste del cliente e da team terzi che cooperano al medesimo progetto, sotto aspetti differenti.

Pertanto, si suggeriscono degli indicatori di valutazione del progetto al fine di individuare i problemi riscontrati e implementare soluzioni valide per ottimizzare i processi; essi monitorano la stabilità

dell’applicazione durante la fase di test, il tempo di risoluzione dei difetti e la classificazione dei bug maggiormente ricorrenti.

Infine, sono stati riportati i codici realizzati relativamente ad alcuni test, spiegando le funzionalità di ciascuna riga e le criticità associate. Ad esempio, al fine di redigere codici duraturi nel tempo, è necessario prevedere la loro manutenzione e aggiornamento.

Bibliografia

Abualhaija, S., Ceci, M., & Sannier, N. (2024). Toward Automated Change Impact Analysis of Financial Regulations. *Workshop on Software Engineering Challenges in Financial Firms*.

Alegroth, E., Feldt, R., & Olsson, H. (2013). Transitioning Manual System Test Suites to Automated Testing: An Industrial Case Study. *IEEE Sixth International Conference on Software Testing, Verification and Validation.*

Appium Documentation. (2024). Tratto da https://appium.io/docs/en/2.11/

Battina, D. S. (2019). Artificial Intelligence in Software Test Automation: A Systematic Literature Review.

Beck, K. (2001). *Manifesto for Agile Software Development.* Agile Alliance.

Berner, S., Weber, R., & Keller, R. (2005). Observations and lessons learned from automated testing. *Proceedings of the 27th international conference on Software engineering*.

Capgemini, Sogeti, & Micro Focus. (2020-2021). World Quality Report.

Capgemini, Sogeti, & OpenText. (2023-2024). *World Quality Report.*

Certificazione ISO/IEC 27001. (2013).

Collins, E., Dias-Neto, A., & de Luc, V. (2012). Strategies for Agile Software Testing Automation: An Industrial Experience.

Crispin., L., & Gregory, J. (2009). *Agile Testing: A Practical Guide for Testers and Agile Teams.* Addison-Wesley.

Cucumber Documentation - Behaviour-Driven Development. (2024). Tratto da https://cucumber.io/docs/bdd/

Cucumber Documentation. (2024). Tratto da https://cucumber.io/docs/cucumber/

Filippetti, A. (2017). Agile History: storia del movimento agile.

IntelliJ Documentation. (2024). Tratto da https://www.jetbrains.com/help/idea/discover-intellij-idea.html

IONOS. (2023). *Modello a spirale: il modello per la gestione dei rischi nello sviluppo di software*. Tratto da https://www.ionos.it/startupguide/produttivita/modello-a-spirale/#c556774

ISO/IEC/IEEE 29119-1. (seconda edizione 2022-01).

Jordan, C., Maurer, F., Lowenberg, S., & Provost, J. (2019). Framework for Flexible, Adaptive Support of Test. *IEEE Robotics and Automation Letters*.

Jose, B. (2021). *Test Automation. A manager's guide.* BCS Learning & Development Limited.

Marick, B. (2006). *Everyday Scripting with Ruby: For Teams, Testers, and You.* Pragmatic Bookshelf.

Mohammad, S. M. (2015). Automation Testing in Information Technology. *International Journal of Creative Research Thoughts (IJCRT)*.

Muzzì, C., & Caramellino, M. (2007). Programmazione Configurativa.

Pagano, A. (2020). *Modello a Cascata*. Tratto da Programming Academy: https://programmingacademy.it/2020/04/modello-a-cascata/

Pandey, S. (2023). AI Automation and Testing.

Payment Card Industry Data Security Standard: Requirements and Testing Procedures. (2024). PCI Security Standards Council.

Polo, M., Reales, P., Piattini, M., & Ebert, C. (2013). Test Automation. *IEEE Software*.

Regolamento (UE) 2016/679 del parlamento europeo e del consiglio. (2016).

Reply. (2023). *Bilancio Annuale 2023*. Tratto da Reply: https://www.reply.com/it/newsroom/financial-news

Sommerville, I. (9th edition - 2011). *Software Engineering.* Pearson.

Springer, C. (2014). *Continuous Software Engineering.*

Sucandra, R., & Rinnova, D. (2024). The Influence of Mobile Banking Service ualityon Customer Satisfaction at Bank Rakyat Indonesia Tanjung Karang Branch Office.

Wiklund, K., Eldh, S., Sundmark, D., & Lundqvist, K. (2017). Impediments for software test automation: A systematic literature review. *Software Testing, Verification and Reliability*.