POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica (Computer Engineering)

Tesi di Laurea Magistrale

Implementazione di un tool ibrido per eseguire test di applicazioni Android

Relatori:
prof. Luca Ardito
prof. Maurizio Morisio

Candidato:
Francesco Petrolino
matricola: s242227

Supervisore Aziendale
Carlo Ferrero

ANNO ACCADEMICO 2018-2019

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

Negli ultimi anni lo sviluppo dei dispositivi Mobile ha subito un'evoluzione esponenziale dal punto di vista tecnico e sociale. La diffusione su larga scala di questi dispositivi ha portato inevitabilmente all'integrazione di questa tecnologia nella vita di tutti i giorni, diventando così parte integrante quasi indispensabile. A seguito di ciò, lo sviluppo di applicazioni in quest'ambito ha interessato una grossa fetta di mercato. Sempre più industrie hanno indirizzato i propri sforzi economici ed il loro interesse nell'integrazione dei propri servizi nel settore Mobile, tramite la creazione di applicazioni e la formazione di esperti in materia.
In particolare, è stata predominante l'evoluzione in ambito Android. Questo sistema operativo è stato sviluppato da Google ed è basato su kernel Linux; esso continua attraverso l'Android Open Source Project, ovvero un software libero che non comprende alcuni firmware dedicati esclusivamente ai produttori di dispositivi e di alcune Google Apps, ad esempio "Google Play". La possibilità di poter interagire in maniera aperta con questo sistema tramite apposite API e di creare applicazioni tramite gli strumenti messi a disposizione dall'azienda stessa in maniera del tutto libera ha fatto sì che nell'ultimo decennio si formassero professionalmente milioni di sviluppatori e nascessero milioni di applicazioni.
Data la diffusione su scala globale e l'introduzione in maniera stabile nel mercato, è necessario che i software siano funzionanti, leggeri ed intuitivi. Da qui è nata l'esigenza di inserire come parte integrante nel processo di sviluppo una fase di testing e validazione. Questo processo risulta fondamentale nel momento in cui bisogna creare un prodotto che rispetti determinate caratteristiche grafiche e funzionali, ed è necessario per accertarsi che siano stati implementati requisiti e specifiche richiesti e per garantire che il software finale risulti utilizzabile e funzionante.
Gli approcci al testing sono molteplici ed ognuno presenta punti di forza ed elementi di criticità.
La metodologia più semplice è il testing manuale: il software, ancora in fase di

sviluppo, viene reso disponibile ad un numero chiuso di persone che ne testa le funzionalità e l'intuitività. Questa operazione emula il momento in cui l'utente finale dovrà utilizzare l'applicazione e dunque interagire con la GUI ed esplorarne le funzionalità.
Come si può intuire, questa esecuzione presenta dei costi elevati in termini di tempo ed affidabilità: i tempi di ricerca di utenti e di interazione tra questi e gli sviluppatori potrebbero essere estremamente lunghi ed i risultati potrebbero non essere soddisfacenti.
Per ovviare a questi problemi, è sorta l'esigenza di sviluppare dei meccanismi di automazione di testing di applicazioni Android. Questo approccio si basa su tecniche mirate alla creazione di casi di test che vengono eseguiti in maniera automatica, simulando verosimilmente le possibili interazioni dell'utente. Lo scopo è quello di testare tutte le funzionalità e le specifiche richieste, così come le operazioni sulla GUI dell'applicazione e le sue risposte.
A seguito di numerosi studi e ricerche da parte di esperti e sviluppatori, sono nate numerose tecniche per automatizzare i test su dispositivi Android.
Lo strumento principale messo a disposizione dagli stessi creatori di Android è Espresso. Grazie a questo framework è possibile progettare e creare script di testing, utilizzando le API per ricreare le interazioni con la GUI. Poiché questa metodologia si basa sul riconoscimento di elementi grafici tramite identificativi specifici, presenta una serie di limitazioni nel momento in cui questi cambiano e dunque non sarà più possibile individuarli. [9] Questo comporta una modifica dei test ogni qualvolta venga fatta una modifica al codice che descrive il layout dell'applicazione. Questa tipologia di test viene definita come seconda generazione.
Per risolvere questo problema, è stata sviluppata un'altra tipologia di testing che si basa sul riconoscimento visuale. Tramite appositi framework, come Eye Automate e SikuliX, è possibile sviluppare dei test che si basano su algoritmi di riconoscimento delle immagini per individuare gli elementi grafici nella GUI con i quali verranno effettuate le interazioni. Anche questa metodologia presenta dei limiti, nel momento in cui i widget possiedono sempre gli stessi identificativi ma il layout cambia, e dunque porta al fallimento degli algoritmi di riconoscimento delle immagini. Questa tipologia di test viene definita come terza generazione.
La soluzione per ottimizzare il processo di GUI testing è quella di combinare le due tipologie: ciò permetterebbe di ovviare al problema del cambio di identificativi degli elementi grafici applicando l'algoritmo di riconoscimento delle immagini, e nel momento in cui questo possa fallire, applicare la ricerca tramite identificativo definito nel codice che definisce il layout.
Questo lavoro di tesi fa parte di una più ampia attività di ricerca, incentrata sullo studio delle criticità dei due metodi di GUI testing e mirata alla realizzazione di un tool che realizzi in maniera semplice ed intuitiva il passaggio fra i due tipi di script.
In particolare, la struttura generale è composta da una serie di componenti che

realizzano la traduzione degli script di seconda generazione che dà come risultato script visuali di terza generazione. L'architettura si compone dei seguenti elementi:

- **Enhancer**: modifica ed arricchisce gli script di seconda generazione per inserire al loro interno degli strumenti che serviranno alla generazione delle interazioni con la GUI.
- **Executor**: si occupa dell'interazione con il dispositivo emulato (AVD).
- **Log Parser**: si occupa della raccolta delle informazioni necessarie per la traduzione dei test e la creazione degli script di terza generazione.
- **Creatore di Script di Terza Generazione**: permette di utilizzare i dati ottenuti dal Log Parser per creare gli script di terza generazione.

Nello specifico, il lavoro svolto e presentato in questo elaborato di tesi si concentra sullo studio e lo sviluppo del componente denominato **Executor**. Per la sua realizzazione sono stati analizzati diversi aspetti considerati fondamentali. Prima di tutto, è stato necessario esplorare il framework Espresso e la sua API, fulcro della creazione di script di seconda generazione, ed entrare nel dettaglio delle sue funzionalità e dei metodi disponibili. Successivamente, la parte strutturalmente più importante è stata l'analisi approfondita dell'Android Debug Bridge (ADB), tramite il quale è stato possibile interfacciarsi ed interagire con il dispositivo emulato (AVD), sul quale le applicazioni sono state installate ed eseguiti i test. Al suo interno, l'Executor contiene le diverse funzionalità di Enhance e Traduzione dei test, con successiva generazione degli script di terza generazione. Dunque sono stati fondamentali l'approfondimento degli altri componenti del progetto e l'integrazione delle loro operazioni. Infine, sono state esplorate le funzionalità dei tool per la creazione ed esecuzione di script di terza generazione, Eye Automate e Sikulix, che mettono a disposizione le API attraverso le quali è stata integrata la possibilità di lanciare i test attraverso l'Executor stesso.
Il tool è stato sviluppato con il linguaggio di programmazione Java, rendendo così la sua esecuzione indipendente dalla macchina, grazie alla presenza della Java Virtual Machine (JVM). Inoltre, è stata necessaria l'integrazione di apposite strutture dati che permettessero l'interazione con la command line, tramite cui è stata eseguita la maggior parte dei comandi per interagire con i dispositivi emulati. L'Executor dispone di un'interfaccia grafica per l'utente finale, sviluppata interamente in Java Swing, framework di Java per lo sviluppo di GUI. Grazie ad essa l'utente finale avrà modo di utilizzare tutte le funzionalità presentate in maniera semplice ed intuitiva.
A seguito dello sviluppo di questo tool, sarà effettuata una operazione di validazione per rispondere ad una serie di Research Question. Questa fase permetterà di verificare le tesi di partenza, di dimostrare l'utilità del software e la sua efficacia in casi d'uso concreti e reali. Il presente elaborato è strutturato nel modo seguente:

- nel Capitolo 2 viene presentato il background di questo progetto tramite esposizione di lavori di ricerca correlati e tramite l'analisi degli elementi di base, partendo dai concetti di testing più generici fino ad arrivare al Mobile GUI testing ed alle sue classificazioni, con particolare attenzione al framework Espresso e agli strumenti di testing visuale;
- nel Capitolo 3 viene presentata l'architettura del progetto ed una visione generale dei componenti che formano la sua struttura, assieme alle scelte di design ed implementazione;
- nel Capitolo 4 viene presentato il punto centrale di questo elaborato, ovvero l'Executor: sono esposti gli aspetti grafici, le funzionalità, le scelte di design e la sua effettiva implementazione.
- nel Capitolo 5 viene presentata la fase di validazione del tool, tramite cui è stato possibile verificare la sua utilità ed efficacia.

Capitolo 2

Background e Related Works

In questo capitolo verrà esposta una visione completa dell'ambito del testing, a partire dai concetti generali per poi concentrarsi sui testing visuali in ambito mobile, con attenzione particolare a quello che riguarda la distinzione tra le varie "generazioni" dei test, i tool di testing visuale e le tecniche di riconoscimento delle immagini.

2.1 Testing: concetti generali

Il testing di applicazioni mobile è stato discusso ed analizzato in numerosi studi. Questo perché l'utilizzo degli smartphone è ormai parte integrante della vita quotidiana, e permette di effettuare operazioni anche critiche, ad esempio pagamenti online. La qualità dei software sviluppati per questi dispositivi è composta da differenti aspetti ed è fondamentale poter garantire, prima del rilascio ufficiale, che l'applicazione sia quanto più stabile ed efficiente possibile, in modo da rendere soddisfacente la User Experience.
Il testing, o collaudo, è uno step che fa parte del ciclo di vita di un software e avviene una volta che il prodotto è stato realizzato e ha superato la fase di debugging. Lo scopo della fase di testing è, una volta che il prodotto è stato terminato, di aumentare la qualità e l'usabilità minimizzando gli errori di programmazione, che potrebbero causare malfunzionamenti. Si parla di riduzione, in quanto risulta infattibile riuscire ad annullare completamente tutti i casi di errore, in quanto non è possibile analizzare tutti gli output del sistema. Tuttavia, si può ridurre quanto più possibile il rischio di failures ed aumentare la qualità del prodotto, che verrà considerato accettabile indipendentemente dal tipo di applicazione. Il collaudo viene inserito nella fase di verifica e validazione: tramite essa ci si vuole accertare che le specifiche del sistema siano state realizzate correttamente e a pieno (verifica) e

che siano congruenti con le specifiche commissionate (validazione). Normalmente, la validazione viene effettuata sul prodotto finale; tuttavia, può essere effettuata su diversi moduli durante la fase di sviluppo, se si limita ad alcune caratteristiche specifiche. Ci sono due tipi di verifica: statica e dinamica. La verifica statica viene realizzata sul programma non in esecuzione, utilizzando alcuni strumenti che analizzano teoricamente il codice per testarne la robustezza su tutti gli input possibili. La verifica dinamica viene, invece, realizzata tramite il testing.
Il testing è composto da diverse operazioni: la preparazione del test, il quale non va improvvisato ma studiato con attenzione; preparazione dell'ambiente di test, che ne permette un'esecuzione controllata attraverso l'inserimento di opportuni dati in input; l'analisi dei risultati, effettuata sui dati in output per ricercare eventuali failures.
Il system testing viene effettuato per un sistema già realizzato, per verificare che siano stati rispettati tutti i requisiti. Questo meccanismo si può calare nell'ambito delle specifiche di funzionalità o delle specifiche di sistema. In questo ambito si differenziano due metodologie specifiche: software testing e mobile-device testing.
Il software testing si basa sull'analisi di un componente software o di sistema per verificarne determinate proprietà, che rappresentano le caratteristiche richieste, le funzioni che deve effettuare, i risultati in base a determinati input. Tutto ciò verifica la qualità del software o del servizio prodotto.
Il mobile-device testing si effettua sia su hardware che su software e serve per garantire la qualità di un determinato dispositivo mobile. Riguarda sia risoluzione e monitoraggio di problemi di applicazioni, sia casi d'uso reali. Questa operazione comprende anche una fase di verifica e di validazione. [8] [1]

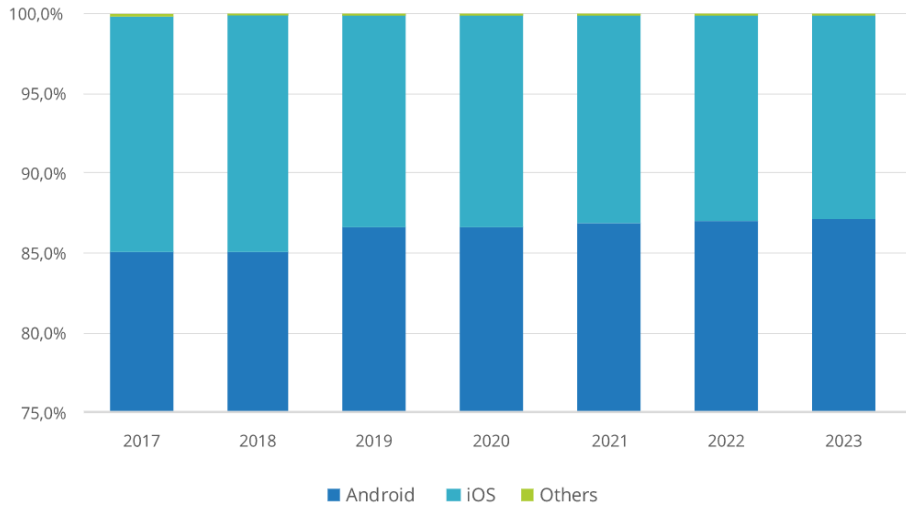
2.1.1 GUI Testing

Il testing della graphical user interface (GUI) serve a verificare che l'interfaccia grafica del prodotto in questione rispetti le specifiche. Per questo scopo, vengono progettati e creati dei test case, che hanno come obiettivo quello di ricoprire tutte le funzionalità di sistema e testare la GUI nella sua interezza. I problemi da affrontare in questo tipo di testing sono il numero delle operazioni e la sequenzialità.
Il primo problema riguarda l'elevato numero di operazioni da coprire attraverso i test case, che possono arrivare a dimensioni notevoli anche nel caso di applicazioni relativamente semplici.
Il secondo riguarda la sequenzialità delle operazioni. In particolare, determinate funzionalità richiedono una serie di eventi in sequenza riguardanti l'interfaccia grafica.
La risoluzione di questi problemi si può trovare nell'automazione della generazione dei test case.

2.2 Mobile GUI testing

Poiché il mondo del mobile si è sviluppato ed allargato in maniera estremamente rapida, negli anni sono state rese a disposizione sul mercato milioni di applicazioni create da altrettanti sviluppatori ed utilizzate da altrettanti utenti, su una vasta gamma di dispositivi diversi. Il grafico in Figura 2.1 mostra l'andamento delle

Figura 2.1. Grafico delle spedizioni passate e previste di Android e iOS



spedizioni passate per i dispositivi con sistemi operativi Android, iOS e altri e le previsioni delle vendite nei prossimi anni.
Per quanto riguarda i dispositivi Android, è stato previsto per il 2019 un incremento dall'85.1% all'86.7%, includendo il lancio sul mercato dei dispositivi 5G. La previsione per i successivi 5 anni è un incremento annuale del 2.1%, con un totale di 1.32 miliardi di spedizioni nel 2023.
Per quanto riguarda iOS, è stato previsto per il 2019 un calo delle spedizioni a 183.5 milioni, il 12.1% rispetto all'anno precedente. La previsione per i successivi 5 anni sarà un calo totale di 0.4%. Questo risultato, verosimilmente, non dipenderà dall'impossibilità momentanea di distribuire dispositivi 5G nel 2019. [19] [20]
I dati globali sono mostrati nella tabella in Figura 2.2. Lo sviluppo di quest'ambito ha subito una crescita notevole, permettendo la definizione di nuove pratiche di programmazione per la creazione delle applicazioni. Questo, unito ad una richiesta del mercato sempre maggiore, ha condotto alla necessità di sviluppare strumenti,

Figura 2.2. Tabella delle vendite totali nel tempo di Android e iOS

Year	2017	2018	2019	2020	2021	2022	2023
Android	85,1%	85,1%	86,7%	86,6%	86,9%	87,0%	87,1%
iOS	14,7%	14,9%	13,3%	13,4%	13,1%	13,0%	12,9%
Others	0,2%	0,0%	0,0%	0,0%	0,0%	0,0%	0,0%
TOTAL	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%	100,0%

servizi e tecniche per il testing in ambito mobile, in modo da garantire la qualità dei prodotti rivolti al mercato.
La grande maggioranza dei test viene effettuata manualmente e questo implica un enorme sforzo in termini di tempo e denaro. Ciò è dovuto ai limiti presenti nell'approccio automatico al testing. Tuttavia, è necessario che questo approccio venga migliorato ed attuato nell'ambito dello sviluppo di applicazioni mobile, data la continua crescita di questo settore. Attualmente, i test automatici non si distaccano del tutto dall'approccio manuale, poiché presentano ancora alcune limitazioni, che potrebbero essere migliorate tramite un contributo da parte degli sviluppatori in modo tale da fornire un metodo più agile e pratico per garantire la qualità delle applicazioni proposte al mercato. [11] [6]
In questa sezione verranno presentati e discussi alcuni strumenti e servizi creati per supportare e migliorare il testing in ambiente mobile. Questa analisi è fondamentale nel momento in cui si vuole progettare un sistema per il testing mobile, poiché è necessario capire quali siano i limiti attuali da superare.
[3] [4] [5]

2.2.1 Framework ed API per l'automazione

Gli strumenti messi a disposizione per gli sviluppatori di GUI Automation operano da interfaccia per acquisire informazioni sullo schermo, in modo tale da poter interagire con la GUI e simulare le azioni dell'utente finale. Forniscono agli sviluppatori le API per poter creare dei test case che operino sull'interfaccia grafica tramite script scritti a mano o tramite registrazione. Questi, solitamente, presentano una successione di azioni da eseguire su dei componenti specifici riconosciuti tramite identificativi o ulteriori attributi, per poi concludersi con un controllo generico sullo stato della GUI. Questi framework sono robusti ed attrezzati alla presenza di diversi tipi di dispositivi, garantendo la creazione di script che siano compatibili nella maggior parte dei casi; tuttavia, in alcune situazioni potrebbero fallire. Inoltre, spesso si incentrano solamente su operazioni di base, non essendo in grado di

supportare operazioni più complesse.
Questi tipi di test richiedono una pesante manutenzione dovuta alle modifiche nel tempo sulla GUI dell'applicazione. I test basati esclusivamente sull'interfaccia grafica presentano un problema di fragilità, poiché una qualsiasi variazione nel layout dell'applicazione conduce al fallimento dell'esecuzione del test. Dunque, per garantire la totale funzionalità, i test vanno modificati nel tempo a seguito delle modifiche della GUI dell'applicazione sulla quale vengono eseguiti. [10] [2] [12]

2.2.2 Strumenti di Record e Replay

Gli strumenti che si basano su Record & Replay (R&R) sono nati con lo scopo di rendere più efficienti e veloci i test rispetto a quelli puramente manuali. Essendo un compromesso tra testing manuale e scriptato, presentano una maggiore robustezza agli errori rispetto al primo e una minore pesantezza rispetto al secondo.
Questo strumento mette a disposizione un modo semplice ed intuitivo per la creazione di test case significativi, anche nel caso in cui lo sviluppatore non abbia una conoscenza approfondita nell'ambito.
In alcuni casi, la struttura dell'applicazione da testare richiede un livello di reattività e di dettaglio tale da poter essere testata solo tramite strumenti che rispondono a queste caratteristiche.
Molte di queste soluzioni presentano alcuni svantaggi che riguardano la bassa accuratezza o tempistica, l'una a discapito dell'altra, e la portabilità degli script creati; mentre, idealmente, un approccio R&R dovrebbe presentare prestazioni elevate in termini di precisione e portabilità. [10]

2.2.3 Tecniche di automazione delle generazione di input

Le tecniche di generazione automatizzata (AIG) di input nascono con lo scopo di rendere meno complessi e tediosi la scrittura ed il recording dei test di script generando automaticamente gli input. Gli obiettivi sono quelli di ridurre il numero di bug e la lunghezza dei casi di test, i quali riescono ad emulare dei casi reali di utilizzo dell'applicazione.
Queste tecniche si possono catalogare in: generazione di input random-based, generazione sistematica di input, generazione di input model-based.
Studi recenti hanno mostrato alcune limitazioni, per le quali queste tecniche ancora non riescono a risolvere alcuni grandi problemi, ad esempio i test "flaky" , la frammentazione, la possibilità di avere diversi casi di test ed i meccanismi poco sviluppati di feedback per gli sviluppatori. [10]

2.2.4 Strumenti per il monitoraggio e la segnalazione di bug

Questi strumenti sono diventati parte integrante del processo di test mobile e possono essere divisi in due tipi: strumenti per la segnalazione di bug e strumenti per monitoraggio di fallimenti e consumo di risorse da parte dei programmi a run-time. Il primo tipo permette di descrivere i bug usando segnalazioni testuali o tramite segnalazioni visuali con immagini; il secondo permette di ricevere solamente segnalazioni da parte di utenti o segnalazioni di crash automaticamente generate. Molto spesso, le revisioni degli utenti o le segnalazioni senza contesto risultano poco utili agli sviluppatori, poiché non sono in grado di specificare il tipo di problema che si è verificato.
Alcuni strumenti sono stati creati per tentare di risolvere questi problemi, con lo scopo di informare gli sviluppatori in maniera più puntuale dei tipi di fallimento e di errori che si sono verificati. Inoltre, potrebbero essere forniti ulteriori dettagli su come l'utente interagisce con le applicazioni. [10]

2.2.5 Servizi di testing mobile

I servizi di testing mobile sono un'utile alternativa alla creazione manuale di script di test, assieme alle tecniche di generazione automatizzata di input. Questi strumenti permettono di ridurre il costo per la generazione di test case o di segnalazione di bug, tramite gruppi di risorse umane che operano nel contesto del testing.
Possono essere distinti 4 tipi di servizi: Traditional Crowd-Sourced Functional Testing, attraverso cui esperti e non esperti nell'ambito del testing si incaricano di inviare report che riguardano problemi nelle applicazioni; Usability testing, il cui scopo è l'analisi e lo sviluppo di interfacce grafiche per creare applicazioni intuitive e user friendly; Security Testing, utile ad analizzare possibili errori nella progettazione dell'applicazione che possano comprometterne la sicurezza; Localization Testing, che si assicura che le applicazioni funzionino indipendentemente dalla locazione e dalla lingua.
Nessuna di queste soluzioni è open source, il che induce gli sviluppatori a raccogliere dati riguardo bug e problemi da contesti meno professionali e precisi. Inoltre, queste soluzioni non risultano scalabili nel momento in cui le applicazioni continuano ad evolvere e a diventare più complesse. [10]

2.2.6 Strumenti per lo streaming del dispositivo

Questi strumenti possono migliorare il processo di testing in ambiente mobile, permettendo la riproduzione di un dispositivo fisico su una macchina personale o tramite accesso via Internet. Essi permettono e supportano la ricerca che implica l'acquisizione di dati durante determinati studi che implicano testing crowdsourced. [10]

2.3 Classificazione delle generazioni di test

A seguito degli sviluppi nell'ambito delle applicazioni mobile, è divenuto di cruciale importanza sviluppare un approccio al testing GUI più potente e semplice, lasciando spazio alla creazione di nuove tecniche di testing automatico. Questa necessità è sorta nel momento in cui le applicazioni sono diventate più GUI intensive, presentando tecniche sempre più avanzate di interazione con l'interfaccia grafica.
Da ciò, è possibile racchiudere in tre generazioni i vari tipi di GUI testing automatizzato in base a come gli elementi grafici vengono riconosciuti. [7]

1. **Prima Generazione**
   Si basa sulle interazioni manuali con l'applicazione, da cui vengono estrapolate le coordinate per replicare successivamente il test. I costi di mantenimento di queste tipologie di test sono elevati a causa della scarsa robustezza alle modifiche della GUI, alle differenze tra gli schermi, come dimensione e risoluzione, e hanno portato ad un progressivo abbandono di questo approccio.

2. **Seconda generazione**
   Si basa sulle interazioni direttamente con widget tramite le loro proprietà ed i relativi valori. Questo tipo di approccio è più robusto rispetto a quello precedente ai cambiamenti dell'applicazione, presenta prestazioni più elevate ed esecuzioni più stabili. Questa tecnica ha comunque una limitazione, in quanto è applicabile ad applicazioni scritte in determinati linguaggi con API GUI conosciute, poiché per accedere al modello della GUI è necessario accedere alle librerie sottostanti che la gestiscono.

3. **Terza generazione**
   Chiamata anche Visual GUI Testing (VGT), si basa sugli algoritmi di riconoscimento delle immagini per interagire con la GUI dell'applicazione, la quale presenta determinate caratteristiche e viene mostrata in un modo specifico sul monitor dell'utente. Sono presenti alcuni limiti di conoscenza per quanto riguarda la possibilità di utilizzare a lungo termine questo tipo di approccio, per problemi di robustezza, nonostante si sia dimostrata la sua applicabilità in differenti contesti.

2.4 Espresso

Espresso [15] è un framework creato da Google che fornisce una serie di API instrumentation-based per la creazione di test sulla UI di applicazioni Android, che si appoggia al test runner AndroidJUnitRunner.
Questo framework è stato progettato e realizzato per il testing automatizzato come

parte integrante del ciclo di sviluppo di un'applicazione: può essere utilizzato per il testing black-box o, più opportunamente, per il testing del codice sorgente.
L'API fornita da Espresso è intuitiva e di facile comprensione e lascia spazio ad eventuali personalizzazioni. Inoltre, opera in modo tale da non essere interferita dal tipo di contenuto, di infrastruttura o di implementazione dell'applicazione e della sua interfaccia.
Il vantaggio principale che si ha è quello di avere una sincronizzazione automatica delle operazioni all'interno dei test con l'interfaccia grafica, così da permettere una simulazione efficace e realistica delle interazioni dell'utente con un'applicazione. Inoltre, Espresso è in grado di riconoscere quando l'applicazione è in uno stato d'attesa, dovuto, ad esempio, al passaggio da una schermata ad un'altra: ciò aumenta l'affidabilità e la velocità di esecuzione dei test e garantisce allo sviluppatore la possibilità di non utilizzare comandi di wait per sopperire ai problemi di timing e sincronizzare le operazioni con l'interfaccia grafica.
Ogni qualvolta venga invocato il metodo **onView**, Espresso attende che venga eseguita l'azione sull'interfaccia finché non si verificano determinate condizioni:

- La coda dei messaggi è vuota.
- Non ci sono più istanze di AsyncTask che stanno eseguendo un'operazione.
- Tutte le risorse che implicano un'attesa sono terminate.

La conferma che ciò avvenga permette di aumentare notevolmente la possibilità che una ed una sola operazione venga effettuata in un determinato momento, garantendo così risultati più affidabili.
Inizialmente è necessario specificare il componente grafico o la view con la quale Espresso dovrà interagire.
Il metodo **onView()** permette di identificare la view corrispondente all'identificativo utilizzato come parametro del metodo stesso. L'oggetto restituito è di tipo ViewInteraction, attraverso cui è possibile interagire con la view.
Per capire in dettaglio come si susseguono le operazioni: tramite ActivityTestRule viene lanciata l'activity appropriata; quindi, vengono eseguiti i metodi con l'annotazione @Before e, successivamente, test con l'annotazione @Test. Espresso gestisce la chiusura dell'activity e, infine, esegue tutti i metodi annotati con @After.
Il metodo onView() permette di identificare la view corrispondente all'identificativo utilizzato come parametro del metodo stesso. L'oggetto restituito è di tipo ViewInteraction, attraverso cui è possibile interagire con la view.
Un esempio di test case Espresso è mostrato in Figura 2.3.

2.4.1 Comandi di Espresso

Le operazioni effettuate sui componenti grafici sono eseguite in Espresso tramite gli oggetti ViewAction con il metodo ViewInteraction.perform(). I componenti

Figura 2.3. Esempio di test case Espresso

```
@Test
public void testDeleteNote(){

    onView(withId(R.id.fab_expand_menu_button)).perform(click());
    onView(withId(R.id.fab_note)).perform(click());
    onView(withId(R.id.detail_title)).perform(typeText( stringToBeTyped: "Test1"));
    onView(withContentDescription( text: "drawer open")).perform(click());
    onView(withText("Test1")).perform(click());
    openContextualActionModeOverflowMenu();
    onView(withText("Trash")).perform(click());
    onView(withText("Nothing here!")).check(matches(isDisplayed()));


}
```

vengono identificati tramite un ViewMatcher e le view all'interno della gerarchia generale sono riconosciute tramite il metodo onView(). Dopo che le azioni sono state eseguite, le ViewAssertion sono verificate dal metodo check() che verifica la correttezza dello stato dalla view corrente.

Operazioni di click

Operazioni semplici che simulano click sulle view individuate nella gerarchia corrente.

click(int inputDevice, int buttonState) : esegue un'azione di click sulla view per uno specifico device in input e uno stato del pulsante.

doubleClick() : esegue un doppio click sulla view.

longClick() :esegue un click lungo sulla view.

Operazioni da tastiera

Azioni che eseguono operazioni sulle TextView o EditTextView della gerarchia del layout corrente.

clearText() : effettua un clear del test sulla view.

replaceText(String stringToBeSet) : effettua un aggiornamento del testo della view, sostituendolo con la stringa specificata come parametro in ingresso al metodo.

typeText(String stringToBeTyped) : seleziona la view effettuando un click e inserisce il testo specificato nella stringa in ingresso.

typeTextIntoFocusedView(String stringToBeTyped) : inserisce il testo specificato come parametro del metodo dentro una view sulla quale è già stato effettuato un click.

pressKey(int keyCode) : esegue l'operazione che preme il tasto specificato tramite keyCode.

pressKey(EspressoKey key) : esegue l'operazione che preme il tasto specificato con i modifier specificati.

Operazioni di Scroll e Swipe

Azioni che eseguono operazioni di trascinamento sulla view della gerarchia corrente.
La funzione scrollTo() è eseguibile solo sulle View che sono state definite come discendenti della classe ScrollView. Le operazioni di swipe possono essere eseguite su qualsiasi view.

scrollTo() : esegue l'operazione di scroll sulla view, la quale deve essere visibile e deve derivare dalla classe ScrollView.

swipeDown() : eseuge l'operazione di swipe dall'alto in basso lungo la linea orizzontale centrale della view. L'operazione non inizia dall'estremo superiore della view ma presenta un piccolo offset.

swipeUp() : esegue l'operazione di swipe dal basso verso l'alto lungo la linea orizzontale centrale della view. L'operazione non inizia dall'estremo inferiore della view ma presenta un piccolo offset.

swipeLeft() : esegue l'operazione di swipe da destra verso sinistra lungo la linea verticale centrale della view. L'operazione non inizia dall'estremo laterale della view ma presenta un piccolo offset.

swipeRight() : esegue l'operazione di swipe da sinistra verso destra lungo la linea verticale centrale della view. L'operazione non inizia dall'estremo laterale della view ma presenta un piccolo offset.

Operazioni speciali

Azioni specifiche per la GUI Android, che eseguono operazioni su elementi propri delle applicazioni Android.

closeSoftKeyboard() : esegue l'operazione di chiusura della tastiera software.

pressBack() : esegue l'operazione di click sul pulsante di "back".

pressBackUnconditionally() : simile a pressBack() ma non genera un'eccezione quando Espresso si trova al di fuori dell'applicazione o del processo sotto test.

pressImeActionButton() : esegue l'operazione di click sul pulsante dell'azione corrente sull'IME (Input Method Editor).

pressMenuKey() : esegue l'operazione che preme il pulsante hardware del menu. (Metodo deprecato)

2.5 Strumenti di testing visuale

In questa sezione verranno illustrati i due strumenti di testing visuale principali utilizzati durante questo lavoro: EyeAutomate e Sikuli.

2.5.1 EyeAutomate

EyeAutomate [17] [13] è un esecutore di script visuali. L'algoritmo di riconoscimento delle immagini adottato permette di localizzare le immagini contenute negli script eseguiti da questo strumento. Essendo basato interamente su Java, è possibile utilizzarlo su qualsiasi piattaforma che supporta la JVM, automatizzando qualsiasi interazione su applicazioni che abbiano un'interfaccia grafica.
EyeAutomate si basa su differenti tecniche di automatizzazione di test su GUI. Possono essere distinte tre diverse generazioni di test:

1. Basati sulla posizione. Vengono utilizzate le coordinate esatte per il record & replay.

2. Basati sui widget. Le applicazioni desktop variano in dimensione dello schermo e posizione su di esso. Perciò, i widged dell'interfaccia sono messi a disposizione in modo tale da poter generare script robusti ai cambiamenti dell'interfaccia. La limitazione di questa tecnica rimane la presenza di widget personalizzati.

3. Visual GUI Testing (VGT). Questa tecnica si basa sul riconoscimento delle immagini e può riconoscere la sezione dello schermo con la quale deve interagire. Così facendo viene aumentata la robustezza alla variazione delle coordinate prefissate e i dettagli dell'interfaccia utente. La VGT permette di automatizzare i test su qualsiasi applicazione in qualsiasi contesto.

L'algoritmo di riconoscimento delle immagini utilizzato da EyeAutomate è denominato "The Eye". Esso permette di localizzare le immagini su qualsiasi schermo, utilizzando tutta la CPU disponibile: questo meccanismo consente di aumentare la sua velocità di esecuzione e la sua efficienza.
Gli elementi necessari all'esecuzione di un test visuale sono: script, immagini, dati, widget e comandi. Sono interamente contenuti in un file compresso con estensione ".eye".
Gli script possono essere creati utilizzando un semplice editor di testo, inserendo per ogni riga un comando parametrizzato con diversi elementi. Tramite l'editor EyeStudio è possibile avere un quadro completo di ciò che compone uno script visuale, ovvero script (.txt), immagini (.png), dati (.csv) e widget (.wid).
Le immagini rappresentano l'area sullo schermo da individuare e con la quale interagire e sostituiscono le coordinate del punto esatto; vengono raccolte tramite il Recorder o importate manualmente dall'esterno. Di default, l'area selezionata corrisponde ad una sezione circolare al centro dell'immagine e la sua corrispondenza verrà verificata con una percentuale di correttezza del 95%.
Il percorso delle immagini viene considerato di default come relativo alla cartella di installazione di EyeAutomate; tuttavia può essere specificato un qualsiasi percorso assoluto.
Un esempio di esecuzione di uno script visuale viene mostrato in Figura 2.4, nella quale si eseguono una serie di comandi di click, inserimento e verifica. Le immagini vengono localizzate tramite "The Eye", il quale selezionerà la sezione visibile con più somiglianza. Qualora dovessero essere presenti più aree che si abbinano perfettamente, verranno ritornate in ordine di distanza dalla posizione attuale.
Alcuni comandi possono ricevere come parametro un semplice testo al posto di un'immagine.
Inoltre, The Eye supporta il riconoscimento attraverso piccoli screenshot: l'immagine viene analizzata e scomposta in piccole aree.
Un obiettivo viene confermato quando viene individuata una determinata percentuale dell'area in analisi.
EyeAutomate supporta anche il riconoscimento basato su vettori. Questa tecnica è molto meno sensibile ai dettagli di rendering, dunque può in diversi contesti con un livello di dettaglio più ampio.
I widget sono script che contengono liste di proprietà, che possono rappresentare un'immagine, una posizione o un identificativo per localizzare un target sullo schermo. I comandi di questi script vengono eseguiti sequenzialmente finché non

Figura 2.4. Esempio di sequenza di comandi di EyeAutomate

viene localizzato l'obiettivo, e non verrà interrotto finché un qualsiasi comando non fallisce.
Uno script visuale può essere suddiviso in "step", per due ragioni fondamentali: renderne più semplice la lettura e renderlo uno script semi-automatico. Iniziano e finiscono rispettivamente con i comandi "Begin" ed "End", e possono essere annidati in una sorta di gerarchia.
L'editor principale per la creazione e la gestione di script e la cattura delle immagini è EyeStudio, così come per il lancio degli script.
Per risvolti pratici, è stata considerata utile la possibilità di eseguire uno o più script tramite riga di comando da terminale. Per fare ciò, è necessario specificare il file "EyeAutomate.jar", che si incarica dell'esecuzione dello script, ed il percorso del file di testo da lanciare:

java -jar EyeAutomate.jar scripts\script.txt

Inoltre, la presenza dei file di log ha dato la possibilità di poter verificare l'esito dei test eseguiti. In particolare, sono presenti tre tipi di file che racchiudono queste informazioni:

- Execution Log: tutti i comandi eseguiti sono registrati nel file "execution_log.txt".

In caso di fallimento, verrà registrato un messaggio di errore.

- Test Log: nel file "test_log.txt" sono registrati tutti gli script assieme a data e ora nei quali gli script sono stati lanciati, completati o falliti.
- Statistiche di esecuzione: EyeStudio registra le statistiche dei test in esecuzione nei file "test_history.csv" e "test_steps.csv".

Un'altra caratteristica fondamentale di EyeAutomate è l'API Java. Questo ha consentito di creare degli script tramite questo linguaggio di programmazione per permettere una maggiore flessibilità nell'utilizzo di questo strumento. Il progetto nel quale verranno inserite le classi Java con all'interno i test creati tramite API di EyeAutomate devono essere provvisti della libreria "EyeAutomate.jar", necessaria per la loro esecuzione.

2.5.2 SikuliX

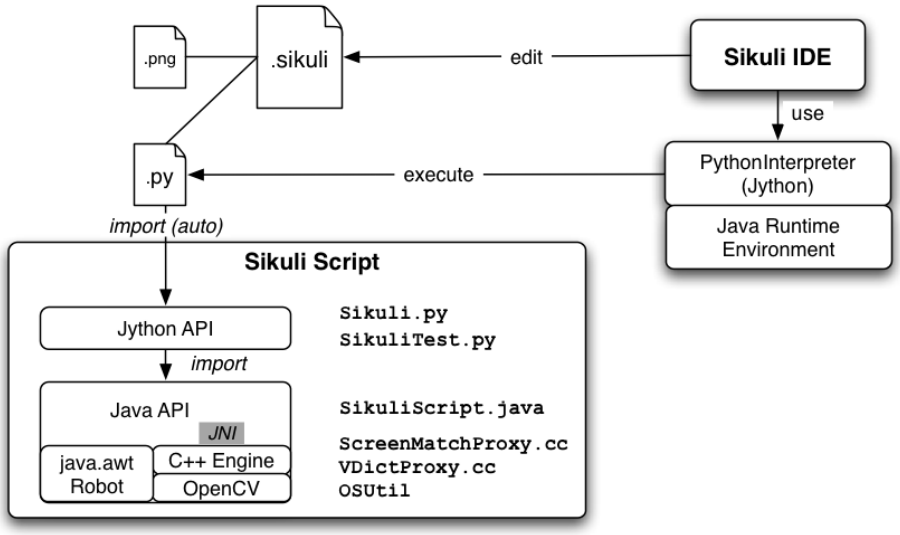
Come è stato già discusso, sono stati sviluppati diversi strumenti per l'automazione di operazioni che implicano interazioni con un'interfaccia grafica. Sono tre le categorie principali:

- La prima si basa sulla registrazione delle azioni effettuate tramite mouse e tastiera. Dopo aver effettuato la registrazione, è possibile riprodurla così com'è o aggiungere ulteriori operazioni o elementi.
- La seconda comprende gli strumenti che presentano una conoscenza approfondita della struttura GUI sulla quale eseguire le operazioni, degli elementi e delle caratteristiche che la compongono.
- La terza riconosce le immagini presenti sullo schermo e simula le azioni di mouse e tastiera per interagire con esse.

SikuliX [16] [14] rientra nella tipologia What You See Is What You Script (WYSIWYS). Si basa sul riconoscimento delle immagini sullo schermo con le quali simula le interazioni, dunque si può catalogare nella terza categoria. Per utilizzarlo, bisogna catturare le immagini sulle quali eseguire le operazioni di click e di inserimento.
Tramite l'utilizzo dell'IDE di SikuliX è possibile gestire il flusso di operazioni visuali, organizzare e catturare le immagini sulle quali eseguire le operazioni, senza la necessità di una conoscenza approfondita di un linguaggio di programmazione (escludendo i comandi di base). Tuttavia, per uno sviluppo più approfondito e ottimizzato, quest'IDE supporta i linguaggi Python, Ruby, JavaScript.
SikuliX può essere parte integrante dello sviluppo di un software, nel momento in cui sorge la necessità di eseguire dei test automatici sull'interfaccia grafica. Questo permette agli sviluppatori di testare il comportamento della GUI, soprattutto se

presenta delle complessità, per verificare che i risultati siano quelli desiderati. Ciò può essere effettuato tramite scripting regolare o tramite l'API Java che implementa le funzionalità di SikuliX.
La limitazione principale di questo strumento è la stretta dipendenza dalle dimensioni in pixel delle immagini, seppur con un minimo di tolleranza: SikuliX fallisce se la sezione sulla quale interagire non corrisponde all'immagine utilizzata per riconoscerla. Sorge così la necessità di avere diversi insiemi di immagini a seconda del contesto nel quale gli script vengono eseguiti, in termini di dimensione e risoluzione. Uno script Sikuli è una libreria Jython e Java che utilizza le immagini per

Figura 2.5. Ciclo di vita dell'esecuzione di uno script tramite Sikuli



generare eventi mouse e tastiera per automatizzare le interazioni con l'interfaccia grafica. La libreria Java è la parte fondamentale dello script Sikuli ed è composta da java.awt.Robot, che gestisce gli eventi di mouse e tastiera per posizionarli nella regione corretta, e da un componente C++ basato su OpenCV, che si incarica del riconoscimento delle aree sullo schermo attraverso le immagini inserite in ingresso. Questi due componenti interagiscono tramite l'interfaccia JNI, che deve essere compilata su ogni piattaforma. Il set di comandi messi a disposizione per l'utente per interagire con questa libreria è fornito da un livello Jython sovrastante.
Lo script si trova sotto forma di cartella sorgente e contiene il file sorgente in Python (.py) nel quale si trova la sequenza di operazioni automatizzate e rappresenta

il test case, e tutte le immagini (.png) utilizzate da file sorgente.
Lo script sotto forma di file compresso (.skl) contiene tutti i file all'interno della cartella .sikuli ed è utile per la distribuzione tramite web o mail, ma può essere anche eseguito tramite riga di comando o tramite l'IDE Sikuli.
Per essere eseguito, lo script Sikuli necessita di uno schermo reale su cui viene visualizzata l'applicazione sulla quale effettuare i test o un dispositivo che possa emulare questo contesto.
Un esempio di script SikuliX è mostrato in Figura 2.6. Un'immagine (preferibil-

Figura 2.6. Esempio di sequenza di comandi di SikuliX

mente .png) è composta da un insieme di pixel ed è catturata dallo schermo tramite screenshot, che può essere effettuato dall'IDE o tramite le caratteristiche apposite di SikuliX.
Il caricamento e lo storage delle immagini viene effettuato nella cartella .sikuli assieme allo script file, nominato con lo stesso nome della cartella. Tuttavia, è supportata anche una lista di locazioni che funge da percorso dell'immagine, che verrà controllata quando si presenta la necessità di caricare l'immagine per verificarne l'esistenza.
Per il riconoscimento delle immagini, SikuliX utilizza OpenCV, ed in particolare

il metodo matchTemplate(). Questo metodo opera su un target da ricercare, posizionato su una base, che consiste in un'immagine di dimesioni pari o maggiori. Per implementare tutto ciò, internamente viene definita una Region con una determinata dimensione. Essa verrà utilizzata per identificare la regione dello schermo della quale verrà catturato un screenshot, utilizzando la classe Java Robot, che verrà utilizzato come base e sarà tenuto in memoria. Il target verrà creato dalle immagini presenti come risorse e sarà anch'esso tenuto in memoria. Entrambi vengono convertiti in oggetti OpenCV (CVMat). A questo punto viene eseguito il metodo matchTemplate() e viene costruita una matrice della dimensione dell'immagine di base, che contiene, per ogni pixel, un valore compreso tra 0.0 e 1.0 che corrisponde al livello di similitudine ottenuto confrontando quelli dell'immagine base con quelli del target a partire dall'angolo in alto a sinistra. La probabilità che il target sia contenuto in quell'area è direttamente proporzionale a questo valore. I risultati superiori e 0.7 vengono considerati nel corso dell'analisi, tutti gli altri generano un'eccezione FindFailed. Successivamente verrà creato un oggetto che rappresenta l'area dello schermo che presenta un'alta probabilità di contenere il target. Se i risultati dovessero essere non accettabili, l'immagine risulta non trovata e viene presa una decisione tra: iniziare una nuova ricerca all'interno di un altro screenshot o segnalare l'errore e terminare la ricerca. Questa operazione termina, in ogni caso, dopo un'attesa di default di 3 secondi se l'immagine non viene trovata, restituendo il risultato FindFailed.
Il tempo impiegato a trovare l'immagine è direttamente proporzionale alla grandezza dell'immagine di base e alla differenza tra la base ed il target. Per velocizzare le operazioni di ricerca, dunque, è consigliato ridurre l'area in analisi.
Per velocizzare le operazioni di testing, SikuliX implementa anche un sistema che verifica che l'immagine sia ancora nella stessa posizione rilevata nell'ultima ricerca, che nel caso di successo, permette di saltare operazioni che richiederebbero alcuni millisecondi.
Come già analizzato per il tool EyeAutomate, una caratteristica che risulta indispensabile è la possibilità di generare un file di Log contenente i messaggi che rappresentano le operazioni effettuate dal Sikuli ed i rispettivi risultati. Per scrivere questi messaggi viene utilizzata la funzione Debug.user("..."), che accetta come parametro in ingresso una stringa definita secondo la definizione Java. La linea del log, dunque, sarà composta dal timestamp, che indica la data e l'orario nei quali si è verificato un determinato evento e la stringa in ingresso. Questo meccanismo è utile nel momento in cui si vuole effettuare il debug dello script, soprattutto nel caso in cui le operazioni falliscano, così come per informare l'utente finale degli esiti dei test.
Sikulix è stato progettato e scritto in Java e mette a disposizione un'API che può essere integrata come libreria Java in programmi di terze parti per utilizzare le sue funzionalità. Ciò ha permesso di realizzare un software che generasse degli script nativi per Sikuli da eseguire tramite riga di comando, ma anche degli script che

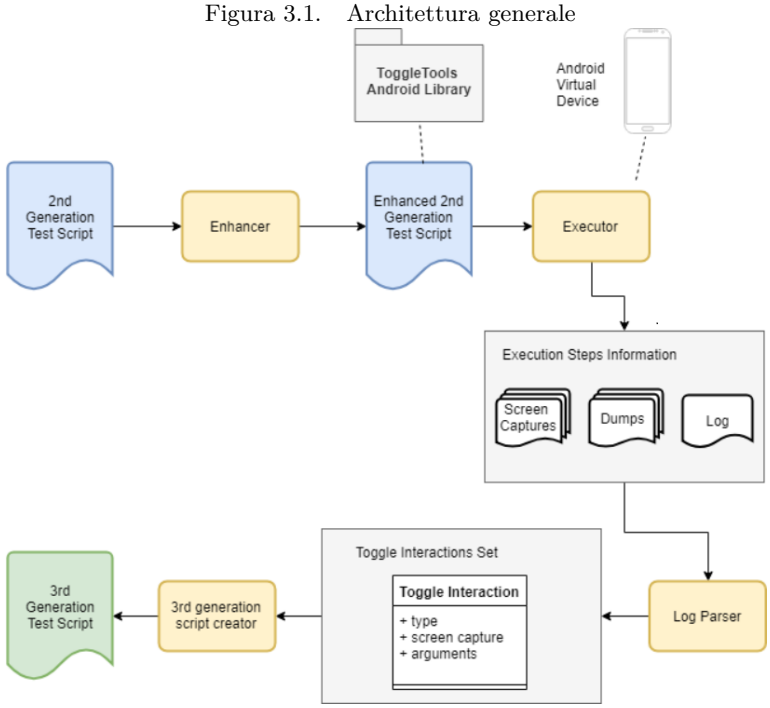
fondono l'API SikuliX e tutte le sue caratteristiche con la potenza e la portabilità di Java. Inoltre, questi script vengono inseriti all'interno di progetti ed eseguiti tramite riga di comando, compilandoli per generare i file .class utilizzando come parametro d'ingresso la libreria sikulix.jar.

Capitolo 3

Architettura e Design

La struttura del progetto è composta da diversi componenti che permettono la traduzione di test di seconda generazione in test di terza generazione. Il diagramma riportato in Figura 3.1 rappresenta l'architettura generale. Il processo inizia con

Figura 3.1. Architettura generale



un'analisi degli script, che vengono modificati integrando gli strumenti necessari per la generazione di interazioni con la GUI.
Le interazioni con l'interfaccia grafica comprendono tutte le possibili azioni che l'utente effettua sul layout dell'applicazione. Le operazioni possibili possono essere semplici, come le varianti dei click e l'inserimento di testo, o più complicate, ad esempio tutte le possibili combinazioni di swipe.
Una tabella con le interazioni è mostrata in Figura 3.2. In questo capitolo verranno

Figura 3.2. Tabella delle interazioni

Type of Espresso interaction	Arguments
Click	None
TypeText	i. Text to type
LongClick	None
ReplaceText	i. Text length; ii. Text replacement
ClearText	i. Text length
DoubleClick	None

analizzati tutti gli elementi che compongono l'architettura generale.

3.1 Enhancer

L'Enhancer è il componente iniziale della catena che si occupa di modificare ed arricchire gli script di seconda generazione per inserire al loro interno gli strumenti che serviranno alla generazione delle interazioni con la GUI. I test Espresso sono composti da una serie di operazioni effettuate sui differenti elementi dell'interfaccia grafica dell'applicazione e da un controllo finale che verifica l'effettivo funzionamento del test. Modificandoli è dunque possibile stampare in un log le informazioni su queste interazioni, indicando di che test e tipo di interazione si tratti. Successivamente, viene effettuato un dump, che è una struttura che descrive la gerarchia dell'Activity corrente. Ogni view della gerarchia visuale è identificata da un nodo nel file .xml e attraverso i suoi attributi. Un esempio di dump è mostrato in Figura 3.3. Questo permetterà, in seguito, di generare i test di terza generazione partendo dalle informazioni presenti sul dispositivo e nel log, ottenute dall'esecuzione dei test arricchiti. Une esempio di log è mostrato in Figura 3.4.

3.2 Executor

L'Executor si occupa dell'interazione con il dispositivo emulato (AVD). Tramite l'AVD Manager è possibile crearne uno nuovo, scegliendo una API e una tra le skin disponibili, alla quale sono associate una specifica risoluzione e dimensione.

Figura 3.3. Esempio di dump

```
<node bounds="[0,1269][1080,1794]" visible-to-
    user="true" selected="false" password="false" long-
    clickable="false" scrollable="false" focused="false"
    focusable="false" enabled="true" clickable="false"
    checked="false" checkable="false" content-desc=""
    package="it.feio.android.omninotes.foss"
    class="android.view.ViewGroup" resource-
    id="it.feio.android.omninotes.foss:id/snackbar_placeholder"
    text="" index="2"/>
<node bounds="[626,957][1059,1773]" visible-to-
user="true" selected="false" password="false" long-
clickable="false" scrollable="false" focused="false"
focusable="false" enabled="true" clickable="false"
checked="false" checkable="false" content-desc=""
package="it.feio.android.omninotes.foss"
class="android.view.ViewGroup" resource-
id="it.feio.android.omninotes.foss:id/fab" text=""
index="3">
    <node bounds="[865,1579][1059,1773]" visible-to-
        user="true" selected="false" password="false" long-
        clickable="true" scrollable="false" focused="false"
        focusable="true" enabled="true" clickable="true"
        checked="false" checkable="false" content-desc=""
        package="it.feio.android.omninotes.foss"
        class="android.widget.ImageButton" resource-
        id="it.feio.android.omninotes.foss:id/fab_expand_menu_button"
        text="" index="6" NAF="true"/>
```

Figura 3.4. Esempio di log

```
10-24 15:01:04.933 16495 16510 D TOGGLELOG: 1540393253715, id, fab_expand_menu_button, click,
10-24 15:00:55.483 16495 16510 D TOGGLELOG: 1540393255097, text, Text note, click,
10-24 15:00:58.580 16495 16510 D TOGGLELOG: 1540393258578, id, detail_title, typetext, Test
10-24 15:01:01.538 16495 16510 D TOGGLELOG: 1540393261538, content-desc, drawer open, click,
10-24 15:01:04.933 16495 16510 D TOGGLELOG: 1540393264932, content-desc, drawer open, click,
10-24 15:01:08.200 16495 16510 D TOGGLELOG: 1540393268199, id, settings_view, click,
```

Successivamente, nell'interfaccia principale è possibile scegliere e lanciare il nuovo dispositivo creato o uno già esistente. Dopo la selezione di un progetto Android contenente le suite di test Espresso, tramite i rispettivi pulsanti, è possibile scegliere tra l'esecuzione di un'intera suite o di un test in particolare, l'enhance della classe, la traduzione da seconda a terza generazione ed il lancio degli script generati. Ogni operazione termina con la visualizzazione dell'output generato: verrà notificato il fallimento di un qualsiasi step intermedio, ad esempio il build del progetto durante la fase di verifica di una suite di test, o la corretta esecuzione dell'operazione in corso.

3.3 Log Parser

Il Log Parser è l'elemento che si occupa della raccolta delle informazioni necessarie per la traduzione dei test e la creazione degli script di terza generazione. Come risultato dello step 1. avremo il test enhanced, nel quale sono saranno state aggiunte le chiamate alla libreria Toggle, anch'essa inserita all'interno del progetto Android di cui fa parte la suite di test. Questa libreria viene utilizzata per ottenere gli snapshot della schermata, salvati come .bmp, e le informazioni sui widget, salvati come .xml, entrambi nella memoria esterna del dispositivo emulato. A questo punto, il Parser prenderà dagli xml le informazioni riguardanti i widget con cui è avvenuta un'interazione, come ad esempio coordinate, altezza e larghezza; questi dati verranno utilizzati per modificare e ritagliare le schermate in bmp. Il risultato di questa operazione sarà dunque una serie di immagini png che rappresentano i vari elementi della GUI dell'applicazione su cui è stato effettuato il test e con cui c'è stata un'interazione. Queste immagini verranno utilizzate nello step successivo per generare gli script di terza generazione. Alla fine di questa fase avremo una serie di informazioni riguardanti il tipo di interazione e l'immagine dell'elemento con il quale è avvenuta.

3.4 Creatore di Script di Terza Generazione

Questo componente ci permette di utilizzare di utilizzare i dati ottenuti dal Log Parser per creare gli script di terza generazione. È possibile generare script di Eyeautomate o Sikuli, a seconda della scelta dello sviluppatore che utilizza il tool, selezionando le rispettive opzioni di traduzione. Questi script vengono creati con l'estensione apposita secondo il tipo di traduzione e vengono dunque riconosciuti ed eseguiti dagli IDE dei due tool, EyeStudio e SikuliX. In più, nella GUI è stato integrato un frame attraverso il quale l'utente può selezionare il path della cartella nella quale gli script sono stati generati ed i path delle API di Eyeautomate o Sikuli, così da poterli eseguire e verificare attraverso un dispositivo emulato connesso. Poiché entrambi i tool comprendono un'API Java, è disponibile una traduzione che genera test basati sul linguaggio Java. Naturalmente, questi test consentono al programmatore di integrare funzionalità aggiuntive che non era possibile ottenere tramite gli script originali. Inoltre, è possibile tradurre i test in script che presentano una combinazione tra Eyeautomate e Sikuli. In particolare, sono presenti due casi differenti: il primo, in cui vengono eseguiti i comandi di Eyeautomate e, in caso questo fallisca, viene eseguito Sikuli; il secondo, in cui, viceversa, viene eseguito prima Sikuli e successivamente, in caso di fallimento, Eyeautomate.

3.5 Esempio di esecuzione

Al fine di rendere più chiara l'esposizione dei componenti dell'architettura e delle loro interazioni, verrà presentato un esempio di esecuzione che mostrerà le varie fasi di traduzione di un test case Espresso.

1. Test Espresso

```
@Test
public void testTest() {

    onView(withId(R.id.fab_expand_menu_button)).perform(click());

    onView(withText("Text note")).perform(click());

}
```

2. Test Espresso Enhanced

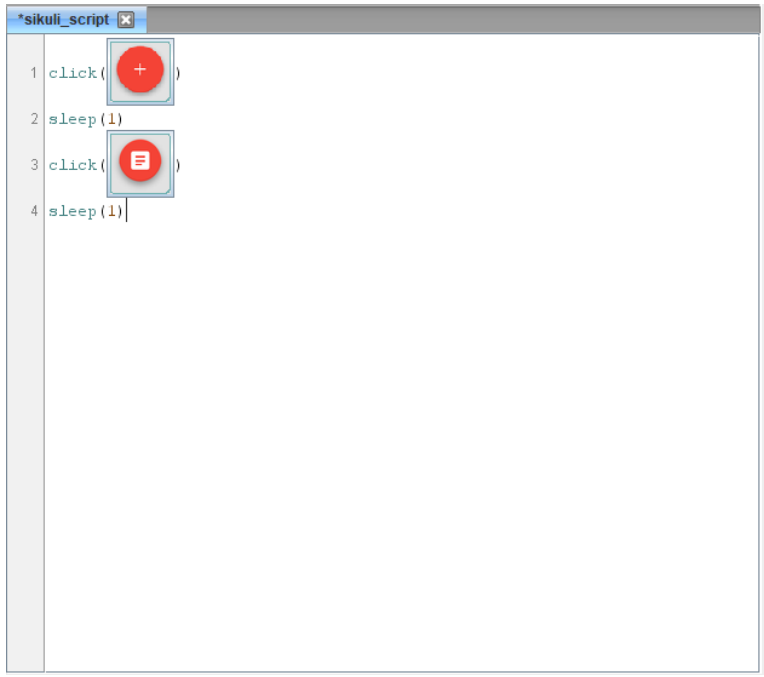
```
@Test
public void testTest() {

    Instrumentation instr = InstrumentationRegistry.getInstrumentation();
    UiDevice device = UiDevice.getInstance(instr);

    Date now = new Date();
    Activity activity = getActivityInstance();
    Log.d( tag: "touchtest",  msg: now.getTime() + ", " + "id" + ", " +
            "fab_expand_menu_button" + ", " + "click" + "," + "");
    TOGGLETools.TakeScreenCapture(now, activity);
    TOGGLETools.DumpScreen(now, device);

    onView(withId(R.id.fab_expand_menu_button)).perform(click());

    try {
        Thread.sleep( millis: 2000);
    } catch (Exception e) {

    }
    now = new Date();
    activity = getActivityInstance();
    TOGGLETools.TakeScreenCapture(now, activity);
    TOGGLETools.DumpScreen(now, device);
    Log.d( tag: "touchtest",  msg: now.getTime() + ", " + "text" + ", "
            + "Text note" + ", " + "click" + "," + "");


    onView(withText("Text note")).perform(click());

}
```

3. Dump
4. Log

```
<node bounds="[0,1269][1080,1794]" visible-to-
    user="true" selected="false" password="false" long-
    clickable="false" scrollable="false" focused="false"
    focusable="false" enabled="true" clickable="false"
    checked="false" checkable="false" content-desc=""
    package="it.feio.android.omninotes.foss"
    class="android.view.ViewGroup" resource-
    id="it.feio.android.omninotes.foss:id/snackbar_placeholder"
    text="" index="2"/>
<node bounds="[626,957][1059,1773]" visible-to-
user="true" selected="false" password="false" long-
clickable="false" scrollable="false" focused="false"
focusable="false" enabled="true" clickable="false"
checked="false" checkable="false" content-desc=""
package="it.feio.android.omninotes.foss"
class="android.view.ViewGroup" resource-
id="it.feio.android.omninotes.foss:id/fab" text=""
index="3">
    <node bounds="[865,1579][1059,1773]" visible-to-
        user="true" selected="false" password="false" long-
        clickable="true" scrollable="false" focused="false"
        focusable="true" enabled="true" clickable="true"
        checked="false" checkable="false" content-desc=""
        package="it.feio.android.omninotes.foss"
        class="android.widget.ImageButton" resource-
        id="it.feio.android.omninotes.foss:id/fab_expand_menu_button"
        text="" index="6" NAF="true"/>
```

```
10-24 15:01:04.933 16495 16510 D TOGGLELOG: 1540393253715, id, fab_expand_menu_button, click,
10-24 15:00:55.483 16495 16510 D TOGGLELOG: 1540393255097, text, Text note, click,
10-24 15:00:58.580 16495 16510 D TOGGLELOG: 1540393258578, id, detail_title, typetext, Test
10-24 15:01:01.538 16495 16510 D TOGGLELOG: 1540393261538, content-desc, drawer open, click,
10-24 15:01:04.933 16495 16510 D TOGGLELOG: 1540393264932, content-desc, drawer open, click,
10-24 15:01:08.200 16495 16510 D TOGGLELOG: 1540393268199, id, settings_view, click,
```

5. Script visuale tradotto


*sikuli_script
1 click( )
2 sleep(1)
3 click( )
4 sleep(1)

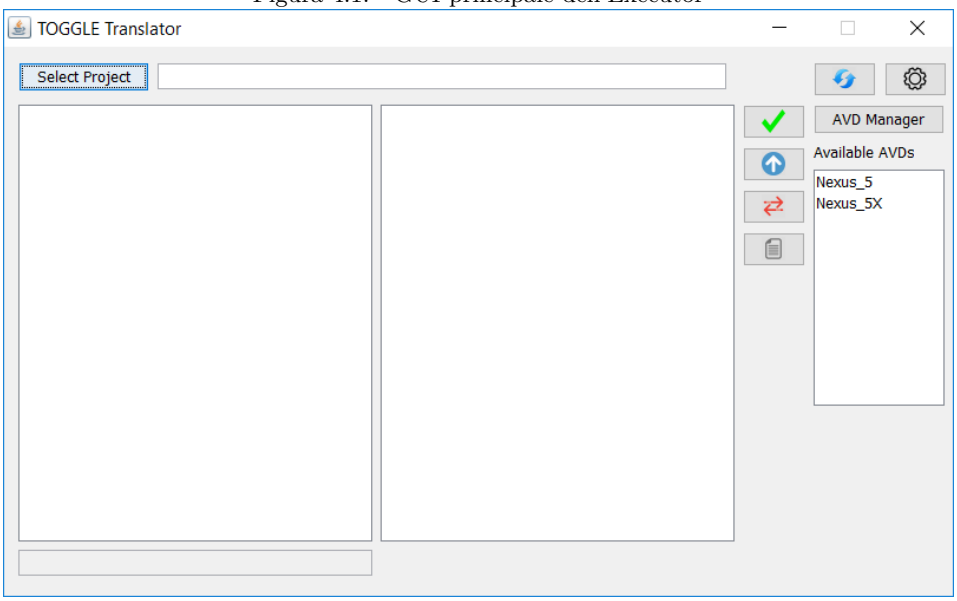
Capitolo 4

Executor

In questo capitolo ci concentreremo sulla struttura e le scelte di design dell'Executor. Questo componente si occupa della creazione di nuovi dispositivi emulati (AVD) e delle interazioni con essi, ed unisce in un'unica interfaccia tutte le operazioni di lancio e verifica delle suite di test, enhance e traduzione dei test, generazione ed esecuzione degli script di terza generazione. Il linguaggio di programmazione utilizzato per la realizzazione del tool è Java. Java è un linguaggio object-oriented, che permette di scrivere software indipendenti dalla piattaforma hardware su cui vengono eseguiti. Infatti, Java si basa sull'omonima piattaforma software, che si comporta da ambiente di esecuzione nel quale è possibile scrivere ed eseguire programmi indipendentemente dall'hardware sottostante. Questa piattaforma è composta da due elementi: la Java Virtual Machine (JVM) e le API Java. La JVM è il componente che rende possibile la virtualizzazione dell'hardware, creando una sorta di macchina alternativa che funge da calcolatore per l'esecuzione del programma. Le API sono un insieme di componenti software, anche chiamate librerie, che si interfacciano con la JVM e hanno il compito di eseguire molteplici operazioni. Poiché l'Executor si basa su questo linguaggio di programmazione, il primo prerequisito per la sua esecuzione è la presenza della piattaforma Java.
La struttura dell'Executor si divide in due macro-sezioni: l'interfaccia grafica (GUI) e la parte logica.

4.1 Interfaccia Grafica (GUI)

Figura 4.1. GUI principale dell'Executor



La GUI del software è stata interamente sviluppata tramite Java Swing, un framework di Java per la creazione di interfacce grafiche orientato ai componenti. Un componente è un oggetto con determinate caratteristiche di comportamento che emette eventi in modo asincrono. Un componente swing viene rappresentato graficamente tramite una combinazione di elementi, ad esempio bordi, colori, scritte, che è possibile modificare. Inoltre, è possibile catturare gli eventi emessi da ogni singolo componente, in modo da intercettare e gestire le interazioni dell'utente con l'interfaccia grafica. Swing segue anche il principio di indipendenza di Java, poiché si basa sulla stessa piattaforma software; non necessita i controlli della GUI del SO nativi per la rappresentazione, ma utilizza le API 2D di Java. I componenti Swing sono completamente indipendenti e possono essere disegnati e realizzati in qualunque modo utilizzando le librerie di Java 2D poiché non devono necessariamente avere un corrispettivo tra i componenti nativi del SO.
L'interfaccia grafica, come mostrato in Figura 4.1, è composta da una serie di widget che si dispongono in tre sezioni verticali. Nella prima, è presente un pulsante che apre un "file chooser" attraverso il quale viene scelto il progetto Android di partenza. Al momento della selezione, viene verificato che il folder selezionato sia

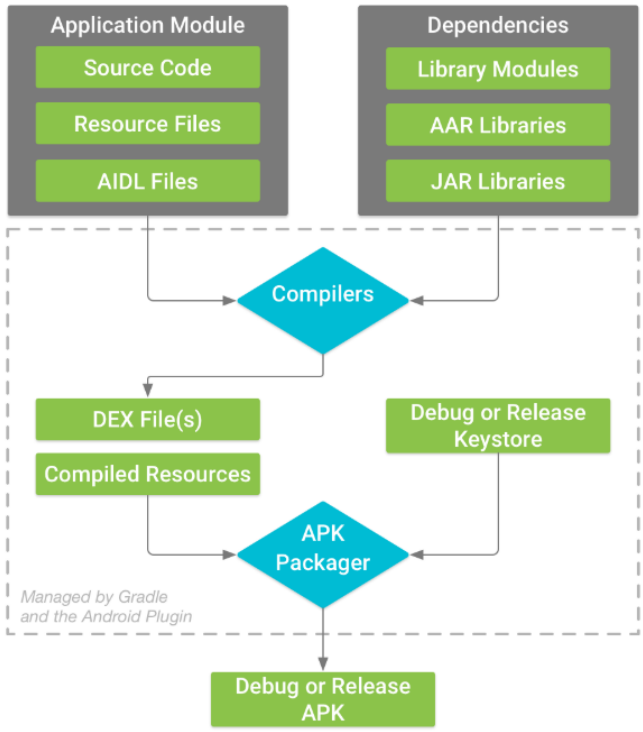
effettivamente un progetto Android valido, controllando che siano presenti i file "gradle" e che sia presente un file "Manifest" in formato XML. Al termine del caricamento del progetto, viene visualizzato il path nella casella di testo posizionata a destra del pulsante. La sezione sottostante è composta da due aree di testo. Nella prima vengono elencati tutte le suite di test Espresso rilevate all'interno di tutti i sub-folder. Le suite vengono individuate tramite una ricerca ricorsiva di tutte le classi .Java; una volta trovate, viene effettuata un'analisi testuale che conferma che il file sia effettivamente una suite di Espresso grazie alla presenza di import specifici e di parole chiave che precedono i metodi stessi. Nella seconda area di testo vengono visualizzati i singoli test presenti nella suite selezionata nella prima area. La seconda sezione è composta da quattro pulsanti che gestiscono le principali operazioni consentite dall'Executor: verifica dei test, enhance della classe, traduzione da seconda a terza generazione e lancio degli script generati. La verifica dei test consiste nell'esecuzione dell'intera classe o del singolo test selezionato sul dispositivo emulato (AVD) connesso. La verifica del test si conclude mostrando all'utente il risultato, sia in caso di successo, sia in caso di fallimento. L'enhance della suite di test consiste nella generazione di una nuova classe che comprende tutti i test presenti nella classe selezionata ai quali vengono aggiunti gli strumenti necessari alla traduzione. La traduzione può essere eseguita solamente sui test su cui è stato già effettuato l'enhance: si basa sull'esecuzione della classe enhanced corrispondente alla suite selezionata, al termine della quale verranno generati gli script tradotti. Il pulsante che gestisce il lancio degli script apre un nuovo frame, attraverso il quale è possibile selezionare il path degli script generati e quale tipo di esecuzione effettuare. Tutte le operazione principali verranno descritte dettagliatamente nel paragrafo successivo.
La terza sezione è la parte che gestisce la creazione ed il lancio del dispositivo emulato. In alto sono presenti due pulsanti: il primo serve ad effettuare un "refresh" della schermata principale, in caso ci siano state delle modifiche nel progetto Android attualmente caricato, ad esempio aggiunta o rimozione di suite di test; il secondo gestisce le impostazioni di traduzione, ovvero la selezione del path nel quale gli script tradotti verranno generati, i tipi di traduzione da effettuare (script nativi di Eyeautomate o Sikuli, classi Java di Eyeautomate o Sikuli, classi Java combinate di Eyeautomate e Sikuli) e la possibilità di effettuare un "clear launch", ovvero il reset completo dell'applicazione prima dell'esecuzione di ogni test. Un terzo pulsante apre un frame che gestisce la creazione di un nuovo AVD. L'AVD manager è composto da una tabella, nella quale sono presenti tutte le "skin" disponibili, ovvero l'aspetto che avrà il dispositivo emulato, disponibili e le relative dimensione, risoluzione e densità di pixel, da una casella di testo nella quale è possibile inserire o modificare il nome del dispositivo da creare (di default, verrà utilizzato il nome presente nella colonna "Name" della skin selezionata), da una sezione in cui è possibile scegliere una tra le API rilevate già installate nella macchina su cui l'Executor è stato eseguito, ed un pulsante che creerà il nuovo dispositivo

emulato combinando tutte le scelte selezionate. Infine, è presente una sezione nella quale sono elencati tutti gli AVD già creati e presenti nella macchina. Da qui è possibile lanciare il dispositivo che corrisponde al nome selezionato, sul quale successivamente verranno eseguiti i test.

4.2 Funzionalità

In questo paragrafo verranno descritte dettagliatamente le varie funzionalità di cui si occupa l'Executor, analizzando le scelte di implementazione da un punto di vista tecnico ed esecutivo.
La parte logica dell'Executor è stata interamente realizzata in Java. Ciò ha permesso di creare un tool indipendente dalla piattaforma hardware, eseguibile su qualsiasi macchina che presenti la Java Platform.
In breve, l'Executor ha il compito di creare un collegamento con l'emulatore connesso, attraverso l'Android Bridge Debug, per effettuare le operazioni di verifica di test Espresso, enhance delle suite di test e traduzione, che si conclude con la creazione degli script terza generazione. Essendo un tool basato su Java e che si interfaccia con il sistema Android, sono necessari alcuni prerequisiti per garantire il corretto funzionamento. In particolare, deve essere presente sulla macchina la piattaforma Java, con annessa installazione della JVM; inoltre, deve essere impostata la variabile d'ambiente che punti al path dove è collocata la JDK installata, in particolare alla sottocartella "bin". Naturalmente, dovrà essere presente anche una versione stabile di Android, tramite cui sarà possibile interfacciarsi all'AVD e impostare un ambiente di testing operativo.

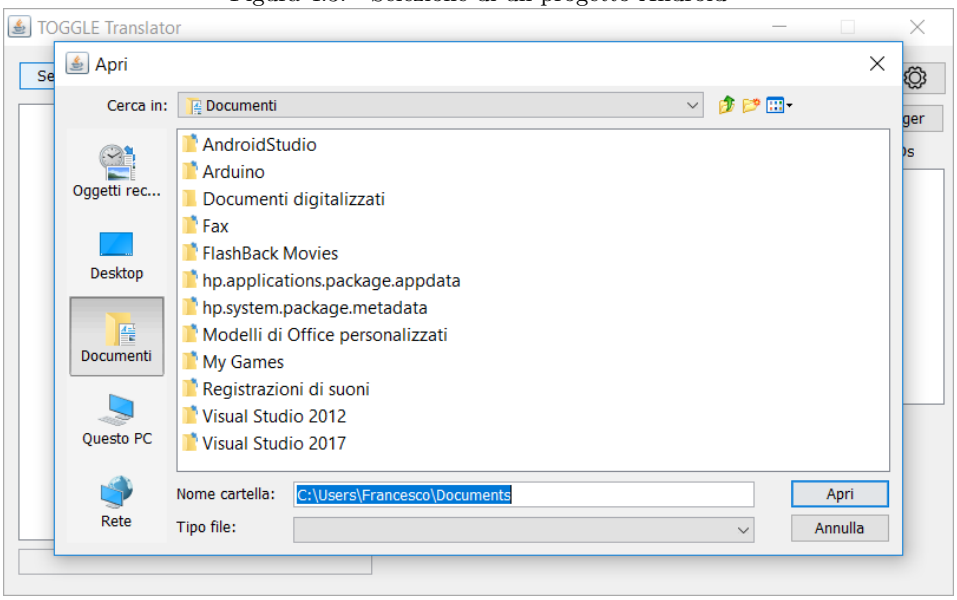
Figura 4.2. Processo di build di un'APK



4.2.1 Caricamento del progetto

La prima operazione da effettuare una volta lanciato il tool è il caricamento del progetto Android dal quale ricavare le classi di test. Per rendere possibile ciò, è presente un pulsante che apre un File Chooser, come mostrato in Figura 4.3, attraverso il quale verrà scelto il folder principale del progetto. Una volta selezionato, verrà effettuata una ricerca ricorsiva all'interno di esso, per garantire che ciò che è stato scelto sia un progetto Android valido. In particolare, dovrà soddisfare due requisiti: la presenza dei file di gradle e la presenza del Manifest. Gradle è un sistema open source per l'automazione dello sviluppo fondato sulle idee di Apache Ant e Apache Maven. Esso viene utilizzato nel contesto di Android per eseguire il build del progetto, così da poter successivamente generare le APK da installare sul dispositivo. Il flow chart del processo di build di un progetto Android è rappresentato in Figura 4.2.
Una volta che il progetto è stato selezionato e verificato, viene effettuata una ricerca all'interno di esso per individuare le suite di test Espresso. Ricorsivamente,

Figura 4.3. Selezione di un progetto Android



vengono analizzati tutti i file dentro la cartella nella quale sono contenuti i sorgenti delle classi di test, in particolare vengono individuate alcune parole chiave presenti unicamente nei test Espresso, ad esempio particolari "import" o "annotations" che precedono i metodi stessi. Una volta che sono state individuate tutte le suite di test ed i relativi metodi, il path del progetto viene visualizzato nella casella di testo lateralmente al pulsante di selezione, mentre i nomi delle classi vengono visualizzati negli spazi sottostanti. A questo punto, è possibile selezionare le differenti suite di test: in un'apposita sezione, vengono visualizzati i nomi di tutti i test presenti nella classe selezionata. Eseguendo la verifica, se è stato selezionato un test in particolare verrà verificato solamente quello; viceversa, se nessun test è stato selezionato, verrà verificata l'intera classe di test, eseguendo in ordine la verifica di tutti i test a partire dal primo.

4.2.2 Android Debug Bridge (adb)

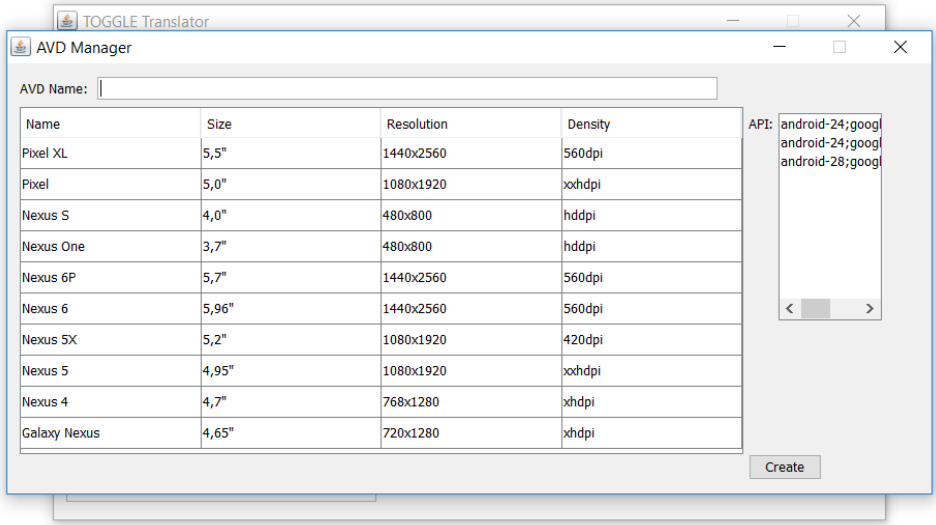
La più importante funzionalità dell'Executor è sicuramente la sua interazione con l'AVD connesso. Prima di analizzare tutti gli aspetti di queste interazioni, che rappresentano il punto di partenza per tutte le altre operazioni, bisogna introdurre l'elemento che sta alla base delle connessioni con l'AVD: l'Android Debug Bridge

(ADB). L'Android Debug Bridge (ADB) [18] è uno strumento command-line che consente di comunicare con un dispositivo emulato e permette diverse operazioni, ad esempio installare applicazioni ed effettuare il debug. Esso mette a disposizione una shell Unix per interagire direttamente con l'AVD. Si basa su di una struttura client-server:

- il client invia comandi ed è eseguito sulla macchina attraverso il comando adb tramite command-line
- il demone esegue i comandi sul dispositivo; viene eseguito tramite un processo in background su ciascun dispositivo
- il server che si occupa della comunicazione tra il client ed il demone; viene eseguito tramite processo in background sulla macchina

Nel momento in cui viene eseguito un client adb, se non è presente un server attivo, ne viene lanciato uno nuovo. Il server ascolta i comandi adb del client sulla porta TCP 5037, usata anche dal client per comunicare.
Ogni operazione eseguita attraverso l'adb è eseguita tramite command line. Perciò, è stato necessario utilizzare un Process Builder all'interno del codice java che lanciasse dal path corretto i comandi specifici dell'operazione da effettuare, inserendo opportunamente i parametri e le opzioni del comando.

Figura 4.4. AVD Manager



4.2.3 AVD Manager

La gestione del dispositivo emulato da parte dell'Executor si basa su due concetti principali: la creazione di un nuovo AVD e la gestione degli AVD già presenti.
La creazione di un nuovo AVD è gestita da un frame che viene aperto a partire dalla finestra principale, cliccando il pulsante "AVD Manager". Come mostrato in Figura 4.4, il frame è composto da una casella di testo attraverso cui modificare il nome del dispositivo che verrà successivamente creato, da una tabella nella quale sono presenti tutte le skin disponibili e le relative dimensioni e da una tabella con le API presenti. Il pulsante "Create" eseguirà un comando specifico che viene parametrizzato con tutte le informazioni raccolte attraverso l'interazione con il frame. Il comando in questione è il seguente:

avdmanager create avd -n ***avdName*** *-k "system-images;****api****;x86" -c* ***size***

"Avdmanager" è un tool da linea di comando di Android che permette di gestire un AVD. In questo caso, attraverso le parole chiave "create AVD" verrà generato un nuovo dispositivo emulato. Sono presenti anche diverse opzioni che parametrizzano la creazione: "**-n**" specifica il nome del dispositivo, "**AVDName**" è il nome specificato nell'apposita area di testo; "**-k**" specifica l'id dell'SDK, "API" è l'API scelta tra le opzioni presenti nell'apposita tabella, "**-c**" specifica la dimensione dell'SD card del nuovo dispositivo, espressa in KB o MB.
La gestione degli AVD già esistenti viene effettuata dal frame principale. In una lista sono presenti i nomi di tutti i dispositivi individuati sulla macchina. Cliccando due volte su uno di essi, viene richiamato un apposito process builder che esegue un comando per lanciare l'AVD selezionato:

emulator -avd ***avdName*** *-port* ***portNum***

"**emulator -avd**" permette di lanciare l'AVD specificato tramite il parametro "**avdName**"; l'opzione "**-port**" permette di specificare la porta TCP attraverso la quale verranno eseguiti i comandi tramite adb e verrà stabilita una connessione con il dispositivo emulato.
al lancio dell'AVD viene verificato che non ci sia già un dispositivo online; in tal caso l'utente verrà avvertito e non verrà effettuato il lancio.

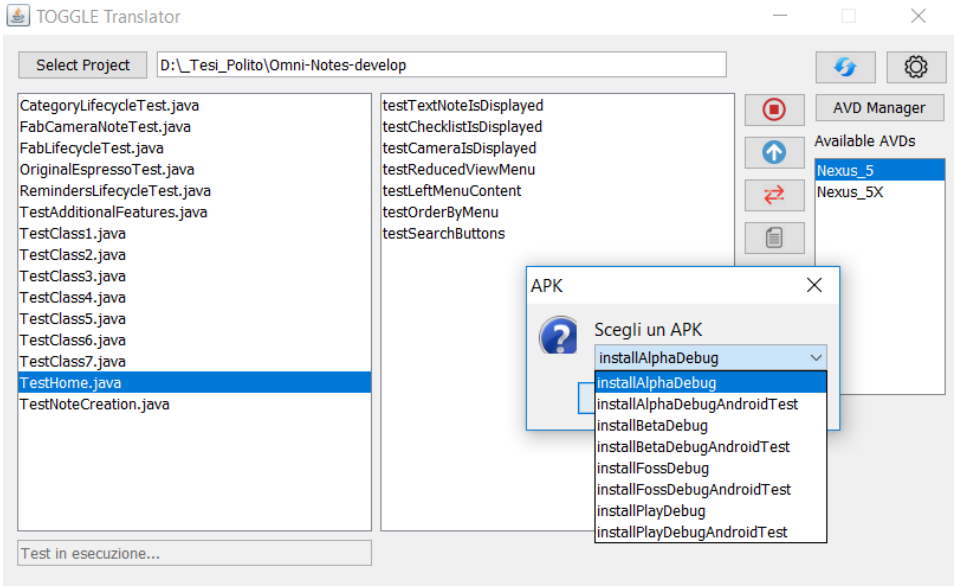
4.2.4 Verifica test

L'operazione di verifica dei test consiste nell'esecuzione di una suite di test o di un test singolo sul dispositivo emulato. Questa operazione può essere effettuata solamente dopo aver caricato un progetto Android valido. Selezionando la classe di test tra tutte quelle identificate all'interno del progetto presenti nell'apposita

area dell'interfaccia principale, ci sono due possibilità: in un'altra area della GUI è possibile scegliere il singolo test tra quelli identificati all'interno della suite selezionata ed eseguire solamente quello, oppure premere il pulsante di verifica ed eseguire in ordine tutti i test presenti all'interno della classe selezionata. La verifica verrà subito interrotta se non viene individuato nessun dispositivo emulato connesso.
Tutte le operazioni che verranno descritte in questo paragrafo, vengono eseguite tramite un Process Builder apposito: viene istanziata una nuova classe attraverso un costruttore, nel quale vengono indicati il comando da eseguire e l'argomento specifico. Per ogni builder, vengono specificati come comando "cmd.exe", ovvero la shell a riga di comando del sistema operativo, e come argomento il comando specifico da eseguire a seconda dell'operazione da effettuare. Inoltre, viene indicata la directory nella quale il builder viene eseguito, a seconda dell'argomento specificato nel costruttore: ad esempio, per eseguire i comandi "adb" verrà indicato il path della directory di Android nella quale è contenuto l'eseguibile che permette di lanciare questo comando.
Inizialmente viene eseguito un comando tramite adb, *adb start-server*, per verificare che ci sia una connessione con il dispositivo emulato e, in caso non ci fosse, stabilirne una nuova.
Successivamente, vengono eseguite alcune manipolazioni della dimensione del log di Android. La finestra di Logcat mostra determinate informazioni, che possono essere del sistema Android stesso oppure messaggi aggiunti all'applicazione tramite la classe Log. Per mostrare delle linee di output specifiche, è possibile utilizzare alcune parole chiave tramite cui il testo verrà filtrato. Questo meccanismo viene utilizzato dal componente dell'architettura del Log Parser che viene richiamato dall'Executor per effettuare la traduzione delle suite di test. Poiché le informazioni presenti nel logat, necessarie per questa operazione, richiedono una determinata dimensione per essere contenute, è necessario, prima di effettuare qualsiasi verifica o traduzione, dimensionare opportunamente il Log. Questo viene fatto attraverso un process builder che esegue tramite riga di comando il tool "logcat" per gestire il Log di Android, che verrà eseguito tramite shell adb. In particolare, viene prima verificata la dimensione effettiva attraverso il comando *adb logcat -g*; se risulta essere troppo piccola, verrà opportunamente modificata ed impostata ad un valore standard scelto opportunamente per riuscire a contenere tutte le informazioni utili. Il comando per modificare la dimensione del log è *adb logcat -G* ***logSize***: logSize è la nuova dimensione scelta, espressa in K o M.
A seguire, viene effettuato il build dell'intero progetto caricato. Questa operazione è fondamentale ed è eseguita ad ogni verifica o traduzione delle suite di test. Così facendo, se l'utente decidesse di modificare il progetto Android originale, ad esempio inserendo test aggiuntivi o rimuovendo test esistenti, ad ogni esecuzione verrà sempre installata la versione aggiornata dell'applicazione sul dispositivo emulato. Questo passaggio è molto importante anche a seguito dell'enhance delle suite di test, poiché viene generata una nuova classe contenente i test modificati e dunque

sarà necessario effettuare un build prima dell'esecuzione per assicurarsi che l'APK sia alla versione più recente.
Lo strumento che viene utilizzato per effettuare il build del progetto è gradlew. Questo tool si trova all'interno del progetto Android stesso, dunque l'Executor utilizza il path estrapolato dopo il caricamento per lanciarlo e generare le APK da installare sul dispositivo.
Il comando specifico eseguito attraverso un process builder è: *gradlew assemble*.
Il tool gradlew accetta come task diverse azioni per assemblare e generare le applicazioni. Per l'Executor è stato scelto di utilizzare l'opzione "assemble": eseguendo questo comando, verranno assemblate tutte le versioni dell'applicazione individuate nel progetto selezionato e verranno generate le rispettive APK.
Una volta eseguito con successo il build del progetto, è necessario installare le applicazioni sul dispositivo emulato per poter successivamente eseguire i test. Dunque, verrà ancora una volta utilizzato il tool gradlew tramite process builder, questa volta però verrà lanciato con l'opzione "tasks". Essa darà in output tutte le operazioni che gradle è in grado di effettuare. Tra tutte verranno individuate solamente quelle con la parola chiave "install", seguite dalla versione dell'APK da installare.

Figura 4.5. Selezione del task per l'installazione di un'APK



L'output così filtrato verrà mostrato all'utente, come è possibile osservare in Figura 4.5, così potrà selezionare ed eseguire un task specifico. La scelta verrà

intercettata ed utilizzata come parametro del comando successivo: *gradlew* ***task***.
Dunque, a seguito di questo comando, in base alla scelta dell'utente, una tra le APK individuate verrà installata sul dispositivo.
Eseguita con successo l'installazione, il passaggio successivo è l'esecuzione vera e propria del singolo test o dell'intera classe.
Il primo passaggio è individuare tutti gli instrumentation disponibili sul dispositivo. Il framework di testing Espresso è basato sugli instrumentation che definiscono il tipo di runner da utilizzare per eseguire i test. Per individuarli, viene eseguito un comando tramite shell adb. In particolare:

adb shell pm list instrumentation

I risultati vengono formattati e salvati sotto forma di stringhe che successivamente serviranno come parametri per le operazioni successive, che verranno eseguite per ogni instrumentation e si concluderanno con il lancio finale dei test.
Le seguenti operazioni eseguite prima dell'effettiva verifica della classe di test vengono effettuate nel caso in cui sia stata scelta la traduzione e generazione degli script di terza generazione.
La prima è la rimozione di tutti i dati che possano essere stati esportati nella memoria del dispositivo in una precedente traduzione eseguita dall'Executor. L'eliminazione di queste informazioni è necessaria perché il processo di generazione degli script di terza generazione utilizza i dump effettuati durante il lancio della classe di test in corso, dunque è importante che non ci siano dati relativi a traduzioni precedenti. Il comando viene eseguito tramite adb. In particolare la shell serve ad accedere all'interno del sistema nel dispositivo emulato, ad esempio, in questo caso, vengono modificate le informazioni all'interno dell'SD card:

*adb shell rm sdcard/****.bmp*** *sdcard/****.xml***

Attraverso questo comando verranno eliminati tutti i file presenti con estensione BMP ed XML.
Lo step successivo riguarda ancora una volta l'eliminazione di informazioni riguardanti la traduzione precedentemente avvenuta: in questo caso però, ad essere ripulito sarà il log. Questa operazione è necessaria perché attraverso il log vengono registrate le interazioni avvenute durante i test con la GUI dell'applicazione sul dispositivo tramite l'utilizzo di parole chiave che serviranno successivamente da filtro. Dunque, è necessario che le vecchie informazioni vengano eliminate e ci siano solo quelle relative all'esecuzione della classe di test di cui si vuole fare la traduzione. Attraverso un Process Builder viene eseguito il comando *adb logcat -c*; "**-c**" è l'opzione utilizzata per effettuare il clear del log in questione.
A seguire, bisogna garantire che l'applicazione sulla quale eseguire i test abbia i

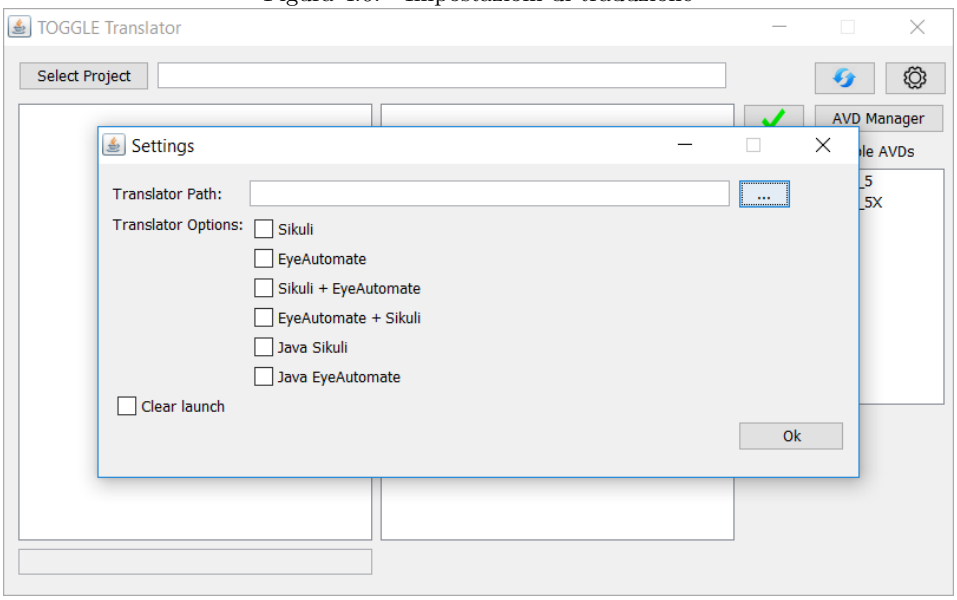
permessi per accedere in lettura e scrittura alla memoria del dispositivo. Questo passaggio è necessario perché la scelta di garantire determinati permessi va a discrezione dell'utente finale dell'applicazione o allo sviluppatore stesso in fase di creazione. Poiché durante la traduzione è fondamentale il salvataggio di alcuni file nella memoria del dispositivo, che verranno poi utilizzati per gli script generati, saranno necessari i permessi di scrittura della memoria, in quanto, in caso contrario, si presenteranno degli errori in fase di esecuzione. I comandi sono due, uno per i permessi di lettura ed uno per quelli di scrittura, e vengono eseguiti tramite shell adb, poiché bisogna interagire con il sistema stesso:

adb shell pm grant ***targetPackage*** *Android.permission.READ_EXTERNAL_STORAGE*

adb shell pm grant ***targetPackage*** *Android.permission.WRITE_EXTERNAL_STORAGE*

Come si può notare, è necessario specificare il package (**targetPackage**) per identificare a quale applicazione garantire tali permessi. Questa informazione viene estrapolata, tramite apposite manipolazioni testuali, dalla stringa che rappresenta l'instrumentation di cui abbiamo discusso in precedenza.

Figura 4.6. Impostazioni di traduzione



L'ultima operazione prima dell'esecuzione dei test viene effettuata a seconda che venga selezionata o meno una tra le impostazioni presenti nel frame dei "Settings". Come viene mostrato in Figura 4.6, spuntando la casella "Clear launch", verrà attivata l'opzione per resettare l'applicazione prima di ogni test che dev'essere eseguito. Questa funzionalità può essere necessaria nel momento in cui viene eseguita un'intera suite di test, in maniera sequenziale, ma ogni test dovrà avere una condizione di partenza indipendente da quello precedente.
Il comando viene eseguito tramite adb shell:

adb shell pm clear ***targetPackage***

Come visto in precedenza, anche questo comando ha bisogno che sia specificato il package dell'applicazione di cui bisognerà effettuare il clear. Il comando viene eseguito tramite adb shell. Il metodo che lancia il Process Builder accetta come parametro, tra tutti quanti, il nome del test specifico da lanciare: se questo parametro ha un valore effettivo, il comando eseguito sarà differente. In particolare, i due comandi sono:

adb shell am instrument -w -e class ***testPackage.testName instrumentation***
adb shell am instrument -w -e class ***testPackage.testName#subTest instrumentation***

Come si può notare, si differenziano dal parametro "subTest", che specifica se il comando eseguirà tutta la suite di test o uno specifico test della classe. Inoltre, "instrumentation" rappresenta, appunto, l'instrumentation usata per eseguire il test all'interno del dispositivo.

4.2.5 Enhance

L'operazione di enhance di una suite di test consiste nell'aggiunta di metodi della classe ToggleTools. Questa classe contiene delle funzioni apposite che, aggiunte ai metodi della suite di test, permetteranno di eseguire la traduzione da seconda a terza generazione.
Il processo di enhance inizia cercando all'interno del progetto Android attualmente caricato dall'Executor la classe ToggleTools. La ricerca avviene in maniera ricorsiva, partendo dal path del progetto. Se la classe non dovesse essere già presente all'interno del progetto, si procede con la copia di essa nel package nel quale è contenuto il test su cui è in atto l'enhance. Questa operazione viene effettuata tramite creazione di un nuovo file nel path di destinazione tramite un **FileWriter**; invece, tramite un **BufferedReader** viene letta la classe ToggleTools presente tra le risorse dell'Executor e ne verrà copiato il contenuto nel nuovo file creato. Così facendo, sarà possibile fare riferimento direttamente a quella classe, generando una nuova suite con i test originali più i metodi di ToggleTools.
L'enhance vero e proprio viene effettuato dall'Enhancer, componente dell'architettura che si occupa di questa operazione, che viene richiamato tramite un metodo di una libreria che contiene le funzioni apposite e la classe principale atte allo scopo.

4.2.6 Traduzione e creazione degli script di 3° generazione

La traduzione di una suite di test viene effettuata selezionando una classe tra le presenti nell'apposita sezione della GUI principale dopo aver caricato il progetto Android. Una volta premuto il pulsante per eseguire la traduzione, viene controllato se è stato già effettuato l'enhance della suite selezionata. Questo step è fondamentale, poiché è possibile effettuare la traduzione solo per le classi alle quali sono stati aggiunti i metodi di ToggleTools, i quali permettono la creazione degli script di terza generazione.
Il processo di traduzione inizia eseguendo l'intera suite di test enhanced sul dispositivo emulato. L'esecuzione della suite avviene attraverso i passaggi precedentemente descritti nel paragrafo **4.2.4**.
In questo paragrafo verranno descritti gli step che, aggiunti alla normale esecuzione della suite, permettono la traduzione delle classi di test.
Per questa operazione è stata definita una serie di prerequisiti e di strutture dati: in particolare, poiché i test Espresso vengono tradotti in test di terza generazione che si basano sulla GUI dell'applicazione, sono state effettuate alcune considerazioni riguardo la dimensione e la risoluzione del dispositivo emulato sul quale vengono eseguiti i test, unite alla risoluzione della macchina sulla quale l'Executor viene utilizzato. Attraverso un processo empirico, sono stati raccolti una serie di rapporti altezza-larghezza che corrispondono al numero di pixel occupato dall'emulatore sullo schermo al variare della sua risoluzione. A seguito di queste considerazioni, è sorta la necessità di utilizzare una struttura dati che potesse contenere, e dalla quale fosse possibile estrapolare, le informazioni riguardanti tutti i tipi di AVD supportati. Per tali ragioni, è stato scelto di utilizzare un file JSON, in modo tale da poter costruire una mappa a più livelli. In particolare, il primo livello corrisponde al nome del dispositivo; per ognuno dei nomi di tutti gli emulatori supportati è presente una coppia che rappresenta, rispettivamente, la risoluzione del monitor della macchina sulla quale vengono eseguiti i test e la dimensione in pixel del dispositivo selezionato. Dunque, prima di tutto viene estrapolato il nome specifico dell'AVD connesso dal suo file di configurazione; successivamente, un comando di sistema restituirà il valore che rappresenta la risoluzione del monitor (ad esempio "1920x1080"). Attraverso il nome del dispositivo, dal file JSON sarà possibile estrapolare la mappa dalla quale ricavare, tramite la chiave che corrisponde alla risoluzione del monitor, i dati dell'emulatore che saranno necessari durante l'operazione di traduzione.
In aggiunta ai dati precedentemente ottenuti, sarà necessario ottenere la risoluzione dell'emulatore connesso (ad esempio "800x480"). A tal proposito, viene eseguito con un ProcessBuilder il seguente comando:

adb shell wm size

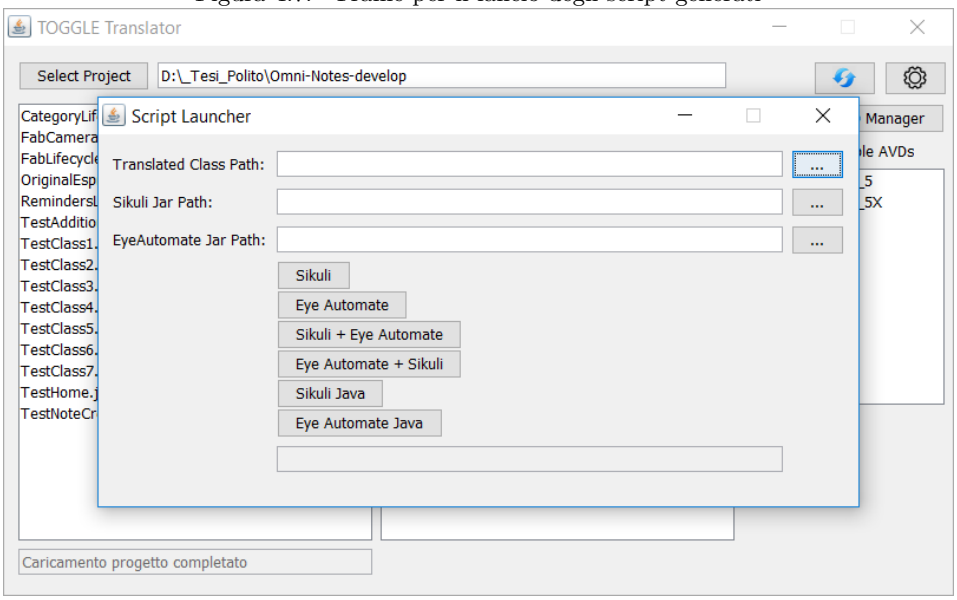
Per creare gli script di terza generazione, verrà istanziata una classe specifica all'interno di ToggleTools, inizializzandola con parametri specifici. Inizialmente, verranno impostate due stringhe necessarie ai fini della generazione: il percorso dal quale eseguire il comando adb ed il nome della MainActivity dell'applicazione sulla quale vengono eseguiti i test. La prima si trova già come informazione all'interno dell'Executor, mentre la seconda viene estrapolata dal Manifest all'interno del progetto: in particolare, questo file viene aperto all'interno del programma e scomposto nelle sue componenti principali come nodi, e su di

essi viene fatta una particolare ricerca con l'utilizzo di parole chiave. Dopodiché la classe viene istanziata, usando le informazioni fin qui raccolte come parametri, ed un metodo apposito per la creazione degli script viene eseguito.
In aggiunta sono presenti anche le opzioni di traduzione, elementi fondamentali senza i quali il processo non può iniziare, che sono selezionate attraverso l'apposito frame, come mostrato in Figura 4.6. A seconda della combinazione selezionata, verranno generati script eseguibili dai tool EyeAutomate e/o Sikuli. Nello specifico, le prime due selezioni creano una cartella nel path specificato come percorso di traduzione, nella quale verrà effettuato il dump delle immagini, che rappresentano i widget specifici della GUI dell'applicazione con la quale il test appena eseguito ha interagito. Verrà creato un folder per ogni test della suite selezionata; all'interno di essa si crea un file con estensione eseguibile dal tool indicato e si copiano, dalla cartella principale, le immagini necessarie ad eseguire il test visuale sull'applicazione.
Le altre opzioni creano una classe Java che contiene il codice generato per uno solo o entrambi i tool. Questa scelta è stata effettuata poiché sia EyeAutomate sia Sikuli presentano un API Java per eseguire i test. Nello specifico, è possibile generare quattro script di due tipi diversi: le prime due contengono rispettivamente il test eseguibile con EyeAutomate e con Sikuli; le altre due contengono entrambe le tipologie, combinate in modo tale che se una fallisca venga eseguita l'altra, e viceversa. Se viene selezionata una di queste quattro opzioni, verrà creata una cartella nel path selezionato per la traduzione che conterrà il progetto Java, che sarà successivamente possibile eseguire tramite Executor stesso. In questo folder, dunque, verranno create le apposite cartelle nelle quali verranno inseriti i sorgenti delle classi di test generati e le librerie di EyeAutomate o Sikuli, a seconda della selezione. Verranno inoltre copiate le immagini scaricate dal dispositivo ottenute dal path scelto per la traduzione, che serviranno per eseguire i test visuali sull'AVD.
Una volta che la generazione è stata completata, vengono eliminate tutte le immagini scaricate dal dispositivo nella cartella principale.

4.2.7 Lancio degli script generati

Dopo la generazione degli script, l'Executor mette a disposizione una modalità di lancio, direttamente tramite interfaccia grafica. Come è possibile vedere in Figura 4.7, è presente un apposito frame attraverso cui è possibile scegliere che tipo di test visuale lanciare sul dispositivo connesso. Nel dettaglio, sono presenti tre aree di testo che rappresentano tre diversi path che l'utente dovrà selezionare. La prima area specifica il percorso nel quale sono stati creati gli script; la seconda e la terza sono, rispettivamente, i percorsi nei quali si trovano le librerie di Sikuli ed EyeAutomate attraverso cui gli script verranno eseguiti. Successivamente, sono presenti una serie di pulsanti che specificano il tipo di script da lanciare. Considerato che l'utente può scegliere liberamente che tipo di traduzione effettuare, non è detto che siano presenti gli script di tutti i tipi: dunque, prima del lancio verrà controllato che nel path indicato ci sia la classe di test del tool scelto dall'utente; in caso contrario, l'esecuzione sarà annullata e ne verrà segnalata l'assenza.
Per il lancio degli script nativi di Sikuli ed EyeAutomate, le operazioni sono molto simili.

Figura 4.7. Frame per il lancio degli script generati



Prima di tutto, verrà chiamato un metodo che esegue il clear dell'applicazione ed il lancio dell'Activity principale. Questo passaggio è necessario poiché i test visuali possono essere eseguiti solamente se il dispositivo è riconoscibile sul monitor della macchina e l'applicazione è avviata nella schermata principale. Questo step viene attuato tramite l'esecuzione di due comandi con un Process Builder:

adb shell pm clear ***package_name***
adb shell am start -n ***package_name/main_activity_name***

Il primo comando effettua il clear dell'applicazione; il secondo serve a lanciare l'activity principale. Come si può notare, entrambi necessitano del package come parametro; in più, per il lancio della main activity è necessario specificarne il nome.
Per eseguire gli script, un Process Builder lancia un comando specifico a seconda del tool da utilizzare. In particolare:

Sikuli:
java -jar sikulix.jar -r ***starting_folder\test_name****sikuli_script.sikuli*

EyeAutomate:
java -jar EyeAutomate.jar ***starting_folder\test_name\eyeautomate_script****eyescript.txt*

Entrambi sono comandi "java" eseguiti con l'opzione "**-jar**", poiché le classi vengono eseguite tramite la libreria del tool selezionato, che va specificata; per entrambi i comandi, viene specificato il path nel quale trovare lo script da eseguire. Il Process Builder utilizza come cartella di partenza quella della libreria specificata tramite apposito file chooser.
Prima dell'esecuzione dei test viene aperto un file con estensione CSV nel quale saranno salvati i risultati dei test in un formato preciso. In particolare, per ogni test eseguito la linea scritta conterrà:

script_type - date(yyyy:MM:dd:HH:mm:ss.SSS) - package_name ; class_name ; test_name ; iteration ; test_result ; execution_time

Per i test di Sikuli, il risultato viene estrapolato direttamente dall'output del terminale; per i test EyeAutomate, invece, è necessario aprire il file di log, in formato TXT, e da lì ricavare l'esito.
Per gli script che utilizzano l'API di Java il processo si basa sul build e sull'esecuzione di un progetto da terminale. Per prima cosa, viene lanciato il comando:

javac -d ..\bin -classpath libraryPath1;libraryPath2;[...] ***classname***

per eseguire il build del progetto Java nel quale è contenuto lo script generato. Successivamente, verrà lanciato il comando:

java -Dfile.encoding=Cp1252 -classpath libraryPath1;libraryPath2;[...] ***className***

che eseguirà il "main"c della classe principale.
Anche in questo caso verranno eseguiti clear e lancio della Main Activity, così come verranno salvati i risultati all'interno di un file CSV. A differenza della situazione precedente, tutto ciò è gestito direttamente dallo script stesso.

Capitolo 5

Validazione

In questo capitolo verrà trattato il processo di validazione effettuato per dimostrare l'efficacia della traduzione delle suite di test di seconda generazione in script di terza generazione

5.1 Design del processo di validazione

L'obiettivo dell'esperimento descritto in questo capitolo è quello di validare l'efficacia e la precisione dei test di terza generazione tradotti dalle classi di test Espresso tramite Toggle. Questa validazione è importante poiché i tool di testing di terza generazione, specialmente quando vengono eseguiti su AVD, presentano errori riguardo il riconoscimento delle immagini.
Per eseguire la validazione, è stato applicato Toggle a due suite di test sviluppate per due diverse applicazioni open-source Android: Omni-Notes, un'applicazione che gestisce note testuali, e Pass-Android, un'applicazione per la gestione di diversi tipi di biglietti elettronici. Queste due applicazioni sono state scelte date le loro differenze per quanto riguarda l'aspetto, lo scopo e le caratteristiche, per ottenere dunque una generalità più ampia dei risultati.
Le caratteristiche principali delle due applicazioni sono riportate nella Tabella 5.1.
Entrambe le suite sono composte da 30 test diversi. Tutti i test sono stati progettati per essere indipendenti tra di loro ovvero vengono tutti eseguiti a partire dallo stato originale dell'applicazione, per assicurarsi che il fallimento di un test non generi una serie di fallimenti a cascata sui test successivi.
I test sono stati creati in modo tale da essere eseguiti su quasi tutte le schermate presenti nell'applicazione, con un numero di interazioni che va da 4 a 18, inclusi i check.
Ogni test è stato eseguito dieci volte. Lo sviluppo e l'esecuzione delle suite di test Espresso sono stati effettuati su Android Studio 3.3; le applicazioni sono state lanciate su un emulatore Nexus 5X API 25. Gli script di terza generazione generati per EyeAutomate e Sikuli sono stati inseriti all'interno di codice Java, ed eseguiti tramite appositi script runners forniti dalle rispettive API.

Tabella 5.1. Caratteristiche delle applicazioni scelte (Febbraio 2019)

	Omni-Notes	PassAndroid
Number of downloads	100.000+	1.000.000+
Number of Releases	118	100
Tested release	6.0.0 beta 7	2.5.0
Java LOCs	48,116	32,309
Number of Activities	13	17
Number of Layout Files	52	19

5.1.1 Research questions

Lo scopo della validazione sperimentale è quello di rispondere alle seguenti domande:

RQ1 Qual è il tasso di successo dei test visuali generati attraverso la traduzione? Quali sono le differenze di affidabilità tra le sei diverse combinazioni di test visuali?

RQ2 Qual è la prestazione dei test visuali? Quali sono le differenze di prestazione tra le sei combinazioni di test visuali e gli script originali di seconda generazione?

RQ3 Quanto è efficace in relazione alla device diversity? Quanto è portabile nel momento in cui viene eseguito su vari dispositivi?

Per rispondere a RQ1, per ciascun test è stato calcolato il Success Rate (SR), definito come:

$$SR_t = N_s/N_{ex}, \tag{5.1}$$

dove N_s è il numero di esecuzioni terminate con successo e N_{ex} è il numero totale di esecuzioni (in questo caso equivale a 10). Questa equazione dà la percentuale di esecuzioni terminate con successo per ciascun test.
Inoltre, utilizzando il numero di esecuzioni terminate positivamente, i test sono stati classificati come *Failing*, *Passing* or *Flaky*. Un test è considerato *Passing* quando tutte e 10 le esecuzioni terminano con successo (dunque, $SR_t = 1$). Un test è considerato *Failing* quando tutte le sue esecuzioni falliscono ($SR_t = 0$). Si assume che quando tutte le esecuzioni di un test terminano negativamente, il motivo è la limitazione propria dei tool per i test di terza generazione, che presentano alcuni problemi a riconoscere alcuni componenti o ad interagire con l'AVD.
Un test è considerato *Flaky* quando alcuni dei suoi test falliscono ($0 < SR_t < 1$), ma non tutti. In dettaglio, questo caso si verifica quando abbiamo un comportamento non predicibile, causato, ad esempio, da un'esecuzione terminata positivamente, seguita da una fallita e, infine, un'altra terminata con successo. Si assume che questa situazione è causata dalla scarsa precisione nell'applicare gli algoritmi di riconoscimento delle immagini o nel ricreare le interazioni dell'utente.

Per rispondere a RQ2, è stato misurato il tempo medio di esecuzione (T_x) di tutti i

test eseguiti con successo. Poiché i diversi casi di test presentano una complessità variabile, il tempo di esecuzione misurato è stato normalizzato per il numero di interazioni contenute in ciascun test. Il tempo di esecuzione misurato dipende dalle istruzioni di "sleep" che sono state inserite per la traduzione delle interazioni e dai possibili errori del primo algoritmo di riconoscimento delle immagini utilizzato nei casi di test combinati di terza generazione. Le istruzioni di "sleep" sono state aggiunte per aumentare il rate di successo dei test, cercando di limitare i problemi di sincronizzazione.

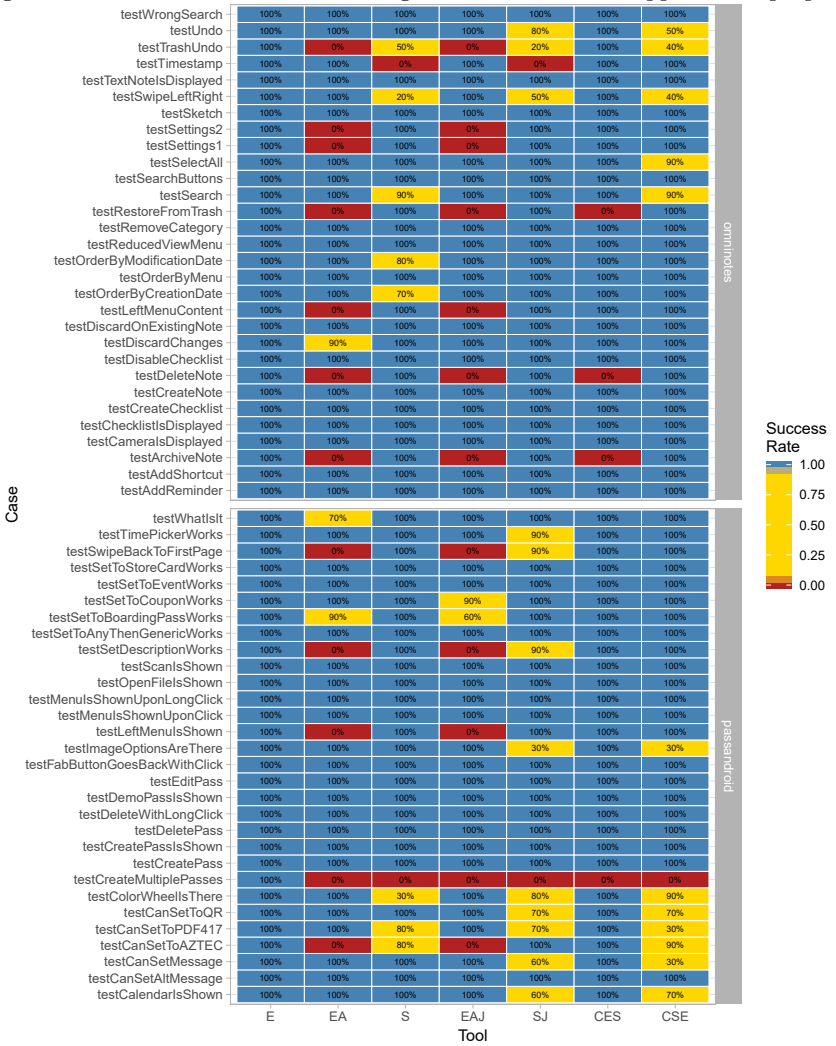
5.2 Risultati sperimentali

In questa sezione verranno riportati i valori misurati per rispondere alle Research Questions esposte precedentemente, assieme ad una valutazione dei tool di testing di terza generazione.

5.2.1 RQ1: Tasso di successo

In Figura 5.1 viene mostrato un riepilogo di tutti i test sviluppati per le due applicazioni, e il tasso di successo ottenuto da ciascun test generato.
È stato osservato che, in generale, c'è una differenza nel tasso di successo significativa tra i tool di terza generazione ($p < 10^{-15}$), ma non è stata riscontrata nessuna differenza sostanziale fra le due applicazioni ($p = 0.56$), mentre c'è un'interazione statistica notevole fra il tool e le applicazioni ($p < 10^{-15}$).
Per OmniNotes, la combinazione di Sikuli per primo ed EyeAutomate secondo (CSE) è risultata essere la migliore; mentre per PassAndroid lo è la combinazione inversa (CES).
In più, è possibile notare come EyeAutomate eseguito tramite Script Runner su entrambe le applicazioni tramite sintassi specifica in plain text, o codice Java tramite la sua API, è risultato il meno performante come numero di test eseguiti con successo. Tra le motivazioni c'è l'incapacità del tool di riconoscere alcuni componenti visuali, così come i problemi nel timing per le interazioni con la GUI delle applicazioni.
Le soluzioni che sono risultate più "flaky" sono quelle di Sikuli. Esso presenta comunque un numero di successi complessivo più alto, nonostante EyeAutomate utilizzi un algoritmo più deterministico. La presenza di un numero maggiore, in proporzione, di test flaky in Sikuli è dovuto principalmente alla presenza di test con operazioni di "swipe", eseguite in modo poco preciso dal tool. Questo problema non è stato presente nelle esecuzioni con EyeAutomate. Le operazioni di swipe sono state utilizzate in un numero cospicuo di test, sia per OmniNotes sia per PassAndroid.
Come si può notare, la soluzione con la maggior precisione è quella con EyeAutomate come principale strumento di riconoscimento per le immagini, appoggiato da Sikuli (CES). Inoltre, questa combinazione non presenta nessun test flaky e ciò potrebbe essere indice di un aumento della robustezza. L'assenza di questi test potrebbe essere spiegata dalla maggiore precisione della libreria di EyeAutomate nel ricreare le interazioni con la grafica, così come nel riconoscere i componenti visuali.
La combinazione dei tool migliora il tasso di successo nel caso di OmniNotes grazie al supporto di uno dei due tool nel caso l'altro fallisca; mentre nel caso di PassAndroid si nota un miglioramento solo nel test combinato CES.

Figura 5.1. Risultati dei test di terza generazione sulle due applicazioni proposte



Quest'ultimo, composto da un test combinato con Sikuli appoggiato da EyeAutomate ha ottenuto risultati migliori rispetto alle due versioni Java separate, (EAJ) e (SJ), ed il caso CSE ha presentato un tasso di successo inferiore. Questi risultati sono giustificati poiché l'API di Sikuli è meno robusta quando sono implicate operazioni di swipe: quando quest'ultima non è eseguita correttamente, il test continua comunque nella sua esecuzione e procede con le successive interazioni e check, che, naturalmente, falliranno. Dall'altra

parte, quando un test EyeAutomate si blocca durante l'individuazione di un componente grafico, utilizzando l'algoritmo di riconoscimento delle immagini di Sikuli è possibile completare il test.

Risposta a RQ1: Nessuno degli script di terza generazione ha ottenuto gli stessi risultati di Espresso. È stato appurato che la migliore soluzione è quella di usare EyeAutomate appoggiato da Sikuli (CES), ma anche che EyeAutomate, in generale, ha ottenuto risultati peggiori di Sikuli.
In più, Sikuli è la soluzione più adatta per quanto riguarda l'utilizzo dell'algoritmo di riconoscimento delle immagini.
Concludendo, se entrambi EyeAutomate e Sikuli sono disponibili, sarebbe opportuno utilizzarli entrambi combinati tra loro.

I risultati di questa valutazione dimostrano come, combinando gli algoritmi, si possano migliorare i risultati e come l'utente possa far leva sui benefici e gli svantaggi. Inoltre, tramite un'ulteriore combinazione con altri tool, o aggiungendo determinismo alle combinazioni, ad esempio utilizzando strumenti specifici per determinate interazioni o immagini, si possono ottenere risultati migliori.

5.2.2 RQ2: Tempo di esecuzione

Il tempo medio di esecuzione è stato calcolato considerando solo i test eseguiti con successo, normalizzati per il numero di interazioni eseguite. I check (sia su un componente grafico specifico sia sulla totalità dello schermo) sono stati considerati come interazioni, dato che l'algoritmo di riconoscimento delle immagini richiede un tempo per identificare un componente, e, in più, sono state aggiunte istruzioni di sleep alla fine di ogni check.
Espresso è stato preso in esempio come soglia di riferimento per gli altri tool. Il numero di interazioni effettuate dai test Espresso è lo stesso degli script tradotti di terza generazione, fatta eccezione per i check finali che non sono presenti nei test originali. Il risultato di questa analisi ha mostrato una differenza significativa tra i tool di seconda e terza generazione ($p < 10^{-15}$).
Espresso ha dimostrato un tempo di esecuzione medio per interazione migliore. La ragione principale può essere spiegata considerando l'utilizzo da parte di questo tool di proprietà che presentano intrinsecamente prestazioni più elevate, dovute al ridotto numero di calcoli da eseguire rispetto all'approccio basato sul riconoscimento delle immagini. In aggiunta, Espresso, essendo integrato nel framework di Android ed essendo capace di filtrare gli intent per il passaggio da un'Activity ad un'altra, è in grado di attendere esattamente il tempo che occorre ad un Activity o ad un componente grafico per apparire sullo schermo, minimizzando i tempi di attesa.
L'utilizzo di test di varianza appositi applicati alla distribuzione del tempo per interazione misurato per le sei differenti tecniche di testing di terza generazioni per entrambe le applicazioni, ha mostrato una differenza significativa del tempo medio per interazione a seconda del tool ($p < 10^{-15}$), nessun effetto evidente sull'applicazione ($p = 0.1$), e, infine, un effetto significativo dell'interazione dei due fattori (?) ($p < 10^{-15}$).
La differenza del tempo medio per interazione dovuta al tipo di applicazione sul quale

sono stati effettuati i test può essere spiegata considerando i pattern diversi delle interazioni delle due applicazioni, esponendo come esempio PassAndroid, che presenta un numero più alto di operazioni (più lunghe) di swipe.
Analisi ulteriori dimostrano che il secondo miglior tool è la versione Java di Sikuli, seguito dall'API Java di EyeAutomate.
Entrambi i tool presentano un'efficienza notevolmente minore quando eseguiti tramite sintassi nativa. Una spiegazione plausibile potrebbe derivare dalla considerazione che i casi di test sono stati eseguiti all'interno dell'ambiente Java, istanziando gli script runner propri delle rispettive librerie.
I test combinati presentano prestazioni leggermente inferiori rispetto al resto dei tool eseguiti individualmente. La ragione sta nell'approccio di Toggle alla generazione dei test, i quali vengono eseguiti sempre con uno dei due tool prima, e, solo dopo essere fallito, dopo un determinato periodo di tempo viene utilizzato il secondo. Il tempo di attesa a seguito della transizione da un tool all'altro aggiunge un overhead notevole ad ogni script.
Malgrado il tempo di attesa del fallimento di un tool prima dell'esecuzione dell'altro, entrambe le soluzioni combinate presentano prestazioni migliori degli script sviluppati con la sintassi nativa del tool.

Risposta a RQ2: L'approccio di seconda generazione ha prestazioni notevolmente maggiori rispetto a qualsiasi altra soluzione.
C'è una differenza significativa tra il tempo medio per interazione calcolato per i sei tool di testing di terza generazione utilizzati, tra cui il più veloce risulta essere Sikuli in Java (SJ).

5.2.3 RQ3: Robustezza alla Device Diversity

Questa research question porta all'attenzione il problema della device diversity.
La device diversity consiste nella presenza di dispositivi con diverse caratteristiche che possono essere emulati e sui quali lanciare le suite Espresso e generare gli script di terza generazione. Questo avviene in quanto Android supporta diversi tipi di AVD, e la lista di quelli emulabili continua a crescere, comprendendo anche quelli meno recenti.
Nel caso dell'esecuzione dei test visuali tramite Sikuli o EyeAutomate, questo potrebbe risultare un problema, poiché ogni dispositivo possiede dimensioni diverse, sia in altezza che in larghezza, così come la densità di pixel. Gli script nativi di questi tool operano attraverso l'utilizzo di immagini che rappresentano i widget dell'applicazione sulla quale effettuare i test, i quali vengono riconosciuti tramite appositi algoritmi. Da qui ne deriva che nel caso in cui si cambiasse dispositivo, il test non verrebbe eseguito con successo in quanto l'immagine che rappresenta il componente con il quale interagire potrebbe avere dimensioni più grandi o più piccole, in base alla dimensione dell'AVD dalla quale è stata estratta.
La dimensione potrebbe variare anche a seconda della risoluzione dello schermo della macchina sulla quale l'emulatore viene lanciato.
L'Executor utilizza una mappa attraverso cui viene associata una dimensione a seconda della risoluzione dello schermo e del tipo di dispositivo. Questi parametri vengono utilizzati per parametrizzare la cattura dello schermo e l'estrapolazione esatta delle immagini

dei componenti che vengono utilizzate dagli script generati tramite la traduzione.
Lo scopo di questa research question è quello di dimostrare l'effettività dell'Executor nei confronti della Device Diversity. Per rispondere a questa domanda sono stati effettuati due tipi differenti di verifiche.
La prima fase di validazione consiste nella traduzione di tutti i test case eseguiti su un dispositivo in una specifica modalità. In particolare, è stato scelto di tradurre tutte le suite di test per il dispositivo Nexus 5X e generare i test tradotti per il tool Sikuli. Tale scelta è stata decisa poiché questa modalità di traduzione eseguita su questo determinato dispositivo presentava il tasso più alto di successo. Successivamente, tutti gli script generati sono stati eseguiti sul resto degli AVD messi a disposizione dall'Executor. I risultati di questa fase sono presentati nella Tabella 5.2.

Tabella 5.2. Risultati dei test tradotti in Sikuli per il dispositivo Nexus 5X eseguiti su tutti i dispositivi

Dispositivo	# test passati	Robustezza non tradotti
Nexus 5X	22	22/30
Nexus S	0	0/30
Nexus One	0	0/30
Nexus 4	0	0/30
Nexus 5	0	0/30
Nexus 6	3	3/30
Nexus 6P	18	18/30
Galaxy Nexus	1	1/30
Pixel	1	1/30
Pixel XL	17	17/30

Media	Percentuale
4,6/30	15%

I risultati dimostrano un numero di insuccessi elevato. Questo accade poiché gli script tradotti per il dispositivo Nexus 5X fanno riferimento ad un set di immagini che sono state appositamente catturate dallo schermo e ritagliate in base alle dimensioni specifiche dell'AVD. Poiché quest'ultime variano a seconda del dispositivo emulato, l'algoritmo di riconoscimento visuale non riesce ad identificare il componente rappresentato nell'immagine.
La seconda fase consiste nella traduzione e successiva generazione degli script di terza generazione per tutti i test case per ogni AVD disponibile. Per coerenza con la fase precedente, è stato scelto di tradurre ogni test e generare gli script per il tool Sikuli. Successivamente, i test tradotti sono stati eseguiti ciascuno per il rispettivo dispositivo. I risultati sono riportati nella Tabella 5.3.

I risultati mostrano un elevato tasso di successo rispetto alla fase precedente. Questo

Tabella 5.3. Risultati dei test tradotti in Sikuli per ciascun dispositivo ed eseguiti sul corrispondente

Dispositivo	# test tradotti	# test passati	# passati / # tradotti	# passati / # totali
Nexus 5X	30	22	22/30	22/30
Nexus S	23	19	19/23	19/30
Nexus One	22	15	15/22	15/30
Nexus 4	30	18	18/30	18/30
Nexus 5	30	19	19/30	19/30
Nexus 6	30	18	18/30	18/30
Nexus 6P	30	18	18/30	18/30
Galaxy Nexus	30	20	20/30	20/30
Pixel	30	21	21/30	21/30
Pixel XL	30	21	21/30	21/30

Media	Percentuale
18,9/30	63%

accade poiché i test vengono tradotti per un dispositivo specifico ed eseguiti sullo stesso. Dunque, le immagini catturate dallo schermo saranno dimensionate in base all'AVD sul quale verranno lanciati gli script e l'algoritmo di riconoscimento visuale sarà in grado di riconoscere i componenti con i quali avvengono le interazioni.
Come è possibile notare, per i dispositivi Nexus S e Nexus One il numero di test tradotti è inferiore rispetto agli altri dispositivi. La spiegazione di questa discrepanza sta nel fatto che la traduzione di alcuni test è stata interrotta nel momento dell'esecuzione delle suite Espresso sulle quali è stato effettuato l'enhance. Il problema è la differente conformazione dei due dispositivi rispetto agli altri: questo fa sì che alcuni componenti presenti di norma nella schermata principale e caratterizzati da uno specifico identificativo vengano posizionati in un'altra sezione. A seguito di questo, il test, progettato per eseguire determinate interazioni in specifiche aree della GUI dell'applicazione, fallirà nel momento in cui non riesce ad individuare il componente con il quale interagire.
Dopo aver effettuato un'analisi più in dettaglio dei risultati, ovvero di quali test in particolare venivano eseguiti con successo o meno, è sorto che la suite di test denominata "TestUndoOrder" presentava una percentuale di fallimenti totale per quasi tutti i dispositivi. Dunque, sono stati riportati i risultati nella Tabella 5.4.
Il numero totale di test è minore rispetto ai risultati precedenti, poiché sono stati eliminati quelli della suite "TestUndoOrder", i quali presentavano un elevato tasso di insuccessi. Dato che la percentuale di test superati si basa sul numero di test eseguiti correttamente e sul numero totale, il risultato finale è più elevato del caso precedente.
Gli esiti di queste fasi di validazione mostrano che gli script tradotti per uno specifico dispositivo ed eseguiti su tutti i dispositivi presentano un tasso di successo nettamente inferiore (15%) rispetto ai risultati degli script generati per ciascun dispositivo ed eseguiti su di esso (63%). Questo porta alla conclusione che il tool per la traduzione garantisce portabilità e robustezza alla device diversity, in quanto la traduzione si incarica di analizzare le dimensioni dell'AVD e del contesto nel quale viene eseguito, permettendo la

Tabella 5.4. Risultati dei test tradotti in Sikuli per ciascun dispositivo ed eseguiti sul corrispondente, esclusa la suite di test "TestUndoOrder"

Dispositivo	# test tradotti	# test passati	# passati / # tradotti	# passati / # totali
Nexus 5X	25	20	20/25	20/25
Nexus S	19	16	16/19	16/25
Nexus One	19	14	14/19	14/25
Nexus 4	30	18	18/25	18/25
Nexus 5	30	18	18/25	18/25
Nexus 6	30	18	18/25	18/25
Nexus 6P	30	18	18/25	18/25
Galaxy Nexus	30	20	20/25	20/25
Pixel	30	20	20/25	20/25
Pixel XL	30	18	18/25	18/25

Media	Percentuale
18/25	72%

creazione di script di terza generazione funzionanti per qualsiasi contesto.

Risposta a RQ3: La generazione di script per EyeAutomate e Sikuli tramite Toggle aumenta la portabilità e la robustezza alla device diversity. Il motivo sta nella capacità del tool di parametrizzare la traduzione e la generazione di tali script con le dimensioni e le risoluzioni sia del dispositivo emulato sia del display sul quale viene eseguito. Il risultato è stato dimostrato dal netto miglioramento dei risultati di esecuzione degli script generati ad hoc per ciascun dispositivo sul dispositivo stesso a confronto con i risultati degli script generati per un solo dispositivo ed eseguiti su tutti gli altri.

Capitolo 6

Conclusione

In questo elaborato è stato presentato il componente dell'architettura del traduttore di script di seconda generazione in script di terza generazione denominato Executor. L'Executor è stato sviluppato come tool software nel quale sono state unite le funzionalità degli altri componenti architetturali; l'interfaccia grafica sviluppata tramite Java Swing permette all'utente di interagire in maniera intuitiva ed eseguire tutte le operazioni disponibili.
Lo scopo di questo progetto è quello di riuscire a creare un meccanismo robusto ed efficace per ovviare alle limitazioni presenti in entrambe le tipologie di GUI testing. Tramite questo strumento è possibile eseguire i test creati tramite framework Espresso su un dispositivo emulato, AVD, il quale potrebbe essere già presente sulla macchina o può essere creato attraverso l'Executor stesso. Successivamente sarà possibile effettuare l'Enhance degli script di seconda generazione e, dunque, aggiungere gli strumenti necessari alla traduzione. Eseguendo i test enhanced, verranno individuate tutte le interazioni che serviranno alla creazione degli script di terza generazione. Una volta creati, gli script potranno essere eseguiti tramite interfaccia messa a disposizione dell'Executor.
La validazione effettuata tramite test sperimentali ha dimostrato l'efficacia del tool e la sua robustezza alla Device Diversity. Infatti, le percentuali di successo dei test eseguiti su diversi dispositivi hanno dimostrato come il processo di traduzione permetta di generare script destinati all'esecuzione sul dispositivo specifico, grazie all'analisi sulle dimensioni e sulle risoluzioni. Ciò permette, dunque, di ridurre drasticamente i costi in termini di tempo e risorse nel momento in cui l'applicazione soggetta a test subisce modifiche sostanziali in termini di struttura e layout per le quali sarà necessario modificare di conseguenza i test case.
Tramite ulteriori ottimizzazioni delle fasi di build e installazione dell'applicazione sul dispositivo emulato, si potranno migliorare i tempi di esecuzione dei test. Nell'ambito della traduzione verranno successivamente comprese ulteriori funzionalità di Espresso per rendere più funzionale e completa l'esperienza di testing visuale.
Il contesto di sviluppo dell'Executor è incentrato sulla traduzione da script di seconda a script di terza generazione: dunque, in futuro verrà implementato il processo inverso,

così da poter ovviare alle limitazioni che presentano gli script di GUI testing basati su algoritmi di riconoscimento delle immagini.
Completati gli opportuni miglioramenti, sarà eseguita un'ulteriore fase di validazione del tool più significativa su un campione più grande di applicativi, comprendendo un insieme di test già esistenti che non siano stati creati appositamente per la validazione. Tramite questo processo sarà possibile verificare l'applicabilità del tool a contesti reali.

Bibliografia

[1] Gianluigi Caldiera, Victor R. Basili, and H. Dieter Rombach. *The goal question metric approach.* Encyclopedia of software engineering (1994): 528-532

[2] Martin Kropp and Pamela Morales. *Automated GUI testing on the Android platform.* Testing Software and Systems: Short Papers (2010), 67.

[3] Muccini, Henry, Antonio Di Francesco, and Patrizio Esposito. *Software testing of mobile applications: Challenges and future research directions.* Automation of Software Test (AST), 2012 7th International Workshop on. IEEE, 2012.

[4] Kirubakaran, B., and V. Karthikeyani. *Mobile application testing Challenges and solution approach through automation.* Pattern Recognition, Informatics and Mobile Engineering (PRIME), 2013 International Conference on. IEEE, 2013

[5] Amalfitano, Domenico, et al. *Testing Android Mobile Applications: Challenges, Strategies, and Approaches.* Advances in Computers 89.6 (2013): 1-52.

[6] Andrea Stocco, Maurizio Leotta, Filippo Ricca, Paolo Tonella. *PESTO: A Tool for Migrating DOM-based to Visual Web Tests.* 2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation

[7] Emil Alégroth, Zebao Gao, Rafael A.P. Oliveira, Atif Memon. *Conceptualization and Evaluation of Component-based Testing Unified with Visual GUI Testing: an Empirical Study.* 2015

[8] Samer Zein, Norsaremah Salleh, John Grundy. *A systematic mapping study of mobile application testing techniques.* The Journal of Systems and Software 117 (2016): 334-356

[9] Riccardo Coppola. *Fragility and Evolution of Android Test Suites.* 2017 IEEE/ACM 39th IEEE International Conference on Software Engineering Companion

[10] Mario Linares-Vásquez, Kevin Moran, Denys Poshyvanyk. *Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing.* 2017 IEEE International Conference on Software Maintenance and Evolution

[11] Porfirio Tramontana, Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino. *Automated functional testing of mobile applications: a systematic mapping study.* 24 October 2018

[12] Riccardo Coppola, Maurizio Morisio, Marco Torchiano. *Mobile GUI Testing Fragility: A Study on Open-Source Android Applications.* IEEE TRANSACTIONS ON RELIABILITY, VOL. 68, NO. 1, MARCH 2019

[13] Luca Ardito, Riccardo Coppola, Maurizio Morisio, Marco Torchiano. *Espresso vs. EyeAutomate: An Experiment for the Comparison of Two Generations of Android GUI Testing.* 2019

[14] Tom Yeh, Tsung-Hsiang Chang, Robert C. Miller. *Sikuli: Using GUI Screenshots for Search and Automation.*

[15] https://developer.android.com/training/testing/espresso

[16] https://sikulix-2014.readthedocs.io/en/latest/scenarios.html

[17] https://eyeautomate.com/wp-content/themes/EyeAutomateTheme/resources/EyeAutomateManual.html#

[18] https://developer.android.com/studio/command-line/adb

[19] https://www.idc.com/promo/smartphone-market-share/os

[20] https://www.macworld.co.uk/feature/iphone/iphone-vs-android-market-share-3691861/