UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

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

Tesi di Laurea Specialistica in Ingegneria del Software

Confronto tra tecniche automatiche ed assistite per la generazione di test case per applicazioni Android

Anno Accademico 2014/2015

relatore

Ch.mo Prof. Porfirio Tramontana

correlatore
Ing. Nicola Amatucci

candidato
Alessia Anniciello
matr. 885-493

Indice

Introduzione

In ogni modello di sviluppo strutturato, sia esso waterfall o agile, un'ampia fetta dell'effort programmato è destinata alla realizzazione ed all'esecuzione di test (unit, system, acceptance). Nel panorama mobile questa fase assume rilievo massimo e diventa imprescindibile per la realizzazione di un prodotto solido e di successo. Sia esso manuale o automatico, un testing soddisfacente richiede risorse e pertanto ciascuna realtà aziendale costruisce il proprio processo accettando dei trade off tra costi e benefici.

Il presente lavoro di tesi si propone di fornire una panoramica completa sul mondo del testing mobile evidenziandone le principali criticità, presentando lo stato dell'arte in termini di framework, tool e strategie ed, infine, proponendo un approccio integrato tra tecniche manuali ed automatiche per ottimizzare tempi e risorse.

Nel primo capitolo saranno elencate e approfondite le principali sfide che lo sviluppo di applicazioni mobile presenta. Contemporaneamente saranno illustrate e confrontate strategie e modalità differenti di realizzazione ed esecuzione dei test offrendo un quadro preciso delle scelte che devono essere effettuate in un ciclo di sviluppo realistico

Capitolo 1: Mobile Testing Challenge

La capillare diffusione di smartphone e device portatili di varia natura ha fatto sì che un gran numero di software house migrassero la loro produzione dai software desktop a quelli mobile. L'esportazione dei processi di sviluppo tradizionali non si è dimostrata abbastanza flessibile da reggere i velocissimi cambiamenti del settore e spesso le aziende si sono trovate a fronteggiare una complessa fase di ristrutturazione dei processi e di training del personale per adeguarsi alle nuove necessità.

Allo stesso tempo un esercito di programmatori autodidatti e di professionisti in cerca di fortuna hanno anch'essi intrapreso la strada del mobile speranzosi di poter accedere, con la giusta dose di intuizione e fortuna, agli enormi profitti che questo mercato sta garantendo.

Anche in questo caso, un approccio spesso destrutturato e privo delle più elementari best practices, ha contribuito ad alimentare il vasto panorama di applicazioni più o meno bacate e di poco successo che affollano i vari market.

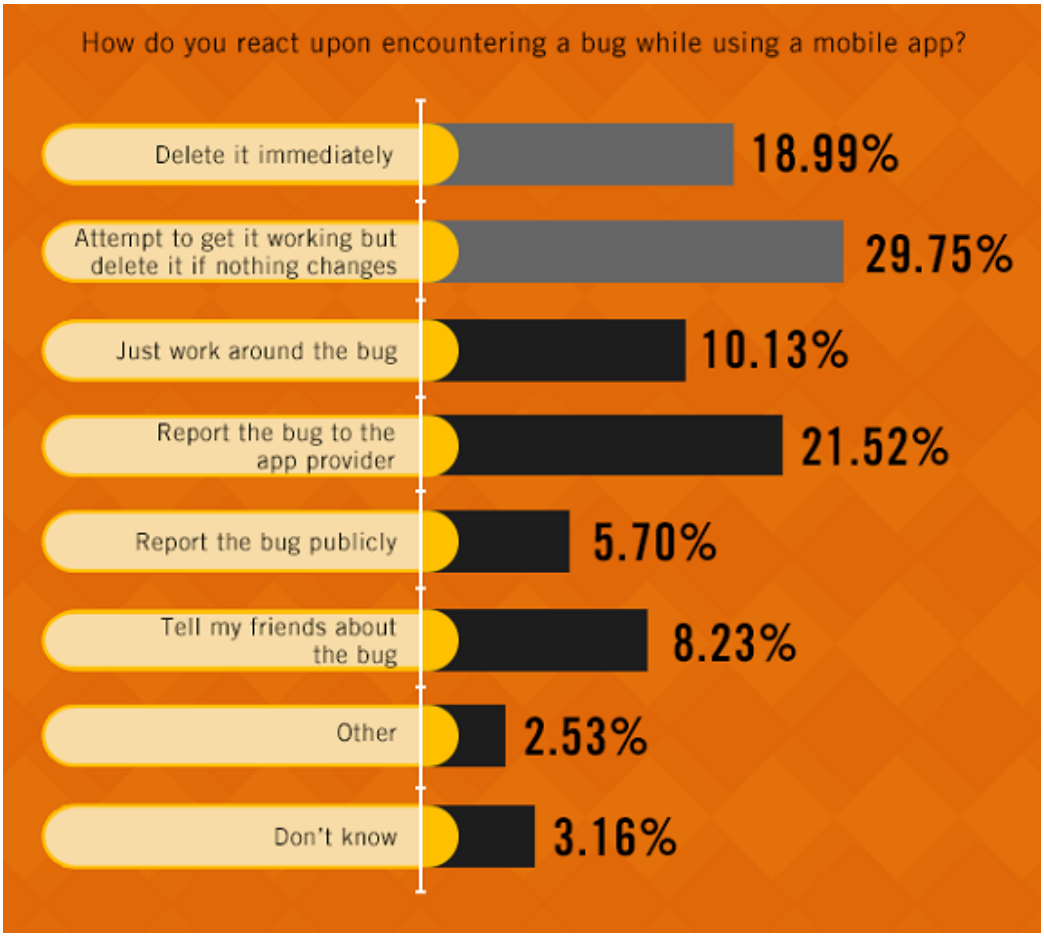
Si è dunque presto fatta sentire l'esigenza di individuare e formalizzare una serie di strategie e framework per lo sviluppo di applicazioni mobile, arrivando alla conclusione che oltre il 50% dell'effort dovrebbe essere destinato alla fase di testing per produrre software di valore e di successo.

Poiché progettare, realizzare e manutenere test suite oltre ad essere un'attività estremamente dispendiosa è sempre stata considerata tediosa e spesso trascurata dagli sviluppatori, specialmente nei contesti aziendali più piccoli, i tools per l'automazione e la semplificazione della stesura/esecuzione dei test hanno riscosso sempre maggior successo.

1.1 Testing automation: un vantaggio competitivo

« *A bug can cost you a customer* » - In questa breve frase è sintetizzata l'importanza di rilasciare software robusto. Lo studio *State of mobile* effettuato nel 2014 dalla SmartBear [1] ha evidenziato che circa il 50% degli utenti tende a rimuovere un'applicazione appena installata se rileva un malfunzionamento, mentre il 15% sceglie di rendere pubblico il proprio disappunto con una segnalazione o un rating negativo.

Un errore che molti commettono è considerare gli utenti come un testing team gratuito che collaborerà, utilizzando l'applicazione, alla prossima stable release. In realtà il cliente spesso non ha ne l'intenzione ne la consapevolezza di segnalare i malfunzionamenti.



Con queste premesse, in un mercato dove le chiavi del successo sono lo starring ed il numero di

download, la fase di testing non dovrebbe mai essere tralasciata dalle aziende che intendono ottenere profitto rapidamente ed un'ampia fetta del budget dovrebbe sempre essere destinata a questa attività.

A testimonianza di questa tendenza nel resoconto del World Quality Report 2014 vengono presentati i seguenti dati: "*Con la rapidità e la portata dei feedback provenienti dai social e dai media online, le grandi aziende sono diventate più consapevoli dell'impatto dannoso che eventuali errori nelle applicazioni provocherebbero alla reputazione aziendale e al valore del brand. La spesa media in percentuale per QA del budget IT complessivo è passata dal 18% nel 2012 al 23% nel 2013, arrivando al 26% nel 2014 (nel Sud Europa e in Italia l'investimento QA per quest'anno toccherà il 20%). La quota di budget per il testing è destinata a crescere ulteriormente nei prossimi anni, con una previsione che raggiunga il 29% entro il 2017 (22% in Italia).*"[2]

Per ottimizzare le risorse investite bisogna considerare una serie di trade-off e progettare una strategia di testing che garantisca una sufficiente copertura degli scenari calcolata tenendo conto del target e delle dimensioni del progetto.

Come vedremo dettagliatamente in seguito, automatizzare è quasi sempre una soluzione vincente ma, le opzioni tra cui scegliere per quanto riguarda i tempi, le modalità e gli strumenti di testing, sono così tante che solo una combinazione consapevole consente di avere un bilanciamento adeguato tra time to market, costi e qualità del risultato.

Nei paragrafi successivi saranno analizzati i principali vantaggi competitivi acquisibili applicando le tecniche di testing automatico con particolare riferimento al mondo Android ed al ben noto meccanismo di recensione e valutazione offerto dal Play Store.

1.1.1 Time to market

Arrivare sul mercato prima degli altri è sempre un vantaggio, nel mobile lo è ancora di più. Statisticamente la prima applicazione robusta a presentarsi al pubblico in una certa categoria ha altissime probabilità di conservare lungamente la vetta della classifica.

A tal scopo obiettivo primario delle aziende deve essere quello di rilasciare nel minor tempo possibile un software sano e compatibile con il maggior numero possibile dei dispositivi sul

mercato. Introdurre l'automatizzazione in tutte le fasi del ciclo di vita del software significa accelerare l'intero processo produttivo aumentando significativamente la qualità.

Ancora nel World Quality Report 2014 leggiamo in proposito che "*Con la pressione sui CIO per ridurre il time-to-market, lo studio evidenzia come diverse organizzazioni hanno adottato e implementato metodologie di delivery più agili. Oltre nove organizzazioni su dieci (93%) intervistate usano metodologie agili nello sviluppo di nuovi progetti (con un aumento dell'83% nel 2013). Ma nonostante questa crescita, molte aziende stanno ancora affrontando alcune sfide come la mancanza di un approccio agile e consolidato al testing (61%), la difficoltà ad applicare la testing automation all'approccio agile (55%) e la mancanza di disponibilità di strumenti di testing agili adeguati (42%)*". [2]

Oltre alla prima release va tenuto presente che gli utenti mobile sono abituati a ricevere frequenti aggiornamenti per le applicazioni più utilizzate e si aspettano un rapido bug fixing e la costante introduzione di nuove feature. Anche in questa fase essere in gradi di verificare rapidamente che le modifiche apportate non abbiano intaccato la qualità del software e le funzionalità stabili costituisce un ulteriore vantaggio rispetto alla concorrenza.

1.1.2 Time&Money saving

Sebbene l'introduzione di processi per il testing automatizzato abbia costi di startup alti esso si traduce a medio termine in un netto risparmio per l'azienda. La fase di inizializzazione richiede sicuramente tempo per la formazione degli addetti, per la progettazione dei casi di test e la stesura degli script e denaro per quanto riguarda le licenze dei tool, l'infrastruttura su cui funzioneranno e lo sforzo iniziale di planning ma questi costi diventano pressoché nulli durante le fasi successive.

All'aumentare del numero delle build questi costi diminuiscono garantendo un ROI più alto rispetto al tradizionale testing manuale. A lungo termine automatizzare si converte in un risparmio economico, è sicuramente meno dispendioso avere un server che esegue i test a intervalli regolari piuttosto che pagare un team specializzato. E' inoltre cosa nota che più avanti nel ciclo di sviluppo un bug viene individuato più costosa sarà la sua risoluzione.

1.1.3 Scalabilità

Gli script realizzati per eseguire i test automatici possono essere facilmente utilizzati in parallelo su emulatori di varie tipologie, su dispositivi reale ed in ambienti cloud per verificare il funzionamento dell'applicazione in un gran numero di scenari differenti. Inoltre sono disponibili un gran numero di framework che garantiscono il funzionamento degli script prodotti in tutte le situazioni descritte. In termini economici, maggiore è il numero di dispositivi con i quali il software è compatibile maggiore sarà la potenziale clientela.

1.1.4 Rating sui market

Rilasciare velocemente il software è importantissimo ma ancora più importante è fare in modo che quanto rilasciato sia gradito alla clientela avviando l'equivalente digitale del classico "passaparola". I canali di distribuzione preferenziali (se non obbligati) per le applicazioni mobile sono i market messi a disposizione dalle differenti piattaforme. Gli utenti scelgono, pagano, scaricano e valutano le applicazioni di loro interesse in questo ambiente "protetto" in cui ogni software è garantito in termini di sicurezza e compatibilità. Punto critico di questa catena di distribuzione è proprio la valutazione. Un software bacato riceverà inevitabilmente un rating negativo che, oltre a ridurne la visibilità, dissuaderà i potenziali clienti.

1.2 La sfida del mobile

Rispetto al mondo del software desktop e di quello web il panorama mobile è estremamente più eterogeneo. Dal punto di vista commerciale questa è un'opportunità interessante per aumentare la clientela diffondendo il proprio prodotto in altri paesi e su tanti dispositivi, dal punto di vista tecnico, invece, doversi confrontare con una moltitudine di device diversi per hardware e software è una sfida complessa che può essere gestita soltanto con un'elevata copertura dei casi di test. I paragrafi seguenti analizzano e discutono ciascuna delle criticità insite nello sviluppo e quindi nel tesiting di applicazioni mobile.

1.2.1 Varietà dell'hardware

Fatta eccezione per il mondo Apple che offre al mercato una ridotta gamma di dispositivi, il

panorama Android e Windows è estremamente variegato.

Oltre alle differenze banali tra produttori e modelli è necessario tenere in considerazione che spesso non sono solo gli smartphone ad utilizzare le applicazioni realizzate ma anche una vasta gamma di prodotti di nuova generazione tra cui tablet, smartwatches, smart tv e oggetti di domotica.

Ciascuno ha inoltre una dotazione in termini di interfacce e sensori che potrebbe differire dai modelli simili introducendo un ulteriore livello di complessità.

Le api messe a disposizione non garantiscono una compatibilità con tutto l'hardware su cui l'applicazione potrebbe trovarsi a girare, d'altra parte è pressoché impossibile testare e quindi garantire il funzionamento su tutto. Altro aspetto non banale legato all'hardware è quello delle prestazioni direttamente correlate alla disponibilità di CPU e memoria.

1.2.2 Varietà del software

Se l'hardware è un problema importante da considerare quello del software lo è ancora di più. I principali sistemi operativi mobile in commercio hanno piattaforme di sviluppo completamente differenti tra loro e sebbene esistano dei tool in grado di operare la conversione di un'applicazione sviluppata in un linguaggio specifico in questa sede non prenderemo in considerazione i problemi legati allo sviluppo per differenti SO.

Come sempre il migliore caso di studio è Android che è presente sul mercato su migliaia di diversi dispositivi con una grandissima varietà di major e minor releases come mostrato dalla tabella sottostante.

Ciascuna versione di Android, contraddistinta da un API Level, aggiunge e sottrae qualcosa alla precedente rendendo per gli sviluppatori molto difficile sfruttare le ultime potenzialità offerte continuando a conservare la retrocompatibilità.

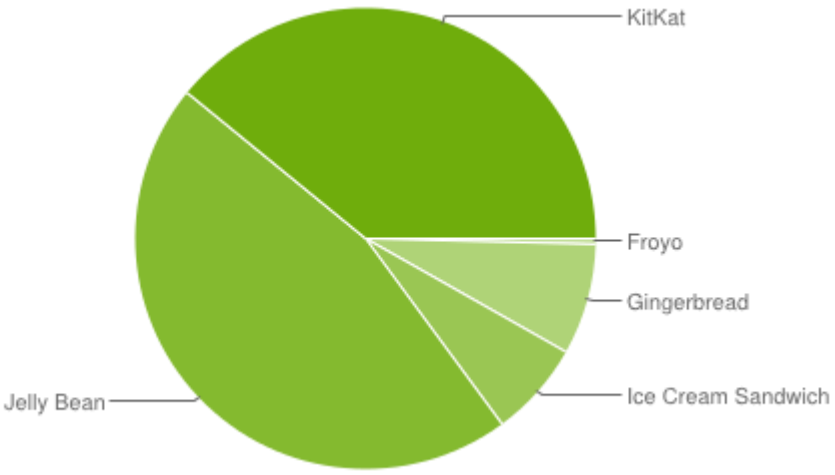
Supportare anche versioni meno recenti del SO dà accesso ad una fetta più ampia di mercato, test script realizzati ad hoc possono garantire che nel corso del suo ciclo di vita il software mantenga la backword compatibility desiderata.

Anche nel caso in cui non sia l'applicazione a modificarsi l'automazione si rivela indispensabile al fine di verificare la robustezza del prodotto in seguito a major e minor release

del sistema operativo target.

Period	Android	iOS	Windows Phone	BlackBerry OS	Others
Q3 2014	84.4%	11.7%	2.9%	0.5%	0.6%
Q3 2013	81.2%	12.8%	3.6%	1.7%	0.6%
Q3 2012	74.9%	14.4%	2.0%	4.1%	4.5%
Q3 2011	57.4%	13.8%	1.2%	9.6%	18.0%

Tabella 1: Source: IDC, 2014 Q3



1.2.3 Differenti operatori di rete

La maggior parte delle applicazioni mobile ha bisogno dell'accesso ad internet per il proprio funzionamento, per salvare dati in cloud o utilizzare servizi web e REST. Nel panorama mondiale convivono ben oltre 400 operatori di rete mobile ciascuno dei quali supporta numerose tecnologie: alcune molto diffuse come LTE, CDMA, GSM, ed altre di minore importanza relative a standard locali come iDEN, FOMA, e TD-SCDMA.

Il punto critico in questa infrastruttura è il passaggio da una rete packet-based come quella cellulare a quella web basata sul protocollo TCP-IP. Questo tunneling è effettuato in modo diverso da ogni operatore. Inoltre spesso gli operatori utilizzano proxy e filtri per limitare particolari categorie di traffico o applicazioni particolarmente band consuming. Non è insolito

inoltre che sul traffico web venga effettuato un pre-scaling per ridurre la dimensione dei dati ricevuti, specialmente nel caso di video o immagini che vengono adattati a dimensioni più consone a display mobili. Purtroppo le reti cellulari non sono molto potenti, hanno copertura limitata e quindi, per testare la connettività attraverso un particolare operatore, bisogna trovarsi nella zona dove risiedono le sue celle.

Naturalmente è impensabile viaggiare in ogni nazione per verificare la compatibilità degli operatori di rete locali e bisogna accettare un compromesso.

Testare su reti Wi-Fi è semplice, economico ed accettabile nelle prime fasi dello sviluppo ma esistono anche dei tools per la simulazione di reti usando connessioni WAN/Wi-Fi. Questi strumenti si rivelano preziosi specialmente per simulare le varie ampiezze di banda (es. 2G, 3G, and 4G) e quindi le prestazioni dell'applicazione in differenti scenari.

1.2.4 Native, hybrid e web application

Le applicazioni per smartphone possono essere realizzate in tre diverse modalità, applicazioni native, applicazioni web mobile e ibride. Di seguito considereremo brevemente i pro e i contro di ciascuna.

Le applicazioni native sono quelle che è possibile scaricare abitualmente dai market online e che risiedono stabilmente nella memoria del device. Sono progettate e realizzate per una specifica categoria di dispositivi e pertanto possono sfruttarne tutte le potenzialità in termini di performance, widget e gestures offerti dalle API, sensori e interfacce. Nel caso Android in particolare la volontà di utilizzare le risorse dell'hardware ospite vengono esplicitamente dichiarate ed accettate dall'utente in fase di installazione garantendo quindi all'applicazione il permesso di operare.

Dal punto di vista commerciale con le applicazioni native è possibile fidelizzare il cliente attirandone l'attenzione con notifiche e aggiornamenti.

Il principale svantaggio è che le applicazioni native soffrono maggiormente del device challenge e risulta molto oneroso il loro porting da una piattaforma all'altra. Infatti i più diffusi sistemi operativi mobile, Apple e Android, differiscono tra loro anche per il linguaggio di programmazione utilizzato. Per quanto riguarda il testing di applicazioni native bisogna

considerare che il ciclo di vita del software comprende oltre all'esercizio anche l'installazione ed un certo numero di aggiornamenti che vanno adeguatamente previsti e testati.

Le applicazioni web non sono altro che versioni ottimizzate per il mobile di web applications spesso già esistenti accessibili dal browser.

Il loro sviluppo è molto economico perché richiede l'utilizzo di tecnologie molto comuni ed in breve tempo è possibile realizzare un'applicazione cross-platform e cross-device. Tuttavia la semplicità di realizzazione non è in grado di ripagare una serie di svantaggi, tra cui il fatto che una web application necessita della connessione ad Internet, ha un ridotto numero di funzionalità e può interagire con l'hardware del device in modo limitato. Inoltre le connessioni degli utenti non sempre hanno velocità sufficienti a garantire una buona user experience in applicazioni che utilizzano massicciamente linguaggi lato client come javascript e che quindi richiedono ad ogni accesso il download di librerie e script degradando le prestazioni del device. Non ultimo è da considerare il ben noto problema della cross compatibility tra browser.

Dal punto di vista commerciale le applicazioni web non risiedendo sul device devono esplicitamente essere cercate e memorizzate dall'utente e sfuggono all'enorme bacino di utenza convogliato dai market.

Un punto di incontro tra queste due tecnologie sono le applicazioni ibride, esse seguono il ciclo di installazione/aggiornamento/disinstallazione di una comune app nativa ma la maggior parte delle funzionalità è realizzata con linguaggi web e visualizzata attraverso il browser. In tal modo i costi del porting sono estremamente ridotti e spesso addirittura azzerati grazie alla disponibilità di tools ad hoc.

1.2.5 Interruzioni

Non bisogna trascurare il fatto che mentre un'applicazione è in esecuzione su un device potrebbe essere interrotta da un processo a più elevata priorità. Nel caso più ovvio per gli smartphone la ricezione di una telefonata o di un SMS potrebbero provocare la sospensione del normale flusso di esecuzione, altrettanto improvviso potrebbe essere il collegamento di un cavo USB o la segnalazione di una ridotta percentuale di batteria. La criticità in questo caso risiede nel prevedere adeguati meccanismi di recovering per garantire una corretta ripresa del

funzionamento.

Tutti i sensori e le interfacce del device costituiscono una possibile fonte di interruzione da gestire correttamente.

1.2.6 Orientamento e configurazioni

L'utente è padrone del proprio device e può decidere arbitrariamente di attivare o disattivare funzionalità, di crittografare i dati, di consentire o vietare determinati tipi di accessi ai sensori e alla rete. Un'applicazione robusta deve tener conto delle infinite combinazioni possibili e non può farlo se non generando automaticamente in modo casuale una serie di test case sufficienti a coprire il maggior numero possibile di differenti comportamenti. In questa categoria può essere inserito uno dei principali e più diffusi errori di sviluppo grafici e cioè la gestione del cambio di orientamento da portrait a landscape e viceversa.

1.2.7 Localizzazione

Oltre alle succitate differenze nella rete cellulare la diffusione di un'applicazione all'esterno del paese di origine ha anche altri risvolti nello sviluppo. I più banali punti di attenzione in questo caso è la verifica di fusi orari, unità di misura, charset e valuta.

1.3 I compromessi del testing

In un contesto aziendale realistico non è pensabile investire risorse infinite nel testing, molto spesso, si stabilisce una soglia accettabile di copertura si mette a punto una strategia che consenta di raggiungerla nel modo più economico possibile in termini di tempo e risorse. Di seguito saranno analizzate alcune delle principali scelte da operare durante la fase di progettazione dei test presentandone vantaggi, svantaggi e combinazioni ideali. Tali scelte sono spesso dettate da trade-off economici o da valutazioni realistiche sulla natura e sul target dell'applicazione oggetto di test.

1.3.1 Emulatori vs Real Device

Abbiamo già sottolineato quanto la varietà di dispositivi mobili costituisca un limite per il

testing.

Testare su dispositivi reali ha il vantaggio di consentire di analizzare tute le limitazioni e gli imprevisti dovuti all'hardware e al software che i clienti andranno effettivamente ad utilizzare. Ovviamente un approccio orientato a testare sul maggior numero possibile di dispositivi reali si rivelerebbe estremamente dispendioso, non solo bisognerebbe acquistare un gran numero di device ma sarebbe necessario anche dotarli di una connessione dati e telefonica. L'enorme offerta di applicazioni non rende vantaggioso per produttori e operatori telefonici offrire un noleggio agevolato o il prestito a scopo di test. Tutto ciò a discapito delle realtà aziendali più piccole. Senza contare che sarebbe complesso gestire un magazzino con centinaia di dispositivi e senza una solido protocollo per la produzione, collezione e analisi dei risultati il testing su device finirebbe per diventare disorganizzato e laborioso. La soluzione ideale per chi sceglie questa strada è individuare un subset di dispositivi che ricoprano le principali categorie target e che godano di un'ampia diffusione. Salvo esigenze specifiche è buona regola evitare di testare su device fuori produzione o con software non più supportato per non incorrere in aumenti esponenziali dei costi.

Dall'altra parte della barricata troviamo gli emulatori, estremamente più economici e di facile gestione. E' sufficiente scaricare il profilo di un nuovo device, messo a disposizione gratis dai produttori, per avere istantaneamente una replica fedele sulla quale testare la compatibilità del prodotto. Poiché l'emulatore nasce proprio allo scopo di agevolare il testing esso è dotato di meccanismi per una diagnostica dettagliata e la raccolta di log ed errori.

Il risparmio deriva dal fatto che una sola piattaforma, se aggiornata frequentemente, consente di far girare l'applicazione su ogni nuovo device presentato sul mercato.

Nonostante questi aspetti positivi gli svantaggi sono numerosi: non è possibile emulare la varietà di hardware faults e interruzioni a cui un dispositivo reale andrà incontro durante l'utilizzo, non è possibile avere un'immagine precisa al pixel della GUI sullo schermo, non è possibile effettuare analisi sulle performance in quanto le risorse sono dipendenti da quelle del sistema che ospita l'emulatore, infine non è possibile testare le interazioni con l'ambiente circostante che potrebbero avere influenza sul dispositivo.

In sintesi è possibile affermare che l'emulatore è ideale per le prime fasi del testing in cui è

necessario ancora individuare defect di tipo funzionale e grafico mentre per test più sofisticati, ad esempio prestazionali e operazionali, è necessario prevedere sessioni di test su dispositivi reali.

1.3.2 Cloud solutions

Una soluzione che consente di testare su un gran numero di dispositivi reali pur godendo di tutti i vantaggi di un emulatore è affidarsi ad un provider di servizi cloud.

Questi provider mettono a disposizione un gran numero di dispositivi reali dotati di un'unità di controllo che consente di utilizzarli da remoto attraverso applicazioni web. In tal modo è possibile riprodurre la pressione di un pulsante, raccogliere screenshot ed utilizzare la tastiera vedendo live quello che succede sul dispositivo.

La responsabilità dei provider è garantire una copertura sufficiente per gli aspetti più critici del testing come la diversità dei device, la localizzazione e la riproduzione di scenari utente come la ricezione di una chiamata o di un SMS. Il tutto fornendo dati di diagnostica e log per consentire l'analisi delle performance oltre che del funzionamento. Spesso è il cliente a definire gli indicatori e i parametri vitali del sistema che desidera tenere sotto controllo durante l'esecuzione dei test.

Il costo di questa soluzione è notevole ma ridotto rispetto all'acquisto di un gran numero di device, inoltre il cloud è condiviso con altri utenti e pertanto si paga soltanto il tempo di testing effettivo sui device di interesse.

Il World Quality Report 2014 riporta che tra le priorità per le aziende IT vi è la limitazione dei costi dell'infrastruttura di test "*Dopo un calo nel 2013, l'adozione del cloud per l'hosting e il testing di applicazioni è tornato a crescere. Il testing nel cloud è aumentato dal 24% dello scorso anno al 32% nel 2014 e si stima una crescita al 49% entro il 2017*". [2]

1.3.3 Manual vs Automated testing

Esistono numerosi modi di progettare ed eseguire il testing di un'applicazione mobile e molto spesso nessuno di essi può essere definito completamente giusto o sbagliato. Uno dei trade-off che le aziende si trovano a valutare più frequentemente è la scelta tra l'esecuzione manuale ed

automatica dei test pianificati.

Un primo vantaggio rilevabile nell'utilizzo di test automatici è la velocità di esecuzione e la loro infinita ripetibilità. Anche la compatibilità del test tra i vari device target è garantita dai numerosi framework, open source ed a pagamento, che, se l'organizzazione è efficiente, consentono agli sviluppatori stessi o al QA team di produrre a basso costo i test necessari.

Uno svantaggio insito nella scrittura di test automatici è la possibilità di introdurre dei bug nel test stesso, tuttavia garantisce di testare tutto quello che si desidera nel modo che si desidera ed è un buon approccio se il team è in grado di utilizzare linguaggi di programmazione e quindi di adattare gli script prodotti ai cambiamenti dell'applicazione. Purtroppo produrre manualmente il test è un'operazione che ha tempi di startup molto lunghi e che risulta inizialmente dispendiosa in quanto richiede personale in grado di programmare e bisogna prevedere la manutenzione dei test man mano che l'applicazione si evolve.

Questi costi iniziali vengono però assorbiti all'aumentare del numero di esecuzioni del test prodotti, traducendosi al superamento di una certa soglia in un netto risparmio.

Le esecuzioni multiple effettuate da una macchina sono inoltre prive della percentuale di errore umano che un tester introdurrebbe eseguendo numerose volte il caso di test.

Un errore diffuso è la tendenza all'eccessiva automatizzazione, bisogna accuratamente selezionare i test che è conveniente automatizzare. Ad esempio è sconsigliabile automatizzare il testing di feature candidate a cambiare nell'immediato futuro o soggette a modifiche frequenti. Bisogna considerare che casi di test molto complessi potrebbero avere costi di automatizzazione troppo elevati rispetto all'esecuzione manuale degli stessi.

In sintesi i casi in cui si rivela più conveniente automatizzare sono:

- la verifica della compatibilità dell'applicazione con una nuova versione del sistema operativo,
- la verifica che l'introduzione di nuove caratteristiche non abbia alterato la retrocompatibilità dichiarata sfruttando nuove API,
- i test di regressione,
- i test più frequenti e di lunga esecuzione,
- i test che hanno risultati predicibili,

- tutto quello che è facile da automatizzare in modo da ridurre i costi di realizzazione.

Dall'altra parte, eseguire manualmente i test ha come primo evidente svantaggio la lentezza, per poter accelerare e parallelizzare il testing su più dispositivi è necessario ingaggiare un numero maggiore di risorse umane con un conseguente aumento dei costi fissi. Il principale vantaggio è che per quanto accurati ed evoluti i framework non riescono a riprodurre in tutto e per tutto le interazioni di un un utente umano. Inoltre per avviare una sessione di test manuale non è necessaria alcuna infrastruttura se non il device reale o emulato ed il tester.

Per queste ragioni il testing manuale trova maggiore spazio nelle realtà aziendali minori o tra gli sviluppatori individuali dove spesso non esiste una divisione tra il team si sviluppo e quello di testing e non sono disponibili fondi per l'infrastruttura di automatizzazione.

1.3.4 Scripting

Nell'ambito della test automation si definisce scripting il processo di progettazione e stesura dei test. Questa fase consiste nel produrre un documento, un foglio di calcolo o un insieme di files sorgenti in cui tutti i casi di test sono raccolti assieme ai risultati attesi per ciascuno, in modo che per ogni esecuzione si possa determinare lo stato di fallimento o successo.

Più comunemente per test script si intende l'unità di software alla base della testing automation, realizzata in un linguaggio di programmazione.

Per fare in modo che gli script prodotti siano eseguibili ovunque e che possano adattarsi a tutti i dispositivi, indipendentemente dalle caratteristiche hardware e software, i linguaggi e le librerie utilizzati devono avere un elevato livello di astrazione. A tal scopo è possibile utilizzare framework e software di testing specializzati che forniscono linguaggi di alto livello per interagire con i widget del device in maniera indipendente dalla piattaforma.

Come analizzato nel paragrafo precedente, il principale svantaggio della produzione di script automatizzabili è il tempo necessario alla loro stesura.

Un buon compromesso tra eseguire i test a mano e sviluppare script automatici risulta essere registrare le interazioni di un utente reale con l'applicazione e convertirle in uno script rieseguibile. In questo modo anche uno scenario di test complesso può essere reso automatico in pochi minuti. I costi di questa tecnica sono molto bassi se si esclude l'acquisto di un tool

specifico. Il test può essere eseguito da persone con esperienza minima purché il dispositivo sia collegato all'ambiente di sviluppo.

A causa delle differenze tra dispositivi e dei differenti livelli di astrazione dei linguaggi non è possibile ottenere una registrazione perfetta ma è possibile produrre script funzionanti intervenendo in percentuale minima sul codice prodotto.

Infine è possibile eliminare completamente la fase di stesura dei test affidandosi a dei tool in grado di ispezionare l'interfaccia dell'applicazione alla ricerca di widget e bottoni. Sebbene questi strumenti siano in grado di esplorare completamente le schermate di un'applicazione il loro utilizzo si rivela utile solo nelle prime fasi dello sviluppo in quanto non è possibile alimentarli con dati e prevedere combinazioni di input complesse.

1.3.5 Data Driven Testing

Si definisce Data Driven Testing una tecnica in cui tutti i parametri, le condizioni, gli input e gli output di un caso di test non sono cablati nel codice di scripting ma vengono prelevati ad ogni esecuzione da un sorgente dati specifica in modo da realizzare la separazione della parte logica.

Molto spesso in applicazioni complesse, il numero di classi di equivalenza dei valori di input che è possibile inserire è troppo alto per poter essere coperto da test manuali o da altrettanti test script. In tali casi in genere si cerca di effettuare solo i test più significativi.

Il Data Driven Testing risolve questo limite garantendo i seguenti vantaggi:

- Può essere progettato per ogni singolo caso d'uso già durante la fase di sviluppo e riutilizzato in più casi di test
- Aumenta la ripetibilità e la riusabilità dei test automatici riducendone la complessità: è possibile riutilizzare lo stesso script migliaia di volte con dati diversi anziché produrre uno script che replica migliaia di volte le stesse operazioni.
- Migliora la manutenibilità: in caso di aggiunta/modifica di campi di input e regole di validazione.
- Separa la logica del test dai dati rendendo gli script più lineari, leggibili e indipendenti dai dati.
- Produce casi di test più realistici : avere la possibilità di modificare spesso i dati di test,

eventualmente producendoli di volta in volta in modo random, aumenta la probabilità di individuare nuovi bug.

- Riduce il numero di Test Case/Script: la separazione consente di rendere modulari e riutilizzabili sia I dati che gli script. Si evitano inutili duplicati e la gestione risulta snellita.

1.4 I tempi del testing e le responsabilità degli sviluppatori

Per produrre software di qualità non è sufficiente prevedere una fase di testing finale, una volta terminato lo sviluppo i bug emersi potrebbero essere costosissimi in termine di rework. Presto e spesso sono le key word per una test strategy di successo.

E importante che nel processo di quality assurance tutti siano coinvolti, progettisti, sviluppatori e tester fin dai primi stadi dello sviluppo.

Già durante la prima stesura del codice ciascuno sviluppatore può essere in grado di individuare potenziali sorgenti di bug e bad practice da evitare. Grazie ai sempre più sofisticati tool per la Static Code Analysis i bug latenti possono essere individuati fin dai primissimi giorni di vita del software.

La Static Code Analysis non fa altro che leggere il software e analizzare i costrutti per individuare potenziali fonti di malfunzionamento e segnalare gli statement che non rispettano gli standard qualitativi e le guideline. E' un'attività che spesso viene citata come parte del White Box Testing ed ha lo scopo di individuare le vulnerabilità nel codice sorgente utilizzando tecniche come Taint Analysis and Data Flow Analysis.

Molto spesso questi tool di analisi sono integrati con i principali IDE per fornire feedback immediati allo sviluppatore riguardo la qualità del software prodotto.

Uno dei tool più utilizzati a tal proposito, specialmente nel mondo Android, è Lint.

Grazie a tool come Lint e Sonar oltre all'analisi è possibile estrarre metriche di qualità come la coverage, la percentuale di duplicazioni, l'eccessivo accoppiamento etc.

1.4.1 Test driven developement

Il Test Driven Development (TDD) è un modello di sviluppo del software che prevede che la stesura dei test automatici avvenga prima di quella del software che deve essere sottoposto a test, e che lo sviluppo del software applicativo sia orientato esclusivamente all'obiettivo di passare i test automatici precedentemente predisposti.

Più in dettaglio, il TDD prevede la ripetizione di un breve ciclo di sviluppo in tre fasi, detto "ciclo TDD". Nella prima fase (detta "fase rossa"), il programmatore scrive un test automatico per la nuova funzione da sviluppare, che deve fallire in quanto la funzione non è stata ancora realizzata. Nella seconda fase (detta "fase verde"), il programmatore sviluppa la quantità minima di codice necessaria per passare il test. Nella terza fase (detta "fase grigia" o di refactoring), il programmatore esegue il refactoring del codice per adeguarlo a determinati standard di qualità.

1.4.2 Continuous Integration

Le applicazioni mobile spesso seguono un ciclo di sviluppo agile contraddistinto da release frequenti con piccoli incrementi delle feature. Un motivo per rilasciare aggiornamenti potrebbe essere trarre beneficio dalle nuove API e dalle nuove versioni del Sistema Operativo, che potrebbero migliorare e/o velocizzare la user experience. Ogni upgrade per quanto minimo richiede un ciclo di test addizionale.

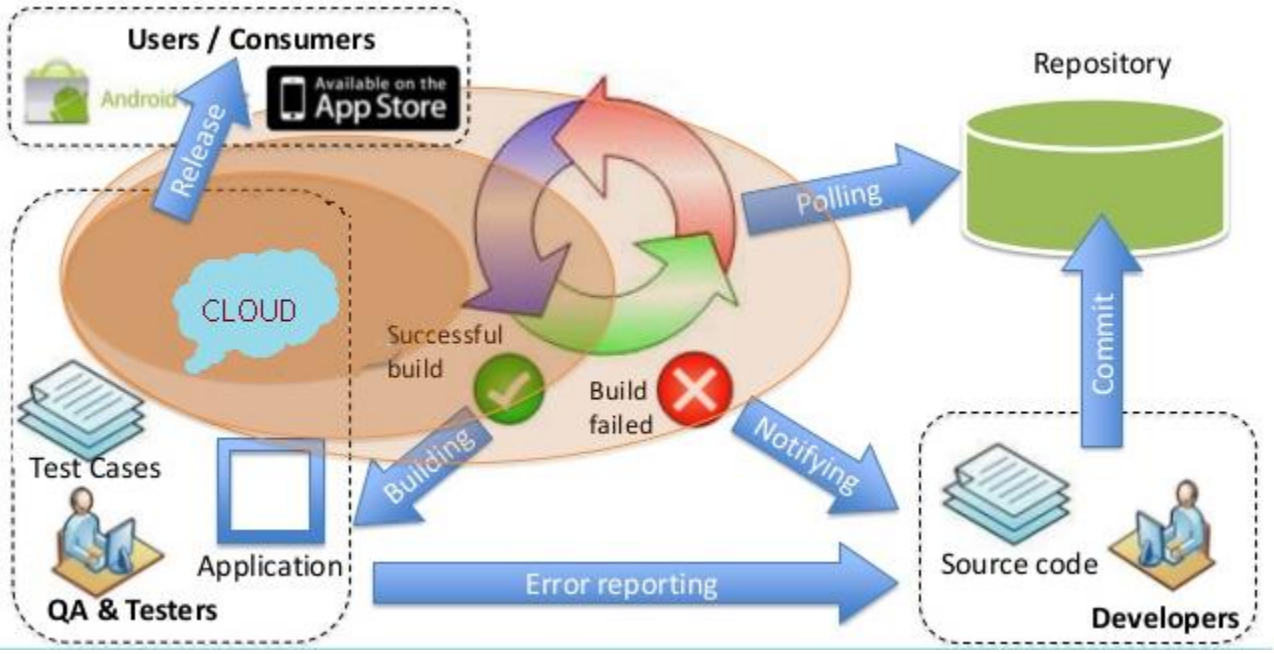
Similmente l'introduzione sul mercato di un nuovo modello di device candidato come top di gamma richiede un'immediata verifica di compatibilità.

Ogni ciclo di test ha un'ampiezza che dipende dalla natura del cambiamento intervenuto, si può passare attraverso i cosidetti "smoke" o "sanity" test per le fix di minor rilievo fino a una suite completa di regression test.

Un processo di sviluppo e rilascio così dinamico deve essere necessariamente strutturato per evitare disorganizzazione e rallentamenti.

La continuous integration è una pratica di gestione del ciclo di vita del software applicata in contesti dove più sviluppatori collaborano, lavorando contemporaneamente sugli stessi file sorgente, e dove sono necessarie frequenti build e release. Con tale tecnica, basata sulla disponibilità di un repository e di un tool per il versioning dei file, più volte al giorno le

modifiche rilasciate vengono integrate, compilate, installate e testate in un ciclo continuo che garantisce la rilevazione in tempo reale di eventuali bug di regressione o derivanti dall'integrazione.



Appare evidente quanto questo approccio possa rivelarsi prezioso nel mondo mobile dove è necessario rapidamente rilasciare nuove feature garantendo che le funzionalità preesistenti non siano state intaccate.

Poiché i tool per la CI gestiscono le varie fasi in maniera completamente automatica è necessario che i test siano realizzati sotto forma di script automatici e sottoposti a versioning come tutto il resto del codice. In qualità di sorgenti, gli script di test possono a loro volta essere soggetti ad errori, regressioni e manutenzione che l'integrazione continua permette di individuare e risolvere il prima possibile.

Il più famoso tool opensource per la continuous integration, Jenkins, offre numerosi plugin per l'integrazione con tools e framework di varia natura nonché la possibilità di integrare script batch per l'invocazione di processi custom.

In un ciclo di vita ideale, più volte al giorno il tool aggiorna il sorgente, verifica che non ci siano errori di compilazione e ne opera l'analisi statica. I deliverable prodotti vengono poi installati su una macchina target, eventualmente un emulatore avviato dal tool stesso, ed i test

vengono eseguiti. A fine ciclo report accurati vengono prodotti e gli errori segnalati agli sviluppatori responsabili.

1.4.3 Android e il meccanismo di segnalazione

Per quanto tempo e denaro si investa nel testing sarà sempre impossibile rilasciare un software privo di errori, è dunque importante riuscire a raccogliere e risolvere segnalazioni anche quando l'applicazione è già in ambiente di esercizio.

A tal proposito molti market prevedono un meccanismo unificato per il reporting da parte dell'utente di eventuali malfunzionamenti che vengono inoltrati allo sviluppatore.

Dotare la propria applicazione di un logging efficiente è utile sia durante il testing che in fase di produzione per recuperare le informazioni dal device e indirizzare rapidamente la risoluzione del bug.

La Google Play Developer Console è la piattaforma messa a disposizione da Android agli sviluppatori per gestire il ciclo di vita delle applicazioni in vendita su Play Store, questo strumento offre anche la possibilità di vedere i dati raccolti in caso di crash e di errori ANR (Application Not Responding).

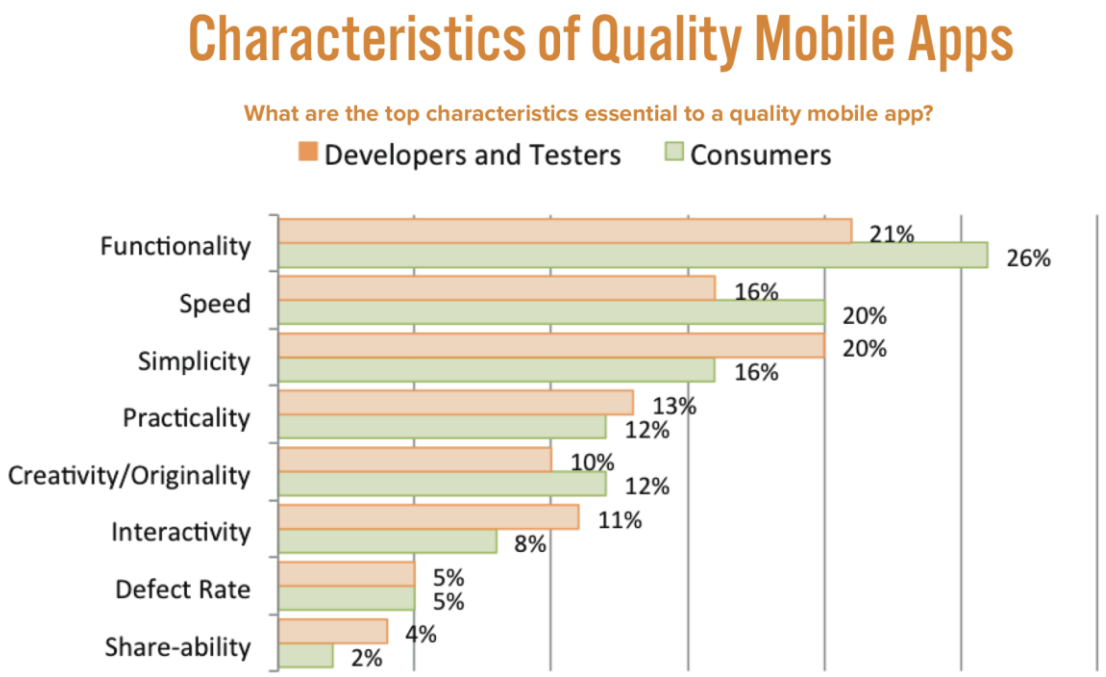
In caso di crash all'utente viene richiesto se desidera inviare una segnalazione di errore e gli si offre la possibilità di allegare un messaggio esplicativo. I dati in questi casi saranno disponibili per la consultazione soltanto se l'utente ha scelto di condividerli.

Se, invece, l'applicazione smette di rispondere, all'utente verrà proposto di chiuderla ed i dati raccolti verranno automaticamente inviati agli sviluppatori al fine di facilitare la risoluzione del problema. Sulla console saranno presentate le informazioni come il tipo di errore, la localizzazione, la frequenza con cui si è presentato. Tali report possono poi essere scaricati da Google Cloud Storage in formato CSV.

1.5 Cosa testare

Non è possibile suddividere in compartimenti stagni gli obiettivi del testing di un'applicazione, sia essa desktop o mobile, tuttavia è possibile individuare una serie di macroaree sulle quali focalizzare l'effort, tenendo comunque presente che ci saranno sovrapposizioni e che i

malfunzionamenti rilevati potrebbero spesso essere interdipendenti. Per offrire una panoramica numerica di quanto detto si riporta il seguente passo del World Quality Report. "*Il testing sta diventando sempre più importante in quanto i consumatori si aspettano una user experience coerente nell'utilizzo di applicazioni multi-canale e di dispositivi in un mondo "always-on". La ricerca mostra che la sicurezza (59%) e le prestazioni (57% - in Italia il 63%) sono le aree principali per la migrazione delle applicazioni su Cloud e che la sicurezza rimane un fattore cruciale nel testing di applicazioni mobili (54%). Tuttavia si riscontra ancora una problematica nel mobile testing in quanto quattro intervistati su dieci hanno dichiarato di non avere abbastanza tempo per testare adeguatamente le loro soluzioni mobile.*".



1.5.1 Comportamento funzionale

Questo tipo di testing spazia da quello più tradizionale, finalizzato a verificare che l'applicazione funzioni come da requisiti e che sia possibile portare a termine i casi d'uso senza intralci. Molte mobile app si comportano come client nei confronti di web service o servizi REST disponibili online. In questo caso bisogna dedicare una parte importante del testing a scenari in cui il server è offline, impiega troppo tempo a rispondere o restituisce risposte

inattese o malformate. Cosa succederebbe se improvvisamente il device perdesse il collegamento a Internet.

Nel caso si utilizzino librerie ed API è importante verificare l'interazione con esse prevedendo un'apposita fase di API testing.

1.5.2 Usabilità e GUI

La parte più esposta di un'applicazione mobile è senza dubbio l'interfaccia. Una grafica accattivante ed user friendly può essere la chiave del successo rispetto ad i concorrenti.

Durante il testing bisogna verificare che ogni schermata sia come ci si aspetta, sia in modalità landscape che portrait, bisogna validare il flusso di navigazione assicurandosi che sia facile navigare tra le schermate e che i dati vengano preservati passando dall'una all'altra. Ad ogni passo i widget (button, text inputs, labels) previsti devono essere visualizzati e correttamente popolati o validati. Ad esempio bisogna verificare la comparsa di un dialog box di errore all'inserimento di un input errato.

Tuttavia la GUI è la parte dell'applicazione che risente maggiormente dei cambiamenti tra device e device. Parametri come la dimensione in pixel dello schermo, la densità ed il form factor possono condizionare seriamente la qualità del rendering.

A tal scopo l'automazione fornisce un grosso aiuto per eseguire i test sul maggior numero di dispositivi, reali o emulati.

1.5.3 Performance e Memoria

L'analisi delle prestazioni e del consumo di risorse è forse è l'aspetto più trascurato dagli sviluppatori meno esperti. Su dispositivi che spesso hanno capacità di calcolo e memoria limitati una disattenzione in fase di sviluppo può constare preziosi cicli di CPU durante l'esecuzione o un ridondante traffico di dati in rete (spesso limitato dagli operatori o dalle offerte attive sul dispositivo).

La complessità computazionale del software condiziona negativamente non solo le prestazioni generali dell'applicazione e dell'intero device ma aumenta anche drasticamente il consumo di batteria.

L'utente scontento tende a disinstallare l'applicazione poco reattiva o troppo dispendiosa.

Ecco perché un'importante fase del ciclo di sviluppo è costituita dai performance e stress test, in grado di individuare eventuali memory leak, malfunzionamenti dovuti al sovraccarico di richieste e all'eventuale mancato rilascio delle risorse acquisite (es. GPS, fotocamera, microfono).

Anche i tool di diagnostica nativi presenti su ciasun sistema operativo possono rivelarsi utili per capire se l'applicazione subisce rallentamenti o ha un picco nella richiesta di risorse.

Altri metodi empirici per rilevare problemi prestazionali potrebbero essere :

- Surriscaldamento del device: se il device si riscalda molto probabilmente la CPU è sotto stress;
- Rapido consumo della batteria : anche questo potrebbe essere sintomo di un codice poco efficiente o di operazioni ridondanti;
- Rallentamento dell'applicazione.

1.5.4 Operational testing

Questa fase verifica che siano messi in atto tutti i controlli necessari affinché l'applicazione prosegua il proprio funzionamento anche in caso di eventi inattesi, occupandosi del backup delle informazioni necessarie e di implementare strategie di recovery per ripristinare la situazione in caso di arresto inatteso. Anche l'upgrade dell'applicazione è un momento critico durante il quale è possibile la perdita dei dati utente se non correttamente gestita.

Tra i più comuni eventi che potrebbero disturbare il comune svolgimento delle funzionalità ricordiamo:

- ricezione chiamate, SMS, MMS
- notifiche da parte di sensori o altre applicazioni
- plug-in e rimozione di cavi USB ed auricolari
- Interruzione della connessione
- Media Player on/off
- Device Power cycle

1.5.5 Installazione

Il software mobile può essere venduto attraverso i canali di distribuzione predisposti da ciascuna piattaforma o semplicemente distribuito standalone in formato eseguibile. In entrambi i casi una fase cruciale per l'accettazione da parte del cliente è quella di installazione.

L'installation testing verifica che il processo di installazione vada avanti senza che l'utente incontri alcuna difficoltà e che tutte le strutture dati e l'eventuale porzione di filesystem necessarie siano allocati senza problemi.

Naturalmente questa fase di testing include anche il processo di upgrade nel caso di ulteriori release e quello di disinstallazione.

1.5.6 Localizzazione

La localizzazione consiste nel modificare un'applicazione esistente per l'utilizzo in un'area geografica specifica . Tipicamente ciò include la traduzione di ogni testo nella lingua locale con un conseguente cambio di charset, cambio di valuta, formato delle date e tanti altri aspetti che richiedono opportune conversioni. In casi limite è addirittura necessaria l'inversione delle direzione del testo.

La localizzazione è dunque un'operazione complessa ma può rivelarsi fondamentale per vendere il prodotto in altre nazioni. Gli utenti privati, in media, preferiscono installare un prodotto che utilizzi la loro lingua mentre le aziende necessitano di applicazioni che supportino le valute e le date locali .

1.5.7 Sicurezza

Le applicazioni che scambiano dati sensibili in rete devono garantire la privacy utilizzando meccanismi di crittografia ed autenticazione (HMAC-SHA).

Bisogna inoltre considerare eventuali vulnerabilità e punti di accesso attraverso i quali potrebbero essere inviate informazioni maliziose. Per sviluppare in modo sicuro è necessario tenere sempre in mente che in rete non esistono fiducia e buona fede e che non è sufficiente affidarsi ai protocolli di base.

Oltre alla sicurezza in rete è necessario considerare anche quella sul filesystem, bisogna

intercettare e impedire l'accesso ai dati ad altre applicazioni o ad altri utenti nel caso di applicazioni multiutente. Spesso questo è già garantito dalle API ma bisogna assicurare all'utente la massima privacy.

Un ultimo punto di vulnerabilità, come in tutte le applicazioni, sono gli input dell'utente che devono sempre essere appropriatamente validati.

Capitolo 2: Stato dell'arte con focus sul recording

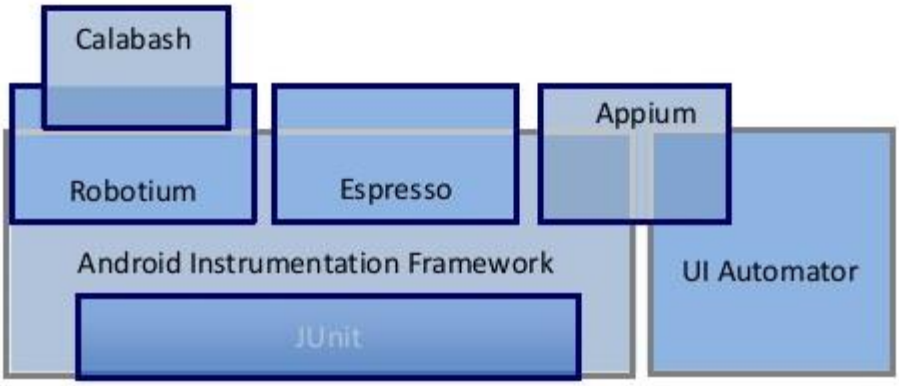
Alla luce delle caratteristiche e delle criticità del mobile testing e del crescente interesse per soluzioni di automation il mercato si è adeguato per offrire risposte alle esigenze degli sviluppatori. In questo capitolo si offre al lettore una panoramica sullo stato dell'arte delle tecnologie in materia presentando dapprima i framework più utilizzati, mattoni elementari per la produzione di test script automatizzabili e poi strumenti più evoluti che si propongono di fondere i vantaggi del testing manuale con quelli del testing automatico.

2.1 Android Automation Frameworks

L'esecuzione automatizzata dei test come evidenziato nei capitoli precedenti è una necessità imprescindibile per qualunque azienda intenda produrre applicazioni in tempi rapidi e a costi competitivi. Trattandosi di un settore complesso e molto eterogeneo è naturale che la stesura di test script possa risultare un'attività onerosa per gli addetti nonché molto vulnerabile all'introduzione involontaria di errori. Per questi motivi, a partire dal lancio della piattaforma di sviluppo Android, sono andati affermandosi una serie di framework per il testing automatizzato che hanno innalzato il livello di astrazione del linguaggio in modo da rendere i test prodotti sempre più leggibili e indipendenti dal device su cui vengono eseguiti.

La maggior parte di questi framework utilizzano il linguaggio Java e sono basati su JUnit, sia per ragioni di integrazione con gli unit test della parte business dell'applicazione che per uniformità al protocollo xUnit per la reportistica.

Alcuni, pur essendo sempre basati su JUnit, utilizzano Python, JavaScript o linguaggi



proprietari.

Nei paragrafi seguenti viene proposta una panoramica dei principali prodotti del settore.

2.1.1 Android Instrumentation Framework

L'Android Instrumentation Framework è un set di API incluso nell'Android Developement Kit che consente sia la scrittura che l'esecuzione di test per applicazioni Android.

Le principali caratteristiche di questo framework sono:

- Test suite basate su JUnit: ciò consente di utilizzare contemporaneamente sia le funzionalità standard per testare le classi che non comprendono chiamate alle API Android che le estensioni necessarie ad interagire con i componenti della UI e le altre interfacce.
- Classi di test specifiche per ciascun component: queste classi offrono metodi di utilità per la creazione di oggetti mock e la gestione specifica del ciclo di vita.
- Struttura dei package e dei progetti di test simile a quelli dell'applicazione: netta riduzione dei tempi di apprendimento necessari allo sviluppo dei test.
- Integrazione con i principali IDE: i plugin disponibili per Eclipse ed Android Studio consentono di accelerare le procedure di creazione dei progetti, delle alberature e dei files di configurazione.
- Funzionalità disponibili anche da command line: per gli utenti di altri IDE e per l'integrazione in processi di CI.

L'approccio dell'Android Testing API consiste nel prevedere dei punti di ingresso all'interno del ciclo di vita di ciascun componente in modo che i principali event handler possano essere invocati programmaticamente oltre che dagli input dell'utente; in questo modo è possibile replicare in tutto e per tutto via script una reale esecuzione dell'applicazione.

Una delle potenzialità di questo framework è la possibilità di eseguire sia il progetto di test che l'AUT nello stesso processo in modo che i test possano invocare direttamente i metodi dell'applicazione.

La classe che si occupa di gestire il running dei test è InstrumentationTestRunner che comunica con il sistema attraverso le API per avviare e caricare i metodi di test. Ogni volta che un Test Project viene installato ed eseguito sul device tutte le eventuali istanze dell'AUT vengono eliminate. L'unico metodo invocato autonomamente dal framework è l' *onCreate()* della main activity, tutti gli altri sono avviati esplicitamente dai metodi di test.

2.1.2 Robotium

Robotium è sicuramente uno tra i più diffuse framework opensource per l'Android Test Automation. E' pensato per facilitare ed automatizzare il testing automatico black-box dell'interfaccia grafica di applicazioni Android.

Le principali potenzialità sono:

- Testing di applicazioni Android sia native che ibride.
- Testing sia da sorgente che da APK
- Minima conoscenza richiesta dell'AUT.
- Gestione di Android activities multiple.
- Velocità di implementazione.
- Linguaggio di test human readable.
- Test case robusti grazie al run-time binding verso i componenti UI.
- Motore di esecuzione dei test.
- Integrazione con Maven, Gradle ed Ant per eseguire i test nell'ambito di un ciclo di CI.

Robotium è stato realizzato al di sopra dell'Android Instrumentation Framework e dunque è compatibile all'occorrenza con tutte le classi di test Android. La principale classe di test

utilizzata è ActivitityInstrumentationTestCase2.

La stesura di test con Robotium richiede l'utilizzo dell'unica classe Solo.

2.1.3 Appium

Appium è uno strumento open-source per l'automazione di applicazioni mobile web, ibride e native realizzate su piattaforme iOS e Android. Le API di Appium sono condivise sia per iOS che per Android consentendo il riutilizzo del codice di test.

La filosofia ufficiale di Appium è delineata dai seguenti quattro principi:

- L'applicazione da testare non deve essere modificata in alcun modo dallo strumento di test.
- Non devono esistere vincoli sul linguaggio da utilizzare per la stesura dei test.
- Non bisogna "reinventare la ruota" ma conviene utilizzare quanto già esistente in campo di testing mobile.
- Un framework per il testing dovrebbe essere opensource non solo in teoria ma anche in pratica.

Il primo principio citato viene perseguito utilizzando framework ufficiali in modo che non sia necessario ricompilare l'applicazione o aggiungere librerie specifiche. Nella fattispecie i framework utilizzati per Android sono UiAutomator, Android Instrumentation Framework e Selendroid.

Per ottenere l'indipendenza dal linguaggio i framework dei vari vendor sono incapsulati nell'API WebDriver che attraverso il protocollo client-server JSON Wire è in grado di inviare richieste http in qualsiasi linguaggio. Le librerie client, disponibili per i linguaggi di maggiore diffusione (Java, Ruby, Python, PHP, JavaScript, e C #) non sono altro che client http.

Questa architettura rende Appium più che un test framework una libreria per l'automazione. WebDriver è diventato lo standard de facto per automatizzare i browser web ed è un progetto di lavoro del W3C. Estendere il protocollo per incontrare le necessità del mondo mobile è il miglior modo per riutilizzare le tecnologie già esistenti.

Più in dettaglio Appium è un webserver che espone servizi REST, riceve le connessioni da un

client, ascolta ed esegue i comandi sul dispositivo mobile e risponde sempre via protocollo HTTP con il risultato dell'esecuzione. L'architettura client-server garantisce l'estendibilità verso altri linguaggi e la possibilità di installare il server anche su una macchina diversa da quella di test.

La comunicazione avviene nell'ambito di una sessione con una richiesta client inviata via POST avente un oggetto JSON che specifica le funzionalità desiderata sotto forma di coppie chiave valore, la sessione è identificata da un ID mantenuto dal server ed inviato in tutte le successive comunicazioni.

2.1.4 Espresso

Espresso è il framework per l'Android Automated Testing utilizzato da Google stesso per testare le principali G-Apps com G+, Maps e Drive.

E' stato recentemente reso open source e messo a disposizione in Developer Preview su Google Code ed il suo primo obiettivo è quello di risolvere i problemi di concorrenza associati al testing UI.

Molto probabilmente sarà incluso nella Android SDK non appena sarà sufficientemente stabile.

Come dichiarato al GTAC 2013 dal team di sviluppatori di Espresso le keywords di questo framework sono : *easy, reliable, durable.*

La facilità di utilizzo è garantita dalla struttura dei comandi che possono essere invocati inline per Individuare il componente, eseguire qualche azione su di esso, verificare l'esito.

Un esempio di utilizzo combinato dei tre metodi onView() perform() check()

L'affidabilità è garantita rendendo completamente trasparenti allo sviluppatore tutte le problematiche di sincronizzazione e tempificazione delle azioni evitando frequenti sleep, timeout e wait.

Infine è robusto perché, se utilizzato correttamente, può sopravvivere alle eventuali modifiche nella UI.

2.1.5 Ui Automator

Con la UIAutomator API è possibile creare test funzionali pur non disponendo del sorgente

dell'applicazione. UiAutomator non è altro che una libreria Java che include API per la creazione di test ed un motore per la loro esecuzione automatica.

Le classi principali del framework sono:

- UiDevice: rappresenta il device sotto test e ogni test script deve essere inizializzato con l'acquisizione di tale device con la chiamata a getUiDevice(). Una volta ottenuto il riferimento al device è possibile invocare su di esso una serie di metodi per simulare la pressione di tasti hardware, per prelevare screenshot e mettere in standby il dispositivo.
- UiSelector: questa classe serve a recuperare uno specific element della GUI specificandone la gerarchia all'interno del layout dell'activity che si sta testando. La selezione può essere effettuata anche filtrando per proprietà come il nome, il valore e lo stato. La forma di selezione più immediata è quella per ID anche se non sempre è a disposizione del tester.
- UiObject: rappresenta il singolo elemento grafico nell'ambito dell'applicazione. Mediante questo riferimento è possibile scatenare su tale oggetto eventi come il click o recuperarne delle proprietà.

Vantaggi di UIAutomator:

- Può essere utilizzato su device con display a differente risoluzione,
- Gli eventi sono object based e non dipendono dalle coordinate,
- Può riprodurre sequenze complesse di azioni utente,
- Può essere eseguito su disopsitivi differenti senza modifiche al codice Java,
- Può simulare la pressione degli hardware button sul device.

Svantaggi:

- Non supporta completamente applicazioni Web e ibride (OpenGL e HTML5) perché non hanno componenti Android UI.
- La stesura dei test in linguaggio JavaScript è Time consuming
- Prima di scrivere i test è necessario identificare ciascun UI Element, a tal scopo è stato creato UIAutomatorViewer, un tool per la scansione e analisi dei componenti grafici di un'applicazione Android che ispeziona la gerarchia ed esamina le proprietà.
- Richiede che l'applicazione supporti l'accessibilità perché il tool dipende da alcune

funzionalità per l'accessibilità del'Android Framework. Anche per identificare i componenti non accessibili dell'applicazione è possibile usare le potenzialità di UIAutomatorViewer. Anche se generalmente il supporto all'accessibilità è nativo e garantito dalle API Android alcune applicazioni che utilizzano componenti custom devono premurarsi di implementare la classe AccessibilityNodeProvider.

2.1.6 Calabash

Calabash è un framework cross platform per la stesura di UI Acceptance Test in linguaggio Cucumber.

Può integrarsi sia con applicazioni sviluppate in Xamarin che con quelle realizzate nei linguaggi nativi di iOS e Android: Objective-C e Java.

L'interazione con gli elementi di interfaccia dell'applicazione può essere semplicemente automatizzata e convertita in script riutilizzabili anche in un processo di Continuous Integration.

La filosofia alla base di Calabash è il Behavior Driven Development (BDD), in cui la stesura del codice è effettuata solo dopo la definizione del comportamento, questo approccio è derivato dal Test Driven Developement ma, anziché partire da una definizione a livello di API, si parte da una specifica di livello funzionale.

L'idea essenziale è rendere eseguibili le specifiche funzionali fornendo un mezzo con cui testare le richieste effettive del cliente. BDD è destinato ad essere un processo in cui tutti i soggetti interessati lavorano per creare una comprensione comune di ciò che deve essere costruito. Si passa così dalle verifiche orientate a testare che il codice sia sviluppato correttamente a quelle che garantiscono che sia prodotto il codice corretto. Ciò elimina anche le incomprensioni nell'interpretazione dei requisiti.

Queste specifiche sono scritte utilizzando un insieme di regole grammaticali derivate dal linguaggio naturale che prendono il nome Gherkin, che a sua volta può essere testato.

L'utilizzo della metodologia BDD non è obbligatorio per usare il framework anche se raccomandato.

Il testing di applicazioni Android con Calabash richiede la disponibilità sia di un computer che

del device, le principali caratteristiche sono le seguenti:

- utilizzo sia con emulatore che con real device,
- Features file che descrive le user story che si andranno a testare, eventualmente più di una in una sola sessione di test,
- definizione degli step: è possibile scegliere tra quelli predefiniti che sono generici e utilizzabili in molti casi o produrne di custom se si ha esperienza,
- Non sono richieste modifiche all'applicazione sotto test,

L'architettura è composta da un client, che per default utilizza Cucumber, che invia i comandi all'Instrumentation Test Server installato sul device e che li comunica all'applicazione sotto test mediante il framework Robotium. La comunicazione client server avviene via socket scambiando dati in formato JSON.

La classe base del Test Server è ActivityInstrumentationTestCase2 di Android SDK.

2.2 Strumenti per il Record&Replay

Il panorama dei framework per la test automation è sempre in crescita ed, oltre ai prodotti presentati, ne esistono una moltitudine commerciali e non.

Una volta effettuata la scelta della libreria di base non resta che progettare gli scenari di test e realizzare gli script nel linguaggio desiderato.

È proprio in questa fase che emergono gli svantaggi dell'approccio automatico: realizzare una test suite equivale in tutto e per tutto allo sviluppo di un software con tutto il project management, la manutenzione e la percentuale di errori ad esso correlati.

Innanzitutto per produrre test in un linguaggio di programmazione è necessario avere personale qualificato, inoltre la prima stesura richiede tempi lunghi ed una progettazione accurata per evitare rework in seguito a modifiche dell'applicazione.

Una frequente fonte di errore legata alla stesura manuale di test automatici risulta essere la necessità di conoscere la gerarchia di ogni view per individuare correttamente via ID o altre proprietà l'elemento di interesse su cui si intende invocare un evento. A tal scopo molti tool offrono strumenti che consentono l'ispezione dell'interfaccia grafica facilitando l'individuazione

di questi identificativi.

Infine non è banale riprodurre programmaticamente in modo fedele l'interazione di un tester umano con l'applicazione.

Anche in questo caso si è trovata una soluzione tecnologica. I tool presentati di seguito appartengono tutti alla famiglia del Record&Replay e sono nati per automatizzare la fase di stesura dei test script.

Questi strumenti sono in grado di registrare le interazioni dell'utente con l'applicazione, attraverso l'emulatore o un real device, e di tradurli in script riutilizzabili.

In questo modo oltre ad un netto risparmio di tempo si ottiene uno scenario di test realistico e si elimina alla radice il problema dell'identificazione dei componenti.

Non tutti i recorder esaminati hanno le stesse potenzialità, tra le feature più utili c'è sicuramente la produzione di script che utilizzano framework opensource: in tal caso l'output prodotto ha un elevato livello di astrazione e quanto registrato in un'unica sessione può essere eseguito più volte su dispositivi con caratteristiche diverse, in ambiente cloud ed eventualmente in un processo di integrazione continua.

Nel paragrafo successivo i principali tool di recording saranno analizzati e confrontati in termine di parametri generali come ad esempio la semplicità di utilizzo e la disponibilità di documentazione, in base a parametri tecnici come la numerosità e qualità degli eventi registrabili ed in base a parametri di interoperabilità come ad esempio il formato dell'output prodotto, la possibilità di utilizzarlo in contesti diversi e là manutenibilità.

Gli svantaggi dell'approccio Record&Replay sono principalmente dipendenti dallo strumento stesso e dai suoi limiti, esistono infatti alcune interazioni complesse anche i tool più sofisticati non sono in grado di registrare, la maggior parte di essi inoltre va soggetta ad errori di temporizzazione dovuti alle eccessiva velocità dell'esecuzione manuale. I recorder analizzati consentono tutti la registrazione anche da emulatore ed in tal caso è evidente che non sarà possibile registrare gesture specifiche.

L'errore da non commettere è considerare il prodotto della registrazione come uno script finito e pronto in quanto bisogna prevedere comunque una fase di testing ed eventualmente di

correzione.

L'ideale utilizzo di questi strumenti in un processo strutturato che preveda una prima stesura dei test ottenuta grazie all'aiuto del computer ma eseguiti da esseri umani sulla base di scenari di test predefiniti. Quanto ottenuto da questa prima fase non deve poi essere utilizzato così com'è ma rielaborato eventualmente per incapsulare l'ho in un contesto Data Driven e successivamente modificato per mantenerlo aggiornato durante l'evoluzione del software. Utilizzando in modo combinato sia il recording che la stesura manuale degli script è possibile passare da un approccio completamente BlackBox, come quello ottenuto registrando le interazioni dei un tester umano, ad uno Whitebox realizzato a partire dagli script della fase precedente con l'aggiunta di istruzioni specifiche per la copertura del codice non ancora testato.

2.3 Panoramica dei recording tool considerati

Nei paragrafi che seguono saranno esaminati i recording tool selezionati ai fini del confronto. Alcuni di essi, come Bot Bot e Selendroid, non hanno una reale efficacia e sono stati analizzati soltanto a scopo di test in quanto opensource. Gli altri, prevalentemente di tipo commerciale, si sono ovviamente dimostrati molto più usabili e sofisticati anche se non è stato possibile esaminare le tecnologie alla base del loro funzionamento.

2.3.1 Bot Bot

Bot Bot è un progetto opensource realizzato da Imaginea e disponibile su GitHub. Si tratta di un tool per la Android Testing Automation che offre anche ridotte funzionalità di Record&Replay. Consente partendo dal sorgente di un'applicazione Android di registrare, archiviare ed eseguire sessioni di test.

Le principali features di BotBot sono:

- Efficacia sia su apk che sul codice sorgente
- Registrazione delle interazioni utente sia su emulatore che su real device
- Utilizzo delle funzioni di Robotium e Nativedriver come keywords nei casi di test
- Memorizzazione delle registrazioni effettuate con possibilità di modifica
- Esportazione degli eventi registrati in formato csv

- Possibilità di approccio Data Driven
- Generazione di report html riguardo l'esito dell'esecuzione dei test.

L'architettura è composta da tre moduli: server, recorder e runner.

Il server si occupa dell'effettiva registrazione, modifica e archiviazione delle sessioni di test. E' accessibile da web e da command line ed offre la possibilità di navigare tra le sessioni salvate, eliminarle, accedere in modifica ai singoli eventi e scaricare ogni sessione in formato csv. È rilasciato sia in versione standalone che come war di cui effettuare il deploy.

Il recorder è un modulo che va aggiunto all'applicazione sotto test per catturare gli eventi ed interagisce via http con il BotBot server per la loro memorizzazione e codifica.

La tecnologia utilizzata per la registrazione è AspectJ, un'estensione di Java per la programmazione aspect-oriented. Il sorgente dell'applicazione deve essere importato sotto l'IDE Eclipse, converutito in progetto AspectJ mediante l'apposito plugin ed integrato con una serie di classi e file di configurazione che determinano gli event handler da intercettare e le informazioni da inviare al server per ciascuno. L'applicazione così modificata va poi installata sul device per avviare la registrazione. Questo approccio sebbene invasivo è ampiamente estendibile e configurabile per l'intercettazione di nuovi eventi.

Il Runner, il componente che si occupa dell'esecuzione dei test e della generazione dei report, è basato su un approccio keyword driven: i casi di test sono definiti in file csv che contengono per ogni evento registrato una parola chiave che lo identifica ed una lista di parametri, questi dati vengono interpretati ed eseguiti sul device/emulatore. I framework supportati dal runner sono Selenium NativeDriver e Robotium mentre i report prodotti sono nel formato Junit e TestNG-xslt.

La documentazione è essenziale e le uniche risorse sono un gruppo su GoogleCode ed il bug tracker di GitHub. Il progetto sembra abbandonato, tuttavia, come anticipato, utilizza una tecnologia efficace ed estendibile ed ha costituito un ottimo caso di studio per la comprensione dei meccanismi di base del recording. Avviare il recorder non è stato complesso ma ha richiesto un insieme di prerequisiti e conoscenze tecniche di base.

2.3.2 Selendroid

Selendroid è un framework opensource per il testing di applicazioni Android basato sull'Android Instrumentation Framework. Le funzionalità sono disponibili sia per emulatori che per real device collegati via usb e può essere utilizzato per una sola applicazione alla volta.

L'architettura di Selendroid è composta da quattro componenti principali:

- Selendroid-Client: libreria java basata su Selenium;
- Selendroid-Server: attivo sul dispositivo assieme all'applicazione sotto test e responsabile dell'automazione;
- AndroidDriver-App: è un dirver specifico per il testing di applicazioni Web Mobile, la classe driver specifica per le Web View è SelendroidWebDriver mentre quella per le applicazioni native è SelendroidNativeDriver.
- Selendroid-Standalone: gestisce i diversi dispositivi Android installando il server e l'applicazione sotto test. La principale classe driver è SelendroidStandaloneDriver che agisce da proxy tra selendroid-client e selendroid-server e si occupa di avviare l'emulatore se necessario, di creare la versione specifica del selendroid-server per l'applicazione sotto test e di installare il tutto sul device. Dopo l'inizializzazione della sessione tutte le richieste e le risposte sono inoltrate dal server al device e viceversa.

Il componente deputato al recording prende il nome di Inspector e non è altro che una webapp inclusa nello standalone server. È accessibile dal browser e consente sia di ispezionare la GUI che di interagire col device.

Le principali caratteristiche sono:

- Esplorazione della gerarchia della GUI
- Visualizzazone delle proprietà di ciascun elemento grafico
- Produzione di screenshot
- Registrazione dei click
- Visualizzazione del codice html di una web view
- XPath helper

L'unico evento in grado di essere registrato è il click su una replica della GUI presentata sul browser. Selendroid costruisce per ogni activity una struttura dati ad albero organizzata in modo che alle coordinate di ciascun click corrisponda un widget e che il corrispondente evento

sia invocato. Limitare la registrazione al solo evento di click è molto riduttivo e rende questo tool molto più utile come inspector che come recorder.

Altro aspetto negativo è l'impossibilità di esportare il codice java prodotto dalla registrazione che resta visibile solo nella web page.

Selendroid, essendo derivato da Selenium, ha alle spalle una community molto attiva anche se è evidente che i progetti per l'ispezione e il recording sono messi in secondo piano e spesso non aggiornati.

2.3.3 Testdroid

Testdroid recorder è un componente del framework Testdroid rilasciato come plugin per Eclipse, lo svantaggio di essere dipendente dall'IDE è comunque bilanciato dal fatto che Eclipse è attualmente uno degli IDE più diffusi tra gli sviluppatori nonché quello adottato ufficialmente da Google per lo sviluppo Android fino al 2014.

Questo recording tool è più evoluto rispetto a quelli analizzati fino ad ora infatti è a pagamento ed ai fini del confronto è stata utilizzata una licenza limitata per la valutazione del prodotto. Il principale vantaggio di questo tool rispetto agli altri è la produzione di script utilizzando il potente framework ExtSolo di Testdroid basato su Robotium che consente un alto livello di astrazione tra dispositivi ed emulatori di diverso tipo e versioni differenti di Android.

I test case possono essere registrati sia a partire dal sorgente che dall'apk dell'applicazione.

Tutto ciò senza la necessità di effettuare il rooting del device e quindi garantendo la compatibilità del recorder con tutti i device a disposizione.

L'invasività è nulla in quanto non è necessario importare librerie o modificare il progetto per inizializzare la registrazione, questo garantisce la massima fedeltà nei risultati dei test effettuati.

Per catturare gli eventi di click sulla GUI ed identificare il widget sul quale sono stati invocati Testdroid utilizza la View Hierarchy di Android. Tali eventi vengono poi memorizzati e tradotti al volo in altrettante linee di codice java che saranno inserite negli script di output. la possibilità di catalogare gli eventi a partire dai widgwt anzichè delle coordinate garantisce la riutizzabilità dei test prodotti anche su device con differenti caratteristiche.

L'integrazione con l'IDE garantisce innanzitutto la semplicità di utilizzo del tool ma rende

trasparenti tutte le operazioni di configurazione e creazione del progetto di test, in particolare se si desidera testare da apk è possibile sfruttare tutte le potenzialità di Eclipse che si occupa di disinstallare, firmare con la chiave di debug e re installare il progetto di test e l'applicazione sul device.

Dal punto di vista strettamente tecnico Testdroid è in grado di catturare la maggior parte delle gestures disponibili : drag&drop, swipe, pinch zoom, multipath drag ed anche la pressione di tasti fisici. è possibile modificare la lingua per testare la localizzazione utilizzando gli stessi script e sono recepiti anche i cambi di orientamento.

Nel corso della registrazione è possibile specificare gli assert desiderati che saranno inclusi nello script prodotto, per farlo sono necessari pochi click.

Un'altra interessante feature è la cattura di screenshot in punti prestabiliti del test anche se a tal scopo è necessario invadere l'applicazione aggiungendo al file di configurazione la permission WRITE_EXTERNAL_STORAGE.

Per coprire il limitato numero di operazioni che non è possibile registrare o che sono impossibili da effettuare manualmente le istruzioni prodotte possono essere modificate sia durante la registrazione stessa che in seguito aggiungendo righe di codice ai test prodotti. Una caratteristica molto interessante è quella degli "sleep", ritardi nell'esecuzione dei test introdotti per corrispondere esattamente a quelli dell'utente durante l'interazione con l'applicazione.

In caso di attese troppo lunghe questi valori possono essere anche adattati alle esigenze.

Uno dei limiti di Testdroid è l'utilizzo delle web view, il tool è in grado di riconosce gli elementi html e gli eventi su di essi ma necessita di caricare delle librerie javascript specifiche che introducono un certo delay tra il caricamento e l'avvio della registrazione.

Testdroid nella sua versione a pagamento offre anche la possibilità di lanciare nel cloud gli script generati e di avviare sull'applicazione un crawler che automaticamente esplora l'applicazione senza neanche la necessità di eseguire manualmente i test.

2.3.4 Robotium

Robotium è uno dei principali framework per l’automazione del testing Android, fu sviluppato nel 2010 da Renas Reda specialista nella test automation.

Nel tempo si è ritagliato un'ampia fetta di mercato nel settore dell'Android QA e può contare su una vivace comunità.

Oltre al popolare framework la Robotium Tech propone il tool commercial Robotium Recorder che provvede a registrare e tradurre in script java i test manuali effettuati su emulatori e device.

Il recorder si integra con i principali IDE, Eclipse ed Android Studio ed è disponibile come plugin con una breve licenza di valutazione. Oltre a garantire i consueti benefici di un tool di recording RobotiumRecorder garantisce l'utilizzo più efficiente possibile del Framework e la sincronizzazione con l'ultima versione disponibile. Robotium Recorder secondo i suoi creatori è in grado di ridurre del 90% il tempo necessario alla stesura di test script.

Il recorder offre le seguenti funzionalità:

- Click to Assert: consente di inserire asserzioni all'interno dello script generato con un semplice click;
- Registrazione possibile sia da apk che da codice sorgente;
- Registrazione di applicazioni native o ibride;
- Riconoscimento dei resource ID dei widget.

2.3.5 Ranorex

Ranorex è un toolbox per l'Automated Software Testing che supporta molteplici ambienti, device e tecnologie.

Favorisce ed accelera la creazione di test case per applicazioni Desktop, Web e Mobile.

Il punto di forza è senza dubbio l'accuratissima tecnica per il riconoscimento degli oggetti che copre la maggior parte delle tecnologie UI, un presupposto necessario per realizzare test robusti e riutilizzabili. Inoltre il Ranorex Object Repository effettua un mapping tra gli ID tecnici individuati e dei nomi logici in modo da minimizzare ulteriormente gli sforzi di manutenzione dei test in seguito a modifiche della GUI.

I framework e le tecnologie UI elencate di seguito sono attualmente supportate:

.NET, WinForms, WPF, Win32, VB6, Java, Qt, Delphi, PowerBuilder, SAPgui, MFC, ActiveX, Microsoft Visual FoxPro, Microsoft Office GUI, Microsoft Access, Microsoft Dynamics AX, Microsoft Dynamics CRM, Microsoft Dynamics NAV, Air, Infragistics,

DevExpress, ComponentOne, Janus, Syncfusion, Telerik, SkinSoft and many more... Desktop Test Automation Tools, HTML, HTML5, JavaScript, Ajax, Silverlight, Flash, Flex, Air, ASP.NET, Google Web Toolkit, YUI library, Ext JS, Ext.Net, Java applet, jQuery, Sencha GXT, Dhtmlx, Sweetdev Ria, MochiKit, MooTools, Pyjs, Rico (Ajax), SmartClient, midori JavaScript Framework, Echo (framework), script.aculo.us, Enyo, ZK (framework), naturalmente cross-browser testing for Internet Explorer, Firefox, Chrome and Safari.

Ranorex viene rilasciato con un IDE proprietario che tra le varie feature include un Recording tool cross platform.

Focalizzando sul mobile Ranorex è in grado di automatizzare il testing di applicazioni iOS, Android e Windows 8 in tutte le più recenti versioni.

Di seguito una lista delle principali caratteristiche:

- Record on Real Mobile Devices
- Click & Go Automation Editor: le azioni registrate sono visualizzate live in una tabella in cui è possibile modificare i valori ed aggiungerne di nuovi. È possibile gestire gli oggetti dell'applicazione all'interno del Ranorex Object Repository per ridurre gli sforzi di manutenzione e consentire l'utilizzo dei test cross device.
- Easy mobile setup: è sufficiente strumenare l'applicazione mediante un wizard e collegare il device desiderato via USB o WiFi
- Non è necessario effettuare il rooting o jailbreaking del device, basta semplicemente avviare l'emulatore o collegare il dispositivo per cominciare.
- Esecuzione dei test automatizzati cross device: il riconoscimento delle azioni è indipendente da localizzazione e risoluzione dunque è possibile effettuare test su ogni tipo di device.
- Validation e Verification: nel corso della registrazione, grazie al meccanismo di riconoscimento degli oggetti, è possibile introdurre dei check su un gran numero di attributi.
- Data-Driven Testing: è possibile trasformare i propri test in modo da seguire l'approccio Data Driven utilizzando uno dei seguenti data connector: SQL, CSV, Excel, Simple Table. L'associazione è effettuata mediante dei placeholder definiti nei moduli e nel

repository.

- È possibile combinare il testing di applicazioni desktop, web e mobile.
- Gestione semplificata dei Test Case: l'IDE di Ranorex consente di creare, modificare ed organizzare i test script prodotti.
- Report sull'esecuzione dei test e sui bug rilevati: I risultati dell'esecuzione di ogni test suite vengono mostrati in forma grafica live fornendo un veloce colpo d'occhio sull'andamento. In aggiunta possono essere prodotti report degli errori corredati di screenshot e dettagli utili a determinarne la causa eventualmente visualizzando l'azione precisa che li ha scatenati.

Per preparare, il device al recording è necessario abilitare nelle impostazioni le opzioni sviluppatore ed attivare il debug, ciò garantisce una maggiore stabilità nella connessione e rende possibile l'accesso ad un set di funzionalità avanzate.

Una volta collegato via USB per poter associare il dispositivo è necessario installare su di esso una service application che gestisce tutte le applicazioni sotto test e che comunica col server per scambiare i risultati. Nel caso non fosse possibile collegare il dispositivo via usb è possibile scaricare l'agent online o mediante un QR code.

Stesso procedimento deve essere seguito per collegare un emulatore.

Una volta riconosciuto il device l'applicazione deve essere strumentata ed installata, di queste operazioni si occupa automaticamente il tool quando si avvia un'operazione di recording.

Durante la fase di Instrumentation un wizard consente di specificare alcune opzioni avanzate come ad esempio la firma dell'apk, la "Full image comparison" e la "tree semplification".

Trattandosi di un tool commerciale la documentazione non specifica le modalità del recording e non è possibile valutare le effettive modifiche apportate all'applicazione, tuttavia viene raccomandato di non rilasciare in ambiente di produzione l'apk generata in questa fase in quanto alcune permission sono aggiunte al file manifest, molto probabilmente quella per l'accesso ad Internet e la comunicazione col server e quella per il prelievo degli screenshot.

La pressione del pulsante record avvia la registrazione vera e propria durante la quale sulla console di Ranorex una tabella viene popolata live con una panoramica degli step registrati con

dettagli sull'elemento grafico coinvolto e sul tipo di evento invocato.

Il manuale utente suggerisce una serie di accortezze da osservare per ottenere la massima fedeltà nel replay.

La live table consente di aggiungere e modificare al volo le azioni ed i parametri ad esse associati nonché di richiedere uno step di validazione.

Come in un registratore basta premere stop per interrompere la sessione di test ed avviarne il replay.

2.3.6 MonkeyTalk

MonkeyTalk è un altro dei più diffusi strumenti per il testing di applicazioni mobile prodotto dalla Gorilla Logic, può essere utilizzato sia per applicazioni Android che iOS e consente di automatizzare sia semplici smoke test che test suite più complesse anche in modalità Data Driven.

Sono disponibili una versione professionale ed una comunitaria open-source in cui è incluso anche il recorder.

Il Record&Playback è cross platform e con un meccanismo intuitivo rende chiunque in grado di registrare rapidamente test funzionali sia da emulatore che da device senza necessita di rooting o jailbreaking. il linguaggio di output degli script è human readable, può essere riutilizzato tra piattaforme differenti e convertito in script Java e Javascript.

Supporta il riconoscimento delle principali gesture come swipe, drag, move e perfino il disegno a mano libera.

Il recorder è integrato in un ambiente desktop che include funzionalità di salvataggio, modifica e replay dei casi di test prodotti semplicemente collegando il device via USB.

Full touch and gesture support: MonkeyTalk supporta le interazioni basate su touch e gesture è in grado di registrare swipe, drag, move e anche il disegno a mano libera.

La piattaforma è rilasciata come applicazione desktop che include le funzionalità di gestione, creazione, modifica ed esecuzione di test automatizzati oltre all'utilissimo strumento di recording. Gli script editor ed il generatore di report agevolano ed accelerano ulteriormente la produzione e grazie all'interfaccia intuitiva riducono la curva di apprendimento.

MonkeyTalk consente di utilizzare sia il linguaggio proprietario di tipo keyword driven che le API Java e JavaScritp. Il linguaggio di MonkeyTalk è di tipo what-you-see-is-what-you-say cioè ricorda il linguaggio naturale ed è comodo da utilizzare anche per chi non ha competenze specifiche di sviluppo.

La struttura di questo linguaggio è la seguente:

```
ComponentType MonkeyId Action Args[] Modifiers[]
```

Il ComponentType può essere interpretato come il nome di una classe di cui sono presenti tante istanze quanti sono gli oggetti corrispondenti nell'applicazione, il MonkeyId è un identificativo che consente di selezionale l'istanza corretta su cui eseguire l'Action cioè uno dei possibili metodi definiti. I componenti godono inoltre dell'ereditarietà e sono tutti discendenti della classe View di Android che può essere utilizzata come wildcard.

Gli Arguments sono obbligatori o meno in base al Component ed all'Action specificati, sono tutti inseriti come stringhe separate da spazio di cui viene poi opportunamente effettuato il casting secondo la necessità. Se un argomento è obbligatorio ma non viene inserito nella dichiarazione viene utilizzato un valore di default per garantire la prosecuzione del test.

I Modifiers sono opzionali, possono essere espressi come lista di coppie chiave valore. Di seguito un esempio di tre system-defined modifier:

```
%timeout=x - Millisecondi di ripetizione di un comando prima del Timeout
%thinktime=y – Delay da anteporre all'azione espresso in millisecondi
%retrydelay=z – Delay tra tentativi successivi espresso in millisecondi
```

Per i più esperti sono inoltre a disposizioni API Java e Javascript che è possibile integrare con altri framework esistenti. Script prodotti con differenti linguaggi possono essere invocati l'uno dall'altro per la massima interoperabilità.

MonkeyTalk è dotato di un set completo di task Ant per eseguire i test e gestire i risultati anche da linea di comando o da Java. Ciò garantisce la possibilità di invocare questi servizi dall'interno di altri processi o in un ciclo di continuous integration.

Durante la registrazione dei casi di test MoneyTalk utilizza un completo meccanismo di

riconoscimento degli oggetti che consente di effettuare verifiche sulle loro proprietà nel corso del test. E' possibile in tal modo creare asserzioni molto specifiche non solo sulla presenza di widget ma anche sulle loro caratteristiche grafiche e non.

Gli script prodotti da MonkeyTalk possono essere riutilizzati in modalità data driven sia per quanto riguarda gli input che per la validazone degli output, è sufficiente compilare un file csv per ottenere una vasta copertura a partire da un singolo script.

I report prodotti in formato html forniscono le azioni che step by step hanno condotto ad un errore corredate di screenshot.Il formato è compatibile con lo standard xUnit garantendo la possibilità di integrazione con i principali management tools.

Per procedere con l'utilizzo di MonkeyTalk è necessario metter su l'ambiente, innanzitutto bisogna installare l'IDE che funge da intermediario per le informazioni inviate dal device e integra le principali funzionalità di editing e recording.

Dopodiché bisogna installare sul device l'applicazione Agent aggiungendo l'apposito jar al codice sorgente dell'applicazione sotto test e modificando il file manifest in modo da garantire la presenza delle permission INTERNET e GET_TASK.

Come il primo tool analizzato anche MonkeyTalk utilizza AspectJ per l'intercettazione esaustiva degli eventi.

Questo approccio è molto efficace ma va a discapito dell'accuratezza dei test in quanto bisogna intervenire abbastanza invasivamente sul sorgente dell'applicazione.

2.4 Tecnologie per il recording

Purtroppo non tutti i tool utilizzati sono opensource e quindi per alcuni di essi non è stato possibile analizzare le tecnologie alla base del recording. Ciononostante è stato possibile individuare alcune caratteristiche comuni. Tutti i recorder hanno un'architettura di tipo client/server il componente server spesso è invisibile all'utente e integrato nell'IDE da cui si avvia e monitora la registrazione. Il componente client invece è talvolta incluso nell'AUT sotto forma di librerie aggiuntive che gestiscono la comunicazione con il server altre volte invece è una vera e propria applicazione agent che deve essere installata sul dispositivo per operare sulle applicazioni. Fatta eccezione per Selendroid Recorder che utilizza un approccio molto

2.5 Confronto tra recording tool

Nel paragrafo successivo saranno mostrati e commentati i risultati ottenuti dal confronto qualitativo dei recording tool presentati. Allo scopo di fornire una panoramica quanto più chiara possibile delle potenzialità e degli svantaggi di ciascuno sono stati individuati un insieme di parametri rispetto ai quali valutarne le prestazioni. La selezione di questi parametri è stata dettata, oltre che da benchmark specifici disponibili online, anche dall'esperienza maturata utilizzandoli e confrontandosi con i principali ostacoli derivanti dall'utilizzo di software di questo tipo.

Una volta individuate le aree di confronto, per ciascuno strumento è stata popolata una checklist, dapprima utilizzando le documentazioni ufficiali ed i wiki online e poi sottoponendo a registrazione un'applicazione di test per verificare la corretta intercettazione degli input esercitati.

La valutazione dei parametri generali come l'usabilità, la qualità della documentazione e la semplicità di installazione è stata effettuata calcolando il tempo necessario alla messa in opera di ciascuno strumento, la necessità di conoscenze tecniche specifiche e l'effettiva esaustività delle guide messe a disposizione.

Questa fase di confronto è stata utile ai fini della selezione di un recorder da utilizzare per il caso di studio presentato nel capitolo successivo.

2.6 Tabella dei risultati commentata

Nei paragrafi successivi sono stati riportati, in forma di checklist tabellare, i risultati ottenuti dal confronto tra i recording tools nelle modalità descritte. I parametri valutati sono stati suddivisi in macro aree e discussi in termini di vantaggi e svantaggi.

2.6.1 Generalità

L'introduzione di un nuovo tool nel ciclo di sviluppo di un software non è un'operazione indolore, bisogna tenere in conto numerosi fattori tra cui la disponibilità dei prerequisiti tecnici, l'integrazione con gli strumenti eventualmente già in uso, il costo delle licenze e la disponibilità

di documentazione ed assistenza. Dal punto di vista economico uno strumento opensource è sicuramente preferibile per abbattere i costi già alti relativi al testing, anche se i tool commerciali hanno maggiori garanzie in termini di qualità ed assistenza. Un approccio valido è quello di MonkeyTalk che mette a disposizione una versione community ed una pro con features aggiuntive. Altri fattori da tenere presenti sono la dipendenza o meno da uno specifico IDE ed eventuali vincoli sulla piattaforma e sulle tecnologie perché potrebbero non essere compatibili con l'architettura già esistente. Nel nostro contesto universitario, considerando i vari fattori, la preferenza ricade su Testdroid, Robotium e MonkeyTalk. Pur avendo una documentazione eccellente e una solida community Ranorex viene escluso sia per il problema della licenza che per il vincolo alla piattaforma Windows e all'intero toolkit proprietario.

	BOT BOT	SELENDROID	TESTDROID	ROBOTIUM	RANOREX	MONKEY TALK
License	Open source	Open source	Share Ware	Share Ware	Closed source	Open Source
Doc&Wiki	Sufficiente	Insufficiente	Buona	Buona	Ottima	Ottima
Usability	Sufficiente	Insufficiente	Ottima	Ottima	Buona	Buona
Supported SO	Tutti	Tutti	Tutti	Tutti	Windows	Tutti
Requirements	Apache Ant, Aspect J	Apache Maven	Eclipse	IDE	-	Aspect J
IDE	No	Eclipse	Eclipse	Eclipse, Android Studio	Ranorex Studio	Monkey Talk IDE

2.6.2 Features

In questo gruppo sono state incluse tutte le caratteristiche desiderabili ma non necessarie offerte dai vari software. La possibilità di ispezionare la GUI, ad esempio, si rivela molto utile nel caso in cui i test prodotti vogliano essere integrati a mano mentre la produzione di screenshot è un

valore aggiunto quando bisogna individuare la causa di un test fallito.

Tutti i recorder esaminati, essendo integrati in piattaforme più ampie, offrono la possibilità di rieseguire i test registrati e la maggior parte operano sia sul sorgente che sull'eseguibile.

Un parametro particolarmente importante ai fini del nostro confronto è la necessità di modificare l'applicazione ai fini del recording.

Fermo restando che per i tool closed source non è stato possibile valutare con accuratezza questo parametro, quanto riportato nelle varie documentazioni lascia presupporre che interventi anche minimi siano sempre necessari e che non è consigliabile rilasciare l'applicazione utilizzata per il test in ambienti di produzione. AspectJ è molto efficace per il tracking degli eventi ma necessita la conversione dell'intero progetto e l'aggiunta di classi e file di configurazione, recorder meno invasivi richiedono comunque l'aggiunta di "permission" al file manifest.

	BOT BOT	SELENDROID	TESTDROID	ROBOTIUM	RANOREX	MONKEY TALK
Inspection	No	Si	No	No	No	Si
Screenshots	No	Si	Si	Si	Si	Si
Replay	Si	Si	Si	Si	Si	Si
Invasività	Alta	Media	Bassa	Bassa	Bassa	Alta
Source/APK	Both	Source	Both	Both	Both	Source
Ability to test multiple apps		No	Si	Si	Si	

2.6.3 Widget

Le API Android nelle differenti versioni offrono un gran numero di componenti o widget sui

quali è possibile definire una serie di event handler che il recorder deve intercettare e tradurre in codice. Le prestazioni dei tool sui vari widget sono direttamente dipendenti dagli eventuali limiti sull'API Level e sulle potenzialità del framework utilizzato per il replay. Componenti complessi come le seek bar che richiedono gesti di trascinamento vengono registrati e tradotti in modi diversi in base all'espressività delle API. Selendroid non essendo un tool di recording puro riesce a recepire solo il click, i tool basati su AspectJ sono potenzialmente estendibili a qualunque evento purché se ne definisca l'interceptor. I tool a pagamento si rivelano i più efficaci nell'ascolto dei componenti.

A questo gruppo di variabili di confronto appartiene anche la modalità di identificazione degli oggetti, utilizzare l'R-Id dell'applicazione è sicuramente il metodo più preciso ma comporta uno stretto accoppiamento del codice di test con quello dell'applicazione, stesso ragionamento per l'identificazione effettuata in base al testo delle label che non sempre è efficace ed è risultata spesso un'alternativa alla mancata identificazione dell'oggetto. Ranorex e MonkeyTalk utilizzano un mapping tra id nativi e custom che consente di disaccoppiare il test in caso di modifiche all'applicazione.

	BOT BOT	SELENDROID	TESTDROID	ROBOTIUM	RANOREX	MONKEY TALK
Object identification	R-id	Coordinate custom id	Coordinate R-id Text label	Coordinate R-id Text label View Class Identifier	Custom static objects structure	Custom Monkey ID
Tabs	-	-	-	-	-	-
Lists	X	X		X	X	X
Grid	-	-	-	-	-	-

Lists						
Scrolling			X	X	X	X
Spinners	X	X	X	X	X	X
Buttons	X	X	X	X	X	X
Text Fields	X	X	X	X	X	X
Seek Bars	-	-	X	X	X	X
Switches	X	X	X	X	X	X
Dialogs	X	X	X	X	X	X
Pickers	-	X	X	X	X	X

2.6.4 Input detection

Anche per quanto riguarda i tipi di eventi ascoltabili i tool hanno manifestato comportamenti differenti. I dispositivi mobili, specialmente quelli touchscreen, hanno un gran numero di interazioni possibili con l'utente, non tutte facilmente deducibili e registrabili da un tool automatico. La checklist di seguito sintetizza il comportamento di ciascuno strumento nei confronti delle principali gesture applicate.

To e	BOT BOT	SELENDROID	TESTDROID	ROBOTIUM	RANOREX	MONKEY TALK
Touch	X	X	X	X	X	X
Long press				X		
Swipe/Scroll			X	X		X
Double touch						

Pinch			X		X	X
Text input	X		X	X	X	X
Hardware Buttons			X	X		X
Landscape & portrait			X			

2.6.5 Device connectivity

Uno dei confronti più significativi è quello riguardo le modalità di collegamento tra il tool di recording ed il device (o emulatore) perché innanzitutto modalità complesse di collegamento e riconoscimento richiedono conoscenze tecniche e quindi tester che le posseggano. Inoltre le modalità di collegamento influenzano direttamente la possibilità di ascoltare più dispositivi contemporaneamente. Questo parametro è incluso nel discorso dell'invasività in qunto in modalità USB è necessario avviare il dispositivo in modalità debug mentre nel caso della connessione LAN bisogna garantire all'applicazione la permission INTERNET e dotare il device di una connessione Wi-Fi in modo che si trovi sulla stessa sottorete del server. Per entrambe le modalità di connessione è indifferente la disponibilità di un real device o di un emulatore.

	BOT BOT	SELENDROID	TESTDROID	ROBOTIUM	RANOREX	MONKEY TALK
Lan	Si	No	No	Si	Si	Si
USB	No	Si	Si	Si	Si	Si (Thetering)
Remote		No			Si	
Emulatore	Si	Si	Si	Si	Si	Si

2.6.6 Output

Ai fini dello studio che si intende presentare, questo gruppo di parametri di confronto è sicuramente il più importante, in quanto determina l'effettiva possibilità di integrazione dell'output di registrazione in un processo di testing strutturato che eventualmente preveda l'ausilio anche di altri tool. Il framework utilizzato per la traduzione in codice dei test è la caratteristica principale, script prodotti con un framework opensource come Robotium sono maggiormente riutilizzabili rispetto a quelli commerciali. I tool commerciali offrono tutti la possibilità di editing live dei test registrati, in tal caso viene prodotto un output intermedio in linguaggio user friendly che specifica componente target, metodo e parametri dell'istruzione registrata.

	BOT BOT	SELENDROID	TESTDROID	ROBOTIUM	RANOREX	MONKEY TALK
Replay Framework	Robotium, Selenium Nativedriver, TestNG	Selenium API	JUnit + extSolo	JUnit+ Robotium solo	Proprietario	Proprietario
Plain Output	Csv	No	No	No	Xml	Mt
Output content	action, R id, parameter	XPath in custom tree	r id, azione parametri	r id, azione parametric	R.Id, class, path, time, labels	ComponentType MonkeyId Action Args… Modifiers…
Replay language	Java	Java, Python, Ruby	Java	Java	C#, VBNet	Javascript

Capitolo 3 : Manual testing vs Automatic testing

Nel presente capitolo sarà presentato un esperimento condotto al fine di confrontare, in termini di efficienza ed efficacia, due tecniche differenti per la generazione di test case: quella manuale computer aided e quella completamente automatica. Lo studio effettuato ha lo scopo di discutere vantaggi e svantaggi di ciascuna tecnica, individuare combinazioni favorevoli di esse ed evidenziare la relazione intercorrente tra le caratteristiche dell'applicazione oggetto di test ed il successo nella copertura ottenuto dalle differenti tecniche analizzate. Il numero esiguo di esperimenti condotti non rende possibile trarre conclusioni generali sull'argomento, tuttavia l'analisi di quali porzioni di codice sono stati esaminati o meno da ciascuna tecnica offre una panoramica sui limiti insiti in ciascuna strategia.

3.1 Tecniche per la generazione di test case

Come discusso nei capitoli precedenti la test automation è fondamentale per accelerare il ciclo di sviluppo di un software mobile, a tal scopo una parte delle risorse disponibili deve essere investita per la produzione dei test script necessari.

Anche per la stesura dei test, come per la loro esecuzione, è possibile distinguere tra tecniche manuali ed automatiche: nel primo caso i test devono essere progettati ed eseguiti manualmente da un operatore, nel secondo caso la produzione dei test è affidata ad un crawler che ispeziona l'applicazione e la alimenta con input casuali traducendo i percorsi esplorati in altrettanti test script.

Il confronto oggetto di studio è stato effettuato tra due tecniche di tipo manuale, approccio Black Box e White Box e due tecniche automatiche, ad ispezione sistematica e casuale.

Le tecniche selezionate sono state poi eseguite su quattro applicazioni Android, con caratteristiche differenti, in modo da poter valutare le differenze in termini di tempo impiegato e percentuale di copertura ottenuta.

I quattro metodi analizzati sono stati scelti affinché avessero differenti requisiti in termini di risorse, figure professionali coinvolte e tempo di esecuzione, così da poter incastonare i risultati ottenuti in uno scenario aziendale realistico in cui una serie di trade off tra costi e benefici devono essere accettati.

3.1.1 Testing Manuale Assistito Black Box

In questa modalità l'applicazione, installata su un device reale o un emulatore, deve essere testata da un utente umano le cui azioni sono registrate live da un apposito tool e tradotte in linguaggio di scripting. Il tester designato, per operare in modalità black box, non deve essere a conoscenza dei dettagli implementativi dell'applicazione e, pur avendo chiare le funzionalità dell'applicazione oggetto di test, non deve essere preventivamente istruito su tutti i possibili casi d'uso in modo da rendere la sua interazione con l'interfaccia quanto più spontanea possibile.

L'ausilio del tool di recording consente di rendere ripetibili e riutilizzabili i test effettuati nonché di renderli disponibili per successive modifiche e per l'analisi della copertura. Questo importante contributo elimina il principale svantaggio insito nella scelta del testing manuale e cioè il costo delle risorse preposte, in quanto il test script viene generato una tantum e rieseguito all'occorrenza senza ulteriori interventi dell'operatore. Il limite dell'utente umano molto spesso risiede nella probabilità di errore durante l'esecuzione degli scenari di test e nella oggettiva impossibilità di esplorare tutti i possibili stati di un'applicazione specialmente nel caso di input con elevata cardinalità. Ciononostante, per un tester umano, è molto più facile rispetto ad una macchina individuare le classi di equivalenza negli insiemi di input e di selezionare i valori potenzialmente critici ed in grado di scatenare validazioni o comportamenti anomali dell'applicazione.

3.1.2 Testing Manuale Assistito White Box

Il White Box è una tecnica di testing in cui è necessario conoscere i dettagli implementativi del sistema per la stesura dei test script in quanto i set di input vengono selezionati in modo da sollecitare tutti i possibili path logici del codice. [3]

Questa modalità di testing differisce dalla precedente per il fatto che il tester esamina il codice sorgente ed i risultati di copertura ottenuti con la tecnica Black Box per integrare i test case e coprire eventuali scenari non esplorati. Per il White Box è necessario che il tester abbia competenze tecniche sufficienti a leggere il codice per dedurre gli input da utilizzare e le azioni da compiere per coprire un particolare blocco. In teoria un approccio simile dovrebbe garantire il 100% della copertura, in pratica, il fatto che i test siano eseguiti manualmente su un emulatore e registrati anziché sviluppati direttamente, filtrano il tipo di input e quindi alcune porzioni di codice restano inesplorate se non sollecitate programmaticamente. Il concreto vantaggio di questa tecnica è che, abbinandola alla precedente, si riesce a massimizzare la copertura in tempi ragionevoli e con un ridotto intervento di personale specializzato.

3.1.3 Automatica mediante ispezione sistematica della GUI [4]

Le tecniche automatiche considerate sono in grado di esplorare la GUI scatenando catene di eventi utente e sistema che consentono di scoprire man mano nuove interfacce e generano ad ogni iterazione un test script che riproduce l'esplorazione effettuata. L'ispezione dell'interfaccia può essere effettuata prevalentemente in due modi: sistematica e random.

Nel caso di ispezione sistematica è ancora possibile utilizzare una strategia di esplorazione breadth first o depth first.

Le tecniche sistematiche dette anche *model learning* operano scatenando gli eventi possibili su tutti gli elementi della GUI individuati e costruiscono un modello del comportamento osservato, questa fase di esplorazione costituisce essa stessa un test di robustezza dell'applicazione finalizzato all'individuazione dei principali crash e che viene poi utilizzato come input per la produzione di uno script automatizzabile. Le tecniche *model learning* effettuano il reverse engineering dell'applicazione man mano che la esplorano utilizzando euristiche per la selezione dei valori di input.

L'esplorazione sistematica non può essere definita completa in quanto non limitata, pertanto i

tempi necessari all'esecuzione sono tali da rendere necessari dei criteri di arresto, ad esempio strategie di path cut che confrontano ciascuna interfaccia raggiunta con quelle già esplorate per interrompere l'ispezione in caso di uguaglianza. Questo approccio ha lo scopo di evitare di ripercorrere sequenze già esplorate ma aumenta la probabilità di lasciare inesplorati alcuni percorsi o di marcare erroneamente un'interfaccia come già ispezionata. Molto spesso infatti la stessa interfaccia manifesta comportamenti differenti in seguito all'immissione di particolari sequenze di input e le funzionalità coinvolte potrebbero non essere raggiunte poiché il crawler si ferma dopo aver rilevato che l'interfaccia è stata già analizzata.

3.1.4 Automatica mediante ispezione casuale della GUI [4]

L'esplorazione casuale contrariamente a quella sistematica seleziona in modo random sia gli eventi da scatenare che i valori di input da utilizzare. Il criterio d'arresto dell'esecuzione è costituito da una soglia, espressa in numero di eventi da scatenare, che determina la fine dell'esecuzione. Le tecniche random si rivelano molto efficaci quando gli input sono molti e difficilmente partizionabili e quando il comportamento dell'applicazione è influenzato dallo stato.

3.2 Obiettivi

Sebbene come detto in precedenza il numero di applicazioni utilizzate per la raccolta dei dati sia troppo piccolo per garantire la validità assoluta dei risultati ottenuti, di seguito sono elencate le principali domande per le quali il presente studio intende proporre una risposta:

Quali sono i limiti e potenzialità di ciascuna tecnica?

In quali fasi del ciclo di vita risultano più utili le tecniche in esame?

È possibile progettare una tecnica che combini efficacemente gli approcci manuale ed automatico per raggiungere un buon compromesso tra efficienza ed efficacia?

È possibile ipotizzare una correlazione tra la copertura ottenibile con le tecniche analizzate e la natura dell'applicazione oggetto di test?

3.3 Parametri di confronto

Al fine di valutare oggettivamente le prestazioni di ciascuna delle tecniche considerate una serie di parametri sono stati valutati per ogni esecuzione.

I parametri di input sono quelli specifici di ciascuna applicazione, necessari per normalizzare i risultati ottenuti:

Linee di codice *#loc*: la più elementare unità di misura per valutare la dimensione e la complessità di un'applicazione. Questo valore fa riferimento esclusivamente alle linee di codice eseguibili.

Numero di input *#i:* numero di text box, check box e select box presenti nell'applicazione ad esclusione dei widget

Dipendenza dagli input d ∈ {molto bassa, bassa, media, alta}: grandezza qualitativa che, tenendo in considerazione il numero di input dell'applicazione, il numero di classi di equivalenza o la cardinalità dell'insieme dei valori ammessi per ciascuno e l'impiego che l'applicazione fa di tali valori, consente di descrivere qualitativamente in che misura i dati forniti in ingresso e la loro sequenza influenzano la logica dell'applicazione e l'opportunità di percorrere dei path all'interno del codice.

I parametri di output sono invece mirati a valutare la tecnica di testing utilizzata in termini di efficienza ed efficacia:

Tempo di esecuzione *t*: tempo impiegato per la generazione dei test script da parte del tool automatico e per l'esecuzione dei test case previsti nell'approccio manuale.

Coverage *%*: percentuale delle linee di codice dell'applicazione che i test generati sono stati in grado di coprire.

3.3.1 Note sul tempo di esecuzione

Non è possibile considerare il tempo impiegato per la generazione dei test, sia manualmente che automaticamente, una metrica oggettiva per valutare l'efficienza della tecnica analizzata.

Nel caso manuale la velocità del tester è variabile da soggetto a soggetto, la generazione può essere rallentata da eventi esterni ma si può considerare pressoché irrilevante l'influenza della potenza di calcolo della macchina su cui sono in esecuzione l'emulatore ed il recorder.

Nel caso automatico, invece, le performance della macchina e la possibilità di parallelizzare l'elaborazione sono parametri che rendono fortemente variabile la durata complessiva della generazione. Il vantaggio derivante dalla parallelizzazione è consistente sopratutto nel caso di ispezione random poiché gli eventi scatenati in ciascuna iterazione sono indipendenti tra loro.

Per la produzione dei risultati che saranno presentati è stata utilizzata una sola macchina con le seguenti caratteristiche:

Modello: *Asus Desktop PC CM6870*

Processore: *Intel Core CPU i5-3330 3GHz*

Memoria: *4 GB*

Tipo di S.O.: *64 bit*

Per ottenere la massima oggettività nel confronto tra le tecniche i tempi di esecuzione calcolati andrebbero normalizzati rispetto alla durata in ore della "giornata lavorativa" dell'esecutore.

Anche in questo caso bisogna distinguere l'approccio manuale da quello automatico.

Consideriamo, ad esempio, una sessione di testing della durata di una settimana a carico di una sola persona: il totale delle ore a disposizione sarà 40. Questo valore è da considerarsi un limite superiore in quanto in una pianificazione reale, oltre alla velocità soggettiva dell'operatore, va sempre considerata una percentuale di tempo che andrà persa per festività, assenze e intervalli di improduttività.

Nelle stesse ipotesi un tool automatico è in grado di offrire un massimo di 168 ore di elaborazione, a velocità molto più elevata. Questa enorme differenza dipende dal fatto che un server può lavorare in modo costante e senza interruzioni anche di notte e nei giorni festivi.

3.3.2 Code coverage

È opportuno spendere qualche ulteriore parola sull'unità di misura dell'efficacia.

La code coverage è una metrica utilizzata nel testing del software e molto spesso associata all'approccio White Box. Offre una visione immediata di quante e quali linee del codice sorgente sono state coinvolte nel testing ed, oltre ad essere indispensabile per l'individuazione

del codice morto, è anche molto utile per evidenziare porzioni di codice che non sono mai sollecitate a causa di malfunzionamenti del software.

L'analisi della copertura, oltre ad indicare quali funzionalità dovranno essere esaminate nei test successivi, ha l'ulteriore beneficio di spingere gli sviluppatori ad effettuare una sorta di code review.

La domanda che sorge spontanea è quale sia la percentuale minima di copertura da raggiungere per considerare sufficiente lo sforzo di testing.

Alcuni considerano il 100% di code coverage il valore auspicabile che garantisce il massimo della qualità, in realtà ogni applicazione è diversa dall'altra e cambiano anche le esigenze di testing. Bisogna sottolineare che la copertura totale è impossibile da raggiungere in quanto, come vedremo in seguito, all'aumentare del numero di righe di codice aumenta fisiologicamente la percentuale di codice morto o non raggiungibile se non esplicitamente sollecitato. Un errore commesso di frequente è investire risorse eccessive per aumentare sempre di più la percentuale di coverage, o per raggiungere una soglia prestabilita, rallentando lo sviluppo e riducendo la produttività del testing. Poiché nello sviluppo software vige la regola dell' 80-20, cioè l'80% delle funzionalità di un'applicazione sono svolte dal 20% del codice, una volta che il core dell'applicazione e le funzionalità più usate sono state coperte da test, gli ulteriori sforzi per aumentare la percentuale spesso non valgono il beneficio di testare aree utilizzate raramente. Basti pensare a tutte le righe di codice dedicate all'*exception handling* che per essere testate esaustivamente richiederebbero decine e decine di negative test apportando benefici minimi alla salute complessiva dell'applicazione.

È importante dunque scegliere una soglia realistica anche di coverage in considerazione delle date di rilascio e delle risorse a disposizione. Un valore comunemente accettato è l'80/85% al momento del rilascio mentre il 20% è una soglia più che accettabile nelle fasi iniziali e intermedie dello sviluppo dove specialmente le funzionalità core devono essere verificate per evitare costi elevati di rework.[11]

3.4 Presentazione dei tool

In questo paragrafo saranno presentati tutti i tool utilizzati per condurre il caso di studio

descritto, sia quelli utilizzati effettivamente per la generazione dei test che quelli ausiliari per la valutazione dell'efficacia di ciascuna tecnica.

3.4.1 Robotium Recorder

Il recording tool utilizzato per il caso di studio è stato selezionato tra quelli esaminati nel precedente capitolo in base a considerazioni di compatibilità ed usabilità. Pur trattandosi di una versione trial è stata sufficiente a svolgere il numero necessario di registrazioni, l'integrazione con Eclipse, IDE opensource di enorme diffusione e, fino al 2014, parte integrante dell'Android SDK, ha ridotto la curva di apprendimento ed ha accelerato una serie di operazioni corollarie come l'installazione/disinstallazione delle applicazioni, lo start&stop dell'emulatore, la configurazione di base dei progetti di test.

Infine Robotium è attualmente alla base di un gran numero di framework per l'Android Testing Automation, garantisce un'estrema compatibilità degli script prodotti anche con piattaforme diverse ed è facilmente integrabile in un ciclo di continuous integration.

Dal punto di vista dell'efficacia il recorder di Robotium si è rivelato un po' meno potente dei concorrenti ma comunque più che sufficiente allo scopo. Le debolezze nella registrazione, come i problemi di sincronizzazione o la registrazione disordinata degli eventi sono stati facilmente risolti pur senza un'approfondita conoscenza del framework grazie all'estrema leggibilità del codice ed all'utilissimo corredo di commenti dei test script.

3.4.2 Android GUI Ripper [4][5]

L'Android GUI Ripper è un tool sviluppato dal Dipartimento di Informatica e Sistemistica dell'Università di Napoli Federico II, che effettua l'esplorazione automatica della GUI di un'applicazione Android costruendone un modello ad albero. L'applicazione oggetto di test viene strumentata e inclusa in un progetto JUnit.

Le principali caratteristiche del Ripper sono:

- Esplorazione automatica della GUI
- Costruzione di un modello ad albero
- Configurazione degli eventi e degli input
- Configurazione della strategia di ispezione (in ampiezza e in profondità)

- Impostazione di un criterio di arresto
- Individuazione della sequenza di eventi che causano crash
- Possibilità di sospendere l'esecuzione e di riprenderla successivamente.

L'architettura è composta principalmente da due componenti: il Driver e il Ripper.

Il Driver risiede sulla macchina di test e gestisce l'esecuzione dell'esperimento utilizzando le utilities ADT e ADB per strumentare l'applicazione sotto test con il testing framework, generare il componente Ripper e ad installarlo sul device target, eventualmente utilizzando degli snapshot predefiniti. Ha inoltre il compito di estrarre ed elaborare report relativi ai risultati.

3.4.3 Emma [6]

Emma è un toolkit open source per il calcolo e la rappresentazione della copertura del codice Java.

Le principali caratteristiche:

- Possibilità di strumentare le classi per la coverage sia *offline* (prima che siano caricate) sia *on the fly* (usando un Instrumenting Application Classloader).
- Calcolo della coverage con granularità di *classe*, *metodo*, *linea*, *basic block*.
- Rilevazione delle coperture parziali di una linea di codice (es. cortocircuito nelle condizioni di un costrutto if, righe contenenti più istruzioni).
- Calcolo di metriche aggregate per una rapida valutazione dei risultati.
- Produzione di output report nei seguenti formati: plain text, HTML, XML.
- Possibilità di configurare soglie di copertura oltre le quali le linee di codice devono essere evidenziate.
- Struttura dei file di output tale che sia possibile effettuare il merge dei risultati di differenti sessioni di analisi.

Quest'ultima caratteristica in particolare si è rivelata preziosa nel calcolo della coverage in caso di applicazione combinata di due o più tecniche di test generation.

3.5 Le applicazioni target

Le tecniche di generazione presentate sono state testate su quattro applicazioni selezionate in base ai seguenti criteri:

- la disponibilità del codice sorgente;
- l'effettiva presenza sul market;
- la retro compatibilità fino almeno alla versione 10 delle API Android;
- la presenza ridotta o nulla di accesso ad internet ed interazione con i sensori;
- una grafica essenziale.

Inoltre, come sarà illustrato in seguito, hanno modalità differenti di interazione con l'utente, che si è cercato di rappresentare qualitativamente con l'indice di dipendenza illustrato nei paragrafi precedenti. Per ciascuna applicazione è riportata una tabella che ne elenca i campi di input e per ciascuno il numero di classi di equivalenza dell'insieme dei possibili valori di ingresso utilizzata, assieme allo studio del codice, per determinare l'indice di dipendenza.

3.5.1 Tippy Tipper

Si tratta di una semplice applicazione che consente di calcolare la mancia dovuta su un conto in base alla percentuale desiderata. Tra le principali caratteristiche ci sono, l'inserimento dell'ammontare della fattura attraverso una custom keypad, selezione della percentuale mediante slider o con tre tasti di selezione rapida, arrotondamento, esclusione di eventuali tax-rates.[7]

L'implementazione è molto semplice e include un ridotto numero di classi.

#loc = 999;, #i=8; d=molto basso

Come mostra la tabella seguente la dipendenza della logica di business dai valori inseriti in input è *molto bassa*. I valori inseriti, infatti, vengono esclusivamente letti e talvolta validati, senza influenzare il funzionamento dell'applicazione.

Nome	Bill	Default%	1st%	2nd%	3rd%	Exclude Tax Rate	Tax Rate	Round Type
#	4	1	1	1	1	2	1	2

3.5.2 Simply Do

Simply Do è una leggerissima applicazione Android che consente di memorizzare liste di TODO e task. [8]

#loc = 1281;, #i=7; d=medio

In questo caso la dipendenza dagli input è *medio* in quanto una parte di essi viene utilizzata per determinare il comportamento dell'applicazione come, ad esempio, la selezione del criterio di ordinamento e la volontà di visualizzare o meno una dialog box di conferma.

Nome	ListName	ItemName	Star	Active	ItemSorting	ListSorting	ShowConfirm
#	3	3	2	2	3	2	2

3.5.3 Munch Life

Applicazione che consente di tenere il punteggio di alcuni giochi con due differenti livelli e di lanciare un dado a sei facce in modo casuale. [9]

#loc = 184; #i=4; d=basso

Anche se i parametri di input dell'applicazione sono pochi determinano il funzionamento della stessa.

Nome	Gender	ScreenSleep	ShowVictoryDialog	MaxLevel
#	2	2	2	4

3.5.4 Fill Up

Consente di registrare il consumo di carburante tenendo traccia di ogni rifornimento. Mantiene un database in cui è possibile inserire i km percorsi, l'importo del rifornimento e la quantità di carburante inserito. FillUp è in grado di calcolare i consumi effettivi di più vetture producendo grafici e statistiche per i dati introdotti. [10]

Tra le principali caratteristiche:

- Gestisce dati per più di un veicolo.
- Effettua calcoli sul costo/efficienza dei carburanti e ne produce grafici.
- Produce report e grafici mensili sulla spesa effettuata e la distanza percorsa.
- Tutti i dati sono memorizzati sul device.

- Import/export dei dati su file CSV sulla SD Card.
- Condivisione dei file CSV con applicazioni cloud

#loc = 3807;, #i=16; d=alto

In quest'applicazione il significato degli input è estremamente rilevante, oltre ad essere presenti un gran numero di parametri di configurazione, alcuni dei dati inseriti non vengono soltanto validati e letti ma anche memorizzati per essere usati in calcoli e validazioni successive.

I controlli effettuati oltre che sui dati correnti lavorano anche sui dati precedenti rendendo importante la sequenza di inserimento.

Nome	VehicleName	VehicleTankSize	Odometer	Date	Cost	Amount	FuelTotal	TankFull
#	3	3	4	1	3	3	3	2

Nome	Notes	DataEntry	Units	Currency	RequireCost	DisplayCost	DisplayNote	FontSize
#	3	2	3	3	6	2	2	4

3.6 Risultati

Nella tabella sottostante è possibile visualizzare i risultati ottenuti in termini di copertura per tutti i metodi di generazione utilizzati.

COVERAGE (%)		**#**	**Black Box**	**White Box**	**Systematic Inspection**	**Random Inspection**	**Unione delle tecniche**
Class	**FillUp**	105	90,48	93,33	66,66	80	93,33
	TippyTipper	42	97,62	97,62	90,48	97,62	97,62
	MunchLife	10	100	100	90	90	100
	SimplyDo	46	95,65		95,65	95,65	95,65
Method	**FillUp**	669	80,27	84,45	62,03	76,68	87,59
	TippyTipper	225	81,33	84,44	64,89	85,78	88,89
	MunchLife	28	96,43	100	92,86	92,86	100
	SimplyDo	246	84,55		76,42	86,58	86,99
Block	**FillUp**	18096	79,4	84,82	59,22	72,36	87,47
	TippyTipper	4253	84,79	87,11	56,10	88,31	90,97
	MunchLife	841	79,55	90,49	74,55	87,16	94,65
	SimplyDo	5523	80,82		71,99	85,97	86,09
Line	**FillUp**	3807	79,60	84,88	60,32	74,59	88,18

	TippyTipper	999	83,10	86,33	57,42	89,61	89,71
	MunchLife	184	79,62	91,20	76,96	86,90	94,57
	SimplyDo	1281	79,33	79,33	70,21	84,86	85,10

FillUp

Per questa applicazione entrambe le tecniche manuali hanno superato quelle automatiche con la prevalenza assoluta di quella WB che ha sovrastato di oltre 10 punti percentuale la random. Questa prevalenza così netta può essere motivata considerando che Fill Up è un'applicazione con un gran numero di activity ed un comportamento fortemente influenzato dalla sequenza dei dati inseriti e dalla loro coerenza. All'aumentare della complessità di una condizione e del numero di passi di cui è composta la sequenza necessaria a raggiungerla diminuisce la probabilità che in modo casuale sia percorsa e quindi si riduce l'efficacia del metodo random.

Le cattive prestazioni del testing sistematico sono dovute al criterio di arresto impostato e cioè al blocco dell'esplorazione in presenza di un'interfaccia grafica simile ad una già visitata.

TippyTipper

Su Tippy Tipper tutte le tecniche eccetto quella sistematica hanno offerto una copertura superiore all'80% con una prevalenza non eclatante di quella random. L'applicazione non è dotata di stato, se non per pochissimi parametri di configurazione, ed ha soltanto quattro activity, pertanto l'inserimento di input casuali è stato sufficiente a raggiungere e migliorare le prestazioni del WB. Il ridotto numero di activity, in questo caso, può costituire la motivazione del gap tra la coverage ottenuta con l'esplorazione sistematica e quella delle altre tecniche.

MunchLife

Munch Life è la più piccola tra le applicazioni analizzate, con soltanto due activity. Nonostante la sua semplicità la modalità di testing più efficace è stata quella White Box. Per motivare questo risultato è necessario considerare nel dettaglio il funzionamento dell'applicazione. L'activity principale è dotata di due contatori ciascuno con un pulsante per incrementarlo e decrementarlo, buona parte della logica implementativa è basata sul raggiungimento di limiti superiori o inferiori per questi contatori. Un'esecuzione random dovendo scegliere ad ogni iterazione il widget da sollecitare ha una probabilità molto bassa di effettuare il numero di incrementi consecutivi necessari a raggiungere tali soglie.

SimplyDo

Con Simply Do si ha di nuovo una prevalenza della tecnica random su quelle manuali con uno scarto del 5% ed una differenza infinitesima con la copertura ottenuta dall'unione delle quattro tecniche. L'applicazione è di medie dimensioni ma ha una logica molto lineare: un'activity per operazioni CRUD sulle liste, un'activity per operazioni CRUD sugli item di una lista e i menù contestuali. La ridotta profondità delle sequenze e la quasi totale indipendenza dalla natura dei dati inseriti ha favorito la prevalenza della tecnica automatica rispetto a quella manuale che ha sofferto di errori umani ed omissioni.

Il confronto sulla base dell'efficienza è invece sintetizzato nelle successive due tabelle che mostrano, rispettivamente, i tempi assoluti impiegati per la generazione dei test nelle diverse modalità e gli stessi valori normalizzati rispetto alla durata della giornata lavorativa dell'esecutore (8h nel caso manuale e 24h nel caso automatico).

TIME (h)	Black Box	White Box	Systematic Inspection	Random Inspection
FillUp	4	6	1,22	144
TippyTipper	1	2	0,82	25
MunchLife	1	2	0,25	5
SimplyDo	2	2	1,58	18

EFFICIENZA	Black Box	White Box	Systematic Inspection	Random Inspection
FillUp	0,5	0,75	0,051	6
TippyTipper	0,125	0,25	0,034	1,042
MunchLife	0,125	0,25	0,010	0,208
SimplyDo	0,25	0,25	0,066	0,75

Dal punto di vista del tempo impiegato i test sistematici si sono rivelati i migliori, tuttavia i valori di copertura non possono essere considerati soddisfacenti. Il confronto invece tra i test manuali e quelli automatici casuali è sempre a favore dei test manuali anche se, tenendo in considerazione i valori normalizzati, le differenze si ridimensionano notevolmente e, nel caso di applicazioni di dimensioni molto ridotte come Munch Life, addirittura l'efficienza del metodo Random è maggiore di quella del White Box.

3.7 Analisi del codice non coperto da alcuna tecnica

In questo paragrafo sarà sintetizzata e discussa la natura delle righe non coperte dall'unione delle tecniche, distinguendo tra i seguenti casi:

a) Righe non raggiungibili (codice morto):
   - classi di test,
   - classi non utilizzate,
   - metodi mai invocati (costruttori superflui, implementazioni di interfacce, codice morto).

b) Righe non raggiungibili per limiti degli strumenti di generazione:
   - catch di eccezioni sul formato numerico di un input,
   - ripristino delle activities,
   - listener per gesture particolari.

c) Righe raggiungibili soltanto se sollecitate programmaticamente via unit test ma non raggiungibili utilizzando la GUI:
   - null check,
   - default case nei costrutti switch case,
   - exception handling.

d) Righe per la gestione di stati anomali del dispositivo o eccezioni:
   - gestione di codici di errore,
   - check delle periferiche.

e) Righe per la gestione del ciclo di vita dell'applicazione:
   - update dell'applicazione,
   - upgrade del db.

f) Righe raggiungibili ma non raggiunte per insufficienza/inadeguatezza del test:
   - Costrutti condizionali attivati da specifiche sequenze di input
   - Validazione di valori limite
   - Omissioni

La presenza di righe ancora raggiungibili impedisce di considerare il valore di copertura ottenuto con l'unione di queste quattro tecniche un limite superiore per il testing.

Di seguito sono mostrati esempi dei casi illustrati per ciascuna applicazione.

FillUp

Per FillUp, l'applicazione di maggiori dimensione e complessità il totale delle righe non coperte è 450 pari all'11,82%. Con questi numeri risulta impraticabile discutere sistematicamente riga per riga quanto sfuggito al testing e si è preferito focalizzare l'attenzione sul codice che non è stato esplorato a causa di limiti degli strumenti.

Come sempre un'ampia fetta del codice non testato è costituita da righe non raggiungibili: nel caso di FillUp abbiamo 40 righe di codice morto, 25 per la gestione dei default, 24 null check e numerose altre per la gestione di eccezioni varie.

Di seguito uno dei costrutti che validano gli input rispetto ai valori limite e che sono rimasti spesso inesplorati, sfuggono al test automatico a causa della finitezza del range numerico dal quale sono estratti gli input random ed a quello manuale per imprecisioni del tester. Si tratta di righe facilmente copribili perfezionando l'approccio white box.

```
377     if ((value < 0) || (value > MAX_ODOMETER)) {
378         throw new NumberFormatException("Value out of range.");
379     }
```

Come anche in altri casi successivi, sono spesso presenti controlli riguardo la possibilità di scrivere sul filesystem che potrebbero essere raggiungibili solo programmaticamente. In teoria sarebbero possibili due approcci, modificare le impostazioni dell'emulatore affinché le condizioni siano verificate o modificare lo strumento automatico affinché applichi anche modifiche all'ambiente.

```
362 if (ExternalStorage.isReadable()) {
363     Intent intent = new Intent(this, FileSelectionActivity.class);
364     File root = Environment.getExternalStorageDirectory();
365     File path = Environment.getExternalStoragePublicDirectory(DOWNLOAD_SERVICE);
366     intent.putExtra(FileSelectionActivity.ROOT,root.getAbsolutePath());
367     intent.putExtra(FileSelectionActivity.PATH,path.getAbsolutePath());
368     intent.putExtra(FileSelectionActivity.EXT,".csv");
369     startActivityForResult(intent,CHOOSE_IMPORT_FILE);
370 } else {
371     Utilities.toast(this,getString(R.string.toast_ext_storage_not_rdbl));
372 }
```

```
53  // special case: display update information
54  if (isUpdateFirstStart()) {
55          showUpdateInformation();
56          return;
57  }
```

TippyTipper

Al testing dell'applicazione TippyTipper è sfuggito il 10,29% del codice pari ad un totale di 103 righe non coperte di cui ben 31 non raggiungibili perché appartenenti a classi e metodi mai utilizzati durante il flusso applicativo.

Di seguito alcuni estratti del report di copertura:

- Nel listato successivo la condizione non può essere soddisfatta perché la GUI non consente l'inserimento di valori inferiori ad 1

```
249 if(people < 1)
250     throw new IllegalArgumentException("Number of people cannot be below one.");
```

- Questa condizione, invece, sarebbe stata verificabile con un'opportuna sequenza di azioni

```
152 if(inflated_splitTax != null)
153     inflated_splitTax.setVisibility(View.GONE);
```

MunchLife

Per MunchLife l'unione di tutte le tecniche non ha coperto solo il 5,43% del codice pari a 10 righe di cui:

- 4 sarebbero state raggiungibili sono via unit test,
- 3 sono impiegate per la cattura di una NumberFormatException che risulta irraggiungibile nel caso manuale per il widget di inserimento vincolato dall'emulatore e nel caso automatico perché gli input random sono sempre numeri interi,
- 3 attivate dalla condizione `if(level > max_level)` che è da considerarsi raggiungibile con un opportuno test

```
91                  try
92                  {
93                          max_level = Integer.parseInt(maxlevelPref);
94                  }
95                  catch(NumberFormatException error)
96                  {
97                          Log.e(TAG, "NumberFormatException: " + error.getMessage());
98                          max_level = 10;
99                  }
100
101                 if(level > max_level)
102                 {
103                         level = max_level;
104                         current_level.setText(Integer.toString(level));
105                         total_level.setText(Integer.toString(level + gear_level));
106                 }
107         }
```

```
225                 switch(item.getItemId())
226                 {
227                         case R.id.reset:
    […]
321                                         case 6:
322                                                 rollview.setImageResource(R.drawable.six);
323                                                 return;
324                                         default:
325                                                 return;
326                                 }
327                         default:
328                                 return;
329                 }
330         }
```

```
253                 switch(id)
254                 {
    […]
286                         default:
287                                 return super.onCreateDialog(id);
288                 }
```

SimplyDo

La quota di codice non coperto per Simply Do è del 4,9% cioè 191 righe.

È conveniente escludere dall'analisi i seguenti casi già discussi:

- 95 righe di codice non raggiungibile di cui 80 appartenenti a 2 classi mai utilizzate, 9 per la definizione di default e 6 di generico codice morto.
- 17 righe impiegate per il null check, molto spesso inserito per robustezza ma superfluo all'interno del flusso applicativo,

- 22 righe per il catch di eccezioni e gestione degli errori del database.

I due listati seguenti mostrano alcune righe che riguardano rispettivamente l'upgrade del database e la verifica di variabili d'ambiente relativi al media:

```
61          @Override
62          public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
63          {
64              Log.d(L.TAG, "Got callback to upgrading database from version " +
65                      oldVersion +
66                      " to "
67                      + newVersion);
68          }
```

```
174         if (!Environment.MEDIA_MOUNTED.equals(state))
175         {
176             Toast.makeText(
177                     this,
178                     R.string.restoreToastMountProblem,
179                     Toast.LENGTH_LONG
180                     ).show();
181             return;
182         }
```

```
106         else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state))
107         {
108             // We can only read the media and we need to write
109             Toast t = Toast.makeText(this, R.string.settingsMediaReadOnly,
Toast.LENGTH_LONG);
110             t.show();
111         }
112         else
113         {
114             // Something else is wrong.
115             Toast t = Toast.makeText(this, R.string.settingsMediaNoMount,
Toast.LENGTH_LONG);
116             t.show();
117         }
```

3.8 Considerazioni sulle tecniche

Una volta esclusi i costrutti che non è stato possibile raggiungere è interessante valutare, applicazione per applicazione, cosa è stato ispezionato da una tecnica e tralasciato dall'altra.

Saranno prese in considerazione soltanto le coperture ottenute con gli approcci White Box e

Random poiché includono le altre e sono sufficienti ad evidenziare eventuali limiti degli strumenti utilizzati.

Per avere un colpo d'occhio sull'efficacia delle tecniche, di seguito viene riportata una tabella che evidenzia la differenza percentuale di copertura tra ciascuna tecnica e l'unione, questo dato è interessante per valutare se e quanto l'approccio manuale e quello automatico offrono un contributo significativo alla copertura.

È interessante notare che per Tippy Tipper e Simply Do, le applicazioni per cui il testing Random si è rivelato più efficace, lo scarto rispetto all'unione è infinitesimo, invece per Fill Up e Munch Life che avevano ottenuto risultati migliori con l'approccio White Box lo scarto rispetto all'unione si attesta attorno al 3%. Questo dato ci suggerisce che il metodo White è più soggetto a limiti dipendenti dalla fallibilità del tester e dalle potenzialità dello strumento di recording, mentre il random, nelle condizioni più favorevoli alla sua applicazione, riesce quasi a includere il White Box.

COVERAGE (Δ%) Rispetto all'unione di tutte le tecniche		#	Black Box	White Box	Systematic	Random
Line	**FillUp**	3807	8,58	3,30	27,86	13,59
	TippyTipper	999	6,61	3,38	32,29	0,10
	MunchLife	184	14,95	3,37	17,61	7,66
	SimplyDo	1281	5,77	5,77	14,89	0,23

Per ogni applicazione è stata costruita una lista delle unità di compilazione evidenziando per ciascuna il numero di righe coperte dal WB e dal Random e la loro differenza.

In tal modo è stato possibili rapidamente individuare le porzioni di codice coperte in esclusiva da una delle tecniche e motivare questo comportamento alla luce degli strumenti utilizzati.

FillUp

FillUp è la più grande e più complessa tra le applicazioni esaminate, oltre ad avere un elevato numero di input mantiene memoria degli input inseriti e la loro sequenza determina differenze sostanziali nel comportamento. L'indice di dipendenza è infatti stato valutato come alto. In queste condizioni, come prevedibile, il testing manuale ha avuto in entrambe le modalità risultati migliori rispetto a quello automatico. Lo scarto tra White Box e Random è notevole,

pari circa al 10%, ciononostante per alcune classi la tecnica automatica si è rivelata comunque vincente rispetto a quella manuale.

Nel primo caso preso in considerazione il testing WB non ha coperto le righe relative al ripristino dell'activity ad esempio in seguito a un cambio di orientamento del dispositivo. Alla base della mancata copertura vi è l'arbitrarietà del testing manuale che demanda all'esecutore la decisione se compiere o meno determinate azioni su un'interfaccia.

```
                                                                                    WB  RAND
91      public void onCreate(Bundle savedInstanceState) {
92          super.onCreate(savedInstanceState);                                     Y   Y
93
94          if (savedInstanceState == null) {                                       Y   Y
95
96                  // get parameters from intent
97              Intent intent = getIntent();                                        Y   Y
98              record = (GasRecord)intent.getSerializableExtra(RECORD);            Y   Y
99              current_odometer = intent.getIntExtra(CURRENT_ODOMETER, -1);        Y   Y
100             tank_size = intent.getFloatExtra(TANK_SIZE, 99999999f);             Y   Y
101
102         } else {                                                                Y   Y
103
104                 // restore the saved state
105                 record = (GasRecord)savedInstanceState.getSerializable("record");   N   Y
106                 current_odometer = savedInstanceState.getInt("current_odometer");   N   Y
107                 tank_size = savedInstanceState.getFloat("tank_size");               N   Y
108
109         }
110
111         loadForm();                                                             Y   Y
112     }
```

In questo secondo caso la prevalenza della tecnica random è dovuta ad un limite del tool di recording che non consentiva di proseguire la registrazione del test nel caso si aprissero intent esterni all'applicazione. Nel caso illustrato, infatti, tra le opzioni del menù contestuale è presente l'esportazione e condivisione dei dati che richiede accesso ad applicazioni esterne.

```
                                                                                    WB  RAND
150     public boolean onOptionsItemSelected(MenuItem item) {
151
152             switch (item.getItemId()) {                                         Y   Y
153
154             case R.id.itemShare:
155                 if (ExternalStorage.isWritable()) {                             N   Y
156                     shareReport();                                              N   Y
157                 } else {
158                     Utilities.toast(this,getString(R.string.storage_not_writable));     N   N
159                 }
```

```
160                     return true;                                                    N  Y
161
162             case R.id.itemSettings:
163                     Intent intent = new Intent(this,Settings.class);                Y  Y
164                     startActivity(intent);                                          Y  Y
165                     return true;                                                    Y  Y
166
167             default:
168                     return super.onOptionsItemSelected(item);                       N  N
169             }
170     }
```

Vediamo invece i listati in cui il metodo manuale ha esplorato più righe di quello automatico. Dalle righe seguenti appare evidente che le condizioni non verificate dipendono dall'insufficienza dei dati inseriti, nel caso automatico l'insufficienza non è dovuta alla quantità dei dati ma alla loro sequenza: per attivare il calcolo del consumo di carburante è necessario inserire una sequenza coerente di rifornimenti partendo dalla situazione di serbatoio pieno. Mentre per un essere umano è immediato inserire un set di dati con queste caratteristiche, per il tool automatico che può spaziare solo in un range [0,99] è molto più improbabile e richiederebbe un numero di eventi estremamente elevato.

```
                                                                                    WB RAND
197 private void appendMileageData() {
198
199         float min = Float.MAX_VALUE;                                            Y  Y
200         float max = 0f;                                                         Y  Y
201         float sum = 0f;                                                         Y  Y
202         int count = 0;                                                          Y  Y
203         for (GasRecord record : records) {                                      Y  Y
204         if (!record.hasCalculation()) continue;                                 Y  N
205         if (record.isCalculationHidden()) continue;                             Y  N
206         float mileage = record.getCalculation().getMileage();                   Y  N
207         min = Math.min(min,mileage);                                            Y  N
208         max = Math.max(max,mileage);                                            Y  N
209         sum += mileage;                                                         Y  N
210         count++;                                                                Y  N
211         }
212
213         String label;
214         String value;
215
216         // average
217         label = getString(R.string.stats_label_mileage_avg);                    Y  Y
218         value = "-";                                                            Y  Y
219         if (count > 0) value = String.format(App.getLocale(),                   Y  H
                        "%.2f %s",sum/count,units.getMileageLabel());
220         appendTableRow(new String[]{label,value});                              Y  Y
221
222         // minimum
223         label = getString(R.string.stats_label_mileage_min);                    Y  Y
224         value = "-";                                                            Y  Y
```

```
225         if (count > 0) value = String.format(App.getLocale(),                    Y  H
                        "%.2f %s",min,units.getMileageLabel());
226         appendTableRow(new String[]{label,value});                               Y  Y
227
228         // maximum
229         label = getString(R.string.stats_label_mileage_max);                     Y  Y
230         value = "-";                                                             Y  Y
231         if (count > 0) value = String.format(App.getLocale(),                    Y  H
                         "%.2f %s",max,units.getMileageLabel());
232         appendTableRow(new String[]{label,value});                               Y  Y
233 }
```

TippyTipper

Tippy Tipper è tra le applicazioni considerate quella con il più basso indice di dipendenza dagli input, la tecnica che si è dimostrata più efficiente in questo caso è quella Random anche se con uno scarto veramente minimo rispetto al White Box pari a circa 11 righe. Di seguito il listato di un metodo, presente in due classi differenti che è stato eseguito soltanto con la tecnica random perché relative al caricamento iniziale delle preference in un oggetto di tipo DialogPreference.

```
                                                                          WB    RAND
85    protected void onSetInitialValue(boolean restore, Object defaultValue)
86    {
87      super.onSetInitialValue(restore, defaultValue);                     N     Y
88      if (restore)                                                        N     Y
89      {
90        Try
91        {
92          mValue = shouldPersist() ? getPersistedInt(mDefault) : 0;       N     H
93        }
94        catch(Exception ex)                                               N     N
95        {
96                mValue = mDefault;                                        N     N
97        }
98      }
99      else
100       mValue = (Integer)defaultValue;                                   N     Y
101   }
```

SimplyDo

Simply Do ha di nuovo visto vincitore il metodo Random con uno scarto di circa 5 punti percentuali rispetto al White Box . La natura molto lineare dell'applicazione ha fatto sì che non fosse possibile migliorare la copertura ottenuta passando dall'approccio Black Box a quello White Box.

Il primo listato analizzato riporta dei casi in cui il WB è stato più efficace coprendo le righe

relative alla creazione iniziale del database. Nel caso specifico la differenza è dovuta al fatto che mentre per il test manuale si è partiti da uno stato "vergine" dell'applicazione nel caso automatico per favorire l'esplorazione è stato necessario predisporre manualmente alcuni dati.

		WB	RAND
45	@Override		
46	public void onCreate(SQLiteDatabase db)		
47	{		
48	db.execSQL("CREATE TABLE lists ("	Y	N
49	+ "id INTEGER PRIMARY KEY,"	Y	N
50	+ "label TEXT"	Y	N
51	+ ");");	Y	N
52	db.execSQL("CREATE TABLE items ("	Y	N
53	+ "id INTEGER PRIMARY KEY,"	Y	N
54	+ "list_id INTEGER,"	Y	N
55	+ "label TEXT,"	Y	N
56	+ "active INTEGER,"	Y	N
57	+ "star INTEGER"	Y	N
58	+ ");");	Y	N
59	}		

Le righe di seguito, invece, mostrano la prevelenza del caso random che, eseguendo le operazioni in sequenza casuale è stato in grado di scatenare un'eccezione durante il restore del database, probabilmente perché non esistente o corrotto in una delle iterazioni successive del tool.

		WB	RAND
198	Try		
199	{		
200	// copy new file into place		
201	SettingsActivity.fileCopy(restoreFile.file, dbFile);	Y	Y
202			
203	// delete backup		
204	dbBakFile.delete();	Y	Y
205	}		
206	catch (Exception e)	N	Y
207	{		
208	// put the old database back		
209	dbFile.delete();	N	Y
210	dbBakFile.renameTo(dbFile);	N	Y
211			
212	Log.e(L.TAG, "Failed to copy restore database into place", e);	N	Y
213			
214	Toast.makeText(	N	Y
215	this,		
216	R.string.restoreToastCopyFailed,		
217	Toast.LENGTH_LONG		
218	).show();		
219	return;	N	Y
220	}		

Su Simply Do molte delle righe non coperte dal caso WB sono da imputarsi ad un errore del tester che non ha effettuato l'eliminazione di un item.

MunchLife

Munch Life è la più piccola tra le applicazioni considerate ed ha una dipendenza dai dati classificata come *media.* La metodologia di testing che ha avuto maggior successo è quella WB, tuttavia il fatto che la copertura ottenuta unendo tutte e quattro le tecniche sia maggiore di essa ha dimostrato che l'ispezione random è comunque riuscita a coprire alcune righe sfuggite al testing WB.

Nel listato di seguito c'è un estratto del codice per cui il WB ha nettamente prevalso sul random nell'esplorazione di due blocchi di codice, il primo relativo alla gestione di una NumberFormatException irraggiungibile con il tool automatico che inserisce solo volori numerici interi e il secondo relativo all'inserimento di valori esterni al range [0,100].

		WB	RAND
29	OnPreferenceChangeListener levelListener = new OnPreferenceChangeListener()		
30	{		
31	public boolean onPreferenceChange(Preference pref, Object newValue)		
32	{		
33	int value = 0;	Y	Y
34	Try		
35	{		
36	value = Integer.parseInt(newValue.toString());	Y	Y
37	}		
38	catch(NumberFormatException error)	Y	N
39	{		
40	Log.w(MunchLifeActivity.TAG,	Y	N
41	"NumberFormatException: " + error.getMessage());	Y	N
42	}		
43			
44	if(value > 1 && value <= 100)	Y	Y
45	{		
46	return true;	Y	Y
47	}		
48	Else		
49	{		
50	Toast.makeText(getApplicationContext(),	Y	N
51	R.string.maxlevelError,	Y	N
52	Toast.LENGTH_SHORT).show();	Y	N
53	return false;	Y	N
54	}		
55	}		
56	};		

Il secondo listato, invece, è un esempio di predominanza della tecnica random relativo alle combinazioni del parametro di configurazione *gender* e dell'orientamento del dispositivo che durante il test manuale è stato difficile ottenere a causa di problemi nella visualizzazione dell'immagina sull'emulatore in modalità *portrait*.

```
                                                                                WB  RAND
400       private OnClickListener genderClickListener = new OnClickListener() {  Y   Y
401       @Override
402       public void onClick(View v) {
403       if(devDisplay.getRotation() == 0) {                                     Y   Y
404       if(gender_female == false) {                                            N   Y
405           gender.setImageResource(R.drawable.female_portrait);                N   Y
406       } else {
407           gender.setImageResource(R.drawable.male_portrait);                  N   Y
408       }
409       } else {
410       if(gender_female == false) {                                            Y   Y
411           gender.setImageResource(R.drawable.female_landscape);               Y   Y
412       } else {
413           gender.setImageResource(R.drawable.male_landscape);                 N   Y
414       }
415       }
416       gender_female = gender_female == true ? false : true;                   H   Y
417       }
418       };
```

3.9 Combinazione delle tecniche automatiche e manuali

Di seguito sono presentati in forma tabellare i risultati ottenuti applicando congiuntamente le quattro tecniche analizzate a due a due. I valori di copertura ottenuti sono stati calcolati grazie alle potenzialità di Emma che consente di effettuare il merge delle coverage ottenute in differenti sessioni di test.

Lo scopo di questo approccio è stato individuare una o più combinazioni che consentirebbero di ottenere un buon bilanciamento tra sforzo umano ed automatico al fine di ottenere risultati soddisfacenti sia in termini di efficacia che di efficienza.

È possibile affermare che una combinazione di tecniche è effettivamente utile se consente di ottenere un valore di copertura migliore rispetto a quello più grande tra le due tecniche utilizzate singolarmente. Questo ci permette, infatti, di dedurre che le due tecniche hanno esplorato porzioni diverse del codice, anche se in minima percentuale. Le combinazioni BB+WB e SYST+RAND sono state riportate solo per completezza. Come già sottolineato BB è incluso in WB per costruzione, dunque la loro unione non sarà mai migliorativa rispetto a WB.

Analogamente è possibile considerare gli eventi eseguiti dall'ispezione sistematica un sottoinsieme di quelli eseguiti da quella random con eventuali piccole differenze relative solo alla casualità degli input inseriti.

Con tutte le combinazioni è stato possibile ottenere una copertura maggiore o uguale a quella di ciascuna delle singole tecniche coinvolte.

Inoltre i risultati ottenuti possono essere considerati globalmente soddisfacenti perché tutte le combinazioni garantiscono una copertura superiore all'80%.

COVERAGE (%)		#	Black Box + White Box	Systematic + Random	Black Box + Systematic	White Box + Systematic	Black Box + Random	White Box + Random	Unione delle tecniche
Class	FillUp	105	93,33	81,90	90,48	93,33	90,48	93,33	93,33
	TippyTipper	42	97,62	97,62	97,62	97,62	97,62	97,62	97,62
	MunchLife	10	100	90	100	100	100	100	100
	SimplyDo	46	95,65	95,65	95,65	95,65	95,65	95,65	95,65
Method	FillUp	669	84,45	77,43	83,11	85,50	84,75	87,59	87,59
	TippyTipper	225	84,44	88,89	85,33	86,22	85,78	88,89	85,78
	MunchLife	28	100	92,86	100	100	100	100	100
	SimplyDo	246	84,55	86,58	84,96	84,96	86,99	86,99	86,99
Block	FillUp	18096	84,82	73,75	82,68	85,61	84,35	87,47	87,47
	TippyTipper	4253	87,11	90,92	88,1	89	90,97	90,97	90,97
	MunchLife	841	90,49	87,16	87,28	91,67	92,03	94,65	94,65
	SimplyDo	5523	80,82	85,97	82,80	82,80	86,09	86,09	86,09
Line	FillUp	3807	84,88	76,06	83,54	85,86	85,79	88,18	88,18
	TippyTipper	999	86,33	89,61	87,03	88,15	89,71	89,71	89,71
	MunchLife	184	91,20	86,90	87,93	92,28	92,39	94,57	94,57
	SimplyDo	1281	79,33	84,86	81,45	81,45	85,10	85,10	85,10

Nella tabella seguente sono riportate le differenze tra il valore di copertura massimo ottenuto con le tecniche singole e quello della combinazione.

COVERAGE (Δ%) Rispetto alla migliore tra le tecniche singole	#	Black Box + White Box	Systematic + Random	Black Box + Systematic	White Box + Systematic	Black Box + Random	White Box + Random

Line	**FillUp**	3807	0,00	-8,82	-1,34	0,98	0,91	3,30
	TippyTipper	999	-3,28	0,00	-2,58	-1,46	0,10	0,10
	MunchLife	184	0,00	-4,29	-3,26	1,09	1,20	3,37
	SimplyDo	1281	-5,53	0,00	-3,41	-3,41	0,23	0,23

La tabella successiva, invece, mostra una sintesi dei delta calcolati rispetto alla copertura ottenuta unendo tutte e quattro le tecniche in esame.

COVERAGE (Δ%) Rispetto all'unione delle quattro tecniche		#	**Black Box + White Box**	**Systematic + Random**	**Black Box + Systematic**	**White Box + Systematic**	**Black Box + Random**	**White Box + Random**
Line	**FillUp**	3807	-3,30	-12,12	-4,65	-2,32	-2,39	0,00
	TippyTipper	999	-3,38	-0,10	-2,68	-1,56	0,00	0,00
	MunchLife	184	-3,37	-7,66	-6,63	-2,28	-2,17	0,00
	SimplyDo	1281	-5,77	-0,23	-3,65	-3,65	0,00	0,00

FillUp

Questa applicazione come riportato in precedenza è quella con la più netta predominanza dell'approccio WB rispetto agli altri. Naturalmente tutte le combinazioni che includono WB hanno una copertura maggiore o uguale ad esso, nel caso WB+RAND il guadagno è del 3,3% mentre nel caso WB+SYST è trascurabile. Considerando che la tecnica random è la meno efficiente ha maggiore interesse prendere in considerazioni combinazioni che non la includano.
Il valore più interessante è quello BB+SYST che con una perdita di solo 1,34% rispetto a WB e del 4,65% rispetto all'unione di tutte le tecniche consente di eseguire il test in tempi rapidi e senza l'intervento di una figura developer.

TippyTipper

Per TippyTipper, applicazione priva di stato e con indice di dipendenza dagli input molto basso, la tecnica di maggior successo si è dimostrata quella Random in grado anche di includere i test WB con un delta pari allo 0.1% rispetto all'unione di tutte le tecniche. Tra le varie combinazioni dunque si rivelano efficaci tutte quelle che includono RAND. Ciononostante è interessante osservare che le combinazioni delle tecniche manuali con SYST offrono comunque coperture molto elevate con scarti molto bassi rispetto all'unione di tutte le tecniche.

MunchLife

Il discorso per MunchLife è analogo a quello di FillUp. La tecnica migliore è quella WB tuttavia dal confronto con l'unione emerge che le tecniche automatiche apportano comunque un contributo alla copertura. La combinazione più valida, se si escludono quelle che includono WB è in questo caso BB+RAND con un delta rispetto all'unione di 2,17%.

SimplyDo

Con SimplyDo pur essendo netta la predominanza della componente Random si ottengono risultati soddisfacenti anche con la combinazione BB+SYST (che nel caso specifico è equivalente a WB+SYST) con un delta rispetto all'unione del 3,65%.

Il contributo aggiuntivo dato dalle tecniche manuali è di solo 0,23% anche se lo scarto complessivo inferiore al 6% potrebbe indurre a considerare anche l'utilizzo del solo testing manuale.

3.10 Confronti significativi

Nei paragrafi successivi vengono proposti una serie di confronti tra combinazioni delle tecniche finalizzati a discutere alcuni parametri significativi: l'intervento umano, l'efficienza, la necessità di coinvolgere figure developer.

3.10.1 Uomo vs macchina

La tabella sottostante mostra il confronto tra le coperture ottenute utilizzando esclusivamente tecniche manuali o automatiche. Poiché la tecnica WB include la BB per come è stata realizzata e la tecnica RAND include ragionevolmente buona parte degli eventi esplorati da SYST a parità di stato iniziale le prevalenze non vengono alterate.

COVERAGE (%)		#	Black Box + White Box	Systematic + Random	All
Line	FillUp	3807	84,88	76,06	88,18
	TippyTipper	999	86,33	89,61	89,71
	MunchLife	184	91,20	86,90	94,57
	SimplyDo	1281	79,33	84,86	85,10

FillUp

FillUp è l'unico caso per cui l'unione della tecnica random con quella sistematica è leggeremente migliorativa, probabilmente a causa di lievi differenze nello stato iniziale dell'applicazione prima del test. Come discusso in precedenza, è un'applicazione complessa il cui comportamento è fortemente influenzato dagli input inseriti e dalla loro sequenza. Per tale ragione ha ottenuto una copertura notevolmente maggiore con le tecniche manuali. Il delta di circa tre punti percentuali che le tecniche manuali hanno rispetto all'unione di tutte le tecniche dimostra tuttavia che ci sono porzioni dell'applicazione che il test manuale non è in grado di testare con la stessa accuratezza delle tecniche automatiche.

TippyTipper

TippyTipper è un'applicazione invece molto semplice, l'unico input previsto è l'importo iniziale che viene semplicemente utilizzato dall'applicazione ma non ne condiziona il funzionamento. In questo caso la combinazione che ha avuto maggior successo è stata quella automatica ed il delta con l'unione di tutte le tecniche risulta trascurabile.

MunchLife

Per MunchLife è possibile condurre lo stesso ragionamento fatto per FillUp. Pur essendo un'applicazione di proporzioni minori, ha comportamento influenzato dagli input e pertanto risulta testabile più efficacemente con le tecniche manuali. Anche per questa applicazione il delta rispetto all'unione evidenzia la capacità delle tecniche automatiche di sollecitare parti del codice che il testing manuale non raggiunge.

SimplyDo

SimplyDo mantiene la prevalenza già discussa della componente random nella combinazione di test automatici. L'indice di dipendenza dai dati basso consente al crawler di esercitare quasi completamente tutti gli stessi percorsi di un test manuale. Questa affermazione è dimostrata dal fatto che il delta tra la combinazione automatica e l'unione di tutte le tecniche è infinitesima.

3.10.2 Efficienti vs efficaci

La tabella successiva, invece, mette a confronto il testing ottenuto combinando le tre tecniche più efficienti (BB, WB, SYST) con quella meno efficiente ma globalmente efficace (RAND).

La copertura ottenuta applicando i tre metodi più rapidi è stata confrontata con quella prodotta dall'ispezione random che richiede tempi notevolmente maggiori alla somma dei tre precedenti. È stato già evidenziato come per una certa categoria di applicazioni il metodo automatico random si riveli più efficace, ciononostante la combinazione tra tecniche manuali e sistematiche si rivela spesso più produttiva dell'applicazione della sola ispezione random ed in tempi molto più ragionevoli. Un altro dato interessante è il bassissimo delta che, in tutti i casi, le tre tecniche combinate hanno rispetto al risultato ottenuto unendo i risultati di tutte e quattro le tecniche.

COVERAGE (%)		#	Black Box + White Box + Systematic	Random	All
Line	FillUp	3807	85,86	74,59	88,18
	TippyTipper	999	88,15	87,46	88,76
	MunchLife	184	92,28	86,90	94,57
	SimplyDo	1281	81,45	84,86	85,10

FillUp

FillUp nel confronto tra tecniche singole ha manifestato una nettissima vittoria del testing WB e questa tendenza si conserva nell'unione che lo include. Il miglioramento ottenuto aggiungendo il testing sistematico a quello manuale tuttavia è poco rilevante e comunque non è in grado di avvicinare la copertura ottenuta a quella dell'unione.

TippyTipper

Nel confronto tra tecniche singole TippyTipper risultava testato meglio dalla tecnica Random con uno scarto di circa il 3,5%. La combinazione delle tecniche rapide riesce a migliorare questo valore portandolo quasi a coincidere con quello dell'unione di tutte le tecniche. Ciò indica che l'ispezione sistematica riesce quasi completamente a colmare il gap tra il testing manuale e quello automatico random.

MunchLife

Nel caso di MunchLife l'approccio WB risultava vincente rispetto a quello random di 4,3% mentre la combinazione esaminata riesce a migliorare questo delta portandolo a 5,38%.

Tuttavia il gap rispetto all'unione delle tecniche è del 2,3% e conferma ancora una volta che l'ispezione random spesso riesce a raggiungere porzioni di codice trascurate dalle altre

tecniche. Questi numeri hanno valenza anche maggiore su un'applicazione di dimensioni estremamente modeste.

SimplyDo

SimplyDo è l'unico caso in controtendenza, per cui la tecnica random risulta vincente sull'unione delle altre tre con 3,4 punti percentuali di scarto ed una differenza infinitesima con il massimo. La componente di copertura del testing sistematico non riesce ad integrare i test manuali questo dimostra che il vantaggio del random oltre che nelle modalità dell'esplorazione dipende anche dallo stato dell'applicazione e dalla numerosità degli input forniti. È ipotizzabile migliorare le prestazioni del test sistematico modificando lo stato iniziale di partenza.

3.10.3 Black vs white

L'ultimo confronto effettuato è ancora di tipo tre tecniche contro una. In questo caso vengono messe a confronto le tecniche black box, quelle cioè che non richiedono l'intervento di uno sviluppatore o comunque che non prevedono l'analisi del codice con la tecnica white box.

Nella lettura dei risultati è necessario tenere presente che il testing White Box include quello Black Box per le modalità con cui è stato generato e che analogamente il test Random include quello Systematic.

Come è possibile vedere in tabella in tutti i casi tranne uno la copertura ottenuta con le tre tecniche si è rivelata più efficace del White Box. Nei casi in cui il testing random si era dimostrato migliore il valore raggiunto è addirittura pari all'unione di tutte le tecniche.

Se ne può dedurre che i test automatici riescono con buona approssimazione ad integrare i test Black Box pur senza accedere al sorgente.

COVERAGE (%)		#	Black Box + Systematic + Random	White Box	All
Line	FillUp	3807	76,06	84,88	88,18
	TippyTipper	999	88,36	86,33	88,76
	MunchLife	184	92,39	91,20	94,57
	SimplyDo	1281	85,10	79,33	85,10

FillUp

FillUp è l'unica tra le applicazioni considerate per cui l'unione BB+Syst+Rand non ottiene un risultato migliorativo rispetto a WB. In questo caso, la prevalenza di circa nove punti percentuali può essere imputata alla particolare complessità dell'applicazione che ha reso poco efficace il testing BB. In questa combinazione, infatti, è proprio il BB che dovrebbe introdurre la componente razionale umana.

TippyTipper

L'applicazione TippyTipper è quella con il minore indice di dipendenza dagli input. Nel confronto tra tecniche singole era stata evidenziata una moderata prevalenza della tecnica random rispetto a quella WB di conseguenza la copertura della combinazione risulta migliore. Considerando che lo scarto rispetto al risultato dell'unione è infinitesimo è possibile affermare che le tecniche automatiche colmano quasi del tutto il gap tra BB e WB.

MunchLife

MunchLife nel confronto tra tecniche singole aveva ottenuto una copertura maggiore con l'approccio WB. Dalla tabella emerge che la combinazione ha superato questo valore e dista meno del 2% da quello dell'unione.

In questo caso è evidente che test manuali ed automatici sono riusciti a sollecitare porzioni diverse del codice e che il gap tra BB e WB è parzialmente colmato dal testing automatico.

SimplyDo

Il caso di SimplyDo è particolare rispetto agli altri in quanto, la natura particolarmente intuitiva dell'applicazione, non ha consentito di ottenere un miglioramento passando all'approccio WB. Con queste ipotesi è immediato dedurre che WB è incluso in BB e dunque la combinazione BB+Syst+Rand coincide con l'unione di tutte le tecniche.

Conclusioni

Il mercato sempre in crescita delle applicazioni mobile per piattaforma Android ha fatto sì che si inasprisse la concorrenza e che la qualità dei prodotti offerti diventasse sempre più importante per sviluppatori e aziende. Rispetto al software tradizionale, oltre ad essere cambiati le modalità di sviluppo ed i canali di distribuzione, è completamente nuovo l'approccio della clientela di riferimento. Le applicazioni per smartphone e tablet sono tantissime, spesso gratuite e l'utente medio non esita ad eliminarle in caso di malfunzionamento o delusione delle aspettative. Le recensioni negative inoltre influiscono negativamente sui potenziali ulteriori acquirenti. Per completare questo scenario sono da considerare anche time to market stringenti e risorse spesso limitate.

È proprio nell'ambito di questo ciclo di sviluppo frenetico, dove spesso non è possibile dedicare risorse sufficienti al testing e più in generale ai processi di Quality Assurance, che nasce l'esigenza di strumenti e soluzioni per l'automatizzazione.

È stato dato ampio spazio alla descrizione dei motivi principali per cui il testing di applicazioni mobile è più critico rispetto a quello delle comuni applicazioni desktop e del perché non può essere assolutamente trascurato.

Mentre sono ormai diffusi e stabili framework e toolkit che consentono l'esecuzione automatica di test su emulatori, dispositivi reali e in cloud, sono ancora in via di miglioramento tecniche e strumenti per la generazione automatica di script eseguibili.

Sono state esaminate due famiglie di metodi per la generazione di questi test, una computer aided, che prevede l'interazione di un tester umano con un tool che registra e riproduce le sue azioni (black box e white box), ed una completamente automatica, nelle due principali modalità

di funzionamento: sistematica e random. I parametri utilizzati per la valutazione delle prestazioni di queste tecniche sono stati la copertura delle linee di codice, espressa in percentuale e l'efficienza espressa in termini di tempo impiegato per la generazione dei test normalizzato rispetto alla differenza tra la durata della giornata lavorativa di uomo e macchina.
Di seguito per ciascuna modalità di testing verranno sintetizzate le caratteristiche di efficienza, efficacia e costo.

Tecniche manuali

Le tecniche manuali per la generazione di test script sono le più diffuse in contesti aziendali medio/piccoli. Prevedono che un tester umano interagisca con l'applicazione mentre un apposito tool per il recording traduca le sue azioni in script.

Vantaggi:

- La generazione dei test grazie all'ausilio del tool di recording non richiede l'utilizzo di personale developer (in modalità black box).
- Grazie all'ausilio del tool di recording è possibile ridurre i tempi di generazione rispetto alla comune stesura manuale dei test.
- Non è necessario predisporre ambienti specifici in quanto la registrazione dei test può essere effettuata su una comune workstation
- Per un tester umano è molto più semplice individuare casi limite e valori critici pertanto ogni scenario di test eseguito è significativo

Svantaggi:

- Senza un piano di test accurato alcune funzionalità meno evidenti e comportamenti specifici potrebbero non essere testati.
- Il tool di recording non è in grado di registrare tutti i tipi di interazioni e spesso traduce le gesture più complesse in azioni più semplici lasciando inesplorati i relativi event handler.
- Il tester umano nell'arco della giornata lavorativa può essere soggetto a momenti di improduttività ed il suo contributo è limitato alle otto ore lavorative quotidiane.
- L'affidabilità del test effettuato, intesa come ripetibilità, è molto bassa in quanto i risultati ottenuto variano notevolmente in base al soggetto esecutore.

- Il tester umano tende a non esplorare combinatorialmente tutte le possibilità, limitandosi euristicamente a testare quelle che gli sembrano più significative. Un esempio di questo limite è costituito dalle liste di opzioni molto lunghe e dall'alternanza
- Spesso i test script prodotti dal tool di recording vanno elaborati per ridurre errori e migliorare la temporizzazione.
- Non è possibile sfruttare tempi morti per la generazione (notti e festivi).

Tecniche automatiche

Le tecniche automatiche operano esplorando l'applicazione a partire dagli elementi della GUI ed invocando eventi su di essa.

Vantaggi:

- Non richiede intervento umano se non per le fasi di startup e analisi dei dati.
- Possono operare anche di notte e nei festivi valorizzando intervalli di tempo che altrimenti sarebbero inutilizzati.
- Ispezionano tutti gli elementi della GUI ed i relativi event handler.
- In modalità sistematica hanno durata molto breve.
- Sono parallelizzabili.
- Producono test script immediatamente funzionanti.

Svantaggi:

- Hanno un range limitato di input tra i quali scegliere in maniera casuale, spesso insufficiente a coprire casi limite ed exception handling.
- L'ispezione di tipo random ha durata molto elevata.
- Per testare sequenze complesse è necessario programmare un numero di eventi troppo elevato.
- Nel caso di applicazioni dotate di stato, far partire tutti i test da uno stesso stato iniziale può impedire l'esplorazione di una parte del codice, ad esempio quello relativo al set up delle risorse.

In sostanza il confronto tra queste quattro tecniche ha evidenziato che nessuna può essere considerata completamente soddisfacente sia in termini di copertura che di tempo necessario

ma che con un'opportuna combinazione di esse è possibile ottenere risultati ottimali.

In un processo di sviluppo reale, tutte le modalità di test generation esaminate sono applicabili soltanto a partire dalla fase di system test, poiché essendo basate sulla GUI necessitano che l'applicazione sia funzionante ed utilizzabile su device.

Mentre nel tradizionale modello waterfall il system test chiude lo sviluppo e precede l'acceptance test del cliente, nell'approccio agile, estremamente diffuso e consigliato per la produzione di software mobile, è possibile utilizzare queste tecniche a valle di ciascuno "sprint" ovvero rilascio di un'unità funzionale consistente e testabile.

A partire dal momento in cui l'applicazione è disponibile per l'installazione è possibile arricchire il processo di integrazione quotidiano con l'esecuzione dei test generati, sia manualmente dallo sviluppatore stesso per la verifica dell'incremento rilasciato che automaticamente con un'ispezione notturna in grado di scatenare un gran numero di eventi nell'intervallo di improduttività.

Nel mondo mobile il ciclo di vita di un software non si arresta al momento del rilascio: come è stato ampiamente discusso nel primo capitolo, per rimanere sulla cresta dell'onda, è necessario che l'applicazione sia costantemente aggiornata e mantenuta.

Ogni modifica apportata al codice, sia essa correttiva o migliorativa, deve essere verificata in modo da individuare regressioni e nuovi bug. In questa fase risulta particolarmente vantaggioso l'utilizzo di ispezioni automatiche (con modalità dipendenti dal tempo a disposizione) per l'individuazione dei principali errori. Un'ideale interazione tra la generazione manuale dei test e quella automatica è la produzione di test case che portino l'applicazione in uno stato specifico a partire dal quale si desidera avviare il crawler.

Sviluppi Futuri

Al fine di trarre conclusioni di maggiore valore teorico sarebbe interessante evolvere il caso di studio presentato aumentandone le dimensioni e riprogettandolo in modo da risolvere ove possibile i limiti sperimentali individuati.

Sulla base delle principali criticità emerse durante l'utilizzo delle quattro tecniche di testing presentate e dell'influenza delle restrizioni dovuti alle modalità della sperimentazione ed ai tool utilizzati sul risultato, sono state individuate le seguenti aree di miglioramento:

- Aumentare il numero di casi di studio: applicare le tecniche di test generation ad un elevato numero di applicazioni consentirebbe di dare maggior credito ad i risultati ottenuti e generalizzare le conclusioni tratte.
- Aumentare il numero di esecutori per il testing manuale: il principale limite di questa tecnica in entrambe le modalità si è rivelato la fallibilità del tester. Tenendo in considerazione la soggettività del test prodotto e quindi dei relativi errori è corretto affermare che all'aumentare del numero di esecutori le principali omissioni/disattenzioni possono essere eliminate avvicinando il risultato a quello ideale.
- Fornire specifiche dettagliate per ciascuna applicazione: in modalità black box, se il tester non è a conoscenza del comportamento dettagliato dall'applicazione, c'è un'elevata probabilità che non tutte le funzionalità siano individuate e testate, falsando i risultati ottenuti.
- Ottimizzare la procedura di selezione casuale degli input: in entrambe le modalità di generazione automatica dei test, uno dei punti deboli si è rivelato l'inserimento di valori di input significativi. Attualmente il tool utilizzato è in grado di fornire esclusivamente

valori numerici interi nel range [0,99] spesso insufficiente per coprire le righe relative all'exception handling in caso di input nullo o scorretto ed alla validazione di valori limite (numeri negativi e soglie). Sarebbe interessante integrare il ripper con un approccio data driven che consenta, con un minimo di configurazione, di selezionare con maggiore precisione i valori forniti in ingresso.

- Perfezionare il criterio di equivalenza tra interfacce: il tool automatico per l'esplorazione sistematica della GUI utilizza come criterio d'arresto l'uguaglianza tra l'interfaccia corrente ed una di quelle già esplorate. Molto spesso questo confronto si è rivelato superficiale ed ha impedito l'esplorazione di nuovi percorsi.
- Selezione accurata degli snapshot utilizzati come punto di avvio dei tool automatici (condizioni iniziali del test): l'analisi della copertura ha evidenziato che, per le applicazioni dotate di stato, lo snapshot utilizzato per avviare le esplorazioni automatiche influenza la copertura ottenuta, ad esempio impedendo di esplorare il codice di set up dell'applicazione e creazione delle strutture dati. Un'opportunità di miglioramento per il tool potrebbe essere la selezione casuale tra un set di snapshot predefiniti, catturati in stati diversi dell'applicazione in modo da fondere le potenzialità dell'intuito umano con quelle del crawling automatico.
- Riduzione della dipendenza dalla casualità: è stato sottolineato che nel caso di ispezione random la probabilità di coprire un blocco di codice si riduce in dipendenza di quanto è lunga e complessa la sequenza di passi da compiere per attivarlo. Questa probabilità aumenta però al crescere del numero di eventi scatenati. Poiché il numero massimo di eventi eseguibili viene configurato prima dell'esecuzione sarebbe utile rendere il tool in grado di calcolare il punto di saturazione e fermarsi autonomamente una volta raggiunto.

Bibliografia

[1] State of Mobile, SmartBear, 2014;

[2] World Quality Report, Capgemini, 2014;

[3] White Box Testing, Wikipedia, http://en.wikipedia.org/wiki/White-box_testing, 31/01/2015;

[4] An Empirical Comparison between Model Learning and Random Testing Techniques in the Context of Android Applications -, Amalfitano, Amatucci, Fasolino, Tramontana - Department of Electrical Engineering and Information Technologies - University Federico II of Naples;

[5] A Toolset for GUI Testing of Android Applications, Amalfitano – Fasolino – Tramontana – De Carmine – Imparato, Dipartimento di Informatica e Sistemistica University of Naples Federico II, Italy;

[6] Emma Coverage Tool, Emma Sourceforge, http://emma.sourceforge.net/, 28/01/2015;

[7] TippyTipper, Google Code, https://code.google.com/p/tippytipper/, 29/01/2015;

[8] SimplyDo, Google Code, https://code.google.com/p/simply-do/, 30/01/2015;

[9] MunchLife, SJGAMES, http://www.sjgames.com/apps/levelcounter/, 31/01/2015;

[10] FillUp, Google Play Store, https://play.google.com/store/apps/details?id=com.github.wdkapps.fillup, 01/02/2015;

[11] Coverage Profiling of Large Applications, SmartBear, http://support.smartbear.com/articles/aqtime/coverage-profiling-of-large-applications, 02/02/2015;

[12] Increase efficiency and productivity with test automation, TestDroid, http://testdroid.com/testdroid/5851/increase-efficiency-and-productivity-with-test-automation, 27/12/2014;

[13] Test early test often testing as part of your app development, TestDroid, http://testdroid.com/testdroid/5876/test-early-test-often-testing-as-part-of-your-app-development, 27/12/2014;
[14] Rely only on real: emulators vs devices, TestDroid, http://testdroid.com/testdroid/5901/rely-only-on-real-emulators-vs-devices, 27/12/2014;
[15] 10 best practices for mobile app testing, TestDroid, http://testdroid.com/testdroid/5837/10-best-practices-for-mobile-app-testing, 27/12/2014;
[16] Emulated vs real device App testing, SmartBear, http://blog.smartbear.com/mobile/emulated-vs-real-device-mobile-app-testing/, 14/12/2014;
[17] The state of mobile development and testing, SmartBear, http://blog.smartbear.com/mobile/the-state-of-mobile-development-and-testing/, 15/12/2014;
[18] UIAutomator, Android Developer, http://developer.android.com/tools/help/uiautomator/index.html, 01/02/2015;
[19] UIAutomator, Android Developer, http://developer.android.com/tools/testing/testing_ui.html, 11/01/2015;
[20] Calabash, Xamarin Developer, http://developer.xamarin.com/guides/testcloud/calabash/introduction-to-calabash/, 12/01/2015;
[21] Mobile Application Testing, Wikipedia, http://en.wikipedia.org/wiki/Mobile_application_testing, 13/01/2015;
[22] Testing mobile Tips, Software quality - TechTarget, http://searchsoftwarequality.techtarget.com/feature/Testing-mobile-apps-Tips-on-manual-automated-cloud-QA, 20/12/2014;
[23] Creating and implementing a mobile testing strategy, Software quality - TechTarget, http://searchsoftwarequality.techtarget.com/tip/Creating-and-implementing-a-mobile-testing-strategy, 20/12/2014;
[24] Mobile testing strategies, Software quality - TechTarget, http://searchsoftwarequality.techtarget.com/tip/Mobile-testing-Nine-strategy-tests-youll-want-to-perform, 20/12/2014;
[25] Tools for extracting system and performance information, Software quality - TechTarget,

http://searchsoftwarequality.techtarget.com/tip/Tools-for-extracting-system-and-performance-information, 20/12/2014;
[26] What is mobile testing, SmartBear, http://smartbear.com/all-resources/articles/what-is-mobile-testing/, 20/12/2014;
[27] 10 Best Practices for mobile app testing, TestDroid, http://testdroid.com/tech/10-best-practices-for-mobile-app-testing, 19/01/2015;
[28] Mobile testing best practices, Belatrix, http://www.belatrixsf.com/index.php/whitepaper-mobile-testing-best-practices-m, 21/01/2015;
[29] Mobile Application Testing, Infosys, http://www.infosys.com/flypp/resources/Documents/mobile-application-testing.pdf , 22/01/2015;
[30] Mobile testing strategies, Keynote, http://www.keynote.com/resources/white-papers/testing-strategies-tactics-for-mobile-applications, 23/01/2015;
[31] Mobile application testing, TCS, http://www.tcs.com/SiteCollectionDocuments/White%20Papers/Mobility_Whitepaper_Mobile-Application-Testing_1012-1.pdf, 25/01/2015;
[32] Android Developer Console, Android Developer, https://developer.android.com/distribute/googleplay/developer-console.html, 10/01/2015;
[33] Advanced Mobile Testing Scenarios, CloudBees, https://developer.cloudbees.com/bin/view/Mobile/Advanced+Scenarios , 27/01/2015;
[34] Android Testing on devices, SauceLabs, https://saucelabs.com/mobile/android-testing/android-testing-on-devices, 30/01/2015;
[35] UIAutomator, Intel Developer, https://software.intel.com/en-us/android/articles/automatic-android-testing-with-uiautomator, 31/01/2015;
[36] Different android test automation frameworks - what works you the best?, TestDroid, 2013;
[37] Robotium, Wikipedia, http://en.wikipedia.org/wiki/Robotium, 01/03/2015;
[38] BotBot Recorder, Imaginea, http://imaginea.github.io/bot-bot/pages/recorder/recorder.html, 10/11/2014;

[38] MonkeyTalk Recorder, CloudMonkey, https://www.cloudmonkeymobile.com/monkeytalk, 20/11/2014;

[39] Ranorex Recorder, Ranorex, http://www.ranorex.com/mobile-automation-testing.html, 01/12/2014;

[40] Testdroid Recorder, Testdroid, http://testdroid.com/, 12/11/2014

[41] Robotium Recorder, Robotium Tech, http://robotium.com/products/robotium-recorder, 13/11/2014;

[42] Selendroid Inspector, Selendroid, http://selendroid.io/, 12/11/2014;

[43] Espresso, GTAC – 2013 Espresso presentation slides, 2013

[44] Android Instrumentation Framework, Android Developers, http://developer.android.com/tools/testing/testing_android.html, 03/01/2015;

[45] Appium, http://appium.io/, 04/01/2015;

[46] Robotium Framework, Google Code, https://code.google.com/p/robotium/;